CSCI E-103: Data Engineering for Analytics
Lecture 01: Introduction to Data Engineering

Harvard Extension School

Fall 2024

■ **Course:**	CSCI E-103: Data Engineering for Analytics
■ **Lecture:**	Lecture 01: Introduction
■ **Instructor:**	Anindita Mahapatra & Eric Gieseke
■ **Objective:**	Understand data engineering fundamentals, the big data ecosystem, and begin hands-on practice with Databricks

Key Summary

This lecture introduces the field of data engineering—the discipline of transforming raw data into valuable insights. We'll explore the 5 V's of Big Data (Volume, Velocity, Variety, Veracity, Value), compare OLTP vs OLAP systems, understand ACID vs BASE properties, trace the evolution from data warehouses through Hadoop to modern Spark-based platforms, and get hands-on with Databricks Free Edition for our first practical exercises.

Contents

1 Why Data Engineering Matters

1.1 The Perfect Storm: Big Data + Cloud + AI

We are living through an unprecedented convergence of three major technological forces:

- **Big Data**: Organizations now collect more data than ever before—terabytes and petabytes of information from every digital interaction
- **Cloud Computing**: On-demand compute and storage resources have democratized access to powerful infrastructure
- **Artificial Intelligence**: Machine learning models can extract patterns and make predictions, but only if fed properly prepared data

Key Information

The Analogy: Think of data as the ”new oil” of the digital economy. Just as crude oil requires refineries to become useful gasoline, raw data requires data engineering pipelines to become actionable insights. The data engineer is the modern-day refinery operator.

1.2 What is Data Engineering?

Definition:

Data Engineering Data Engineering is the practice of **turning raw data into valuable insights**. It encompasses the design, construction, and maintenance of systems and infrastructure that enable the collection, storage, transformation, and delivery of data at scale.

The key phrase businesses care about is **”Speed to Insights”**—the time from when data is generated to when decision-makers can act on meaningful conclusions. Data engineers are responsible for minimizing this time.

1.3 The Data Engineering Venn Diagram

Data Engineering sits at the intersection of three disciplines:

1. **Software Engineering**: System design, distributed computing, DevOps, service deployment
2. **Data Science**: Understanding of ML algorithms, statistical modeling, business intelligence
3. **Database/Infrastructure**: Data structures, ETL pipelines, storage systems, query optimization

Example:

The Carbon to Diamond Analogy Data refinement is like transforming carbon into diamond:
1. **Raw Data**: A messy pile of carbon atoms (e.g., application log files)
2. **Sorted Data**: Carbon arranged by some criteria (e.g., logs sorted by date and user)
3. **Cleaned Data**: Impurities removed (e.g., error logs filtered, user IDs mapped)
4. **Visualized Data**: Presented in digestible form (e.g., hourly access charts)

5. **Story/Insight**: Business meaning extracted (e.g., ”3 AM spike in errors correlates with specific region”)

Each transformation adds value, just as pressure and time transform carbon into diamond.

2 Data Personas: Who Works with Data?

In any data-driven organization, multiple roles collaborate to extract value from data. Understanding these personas helps clarify where data engineering fits in the bigger picture.

2.1 The Five Key Data Personas

Table 1: *Data Personas and Their Responsibilities*

Role	Primary Responsibilities
Data Engineer	Builds and maintains data pipelines. Responsible for ETL (Extract, Transform, Load), data quality, and infrastructure. Creates the "plumbing" that enables all other data work.
BI Analyst	Creates dashboards and reports using SQL. Takes refined data and presents it in consumable formats for business stakeholders.
Data Scientist	Performs exploratory data analysis (EDA) and builds machine learning models. Uses statistical methods to uncover patterns and make predictions.
MLOps Engineer	Handles automation of ML pipelines. Ensures models are reliably deployed, monitored, and maintained in production.
Data Leader	Strategic role (CDO, CIO). Sets data governance policies and drives organizational data strategy.

2.2 The Hidden Complexity of ML Systems

Warning

The Iceberg Illusion: When people think of AI/ML, they often focus only on the model code—the algorithm that makes predictions. But in reality, **90% or more of a production ML system is data engineering work**.

The model is just the tip of the iceberg. Below the surface lies:

- Data collection and ingestion
- Data validation and quality checks
- Feature extraction and engineering
- Resource management and compute allocation
- Serving infrastructure
- Monitoring and alerting
- Configuration management

All of these fall within the data engineer's domain.

3 The Five V's of Big Data

Big Data isn't just ”a lot of data”—it's characterized by five distinct dimensions known as the 5 V's.

3.1 Volume: The Size of Data

Definition:

Volume The physical amount of data being stored and processed, typically measured in terabytes (TB), petabytes (PB), or even exabytes (EB). When data is too large to fit on a single machine, you're dealing with Big Data's volume challenge.

Scale perspective:

- 1 Kilobyte (KB) = 1,000 bytes ≈ a short text message
- 1 Megabyte (MB) = 1,000 KB ≈ a high-resolution photo
- 1 Gigabyte (GB) = 1,000 MB ≈ an hour of HD video
- 1 Terabyte (TB) = 1,000 GB ≈ a library of 500 hours of movies
- 1 Petabyte (PB) = 1,000 TB ≈ all photos on Facebook (early 2010s)

3.2 Velocity: The Speed of Data

Definition:

Velocity How fast data is generated, collected, and processed. This ranges from batch processing (periodic bulk updates) to real-time streaming (continuous flow).

Three processing modes:

1. **Batch Processing**: Collect data over time, process in bulk at scheduled intervals (e.g., nightly reports)
2. **Micro-batch**: Small chunks processed at short intervals (e.g., every 5 minutes)
3. **Streaming/Real-time**: Data processed immediately as it arrives (e.g., fraud detection)

3.3 Variety: The Types of Data

Definition:

Variety The different forms and formats of data that must be handled. Modern systems must process structured, semi-structured, and unstructured data together.

Important:

The 80-90% Reality Approximately 80-90% of enterprise data today is **unstructured**. This includes images, audio, video, documents, and text—precisely the data that feeds modern AI models. Traditional SQL databases only handle the remaining 10-20%.

Table 2: *Data Structure Types*

Type	Characteristics	Examples
Structured	Fixed schema, rows/-columns	SQL databases, spreadsheets
Semi-structured	Schema embedded in data	JSON, XML, Parquet files
Unstructured	No predefined schema	Images, audio, video, text documents

3.4 Veracity: The Quality of Data

Definition:

Veracity The trustworthiness and accuracy of data. Can you believe what the data is telling you? This encompasses data quality, noise, bias, and completeness.

The principle **GIGO (Garbage In, Garbage Out)** applies: even the most sophisticated ML model will produce useless results if trained on poor-quality data.

3.5 Value: The Business Impact

Definition:

Value The ultimate worth of data to the business. Of all five V's, this is the most important—all the volume, velocity, variety, and veracity management is meaningless if the data doesn't create business value.

Key Information

The Value Question: Before investing in processing any dataset, always ask:

- What business question will this answer?
- Is the insight actionable?
- Does the potential value justify the processing cost?

4 Data Temperature: Hot vs Cold

Beyond the 5 V's, data engineers also think about data ”temperature”—how frequently data is accessed and how urgently it needs to be processed.

4.1 The Temperature Spectrum

Definition:

Data Temperature A classification of data based on how recently it was generated and how frequently it's accessed:

- **Hot Data**: Recent, frequently accessed, requires immediate processing
- **Warm Data**: Moderate access frequency, intermediate processing needs
- **Cold Data**: Historical, rarely accessed, can be processed in batch

Example:

Temperature Examples

- **Hot**: Current stock prices, real-time fraud detection signals, live website traffic
- **Warm**: Last month's sales data, recent user activity logs
- **Cold**: Historical archives, compliance records, data from years ago

4.2 Temperature and Business Value

Warning

The Staleness Problem: As data ”cools down,” its value typically decreases. A fraud detection signal is most valuable in the moment it's generated. A week later, the fraudulent transaction has already cleared.

This is why businesses increasingly demand real-time processing—but real-time comes at a cost. Always ask: ”Do we **really** need this in real-time? What action will we take with the insight?”

5 OLTP vs OLAP: Two Worlds of Data Processing

Understanding the distinction between transactional and analytical systems is fundamental to data engineering.

5.1 OLTP: Online Transactional Processing

Definition:

OLTP Online Transactional Processing systems handle real-time, day-to-day operations. They process individual transactions quickly—inserts, updates, deletes—and maintain the current state of business operations.

Analogy: Think of OLTP as the **cash register** at a store. Every sale, return, or exchange is recorded immediately. The system must be fast and accurate for each individual transaction.

Characteristics:

- Many small, fast transactions
- Normalized data (minimal redundancy)
- Current data (real-time state)
- Users: Front-line employees, customers
- Examples: ATM withdrawals, online shopping cart, seat reservations

5.2 OLAP: Online Analytical Processing

Definition:

OLAP Online Analytical Processing systems support complex queries across large volumes of historical data. They're optimized for reading and aggregating data, not for individual transaction updates.

Analogy: Think of OLAP as the **corporate headquarters analytics department**. They don't process individual sales—they analyze millions of past sales to find patterns like "Q3 revenue by region" or "customer churn rate over time."

Characteristics:

- Complex queries across large datasets
- Denormalized data (optimized for reading)
- Historical data (aggregated over time)
- Users: Analysts, executives, data scientists
- Examples: Quarterly sales reports, customer segmentation, trend analysis

Table 3: *OLTP vs OLAP Comparison*

Aspect	**OLTP**	**OLAP**
Purpose	Real-time transactions	Business analysis
Analogy	Cash register	Corporate analytics
Data	Current, operational	Historical, aggregated
Operations	Insert, Update, Delete	Complex SELECT queries
Users	Employees, customers	Analysts, executives
Response time	Milliseconds	Seconds to minutes

5.3 Comparison Table

5.4 ETL: Bridging OLTP and OLAP

Definition:

ETL **Extract, Transform, Load** is the process that moves data from operational (OLTP) systems to analytical (OLAP) systems:

1. **Extract**: Pull data from source systems (databases, APIs, files)
2. **Transform**: Clean, validate, aggregate, and restructure the data
3. **Load**: Write the transformed data into the analytical system

Key Information

Reverse ETL: When insights from analytical systems need to flow **back** to operational systems (e.g., ML model predictions feeding into a CRM), this is called Reverse ETL—a growing pattern in modern data architectures.

6 SQL vs NoSQL: ACID vs BASE

How data is stored depends on the consistency requirements of your use case.

6.1 ACID Properties (SQL/Relational Databases)

Definition:

ACID ACID is a set of properties guaranteeing reliable transaction processing in traditional relational databases:

- **Atomicity**: Transactions complete entirely or not at all (no partial updates)
- **Consistency**: Data always moves from one valid state to another
- **Isolation**: Concurrent transactions don't interfere with each other
- **Durability**: Once committed, data persists even after system failure

Example:

Bank Transfer (ACID Required) When transferring $1,000 from Account A to Account B:

- Account A: -$1,000
- Account B: +$1,000

Both operations must succeed or **both** must fail. If only one succeeds, money is created or destroyed—a catastrophic inconsistency. This is why financial systems require ACID compliance.

6.2 BASE Properties (NoSQL Databases)

Definition:

BASE BASE is a more relaxed consistency model for distributed NoSQL systems:

- **Basically Available**: The system guarantees availability—it will always respond
- **Soft State**: Data may be inconsistent across nodes temporarily
- **Eventual Consistency**: Given enough time, all nodes will converge to the same state

Example:

Social Media Likes (BASE Acceptable) When you ”like” a post on social media:

- Your friend might see 100 likes
- You might see 101 likes
- After a few seconds, both see 101 likes

This temporary inconsistency is acceptable because:

1. It's not financially critical
2. High availability (service never down) matters more
3. Users tolerate slight delays in like counts

6.3 The CAP Theorem

Important:

CAP Theorem In any distributed data system, you can only guarantee **two out of three** properties simultaneously:

- **Consistency**: All nodes see the same data at the same time
- **Availability**: Every request receives a response (success or failure)
- **Partition Tolerance**: System continues operating despite network failures

Since network partitions are inevitable in distributed systems, the real choice is between CP (consistency + partition tolerance) and AP (availability + partition tolerance):

- **CP systems**: Traditional RDBMS, prioritize consistency
- **AP systems**: NoSQL (Cassandra, DynamoDB), prioritize availability

7 Evolution of Data Platforms

Data platforms have evolved dramatically over the past 40 years. Understanding this history helps explain why modern architectures look the way they do.

7.1 Generation 1: Data Warehouses (1980s)

- **Purpose**: Business intelligence and reporting
- **Data type**: Structured data only (SQL)
- **Technology**: Proprietary systems (Teradata, Oracle)
- **Limitations**: Expensive hardware, no unstructured data support, limited scalability

7.2 Generation 2: Hadoop and the Data Lake (2010s)

Definition:

Hadoop An open-source framework for distributed storage (HDFS) and processing (MapReduce) across clusters of commodity hardware. Originated at Yahoo in 2006.

Key innovations:

- **Commodity hardware**: Use cheap, off-the-shelf servers instead of expensive proprietary machines
- **HDFS**: Hadoop Distributed File System—store data across many machines with 3x replication
- **Data Lake**: Store **all** data (structured, semi-structured, unstructured) in raw form
- **Open source**: Free software, vibrant ecosystem

Warning

Why Hadoop/MapReduce Was Slow
MapReduce processing follows a Map $\rightarrow$ Reduce pattern. The critical problem: **every stage writes intermediate results to disk** (HDFS), then the next stage reads from disk.
Disk I/O is thousands of times slower than memory access. This made Hadoop processing painfully slow for iterative algorithms (like machine learning).

7.3 Generation 3: Apache Spark (2012+)

Definition:

Apache Spark A unified analytics engine for large-scale data processing. Originally developed at UC Berkeley's AMPLab, Spark is 10-100x faster than Hadoop MapReduce for most workloads.

Key Summary

Why Spark is Fast: In-Memory Processing
Spark's key innovation: keep data **in RAM** (memory) as much as possible, only spilling to disk when necessary.

Additionally, Spark uses:

- **DAG (Directed Acyclic Graph)**: Plans the entire computation before executing
- **Lazy Evaluation**: Transformations aren't executed until an "action" (like `count()` or `collect()`) triggers computation
- **RDD/DataFrame abstraction**: Resilient Distributed Datasets provide fault tolerance without constant disk writes

7.4 Generation 4: The Lakehouse (2020s)

Definition:

Lakehouse A modern architecture combining the best of Data Warehouses (ACID transactions, governance, reliability) with Data Lakes (flexibility, all data types, low cost). Examples: Databricks Delta Lake, Apache Iceberg.

- **From Data Warehouse**: ACID transactions, schema enforcement, data governance
- **From Data Lake**: Support for all data types, scalable cloud storage, open formats
- **Key technology**: Delta Lake adds a transaction log to Parquet files, enabling ACID on data lakes

8 Understanding Spark Architecture

Since we'll use Spark extensively through Databricks, understanding its architecture is essential.

8.1 Core Components

1. **Driver Program**: The master process that coordinates the overall execution
2. **Cluster Manager**: Allocates resources across the cluster (YARN, Kubernetes, or Spark Standalone)
3. **Worker Nodes**: Execute the actual computation tasks
4. **Executors**: JVM processes on worker nodes that run tasks and cache data

8.2 RDDs and DataFrames

Definition:

RDD (Resilient Distributed Dataset) The original Spark abstraction. RDDs are:

- **Resilient**: Can recover from node failures
- **Distributed**: Data partitioned across cluster nodes
- **Immutable**: Transformations create new RDDs, never modify existing ones

Definition:

DataFrame A higher-level abstraction built on RDDs. DataFrames organize data into named columns (like a SQL table) and enable Spark's query optimizer to generate efficient execution plans. **This is what you'll use in practice.**

8.3 Transformations vs Actions

- **Transformations**: Operations that define a new DataFrame (e.g., `select()`, `filter()`, `groupBy()`). They are **lazy**—nothing executes immediately.
- **Actions**: Operations that trigger actual computation (e.g., `count()`, `collect()`, `show()`). Only when an action is called does Spark execute the transformation pipeline.

Example:

Lazy Evaluation in Action

```python
# These are transformations - nothing executes yet
df = spark.read.csv("data.csv")
df_filtered = df.filter(df["age"] > 21)
df_selected = df_filtered.select("name", "age")

# This is an action - NOW everything executes
df_selected.count()  # Triggers the entire pipeline
```

Spark waits until the `count()` action to optimize and execute the entire pipeline at once.

9 Cloud Computing Models: IaaS, PaaS, SaaS

Understanding cloud service models helps contextualize where Databricks fits.

9.1 The Pizza Analogy

Example:

Pizza as a Service Imagine you want pizza:

- **IaaS (Infrastructure as a Service)**: You get a kitchen with an oven. You buy flour, tomatoes, cheese, and make everything from scratch. Maximum control, maximum effort.
- **PaaS (Platform as a Service)**: You get pre-made dough and sauce. You just add toppings and bake. Less work, less control.
- **SaaS (Software as a Service)**: You order delivery. Pizza arrives ready to eat. Minimal effort, no control over how it's made.

Table 4: *Cloud Service Models*

Model	You Manage	Provider Manages	Examples
IaaS	OS, runtime, data, apps	Hardware, networking, virtualization	AWS EC2, Azure VMs
PaaS	Data, applications	Everything else	Heroku, Google App Engine
SaaS	Just use it	Everything	Gmail, Salesforce, **Databricks**

9.2 Databricks as SaaS

Databricks is a SaaS platform for data engineering and analytics. You don't manage:

- Cluster provisioning and scaling
- Spark installation and configuration
- Storage infrastructure
- Security patching

You focus on:

- Writing data pipelines
- Building ML models
- Creating dashboards and reports

10 The Medallion Architecture: Bronze, Silver, Gold

A common pattern for organizing data in modern lakehouse architectures.

Definition:

Medallion Architecture A multi-layer approach to organizing data by quality level:

- **Bronze Layer**: Raw data, ingested as-is from sources
- **Silver Layer**: Cleaned, validated, and enriched data
- **Gold Layer**: Aggregated, business-level data ready for consumption

10.1 Why Bronze Matters

Key Information

Never Throw Away Raw Data

The Bronze layer preserves original data because:

1. Business logic may change, requiring reprocessing
2. New use cases may need different transformations
3. Debugging issues requires access to source data
4. Compliance may require data lineage tracking

Storage is cheap. Recreating lost data is expensive or impossible.

10.2 Layer Characteristics

Table 5: *Medallion Architecture Layers*

Layer	Data State	Transformations	Users
Bronze	Raw, as-is	None or minimal	Data engineers
Silver	Cleaned, validated	Deduplication, type casting, null handling	Data engineers, analysts
Gold	Business-ready	Aggregation, joins, business logic	Business users, dashboards

11 Lab 0: Getting Started with Databricks

11.1 Databricks Free Edition Setup

Warning

Free Edition vs Trial Account

Databricks offers two types of accounts:

- **Free Edition**: Completely free forever for learning. Look for ”Free Edition” logo in top-left.
- **Trial Account**: 14-day trial of enterprise features, then requires payment.

Make sure you sign up for the Free Edition! The Trial account has different features and may cause notebook loading errors.

11.2 Key Concepts in Databricks

1. **Workspace**: Your personal file system in Databricks for notebooks and folders
2. **Notebook**: An interactive document mixing code, text, and visualizations
3. **Cluster/Compute**: The processing power that runs your code
4. **Catalog**: Top-level container for organizing data (Unity Catalog)
5. **Schema**: A collection of tables within a catalog (like a database)
6. **Volume**: A path to cloud storage for files

11.3 Magic Commands

In Databricks notebooks, magic commands change the cell’s language:

```
%python     # Execute Python code (default)
%sql        # Execute SQL queries
%md         # Render Markdown text
%sh         # Run shell commands
%fs         # File system operations (DBFS)
```

11.4 Your First Spark Code

```
# Path to a public dataset in Databricks
file_path = "/databricks-datasets/bikeSharing/data-001/day.csv"

# Read CSV into a Spark DataFrame
df = spark.read.format("csv") \
         .option("header", "true") \
         .option("inferSchema", "true") \
         .load(file_path)

# Display the results in a nice table
display(df)
```

Listing 1: Reading a CSV file into a DataFrame

11.5 Creating Tables with SQL

```sql
-- Set context to your catalog and schema
USE CATALOG csci_e103;
USE SCHEMA lab00;

-- Create a table from existing data
CREATE TABLE IF NOT EXISTS bike_data AS
SELECT * FROM parquet.`/databricks-datasets/samples/lending_club/parquet/`;

-- Query your new table
SELECT * FROM bike_data LIMIT 10;
```

Listing 2: Creating and querying a table

Key Information

Delta Lake is the Default

When you `CREATE TABLE` in Databricks, it automatically creates a Delta table (not regular Parquet). Delta adds:

- ACID transactions
- Time travel (access previous versions)
- Efficient updates and deletes

12 Best Practices for Data Platforms

12.1 Architectural Principles

1. **Decouple Storage and Compute**: Storage grows forever; compute should scale independently based on workload needs.
2. **Use Open Formats**: Avoid vendor lock-in by using open formats like Parquet, Delta Lake, or Iceberg. If your data is in a proprietary format, migration becomes expensive.
3. **Service-Oriented Design**: Build reusable, modular components. Each pipeline should do one thing well.
4. **Right Tool for the Right Job**: Not every problem needs the same solution. Consider trade-offs:
   - Latency requirements
   - Throughput needs
   - Access patterns
   - Cost constraints

12.2 Business Alignment

Key Information

Technology Serves Business
Every technical decision should have business justification:
- What problem does this solve?
- What's the expected ROI?
- Who is the economic buyer?
- What are the success metrics (KPIs)?

The thrill of trying new technology isn't sufficient justification. Projects without business alignment rarely make it to production.

12.3 Build vs Buy

- **Build**: Leverage in-house expertise, full control, but costs time
- **Buy**: Faster time-to-market, proven solutions, but costs money and reduces control

When competitors are catching up and you lack internal expertise, buying may be the right choice even if building would be technically possible.

13 Summary and Key Takeaways

Key Summary

What We Learned

1. **Data Engineering** transforms raw data into valuable insights—it's the "plumbing" that makes analytics and ML possible.
2. **The 5 V's of Big Data**: Volume (size), Velocity (speed), Variety (types), Veracity (quality), and Value (business impact).
3. **OLTP vs OLAP**: Transactional systems (fast, current data) vs analytical systems (complex queries, historical data). ETL bridges them.
4. **ACID vs BASE**: Strong consistency (SQL/relational) vs eventual consistency (NoSQL). Choose based on use case requirements.
5. **CAP Theorem**: In distributed systems, choose two of three: Consistency, Availability, Partition Tolerance.
6. **Platform Evolution**: Data Warehouses → Hadoop/Data Lakes → Spark → Lakehouse. Each generation solved problems of the previous.
7. **Spark's Speed**: In-memory processing + lazy evaluation + DAG optimization = 100x faster than Hadoop MapReduce.
8. **Medallion Architecture**: Bronze (raw) → Silver (cleaned) → Gold (business-ready) for progressive data refinement.
9. **Databricks**: A SaaS platform providing managed Spark, Delta Lake, and collaborative notebooks.

Warning

Action Items for This Week

- ☐ Create a Databricks **Free Edition** account (NOT Trial!)
- ☐ Import Lab 0 files into your workspace
- ☐ Connect to serverless compute and run the initialize notebook
- ☐ Complete all Lab 0 notebooks to familiarize yourself with the platform
- ☐ Read Chapters 1-3 of "Simplifying Data Engineering and Analytics with Delta"

CSCI E-103: Data Engineering for Analytics
Lecture 02: Data Modeling and Metadata

Harvard Extension School

Fall 2024

■ **Course:**	CSCI E-103: Data Engineering for Analytics
■ **Lecture:**	Lecture 02: Data Modeling & Metadata
■ **Instructor:**	Anindita Mahapatra & Eric Gieseke
■ **Objective:**	Master data modeling techniques (conceptual, logical, physical), understand OLTP vs OLAP systems, compare DWH architectures, and learn data formats and compression

Key Summary

This lecture covers the core discipline of **data modeling**—organizing data to meet business needs. We explore the three modeling levels (conceptual, logical, physical), contrast OLTP and OLAP systems, examine dimensional modeling with Star and Snowflake schemas, and compare data warehouse architectures (Inmon, Kimball, Data Vault). We also cover metadata management, data formats (CSV, JSON, Parquet, Delta Lake), compression techniques, and data profiling. The lab introduces housing price prediction using linear regression.

Contents

1 Review: Key Concepts from Lecture 01

Before diving into data modeling, let's reinforce the foundational concepts:

Table 1: *Lecture 01 Key Terms Review*

Term	Key Explanation
ACID	Atomicity, Consistency, Isolation, Durability (relational databases, bank transactions)
BASE	Basically Available, Soft State, Eventual Consistency (NoSQL, social media)
CAP Theorem	Distributed systems can only have 2 of 3: Consistency, Availability, Partition Tolerance
IaaS/PaaS/SaaS	Cloud service models (Infrastructure, Platform, Software as a Service)
Lakehouse	Combines data lake (flexibility) + data warehouse (reliability)
DAG	Directed Acyclic Graph—defines workflow in data pipelines
ETL	Extract, Transform, Load—moving data between systems
5 V's of Big Data	Volume, Velocity, Variety, Veracity (accuracy), Value
Spark Advantage	In-memory processing (100x faster than Hadoop's disk-based MapReduce)

Warning

The Biggest Challenge: Data Quality and Staleness
According to industry research, the biggest problems in big data aren't volume or velocity—they're ensuring data is **accurate** and **up-to-date**. "Garbage In, Garbage Out" (GIGO) remains the cardinal rule: no amount of sophisticated modeling can fix fundamentally bad data.

2 What is Data Modeling?

Definition:

Data Modeling Data modeling is the process of **organizing and structuring data** to meet business process requirements. It creates a "blueprint" that defines how data is stored, accessed, and related—transforming real-world complexity into a computer-understandable structure.

2.1 Why Data Modeling Matters

1. **Consistency and Quality**: Standardizes names, rules, and formats across the organization
2. **Efficient Storage and Retrieval**: Optimizes how data is persisted and queried
3. **Communication Tool**: Creates common vocabulary between business and technical teams
4. **Early Error Detection**: Catches inconsistencies in design phase, not production
5. **Documentation**: Serves as living documentation of data assets

Key Information

Cost of Late Discovery
Fixing a data model error during design costs \$1. Fixing the same error in development costs \$10. Fixing it in production costs \$100+. Data modeling is an investment that pays dividends throughout the system's lifecycle.

2.2 The Three Levels of Data Modeling

Data modeling progresses from abstract business concepts to concrete technical implementations:

2.2.1 1. Conceptual (Semantic) Model

- **Purpose**: Define core business concepts and rules
- **Audience**: Business stakeholders, domain experts
- **Focus**: Entities and relationships (e.g., "Customer purchases Product")
- **Technical details**: None—purely business-focused

2.2.2 2. Logical Model

- **Purpose**: Define data structure, attributes, and relationships in detail
- **Audience**: Developers, data architects, business analysts
- **Focus**: Entity attributes, data types, keys, cardinality
- **Technical details**: Technology-agnostic (not tied to specific database)

2.2.3 3. Physical Model

- **Purpose**: Translate logical model into specific database technology

- **Audience**: DBAs, data engineers
- **Focus**: Table names, column types, indexes, partitions, constraints
- **Technical details**: Fully specified for target platform (e.g., PostgreSQL, Delta Lake)

Example:

Online Bookstore Example **1. Conceptual Model**: "Customer orders Book"
2. Logical Model:

- Customer(CustomerID [PK], Name, Email)
- Book(BookID [PK], Title, Author)
- Order(OrderID [PK], OrderDate, CustomerID [FK], BookID [FK])

3. Physical Model (PostgreSQL):

```sql
CREATE TABLE T_CUSTOMER (
  CUST_ID SERIAL PRIMARY KEY,
  CUST_NAME VARCHAR(100) NOT NULL,
  EMAIL VARCHAR(255) UNIQUE
);
CREATE TABLE T_BOOK (
  BOOK_ID SERIAL PRIMARY KEY,
  TITLE VARCHAR(500) NOT NULL,
  AUTHOR VARCHAR(200)
);
CREATE TABLE T_ORDER_ITEMS (
  ORDER_ID INT NOT NULL,
  BOOK_ID INT REFERENCES T_BOOK(BOOK_ID),
  QUANTITY INT DEFAULT 1,
  PRIMARY KEY (ORDER_ID, BOOK_ID)
);
```

3 OLTP vs OLAP: Two Different Worlds

Database systems are designed for fundamentally different purposes. Understanding this distinction is crucial for choosing the right modeling approach.

3.1 OLTP: Online Transaction Processing

Definition:

OLTP Systems designed for real-time operational transactions—handling many concurrent users performing fast, individual read/write operations.

Analogy: The **bank teller's computer**—processing deposits, withdrawals, and transfers in real-time.

Characteristics:

- Many users, short transactions
- Data integrity is critical (ACID compliance)
- **Normalized** data (minimize redundancy)
- Current, operational data
- Examples: ATM systems, e-commerce carts, reservation systems

3.2 OLAP: Online Analytical Processing

Definition:

OLAP Systems designed for complex analytical queries across large volumes of historical data—supporting business intelligence and decision-making.

Analogy: The **corporate headquarters analytics department**—analyzing years of sales data to find patterns.

Characteristics:

- Few users, complex queries
- Query speed is critical
- **Denormalized** data (optimize for reads)
- Historical, aggregated data
- Examples: Sales reports, customer segmentation, trend analysis

Table 2: *OLTP vs OLAP Systems*

Aspect	OLTP	OLAP
Purpose	Day-to-day operations	Decision support
Design	Application-oriented	Subject-oriented
Data	Current, up-to-date	Historical, summarized
Operations	Read/Write/Update	Mostly Read (scans)
Data Size	Gigabytes	Terabytes to Petabytes
Performance	Transaction throughput	Query response time
Modeling	**ER Model (Normalized)**	**Dimensional (Denormalized)**

3.3 Comparison Table

Key Information

The Bridge: ETL

ETL (Extract, Transform, Load) connects OLTP and OLAP worlds. Data flows from operational systems (OLTP) through transformation pipelines into analytical systems (OLAP), enabling business intelligence without impacting operational performance.

4 Dimensional Modeling for Analytics

OLAP systems use **dimensional modeling** to optimize analytical queries. This approach separates data into two categories: Facts and Dimensions.

4.1 Facts and Dimensions

Definition:

Fact Table Contains the **measurable, quantitative data**—the "what" of business events.

- Numeric measures: sales amount, quantity, clicks
- Foreign keys to dimension tables
- Very large (millions/billions of rows), narrow (few columns)

Definition:

Dimension Table Contains the **descriptive context**—the "who, when, where, what" of business events.

- Descriptive attributes: customer name, product category, date details
- Provides filtering and grouping criteria
- Relatively small, wide (many columns)

Example:

Sales Analysis **Question**: "What was total revenue by product category and region last quarter?"
Fact Table: Sales transactions with amount, quantity, profit
Dimension Tables:

- Product: category, brand, price tier
- Customer: region, segment, acquisition date
- Time: date, quarter, year, day of week

4.2 Star Schema

Definition:

Star Schema A dimensional model where one central **fact table** is surrounded by multiple **dimension tables**. The visual representation resembles a star.

Characteristics:

- Dimension tables are **denormalized** (contain redundant data)
- Only **one join** needed between fact and dimension
- **Fast query performance**
- Higher storage due to redundancy

4.3 Snowflake Schema

Definition:

Snowflake Schema An extension of star schema where dimension tables are **normalized** into sub-dimensions. The visual representation resembles a snowflake.

Characteristics:

- Dimension tables are **normalized** (separated into related tables)
- **Multiple joins** needed to traverse dimension hierarchies
- **Lower storage** due to reduced redundancy
- Slower query performance

Table 3: *Star Schema vs Snowflake Schema*

Aspect	**Star Schema**	**Snowflake Schema**
Dimension tables	Denormalized	Normalized
Data redundancy	High	Low
Joins required	Few (typically 1)	Many (multiple levels)
Query performance	**Faster**	Slower
Storage efficiency	Lower	Higher
Complexity	Simple	Complex

Key Information

Modern Preference: Star Schema
With cheap storage and powerful compute, most modern data warehouses prefer star schemas. The query performance benefits outweigh storage costs. Snowflake schemas are used when storage is truly constrained or when data governance requires strict normalization.

5 NoSQL Data Modeling

NoSQL databases require a fundamentally different mindset for data modeling.

5.1 The Key Difference: Query-First Design

Warning

Different Questions

- **SQL/Relational**: "Given this data structure, what questions can I answer?"
- **NoSQL**: "Given the questions business needs answered, how should I structure data?"

NoSQL is **schema-flexible**, not schema-free. The modeling is just as important—it's driven by **access patterns** rather than normalization theory.

5.2 Denormalization is the Norm

NoSQL databases often don't support joins (or support them poorly). Therefore:

- **Duplicate data** to avoid joins
- Structure data so **one query retrieves everything needed**
- Optimize for **read performance** over storage efficiency

5.3 Embedded vs Referenced Documents

Example:

Blog Posts and Comments **Embedded Model (Denormalized)**:

```
{
  "post_id": "p123",
  "title": "My First Post",
  "content": "Hello world!",
  "comments": [
    { "user": "alice", "text": "Great post!" },
    { "user": "bob", "text": "Welcome." }
  ]
}
```

Pros: One query gets everything
Cons: Document grows unbounded with comments
Referenced Model (Normalized):

```
// Posts Collection
{ "post_id": "p123", "title": "My First Post" }

// Comments Collection
{ "comment_id": "c1", "post_id": "p123", "user": "alice" }
{ "comment_id": "c2", "post_id": "p123", "user": "bob" }
```

Pros: Scalable, documents stay small
Cons: Two queries (or expensive join) needed

Key Information

Rule of Thumb

- **Embed** when: Data is accessed together, bounded growth, 1:1 or 1:few relationships
- **Reference** when: Data accessed independently, unbounded growth, many:many relationships

6 Data Warehouse Architectures

Three major approaches have shaped how organizations build data warehouses.

6.1 Inmon (Top-Down)

Definition:

Inmon Approach Build a **centralized, normalized (3NF) enterprise data warehouse** first. Then create department-specific data marts from this single source of truth.

Analogy: "Paint the entire house first, then decorate individual rooms"

Characteristics:

- Third Normal Form (3NF) for the central DWH
- Data marts derived from the warehouse
- Strong data consistency and integrity
- Long initial implementation time

6.2 Kimball (Bottom-Up)

Definition:

Kimball Approach Build **department-specific data marts** (using star schema) first. The enterprise warehouse emerges as the collection of these marts connected by conformed dimensions.

Analogy: "Build LEGO blocks first, then assemble the structure"

Characteristics:

- Dimensional model (star/snowflake) from the start
- Faster initial delivery of business value
- Risk of inconsistent data across marts
- Conformed dimensions needed to maintain consistency

6.3 Data Vault (Hybrid)

Definition:

Data Vault A modeling technique designed for **flexibility and auditability**. Separates data into three components: Hubs (business keys), Links (relationships), and Satellites (attributes and history).

Analogy: "A vault that tracks every change to every piece of data"

Components:

- **Hubs**: Core business entities (e.g., CustomerID)

- **Links**: Relationships between hubs
- **Satellites**: Descriptive attributes and their change history

Advantages:

- Easily add new data sources
- Full audit trail of all changes
- Handles change better than Inmon/Kimball

Table 4: *DWH Architecture Comparison*

Aspect	**Inmon**	**Kimball**	**Data Vault**
Approach	Top-down	Bottom-up	Hybrid
Structure	3NF	Star schema	Hub/Link/Satellite
Time to value	Slow	Fast	Medium
Single source of truth	Yes	No	Yes
Handle change	Poor	Poor	**Good**
Complexity	Medium	Low	High

7 Modern Architecture: Medallion (Bronze/Silver/Gold)

The Medallion Architecture organizes data by quality level in lakehouse environments.

7.1 The LEGO Analogy

Key Information

Data Conformance = Building with LEGO

Raw data is like a pile of mismatched LEGO pieces. Data engineering transforms them into:

1. **Sorted**: Organized by color/type
2. **Arranged**: Grouped logically
3. **Consistent**: Standardized sizes

The result: "Conformed data" that anyone can build with.

7.2 The Three Layers

Definition:

Bronze Layer (Raw) **Purpose**: Ingest source data with minimal transformation

Characteristics:

- Data as-is from sources
- Maybe add ingestion timestamp, source file name
- Preserves original for audit and reprocessing

Definition:

Silver Layer (Cleaned) **Purpose**: Clean, validate, and enrich data

Characteristics:

- Deduplicated
- Data types corrected
- Null values handled
- Business rules applied

Definition:

Gold Layer (Business-Ready) **Purpose**: Aggregated, business-level data for consumption

Characteristics:

- Star schemas and data marts
- Pre-computed aggregations
- Ready for dashboards, reports, ML

Warning

Why Keep Bronze?

Never throw away raw data! Business logic changes, new use cases emerge, and you may need to reprocess from scratch. Storage is cheap—recreating lost data is expensive or impossible.

8 Metadata: Data About Data

Definition:

Metadata Information that describes, explains, and provides context about data. It answers questions like: Who created this? Where did it come from? How should it be used?

8.1 Categories of Metadata

1. **Business Metadata**: Business definitions, KPIs, data ownership, glossary
2. **Technical Metadata**: Schema, data types, lineage, storage location
3. **Operational Metadata**: SLAs, refresh frequency, usage patterns, freshness

8.2 Key Metadata Questions

Table 5: *Essential Metadata Questions*

Category	Questions
Who	Created? Manages? Uses? Owns? Regulates?
What	Business definition? Rules? Abbreviations?
Where	Stored? Source? Used? Backup?
When	Created? Updated? Retention period?
Why	Purpose? Business drivers?
How	Formatted? Accessed? Protected?

8.3 Data Catalogs

Definition:

Data Catalog A centralized repository for storing and managing metadata. Enables data discovery, governance, lineage tracking, and quality monitoring.

Examples: Databricks Unity Catalog, AWS Glue Catalog, Apache Atlas

9 Data Formats

Choosing the right data format significantly impacts storage, performance, and compatibility.

9.1 Text vs Binary Formats

Table 6: *Data Format Comparison*

Format	Type	Orientation	Splitable	Use Case
CSV	Text	Row	No	Simple interchange
JSON	Text	Row	No*	APIs, documents
Avro	Binary	Row	Yes	Streaming, schema evolution
Parquet	Binary	Column	Yes	Analytics, DWH
ORC	Binary	Column	Yes	Hive/Hadoop analytics
Delta	Binary	Column	Yes	Lakehouse (ACID)

9.2 Row vs Column Orientation

- **Row-oriented**: Fast writes, good for OLTP
- **Column-oriented**: Fast analytical queries (aggregations), better compression

Warning

Why Avoid Text Formats for Big Data?

- Inefficient storage (numbers as strings)
- No type checking (cars in numeric fields)
- No native compression
- Not splitable (can't parallelize)
- JSON repeats metadata in every record

9.3 Delta Lake

Definition:

Delta Lake An open-source storage layer that brings **ACID transactions** to data lakes. Built on Parquet files with a transaction log.

Key Benefits:

- ACID transactions on data lakes
- Time travel (access previous versions)
- Efficient upserts (UPDATE + INSERT in one operation)
- Schema enforcement and evolution

- Unified batch and streaming

10 Data Compression

Definition:

Compression Encoding data using fewer bits than the original representation, reducing storage and I/O bandwidth requirements.

10.1 Lossy vs Lossless

- **Lossy**: Some information lost (acceptable for images, audio)
- **Lossless**: Original data fully recoverable (required for databases)

10.2 Common Codecs

Table 7: *Compression Algorithms*

Codec	Splitable	Compression Ratio	Speed
Gzip (.gz)	No	High	Medium
Bzip2 (.bz2)	No	Very High	Slow
LZO (.lzo)	Yes	Medium	Fast
Snappy (.snappy)	Yes	Medium	Very Fast
LZ4	Yes	Medium	Very Fast
Zstandard (.zst)	Yes	High	Fast

Key Information

Splitable Matters

For distributed processing, compression format must be **splitable**—otherwise you can't parallelize across workers. Snappy and LZ4 are popular choices for big data because they're fast and splitable.

11 Data Profiling

Definition:

Data Profiling The process of examining data to understand its structure, quality, and characteristics before using it for analysis or ML.

11.1 Profiling Techniques

1. **Structural Discovery**: Schema consistency, data types, format correctness
2. **Content Analysis**: Min/max, mean, median, standard deviation, null counts, unique values
3. **Relationship Discovery**: Connections between datasets, entity resolution
4. **Data Correlation**: Univariate and multivariate analysis, identifying dependent variables
5. **Visualization**: Outlier detection, distribution analysis

11.2 Spark Data Profiling

```python
# Quick statistics
df.describe().show()

# More detailed summary
df.summary().show()

# Check nulls
from pyspark.sql.functions import col, isnan, when, count
df.select([count(when(col(c).isNull(), c)).alias(c)
           for c in df.columns]).show()
```

Listing 1: Basic Data Profiling in Spark

12 Lab 01: Housing Price Prediction

12.1 Overview

This lab walks through a complete ML workflow using housing data:

1. Load and explore data
2. Feature engineering
3. Train/test split
4. Linear regression model
5. Evaluate with RMSE

12.2 Key Steps

```python
# Read CSV into Spark DataFrame
df = spark.read.format("csv") \
    .option("header", "true") \
    .option("inferSchema", "true") \
    .load("/path/to/housing.csv")

# Add derived columns
from pyspark.sql.functions import from_unixtime, col
df = df.withColumn("date", from_unixtime(col("date")/1000))
df = df.withColumn("zipcode", col("zipcode").cast("string"))

# Write as Delta table
df.write.format("delta").mode("overwrite").saveAsTable("housing")
```

Listing 2: Loading and Preparing Housing Data

12.3 Training the Model

```python
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import numpy as np

# Split data
train_df, test_df = df.randomSplit([0.8, 0.2], seed=123)

# Train model
model = LinearRegression()
model.fit(X_train, y_train)

# Predict
predictions = model.predict(X_test)

# Evaluate
rmse = np.sqrt(mean_squared_error(y_test, predictions))
```

```python
print(f"RMSE: {rmse:.2f}")
```

Listing 3: Linear Regression with scikit-learn

12.4 Understanding RMSE

Definition:

RMSE (Root Mean Square Error) A measure of the average magnitude of prediction errors. Lower is better.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Where y_i is actual value and $\hat{y}_i$ is predicted value.

Key Information

Interpreting RMSE

RMSE depends on the scale of your target variable. For housing prices ranging from $50,000 to $5,000,000, an RMSE of $50,000 might be excellent. For prices ranging $100-$500, the same RMSE would be terrible. Always consider RMSE relative to your data range.

13 Summary and Best Practices

Key Summary

Key Takeaways

1. **Data Modeling** progresses: Conceptual → Logical → Physical
2. **OLTP** (operational) uses normalized models; **OLAP** (analytical) uses denormalized dimensional models
3. **Star Schema**: Fast queries, more storage (denormalized dimensions)
4. **Snowflake Schema**: Less storage, slower queries (normalized dimensions)
5. **DWH Approaches**: Inmon (top-down, 3NF), Kimball (bottom-up, dimensional), Data Vault (hybrid, audit-friendly)
6. **Medallion Architecture**: Bronze (raw) → Silver (cleaned) → Gold (business-ready)
7. **NoSQL Modeling**: Query-first design, denormalize for read performance
8. **Data Formats**: Use columnar formats (Parquet, Delta) for analytics; avoid text formats for big data
9. **Compression**: Choose splitable formats (Snappy, LZ4) for distributed processing
10. **Metadata**: Essential for data discovery, governance, and trust

Warning

Design Principles

- ☐ Design a **system**, not just a schema—schemas evolve
- ☐ Start with **core business data**—don't try to model everything at once
- ☐ Document with clear **metadata**—future you will thank you
- ☐ Identify **consumption patterns**—model for how data will be queried
- ☐ Match **compression and format** to data characteristics and access patterns

CSCI E-103: Data Engineering for Analytics
Lecture 03: Data Pipelines, ETL, and Streaming

Harvard Extension School

Fall 2024

■ **Course:**	CSCI E-103: Data Engineering for Analytics
■ **Lecture:**	Lecture 03: Data Pipelines & Streaming
■ **Instructor:**	Anindita Mahapatra & Eric Gieseke
■ **Objective:**	Understand data pipelines, ETL/ELT processes, batch vs streaming processing, and Lambda/Kappa architectures

Key Summary

This lecture covers the backbone of data engineering: **data pipelines**. We explore different pipeline types (batch, streaming, ETL, ELT), dive deep into streaming concepts (triggers, checkpoints, watermarks), and compare the **Lambda Architecture** (separate batch and streaming layers) with the modern **Kappa Architecture** (unified streaming). We also examine popular streaming frameworks (Kafka, Flink, Spark Structured Streaming) and discuss real-world trade-offs in pipeline design.

Contents

1 Review: Key Concepts from Lecture 02

Table 1: *Data Modeling and Storage Review*

Concept	**Key Explanation**	**Example**
3NF (Normalization)	Minimize data redundancy, ensure data integrity	Separate Customer, Order, Product tables
Denormalization	Accept redundancy for faster query performance	Store ”last_order_date” in Customer table
ETL / ELT	Extract → Transform → Load vs Extract → Load → Transform	Traditional DWH vs Data Lake
Star Schema	Fact table surrounded by dimension tables (denormalized)	Sales fact + Date, Customer, Product dims
Key-Value Store	Simplest NoSQL: key + value pairs	Amazon S3, Redis
Document Store	Flexible JSON/XML document storage	MongoDB, CouchDB
Columnar Store	Column-oriented for analytical workloads	Cassandra, DynamoDB
Parquet	Binary columnar format, optimized for analytics	Hadoop/Spark ecosystem
Delta Lake	ACID transactions on data lakes (Parquet + transaction log)	Databricks lakehouse
Metadata	Data about data (schema, lineage, history)	Delta transaction log

2 Table Types in Data Platforms

Before diving into pipelines, let's understand the different types of tables you'll encounter:

2.1 Permanent vs Temporary Tables

- **Permanent Tables**: Persist data durably—data survives session/cluster restarts
- **Temporary Tables**: Exist only for a limited scope
  - **Local**: Visible only within the current session
  - **Global**: Shared across sessions within the same cluster

2.2 Views and Materialized Views

Definition:

View (Virtual Table) A stored query definition—not the data itself. Every time you query a view, it re-executes the underlying query.
Pros: Always current data, no storage overhead
Cons: Slower (recomputes every time)

Definition:

Materialized View A view where the query results are **cached/stored**. Results are precomputed and persisted.
Pros: Much faster queries (no recomputation)
Cons: Can return stale data if not refreshed; good for aggregations and BI reports

2.3 Streaming Tables

Definition:

Streaming Table A special table type for streaming pipelines that automatically updates as new data arrives. The table maintains state and incrementally processes incoming events.

2.4 Managed vs External Tables

Table 2: *Managed vs External Tables*

Aspect	Managed Table	External Table
Location	Platform-managed (warehouse path)	User-specified external path
Data ownership	Platform owns data + metadata	Platform owns only metadata
DROP behavior	**Deletes data + metadata**	Deletes only metadata
Sharing	Less portable	More portable/shareable
Risk of loss	Higher (DROP = data gone)	Lower (data remains after DROP)

3 What is a Data Pipeline?

Definition:

Data Pipeline A data pipeline is the complete process of moving data from one point (source) to another (destination). Think of it as ”plumbing” for data—the infrastructure that enables data flow through an organization.

Pipelines can range from simple (copy file A to location B) to complex (ingest from multiple sources, apply multiple transformations, write to multiple destinations).

3.1 ETL Pipelines: A Special Purpose

Definition:

ETL Pipeline A specialized type of data pipeline designed to make data ”analytics-ready.” ETL (Extract, Transform, Load) pipelines:

- **Extract**: Pull data from source systems
- **Transform**: Clean, validate, enrich, aggregate
- **Load**: Write to destination (data warehouse, data mart, ML system)

Key Information

ETL vs ELT

- **ETL**: Transform *before* loading—traditional approach for structured data warehouses
- **ELT**: Load raw data first, transform *later*—modern approach for data lakes where schema may not be known upfront

3.2 Pipeline Lifecycle

Building a data pipeline is like building software:

1. **Design**: Define sources, transformations, destinations
2. **Implement**: Write the pipeline code
3. **Test**: Validate correctness with sample data
4. **Deploy**: Move to production environment
5. **Monitor**: Track execution, detect failures, measure performance
6. **Iterate**: Handle changes, update, redeploy

3.3 Pipeline Triggers

How does a pipeline know when to run?

- **File Arrival**: Triggered when new files appear in a directory
- **Schedule**: Cron-based (e.g., ”every day at 6 AM”, ”every Monday”)

- **Manual**: User explicitly initiates execution
- **Event-Based**: Triggered by external events (API call, message queue)

4 Pipeline Types

Table 3: *Data Pipeline Types*

Type	**Processing Style**	**Tools**	**Use Cases**
Batch	Periodic bulk processing	Spark, AWS Glue	Daily reports, billing
Streaming	Continuous event processing	Kafka, Flink, Kinesis	Fraud detection, IoT
ETL	Extract → Transform → Load	Informatica, Talend, dbt	DWH loading
ELT	Extract → Load → Transform	dbt, Snowflake	Data lake processing
Replication	Sync data between systems	Fivetran, AWS DMS	OLTP → OLAP sync
ML Pipeline	Data prep for ML training/inference	MLflow, Kubeflow	Model training
Orchestration	Coordinate multiple pipelines	Airflow, Prefect	Complex workflows

5 Why Streaming?

Important:

Speed **"It's all about SPEED."**
The goal of streaming is to transform event streams into actionable insights **faster**. When business decisions depend on real-time data, batch processing (hours/days) isn't acceptable.

5.1 The Speed Spectrum

Not every use case requires sub-second latency. Choose the right speed for your business needs:

Table 4: *Latency Requirements by Use Case*

Latency	Type	Use Cases
Hours to Days	**Batch**	ETL, billing, BI reports, ad-hoc analytics
Minutes	**Near Real-Time**	Mobile/IoT ingestion, log aggregation, clickstream
Seconds/Sub-second	**Real-Time**	Fraud detection, trading, gaming, ML inference

Warning

Don't Over-Engineer
Real-time processing is expensive (more compute, more complexity). Ask: "Do we **really** need this in real-time? What action will be taken with the insight?" If the answer is "generate a weekly report," batch is fine.

6 Streaming Concepts

6.1 Core Terminology

Table 5: *Essential Streaming Concepts*

Term	Description
Source & Sink	Every pipeline has a source (where data comes from) and sink (where data goes)
File-Based vs Event-Based	File-based: data lands on disk, then processed (slower). Event-based: data processed from memory/queue (faster, e.g., Kafka)
Micro-batch vs Continuous	Micro-batch: process small chunks periodically (Spark default). Continuous: process each event immediately (Flink)
Trigger	The interval at which micro-batches execute (e.g., every 30 seconds)
Output Modes	How to write to sink: Append (add new rows), Complete (overwrite all), Update (modify changed rows)
Checkpoint	**[Critical]** "Game save point"—records processing progress for fault recovery
Window	Time interval for aggregations (e.g., "sum over last 5 minutes")
Watermark	**[Critical]** How long to wait for late-arriving data before closing a window

6.2 Checkpoints: The Game Save Point

Definition:

Checkpoint A checkpoint records the exact position in the data stream that has been successfully processed. If the pipeline fails, it can restart from the checkpoint rather than reprocessing everything from the beginning.

Why checkpoints matter:

- **Exactly-once semantics**: Ensures data isn't processed twice or dropped
- **Fault tolerance**: Recover gracefully from failures
- **No manual tracking**: The platform manages "what's been processed" automatically

6.3 Watermarks: Handling Late Data

Definition:

Watermark A watermark defines the maximum allowed lateness for data. Data arriving after the watermark threshold is considered "too late" and may be dropped or handled separately.

Example:

Watermark Analogy Imagine a bus that waits 10 minutes past the scheduled departure time for late passengers. After 10 minutes, the bus leaves regardless.
Similarly, a 10-minute watermark means: ”Wait up to 10 minutes for late-arriving events. After that, close the aggregation window and move on.”

Warning

Watermarks Prevent OOM
Without watermarks, aggregation operations would need to keep state **forever** (in case late data arrives). Watermarks bound the state, preventing memory exhaustion.

7 Lambda vs Kappa Architecture

Warning

Terminology Alert: Lambda Architecture ≠ AWS Lambda
The **Lambda Architecture** discussed here is an **industry-standard architectural pattern** for data pipelines. It has nothing to do with AWS Lambda (Amazon's serverless compute service). The naming is coincidental.

7.1 Lambda Architecture (Old School)

Lambda Architecture was designed when streaming technology was immature. It tried to get the best of both worlds: **batch reliability** and **streaming speed**.

Definition:

Lambda Architecture A data processing architecture with **two parallel paths**:

1. **Batch Layer**: Processes all data periodically (slow but accurate)
2. **Streaming Layer**: Processes real-time data (fast but approximate)
3. **Serving Layer**: Merges results from both layers for queries

Pros:

- Balances speed and reliability

Cons (significant):

- **Code duplication**: Same logic implemented twice (batch + streaming)
- **Complexity**: Two pipelines to maintain
- **Reconciliation nightmare**: Ensuring batch and streaming results match is extremely difficult

7.2 Kappa Architecture (Modern)

Kappa Architecture emerged as streaming frameworks matured and could handle both real-time and batch workloads.

Definition:

Kappa Architecture A simplified architecture with **one unified streaming layer** that handles all data—whether real-time or batch.
Core idea: "Treat everything as a stream."

Pros:

- **Simplicity**: One codebase, one pipeline
- **Scalability**: Horizontal scaling
- **No reconciliation**: Results are inherently consistent

7.3 FAQ: How Does Kappa Handle Batch?

Key Information

Q: If Kappa is streaming-only, how do I run daily batch jobs?

A: Batch is just "streaming with a very long trigger interval."

Instead of `spark.read` (batch API), use `spark.readStream` (streaming API) with:

```python
.trigger(processingTime='24 hours')  # Wake up once per day
# or
.trigger(once=True)  # Run once, process all available data
```

The **design** is streaming (using `readStream`), but the **behavior** is batch-like. You still get checkpoint benefits—the platform tracks what's been processed.

7.4 Comparison Table

Table 6: *Lambda vs Kappa Architecture*

Aspect	Lambda	Kappa
Pipelines	Two (batch + streaming)	One (streaming only)
Code duplication	Yes (same logic twice)	No
Complexity	High	Low
Reconciliation	Required (difficult)	Not needed
Batch support	Separate batch layer	Streaming with long trigger
Modern preference	Legacy	**Recommended**

8 Streaming Processing Frameworks

Table 7: *Streaming Framework Comparison*

Framework	Processing Model	Latency	Best For	Notes
Kafka Streams	Event-at-a-time	Very low (<10ms)	Kafka integration	Lightweight Java library
Apache Flink	True streaming	Very low (ms)	Ultra-low latency, CEP	Complex stateful processing
Spark Structured Streaming	Micro-batch (default)	Medium (100ms+)	Unified batch+stream	Spark ecosystem
Apache Storm	Event-at-a-time	Very low	Legacy	Mostly obsolete
Apache Samza	Event-at-a-time	Low	LinkedIn use cases	Niche
KSQLDB	Continuous SQL	Low	SQL over Kafka	Easy Kafka streaming
Amazon Kinesis	Managed streaming	Medium	AWS native	Serverless option
Google Dataflow	Unified batch+stream	Low-Medium	GCP native	Apache Beam based
Azure Stream Analytics	SQL-based streaming	Medium	Azure native	Low-code

8.1 Framework Selection Guide

Key Summary

Rule of Thumb

- **Ultra-low latency + complex event processing?** → Apache Flink
- **Unified batch + streaming with Spark ecosystem?** → Spark Structured Streaming
- **Already using Kafka + simple processing?** → Kafka Streams or KSQLDB
- **Cloud-native, serverless preferred?** → Kinesis, Dataflow, or Azure Stream Analytics

8.2 File-Based vs Event-Based Streaming

- **File-Based**: Data lands on disk (S3, ADLS), then processed
  - Slower (disk I/O)
  - Simpler to implement
  - Spark autoloader, Delta Live Tables

- **Event-Based**: Data flows through message queue, processed in-memory
  - Faster (no disk)
  - More complex
  - Kafka, Kinesis, Event Hubs

9 Spark Structured Streaming

Spark Structured Streaming is the foundation for streaming in the Databricks/Lakehouse ecosystem.

9.1 Basic Streaming Pattern

```python
# Read from stream (note: readStream, not read)
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "host:9092") \
    .option("subscribe", "topic_name") \
    .load()

# Transform the data
transformed = df \
    .selectExpr("CAST(value AS STRING) as json_data") \
    .select(from_json("json_data", schema).alias("data")) \
    .select("data.*")

# Write to stream
query = transformed.writeStream \
    .format("delta") \
    .option("checkpointLocation", "/path/to/checkpoint") \
    .trigger(processingTime="30 seconds") \
    .start("/path/to/output")
```

Listing 1: Basic Spark Streaming Pipeline

9.2 Key Differences: Batch vs Streaming API

Table 8: *Batch vs Streaming API*

Batch	Streaming
`spark.read`	`spark.readStream`
`df.write`	`df.writeStream`
No checkpoint needed	**Checkpoint required**
Runs once, completes	Runs continuously (or triggered)

9.3 Trigger Options

```python
# Fixed interval micro-batch
.trigger(processingTime="10 seconds")

# Process all available data once, then stop
.trigger(once=True)

# Newer: available-now (process all, checkpoint, stop)
.trigger(availableNow=True)
```

```

# Continuous (true streaming, experimental)
.trigger(continuous="1 second")
```

Listing 2: Trigger Options

10 Trade-offs in Streaming Design

10.1 The Iterative Design Process

Streaming pipeline design is not "design once, build forever." It's an iterative process balancing requirements and costs:

1. **Understand Goals**: What latency does business need? What's the SLA?
2. **Define Strategy**: Which framework? What trigger interval? Watermark settings?
3. **Estimate Resources**: How many VMs? What cluster size? Storage tier?
4. **Calculate Costs**: What's the monthly bill?
5. **Re-evaluate Trade-offs**: If too expensive, can we relax latency requirements?

10.2 The Three Dimensions

- **Scalability**: How much data? What TPS (transactions per second)?
- **Processing**: How many transformations? Joins? Stateful operations?
- **Quality**: Exactly-once semantics? Deduplication? Disaster recovery?

10.3 Real-World Scenarios

Warning

Scenario 1: Storage Costs Higher Than Compute
Symptom: 70% of costs are storage, only 30% compute
Root Causes:

- **Small files problem**: Low-frequency streaming creates many tiny files. Cloud storage charges per API call (LIST, GET), so millions of small files = huge bills.
- **Wrong storage tier**: Active data stored in "Cool" tier (cheap storage, **expensive access**). Should be in "Hot" tier.

Solution: Increase trigger interval to create larger files; keep active data in hot storage.

Example:

Scenario 2: Multi-Tenancy **Goal**: Process data for 100 different customers, isolated from each other.
Bad approach: Create 100 separate clusters and jobs.
Problem: Massive cost, low resource utilization, operational nightmare.
Good approach: One large cluster processing all data, with `partitionBy("client_id")` to physically separate data in storage. Create views per customer in the serving layer.

11 Databricks Lakeflow and Jobs

11.1 Lakeflow Components

Databricks Lakeflow is the end-to-end data engineering platform:

1. **Connect**: Connectors for data ingestion (Oracle, SQL Server, Salesforce, Workday)
2. **Pipelines**: Transformation logic (Medallion architecture, Delta Live Tables)
3. **Jobs**: Workflow orchestration (scheduling, dependencies, monitoring)

11.2 Jobs Features

Table 9: *Databricks Jobs Capabilities*

Feature	**Description**
File Arrival Triggers	Start job when new files appear
Conditional Tasks	If/else branching based on previous task results
Job Parameters	Pass parameters to customize job execution
Webhooks	Notifications to Slack, email, PagerDuty
Duration Alerts	Alert if job exceeds expected runtime
Task Looping	For-each construct for repeated execution
Continuous Execution	Long-running streaming jobs
System Tables	Audit logs of all job executions
Modular Workflows	Call other jobs/workflows

12 Summary and Key Takeaways

Key Summary

Key Takeaways

1. **Data Pipelines** are the ”plumbing” that moves data through an organization. ETL pipelines are specialized for analytics readiness.
2. **Pipeline Types**: Batch (periodic), Streaming (continuous), ETL/ELT, Replication, ML, Orchestration
3. **Streaming Concepts**:
   - **Checkpoint**: ”Game save point” for fault recovery
   - **Watermark**: How long to wait for late data
   - **Trigger**: When to execute micro-batches
4. **Lambda Architecture**: Two pipelines (batch + streaming) merged in serving layer. Complex, code duplication, reconciliation headache. **Legacy approach.**
5. **Kappa Architecture**: Single streaming pipeline handles everything. Batch = streaming with long trigger. **Modern standard.**
6. **Framework Selection**:
   - Ultra-low latency → Flink
   - Unified batch+stream → Spark Structured Streaming
   - Kafka ecosystem → Kafka Streams or KSQLDB
7. **Trade-offs**: Balance latency, cost, and complexity. Not everything needs real-time processing.
8. **Batch API**: `spark.read` / `spark.write`
   Streaming API: `spark.readStream` / `df.writeStream`

Warning

The Golden Rule
Always ask: ”Do we really need real-time?”
Real-time is expensive. If the insight drives a weekly report, batch is fine. Design for business requirements, not technical elegance.

CSCI E-103: Data Engineering for Analytics
Lecture 04: Data Transformations, Design Patterns, and Compliance

Harvard Extension School

Fall 2024

■ **Course:**	CSCI E-103: Data Engineering for Analytics
■ **Lecture:**	Lecture 04: Transformations & Patterns
■ **Instructor:**	Anindita Mahapatra & Eric Gieseke
■ **Objective:**	Master data engineering design patterns, understand compliance (GDPR/CCPA), Spark internals (partitions, joins), and CDC/SCD concepts

Key Summary

This lecture covers essential data transformation concepts. We explore **design patterns** for big data (Lambda, Kappa, CQRS, Data Mesh), understand **compliance requirements** (GDPR, CCPA) and their engineering implications, dive into **Spark internals** (jobs/stages/tasks, table vs Spark partitions, Z-ordering, join strategies), and learn about **CDC (Change Data Capture)** and **SCD (Slowly Changing Dimensions)** for tracking data changes over time.

Contents

1 What Are Design Patterns?

Definition:

Design Pattern A design pattern is a **reusable, proven solution template** for common problems in software or data engineering. Patterns provide tested approaches that you can adapt to your specific context.

1.1 Why Use Design Patterns?

1. **Embody Good Design Principles**: Patterns naturally incorporate principles like abstraction, separation of concerns, and divide-and-conquer
2. **Common Vocabulary**: Instead of explaining complex designs every time, say ”let’s use Lambda architecture” and everyone understands
3. **Proven Solutions**: These patterns have been battle-tested in production systems
4. **Faster Development**: Don’t reinvent the wheel—use established templates

Warning

Apply Judiciously
”When you have a hammer, everything looks like a nail.” Don’t force-fit patterns. Always evaluate whether a pattern actually fits your problem context (data volume, latency requirements, cost constraints).

2 Big Data Design Patterns

Big data patterns differ from traditional software patterns (Gang of Four). They focus on scale, velocity, volume, and distributed processing challenges.

2.1 Patterns by Pipeline Stage

Table 1: *Design Patterns by Pipeline Stage*

Stage	**Pattern**	**Purpose**
Modeling	Star/Snowflake Schema	Analytical data warehouse structure
	Vault Modeling	Preserve all change history
Ingestion	Connector Pattern	Uniform interface to diverse sources
	Lambda/Kappa	Handle batch + streaming
	Compute/Storage Separation	Scale independently
Transform	Schema on Read/Write	When to define schema
	ACID Transactions	Data integrity at scale
	Multi-Hop Pipeline	Progressive data refinement
Storage	Columnar Storage	Fast analytical queries
	Denormalized Tables	Avoid expensive joins
Analytics	In-Stream Analytics	Real-time processing before storage

2.2 Architecture Patterns

2.2.1 Lambda Architecture

Splits data flow into two paths:

- **Batch Layer**: Slow but accurate—processes all historical data
- **Speed Layer**: Fast but approximate—processes real-time data
- **Serving Layer**: Merges results from both layers

Use case: When you need both real-time insights AND historically accurate analysis.

2.2.2 Kappa Architecture

Single streaming pipeline handles everything:

- All data treated as a stream
- For "batch" needs, replay the stream from beginning
- Simpler than Lambda (one codebase)

Use case: Real-time processing, event-driven systems, IoT.

2.2.3 Event-Driven Architecture (EDA)

System reacts to events (state changes):

- **Event Producers**: Generate events
- **Event Streaming**: Kafka, Event Hubs
- **Event Consumers**: React to events

Use case: Microservices, e-commerce transaction processing.

2.3 Storage Patterns

2.3.1 CQRS (Command Query Responsibility Segregation)

Definition:

CQRS Separate the system's **write operations** (commands) from **read operations** (queries) into different models that can be scaled independently.

Why? Write-heavy workloads and read-heavy workloads have different requirements. Separating them allows optimal scaling for each.

2.3.2 Polyglot Persistence

Definition:

Polyglot Persistence ”Use the right database for the job.” Instead of forcing all data into one database type, use multiple specialized databases:

- Relational DB for transactional data
- Document store for unstructured data
- Graph DB for relationship-centric data

2.3.3 Data Lake Pattern

Store all data (structured, semi-structured, unstructured) in its **raw form**:

- ”Store now, figure out how to use later”
- Preserves original data for future unknown use cases
- Uses Schema-on-Read (define schema when reading)

2.4 Processing Patterns

2.4.1 Micro-batch Processing

Process streaming data in small, regular intervals:

- Not true streaming, but ”near real-time”
- Batch interval defines processing frequency

- Spark Structured Streaming default mode

2.4.2 Multi-Hop (Medallion) Architecture

Progressive data refinement through stages:

- **Bronze (Landing)**: Raw data, minimal transformation
- **Silver (Refined)**: Cleaned, validated, business logic applied
- **Gold (Aggregated)**: Analytics-ready, pre-computed aggregations

Key insight: As you move toward Gold, data quality increases but data availability decreases (Bronze is available immediately).

2.4.3 Minimize Data Movement

Moving petabytes is expensive. Use:

- **Time Travel**: Version data—access historical states without copying
- **Zero-Copy Clone**: Clone tables by copying only metadata, not data
- **Delta Share**: Share data access without physical transfer

2.5 Other Notable Patterns

- **CAP Theorem Selection**: Choose CP (consistency) or AP (availability) based on needs
- **Data Mesh**: Decentralized data ownership—each domain team owns their data as a "product"
- **Connector/Bridge Pattern**: Uniform API across diverse data sources (e.g., JDBC)

3 Data Compliance: GDPR and CCPA

Data engineers have legal and ethical responsibilities when handling personal data.

3.1 Key Regulations

Definition:

GDPR (General Data Protection Regulation) EU regulation protecting personal data of EU citizens. Applies to ANY company processing EU citizen data, regardless of location. Heavy fines for violations (up to 4% of global revenue).

Definition:

CCPA (California Consumer Privacy Act) California law providing similar protections for California residents.

3.2 Engineering Implications

Two key rights create engineering challenges:

3.2.1 1. Right of Erasure (Right to be Forgotten)

Users can request deletion of their personal data.

Warning

The Challenge
In a data lake with petabytes of data across millions of files, how do you find and delete one user's records? Traditional formats like Parquet are immutable—you can't just delete a row.
Solution: Delta Lake, Hudi, Iceberg support fine-grained deletes.

3.2.2 2. Right of Portability

Users can request their data be exported and transferred to another service.

3.3 Technical Solutions

- **Fine-grained Updates/Deletes**: Use Delta Lake for row-level operations
- **Pseudonymization**: Replace identifiers with tokens
  - Store [User ID $\leftrightarrow$ Token] mapping separately
  - All analytics use tokens only
  - For erasure: just delete the mapping—data becomes unlinkable

4 Spark Execution: Jobs, Stages, Tasks

Understanding how Spark executes your code helps you write more efficient pipelines.

4.1 Execution Hierarchy

Key Summary

Job → Stages → Tasks

1. **Job**: Created for each **action** (e.g., `save()`, `collect()`)
2. **Stage**: Jobs split at **shuffle** boundaries (data redistribution points)
3. **Task**: Smallest unit—one task per partition, runs on executor cores in parallel

4.2 Shuffle Operations

Definition:

Shuffle The process of redistributing data across cluster nodes. Required for operations like `groupBy()`, `join()`, `reduceByKey()` where data from different partitions must be combined.

Why expensive?

- Data transferred over network between executors
- Disk I/O for intermediate data
- Synchronization overhead

Minimize shuffles when possible!

5 Table Partitions vs Spark Partitions

These are two completely different concepts that beginners often confuse.

Important:

Library Analogy

- **Table Partitions** = Physical bookshelves organized by category (reduces disk I/O)
- **Spark Partitions** = Number of librarians working in parallel (increases CPU parallelism)

5.1 Comparison Table

Table 2: *Table Partitions vs Spark Partitions*

Aspect	Table Partitioning	Spark Partitioning
Level	Database/Table	Processing/Runtime
What it is	Physical organization on disk by column values	Logical distribution across cluster nodes
Purpose	Minimize data scans (disk I/O)	Maximize parallelism (CPU)
Controlled by	User (explicit DDL)	Spark (dynamic, based on data/config)
Optimization	Partition Pruning	Shuffle optimization

5.2 Table Partitioning Best Practices

- Choose columns frequently used in `WHERE` clauses (date, country, region)
- Avoid high-cardinality columns (user_id)—creates millions of tiny partitions
- Target at least 1GB per partition
- **Modern advice**: For data < 1TB, don't partition—let Delta Lake optimize

```sql
CREATE TABLE sales (
    id INT,
    amount DECIMAL,
    sale_date DATE,
    country STRING
)
PARTITIONED BY (country, sale_date);
```

Listing 1: Creating a Partitioned Table

5.3 Z-Ordering

Definition:

Z-Ordering A Delta Lake optimization that co-locates related data on disk based on multiple columns. Improves query performance through data skipping.

When to use: Columns frequently filtered together but not suitable as partition keys.

```sql
-- Compact files and organize by ip_address, port
OPTIMIZE network_logs
ZORDER BY (ip_address, port);
```

Listing 2: OPTIMIZE with Z-Ordering

6 Spark Join Strategies

Spark automatically selects the most efficient join strategy based on data characteristics.

6.1 Broadcast Hash Join

- **Condition**: One table is small (fits in memory)
- **Mechanism**: Broadcast small table to all executors
- **Pros**: No shuffle, no sort—very fast
- **Cons**: Only works with small tables

```python
from pyspark.sql.functions import broadcast

# Explicitly broadcast the small table
df_result = df_large.join(broadcast(df_small), "key")
```

Listing 3: Forcing Broadcast Join

6.2 Shuffle Hash Join

- **Condition**: One side is 3x+ smaller, partition fits in memory
- **Mechanism**: Shuffle data, build hash table on smaller side
- **Pros**: Handles larger tables than broadcast
- **Cons**: Requires shuffle (no sort)

6.3 Sort Merge Join

- **Condition**: Default for large tables
- **Mechanism**: Shuffle both sides, sort, then merge
- **Pros**: Handles any data size, very robust
- **Cons**: Requires shuffle AND sort—slowest

Key Summary

Join Strategy Selection

1. **Small table?** → Broadcast Hash Join (fastest)
2. **Medium asymmetry?** → Shuffle Hash Join
3. **Both large?** → Sort Merge Join (default)

7 CDC and SCD: Tracking Data Changes

7.1 CDC: Change Data Capture

Definition:

CDC (Change Data Capture) A technique for identifying and capturing changes (inserts, updates, deletes) in source data so they can be replicated to target systems incrementally.

Why CDC?

- Full table reloads are expensive and slow
- Only process what changed since last sync
- Near real-time data synchronization

Common CDC approaches:

- **Log-based**: Read database transaction logs (most accurate)
- **Timestamp-based**: Query records modified after last sync
- **Trigger-based**: Database triggers capture changes

7.2 SCD: Slowly Changing Dimensions

Definition:

SCD (Slowly Changing Dimension) A dimension table attribute that changes over time (e.g., customer address). SCD defines how to handle historical values when changes occur.

7.2.1 SCD Type 1: Overwrite

- Simply update the record—no history kept
- **Use case**: Corrections, when history doesn't matter

```sql
UPDATE customers
SET address = 'New Address'
WHERE customer_id = 123;
```

Listing 4: SCD Type 1: Simple Update

7.2.2 SCD Type 2: Add New Row

- Keep all historical versions as separate rows
- Add columns: `effective_date`, `end_date`, `is_current`
- **Use case**: Full audit trail, historical analysis

```
-- Old record
| customer_id | address     | effective | end_date    | current |
|-------------|-------------|-----------|-------------|---------|
| 123         | Old Address | 2020-01-01| 2024-01-15 | false    |
| 123          | New Address | 2024-01-15| 9999-12-31 | true     |
```

Listing 5: SCD Type 2: Historical Record

7.3 MERGE Statement for CDC/SCD

Delta Lake's MERGE handles upserts (update or insert) efficiently:

```sql
MERGE INTO target_table t
USING source_table s
ON t.id = s.id
WHEN MATCHED THEN
    UPDATE SET t.value = s.value
WHEN NOT MATCHED THEN
    INSERT (id, value) VALUES (s.id, s.value)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE;
```

Listing 6: MERGE for Upsert Operations

8 Delta Lake Operations

8.1 OPTIMIZE: File Compaction

```sql
-- Compact small files into larger ones
OPTIMIZE my_table;

-- Compact and Z-order
OPTIMIZE my_table ZORDER BY (column1, column2);
```

Listing 7: OPTIMIZE Command

Why? Small files hurt query performance due to file listing overhead.

8.2 VACUUM: Clean Old Versions

```sql
-- Remove files older than retention period (default 7 days)
VACUUM my_table;

-- Custom retention (requires safety flag)
VACUUM my_table RETAIN 168 HOURS;  -- 7 days
```

Listing 8: VACUUM Command

Warning: After VACUUM, time travel to cleaned versions is impossible.

8.3 Time Travel

```sql
-- Query specific version
SELECT * FROM my_table VERSION AS OF 10;

-- Query by timestamp
SELECT * FROM my_table TIMESTAMP AS OF '2024-01-01';

-- Restore to previous version
RESTORE TABLE my_table TO VERSION AS OF 5;
```

Listing 9: Time Travel Queries

8.4 Clone: Zero-Copy Duplication

```sql
-- Shallow clone (metadata only)
CREATE TABLE my_table_clone SHALLOW CLONE my_table;

-- Deep clone (full copy)
CREATE TABLE my_table_copy DEEP CLONE my_table;
```

Listing 10: Clone Operations

9 Summary and Key Takeaways

Key Summary

Key Takeaways

1. **Design Patterns** provide proven solutions for common data engineering problems. Use them judiciously—match pattern to problem.
2. **Key Architecture Patterns**:
   - Lambda: Separate batch + streaming layers
   - Kappa: Unified streaming layer
   - CQRS: Separate read and write models
   - Data Mesh: Decentralized data ownership
3. **Compliance (GDPR/CCPA)**: Right to erasure and portability require fine-grained data operations. Use Delta Lake + pseudonymization.
4. **Spark Execution**: Job $\rightarrow$ Stages (shuffle boundary) $\rightarrow$ Tasks (partition)
5. **Two Types of Partitions**:
   - Table Partitions: Physical organization, minimize disk I/O
   - Spark Partitions: Logical distribution, maximize parallelism
6. **Z-Ordering**: Co-locate related data for multi-column filtering
7. **Join Strategies**: Broadcast (small table) > Shuffle Hash > Sort Merge (default)
8. **CDC**: Capture incremental changes from source systems
9. **SCD**: Handle dimension changes—Type 1 (overwrite) vs Type 2 (history)
10. **Delta Lake Operations**: OPTIMIZE (compact), VACUUM (clean), Time Travel, Clone

Warning

Practical Advice

☐ For data < 1TB, skip table partitioning—let Delta optimize

☐ Minimize shuffles in Spark—they're expensive

☐ Use MERGE for CDC/SCD operations

☐ Set up regular OPTIMIZE jobs for frequently updated tables

☐ Always consider compliance early—retrofitting is painful

CSCI E-103: Reproducible Machine Learning
Lecture 05: Data Lakes, Lakehouses, and Delta Lake

Harvard Extension School

Fall 2025

■ **Course:**	CSCI E-103: Reproducible Machine Learning
■ **Week:**	Lecture 05
■ **Instructors:**	Anindita Mahapatra & Eric Gieseke
■ **Objective:**	Master the evolution from data silos to data lakehouses, understand Delta Lake's role in providing reliability, and learn job orchestration fundamentals

Contents

1 Lecture Overview

Key Summary

This lecture covers the evolution of data storage paradigms and introduces the Data Lakehouse architecture:

Core Topics:

- **Data Silos** – The problem of isolated, compartmentalized data
- **Data Lakes** – Centralized storage for all data types (Schema on Read)
- **Data Swamps** – The danger of ungoverned data lakes
- **Data Lakehouses** – Combining lake flexibility with warehouse reliability
- **Delta Lake** – The protocol that enables ACID transactions on data lakes
- **Medallion Architecture** – Bronze/Silver/Gold data organization
- **Job Orchestration** – Managing multi-task data pipelines
- **Data Mesh vs. Data Fabric** – Organizational vs. technical paradigms

The central theme of this lecture is understanding how to "hydrate" a data lake properly so it doesn't become a data swamp. This means not just pushing data in, but ensuring you can effectively pull data out with reliability and governance.

2 Key Terminology Reference

Before diving into the details, here is a comprehensive reference table of the key concepts covered in this lecture:

Term	**Core Description (Analogy)**	**Key Characteristics**	**Relationship**
Data Silo	Isolated "data islands" per department. (e.g., Finance's Excel, Marketing's CRM)	- Data isolation and duplication - Cross-functional analysis impossible	The **problem** Data Lake solves
Data Lake	Giant "lake" storing all data in raw format.	- All data types (structured/unstructured) - **Schema on Read** - Low storage cost	Solves silos. Can become a swamp.
Data Swamp	Ungoverned lake where nobody knows what data exists or where.	- No governance - Missing metadata - Lost data trust	**Failed** Data Lake
Data Warehouse	"Department store" for cleaned, structured data only.	- Structured data only - **Schema on Write** - BI/Reports, high performance	Complementary to Lake. (Analogy: Ferrari)
Data Lakehouse	**Hybrid** combining Lake flexibility with Warehouse reliability.	- Lake + Warehouse - Single platform for BI and AI	Evolution of Data Lake. Delta Lake enables this.
Delta Lake	"Management protocol" making Data Lakes reliable.	- Parquet-based + Transaction Log - **ACID, Time Travel** support	**Core technology** for Lakehouses
Medallion Arch.	3-tier data refinement pipeline: Bronze → Silver → Gold	- **Bronze**: Raw data - **Silver**: Cleaned, filtered - **Gold**: Aggregated, business-ready	**Methodology** for managing Lakehouse data
DAG	"Directed Acyclic Graph" – represents task dependencies.	- Direction: execution order - Acyclic: no circular dependencies	**Operating principle** of Spark and workflows
Data Mesh	**Organizational culture** where each business domain owns its data.	- Decentralized ownership - Data as a Product	**Organizational/culture** strategyapplicable with Lakehouses
Data Fabric	**Technical architecture** connecting data across environments.	- Hybrid/multi-cloud integration - Metadata-driven automation	**Technology/architecture** focused

3 Evolution of Data Storage: From Silos to Lakehouses

Data storage and utilization approaches have evolved in response to changing business requirements. Understanding this evolution helps explain why modern architectures like the Lakehouse exist.

3.1 Data Silos: Isolated Islands

Definition:

Data Silo A **Data Silo** is a state where data is isolated across different departments or systems within an organization, preventing cross-functional analysis and creating inefficiencies.

Example:

The Isolated Islands Analogy Consider Company A: The Finance team stores revenue data in their Excel files, while Marketing stores customer click data in a separate marketing platform. Because these two data sources aren't connected, answering the question "Which marketing activities actually drove sales?" becomes nearly impossible.
Each department is trapped on its own "data island" with no bridge to the others.

Why do Data Silos form?

- **Structural Issues:** Systems were designed without data sharing in mind
- **Political Issues:** Departments treat data as "property" and refuse to share
- **Growth Issues:** New systems introduced during expansion don't integrate with existing ones
- **Vendor Lock-in:** SaaS solutions make data export difficult

Caution

Data silos lead to:

- Duplicate data maintenance costs
- Inconsistent "versions of truth"
- Inability to get a holistic view of customers (the "Customer 360" problem)
- Lost business opportunities due to fragmented insights

3.2 Data Lakes: The Great Unifier

Definition:

Data Lake A **Data Lake** is a centralized repository that stores all types of data (structured, semi-structured, unstructured) in their original raw format at any scale.

The Data Lake emerged to solve the silo problem with a simple philosophy: "Store everything in one place, figure out how to use it later."

Key Characteristics:

- **All Data Types:** Handles structured (tables), semi-structured (JSON, XML), and unstructured

(images, videos, logs)

- **Raw Storage:** Data is stored as-is without transformation
- **Low Cost:** Built on cheap cloud storage (S3, ADLS, GCS)
- **Schema on Read:** No schema required at write time

Very Important:

	Schema on Read (Data Lake)	**Schema on Write** (Dat
Schema on Read vs. Schema on Write	1. **Write:** Dump anything, no schema check 2. **Read:** Define/interpret schema at query time	1. **Write:** Strict schema 2. **Read:** Fast reads schema
	Pros: Fast ingestion, flexible **Cons:** Complex reads, quality issues	**Pros:** Data quality guara **Cons:** Rigid, can't store

3.3 Data Swamps: When Lakes Go Wrong

The flexibility of Data Lakes comes with a dangerous catch: without proper governance, they become **Data Swamps**.

Definition:

Data Swamp A **Data Swamp** is a Data Lake that has devolved into an unusable state due to lack of metadata management and governance. Nobody knows what data exists, where it is, or whether it can be trusted.

Example:

Warehouse vs. Junkyard Analogy **Well-governed Data Lake** = Amazon's fulfillment center

- Every item has a label and barcode
- Exact location is tracked in the system
- Can find anything in seconds

Data Swamp = Municipal junkyard

- Items dumped randomly
- No inventory or organization
- Something valuable might be there, but good luck finding it!

Caution

The Pendulum Metaphor

Think of data management as a pendulum swinging between two extremes:

$$\textbf{Data Silos} \longleftrightarrow \textbf{Data Swamps}$$
$$\text{(Too rigid, can't use data)} \longleftrightarrow \text{(Too loose, can't trust data)}$$

The goal is to find the **middle ground**: a **Governed Data Lake** that provides flexibility while maintaining quality and discoverability.

3.4 Data Warehouses: The Structured Alternative

Definition:

Data Warehouse (DW) A **Data Warehouse** is a system designed for Business Intelligence (BI) and reporting that stores only cleaned, structured, and processed data with strict schema enforcement.

Key Characteristics:

- **Schema on Write:** Data must conform to predefined structure
- **Primary Users:** Business Analysts, SQL users
- **Strengths:** Reliable data, very fast queries
- **Weaknesses:** Expensive, can't handle unstructured data, limited for ML use cases

3.5 Data Lake vs. Data Warehouse Comparison

Example:

Ferrari vs. Tractor Trailer Analogy **Data Warehouse (Ferrari):**
- Incredibly fast and sleek (high-performance queries)
- But can only carry 2 passengers (structured data only)
- No cargo space (limited flexibility)

Data Lake (Tractor Trailer):
- Not quite as fast as the Ferrari
- But can haul any type of cargo (all data types)
- Unlimited capacity (petabyte scale)

Characteristic	Data Lake	Data Warehouse
Data Types	All (structured, semi, unstructured)	Structured only
Data State	Raw and processed	Processed only
Schema	Schema on Read	Schema on Write
Process	ELT (Extract, Load → Transform)	ETL (Extract, Transform → Load)
Primary Users	Data Scientists, ML Engineers	Business Analysts, SQL users
Main Use Cases	AI/ML modeling, data exploration	BI reports, dashboards
Cost Model	Low (storage/compute separated)	High (storage/compute coupled)
Data Format	Open formats (Parquet, Delta)	Proprietary formats

3.6 Data Lakehouse: Best of Both Worlds

Historically, organizations had to maintain both a Data Warehouse (for BI) and a Data Lake (for AI/ML). This created:

- Data duplication between systems

- DW/DL silos (ironic, given lakes were meant to eliminate silos!)
- Double infrastructure costs
- Data synchronization nightmares

Definition:

Data Lakehouse A **Data Lakehouse** is a unified platform that combines the **low cost and flexibility** of Data Lakes with the **reliability, governance, and performance** of Data Warehouses.

How is this achieved? Through technologies like **Delta Lake** that add data warehouse capabilities on top of data lake storage.

Key Information

Why Lakehouses Matter:

- Single platform for both BI and AI workloads
- No data duplication or sync issues
- Reduced infrastructure costs
- Open formats prevent vendor lock-in
- ACID transactions ensure reliability

4 Delta Lake: The Foundation of Reliable Lakehouses

4.1 What is Delta Lake? (A Protocol, Not a Format)

Very Important:

Common Misconception **Delta Lake is NOT a file format!**
Delta Lake is a **storage protocol** or **management layer**. The actual data files are still stored as efficient **Parquet** files.
Delta Lake adds a **_delta_log** folder containing transaction logs on top of Parquet files, enabling powerful capabilities that raw Parquet cannot provide.

Why was Delta Lake needed?

Traditional Data Lake files (Parquet, CSV in Hadoop/S3) were immutable and had critical limitations:

- **No Updates/Deletes:** Modifying a single row required rewriting entire files
- **No Transaction Safety:** Failed jobs could corrupt data (duplicates, partial writes)
- **No Concurrency:** Multiple readers/writers caused inconsistency
- **No Quality Guarantees:** No schema enforcement after initial write

Delta Lake solves all these problems, making data lakes as reliable as databases.

Key Information

Delta Lake Alternatives:
Delta Lake is not the only solution. Similar technologies include:

- **Apache Iceberg** – Created by Netflix, now widely adopted
- **Apache Hudi** – Created by Uber

All three provide similar ACID guarantees. Delta and Iceberg are working on interoperability.

4.2 Delta Lake Core Capabilities

1. **ACID Transactions:** Like traditional databases, Delta Lake guarantees:
   - **Atomicity:** Operations either complete fully or not at all
   - **Consistency:** Data remains valid before and after transactions
   - **Isolation:** Concurrent operations don't interfere with each other
   - **Durability:** Committed changes persist
2. **Schema Management:**
   - **Schema Enforcement:** Rejects data that doesn't match table schema
   - **Schema Evolution:** Allows intentional schema changes (adding columns)
3. **Time Travel:** Every change is logged in _delta_log, enabling queries like:

```sql
-- Query data as it was at version 5
SELECT * FROM my_table VERSION AS OF 5;

-- Query data as it was at a specific time
```

```sql
SELECT * FROM my_table TIMESTAMP AS OF '2025-10-26 03:00:00';
```

4. **DML Operations:** Direct SQL operations on lake files:

```sql
UPDATE my_table SET column = 'value' WHERE condition;
DELETE FROM my_table WHERE condition;
MERGE INTO target USING source ON condition
  WHEN MATCHED THEN UPDATE ...
  WHEN NOT MATCHED THEN INSERT ...;
```

5. **Performance Optimization:**
   - **Data Skipping:** Statistics-based file filtering
   - **Z-Ordering:** Multi-dimensional data clustering
   - **OPTIMIZE:** Compacts small files into larger ones

5 Data Architecture Principles

5.1 Medallion Architecture

The Medallion Architecture is the most widely used pattern for organizing data in a Lakehouse. It separates data into three layers based on quality and refinement level.

Example:

Ore Refinery Analogy Think of data processing like refining ore:
Bronze (Raw Ore): Fresh from the mine, unprocessed. Contains impurities but preserves everything.
Silver (Refined Metal): Impurities removed, standardized form. Ready for manufacturing.
Gold (Finished Product): Shaped into specific products for end consumers.

Layer	Bronze	Silver	Gold
Data State	Raw (as received)	Cleaned, filtered	Aggregated, business-ready
Key Operations	- Ingest all source data- Preserve original format- Add ingestion metadata	- Remove nulls/duplicates- Standardize types- Join multiple sources- Convert to Delta format	- Business aggregations- KPI calculations- ML feature tables- Dashboard-ready views
Data Structure	Same as source system	3NF or similar	Denormalized, Star Schema
Primary Users	Data Engineers	Data Engineers, Scientists	Business Analysts, Scientists
Update Frequency	Real-time or batch	Batch (Bronze → Silver)	Batch (Silver → Gold)

Key Information

Why Medallion Architecture Works:

- **Reprocessing:** If transformation logic changes, reprocess from Bronze
- **Debugging:** Compare Bronze vs. Silver to trace data issues
- **Multiple Uses:** One Silver table can feed multiple Gold tables
- **Clear Ownership:** Each layer has defined responsibilities

5.2 Six Guiding Principles for Lakehouses

1. **Treat Data as Products:** Don't just accumulate data—curate it into trusted, well-documented "data products" that internal customers can rely on.
2. **Eliminate Data Silos, Minimize Data Movement:** Keep data in one place (the Lake) and let systems access it directly. Every copy creates sync issues and stale data risks.
3. **Democratize Value Creation Through Self-Service:** Enable business users to access and analyze data themselves, under proper governance. Don't bottleneck everything through the data team.
4. **Adopt Organization-Wide Data Governance:** Implement centralized access control, quality management, and cataloging (e.g., Unity Catalog). Without governance, you get a swamp.
5. **Use Open Interfaces and Formats:** Avoid vendor lock-in by using open formats like Parquet and Delta Lake. This preserves flexibility and enables ecosystem tools.
6. **Build for Scale, Optimize for Performance and Cost:** Design for growth from the start. Balance performance needs against cost constraints—don't over-provision, but don't bottleneck either.

5.3 Data Mesh vs. Data Fabric

Two concepts frequently discussed alongside Lakehouses are Data Mesh and Data Fabric. Despite similar-sounding names, they address different concerns.

Very Important:

Core Difference: Organization vs. Technology **Data Mesh:** An **organizational/cultural** approach. "Decentralize data ownership to domain teams who understand their data best."
Data Fabric: A **technological/architectural** approach. "Create a unified layer that connects data across all environments (cloud, on-prem, hybrid)."

Dimension	Data Mesh	Data Fabric
Core Idea	Decentralization	Integration and connection
Primary Focus	Organization, people, process	Technology, architecture, automation
Data Ownership	Domain teams (business units)	Central IT team (or delegated)
Governance Model	Federated (local rules under global standards)	Centralized (embedded via metadata)
Data Treatment	Data = Product (discoverable, trustworthy)	Data = Accessible Asset
Best Fit	Large enterprises with complex org structure	Enterprises with hybrid/multi-cloud complexity
Analogy	Federation of cities, each managing its own infrastructure	Highway system connecting all cities

Four Pillars of Data Mesh:

1. **Domain Ownership:** Each business domain owns and manages its data
2. **Data as a Product:** Data must be discoverable, addressable, and trustworthy
3. **Self-Service Infrastructure:** Teams can create resources on demand
4. **Federated Computational Governance:** Global standards, local implementation

6 Data Pipeline Operations and Debugging

6.1 Data Consolidation Tools

Getting data into the Data Lake requires ingestion tools. Here are the primary methods in Databricks:

1. **AutoLoader:**
   - Purpose: Continuous, incremental ingestion
   - Monitors cloud storage folders for new files
   - Automatically detects and processes new arrivals
   - Uses `cloudFiles` format (streaming-based)
   - Scales to billions of files
2. **COPY INTO:**
   - Purpose: One-time bulk batch ingestion
   - SQL command for loading data
   - **Idempotent:** Running multiple times doesn't create duplicates
   - Simpler than AutoLoader for single-load scenarios
3. **Delta Live Tables (DLT):**
   - Next-generation declarative ETL
   - Define "what" you want, system handles "how"
   - Built-in data quality constraints
   - Automatic error handling and recovery
4. **Local File Upload:**
   - Upload small files directly via Databricks UI
   - Limit: 10 files, 2GB total
   - Supports: CSV, TSV, JSON, Avro, Parquet, TXT, XML

6.2 Job Orchestration

Complex data pipelines involve multiple interconnected tasks. Managing these is called **Job Orchestration**.

Definition:

Workflow/Job A **Workflow** (or Job) is a collection of tasks with defined dependencies, schedules, and failure handling. Tasks can be notebooks, SQL scripts, Python files, or other executables.

Key Orchestration Concepts:

- **Dependencies (DAG):** Tasks form a Directed Acyclic Graph:
  - **Parallel:** Independent tasks run simultaneously
  - **Sequential:** Dependent tasks wait for predecessors
- **Scheduling:**
  - Cron expressions for time-based triggers ("every day at 3 AM")

  - Event triggers (“when new file arrives”)
  - Continuous mode (always running)
- **Failure Handling:**
  - Automatic retries for transient failures
  - Timeouts for hung jobs
  - Alert notifications to pipeline owners
- **Repair and Run:** If a 10-task pipeline fails at task 8, you can “repair” from task 8 onwards, reusing results from tasks 1-7. This saves time and compute costs.

6.3 The 4 S’s of Job Performance Issues

When Spark jobs run slowly, investigate these four common causes:

Example:

Understanding Performance Issues Through Analogies

1. Spill:

- **Symptom:** Memory (RAM) insufficient, data written to disk temporarily
- **Analogy:** Cooking on a tiny counter—you have to put ingredients on the floor and keep picking them up
- **Solution:** Use nodes with more memory, increase T-shirt size (serverless)

2. Shuffle:

- **Symptom:** Large data exchange between nodes during sorts or joins
- **Analogy:** Team project where everyone needs materials from each other, spending all time sending Slack messages
- **Solution:** Optimize partitioning strategy to minimize cross-node movement

3. Skew/Stragglers:

- **Symptom:** Data distributed unevenly; one node does most work while others idle
- **Analogy:** 5-person team project where 1 person gets 80% of the work—everyone waits for that one person to finish
- **Solution:** Change partition keys for even distribution (e.g., “country + city” instead of just “country”)

4. Small Files:

- **Symptom:** Millions of tiny files create more I/O overhead than actual processing
- **Analogy:** Reading a book split into 1 million single-page files—opening/closing files takes longer than reading
- **Solution:** Use OPTIMIZE command to compact small files into larger ones

6.4 Monitoring Approaches

Traditional Compute (Ganglia Metrics):

- Visual dashboards for CPU, memory, network usage

- Click on individual nodes to see detailed graphs
- Useful for diagnosing spill (high memory + disk I/O) or I/O-bound operations

Serverless:

- No Ganglia-level metrics available
- Uses T-shirt sizing (Small, Medium, Large, X-Large)
- Simplified management but less fine-grained tuning
- Check query history for statistics and performance data

7 Lab: Delta Lake Features

This section covers hands-on Delta Lake operations from the lab.

7.1 Converting Parquet to Delta

Converting existing Parquet data to Delta Lake format is straightforward:

```sql
-- Create a Delta table from existing Parquet data
CREATE TABLE IF NOT EXISTS loans_by_state_delta
USING delta
LOCATION '/path/to/delta/table'
AS SELECT * FROM parquet_table_view;

-- Verify the table format
DESCRIBE DETAIL loans_by_state_delta;
-- Result shows: format: delta
```

Listing 1: Creating a Delta table from Parquet data

7.2 Concurrent Streaming and Batch Operations

One of Delta Lake's powerful features is allowing simultaneous streaming reads and batch writes:

```python
# Start a streaming read from the Delta table
streaming_query = (
    spark.readStream
        .format("delta")
        .load("/path/to/delta/table")
        .writeStream
        .format("console")
        .start()
)
```

Listing 2: Reading Delta table as a stream

```sql
-- Insert batch data while streaming is running
INSERT INTO loans_by_state_delta VALUES ('IA', 450);
INSERT INTO loans_by_state_delta VALUES ('IA', 450);
-- ... repeat for more inserts
-- The streaming query will automatically pick up these changes
```

Listing 3: Batch inserts while streaming is active

7.3 DML Operations (DELETE, UPDATE)

Operations impossible on raw Parquet files work seamlessly on Delta tables:

```sql
-- Delete specific rows (impossible on plain Parquet!)
DELETE FROM loans_by_state_delta WHERE state = 'IA';
```

Listing 4: Row-level delete operation

7.4 MERGE (Upsert)

The MERGE command provides atomic "upsert" (update or insert) functionality:

```sql
MERGE INTO loans_by_state_delta AS target
USING new_updates_table AS source
ON target.state = source.state

WHEN MATCHED THEN
  -- State exists: update the count
  UPDATE SET target.count = source.new_count

WHEN NOT MATCHED THEN
  -- State doesn't exist: insert new row
  INSERT (state, count) VALUES (source.state, source.new_count);
```

Listing 5: MERGE operation for upserting data

7.5 Schema Evolution

Delta Lake enforces schema by default but allows intentional evolution:

```python
# Without mergeSchema, this fails if 'amount' column is new
df.write \
    .format("delta") \
    .mode("append") \
    .option("mergeSchema", "true") \  # Allow new columns
    .save("/path/to/delta/table")
```

Listing 6: Enabling schema evolution during write

7.6 Time Travel

Every change is recorded, enabling historical queries:

```sql
-- View all changes to the table
DESCRIBE HISTORY loans_by_state_delta;
-- Shows: version, timestamp, operation, operationParameters

-- Query specific version
SELECT * FROM loans_by_state_delta VERSION AS OF 0;
-- Returns data as it was initially created (Iowa missing)

SELECT * FROM loans_by_state_delta VERSION AS OF 9;
-- Returns data after all modifications (Iowa = 10)

-- Query by timestamp
SELECT * FROM loans_by_state_delta
TIMESTAMP AS OF '2025-10-26 03:00:00';
```

Listing 7: Viewing table history and time travel

8 Lab: Databricks Workflows

8.1 Creating a Job

Steps to create a workflow:

1. Navigate to **Jobs & Pipelines** in Databricks
2. Click **Create Job**
3. Add tasks (notebooks, SQL, Python files, etc.)
4. Configure dependencies between tasks
5. Set scheduling (cron, file arrival, continuous)
6. Configure alerts and retry policies

8.2 Task Dependencies

- Tasks with **no dependencies** run in parallel
- Tasks with **dependencies** wait for predecessors to complete
- Removing a dependency makes tasks parallel
- The visual DAG editor shows the execution flow

8.3 Parameters

Workflows support two types of parameters:

Job Parameters: Global to the entire workflow

```python
# Access job parameter in notebook
job_param = dbutils.widgets.get("job_param1")
```

Task Parameters: Specific to individual tasks

```python
# Access task parameter in notebook
task_param = dbutils.widgets.get("param1")
```

8.4 Scheduling Options

- **Scheduled:** Use cron syntax (e.g., "0 9 * * 1" = every Monday at 9 AM)
- **File Arrival:** Trigger on new files in a location
- **Continuous:** Always running (expensive, use for real-time needs)

9 Industry Adoption and Maturity

9.1 Lakehouse in the Gartner Hype Cycle

According to Gartner's analysis:

2021: Lakehouses were on the "Innovation Trigger" upslope—gaining attention but not yet proven.

2025: Lakehouses have passed through the "Trough of Disillusionment" and are climbing the "Slope of Enlightenment." Expected to reach the "Plateau of Productivity" within 2 years.

Key Information

MIT Report Finding:
Approximately 70% of CIOs surveyed reported that their organizations have adopted Lakehouse architecture. This indicates strong enterprise acceptance of the paradigm.

9.2 Why Lakehouses Are Succeeding

Before Lakehouses, organizations maintained separate platforms for:

- Streaming (Kafka, Flink)
- Batch processing (Spark)
- Machine Learning (separate ML platforms)
- BI/Analytics (Data Warehouse)
- Data Storage (Data Lake)

Keeping 4-5 systems synchronized, reconciled, and operational was a nightmare.

The Lakehouse Promise:

"Data gravity is the most important thing. Once you've massaged, cleansed, curated, and governed your data in one place—that's it. All use cases should run from there because specialized engines can access it."

Key enabling technologies:

1. **Delta Lake:** ACID transactions, time travel, DML operations
2. **Unity Catalog:** Centralized governance and metadata
3. **Photon Engine:** Rewritten Spark for warehouse-level performance
4. **Serverless Compute:** Simplified infrastructure management

10 One-Page Summary

Evolution of Data Storage

1. Data Silos (Problem): Isolated data islands. Departments can't share or integrate data.
↓
2. Data Lake (Solution): All data in one place, raw format, Schema on Read.
↓
3. Data Swamp (Risk): Lake without governance becomes unusable mess.
↓
4. Data Lakehouse (Evolution): Lake flexibility + Warehouse reliability in one platform.

Delta Lake: The Foundation

What it is: A protocol (not format!) adding transaction logs to Parquet files.
Key Capabilities:

- **ACID Transactions:** Reliable reads/writes
- **Time Travel:** Query historical data
- **DML Support:** UPDATE, DELETE, MERGE on lake files
- **Schema Evolution:** Controlled schema changes

Medallion Architecture

Bronze → Silver → Gold

- **Bronze:** Raw data as ingested (preserve everything)
- **Silver:** Cleaned, validated, joined (the work happens here)
- **Gold:** Aggregated, business-ready (what users consume)

The 4 S's of Performance Issues

- **Spill:** Memory overflow → disk usage (slow I/O)
- **Shuffle:** Cross-node data exchange (network bottleneck)
- **Skew:** Uneven data distribution → straggler nodes
- **Small Files:** Too many tiny files → I/O overhead

Data Mesh vs. Data Fabric

Data Mesh: Organizational culture. Decentralize data ownership to domain teams.
Data Fabric: Technical architecture. Unified access layer across all environments.
Both can be built on top of a Lakehouse platform!

CSCI E-103: Reproducible Machine Learning
Lecture 06: Business Intelligence and Data Visualization

Harvard Extension School

Fall 2025

■ **Course:**	CSCI E-103: Reproducible Machine Learning
■ **Week:**	Lecture 06
■ **Instructors:**	Anindita Mahapatra & Eric Gieseke
■ **Objective:**	Understand Business Intelligence (BI) fundamentals, the Lakehouse architecture for BI, and practical Databricks SQL features including AI functions, Lakeview dashboards, and Genie

Contents

1 Lecture Overview

Key Summary

This lecture focuses on Business Intelligence (BI) analytics and data visualization within the context of modern data platforms:

Core Topics:

- **Data Lake vs. Data Warehouse** – Understanding the trade-offs
- **Lakehouse Architecture** – Combining the best of both worlds
- **Business Intelligence (BI)** – Definition, purpose, and comparison with BA
- **BI Personas** – The BI Analyst and their primary skill (SQL)
- **Data Modeling for BI** – Star Schema and dimensional modeling
- **ETL vs. Data Federation** – When to move data vs. query in place
- **Databricks SQL Features** – Serverless, Materialized Views, Streaming Tables
- **AI Functions in SQL** – Embedding LLMs directly in queries
- **Lakeview Dashboards and Genie** – Self-service BI and natural language analytics

Example:

The Crystal Ball Analogy Think of BI as the business executive's "crystal ball." Businesses want to use data to clearly see what's happening today (insight) and, more importantly, predict what will happen tomorrow (foresight) to gain competitive advantage.

2 Key Terminology Reference

Term	**Description**	**Full Name**	**Notes**
BI	Technology to collect, analyze, and visualize data for better business decisions (What happened? How?)	Business Intelligence	Reports, dashboards
BA	Uses past data to explain present and predict future (Why? What's next?)	Business Analytics	Statistics, predictive modeling
Data Warehouse	Fast, expensive "data library" storing only structured data with predefined schema	Data Warehouse (DW)	Schema-on-Write
Data Lake	Cheap, massive "data garage" storing all data types in raw format	Data Lake (DL)	Schema-on-Read
Lakehouse	Unified architecture: DW performance/reliability on DL's cheap storage	Lakehouse	DW + DL combined
Medallion Arch.	3-tier data organization: Bronze (raw) → Silver (cleaned) → Gold (aggregated)	Medallion Architecture	
Data Federation	Querying remote data sources without physically moving data	Data Federation	No ownership
View	Virtual table defined by a query; executed on access (no storage)	View	
Materialized View	Pre-computed query results stored physically for fast access	Materialized View (MV)	
Concurrency	How many simultaneous queries/operations the system can handle	Concurrency	KPI for BI
Latency	Time from query request to result delivery (delay)	Latency	KPI for BI

3 Evolution of Data Storage

3.1 Data Warehouse vs. Data Lake

Example:

Library vs. Garage Analogy **Data Warehouse (DW) = Well-organized library**

- Only accepts "books" (structured data)
- Librarian immediately catalogs and shelves everything (Schema-on-Write)
- Finding a specific book (BI query) is very fast and accurate
- Cannot store videotapes or photos (unstructured data)
- Expensive to build and maintain

Data Lake (DL) = Garage that stores everything

- Accepts books, photos, videos, broken bicycles... everything (all data types)
- Just throw things in raw (Schema-on-Read)
- Storage is very cheap
- Finding something later takes time (slower queries)
- Without management, becomes a **Data Swamp**

Dimension	**Data Lake**		**Data Warehouse**	
	Pros	**Cons**	**Pros**	**Cons**
Storage	All file types (open format)	Lower data quality; file-level access	High reliability; fine-grained access	Structured only; proprietary formats
Compute	Very economical (storage/compute separated)	Operational complexity	Easy to use; high concurrency, low latency	Expensive to scale (coupled)
Consumption	Rich tool ecosystem (ML, AI, DS)	Not optimized for BI	SQL-optimized (BI)	Limited ML/streaming support

3.2 The Two-Tier Problem and Lakehouse Solution

Historically, organizations used both systems together:

1. **First Generation (DW Only):** Only structured data via ETL into DW. BI only.
2. **Second Generation (Two-tier: Lake + DW):** All data stored in DL. Then ETL again to copy BI-relevant data into a separate DW. ML/DS uses DL; BI uses DW.

Caution

Problems with Two-Tier Architecture:

- **Data Duplication:** Same data stored in both Lake and Warehouse, wasting cost
- **Increased Complexity:** Managing and synchronizing two separate systems
- **Data Freshness Issues:** ETL from Lake to DW takes time; BI users may not see latest data

Definition:

Lakehouse A **Lakehouse** is a unified architecture that provides Data Warehouse capabilities (ACID transactions, governance, fast queries) on top of Data Lake's cheap, flexible, open storage (S3, ADLS). Example: Databricks with Delta Lake.

Lakehouse Advantages:

- **Single System:** No data duplication or separate system management
- **All Workloads:** BI, reporting, Data Science, ML from the **same single data source**
- **Cost Efficiency:** Warehouse performance at Lake costs
- **Freshness:** Data in one place = single source of truth; BI always queries latest data

3.3 Historical Evolution of Data Storage

1. **Spreadsheets:** Most primitive (CSV files)
2. **Data Warehouses:**
   - Bill Inmon: ER model-based, normalized (3NF) central DW
   - Ralph Kimball: Business-user focused, denormalized dimensional models (Star/Snowflake)
3. **MPP Databases:** Teradata, Greenplum – distributed data and compute across nodes; TB-scale but expensive
4. **NoSQL / BigTable:** Google's PB-scale columnar storage
5. **Hadoop / Data Lakes:** Horizontal scaling on commodity hardware; separated storage from compute
6. **Data Mesh / Fabric:** Decentralized ownership (Mesh) and unified access layers (Fabric)
7. **Lakehouse:** Current architecture combining DL + DW capabilities

Key Information

The Journey Continues: Technology keeps evolving. AI and LLMs are already transforming the field further. What comes after Lakehouse? Hard to say, but change is constant.

4 Business Intelligence (BI) Fundamentals

4.1 What is Business Intelligence?

Definition:

Business Intelligence (BI) **Business Intelligence** encompasses all technologies, applications, and processes used to collect, integrate, analyze, and present business information to support **better business decision-making**.

Core Philosophy of BI

"Data is what you need to do analytics.
Information is what you need to do business."

BI transforms raw data into actionable **information** that business leaders can use.

BI Components:

- **Data Analysis:** Data exploration and querying
- **Visual Analytics:** Charts, graphs, dashboards
- **Advanced Analytics:** Predictions, statistics (overlaps with BA)
- **Data Governance:** Quality, security, access control
- **Strategy Documentation:** Business mission and strategy

4.2 BI vs. BA: What's the Difference?

Business Intelligence (BI)	**Business Analytics (BA)**
Uses **past and present** data	Uses **past** data
Focuses on **"What"** and **"How"** happened (Descriptive)	Explains **"Why"** and predicts **"What will happen"** (Explanatory, Predictive)
Example Questions: • "What was last quarter's revenue?" (What) • "Which product sold most?" (Who) • "When did sales peak?" (When)	**Example Questions:** • "Why did that product sell well?" (Why) • "Will this trend continue?" (Prediction) • "What if we raise prices 10%?" (What-if)
Key Techniques: Reporting, Dashboards, OLAP, Ad-hoc queries	**Key Techniques:** Statistical analysis, Data mining, Predictive modeling, A/B testing

Key Information

BI Trend: Modern BI is evolving from purely "descriptive" analysis (past reporting) toward "prescriptive" analysis (recommending what actions to take).

4.3 The BI Process (5 Steps)

1. **Collect:** Integrate data from source systems (CRM, ERP, etc.) into warehouse/lakehouse via ETL
2. **Organize:** Structure data in analysis-friendly models (OLAP cubes, Star Schema)
3. **Analyze:** BI analysts query data using **SQL**
4. **Visualize:** Present results as charts, dashboards, reports
5. **Decide:** Executives and teams use visualized information for strategic decisions

5 BI Personas and Data Modeling

5.1 The BI Analyst Persona

While many roles exist in the BI workflow (Data Engineers, Data Scientists), the core **consumer** of BI is the **BI Analyst**.

- **Primary Skill: SQL** The BI Analyst's main tool is **SQL**. They use it to explore data, answer business questions, and extract data for dashboards.
- **Data Consumed: Curated Data** BI Analysts don't work with raw Bronze data. They primarily use data that Data Engineers have cleaned (Silver) and aggregated for business use (Gold).
- **Role: Analytics Engineering** Increasingly, BI Analysts are expected to model data and curate Gold tables themselves using SQL—this is called "Analytics Engineering."

5.2 Data Modeling for BI: Star Schema

BI queries must be very fast (low latency), so data must be structured optimally. The most popular approach is **Dimensional Modeling**, specifically the **Star Schema**.

Definition:

Star Schema A **Star Schema** looks like a "star":

- **Fact Table (Center):** Contains business event measurements (numeric data): sales_amount, quantity_sold
- **Dimension Tables (Points):** Surround the fact table, providing context: dim_customer, dim_product, dim_time

Example:

Star Schema for Online Store Sales

- **Fact_Sales:** {date_key, product_key, customer_key, sales_amount, quantity}
- **Dim_Time:** {date_key, date, month, year, quarter, day_of_week}
- **Dim_Product:** {product_key, product_name, category, brand}
- **Dim_Customer:** {customer_key, customer_name, city, country}

Query: "Q1 2025 sales by category for Seoul customers?" — Fast, simple JOINs!

Snowflake Schema: A variation where dimension tables are further normalized and linked to additional tables.

Caution

Data Vault Model: Data Vault uses Hubs (core business keys), Links (relationships), and Satellites (descriptive attributes). It's flexible for the Silver layer, but the **Gold layer for BI typically still uses Star Schema** for query performance.

6 ETL vs. Data Federation

Definition:

Data Federation **Data Federation** queries external data sources (Oracle, Redshift, Snowflake) **without physically moving the data**. It's **read-only** access where you don't "own" the data.

Very Important:

Key Difference: Ownership **ETL:** You extract, transform, and load data into your lakehouse. The lakehouse **owns** that data. Significant compute is spent curating through Bronze $\rightarrow$ Silver $\rightarrow$ Gold. **Federation:** You query data where it lives (external system). **No ownership**. Good for modest, reference/lookup data. Query pushdown translates your SQL to the remote system's native query.

When to Use Each:

- **Federation:** Small data volumes, reference/lookup data, one-off joins
- **ETL:** Large data volumes, performance-critical queries, when you need low latency

Example:

Federation Use Case Your core transactional data is in the lakehouse, but you need to join with a reference table from an on-prem Oracle system. Rather than ETL the entire Oracle table, you federate: write one SQL query that joins your lakehouse table with the Oracle table. The system translates the WHERE clause and does a "push down" to Oracle, bringing back only the needed rows.

7 Databricks SQL for BI

Databricks provides comprehensive BI and data warehousing capabilities through **Databricks SQL (DBSQL)**.

7.1 Platform Architecture

Source: All data types (structured, unstructured, streaming)

Ingest: ETL (data moves in) or Federation (query in place)

Transform: Medallion Architecture (Bronze → Silver → Gold)

Query and Process: DBSQL for BI; Spark/ML for Data Science

Governance: Unity Catalog for access control, lineage, audit

Engine: Photon (C++ rewrite of Spark for vectorized performance)

Serve/Analysis: Lakeview Dashboards, BI tool integrations, Lakehouse Apps

7.2 Key BI Features in DBSQL

1. Serverless Compute:

- Instant compute allocation—no waiting 3-6 minutes for VMs to spin up
- Auto-scaling based on workload
- Auto-termination after inactivity (saves cost)

2. Streaming Tables:

```sql
CREATE STREAMING TABLE my_streaming_table
AS SELECT * FROM cloud_files('/path/to/data', 'json');
```

Listing 1: Creating a Streaming Table from Cloud Storage

No complex code—just SQL to process streaming data automatically.

3. Materialized Views (MV):

```sql
CREATE MATERIALIZED VIEW revenue_by_route AS
SELECT route_id, SUM(fare_amount) as total_revenue
FROM trips
GROUP BY route_id;
```

Listing 2: Creating a Materialized View

Pre-computed results stored physically. Dashboards query MVs for instant response. MVs auto-update incrementally when source data changes.

4. Concurrency and Scaling:

- **Concurrency:** How many simultaneous queries can run without slowdown
- **Scaling:** Can the system handle growing data volumes?
- Ideal: Queries execute (green) without being queued (yellow)

5. Additional Features:

- SQL Editor with intelligent autocomplete
- Parameterized queries (different users see different data based on parameters)
- Query history and profiling
- Row-level security and column masking
- Geospatial support (H3)
- Workflow integration (dashboards as workflow tasks)

8 AI Functions in SQL

Databricks SQL allows embedding LLMs directly within SQL queries—a revolutionary capability.

8.1 ai_query(): External LLM Calls

```sql
SELECT
  sku_id,
  product_name,
  ai_query(
    "my-openai-endpoint",  -- Pre-registered model endpoint
    "You are a marketing expert. Generate a 30-word
      promotional text for product: " || product_name
  ) AS promotional_text
FROM retail_products;
```

Listing 3: Using ai_query() for Product Marketing

This embeds an LLM call within your SELECT statement, enriching your data with AI-generated content.

8.2 Built-in AI Functions

Databricks provides pre-built AI functions for common tasks—no external model setup needed:

```sql
-- Sentiment Analysis
SELECT ai_analyze_sentiment('I am happy');
-- Returns: 'positive'

-- Classification
SELECT ai_classify('My password is leaked',
                   ARRAY('urgent', 'not urgent'));
-- Returns: 'urgent'

-- Information Extraction
SELECT ai_extract('John Doe lives in New York',
                  ARRAY('person', 'location'));
-- Returns: {"person": "John Doe", "location": "New York"}

-- Grammar Correction
SELECT ai_fix_grammar('This sentence have some mistake');
-- Returns: 'This sentence has some mistakes'

-- Sensitive Data Masking
SELECT ai_mask('My email is john@example.com', ARRAY('email'));
-- Returns: 'My email is [MASKED]'
```

Listing 4: Built-in AI Functions

There are approximately 15-20 built-in AI functions covering sentiment, classification, extraction, summarization, masking, similarity, and more.

9 Lakeview Dashboards and Genie

9.1 Lakeview Dashboards

Definition:

Lakeview Databricks' built-in dashboard tool for data visualization directly within the platform.

Components:

- **Data Tab:** Define datasets that power the dashboard
- **Visualization Widgets:** Charts, graphs (bar, scatter, line, etc.)
- **Text Boxes:** Documentation and labels
- **Filters:** Interactive controls (dropdowns, date pickers)

Natural Language Creation: The built-in **Assistant** lets you create charts using natural language:

> "Show me number of trips by pickup zip code"

The assistant generates the visualization, which you can then customize.

Caution

When to Use Lakeview vs. PowerBI/Tableau:

- **Lakeview:** No additional licensing cost. Quick, operational dashboards for data engineers and analysts.
- **PowerBI/Tableau:** Sophisticated, polished dashboards for C-level executives with specific branding requirements.

9.2 Publishing and Sharing

Share (Internal): Grant view/edit access to other Databricks users in your organization.

Publish (External): Share with users who **don't have** Databricks accounts.

- Use "Embed credentials" option
- **Cost Warning:** When external users view a published dashboard and results aren't cached, compute costs are charged to the **publisher's account**
- Results are cached for 24 hours by default

9.3 Databricks Genie: Conversational AI Analytics

Definition:

Genie **Genie** is a conversational AI assistant available on published dashboards that allows users to ask questions in natural language.

Use Case Scenario: A marketing manager is viewing a dashboard created by the BI team. They wonder: "What if I filter to only male customers in their 20s?"

Previously, this would require a request to the BI team and weeks of waiting. Now, they simply ask Genie in natural language and get an instant answer.

How It Works:

1. **User Question:** "What is the average trip duration?"
2. **Genie Response:** "The average is 13.7 minutes."
3. **Transparency:** Genie shows the SQL query it generated:

```sql
SELECT AVG(dropoff_time - pickup_time)
FROM trips
WHERE pickup_time IS NOT NULL
  AND dropoff_time IS NOT NULL;
```

Very Important:

Why Genie Doesn't Hallucinate Unlike ChatGPT which tries to please users by answering any question (potentially making things up), Genie's scope is **strictly limited to the metadata of the specific tables** defined in the dashboard.
If you ask "What's the weather like?"—Genie will respond: "Sorry, I can only answer questions about the data available."
Genie generates **pure SQL** based on table schemas. No LLM creativity involved in the answer—just SQL execution.

Customizing Genie: You can provide instructions and example queries to improve Genie's understanding:

- "By trip duration, I mean duration in seconds, not minutes"
- Pre-written SQL for complex calculations

10 Databricks Platform Integration

10.1 SQL Warehouse Connectivity

Databricks SQL Warehouses can connect to many external tools:

- **BI Tools:** Tableau, PowerBI, Looker
- **Development:** dbt, Python, Java, NodeJS, Go
- **JDBC/ODBC:** Standard database connectivity

Connection details are available in the warehouse settings, including JDBC URLs.

10.2 Monitoring and Query History

Query History: Every query is logged with:

- Who ran it
- Duration (execution time vs. queue time)
- Status (success/failed)
- Query profile (memory usage, rows returned, scan details)

Warehouse Monitoring:

- Blue bars = queries running
- Orange bars = queries queued
- Ideal: mostly blue, minimal orange

Administrators use this to:

- Identify heavy users
- Debug failed queries
- Plan chargeback policies
- Optimize performance

10.3 Workflow Integration

Dashboards can be tasks in workflows:

- As new data arrives via pipeline, the dashboard auto-refreshes
- Combine Data Engineering, ML, and BI tasks in a single workflow
- Set up alerts if data volume drops (pipeline health monitoring)

11 One-Page Summary

Storage Comparison: Warehouse vs. Lake

- **Warehouse (DW):** Strict library (structured data, Schema-on-Write, BI/SQL optimized, high cost, high performance)
- **Lake (DL):** Everything garage (all data types, Schema-on-Read, ML/DS optimized, low cost, lower performance)

Lakehouse: Unified Architecture

Lake's cheap storage + Warehouse's performance/reliability/governance

- Single system for both BI and ML/DS workloads
- Eliminates data duplication and ETL complexity

BI vs. BA: Different Questions

- **BI:** "What happened?" (past/present reporting, dashboards)
- **BA:** "Why did it happen? What will happen?" (statistics, predictive modeling)

BI Analyst and Data Modeling

- **BI Persona:** Core skill is **SQL**
- **Data Modeling: Star Schema** (central Fact table + surrounding Dimension tables) is most efficient for BI queries (Kimball approach)

Databricks SQL Key Features

- **Serverless Compute:** No cluster boot wait time
- **Materialized Views:** Pre-computed queries for fast dashboards
- **Streaming Tables:** SQL-only streaming data processing
- **AI Functions:** LLMs embedded directly in SQL

SQL + AI: Intelligent Queries

- **ai_query():** Call external LLMs (GPT, etc.) from SQL
- **ai_analyze_sentiment(), ai_classify()...:** Built-in AI functions for instant sentiment, classification, extraction

Dashboards and AI Assistant: Lakeview and Genie

- **Lakeview:** Built-in Databricks dashboards (operational, free)
- **Genie:** Conversational AI on published dashboards. Natural language $\rightarrow$ SQL
- **Genie's Strength:** No hallucination—only references specified table metadata

CSCI E-103: Reproducible Machine Learning
Lecture 07: Operationalizing Data Pipelines

Harvard Extension School

Fall 2025

■ **Course:**	CSCI E-103: Reproducible Machine Learning
■ **Week:**	Lecture 07
■ **Instructors:**	Anindita Mahapatra & Eric Gieseke
■ **Objective:**	Learn to transform prototype notebooks into production-ready automated data pipelines with proper requirements, quality assurance, governance, and declarative frameworks like Delta Live Tables (DLT)

Contents

1 Lecture Overview

Key Summary

This lecture covers the critical process of **operationalizing data pipelines**—transforming a data scientist's proof-of-concept notebook into a reliable, automated production system.

Core Topics:

- **Requirements Definition** – Functional vs. Non-functional requirements
- **Six Guiding Principles** – Best practices for production pipelines
- **Data Quality and Validation** – Handling corrupt records, missing values
- **Data Governance** – Access control, lineage, cataloging
- **Scalability and Elasticity** – Horizontal vs. vertical scaling
- **Lakeflow and Delta Live Tables (DLT)** – Declarative pipeline frameworks
- **CDC Automation** – `apply_changes()` for Insert/Update/Delete
- **Data Quality Expectations** – Built-in validation with DLT

Example:

Kitchen Recipe vs. Factory Production **PoC (Notebook):** A chef's experimental recipe in a kitchen. Tastes great, but makes only one serving at a time. Quality varies with the chef's mood.
Production (Operationalized): The same recipe adapted for a large food factory—producing thousands of units daily with consistent quality, automated processes, and quality control checkpoints.

2 Key Terminology Reference

Term	**Full Name**	**Description**
Operationalizing	-	Transforming a prototype into a production-ready automated system
PoC	Proof of Concept	Initial model validating technical feasibility (e.g., Jupyter notebook)
Functional Req.	Functional Requirements	Defines **what** the system should do (features)
Non-Functional Req.	Non-Functional Requirements	Defines **how** the system should perform (quality, performance)
Medallion Arch.	Medallion Architecture	3-tier data organization: Bronze → Silver → Gold
Data Governance	-	Policies managing data quality, security, and accessibility
Data Lineage	-	Tracking where data came from and how it was transformed
ETL / ELT	Extract, Transform, Load	Data processing order. ETL: traditional. ELT: modern lakehouse
Lakeflow	-	Databricks' unified pipeline tool (ingest, transform, orchestrate)
DLT	Delta Live Tables	Declarative ETL framework that simplifies complex pipeline building
Declarative Pipeline	-	Define "what" you want; engine handles "how"
Autoloader	-	Automatic detection and processing of new files in cloud storage
CDC	Change Data Capture	Identifying and tracking changes (Insert, Update, Delete) in source data
Scalability	-	Ability to increase system capacity for growing workloads
Elasticity	-	Ability to dynamically scale resources up/down based on demand
Checkpoint	-	"Bookmark" recording where streaming processing left off

3 Why “Operationalize” Pipelines?

A data scientist’s prototype notebook demonstrates that a problem *can* be solved, but it’s not suitable for daily business use. Production pipelines must have:

- **Reliability:** Consistent, correct results every time
- **Automation:** No manual intervention required
- **Scalability:** Handles growing data volumes
- **Monitoring:** Alerts when something goes wrong
- **Governance:** Proper access control and data quality

The goal is to take a “raw notebook” and turn it into a “reliable automated data pipeline that continuously produces trusted data for downstream consumers” (business users, data scientists, ML models).

4 Requirements: The Foundation of Design

Before designing anything, you must clearly understand what the business wants and expects. Requirements are divided into two categories:

4.1 Functional Requirements ("What")

Definition:

Functional Requirements Specify the **specific functions** the system must provide to users.

Examples:

- Business rules (e.g., cancel transactions under certain conditions)
- Authentication and authorization levels
- External interfaces (connect to multiple data sources)
- Reporting requirements (daily sales summary)
- Audit tracking (who modified what data)
- Historical data access requirements

4.2 Non-Functional Requirements ("How")

Definition:

Non-Functional Requirements Specify the **quality, performance, and constraints** of the system. Often overlooked but equally critical.

Examples:

- **Performance:** Latency (response time), Throughput (queries per second)
- **Scalability:** Handle 10x data growth
- **Availability:** System uptime percentage (e.g., 99.99% = "Four Nines")
- **Reliability:** Correct behavior, error recovery
- **Security:** Access control, compliance (GDPR, HIPAA)
- **Maintainability:** Ease of updates and modifications
- **Cost:** Budget constraints for infrastructure

Caution

The Requirements Trap:

- **Over-design:** Building a sub-second real-time system when the business is fine with daily batch updates wastes enormous resources.
- **Under-design:** Delivering a 10-second response when customers expect under 1 second makes the system useless.

Example: "Five Nines" (99.999%) availability allows only **5 minutes of downtime per year**—

extremely expensive to achieve. Always ask: *Is this actually required?*

5 Six Guiding Principles for Production Pipelines

These principles help build efficient, robust data lakehouses:

5.1 1. Curate Data and Offer Trusted Data as Products

Think of your pipeline output as a **product** that internal consumers (business users, data scientists) can trust and use.

Key Information

Medallion Architecture:

- **Bronze (Raw):** Source data stored unchanged (ore from the mine)
- **Silver (Curated):** Cleaned, filtered, enriched data (refined metal)
- **Gold (Final):** Aggregated, business-ready data (finished jewelry)

Quality and trust **increase** as data moves through layers; processing **cost** also increases.

5.2 2. Remove Data Silos and Minimize Data Movement

Moving and copying data creates cost, latency, quality issues, and isolated "silos."

Solutions:

- **Shallow Clones:** Copy metadata only, not actual data
- **Views:** Virtual tables that query underlying data
- **Delta Time Travel:** Query historical versions without separate backups
- **Data Sharing:** Share datasets directly rather than copying

5.3 3. Democratize Access Through Self-Service

Enable business users to explore and use data themselves, not just the data team.

Critical Prerequisite: Proper **guardrails** (governance) must prevent unauthorized changes or access to sensitive data.

5.4 4. Adopt Organization-Wide Data Governance

Governance isn't just security—it's a comprehensive system managing data's entire lifecycle:

- **Data Quality:** Constraints ensuring accuracy and consistency
- **Data Catalog & Lineage:** Metadata definitions, tracking data origin and transformations
- **Access Control:** Who can access what data (row/column level, PII masking)
- **Audit Logs:** Recording who accessed or modified data

5.5 5. Use Open Interfaces and Formats

Use open formats (Delta, Parquet) instead of proprietary vendor-locked formats:

- **No Vendor Lock-in:** Freely migrate to other platforms
- **Interoperability:** Easy integration with third-party tools
- **Lower Cost:** Avoid expensive proprietary licenses

5.6 6. Build to Scale and Optimize for Performance and Cost

Very Important:

Decoupling Storage and Compute The most critical architectural principle:
Traditional (Coupled): Storage and CPU bound together on one server. If data grows 10x, you must buy 10x more expensive CPUs too. (e.g., ElasticSearch)
Modern Lakehouse (Decoupled): Store data in cheap cloud storage (S3, ADLS) infinitely. Only "rent" compute clusters when processing is needed.

Example:

Warehouse vs. Factory Analogy

- **Storage:** A **warehouse** for storing goods. Cheap to expand infinitely.
- **Compute: Factory machines** for processing goods. Expensive, but you only pay when machines are running.

Separating them means you pay only for what you use, when you use it.

6 Data Quality and Validation

"Garbage In, Garbage Out (GIGO)"—The entire value of a data pipeline depends on data quality.

6.1 Dimensions of Data Quality

- **Accuracy:** Does the data match reality?
- **Consistency:** Is data coherent across the system? (e.g., "NY" vs "New York")
- **Completeness:** Is required data missing?
- **Timeliness:** Is data fresh enough for the use case?
- **Integrity:** Are relationships (foreign keys) correct?

6.2 Handling Corrupt Records in Spark

Source data can be "corrupted" due to schema mismatches, format errors, or missing values. Spark provides three modes:

Caution

Spark's Three Parse Modes:

1. PERMISSIVE (Default):

- Stores bad records in a `_corrupt_record` column
- Fills parseable columns with `null`
- Pipeline continues—analyze bad data later

2. DROPMALFORMED:

- Silently drops (ignores) corrupt records
- Use when some data loss is acceptable (e.g., log analysis)

3. FAILFAST:

- Immediately stops and throws an exception on any corrupt record
- Use for critical data where no errors are tolerable (e.g., financial transactions)

Additionally, use `option("badRecordsPath", "path")` to save corrupt records to a separate location for later analysis.

6.3 Handling Missing Values and Duplicates

Duplicates:

```
df.dropDuplicates(["id", "color"])  # Remove duplicates based on keys
```

Missing Values:

- **Drop:** `df.dropna()` — Remove rows with missing values (risk: data loss)
- **Placeholder:** Fill with `-1` or `"N/A"`
- **Basic Imputing:** `df.na.fill({"temperature": 68, "wind": 6})`
- **Advanced Imputing:** Use ML models to predict missing values

7 Scalability, Elasticity, and Availability

7.1 Scalability: Handling Growing Workloads

Definition:

Scalability A system is **scalable** if adding resources proportionally increases its capacity.

Two Types:

- **Vertical Scaling (Scale Up):** Upgrade to a bigger, more powerful machine
- **Horizontal Scaling (Scale Out):** Add more machines (preferred for big data)

7.2 Elasticity: Dynamic Resource Adjustment

Definition:

Elasticity The ability to **automatically** scale resources up or down based on current demand.

Databricks Serverless is a perfect example: you pay nothing when not using compute, and it automatically scales up during peak processing, then scales down when done.

7.3 Reliability and Availability

- **Reliability:** Is the system producing correct results?
- **Availability:** Is the system accessible when users need it?

Availability is often measured in "nines":

- 99% ("Two Nines") = 3.65 days downtime/year
- 99.9% ("Three Nines") = 8.76 hours downtime/year
- 99.99% ("Four Nines") = 52.6 minutes downtime/year
- 99.999% ("Five Nines") = 5.26 minutes downtime/year

Key Information

Always clarify availability requirements with the business. Five Nines is extremely expensive to achieve and may not be necessary for all systems.

8 Lakeflow and Delta Live Tables (DLT)

Lakeflow is Databricks' unified platform for data ingestion, transformation, and orchestration. **Delta Live Tables (DLT)** is the core feature enabling **declarative pipelines**.

8.1 Imperative vs. Declarative Pipelines

Example:

Ordering at a Restaurant **Imperative (Traditional):** Tell the chef step-by-step: "1. Grill 200g beef, 2. Wash vegetables, 3. Warm the bread..." You must handle every detail including errors, retries, and scaling.
Declarative (DLT): Order from the menu: "One steak, please." The kitchen (DLT engine) handles recipes, cooking, quality checks, and serving automatically.

With DLT, developers focus on **business logic** (transformations), while the engine handles:

- Error handling and retries
- Scaling and optimization
- State management (checkpoints)
- Dependency management (DAG)
- Monitoring and observability

8.2 DLT Building Blocks

1. Streaming Tables:

- Process data incrementally (once per record)
- Maintain state (know what's already processed)
- Handle Bronze $\rightarrow$ Silver transformations

2. Materialized Views:

- Pre-computed aggregations for Gold layer
- Automatically refreshed incrementally
- Great for dashboards and reporting

```sql
-- Streaming Table from Autoloader
CREATE STREAMING TABLE bronze_transactions
AS SELECT * FROM cloud_files("/path/data", "json");

-- Streaming Table from Kafka
CREATE STREAMING TABLE kafka_events
AS SELECT * FROM read_kafka(brokers => "host:9092");

-- Materialized View (auto-refreshed)
CREATE MATERIALIZED VIEW gold_summary AS
SELECT category, SUM(amount) as total
```

```sql
FROM silver_transactions
GROUP BY category;
```

Listing 1: DLT SQL Examples

8.3 CDC Automation with apply_changes()

The most powerful DLT feature: automatically handling Insert, Update, Delete operations from source data.

```python
import dlt
from pyspark.sql.functions import col, expr

# 1. Bronze: Ingest raw data with Autoloader
@dlt.table(comment="Raw transactions from Autoloader")
def bronze_transactions():
    return (
        spark.readStream
            .format("cloudFiles")
            .option("cloudFiles.format", "json")
            .load("/path/to/source")
    )

# 2. View: Clean data before applying to Silver
@dlt.view(comment="Cleaned transactions")
def clean_transactions():
    return (
        dlt.read_stream("bronze_transactions")
            .selectExpr(
                "transaction_id",
                "CAST(amount AS DOUBLE)",
                "operation_type",  # 'INSERT', 'UPDATE', 'DELETE'
                "CAST(update_timestamp AS LONG) as sequence"
            )
    )

# 3. Silver: Apply CDC changes
@dlt.table(comment="CDC applied Silver table")
def silver_transactions():
    dlt.apply_changes(
        target = "silver_transactions",
        source = dlt.read_stream("clean_transactions"),
        keys = ["transaction_id"],         # Primary key
        sequence_by = col("sequence"),     # Ordering column
        apply_as_deletes = expr("operation_type = 'DELETE'")
    )
```

Listing 2: DLT CDC Pipeline Example

Key Information

apply_changes() Parameters:

- **target:** Final destination table
- **source:** Streaming DataFrame with changes
- **keys:** Primary key columns for record identification
- **sequence_by:** Column determining change order (latest wins)
- **apply_as_deletes:** Condition for DELETE operations

9 Data Quality with DLT Expectations

DLT allows declaring data quality rules directly in pipeline definitions:

```python
import dlt

# Drop rows that fail this expectation
@dlt.expect_or_drop("valid_age", "age > 0 AND age < 120")

# Fail entire pipeline if this expectation fails
@dlt.expect_or_fail("valid_email", "email IS NOT NULL")

# Log but keep rows that fail (for monitoring)
@dlt.expect("reasonable_score", "score >= 50")

@dlt.table(comment="Quality-controlled Silver table")
def users_silver():
    return dlt.read_stream("users_bronze")
```

Listing 3: DLT Expectations Example

Three Behaviors on Violation:

- **expect_or_fail:** Stop pipeline immediately (like FAILFAST)
- **expect_or_drop:** Drop the violating row (like DROPMALFORMED)
- **expect:** Keep the row but log the violation for monitoring

Key Information

Quarantining Pattern: Instead of dropping or failing, route bad data to a separate "quarantine table." The main pipeline continues, while bad records are reviewed, fixed, and re-injected later.

9.1 Schema Evolution

DLT provides options for handling schema changes:

- **Enforce Schema:** Fail if schema doesn't match (strict)
- **Merge Schema:** Automatically add new columns
- **Evolution Strategies:** Various options for handling schema drift

10 Pipeline Execution and Monitoring

10.1 Running DLT Pipelines

Execution Modes:

- **Triggered:** Run once when manually started or scheduled
- **Continuous:** Always running, processing data as it arrives

Refresh Options:

- **Incremental:** Process only new/changed data since last run
- **Full Refresh:** Reprocess everything from scratch (resets state)
- **Selective Refresh:** Refresh only specific tables

10.2 Observability

DLT provides comprehensive monitoring:

- **Event Logs:** Every action recorded to Delta tables
- **Query History:** See actual queries executed by the engine
- **Query Profiles:** Memory usage, rows processed, performance metrics
- **DAG Visualization:** Visual representation of pipeline dependencies

10.3 Checkpoint Management

Caution

Common Streaming Error: Checkpoint Conflicts

When running structured streaming (not DLT), you may encounter checkpoint errors when:

- Running the same query after code changes
- Source data was deleted but checkpoint still exists

Manual Solution:

```
checkpoint_path = "/path/to/checkpoint"
dbutils.fs.rm(checkpoint_path, recurse=True)  # Delete old checkpoint
# Then restart your streaming query
```

Better Solution: Use DLT, which manages checkpoints automatically. “Full Refresh” resets all state without manual intervention.

11 One-Page Summary

PoC vs. Production

PoC (Prototype): Manual, unstable, no quality guarantees, one-time analysis
↓ **Operationalizing** ↓
Production: Automated, reliable, quality guaranteed, scalable, monitored

Requirements

- **Functional:** What? (features, functions)
- **Non-Functional:** How? (performance, availability, cost) ← Often overlooked!

Medallion Architecture

Bronze (Raw) → Silver (Cleaned/Validated) → Gold (Aggregated/Final Product)

Six Guiding Principles

1. Data as Products (curated, trusted)
2. Remove Silos (minimize data movement)
3. Self-Service + Guardrails (governance)
4. Organization-Wide Governance
5. Open Formats (no vendor lock-in)
6. Scale & Optimize (decouple storage/compute)

DLT Core Features

- **Declarative:** Define "What", engine handles "How"
- **CDC Automation:** `dlt.apply_changes()`
- **Quality Automation:** `@dlt.expect_...`
- **State Management:** Checkpoints, Full Refresh handled automatically

Scaling

- **Horizontal (Scale Out):** Add more machines (preferred)
- **Vertical (Scale Up):** Upgrade to bigger machines
- **Critical:** Always decouple Storage (cheap) from Compute (expensive)!

Data Quality Modes

- **PERMISSIVE:** Keep bad data in _corrupt_record column
- **DROPMALFORMED:** Silently drop bad records
- **FAILFAST:** Stop immediately on any error

Lecture 08: Reproducible Machine Learning

CSCI E-103: Reproducible Data Science and Machine Learning

Harvard University

■ **Course:**	CSCI E-103: Reproducible Data Science
■ **Week:**	Lecture 08
■ **Instructors:**	Anindita Mahapatra & Eric Gieseke
■ **Objective:**	Understand how to achieve reproducibility in ML projects through MLOps culture and MLflow tools

Contents

1 The Core Problem: Why Is ML Reproducibility So Hard?

Lecture Overview

This lecture addresses a fundamental challenge in machine learning: **Reproducibility**—the ability to recreate the exact same model and results given the same data and code. While this sounds simple, it's extraordinarily difficult in practice.

Key Topics:

- The "Hidden Technical Debt" in ML systems
- ML lifecycle and the roles of Data Engineer, Data Scientist, and ML Engineer
- Feature Engineering and Feature Stores
- ML Pipelines: Transformers vs. Estimators
- Model Drift and its four types
- MLOps culture and MLflow as the solution

1.1 The Iceberg Metaphor: Hidden Technical Debt

One of the most important insights in ML engineering comes from Google's famous paper on "Hidden Technical Debt in Machine Learning Systems." The key revelation is:

The Misconception vs. Reality

Common Misconception: "Machine Learning is about writing sophisticated algorithms and model code."

Reality: The actual ML code is just a **tiny fraction** of a real-world ML system. It's like the tip of an iceberg—what you see above water is tiny compared to the massive infrastructure lurking beneath.

The Hidden Infrastructure (90%+ of the system):

- **Data Collection**: Gathering data from multiple sources
- **Data Verification**: Ensuring data quality and integrity
- **Configuration**: Managing hyperparameters, feature flags, data versions
- **Feature Extraction**: Engineering meaningful inputs for models
- **Infrastructure**: Managing compute resources (CPU, GPU, clusters)
- **Monitoring**: Tracking model performance and data drift
- **Serving**: Deploying models to production systems

Example: Analogy: Building a House

Think of an ML system like building a house:

- **ML Code** = The visible part of the house (walls, roof, windows)
- **Infrastructure** = Everything you don't see (foundation, plumbing, electrical, HVAC)

You can have a beautiful house, but if the foundation is weak or the plumbing is broken, the house is unusable. Similarly, you can have a brilliant ML algorithm, but without solid data pipelines, monitoring, and serving infrastructure, it's worthless in production.

Why does this matter for reproducibility?

To reproduce a model, you don't just need the ML code—you need to recreate the **entire environment**: the exact data version, the library versions, the configuration settings, the feature engineering logic, and more. This is why reproducibility is so challenging.

1.2 The Fragmented Ecosystem Problem

ML projects don't use a single tool. They involve a complex mix of:

- **Languages**: Python, R, SQL, Scala, Java
- **Frameworks**: Scikit-learn, PyTorch, TensorFlow, XGBoost, Spark MLlib
- **Deployment Environments**: Docker, Kubernetes, AWS SageMaker, Databricks

Dependency Hell

"It worked on my laptop!"
This is the most common phrase in ML engineering. When you manually connect different tools, you enter what's called **dependency hell**:

- You trained a model with `pandas==1.3.0` but production has `pandas==1.5.0`
- Your colleague uses `numpy==1.21` while you have `numpy==1.23`
- The server has Python 3.8 but you developed on Python 3.10

Even tiny version differences can cause different results or outright failures.

1.3 Models Are Living Assets

Perhaps the most counterintuitive aspect of ML: **models are not static artifacts**. They're "living" assets that degrade over time.

Definition: Model Drift

Model Drift is the phenomenon where a deployed model's performance degrades over time because the real-world data patterns have changed since the model was trained.
Think of it like a driver's license: passing the driving test (training) doesn't mean you can handle every situation on real roads (production), especially when traffic laws change or new types of intersections appear.

This means that even if you perfectly reproduce a 6-month-old model, it might be **useless** because the world has changed. What truly matters is:

1. **Tracking** exactly how that model was built (so you can understand what worked)
2. Having an **automated pipeline** to quickly retrain on fresh data

This is precisely what MLOps and MLflow solve.

2 The ML Lifecycle and Key Roles

2.1 The Six Stages of an ML Project

Every ML project follows a cyclical pattern, not a linear one. Understanding this lifecycle is crucial.

The ML Project Lifecycle

Forward Process:

1. **Raw Data** → Collect data from various sources
2. **ETL (Cleanse)** → Extract, Transform, Load—clean and prepare data
3. **Featurize** → Create meaningful features for the model
4. **Train** → Train the model (the "small" part!)
5. **Serve/Inference** → Deploy and serve predictions
6. **Monitor** → Watch for performance degradation

Feedback Loop: When monitoring detects drift → Return to step 1, 2, or 3 to retrain

Example: The Feedback Loop in Action

Scenario: You built a fraud detection model in January.
May: Monitoring shows false positive rate increased by 40%.
Investigation: New payment methods (Apple Pay, crypto) have emerged that your model never saw during training.
Action:

- Go back to **Featurize**: Add features for new payment types
- **Train**: Retrain on 6 months of new data
- **Deploy**: Push the updated model to production

This is why ML is a **continuous cycle**, not a one-time project.

2.2 The Three Key Personas

The ML lifecycle requires collaboration between three specialized roles:

The Silo Problem

Traditionally, these three roles work in **silos** (separate teams with poor communication):

1. DS builds a model in Jupyter Notebook → Hands off code to MLE
2. MLE rewrites the code for production (Docker, APIs) → Different environment!
3. Versions get confused, environments differ, results can't be reproduced
4. Deployment takes **weeks or months** instead of hours

MLOps and **MLflow** break down these silos by standardizing how models are tracked, packaged, and deployed.

Table 1: *The Three Pillars of ML Projects*

Aspect	Data Engineer (DE)	Data Scientist (DS)	ML Engineer (MLE)
Primary Mission	Build data infrastructure	Extract insights & build models	Productionize & operate models
Key Tasks	- Data ingestion - Data storage - ETL pipelines - Data curation	- Feature engineering - Model development - Training & tuning - Validation	- Model evaluation - Packaging - Deployment & serving - MLOps pipelines
Analogy	Raw material supplier & factory builder	Prototype developer	Mass production line operator
Where in Medallion	Bronze → Silver	Silver → Gold	Gold → Production

3 Feature Engineering and Feature Stores

3.1 What is Feature Engineering?

Definition: Feature Engineering

Feature Engineering is the process of using domain knowledge to transform raw data into meaningful inputs (features) that help ML algorithms learn more effectively.

Andrew Ng's famous quote: "Applied machine learning is basically feature engineering."

This means that the quality of your features often matters more than the sophistication of your algorithm. Good features can make a simple model outperform a complex one with poor features.

Example: From Raw Data to Features

Raw Transaction Data:

- Customer ID, Product Name, Purchase DateTime, Amount

Engineered Features:

- `avg_transaction_amount`: Average purchase amount per customer (aggregation)
- `purchase_frequency_7d`: Number of purchases in last 7 days (time-window aggregation)
- `is_weekend`: Whether purchase was on weekend (transformation)
- `weather_on_purchase`: Weather conditions at purchase time (external data enrichment)
- `customer_lifetime_value`: Predicted total revenue from customer (complex aggregation)

The model learns much more from `purchase_frequency_7d` than from raw timestamps. This is the power of feature engineering.

3.2 Why Feature Engineering is Expensive

Creating good features often requires:

- **Domain expertise**: Understanding what matters in your business
- **Significant computation**: "7-day rolling average" requires scanning millions of records

- **Time-consuming iteration**: Testing which features actually help

3.3 The Training-Serving Skew Problem

Here's where reproducibility becomes critical:

Training-Serving Skew

Symptom: "My model had 99% accuracy in training, but only 60% in production!"
Cause: Features were computed **differently** during training vs. serving:
During Training (Offline):

```sql
-- Batch computation, can be slow
SELECT AVG(purchase_amount) as avg_purchase
FROM transactions
WHERE customer_id = ?
GROUP BY customer_id
```

During Serving (Online):

```python
# Real-time computation, must be fast
avg_purchase = (sum_so_far + new_purchase) / (count + 1)
```

Even small differences (rounding, null handling, time zones) can cause features to be **slightly different**. These small differences compound and destroy model performance.

3.4 Feature Stores: The Solution

Definition: Feature Store

A **Feature Store** is a centralized repository for storing and serving ML features. It ensures that the **exact same features** are used during both training and serving.
Key Benefits:

1. **Consistency**: Features computed once, used everywhere (eliminates skew)
2. **Reusability**: Team A's expensive "customer profile" feature can be reused by Team B, C, D
3. **Efficiency**: Avoid redundant computation of expensive features
4. **Governance**: Track feature lineage, ownership, and freshness

Example: Feature Store in Action

Without Feature Store:

- Team A computes "customer_lifetime_value" for their fraud model
- Team B computes it again (slightly differently) for their churn model
- Team C computes it yet again for their recommendation model
- Result: 3x computation cost, inconsistent features, debugging nightmares

With Feature Store:

- Feature team computes "customer_lifetime_value" once

- Stores it in the Feature Store with documentation
- Teams A, B, C all read the **exact same feature**
- Training and serving both use the Feature Store as the source of truth

4 ML Pipelines: Transformers vs. Estimators

When building ML systems, especially with frameworks like Spark ML, you'll encounter two fundamental concepts: **Transformers** and **Estimators**. Understanding the difference is crucial.

4.1 The Key Distinction: Learning vs. Not Learning

Table 2: *Transformers vs. Estimators*

Aspect	Transformer	Estimator
Purpose	Data preprocessing/transformation	**Learning** from data
Key Method	`.transform()`	`.fit()`
Input → Output	DataFrame → DataFrame	DataFrame → **Model**
Does it Learn?	No (applies fixed rules)	**Yes** (learns from data)
Examples	- StringIndexer - OneHotEncoder - StandardScaler - VectorAssembler	- LinearRegression - RandomForestClassifier - XGBoostRegressor - KMeans

Example: Cooking Pipeline Analogy

Imagine a food assembly line:
Transformers (Food Prep Machines):

- **StringIndexer** = Onion peeler (string "red" → number 0)
- **StandardScaler** = Portion measurer (normalize ingredient weights)
- **VectorAssembler** = Ingredient combiner (put all prepped items together)

Estimator (The Chef):

- **LinearRegression** = A chef who **learns** the perfect recipe by tasting many dishes
- Input: Ingredients (DataFrame)
- Output: A trained recipe (Model) that can make predictions on new ingredients

4.2 The Critical Insight: Estimators Produce Models

Here's the key insight that often confuses beginners:

Key Summary

When you call `.fit()` on an **Estimator**, the output is a **Model**.
And crucially, this **Model** is itself a **Transformer**!
Why? Because once trained, the model can `.transform()` new data into predictions.
Conceptual Flow:

1. `Estimator.fit(training_data)` → Returns a `Model`
2. `Model.transform(new_data)` → Returns predictions (DataFrame)

```python
from sklearn.preprocessing import StandardScaler  # Transformer
from sklearn.linear_model import LinearRegression  # Estimator

# Transformer: transform() takes data, returns transformed data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_train)  # fit() learns stats, transform()
    applies

# Estimator: fit() takes data, returns a trained MODEL
model = LinearRegression()
model.fit(X_scaled, y_train)  # Learning happens here!

# The trained model can now transform (predict) new data
predictions = model.predict(X_test_scaled)  # Model acts like a Transformer
```

Listing 1: Transformer vs. Estimator in Code

5 Model Drift: Why Models Degrade Over Time

5.1 Understanding Model Drift

Definition: Model Drift

Model Drift is the degradation of model performance over time due to changes in the underlying data patterns that the model was trained on.
Key Insight: Models are trained on historical data. When the "future" arrives and looks different from the "past," the model struggles.

Example: The Driver's License Analogy

Imagine you passed your driving test in 2010 (training data).
2020 Problems:

- New traffic laws you never learned
- Electric vehicles with different behaviors
- Roundabouts that didn't exist in your town
- GPS navigation (you only knew paper maps)

Your 2010 "model" (driving skills) is now outdated. This is model drift—the world changed, but your model didn't.

5.2 The Four Types of Drift

Understanding **what** drifted helps you know **how** to fix it:

Table 3: *The Four Types of Model Drift*

Drift Type	What Changed?	Response Strategy
Feature Drift	The distribution of **input features (X)** changed. *Example: A new product category appears that the model never saw.*	Review feature engineering. Retrain with new data.
Label Drift	The distribution of **target labels (Y)** changed. *Example: Fraud rate suddenly doubles due to new attack vectors.*	Review label generation process. Retrain with new data.
Prediction Drift	The distribution of **model predictions** ($\hat{Y}$) changed. *Example: Model suddenly predicts everything as "not fraud."*	Investigate model behavior. Check for data pipeline issues.
Concept Drift	The **relationship between X and Y** fundamentally changed. **(Most severe!)** *Example: COVID-19 changed how "weekend" relates to "store visits"*	**Simple retraining won't work!** Need new features, new model architecture.

Concept Drift is the Most Dangerous

Feature drift and label drift can often be fixed by retraining on new data.

Concept drift means the fundamental rules of your domain have changed. The relationship that existed in your training data **no longer exists**.

COVID-19 Example:

- **Pre-COVID**: “Weekend” → “High store traffic” (strong positive correlation)
- **During COVID**: “Weekend” → “Zero store traffic” (correlation broken)
- **Post-COVID**: “Weekend” → “Moderate online traffic” (new relationship)

Your old model learned the wrong relationships. You need to completely rethink your features and possibly your entire model architecture.

6 The Solution: MLOps and the Three "Ops"

6.1 From DevOps to MLOps

To understand MLOps, first understand the "Ops" progression:

The Evolution of "Ops"

- **DevOps** = Development + Operations
  - Automates building, testing, and deploying **software code**
  - Tools: Git, Jenkins, Docker, Kubernetes
- **DataOps** = Data + Operations
  - Automates building, testing, and deploying **data pipelines**
  - Tools: Airflow, dbt, Delta Live Tables
- **MLOps** = Machine Learning + Operations
  - Automates training, testing, deploying, and monitoring **ML models**
  - Tools: MLflow, Kubeflow, SageMaker

MLOps = DataOps + ModelOps + DevOps

It's the integration of all three: data pipelines feed feature stores, which feed model training, which feeds model serving, which feeds monitoring, which triggers retraining.

6.2 Why MLOps Matters

MLOps Goals

1. **Reproducibility**: Anyone can recreate any model at any time
2. **Automation**: Models retrain and redeploy automatically when drift is detected
3. **Collaboration**: DS, DE, and MLE work from the same platform
4. **Governance**: Track who created what model, with what data, for what purpose
5. **Speed**: Deploy models in hours, not months

7 MLflow: The Swiss Army Knife for MLOps

7.1 What is MLflow?

Definition: MLflow

MLflow is an open-source platform for managing the entire ML lifecycle. Created by Databricks, it has become the de facto standard—even competitors use it.

Key Characteristics:

- **Open Source**: Free, community-backed, works anywhere
- **Framework Agnostic**: Works with Scikit-learn, PyTorch, TensorFlow, XGBoost, etc.
- **API First**: REST APIs, Python/R/Java clients
- **Modular**: Use all components or just the ones you need

7.2 The Four Components of MLflow

MLflow provides four integrated components that solve the reproducibility crisis:

Table 4: *MLflow's Four Components*

Component	Problem It Solves	What It Does
1. Tracking	"What parameters did I use for that good model last week?"	Records all experiments: parameters, metrics, artifacts, code
2. Projects	"It worked on my laptop but fails on the server"	Packages code + environment (dependencies) together
3. Models	"How do I serve a PyTorch model in a Spark pipeline?"	Standard model format with multiple "flavors" for deployment
4. Model Registry	"Which model is in production? Is v3 tested?"	Central hub for model versioning, staging, and promotion

7.2.1 Component 1: MLflow Tracking

MLflow Tracking: Your Experiment Notebook

Problem: Data scientists run hundreds of experiments, changing parameters each time. "Which hyperparameters gave me the best result?" becomes impossible to answer.

Solution: Automatically log everything about every experiment.

What Gets Tracked:

- **Parameters**: Input hyperparameters (e.g., `learning_rate=0.01`)
- **Metrics**: Output performance measures (e.g., `accuracy=0.95`)
- **Artifacts**: Files—the model itself, plots, feature importance graphs
- **Source**: The code that ran the experiment (e.g., notebook version)
- **Tags**: Custom metadata (e.g., `team=fraud-detection`)

```python
import mlflow
```

```python

# OPTION 1: Manual Logging (explicit control)
with mlflow.start_run(run_name="my_experiment"):
    # Log input parameters
    mlflow.log_param("learning_rate", 0.01)
    mlflow.log_param("max_depth", 5)

    # Train your model (any framework)
    model = train_my_model(...)

    # Log output metrics
    mlflow.log_metric("accuracy", 0.95)
    mlflow.log_metric("f1_score", 0.92)

    # Log artifacts (files)
    mlflow.log_artifact("feature_importance.png")
    mlflow.sklearn.log_model(model, "model")

# OPTION 2: AutoLog (magic one-liner!)
mlflow.autolog()  # <-- This single line logs EVERYTHING automatically

# Now just train as usual - MLflow captures all params, metrics, model
model = XGBClassifier(learning_rate=0.01, max_depth=5)
model.fit(X_train, y_train)  # autolog captures this automatically!
```

Listing 2: MLflow Tracking: Manual vs. AutoLog

7.2.2 Component 2: MLflow Projects

MLflow Projects: Environment Packaging

Problem: "It worked on my machine" — the eternal curse of data science.
Solution: Package the code **and** its environment together.
How It Works: Your project folder contains an `MLproject` file that specifies:

- The entry point (e.g., `python train.py`)
- The environment file (e.g., `conda.yaml` or `requirements.txt`)

Anyone can then run `mlflow run <project_path>` and MLflow automatically:

1. Creates an isolated environment with exact library versions
2. Runs the code in that environment
3. Guarantees reproducible results

7.2.3 Component 3: MLflow Models

MLflow Models: Universal Model Format

Problem: DS trained with Scikit-learn, but MLE needs to serve via Spark UDF or REST API.
Solution: Save models in a standardized format with multiple "flavors."
Model Flavors:

- `python_function` (pyfunc): Universal wrapper—any model can be called as a Python function
- `sklearn`: Native Scikit-learn format
- `pytorch`: Native PyTorch format
- `spark`: Ready for Spark UDF deployment

Key Benefit: Save a model once, deploy it anywhere. The MLE doesn't need to know if it was originally Scikit-learn or PyTorch—they just use the `pyfunc` flavor.

7.2.4 Component 4: MLflow Model Registry

MLflow Model Registry: Git for Models

Problem: "Which model is currently in production? Is v3 tested? Who approved v2?"
Solution: A central repository for managing model versions and lifecycle stages.
Key Features:

- **Versioning**: Every registered model gets v1, v2, v3, etc.
- **Stage Management**: Models transition through stages
  - **None**: Just registered, not deployed
  - **Staging**: Being tested (A/B testing)
  - **Production**: Serving real traffic
  - **Archived**: Retired, kept for audit
- **Aliases**: Human-readable names like "champion" or "best_model"

8 Practical MLflow Usage

8.1 The Complete MLflow Workflow

Here's how all four components work together:

```python
import mlflow
from sklearn.model_selection import train_test_split
from xgboost import XGBRegressor

# 1. SETUP: Configure MLflow to use Unity Catalog for model storage
mlflow.set_registry_uri("databricks-uc")

# 2. TRACKING: Start an experiment run
mlflow.set_experiment("/my_experiments/house_price_prediction")

with mlflow.start_run(run_name="xgboost_tuning_v1"):

    # 3. Log parameters (inputs)
    params = {"max_depth": 6, "learning_rate": 0.1, "n_estimators": 100}
    mlflow.log_params(params)

    # 4. Train the model
    model = XGBRegressor(**params)
    model.fit(X_train, y_train)

    # 5. Evaluate and log metrics (outputs)
    predictions = model.predict(X_test)
    rmse = mean_squared_error(y_test, predictions, squared=False)
    mlflow.log_metric("rmse", rmse)

    # 6. MODELS: Log the model with signature (input/output schema)
    signature = mlflow.models.infer_signature(X_test, predictions)
    mlflow.xgboost.log_model(
        model,
        "model",
        signature=signature,
        input_example=X_test[:5]
    )

    # 7. Log additional artifacts
    import matplotlib.pyplot as plt
    # ... create feature importance plot ...
    mlflow.log_figure(fig, "feature_importance.png")

# 8. REGISTRY: Register the best model
model_uri = f"runs:/{mlflow.active_run().info.run_id}/model"
mlflow.register_model(model_uri, "my_catalog.my_schema.house_price_model")

# 9. Load and use the registered model for predictions
```

```python
loaded_model = mlflow.pyfunc.load_model(
    "models:/my_catalog.my_schema.house_price_model@best"  # Using alias
)
new_predictions = loaded_model.predict(new_data)
```

Listing 3: Complete MLflow Workflow

8.2 Hyperparameter Tuning with MLflow

When doing hyperparameter tuning (with Optuna, Hyperopt, etc.), use **nested runs**:

```python
import optuna

def objective(trial):
    # Each trial is a nested (child) run
    with mlflow.start_run(nested=True):
        params = {
            "max_depth": trial.suggest_int("max_depth", 2, 10),
            "learning_rate": trial.suggest_float("lr", 0.01, 0.3),
        }
        mlflow.log_params(params)

        model = XGBRegressor(**params)
        model.fit(X_train, y_train)
        rmse = evaluate(model, X_val, y_val)
        mlflow.log_metric("rmse", rmse)

        return rmse  # Optuna minimizes this

# Parent run contains all child runs
with mlflow.start_run(run_name="hyperparameter_tuning"):
    study = optuna.create_study(direction="minimize")
    study.optimize(objective, n_trials=50)

    # Log the best parameters
    mlflow.log_params(study.best_params)
    mlflow.log_metric("best_rmse", study.best_value)
```

Listing 4: Hyperparameter Tuning with Nested Runs

8.3 Important Lab Modifications

Lab Environment Modifications

When running the labs in the free Databricks Community Edition:

1. Change Catalog Name: Replace `main.default` with your own catalog (e.g., `cscie103_catalog.default`)

2. Use Local Trials Instead of SparkTrials:

```python
# ORIGINAL (may fail in serverless):
```

```python
from hyperopt import SparkTrials
trials = SparkTrials(parallelism=4)

# MODIFIED (works everywhere):
from hyperopt import Trials
trials = Trials()  # Single-node execution
```

3. Reduce Trial Count for Faster Testing: Change `n_trials=50` to `n_trials=10` or `max_evals=10`

9 A/B Testing and Model Promotion

9.1 The Champion-Challenger Pattern

When you have a new model, you don't immediately replace the production model. Instead, you use **A/B testing**:

Definition: Champion-Challenger Testing

- **Champion**: The current production model (proven performance)
- **Challenger**: The new model (potentially better, but unproven)

Process:

1. Deploy challenger to **Staging**
2. Route 10-20% of traffic to challenger, 80-90% to champion
3. Monitor performance metrics for both
4. If challenger wins consistently, promote to **Production**
5. Move old champion to **Archived**

```python
from mlflow import MlflowClient

client = MlflowClient()

# Register a new model version
model_version = mlflow.register_model(model_uri, "fraud_detection_model")

# Promote to Staging for A/B testing
client.set_registered_model_alias(
    "fraud_detection_model",
    "challenger",
    model_version.version
)

# After successful A/B testing, promote to Production
client.set_registered_model_alias(
    "fraud_detection_model",
    "champion",  # or "production"
    model_version.version
)

# Archive the old champion
client.delete_registered_model_alias("fraud_detection_model", "old_champion")
```

Listing 5: Model Stage Transitions in MLflow

10 Key Metrics and Evaluation

10.1 Common ML Metrics

Table 5: *Common ML Evaluation Metrics*

Task Type	Metric	When to Use
4***Regression**	RMSE (Root Mean Squared Error)	General purpose, penalizes large errors
	MAE (Mean Absolute Error)	When outliers should have less impact
	R^2 (R-squared)	Explains variance in predictions
	MAPE	When you need percentage error
5***Classification**	Accuracy	Balanced classes only!
	Precision	When false positives are costly (spam detection)
	Recall	When false negatives are costly (disease detection)
	F1 Score	Balance of precision and recall
	AUC-ROC	Overall model discrimination ability

10.2 The Confusion Matrix

For classification, the confusion matrix is essential:

Example: Understanding the Confusion Matrix

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Key Insight: Different domains care about different quadrants!

- **Medical Diagnosis**: Minimize FN (don't miss cancer!)
- **Spam Filter**: Minimize FP (don't delete important emails!)
- **Fraud Detection**: Balance both (catch fraud but don't block legitimate users)

11 Quick Review: ML Concepts from Previous Lectures

11.1 Scaling Concepts Review

Table 6: *Scaling Terminology Quick Reference*

Term	**Definition**
Scale Out (Horizontal)	Add more nodes/workers to the cluster
Scale Up (Vertical)	Upgrade to larger instance (more CPU/RAM)
Elasticity	Ability to automatically grow/shrink resources
Availability	Ability to use service at any time (uptime guarantee)
Autoscale	Dynamic adjustment of node count based on load

11.2 ML Algorithm Categories Review

Table 7: *ML Algorithm Categories*

	Supervised	**Unsupervised**
Discrete (Classification)	Classification (spam/not spam)	Clustering (customer segments)
Continuous (Prediction)	Regression (house prices)	Dimensionality Reduction (PCA)

11.3 BI vs. BA Review

- **Business Intelligence (BI)**: “What happened?” — Descriptive, dashboards, reports
- **Business Analytics (BA)**: “Why did it happen? What will happen?” — Predictive, ML models

12 Summary: One-Page Quick Reference

The Problem: ML Reproducibility Crisis

1. Hidden Technical Debt: ML code is the tip of the iceberg. Data management, infrastructure, monitoring are the 90% below water.

2. Fragmented Ecosystem: Python, R, TensorFlow, Spark—too many tools, version conflicts, “worked on my machine” syndrome.

3. Model Drift: Models are living assets that degrade as real-world patterns change.

The Solution: MLOps + MLflow

MLOps: Culture + Practice of automating the entire ML lifecycle.

MLflow: Open-source framework implementing MLOps with 4 components.

MLflow Tracking (Experiment Notebook)

Purpose: Never forget what you did.

What it logs: Parameters, metrics, artifacts, source code.

Magic command: `mlflow.autolog()` — logs everything automatically.

MLflow Projects (Environment Packaging)

Purpose: “Works on any machine.”

How: Packages code + `requirements.txt` together.

Run anywhere: `mlflow run <project>` reproduces exact environment.

MLflow Models (Standard Format)

Purpose: Train once, deploy anywhere.

Flavors: `pyfunc`, `sklearn`, `spark`, `pytorch`.

Key benefit: MLE doesn’t care what framework DS used.

MLflow Model Registry (Git for Models)

Purpose: Track model versions and lifecycle.

Stages: None → Staging → Production → Archived

Aliases: Human names like “champion”, “best_model”

Key Concepts Quick Reference

- **Feature Store**: Central repository for features; prevents training-serving skew
- **Transformer**: Applies fixed rules (`.transform()`) — no learning
- **Estimator**: Learns from data (`.fit()`) — produces Models
- **Model Drift**: Performance degradation over time (Feature, Label, Prediction, Concept)
- **A/B Testing**: Champion vs. Challenger model comparison

Lecture 09: MLOps and The Importance of Good Data

CSCI E-103: Reproducible Data Science and Machine Learning

Harvard University

■ **Course:**	CSCI E-103: Reproducible Data Science
■ **Week:**	Lecture 09
■ **Instructors:**	Anindita Mahapatra & Eric Gieseke
■ **Objective:**	Master MLOps workflows, understand the roles involved, and learn why data quality matters more than model complexity

Contents

1 What is MLOps and Why Does It Matter?

Lecture Overview

This lecture focuses on the **operational** side of machine learning: how to take a model from a Jupyter notebook to a production system that reliably delivers value.

Key Topics:

- The definition and importance of MLOps
- The five key personas in ML projects
- The three core pipelines: Training, Inferencing, Monitoring
- Model-Centric vs. Data-Centric AI
- Deployment strategies: Deploy Models vs. Deploy Code
- Hyperparameter tuning and model selection
- End-to-end MLOps lab walkthrough (Customer Churn Prediction)

1.1 From DevOps to MLOps

In traditional software, **DevOps** (Development + Operations) revolutionized how we build and deploy applications by automating builds, tests, and deployments.

But AI systems are fundamentally different:

- **Traditional Software = Code**
- **AI Systems = Code (Model/Algorithm) + Data**

Because AI systems depend on **both** code and data, they are sensitive to changes in either. A model trained on last year's data might perform poorly on this year's data, even if the code hasn't changed.

Definition: MLOps

MLOps (Machine Learning Operations) is the set of standards, tools, processes, and methodologies that aim to:

- **Optimize time and efficiency** in ML development
- **Ensure quality** of models in production
- **Guarantee governance** (security, compliance, auditability)

It's the intersection of **Machine Learning**, **Data Engineering**, and **DevOps**.

1.2 Business Context: Customer Churn Prediction

Every ML project starts with a business problem. Let's use **customer churn prediction** as our running example.

Example: The Business Case for Churn Prediction

The Problem:

- Customers sign up for a service, use it for a while, then leave ("churn")
- Acquiring a new customer costs 5-10x more than retaining an existing one
- The business wants to **predict** which customers are about to leave **before** they do

The ML Solution:

- Build a model that scores each customer's "churn risk"
- When a customer is flagged as high-risk, offer them a discount or incentive to stay
- Result: Reduced churn, saved revenue, happier customers

Why MLOps?

- The model needs to be trained on historical data
- It needs to be deployed so the marketing team can act on predictions
- It needs to be monitored—customer behavior changes over time!
- If performance degrades, it needs to be automatically retrained

MLOps is the system that makes all of this happen reliably and repeatedly.

2 The Five Key Personas in MLOps

MLOps is not a solo endeavor. It requires collaboration between specialists with different skills.

2.1 1. Business Stakeholder / Business Analyst

- **Primary Mission:** Translate vague business goals into clear, measurable ML problems
- **Key Responsibilities:**
  - **Define the problem:** "What is churn?" $\rightarrow$ "A customer who hasn't purchased in 90 days OR cancelled their subscription within 30 days"
  - **Set KPIs:** How will success be measured? (e.g., reduction in churn rate, ROI of interventions)
  - **Stakeholder alignment:** Work with marketing, sales, product, and finance to ensure the model serves real business needs
  - **Validate results:** After deployment, verify that the model is actually reducing churn
- **Deliverables:** Problem statement, data requirements, evaluation criteria, business impact analysis

2.2 2. Data Engineer (DE)

- **Primary Mission:** Build and maintain the data infrastructure
- **Key Responsibilities:**
  - **Data collection:** Pull data from CRM systems, web logs, billing, support tickets
  - **ETL/ELT pipelines:** Extract, Transform, Load data into usable formats
  - **Data quality:** Handle missing values, duplicates, inconsistencies, schema validation
  - **Infrastructure:** Manage scalable storage on AWS, Azure, or GCP
  - **Lineage & versioning:** Track where data came from and which version was used
- **Deliverables:** Production-grade, trusted datasets ready for analysis and modeling

2.3 3. Data Scientist (DS)

- **Primary Mission:** Transform data into predictive insights (build the model)
- **Key Responsibilities:**
  - **Exploratory Data Analysis (EDA):** Understand patterns in the data
  - **Feature Engineering:** Create meaningful variables (e.g., "number of support tickets in last 30 days")
  - **Model Selection & Training:** Experiment with algorithms (Logistic Regression, Random Forest, XGBoost)
  - **Evaluation:** Measure performance using accuracy, precision, recall, F1
  - **Interpretation:** Understand which factors drive churn (feature importance)
  - **Collaboration:** Work with business stakeholders to validate that the model makes sense
- **Deliverables:** Trained models, feature sets, validation reports, insights

2.4 4. Machine Learning Engineer (MLE)

- **Primary Mission:** Take the model from prototype to production and keep it running
- **Key Responsibilities:**
  - **Deployment:** Expose the model via APIs or batch scoring pipelines
  - **CI/CD:** Automate retraining, testing, version control, and deployment
  - **Monitoring:** Track model performance, data drift, latency, and bias
  - **Scaling:** Ensure the model can handle production traffic
  - **Feedback integration:** Incorporate new data and trigger retraining when needed
- **Deliverables:** Production churn prediction service, monitoring dashboards, automated retraining workflows

2.5 5. Data Governance Officer

- **Primary Mission:** Ensure compliance and security across the entire pipeline
- **Key Responsibilities:**
  - Data security and access controls
  - PII (Personally Identifiable Information) anonymization
  - Regulatory compliance (GDPR, HIPAA, etc.)
  - Audit trails and documentation

Role Overlap in the ML Pipeline

Different stages of the pipeline require collaboration:

Stage	**Primary Owners**
Data Prep	Data Engineer
EDA & Feature Engineering	Data Scientist
Model Training & Validation	Data Scientist + ML Engineer
Deployment	ML Engineer
Monitoring	ML Engineer
Business Validation	Business Analyst + Data Scientist
Governance	Data Governance (all stages)

3 The Three Core Pipelines of MLOps

MLOps consists of three interconnected pipelines that form a continuous cycle.

3.1 Pipeline 1: Model Training

Training Pipeline Goal

Transform raw data into a trained model artifact that generalizes well (doesn't overfit).

Key Steps:

1. **Data Preparation:** Split data into Train / Validation / Test sets
2. **Feature Engineering:** Create meaningful features from raw variables
3. **Model Selection:** Choose appropriate algorithms
4. **Hyperparameter Tuning:** Optimize model settings to improve accuracy and reduce overfitting
5. **Evaluation:** Measure performance (Accuracy, Precision, Recall, F1, AUC-ROC)
6. **Artifact Storage:** Save the trained model and log metadata to MLflow

3.2 Pipeline 2: Model Inferencing (Serving)

Inferencing Pipeline Goal

Use the trained model to generate predictions on new data and support business decisions.

Two Main Approaches:

Table 1: *Batch vs. Real-time Inferencing*

Aspect	Batch Inferencing	Real-time Inferencing
How it works	Process large datasets periodically	Return predictions instantly per request
Latency	Minutes to hours	Milliseconds to seconds
Example	"Every night, score all customers for churn risk"	"When customer hovers over cancel button, show discount popup"
Infrastructure	Scheduled jobs (Airflow, Databricks)	REST APIs, streaming (Kafka)

Key Considerations: Latency, throughput, scalability, security

3.3 Pipeline 3: Model Monitoring

Monitoring Pipeline Goal

Continuously track model performance in production and detect issues before they cause business harm.

What to Monitor:

Table 2: *Types of Drift to Monitor*

Drift Type	**What Changed?**	**How to Detect**
Data Drift	The statistical distribution of **input features** changed *Example: New marketing campaign brings in customers from a different demographic*	Compare input data statistics (mean, variance, distribution) against baseline
Concept Drift	The **relationship** between inputs and target changed *Example: Customers now churn due to poor service, not price*	Most difficult. Monitor business KPIs even when data drift is absent
Model Decay	Overall model performance degradation	Track accuracy, F1, precision, recall against ground truth when available

Actions on Detection:

- Send alerts to data scientists
- Trigger automatic retraining pipeline
- Roll back to a previous model version

Example: How is Drift Detected Automatically?

Data Drift:

1. During training, save statistical profiles (mean, std, distribution) of each feature as a **baseline**
2. In production, periodically compute the same statistics on incoming data
3. Use statistical tests (KS test, Jensen-Shannon divergence) to compare
4. If difference exceeds threshold $\rightarrow$ alert!

Model Drift:

1. When ground truth becomes available (e.g., did the customer actually churn?), compare predictions to reality
2. Track metrics over time (e.g., F1 score by week)
3. If metrics drop below threshold $\rightarrow$ trigger retraining

Concept Drift:

- Hardest to detect automatically
- Often manifests as: "Data looks the same, model metrics look okay, but business KPIs are declining"
- Requires human investigation and domain expertise

4 Deployment Strategies: Models vs. Code

When promoting a model from Dev → Staging → Production, there are two main strategies.

4.1 Strategy 1: Deploy Models

- **How it works:** Train the model in Dev. Take the resulting **model artifact (file)** and copy it to Staging, then to Production.
- **Flow:** [Train in Dev] → [Model file] → [Copy to Staging] → [Copy to Prod]
- **Pros:** Faster (no retraining), less compute cost
- **Cons:** Model was trained on Dev data, which may not represent Production data

4.2 Strategy 2: Deploy Code

- **How it works:** Develop the **training code** in Dev. Copy the code to Staging and Production. **Retrain the model in each environment using that environment's data.**
- **Flow:** [Develop code in Dev] → [Copy code to Staging] → [Retrain with Staging data] → [Copy code to Prod] → [Retrain with Prod data]

4.3 Why Deploy Code is Often Preferred

Table 3: *Benefits of Deploy Code Strategy*

Benefit	Explanation
Data Access Control	**The biggest advantage.** Sensitive production data never needs to leave the production environment. Each environment trains only on its own data.
Reproducibility	Engineering controls the training environment in each stage, making reproduction easier
Real Data Issues	Production data has skews and volumes that Dev data doesn't. Training in Prod catches issues that only appear at scale.
Modular Code	Forces data scientists to write clean, modular, testable code instead of handing off notebooks

Downsides of Deploy Code

- **More compute cost:** Model is retrained in every environment
- **Infrastructure requirements:** Requires CI/CD setup with unit/integration testing
- **Skill requirements:** Data scientists must write production-quality code, not just notebooks

5 Data-Centric AI: The Key Insight

This is one of the most important concepts in modern MLOps.

5.1 Two Ways to Improve AI

1. **Model-Centric AI:** "How can I improve the model/algorithm/code to get better performance?"
2. **Data-Centric AI:** "How can I systematically improve the **data** to get better performance?"

Definition: Data-Centric AI

Data-Centric AI is the approach of improving AI system performance by focusing on the quality, consistency, and coverage of the **data** rather than tweaking the model architecture or hyperparameters.

5.2 Big Data vs. Good Data

Having more data (Big Data) doesn't guarantee better models. **Bad data in, bad predictions out.**

What is "Good Data"?

- **Unambiguous labels:** Ground truth is consistent and correct
- **Good coverage:** Includes important edge cases and scenarios (e.g., if predicting for all states, don't train only on Massachusetts)
- **Timely feedback:** Fresh data from production is quickly incorporated
- **Appropriately sized:** Not too small (underfitting), not unnecessarily large

5.3 Why Data-Centric AI Wins

Example: The Cooking Analogy

Data is the ingredients. The model is the recipe.
Even the best chef (model) can't make a great dish with rotten ingredients (bad data).
Model-Centric: "Let's adjust the cooking time from 15 minutes to 16 minutes." $\rightarrow$ Minor improvement
Data-Centric: "Let's source fresher, higher-quality ingredients." $\rightarrow$ Major improvement

Real-World Evidence:

In industrial case studies (steel defect detection, solar panel inspection), improving data quality led to **10-16% improvement** in model performance, while tweaking the model algorithm led to only **0-4% improvement**.

The takeaway: MLOps should prioritize processes that ensure high-quality, consistent data flowing into your models.

Table 4: *Model-Centric vs. Data-Centric Results (Example)*

Use Case	Model-Centric Improvement	Data-Centric Improvement
Steel Defect Detection	+0%	+16.9%
Solar Panel Inspection	+0.4%	+3.1%
Surface Inspection	+0.3%	+4.0%

6 Hyperparameter Tuning and Model Selection

A critical part of the Training Pipeline is finding the best model configuration.

6.1 What are Hyperparameters?

Definition: Hyperparameters

Hyperparameters are model settings that the developer chooses **before** training, as opposed to **parameters** which the model learns during training.

Examples:

- Learning rate
- Number of trees in a forest
- Maximum tree depth
- Regularization strength

6.2 Tuning Methods

Table 5: *Hyperparameter Tuning Methods*

Method	How It Works	Pros / Cons
Grid Search	Try **all combinations** of specified parameter values	Simple but expensive (exponential growth: $O(n^k)$)
Random Search	Randomly sample from parameter space	Often more efficient than grid search for the same compute budget
Bayesian Search	Use past results to intelligently choose next combination	Converges faster to optimal; used by Optuna, Hyperopt
Genetic Algorithms	Combine "good" and "bad" parameter sets like breeding	Good for very large search spaces (deep learning)

Tools: `Optuna`, `Hyperopt` (covered in Lecture 08)

6.3 K-Fold Cross Validation

Definition: K-Fold Cross Validation

A technique to evaluate model performance without "cheating" by using the test set.

Process (K=3):

1. Split training data into 3 "folds": [Fold 1], [Fold 2], [Fold 3]
2. **Round 1:** Train on [1, 2], validate on [3]
3. **Round 2:** Train on [1, 3], validate on [2]
4. **Round 3:** Train on [2, 3], validate on [1]
5. Final score = **average** of 3 validation scores

Benefit: The test set (holdout) remains untouched until final evaluation, preventing overfitting to the validation set.

7 MLOps Maturity Model

Organizations don't achieve full MLOps maturity overnight. There's a progression:

Table 6: *MLOps Maturity Levels*

Level	Characteristics	Example
Level 1: Beginner	Use familiar tools; manual processes; plan for more complex workloads	Data scientist trains model manually, emails pickle file to engineering
Level 2: Intermediate	Scale data and workloads; focus on **automation and reproducibility**; unify data teams	Automated retraining on schedule; MLflow tracking in place
Level 3: Advanced	**Faster production cycles** (deploy multiple times daily); end-to-end automation; **resilience testing** (Chaos Monkey); robust rollback	Netflix-style: intentionally break production to test recovery time

Example: Netflix's Chaos Monkey

Netflix intentionally introduces failures into their production systems to test how quickly the system can recover. This philosophy extends to MLOps: if a model deployment fails, how quickly can you roll back? How fast can you retrain?

The goal is to not fear failure, but to build systems that recover gracefully.

8 Champion-Challenger Pattern

When deploying a new model, you don't immediately replace the production model. Instead, you use a **Champion-Challenger** approach.

Definition: Champion-Challenger

- **Champion:** The current production model (proven performance)
- **Challenger:** The new candidate model (potentially better, but unproven)

Process:

1. Train a new model (Challenger)
2. Compare Challenger's metrics (F1, accuracy) against Champion's
3. If Challenger is better → promote Challenger to Champion
4. Archive the old Champion

In MLflow, this is implemented using **Model Aliases**:

```python
# Set new model as challenger
client.set_registered_model_alias("churn_model", "challenger", version)

# After validation, promote to champion
client.set_registered_model_alias("churn_model", "champion", version)

# Load model by alias for inference
model = mlflow.pyfunc.load_model("models:/churn_model@champion")
```

9 End-to-End MLOps Lab Walkthrough

The lab demonstrates a complete MLOps pipeline for customer churn prediction using Databricks, MLflow, and Unity Catalog.

9.1 Lab Architecture

- **Data Layer:** Delta Lake tables in Unity Catalog
- **ML Lifecycle:** MLflow for tracking, models, and registry
- **Governance:** Unity Catalog for data and model governance
- **Algorithm:** LightGBM classifier

9.2 Step 1: Data Preparation & Feature Engineering

```python
# Read raw customer data
telco_df = spark.table("mlops_churn_bronze_customer")

# Create new feature: number of optional services
def add_optional_services(df):
    service_cols = ['online_security', 'online_backup',
                    'device_protection', 'tech_support']
    df['num_optional_services'] = (
        df[service_cols].apply(lambda x: (x == 'Yes').sum(), axis=1)
    )
    return df

# Save to Delta table for training
cleaned_df.write.format("delta").saveAsTable("mlops_churn_training")
```

Listing 1: Feature Engineering

9.3 Step 2: Model Training with Data Lineage

The key insight: **Log which data was used to train the model.**

```python
import mlflow

# Load and log the SOURCE DATA
source_dataset = mlflow.data.load_delta(
    table_name="catalog.schema.mlops_churn_training",
    version=latest_version  # Track exact version!
)
mlflow.log_input(source_dataset, context="training")

# Now train model
with mlflow.start_run():
    mlflow.autolog()  # Automatically log params, metrics, model

    model = LGBMClassifier(**params)
    model.fit(X_train, y_train)

    # Model + data lineage now tracked together!
```

Listing 2: Logging Data Lineage (Critical!)

Important: Why Log Data Lineage?

When something goes wrong in production, you need to answer: "Which data and which code produced this model?"
Without data lineage, you can't reproduce the model. MLflow's `log_input()` stores **metadata** (not a copy) linking the model to the exact data version used.

9.4 Step 3: Register to Model Registry

```python
# Search for best run by F1 score
best_run = mlflow.search_runs(
    experiment_ids=[exp_id],
    filter_string="status = 'FINISHED'",
    order_by=["metrics.f1_score DESC"],
    max_results=1
).iloc[0]

# Register to Unity Catalog
model_uri = f"runs:/{best_run.run_id}/model"
mlflow.register_model(model_uri, "catalog.schema.mlops_churn")

# Set alias
client.set_registered_model_alias("mlops_churn", "challenger", 1)
```

Listing 3: Finding Best Model and Registering

9.5 Step 4: Champion-Challenger Validation

```python
# Load challenger model
challenger = mlflow.pyfunc.load_model("models:/mlops_churn@challenger")
challenger_f1 = get_metric(challenger, "f1_score")

# Try to load champion (may not exist yet)
try:
    champion = mlflow.pyfunc.load_model("models:/mlops_churn@champion")
    champion_f1 = get_metric(champion, "f1_score")

    if challenger_f1 > champion_f1:
        print("Challenger wins! Promoting...")
        client.set_registered_model_alias("mlops_churn", "champion", version)
except:
    print("No champion found. Challenger becomes champion.")
    client.set_registered_model_alias("mlops_churn", "champion", version)
```

Listing 4: Promoting Challenger to Champion

9.6 Step 5: Batch Inference

```python
# Load champion model
champion_model = mlflow.pyfunc.load_model("models:/mlops_churn@champion")

# Create Spark UDF for distributed scoring
predict_udf = mlflow.pyfunc.spark_udf(spark, "models:/mlops_churn@champion")

# Score new customer data
inference_df = spark.table("mlops_churn_inference")
scored_df = inference_df.withColumn(
    "prediction",
    predict_udf(*feature_columns)
)

# Result: DataFrame with 'prediction' column (1=churn, 0=stay)
scored_df.write.format("delta").saveAsTable("churn_predictions")
```

Listing 5: Scoring New Data

10 Web Hooks and Automation

MLOps pipelines can be automated using **webhooks**—HTTP callbacks that trigger actions when events occur.

Example: Webhook Use Cases

- **Model registered:** Send Slack notification to ML team
- **Model promoted to staging:** Trigger automated testing pipeline
- **Drift detected:** Trigger retraining job
- **Model promoted to production:** Update documentation

```python
# When a model is registered, trigger testing
@webhook.on("model_registered")
def run_tests(event):
    model_name = event["model_name"]
    version = event["version"]

    # Trigger Databricks job
    databricks_api.run_job(
        job_id="test_model_job",
        parameters={"model": model_name, "version": version}
    )

    # Send Slack notification
    slack.send(f"New model {model_name} v{version} registered!")
```

Listing 6: Webhook Example (Conceptual)

11 Summary: One-Page Quick Reference

MLOps = ML + Dev + Ops

The practice of reliably and repeatedly developing, deploying, and monitoring ML models in production.

The 5 Personas

- **Business Analyst**: Define the problem, set KPIs
- **Data Engineer**: Build data pipelines, ensure quality
- **Data Scientist**: Build and validate models
- **ML Engineer**: Deploy, monitor, scale
- **Governance Officer**: Security and compliance

The 3 Pipelines

- **Training**: Data → Model artifact
- **Inferencing**: Model + New data → Predictions
- **Monitoring**: Track drift, trigger retraining

Data-Centric AI

Improving **data quality** yields more performance gain than tweaking model algorithms.
Good Data = Unambiguous labels + Good coverage + Timely feedback + Right size

Deploy Strategy

Deploy Code (preferred): Copy training code to each environment, retrain with that environment's data

- Better data access control (Prod data stays in Prod)
- Catches production data issues
- Forces modular code

Champion-Challenger Pattern

Don't blindly replace production models. Compare:

- Champion = current production model
- Challenger = new candidate model
- If Challenger F1 > Champion F1 → Promote Challenger

Key MLflow Commands

```
mlflow.log_input(dataset, context="training")  # Data lineage
mlflow.autolog()                                # Auto-log everything
mlflow.register_model(uri, "model_name")        # Register to registry
client.set_registered_model_alias(name, alias, version)  # Set alias
mlflow.pyfunc.load_model("models:/name@alias")  # Load by alias
mlflow.pyfunc.spark_udf(spark, uri)             # Batch scoring
```

Lecture 10: Model Bias, Data Imbalance, and Real-World Fraud Detection

CSCI E-103: Reproducible Data Science and Machine Learning

Harvard University

■ **Course:**	CSCI E-103: Reproducible Data Science
■ **Week:**	Lecture 10
■ **Instructors:**	Anindita Mahapatra & Eric Gieseke
■ **Objective:**	Understand model errors, handle imbalanced data, and learn from a real-world fraud detection system at scale

Contents

1 The Central Theme: Data Problems Lead to Model Problems

Lecture Overview

This lecture explores the critical relationship between data quality and model performance. Poor data doesn't just affect accuracy—it can create biased, unfair, and even harmful models.

Key Topics:

- Machine Learning Errors: Bias, Variance, and Irreducible Error
- The Bias-Variance Tradeoff
- Model Bias: Why models can be unfair
- Class Imbalance: When your data is 99% one thing
- SMOTE: Synthetic Minority Oversampling Technique
- AutoML: Blackbox vs. Glassbox approaches
- Data Classification for PII detection
- Real-World Case Study: Fraud Detection at 15ms latency

2 Machine Learning Errors: The Three Components

Every ML model makes errors. Understanding **why** models err is crucial to improving them.

The Total Prediction Error

Total Error = Bias^2 + Variance + Irreducible Error

2.1 1. Bias (Underfitting)

Definition: Bias

Bias is the error introduced by approximating a real-world problem (which may be extremely complicated) with a too-simple model.
Symptom: The model fails to capture the true relationship between features and target.
Result: Poor performance on **both** training AND test data.

Example: High Bias Model

Imagine you're trying to predict house prices, which depend on many factors in complex, non-linear ways.
If you use simple linear regression (a straight line), you might get:

- Training accuracy: 60%
- Test accuracy: 58%

Both are bad! The model is too simple to capture reality.
Analogy: Trying to describe a curvy mountain road as "mostly flat."

2.2 2. Variance (Overfitting)

Definition: Variance

Variance is the error introduced when a model is too sensitive to small fluctuations (noise) in the training data.
Symptom: The model memorizes the training data instead of learning general patterns.
Result: Excellent performance on training data, **terrible** performance on test data.

Example: High Variance Model

Using an extremely complex model (like a deep neural network with no regularization) on limited data:

- Training accuracy: 99.5%
- Test accuracy: 62%

The model "cheated" by memorizing answers instead of learning rules.
Analogy: A student who memorizes all practice exam answers but can't solve new problems.

2.3 3. Irreducible Error

Definition: Irreducible Error

Irreducible Error is the inherent noise in the data that cannot be reduced regardless of the model. It represents the randomness in real-world phenomena.
This is the floor—no model can be more accurate than the noise allows.

2.4 The Bias-Variance Tradeoff

Important: The Fundamental Tradeoff

Bias and variance are **inversely related**:

- **Simple model** (fewer parameters) → High Bias, Low Variance
- **Complex model** (more parameters) → Low Bias, High Variance

The goal of ML is to find the **"sweet spot"**—a model complex enough to capture the true patterns (low bias) but not so complex that it fits noise (low variance).

Table 1: *Bias vs. Variance Characteristics*

Characteristic	**High Bias (Underfitting)**	**High Variance (Overfitting)**
Model complexity	Too simple	Too complex
Training error	High	Very low
Test error	High	High (much higher than training)
Cause	Not enough learning	Too much learning (noise included)
Fix	More complex model, more features	Regularization, more data, simpler model

3 Model Bias: When AI Becomes Unfair

This is a different kind of "bias"—not underfitting, but **unfairness**. When models systematically disadvantage certain groups of people.

3.1 Why Does Model Bias Happen?

3.1.1 1. Human Cognitive Bias in Data

Models learn from data. If data reflects human prejudices, the model learns those prejudices.

Example: Facial Recognition Accuracy Disparity

A NIST study found that facial recognition systems from major companies (Microsoft, IBM, etc.) had dramatically different accuracy rates:

- **Light-skinned males:** 90%+ accuracy
- **Dark-skinned females:** 60-70% accuracy

Root Cause: The training datasets were **under-represented** in dark-skinned faces, especially females. The model simply didn't have enough examples to learn from.

Key Insight: This isn't an algorithm problem—it's a **data collection** problem. The humans who created the dataset inadvertently (or sometimes systematically) included fewer examples of certain groups.

Example: Microsoft's Tay Chatbot

In 2016, Microsoft released a chatbot named "Tay" that learned from Twitter conversations.
Within 24 hours, malicious users taught it to say racist, sexist, and Holocaust-denying statements.
Lesson: Models are only as good as their training data. Garbage in, garbage out—but magnified.

3.1.2 2. Poor Quality Training Data

- **Low resolution:** Images too blurry to distinguish (like the dog vs. arctic fox example)
- **Mislabeled data:** Human labelers may have their own biases (e.g., associating "evil" with dark colors)
- **Incomplete coverage:** Data missing important edge cases or scenarios

3.2 How to Reduce Model Bias

Strategies for Reducing Bias

1. **Ensure Representative Data (Most Important!):**
   - Audit your training data for demographic representation
   - Actively collect more data from underrepresented groups
   - Consider the source of your data and its inherent biases
2. **Use Appropriate Models:**

- Don't use linear models for non-linear relationships
- Tree-based algorithms are often better at handling bias

3. **Apply Weighting or Penalized Models:**
   - Give more weight to underrepresented groups
   - Use class weights in your loss function
4. **Extensive Hyperparameter Tuning:**
   - Don't just use defaults—optimize for fairness metrics
5. **Ensemble Methods (for reducing variance):**
   - Combine weak and strong learners
   - Random forests, boosting, etc.

4 Class Imbalance: The 99-1 Problem

A special and extremely common case of data problems.

Definition: Class Imbalance

Class Imbalance occurs when one class (label) in your training data is much more frequent than another.

Examples:

- Fraud detection: 99.9% legitimate, 0.1% fraud
- Cancer diagnosis: 98% healthy, 2% cancer
- Anomaly detection in manufacturing
- Network intrusion detection

4.1 The Accuracy Trap

Most ML algorithms optimize for **accuracy**—the percentage of correct predictions.

Why Accuracy is Misleading

Consider a fraud detection dataset with 99.9% legitimate transactions.

A model that **always predicts "legitimate"** will have:

Accuracy = 99.9%

Sounds great, right? But this model catches **zero fraud**. It's completely useless for its intended purpose.

The lesson: **Accuracy is a terrible metric for imbalanced data.**

4.2 Better Metrics: Precision, Recall, and F1

Definition: Confusion Matrix Terminology

- **True Positive (TP):** Model predicted Fraud, actually was Fraud
- **True Negative (TN):** Model predicted Legitimate, actually was Legitimate
- **False Positive (FP):** Model predicted Fraud, but was actually Legitimate (Type I Error)
- **False Negative (FN):** Model predicted Legitimate, but was actually Fraud (Type II Error)

Example: Cancer Screening

In cancer diagnosis, **recall is critical**. Missing an actual cancer case (false negative) can be fatal.

A model with:

- Precision: 90% (some false alarms)
- Recall: 99% (catches almost all cancers)

is far better than:

- Precision: 99% (rarely wrong when it says cancer)
- Recall: 70% (misses 30% of cancers!)

Table 2: *Key Metrics for Imbalanced Data*

Metric	Formula	When to Prioritize
Precision	$\frac{TP}{TP+FP}$	"Of things I flagged as fraud, how many were actually fraud?" Prioritize when **false positives are costly** (blocking legitimate customers)
Recall	$\frac{TP}{TP+FN}$	"Of all actual frauds, how many did I catch?" Prioritize when **false negatives are costly** (missing actual fraud/cancer)
F1 Score	$2 \times \frac{Precision \times Recall}{Precision+Recall}$	Harmonic mean of precision and recall. Use when **both matter**—the primary metric for imbalanced data

4.3 Resampling Techniques

Table 3: *Resampling Strategies*

Technique	How It Works	Pros/Cons
Undersampling	Remove samples from the majority class until balanced	Simple, but **loses valuable data**
Oversampling	Duplicate samples from the minority class	Simple, but can cause **overfitting** (memorizing duplicates)
SMOTE	Generate **synthetic** minority samples	Best of both worlds—creates new data points

4.4 SMOTE: Synthetic Minority Oversampling Technique

How SMOTE Works

SMOTE doesn't just copy existing minority samples—it **creates new synthetic ones**.

Algorithm:

1. Pick a minority class sample x_i
2. Find its k nearest neighbors (also minority class)
3. Randomly select one neighbor x_j
4. Create a new synthetic point **along the line** between x_i and x_j:

$$x_{new} = x_i + \lambda \cdot (x_j - x_i), \quad \text{where } \lambda \in [0, 1]$$

This effectively "fills in" the feature space around minority samples, giving the model a richer understanding of the minority class.

```python
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split

# CRITICAL: Split BEFORE applying SMOTE!
```

```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Apply SMOTE only to training data
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)

# Now train your model
model.fit(X_train_resampled, y_train_resampled)

# Evaluate on original (non-resampled) test data
model.score(X_test, y_test)
```

Listing 1: Using SMOTE in Python

Critical SMOTE Warning

NEVER apply SMOTE before train/test split!
If you SMOTE the entire dataset first, then split, synthetic data may leak between train and test sets, causing artificially inflated test performance (data leakage).

Correct order:

1. Split data into train/test
2. Apply SMOTE to training set only
3. Evaluate on original (un-SMOTEd) test set

5 AutoML: Blackbox vs. Glassbox

Definition: AutoML

AutoML (Automated Machine Learning) automates the process of:

- Feature engineering
- Model selection
- Hyperparameter tuning
- Model evaluation

You provide data, it provides a trained model.

5.1 Blackbox AutoML

- **How it works:** Upload data, click "train," get a model
- **Examples:** DataRobot, some commercial tools
- **Problem:** You can't see how the model was built. When things go wrong (and they will), you can't diagnose or fix them. For enterprise use cases requiring auditability, this is a dealbreaker.

5.2 Glassbox AutoML (Databricks Approach)

- **How it works:** Same automation, but every step is exposed as a **notebook**
- **What you get:**
  1. **Data Exploration Notebook:** Automatic EDA with profiling
  2. **Best Model Notebook:** Full source code for the winning model
  3. **MLflow Integration:** All experiments tracked
  4. **SHAP Explanations:** Feature importance built in
- **Advantage:** "Citizen data scientists" can start quickly, but experts can take over and customize

Table 4: *Models Available in Databricks AutoML*

Classification	Regression	Forecasting
Decision Trees	Decision Trees	Prophet
Random Forests	Random Forests	Auto-ARIMA
Logistic Regression	Linear Regression	DeepAR
XGBoost	XGBoost	
LightGBM	LightGBM	

Example: AutoML Demo Walkthrough

Scenario: Customer churn prediction

Steps:

1. Select dataset (churn table from Unity Catalog)
2. Choose target column ("Churn")

3. Set evaluation metric (F1 Score—because churn is imbalanced!)
4. Set timeout (15 minutes)
5. Click “Start AutoML”

Results:

- AutoML runs Decision Trees, Random Forest, XGBoost, LightGBM in parallel
- **Best model:** LightGBM (based on F1 score)
- Click “View notebook for best model” to see full source code
- All experiments logged to MLflow
- SHAP analysis shows which features matter most

6 Data Classification: Protecting Sensitive Information

Models can inadvertently leak sensitive information (PII). If your training data contains social security numbers and someone asks the model about them...

Definition: PII - Personally Identifiable Information

Information that can identify an individual:

- Name, phone number, email address
- Social Security Number, driver's license
- IP address, location data
- Bank account numbers

PII must be detected and removed before model training.

6.1 Automated Data Classification in Databricks

Unity Catalog can automatically scan tables and detect sensitive columns:

1. Enable "Data Classification" at the catalog level
2. System scans all tables (takes about 15 minutes)
3. Results show which columns contain: Name, Phone, Email, SSN, IP Address, etc.
4. Tags are automatically applied for governance

This prevents accidentally feeding PII into models, which could create both legal liability and privacy leaks.

7 Real-World Case Study: Fraud Detection at Scale

Eric Gieseke shares a real production fraud detection system he built. This is where all the theory meets hard engineering constraints.

7.1 Business Requirements

Table 5: *Fraud Detection System Requirements*

Requirement	Target
Accuracy	High confidence—catch real fraud, don't block legitimate transactions
Latency	**15 milliseconds** (ms) response time
Throughput	50,000+ transactions per second (TPS)
Maintainability	Easy to add/modify features and rules

Example: Why 15ms is Incredibly Hard

For context:

- A typical hard disk seek time: 10-15ms
- A network round-trip: 1-100ms depending on distance
- Human blink: 100-400ms

The entire fraud decision—load customer history, compute features, run ML model, apply rules, return decision—must happen in the time it takes a hard disk to **find** data, let alone read it.

7.2 Features for Fraud Detection

The data science team developed hundreds of features:

Table 6: *Example Fraud Detection Features*

Feature Type	Examples
Transaction-based	Distance from customer's home address
	Difference from average transaction amount
	Number of transactions today
	Time since last transaction
Dimension-based	Merchant's average transaction amount
	Customer's transaction frequency
	Terminal's fraud history
	Geographic region risk score

7.3 Architecture: Lambda Architecture

To meet both 15ms real-time AND large-scale batch requirements, they used **Lambda Architecture**.

Lambda Architecture Overview

Lambda Architecture splits data processing into three layers:

1. Batch Layer (Slow but Complete):

- Stores all historical data (immutable)
- Runs nightly/hourly batch jobs (Hadoop/Spark)
- Computes features on entire dataset (e.g., "customer's average transaction over 1 year")
- Results stored in Feature Store

2. Speed Layer (Fast but Incremental):

- Processes real-time events as they arrive
- Uses Complex Event Processing (CEP)
- Computes real-time features (e.g., "transactions in last 10 minutes")

3. Serving Layer (Query Layer):

- Combines batch and speed results
- Serves queries with low latency
- Cassandra used for this layer

7.4 Key Innovation 1: Metadata-Driven Code Generation

- **Problem:** Hundreds of features need code for both batch (SQL) AND real-time (CEP language). Writing and maintaining both is error-prone.
- **Solution:** Define features as **metadata**, not code.

```
Feature: customer_avg_transaction_30d
Type: Average
Field: transaction_amount
Window: 30 days
Dimension: customer_id
```

- **Code Generator** reads metadata and automatically produces:
  - SQL for batch processing (Spark)
  - EPL for real-time processing (CEP)
- **Benefit:** Add a new feature by editing metadata, not writing two sets of code

7.5 Key Innovation 2: Circular Buffer (Ring Buffer)

Example: The Problem

To compute "customer's average transaction amount over last 7 days," you'd normally:

1. Query database for all transactions in last 7 days
2. Sum them up
3. Divide by count

For millions of customers, this is **impossible in 15ms**.

Circular Buffer Solution

Instead of storing individual transactions, store **aggregates per time bucket**.

Structure (7-day buffer):

Sun	Mon	Tue	Wed	Thu	Fri	Sat
(count, sum)	(count, sum)	(count, sum)	(count, sum)	(count, sum)	(count, sum)	(count, sum)

Example:

```
Sun: (3, $150)   Mon: (5, $280)   Tue: (2, $90)   ...
```

To compute 7-day average:

$$\text{Average} = \frac{\sum \text{sums}}{\sum \text{counts}} = \frac{150 + 280 + 90 + \ldots}{3 + 5 + 2 + \ldots}$$

Memory footprint: Just 14 numbers per customer per feature (2 numbers × 7 days)

When day changes: Overwrite oldest bucket with zeros, start fresh

This was novel enough that the company **patented** it.

7.6 Key Innovation 3: Cassandra as Feature Store

- **Why Cassandra?**
  - Write speed: Extremely fast (writes are "fire and forget")
  - **Read speed: 2.3ms achieved**—critical for 15ms budget
  - Scalable to petabytes
  - Supports wide rows (billions of columns per row)
- **Data Model:**
  - Only 2 tables: **Events** (fact table) and **Dimensions**
  - New features = new columns (schema-less flexibility)
  - Sparse data handled efficiently

7.7 System Flow Summary

1. **Fraud Analyst** defines new feature in Metadata Service
2. **Batch Processing** (nightly) computes historical feature values → stores in Cassandra
3. **Code Generator** creates real-time CEP code
4. **Customer** swipes card at merchant
5. **Payment Service** calls Fraud Detection Service
6. **Fraud Detection** (within 15ms):
   (a) Retrieves pre-computed features from Cassandra (2.3ms)
   (b) Computes real-time features using Circular Buffers (in-memory)
   (c) Runs ML model + rules

   (d) Returns “Approve” or “Decline”

7. **Customer** completes purchase (or doesn’t)

7.8 Disaster Recovery

For mission-critical payment systems:

- Two data centers (US and Europe)
- If US fails, traffic automatically routes to Europe
- Slightly higher latency acceptable vs. complete outage
- Data replication between centers

8 Summary: One-Page Quick Reference

ML Errors: Bias vs. Variance

- **Bias (Underfitting):** Model too simple. Both train and test error high.
- **Variance (Overfitting):** Model too complex. Train error low, test error high.
- **Tradeoff:** Simple ↔ Complex. Find the sweet spot.

Model Bias (Fairness)

- **Cause 1:** Human bias in data (under-representation of groups)
- **Cause 2:** Poor quality labels, mislabeled data
- **Fix:** Ensure representative, diverse training data (most important!)

Class Imbalance (99-1 Problem)

- **Problem:** Accuracy is meaningless (99% by always predicting majority)
- **Metrics:** Use **F1 Score** (harmonic mean of Precision and Recall)
- **Fix: SMOTE** (create synthetic minority samples)
- **Warning:** Apply SMOTE to training set **only**, after train/test split!

AutoML: Blackbox vs. Glassbox

- **Blackbox:** Magic model, no visibility (enterprise unfriendly)
- **Glassbox:** Full notebooks, MLflow tracking, SHAP explanations
- **Use case:** Quick baseline, data validation, citizen data scientists

Fraud Detection at 15ms

- **Architecture:** Lambda (Batch + Speed + Serving layers)
- **Key 1: Circular Buffer** - Store (count, sum) per time bucket, not individual records
- **Key 2: Cassandra** - 2.3ms reads for feature retrieval
- **Key 3: Code Generation** - Define features as metadata, auto-generate SQL/CEP
- **Lesson:** The difference between 15ms and 100ms is the difference between possible and impossible

■ **Course:** CSCI E-103: Reproducible Machine Learning

■ **Week:** Lecture 11

■ **Instructors:** Anindita Mahapatra & Eric Gieseke

■ **Objective:** Master the fundamentals of Large Language Models (LLMs), Retrieval Augmented Generation (RAG), and AI Agents for enterprise applications

Contents

1 Introduction: The LLM Revolution

This lecture covers one of the most transformative technologies in modern computing: **Large Language Models (LLMs)** and **AI Agents**. We'll explore how these technologies have evolved, how to deploy them effectively in enterprise settings, and how to mitigate their inherent risks.

Lecture Overview

Why This Matters:

- LLMs are reaching a **tipping point** in capability and accessibility
- **AGI (Artificial General Intelligence)** predictions suggest human-level AI could arrive as soon as 2026
- Every company will become a **data and AI company** first, with their core business layered on top
- Understanding LLM deployment is now an essential skill for data engineers and ML practitioners

1.1 Learning Objectives

By the end of this lecture, you will be able to:

1. **Understand** the evolution from rule-based systems to Generative AI
2. **Explain** key LLM terminology (RAG, Fine-tuning, Embeddings, Hallucination)
3. **Design** LLM deployment strategies based on cost-quality tradeoffs
4. **Implement** RAG systems for enterprise applications
5. **Build** AI agents using Databricks tools (Genie, Agent Bricks, AI Gateway)
6. **Manage** LLM risks including hallucinations, bias, and security vulnerabilities

2 The Evolution of AI: From Rules to Generation

2.1 The AI Technology Stack

AI technologies form a hierarchical relationship, with each layer building upon the previous:

Layer	Description
Artificial Intelligence	The broadest category: any system that mimics human intelligence
Machine Learning	Systems that learn from data without explicit programming
Deep Learning	Neural networks with multiple layers (inspired by the human brain)
Generative AI	AI that creates *new* content rather than just classifying existing data

Key Information

Key Distinction - Traditional AI vs. Generative AI:

- **Traditional AI:** "Is this email spam or not spam?" (classification)
- **Generative AI:** "Write me a professional email to decline a meeting" (creation)

Generative AI goes beyond analysis—it produces *new* data based on patterns learned from existing data.

2.2 Types of Generative AI Models

Definition:

LLM (Large Language Model) A neural network trained on vast amounts of text data to understand and generate human language. Examples include GPT-4, Claude, Gemini, and Llama.

Analogy: Think of an LLM as a student who has read every book in the world's largest library and can now discuss any topic or write on any subject.

Definition:

GAN (Generative Adversarial Network) A model architecture primarily used for generating images and videos. Used in applications like:

- **Deepfakes:** Creating realistic fake videos of people
- **Style Transfer:** Transforming photos into paintings
- **Image Generation:** Creating photorealistic images from scratch

Definition:

Diffusion Models Advanced generative models that create content by gradually removing noise from random data. Powers:

- **DALL-E:** Text-to-image generation
- **Stable Diffusion:** Open-source image generation
- **Video Generation:** Creating lip-synced avatars

3 LLM Capabilities and Enterprise Use Cases

3.1 What Can LLMs Do?

LLMs have fundamentally changed what's possible with AI:

1. **Democratize Knowledge Access**
   - All human knowledge encoded in a single model
   - Accessible in any language (multilingual capabilities)
   - Available to anyone with internet access
2. **Process Unstructured Data**
   - Transcribe video/audio to text

- Summarize documents, meeting minutes, patents
- Extract structured data from free-form text

3. **Improve Knowledge Worker Productivity**
   - Code generation and debugging
   - Legal document review
   - Customer feedback analysis
   - Marketing trend identification
4. **Enable Self-Improvement**
   - Models can evaluate their own outputs
   - Tune product recommendations based on feedback
   - Generate training data for other models

Example:

Real-World LLM Applications (Demo) The lecture demonstrated a multilingual AI assistant with multiple capabilities:

Features Demonstrated:

- **Voice Interaction:** Users speak questions in natural language
- **Multilingual Response:** System answers in Russian, Spanish, etc.
- **RAG Integration:** Answers based on specific business knowledge base
- **Lip-Synced Avatar:** Diffusion model generates realistic speaking animation
- **E-commerce Search:** ”blue sarees” search understands intent despite misspelling
- **Virtual Try-On:** Diffusion model shows how clothes look on uploaded photos

3.2 The Urgency to Adopt LLMs

Important:

Why Companies Must Move Fast

1. **Competitive Pressure:** Early adopters gain significant advantages
2. **Accessibility:** LLMs are now economical enough for any business
3. **Integration Need:** Must connect LLMs to existing company data
4. **Security Requirements:** Need to customize and secure AI for enterprise use

4 Essential LLM Terminology

Understanding these terms is crucial for working with modern AI systems:

5 LLM Deployment Strategies: The Maturity Curve

Choosing the right deployment approach depends on your use case, budget, and accuracy requirements.

Term	**Definition and Example**
Hallucination	When the model confidently generates false information. *Example:* Generating fake URLs that look real but don't exist
Temperature	Controls randomness in outputs (0 = deterministic, 1 = creative) Setting to 0 reduces hallucinations but may limit creativity
Grounding	Connecting AI responses to real-world facts and data Ensures answers are based on verifiable information
Prompt Engineering	Crafting instructions to get desired behavior from LLMs *Example:* "You are a helpful restaurant server..."
Zero-Shot Learning	Asking the model to perform a task with just a prompt No examples provided—relies entirely on pre-training
Few-Shot Learning	Providing examples in the prompt to guide the model Generally improves accuracy over zero-shot
Chain of Thought	Instructing the model to show its reasoning steps Improves accuracy and interpretability
Modality	Type of data: text, image, audio, or video **Multimodal** models handle multiple types (e.g., GPT-4o)
Transformer	Neural network architecture underlying modern LLMs Components: Encoder, Decoder, Embeddings
RLHF	Reinforcement Learning from Human Feedback Humans rate outputs to improve model behavior

Table 1: *Essential LLM Terminology*

5.1 The Four Levels of LLM Customization

LLM Deployment Strategies (Easy to Hard)

Level 1: Prompt Engineering

- **What:** Craft careful instructions for existing models
- **Cost:** Very low (API costs only)
- **Data Required:** None
- **Quality:** Basic—limited by model's training cutoff
- **Best For:** General tasks, quick prototypes

Level 2: RAG (Retrieval Augmented Generation) — RECOMMENDED

- **What:** Search your data, feed relevant chunks to the model

Warning

Key Insight: Most organizations should start with RAG, not fine-tuning or pre-training. RAG provides the best cost-quality tradeoff and allows you to:

- Include the latest information (no training cutoff)
- Reduce hallucinations (answers grounded in your documents)
- Maintain data privacy (your data stays in your database)
- Update knowledge easily (just update the vector database)

6 RAG: Retrieval Augmented Generation

RAG is the most cost-effective and practical approach to customizing LLMs for enterprise use.

6.1 Why RAG Is Necessary

LLMs have two fundamental limitations:

1. **Training Cutoff:** Models don't know anything after their training date
   - GPT-4 doesn't know who won yesterday's election
   - Can't answer questions about your company's latest policy
2. **No Access to Private Data:** Models never saw your internal documents
   - Can't answer "What's our return policy?"
   - Can't help with company-specific procedures

Key Information

The RAG Solution:
RAG gives the LLM an "open book test"—instead of relying solely on memorized knowledge, we search relevant documents and include them in the prompt. The model then answers based on this retrieved context.

6.2 How RAG Works: The Complete Pipeline

Definition:

RAG Architecture **Phase 1: Data Ingestion (Offline, One-Time Setup)**

1. **Document Collection:** Gather all relevant documents (PDFs, manuals, policies)
2. **Chunking:** Split documents into smaller pieces (800-1200 characters optimal)
3. **Embedding:** Convert text chunks into numerical vectors (embeddings)
4. **Storage:** Store embeddings in a Vector Database (Pinecone, Chroma, etc.)

Phase 2: Query Processing (Real-Time)

1. **User Query:** User asks a question
2. **Query Embedding:** Convert the question to a vector

3. **Similarity Search:** Find document chunks with similar vectors
4. **Context Augmentation:** Add retrieved chunks to the prompt
5. **LLM Generation:** Model generates answer using retrieved context
6. **Response:** Return answer to user (optionally with source citations)

Example:

RAG Query Flow **User Question:** ”What is our company’s remote work policy?”
Behind the Scenes:

1. Question converted to vector: [0.23, -0.45, 0.67, ...]
2. Vector DB searched for similar vectors
3. Top 3 most relevant document chunks retrieved:
   - HR Policy Manual, Section 4.2 (similarity: 0.92)
   - Employee Handbook, Chapter 7 (similarity: 0.88)
   - COVID-19 Work Guidelines (similarity: 0.85)
4. Prompt sent to LLM:

```
Based on the following documents, answer the user's question.

Documents:
[Retrieved chunks here...]

Question: What is our company's remote work policy?
```

5. LLM generates response grounded in the documents

6.3 RAG Best Practices

Parameter	**Recommendation**
Chunk Size	800-1200 characters (optimal for semantic coherence)
Chunk Overlap	100-200 characters (ensures context isn’t lost at boundaries)
Number of Retrieved Chunks	3-5 (balance between context and token limits)
Embedding Model	OpenAI Ada, Sentence Transformers, Cohere Embed
Vector Database	Pinecone, Chroma, Milvus, PostgreSQL with pgvector

Table 2: *RAG Configuration Best Practices*

Important:

Common RAG Pitfall: Version Management **Problem:** When policies are updated, old versions remain in the vector database. The LLM might retrieve outdated information.

Solutions:

- Replace entire documents when updating (full CRUD support)
- Add metadata timestamps and filter by date
- Include version numbers in document content
- Periodic reindexing to remove stale content

7 Databricks AI Platform: Tools and Capabilities

Databricks provides an integrated platform for building, deploying, and managing LLM applications.

7.1 Platform Architecture Overview

Component	**Purpose**
Unity Catalog	Governance for all assets (tables, models, functions, vector indices)
Delta Sharing	Zero-copy data sharing without ETL
Feature Store	Store and serve ML features
Model Registry	Version and manage ML models (part of MLflow)
Vector Search	Managed vector database for RAG applications
AI Gateway	Central entry point for all models with security, logging, rate limiting
Lakehouse Apps	Interactive applications built on top of data and models

7.2 Genie: Natural Language to SQL

Definition:

Genie Genie is an agentic BI tool that allows users to query structured data using natural language.

How It Works:

1. User asks: "What were our top 5 products by sales last month?"
2. Genie examines table metadata (schemas, column names)
3. Generates SQL: `SELECT name, SUM(sales) FROM products...`
4. Executes query and returns results in natural language

Key Advantage: Because Genie generates SQL from metadata (not data), it hallucinates less than general LLMs. The schema is deterministic—a column either exists or it doesn't.

```sql
-- User Question: "Show average premium by occupation and risk level"
-- Genie generates:
```

```sql
SELECT
    occupation_type,
    risk_level,
    AVG(premium_amount) as avg_premium
FROM insurance_policies
GROUP BY occupation_type, risk_level
ORDER BY avg_premium DESC;
```

Listing 1: Genie-Generated SQL Example

7.3 Agent Bricks: No-Code RAG Agent Builder

Definition:

Agent Bricks Agent Bricks is like AutoML for RAG agents—build production-ready chatbots without coding.

Steps to Create a Knowledge Assistant:

1. Upload documents to Unity Catalog Volume
2. Select the volume path in Agent Bricks
3. Provide a name and description
4. Click "Create Agent"
5. System automatically: chunks documents, creates embeddings, builds vector index

Use Cases:

- Customer support chatbots
- Internal policy assistants
- Technical documentation search
- HR FAQ systems

7.4 AI Functions: SQL-Native AI

Execute LLM capabilities directly within SQL queries:

```sql
-- Analyze sentiment and summarize customer reviews
SELECT
    review_id,
    review_text,
    ai_analyze_sentiment(review_text) AS sentiment,
    ai_summarize(review_text, 20) AS summary
FROM customer_reviews
WHERE created_date > '2025-01-01';

-- Extract specific information from text
SELECT
    ai_extract(contract_text, 'termination_clause') AS termination_clause,
    ai_extract(contract_text, 'payment_terms') AS payment_terms
FROM legal_contracts;
```

Listing 2: AI Functions in SQL

7.5 AI Gateway: Central Model Management

Key Information

AI Gateway provides a unified entry point for all model interactions:

- **Rate Throttling:** Prevent API abuse and control costs
- **Credential Management:** Centralized API key handling
- **Payload Logging:** Record all requests/responses for auditing
- **A/B Testing:** Route traffic between model versions
- **Guardrails:** Content filtering and safety checks

Analogy: AI Gateway is like Amazon API Gateway, but specifically designed for ML model endpoints.

7.6 Unity Catalog Functions as Agent Tools

You can register SQL functions in Unity Catalog and use them as tools for AI agents:

```sql
-- Create a function that agents can call
CREATE FUNCTION catalog.schema.get_client_quote(client_name STRING)
RETURNS TABLE (
    quote_id STRING,
    client STRING,
    coverage DECIMAL,
    premium_estimate DECIMAL,
    product_option STRING,
    health_rating STRING
)
COMMENT 'Returns all quote details for a specified client'
AS
SELECT
    quote_id, client, coverage,
    premium_estimate, product_option, health_rating
FROM insurance_quotes
WHERE client = client_name;
```

Listing 3: UC Function as Agent Tool

Example:

Multi-Agent Supervisor The most advanced pattern is a **Multi-Agent Supervisor** that orchestrates multiple specialized agents:

Architecture:

1. **Supervisor Agent:** Understands user intent and routes requests

2. **Genie Agent:** Handles structured data queries (SQL)
3. **Knowledge Assistant:** Handles unstructured data (RAG)
4. **UC Functions:** Execute specific business logic
5. **External MCP Servers:** Connect to external services

Example Flow:

- User: ”What’s the risk score for John Smith and explain our underwriting policy?”
- Supervisor: Routes ”risk score” to Genie, ”underwriting policy” to Knowledge Assistant
- Both agents execute in parallel
- Responses combined and returned to user

8 LLM Risks, Limitations, and Mitigation

With great power comes great responsibility. LLMs introduce new categories of risk that must be actively managed.

8.1 Hallucination: The Confident Liar

Definition:

Hallucination When an LLM generates plausible-sounding but factually incorrect information with complete confidence.

Why It Happens:

- LLMs are trained to generate *fluent* responses, not necessarily *true* ones
- Models are rewarded for providing answers, even when they should say ”I don’t know”
- Pattern matching can produce text that looks correct but isn’t

Example: When asked to generate image URLs, ChatGPT confidently produces URLs that look legitimate but lead to 404 errors.

Warning

Can Hallucination Be Eliminated?

No—hallucination is intrinsic to the generative nature of LLMs. It’s like asking if floating-point rounding errors can be eliminated. However, it can be **significantly reduced**:

- Set temperature to 0 (most deterministic output)
- Use RAG to ground responses in verified documents
- Implement chain-of-thought prompting for reasoning transparency
- Add guardrail models to verify outputs
- Design prompts that explicitly allow ”I don’t know” responses

8.2 Bias: Reflections of Training Data

- **Source:** Models inherit biases present in their training data

- **Example:** A model trained primarily on US medical data may give incorrect recommendations for diseases more prevalent in other regions
- **Impact:** Can perpetuate discrimination, exclusion, and unfair treatment

Mitigation Strategies:

1. Ensure training data diversity
2. Explicitly document known limitations
3. Implement fairness testing and auditing
4. Use human review for high-stakes decisions

8.3 Security Threats

Threat	**Description**	**Mitigation**
Prompt Injection	Malicious inputs that trick the model into ignoring instructions	Input validation, prompt sanitization
Jailbreaking	Bypassing safety guardrails to generate harmful content	Multiple layers of content filtering
Data Exfiltration	Model reveals sensitive training data	Data sanitization, PII masking
Multilingual Attacks	Using non-English prompts to bypass safety filters	Multilingual safety testing

Table 3: *LLM Security Threats and Mitigations*

8.4 Privacy and Regulatory Compliance

Important:

EU AI Act Europe's **EU AI Act** mandates disclosure of:

- Training data sources
- Compute resources used
- Model deployment details
- Known limitations and biases

Implication: Significant documentation overhead, but ensures responsible AI development.

8.5 Ethical Considerations and ESG

Definition:

ESG (Environmental, Social, Governance) Companies are increasingly evaluated on their ESG scores:

- **Environmental (E):** LLM training consumes enormous energy. A single GPT-4 training run can emit hundreds of tons of CO2.

- **Social (S):** AI systems that discriminate or generate toxic content harm society.
- **Governance (G):** Proper oversight, audit trails, and accountability mechanisms.

Key Insight: Laws can be enforced, but ethics must be culturally adopted. As AI practitioners, we have a collective moral responsibility.

9 The Future: AGI, Superintelligence, and Societal Impact

9.1 The Path to AGI

Definition:

AGI and Superintelligence

- **AGI (Artificial General Intelligence):** AI as capable as humans across all cognitive tasks. Predictions suggest this could arrive as early as 2026.
- **Superintelligence:** AI that surpasses human intelligence by orders of magnitude. Once achieved, could rapidly self-improve.

Warning

Book Recommendation: "Superintelligence" by Nick Bostrom

This book explores what happens when AI becomes smarter than humans:

- The "alignment problem": ensuring AI goals match human values
- Existential risks from uncontrolled AI development
- The importance of safety research before capability research

As practitioners building these systems, we have a responsibility to understand these risks.

9.2 Impact on Employment

Key Information

The Changing Job Landscape:

- **Entry-level jobs:** Most at risk—routine cognitive tasks easily automated
- **Experienced practitioners:** Demand increasing—someone must build and maintain AI systems
- **New roles emerging:** Prompt engineers, AI ethics officers, AI trainers

The Paradox: If entry-level jobs disappear, how do workers gain experience to become senior practitioners? This is a fundamental societal challenge we must address.

9.3 AI as a Democratizing Force

Despite risks, LLMs also offer tremendous positive potential:

1. **Education:** AI tutors available 24/7, personalized to each student (Khan Academy's vision)
2. **Knowledge Transfer:** Bridging the gap between retiring domain experts and new workers

3. **Global Access:** Multilingual capabilities enable knowledge access regardless of native language
4. **Government Efficiency:** Albania using LLMs to reduce corruption in government contracting
5. **Healthcare:** Democratizing access to medical knowledge in underserved regions

10 AI Maturity Curve: Building Enterprise AI Capability

10.1 The Progression from BI to AI

Stage	Description	Complexity
BI/Dashboards	Traditional reporting: what happened?	Low
AI Functions	SQL-native AI: sentiment, summarization	Medium-Low
Knowledge Agents	RAG-based Q&A on documents	Medium
Multi-Tool Agents	Agents that call multiple functions/APIs	Medium-High
Multi-Agent Orchestration	Supervisor routing to specialized agents	High

10.2 Use Case Selection: Start Smart

Important:

Selecting Your First LLM Use Case Your first LLM project determines organizational adoption momentum. Choose wisely:

Ideal First Use Case:

- **High Feasibility:** Technical complexity is manageable
- **High Value:** Clear ROI that stakeholders can see
- **Low Risk:** Mistakes don't cause major harm
- **Reusable:** Learnings apply to future projects

Bad First Use Case:

- The ”hottest” or most ambitious idea
- Customer-facing without extensive testing
- Mission-critical with no fallback

10.3 Key Principles for Enterprise AI

1. **Every company will be a data and AI company first**—core business becomes secondary
2. **Differentiation comes from your data**, not the model—everyone can use ChatGPT
3. **Many small, purpose-built models** often beat one giant model
4. **Models are improving and getting cheaper**—what's impossible today may be trivial tomorrow
5. **Benefits outweigh risks**, but only with proper guardrails

11 Practical Lab Guide: Building Agents in Databricks

11.1 Lab Overview

The lecture included hands-on demonstrations of building three types of agents:

1. **Genie Space:** Natural language queries on structured insurance data
2. **Knowledge Assistant:** RAG agent for underwriting guidelines PDF
3. **Multi-Agent Supervisor:** Orchestrating multiple specialized agents

11.2 Step-by-Step: Creating a Knowledge Assistant

```python
# Step 1: Get catalog and schema from widgets
catalog_name = dbutils.widgets.get("catalog_name")
schema_name = dbutils.widgets.get("schema_name")

# Step 2: Create schema if not exists
spark.sql(f"CREATE SCHEMA IF NOT EXISTS {catalog_name}.{schema_name}")

# Step 3: Upload documents to Volume
# Documents should be in: /Volumes/{catalog}/{schema}/{volume_name}/

# Step 4: Create tables and functions
spark.sql(f"""
CREATE TABLE IF NOT EXISTS {catalog_name}.{schema_name}.insurance_quotes (
    quote_id STRING,
    client STRING,
    coverage DECIMAL(10,2),
    premium_estimate DECIMAL(10,2),
    product_option STRING,
    health_rating STRING
)
""")
```

Listing 4: Setting Up the Data

11.3 Agent Testing in Playground

Once your agent is created, use the Playground for testing:

1. Navigate to the agent in the Databricks UI
2. Click "Open in Playground"
3. Enter test questions
4. Review:
   - Response content and accuracy
   - Token usage and latency
   - Source citations (for RAG)

- Chain of thought reasoning (if enabled)

5. Enable ”AI Judge” for automated evaluation
6. Compare responses across different models

Example:

Playground Testing **Question:** ”How are applications categorized into preferred plus, preferred, standard, or substandard?”
Agent Response: ”The system uses a multi-tier risk classification system to categorize applicants based on their overall risk profile...”
Citations:

- Page 1: Risk Classification Criteria
- Page 1: Underwriting Process Flow

Metrics:

- Tokens used: 847
- Latency: 2.3 seconds
- AI Judge score: 4.5/5

12 Quick Summary: One-Page Review

Key Summary

Key Takeaways from Lecture 11:

1. **AI Evolution:** Rule-based → ML → Deep Learning → **Generative AI (LLMs)**
2. **LLM Deployment Strategy:** Start with **RAG** (not fine-tuning or pre-training)
   - Most cost-effective approach
   - Grounds responses in your data
   - Reduces hallucinations
   - Easy to update knowledge
3. **RAG Pipeline:** Chunk → Embed → Store in Vector DB → Retrieve → Generate
4. **Databricks Tools:**
   - **Genie:** Structured data (SQL), high accuracy
   - **Agent Bricks:** Unstructured data (PDF), rapid prototyping
   - **AI Gateway:** Security, logging, rate limiting
   - **AI Functions:** SQL-native AI capabilities
   - **UC Functions:** Custom tools for agents
5. **Critical Risks:**
   - Hallucination (false confidence)
   - Bias (discriminatory outputs)
   - Security (prompt injection, data leakage)

- Privacy (PII exposure)

6. **Mitigation Strategies:**
   - RAG for grounding
   - Temperature = 0 for determinism
   - Guardrail models for verification
   - Human-in-the-loop for high stakes
   - RLHF for continuous improvement
7. **Future Outlook:**
   - AGI potentially by 2026
   - Every company becomes AI-first
   - Differentiation through proprietary data
   - Ethical AI is mandatory, not optional

13 Frequently Asked Questions (From Class Discussion)

Q: Are hallucinations intrinsic to LLMs, or can they be eliminated?

A: Hallucinations are inherent to the generative nature of LLMs. The models are trained to produce fluent outputs, not necessarily true ones. While they can be significantly reduced (RAG, temperature=0, careful prompting), they likely cannot be completely eliminated. Think of it like irreducible error in traditional ML—some error will always remain.

Q: How do we handle document versioning in RAG?

A: This is a complex problem. If you have Policy v1, v2, and v3 all indexed, the model might retrieve outdated information. Solutions:

- Replace entire documents when updating (full CRUD)
- Add timestamp metadata and filter by recency
- Use relationship graphs to link document versions
- Explicitly include version numbers in your documents

Q: Why does ChatGPT seem to ”remember” me across sessions?

A: Modern LLMs like ChatGPT store conversation histories and can summarize past interactions. They don’t have true memory, but they can retrieve and reference previous conversations to provide more personalized responses. This feels like memory but is actually sophisticated context retrieval.

Q: What are we doing to prevent prompt injection and jailbreaks?

A: This is the ”alignment problem”—ensuring models behave as intended. Current approaches include:

- Guardrails and content filtering
- Input validation and sanitization

- Using larger models as judges/guards
- RLHF to train models to refuse malicious requests
- Regulatory frameworks (EU AI Act)

As practitioners, we have a responsibility to implement these safeguards and advocate for responsible AI development.

■ **Course:**	CSCI E-103: Reproducible Machine Learning
■ **Week:**	Lecture 12
■ **Instructors:**	Ram Sriharsha (Guest Speaker)
■ **Objective:**	Master the fundamentals of Data and AI Governance, including information governance frameworks, Unity Catalog implementation, and attribute-based access control (ABAC)

Contents

1 Introduction: Why Governance Matters

This lecture covers **Data and AI Governance**—the frameworks, policies, and technologies that enable organizations to maximize the value of their data assets while minimizing security and compliance risks.

Lecture Overview

Key Learning Objectives:

- **Understand** the hierarchy: Information Governance $\supset$ Data Governance $\supset$ AI Governance
- **Learn** the four pillars: Policies, Procedures, Standards, and Controls
- **Implement** governance using Databricks Unity Catalog
- **Apply** Attribute-Based Access Control (ABAC) for fine-grained security
- **Monitor** data quality and track data lineage automatically

Key Information

Why Should You Care?
"Data is the new oil"—but unlike oil, data is **renewable and limitless**. Organizations constantly produce, collect, and process data. The challenge is:

- **Maximize Value:** Make data accessible to drive business decisions
- **Minimize Risk:** Protect sensitive data from breaches and regulatory violations

These two goals are in constant tension. Governance is the art of balancing them.

2 Information Governance: The Big Picture

2.1 What Is Information Governance?

Definition:

Information Governance Information Governance is the overarching framework that encompasses **all** organizational information—physical documents, digital files, knowledge assets, and AI models.
Scope: Anything that is created, collected, stored, processed, or shared.
Examples:

- Locking your laptop when leaving your desk
- Shredding printed documents with sensitive information
- Policies about who can access which email folders

2.2 The Governance Hierarchy

Information Governance contains two major subsets:

Type	**Focus Area**
Data Governance	Managing digital data: quality, security, lifecycle, access control
AI Governance	Managing AI models: bias prevention, explainability, ethical use, training data lineage

Warning

You Can't Walk Before You Run:
AI Governance is **built on top of** Data Governance. If your data is messy, inconsistent, or poorly secured, you cannot build fair, transparent, and safe AI systems.
Example: If you don't know the lineage of your training data, how can you verify the model isn't trained on copyrighted or biased content?

2.3 Goals of Information Governance

1. **Maximize Information Value**
   - Make data discoverable and accessible to authorized users
   - Enable self-service analytics and AI development
2. **Mitigate Risk**
   - Prevent data breaches (reputational and financial damage)
   - Ensure regulatory compliance (GDPR, HIPAA, PCI-DSS, CCPA)
3. **Enhance Security**
   - Implement multi-factor authentication
   - Encrypt data at rest and in transit
4. **Optimize Lifecycle Management**
   - Define data retention policies
   - Archive or delete data when no longer needed
5. **Promote Transparency and Accountability**
   - Clear roles: Data Stewards, Governance Officers, Catalog Owners
   - Audit trails for all data access and modifications

Pillar	**Definition**	**Example (GDPR)**
Policies	High-level rules established by leadership or regulators	”Individuals have the right to request deletion of their personal data”
Procedures	Step-by-step instructions for implementing policies	”Process data deletion requests within 45 days”
Standards	Technical specifications and best practices	”Use AES-256 encryption; minimize data storage”
Controls	Mechanisms that enforce policies	”Multi-factor authentication; access logs; role-based permissions”

Table 1: *The Four Pillars of Governance*

3 The Four Pillars: Policies, Procedures, Standards, Controls

3.1 Framework Overview

3.2 How They Work Together

Example:

GDPR Compliance Flow **Policy:** GDPR mandates the ”Right to Be Forgotten”

Procedure:

1. User submits deletion request
2. Request logged in ticketing system
3. Data team identifies all user data across systems
4. Data deleted or anonymized within 30 days
5. Confirmation sent to user

Standards:

- Data must be encrypted at rest
- Access controlled via Role-Based Access Control (RBAC)
- Minimum data retention periods defined

Controls:

- Technical: Encryption, access control lists
- Administrative: Background checks for data handlers
- Physical: Secure data centers

4 Data Governance vs AI Governance

4.1 Data Governance

Definition:

Data Governance Data Governance focuses on managing **digital data assets**—their quality, security, lifecycle, and accessibility.

Key Questions:

- Who can access this dataset?
- Is this data encrypted?
- How long should we retain this data?
- Is the data accurate and up-to-date?

4.2 AI Governance

Definition:

AI Governance AI Governance extends data governance to **AI models and their outputs**.

Key Questions:

- Is the training data free of bias?
- Can the model's decisions be explained?
- Was the training data legally obtained?
- Does the model produce fair outcomes across demographic groups?

Important:

AI Governance is Critical Now With the rise of LLMs trained on internet-scale data:

- **Copyright concerns:** Was copyrighted material used for training?
- **Privacy violations:** Does the model memorize PII from training data?
- **Indemnification:** If you use an LLM to generate content that infringes IP, who is liable?

Organizations are increasingly asking: ”What is our legal exposure when using third-party AI models?”

5 Governance Maturity Model

Organizations progress through maturity levels. **You cannot skip levels**—each stage builds on the previous.

Key Information

The Goal for Most Organizations: Level 3 (Defined/Proactive)

At this level, you have:

- Documented policies and procedures

Level	Stage	Characteristics
1	**Initial/Aware**	No formal governance. Individuals manage data ad-hoc.
2	**Reactive/Managed**	Problems trigger responses. Some documentation exists.
3	**Defined/Proactive**	Enterprise-wide standards established. Most organizations target this.
4	**Quantified**	Governance effectiveness measured with metrics.
5	**Optimized**	Automated detection and remediation. Continuous improvement.

Table 2: *Data Governance Maturity Model*

- Clear roles and responsibilities
- Automated access controls
- Regular audits and compliance checks

6 Governance Operating Models

How governance is implemented depends on organizational culture and structure.

Model	Characteristics	Pros	Cons/Best For
Centralized	Central team controls all governance	High consistency, strong security	Slow, bottlenecks (regulated industries)
Decentralized	Each department self-governs	Fast innovation, flexibility	No standards, duplication (startups)
Federated	Central guidelines + local execution	Balance of control and autonomy	Coordination challenges (enterprises)
Hybrid	Core data centralized, rest decentralized	Protects sensitive data + efficiency	Complex structure

Table 3: *Governance Operating Models*

Example:

Federated Model in Practice **Central Governance Office:**

- Defines global policies (e.g., ”All PII must be encrypted”)
- Maintains the enterprise data catalog
- Conducts compliance audits

Business Unit Data Stewards:

- Implement policies within their domain
- Define local schemas and data quality rules
- Grant access to their datasets

7 Databricks Unity Catalog: Implementation

Unity Catalog (UC) is Databricks' unified governance solution for all data and AI assets.

7.1 The Three-Level Hierarchy

`Metastore` → `Catalog` → `Schema` → `Table / Volume / Model / Function`

Level	Description
Metastore	Top-level container (typically one per cloud region). Stores all metadata.
Catalog	Largest grouping of data assets. Examples: `prod`, `dev`, `hr_data`
Schema	Logical grouping within a catalog (equivalent to a database)
Table/Volume	Actual data. Tables = structured; Volumes = unstructured files

Table 4: *Unity Catalog Hierarchy*

7.2 Managed vs External Tables

Type	Managed Table	External Table
Storage	Databricks manages location	You specify cloud storage path
Lifecycle	DROP TABLE deletes data files	DROP TABLE removes metadata only
Use Case	Recommended for most scenarios	Legacy data, shared storage

Table 5: *Managed vs External Tables*

7.3 How Unity Catalog Security Works

1. **User submits query:** `SELECT * FROM catalog.schema.table`
2. **Access Control check:** Does user have SELECT permission?
3. **If authorized:** UC delegates to cloud IAM role to fetch data
4. **Data returned:** User sees only what they're permitted to see

Key Information

Two-Layer Security:

- **Layer 1 (Cloud IAM):** Controls access to storage buckets (get, put, list)
- **Layer 2 (Unity Catalog):** Fine-grained control (SELECT, INSERT on specific tables/columns)

Advantage: You don't need hundreds of IAM roles for different access patterns. One IAM role with broad access + UC for fine-grained control.

8 ABAC: Attribute-Based Access Control

8.1 RBAC vs ABAC

Approach	**RBAC (Role-Based)**	**ABAC (Attribute-Based)**
How it works	Assign permissions to roles; assign roles to users	Define policies based on data attributes (tags)
Example	”Managers can see all HR data”	”Anyone querying PII-tagged columns sees masked data”
Scalability	Role explosion as permissions grow	Scales well with tags
Flexibility	Static; requires role changes	Dynamic; tag changes propagate automatically

Table 6: *RBAC vs ABAC Comparison*

8.2 ABAC in Unity Catalog

The ABAC workflow in Databricks:

1. **Create Governance Tags:** Define tag names and allowed values
2. **Tag Data Assets:** Apply tags to columns/tables (manually or via AI classification)
3. **Create ABAC Policies:** Define rules based on tags
4. **Automatic Enforcement:** Policies apply whenever tagged data is accessed

```sql
-- Step 1: Create a masking function
CREATE FUNCTION ssn_mask(ssn STRING)
RETURNS STRING
RETURN
    CASE
        WHEN is_account_group_member('admin_group') THEN ssn
        WHEN is_account_group_member('analyst_group') THEN CONCAT('***-**-',
           RIGHT(ssn, 4))
        ELSE '***-**-****'
    END;

-- Step 2: Apply mask to a column
ALTER TABLE employees
ALTER COLUMN social_security_number
SET MASK ssn_mask;
```

Listing 1: Creating a Column Masking Function

8.3 ABAC Policy Types

Definition:

Row-Level Filtering Restrict which **rows** a user can see based on a condition.

Example: Sales reps can only see customers in their assigned region.

```sql
-- Only show rows where region matches user's region
CREATE FUNCTION region_filter()
RETURNS BOOLEAN
RETURN region = current_user_region();
```

Definition:

Column-Level Masking Transform or hide **column values** based on user permissions.

Example: Non-admin users see email as `j***@company.com`

8.4 Automatic Data Classification

Unity Catalog can automatically detect PII using AI models:

- **Email addresses:** Detected and tagged automatically
- **Phone numbers:** Pattern recognition
- **Names, locations:** NER-based detection
- **Credit card numbers:** Regex + Luhn validation

Key Information

Auto-Classification + ABAC = Powerful Automation

When you combine:

1. Auto-classification (AI detects email columns)
2. Governance tags (email columns get ”PII” tag)
3. ABAC policy (PII-tagged columns are masked for analysts)

New tables are automatically protected without manual intervention!

9 Data Quality Monitoring

9.1 Why Monitor Data Quality?

”Garbage in, garbage out” applies doubly to AI. If your data quality degrades, your models and reports become unreliable.

Metric	Description
Freshness	How recently was the data updated?
Completeness	Are all expected records present? (No sudden drops)
Anomaly Detection	Have data patterns changed unexpectedly?
Data Profiling	Statistics: min, max, nulls, distributions

Table 7: *Data Quality Metrics*

9.2 Key Metrics

Example:

Completeness Monitoring **Normal Pattern:** 10,000 rows daily
Day 1: 10,200 rows
Day 2: 9,800 rows
Day 3: 10,100 rows
Day 4: 0 rows $\leftarrow$ **ALERT!**
The monitoring system detects the anomaly and triggers an alert before downstream systems are affected.

9.3 Setting Up Monitoring in Databricks

Data quality monitoring is enabled with a single click:

1. Navigate to schema in Unity Catalog
2. Click ”Enable Quality Monitoring”
3. System automatically tracks freshness, completeness, anomalies
4. Alerts can be configured for threshold violations

10 Lineage: Tracing Data Origins

10.1 What Is Data Lineage?

Definition:

Data Lineage A visual representation of data’s journey: where it came from, how it was transformed, and where it goes.
Analogy: A family tree (genealogy) for your data.

10.2 Why Lineage Matters

1. **Debugging:** ”This dashboard number looks wrong. Where did it come from?”
2. **Impact Analysis:** ”If I change this column, what downstream reports break?”
3. **Compliance:** ”Can we prove this data wasn’t derived from restricted sources?”

4. **AI Governance:** "What data was used to train this model?"

10.3 Lineage in Unity Catalog

Unity Catalog automatically captures lineage:

- **Table-level:** Which tables feed into which tables
- **Column-level:** How specific columns are derived (e.g., substring, join)
- **Custom lineage:** Connect external sources (Salesforce) and targets (PowerBI)

Example:

Lineage Use Case **Scenario:** A BI report shows incorrect revenue figures.

With Lineage:

1. Navigate to the report's source table
2. Click "View Lineage"
3. Trace back through transformations
4. Discover: A join condition was changed 3 days ago
5. Fix the join and reprocess

11 Lakehouse Federation: Unified Governance

11.1 The Problem

Organizations have data in multiple systems:

- Snowflake data warehouse
- PostgreSQL operational database
- MySQL legacy systems
- Cloud storage (S3, Azure Blob)

Managing governance separately in each system is impractical.

11.2 The Solution: Federation

Definition:

Lakehouse Federation Connect external databases to Unity Catalog **without copying data**. Query remote data as if it were local, with unified governance.

Analogy: An embassy on foreign soil—your laws (governance rules) apply, even though the data physically resides elsewhere.

11.3 How Federation Works

1. **Create Connection:** Register external database (Snowflake, PostgreSQL, etc.)

2. **Create Foreign Catalog:** Maps external schemas to UC
3. **Query with Pushdown:** Queries are pushed to the source system for efficiency
4. **Unified Governance:** Same grant statements, same ABAC policies

```sql
-- Create connection to external Snowflake
CREATE CONNECTION snowflake_conn
TYPE snowflake
OPTIONS (
    host = 'account.snowflakecomputing.com',
    warehouse = 'COMPUTE_WH'
);

-- Create foreign catalog
CREATE FOREIGN CATALOG snowflake_catalog
USING CONNECTION snowflake_conn;

-- Query as if local!
SELECT * FROM snowflake_catalog.schema.table;
```

Listing 2: Creating a Federated Connection

12 Delta Sharing: Secure Data Exchange

12.1 The Challenge of Data Sharing

Traditional methods are insecure and inefficient:

- Email CSV files (security nightmare)
- FTP transfers (no access control)
- Copy data to partner's system (data duplication)

12.2 Delta Sharing Solution

Definition:

Delta Sharing An open protocol for securely sharing data **without copying**. Recipients don't need Databricks—they can consume via Pandas, Tableau, PowerBI.

What Can Be Shared:

- Tables (Delta format)
- Volumes (files)
- Notebooks
- AI Models
- Even federated tables from external sources!

Key Information

Cross-Cloud Sharing:
You can share a Snowflake table (connected via Federation) with a partner on AWS who uses Pandas. The data never leaves Snowflake, but governance is controlled through Unity Catalog.

13 The Governance Platform Wars

13.1 Who's Competing?

Every major data platform is building governance capabilities:

Platform	**Governance Layer**	**Differentiator**
Databricks	Unity Catalog	ML/AI-first; models, volumes, functions
Snowflake	Polaris + Horizon	SQL warehouse heritage; Iceberg focus
Microsoft	Fabric	Office 365 integration; broad enterprise
AWS	Glue Catalog + Lake Formation	Native AWS integration

Table 8: *Governance Platform Comparison*

Key Information

The Winner's Strategy:
The platform that can govern **everyone's assets**—not just their own—will win. If Databricks can effectively govern Snowflake data and vice versa, the most interoperable platform gains the most ”assets under management.”

14 Real-World Governance Considerations

14.1 Data Breaches and Reputational Risk

Example:

Case Study: Capital One Breach In 2019, Capital One suffered a major data breach affecting 100+ million customers.

Consequences:

- $80 million in regulatory fines
- Class action lawsuits
- Years of reputational damage
- Increased scrutiny from regulators

Lesson: The cost of poor governance far exceeds the cost of implementing it.

14.2 The Governance-Agility Tradeoff

- **Too Strict:** ”I need 5 approvals to access any data”—innovation stalls

- **Too Loose:** "Everyone has access to everything"—breaches happen

Solution: Risk-Based Governance

- **High-risk data (PII, financial):** Strict controls, approval workflows
- **Low-risk data (public datasets, experiments):** Permissive access

15 Quick Summary: One-Page Review

Key Summary

Key Takeaways from Lecture 12:

1. **Governance Hierarchy:** Information Governance $\supset$ Data Governance $\supset$ AI Governance
2. **Four Pillars:** Policies (what) $\rightarrow$ Procedures (how) $\rightarrow$ Standards (specifications) $\rightarrow$ Controls (enforcement)
3. **Unity Catalog Structure:** Metastore $\rightarrow$ Catalog $\rightarrow$ Schema $\rightarrow$ Table/Volume/Model
4. **ABAC Advantages:**
   - Tag-based policies scale better than per-table permissions
   - Auto-classification discovers PII automatically
   - Policies propagate to new data without manual intervention
5. **Data Quality:**
   - Monitor: Freshness, Completeness, Anomalies
   - Enable with one click in Unity Catalog
6. **Lineage:**
   - Automatically captured for all Databricks operations
   - Custom lineage for external sources/targets
7. **Federation:**
   - Query external databases without copying data
   - Unified governance across heterogeneous systems
8. **Delta Sharing:**
   - Share data without copying
   - Recipients don't need Databricks

16 Frequently Asked Questions

Q: Why use Unity Catalog if we already have cloud IAM roles?

A: Cloud IAM controls access at the file/folder level. Unity Catalog provides fine-grained control at the table, column, and row level. You can have one IAM role with broad storage access, and use UC for precise governance.

Q: Does governance slow down innovation?

A: Initially, there's overhead in setting up policies. Long-term, governance **accelerates** work by:

- Reducing time spent finding/validating data
- Avoiding security incidents that halt projects
- Enabling self-service access to pre-approved datasets

Q: What's the difference between RBAC and ABAC?

A: RBAC assigns permissions to roles ("Managers can see HR data"). ABAC uses attributes/tags ("Anyone querying PII sees masked data"). ABAC is more flexible and scales better.

Q: How does federation handle performance?

A: Queries are pushed down to the source system. If PostgreSQL is efficient at filtering, that filter runs on PostgreSQL—not in Spark. This minimizes data movement.

■ **Course:** CSCI E-103: Reproducible Machine Learning

■ **Week:** Lecture 13

■ **Instructors:** Eric Gieseke & Ram Sriharsha

■ **Objective:** Master the continuous improvement cycle: CI/CD pipelines, Infrastructure as Code (IaC), observability, data quality monitoring, and Databricks Asset Bundles (DABs)

Contents

1 Introduction: Beyond ”Working Code”

Building a data pipeline that ”works” is just the beginning. In enterprise environments, the real challenge is:

- How do you **deploy reliably** across environments (dev, staging, production)?
- How do you **automate testing** so bugs are caught early?
- How do you **monitor quality** so bad data doesn’t corrupt downstream systems?
- How do you **manage costs** so your cluster doesn’t run up a $100,000 bill?

This lecture bridges the gap between ”data science” and ”data engineering operations.”

Lecture Overview

Key Topics:

1. **Software Development Lifecycle (SDLC):** Dev → Staging → Production
2. **Infrastructure as Code (IaC):** Terraform for reproducible infrastructure
3. **CI/CD Pipelines:** Automated testing and deployment
4. **Observability:** System tables, cost monitoring, data quality
5. **Databricks Asset Bundles (DABs):** Modern deployment packaging

2 Software Development Lifecycle (SDLC)

2.1 The Three-Environment Model

Never deploy code directly to production. Instead, code travels through three environments:

Aspect	**Development**	**Staging**	**Production**
Purpose	Experiment and implement	Test with realistic data	Serve real users
Data	Fake/sample data	Near-production volume	Real data
Compute	Single node, spot instances	Larger clusters	High-availability clusters
Credentials	Developer accounts	Service principals begin	**Only service principals**
Execution	Interactive (notebooks)	Transitioning to automated	**Fully automated**
Cost Focus	Minimize (use spot)	Balance cost/realism	Prioritize reliability

Table 1: *SDLC Environment Comparison*

Definition:

Service Principal A **Service Principal** is a ”robot account”—an identity used by automated processes, not humans. In production, pipelines should never run under a developer’s personal credentials.

Why? If a developer leaves the company and their account is deactivated, any pipeline running under their credentials will break.

2.2 Best Practices for Building Data Products

1. **Understand the Use Case**
   - Document business requirements and domain-specific challenges
   - Define functional requirements (what it does) and non-functional requirements (performance, scale)
   - Establish Service Level Agreements (SLAs) upfront
2. **Build for Scale from Day One**
   - A pipeline that works on 1GB will often fail on 1TB
   - Test with production-scale data in staging
3. **Use Repeatable Patterns**
   - Configuration-driven pipelines (not hardcoded values)
   - Document best practices for your team
4. **Optimize Early**
   - Prefer DataFrame APIs over custom UDFs (much faster)
   - Use binary formats (Delta, Parquet) over CSV
   - Enable caching during ML training
5. **Control Costs**
   - Enable auto-termination (clusters shut down after inactivity)
   - Use autoscaling (scale up when needed, scale down when idle)
   - Use spot instances for development and training
   - Tag resources for chargeback analysis

3 Service Level Agreements (SLAs)

3.1 SLA Types by Workload

Workload Type	**Key SLA Metric**
Batch Processing	Volume of data + Total processing time (e.g., ”Process 1TB in 2 hours”)
Streaming	Transactions Per Second (TPS) (e.g., ”Sustain 10,000 TPS”)
Business Intelligence	Query latency (e.g., ”Dashboard loads in < 1 second”)
Concurrency	Number of simultaneous users (e.g., ”Support 1,000 concurrent analysts”)

Table 2: *SLA Metrics by Workload Type*

3.2 Availability and Disaster Recovery

Definition:

Availability (Uptime)

$$\text{Availability} = \frac{\text{Total Uptime}}{\text{Total Uptime} + \text{Total Downtime}} \times 100\%$$

Common Targets:

- **Five 9s (99.999%):** Only 5 minutes downtime per year (extremely demanding)
- **Four 9s (99.99%):** About 52 minutes downtime per year
- **Three 9s (99.9%):** About 8.7 hours downtime per year

Definition:

RTO and RPO **RTO (Recovery Time Objective):** How quickly must the system be restored after failure?

- Example: "RTO = 15 minutes" means system must be back online within 15 minutes of an outage

RPO (Recovery Point Objective): How much data can you afford to lose?

- Example: "RPO = 1 hour" means backups every hour; you might lose up to 1 hour of data

Key Information

Multi-Region Deployment:
For high availability, deploy across multiple cloud regions. If one data center goes down, traffic fails over to another region. This is expensive but essential for mission-critical applications.

4 Infrastructure as Code (IaC)

4.1 Why Infrastructure as Code?

Example:

The Problem Without IaC **Scenario:** Your team needs 50 identical Databricks workspaces, each with specific network configurations, security groups, and cluster policies.
Manual Approach: Click through the AWS/Azure console 50 times. Hope you don't make mistakes. Document nothing. When someone asks "why is workspace 37 different?" you have no idea.
IaC Approach: Write code that describes the workspace. Run it 50 times. Every workspace is identical. Changes are version-controlled in Git.

4.2 IaC Principles

1. **Define Everything as Code:** Networks, storage, workspaces, clusters—all defined in configuration files

2. **Version Control:** Infrastructure code lives in Git, with full history
3. **Continuous Testing:** Validate infrastructure changes before applying
4. **Small, Independent Pieces:** Change one component without affecting others

4.3 Provisioning vs Configuration Management

Aspect	Provisioning	Configuration Management
Purpose	Create infrastructure from scratch	Modify existing infrastructure
Philosophy	Immutable (replace, don't modify)	Mutable (update in place)
Tools	Terraform, CloudFormation, Bicep	Ansible, Chef, Puppet
Example	"Create a VPC with these subnets"	"Update Python to 3.11 on all servers"
Agent Required?	No (uses APIs)	Yes (agents on each server)

Table 3: *Provisioning vs Configuration Management*

4.4 Terraform for Databricks

Terraform is the dominant provisioning tool, with strong Databricks support.

```
# Define the cloud provider
provider "azurerm" {
  features {}
}

# Create a resource group
resource "azurerm_resource_group" "rg" {
  name     = "databricks-rg"
  location = "East US"
}

# Create Databricks workspace
resource "azurerm_databricks_workspace" "workspace" {
  name                = "my-databricks"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  sku                 = "premium"
}

# Output the workspace URL
output "workspace_url" {
  value = azurerm_databricks_workspace.workspace.workspace_url
}
```

Listing 1: Terraform Example: Databricks Workspace

```
# Initialize Terraform (download providers)
terraform init

# Preview what will be created (dry run)
terraform plan

# Apply changes (actually create resources)
terraform apply

# Destroy everything (clean up)
terraform destroy
```

Listing 2: Terraform Workflow

Key Information

Terraform State:
Terraform maintains a **state file** that tracks what resources exist. When you run `terraform apply`, it compares desired state (your code) vs. actual state (state file) and only changes what's different.

5 CI/CD: Continuous Integration and Continuous Delivery

5.1 What Is CI/CD?

Definition:

CI (Continuous Integration) Every time a developer commits code:

1. Code is automatically pulled from the repository
2. Build process runs (compile code, build packages)
3. Unit tests execute automatically
4. If tests fail, the developer is notified immediately

Definition:

CD (Continuous Delivery/Deployment) After CI passes:

1. Code is automatically deployed to staging
2. Integration tests run against staging
3. If approved (automatically or manually), deploy to production

Continuous Delivery: Requires manual approval before production deploy
Continuous Deployment: Fully automated to production (riskier but faster)

5.2 The CI/CD Pipeline Flow

`Commit` → `Build` → `Unit Test` → `Deploy to Staging` → `Integration Test` → `Deploy to Prod`

Warning

Always Have a Rollback Plan:
When deploying to production, have an automated way to roll back to the previous version if something goes wrong. This could be:

- Blue-green deployment (instant switch between versions)
- Canary deployment (gradually shift traffic to new version)
- Simple rollback script

5.3 CI/CD Tools

- **Source Control:** GitHub, GitLab, Bitbucket
- **CI/CD Servers:** Azure DevOps, Jenkins, GitHub Actions, GitLab CI
- **For Databricks:** Use Databricks REST APIs + Databricks Asset Bundles

6 Workspace Organization and Data Mesh

6.1 Workspace as Isolation Boundary

In Databricks, a **workspace** is the smallest isolation boundary:

- Own set of clusters, jobs, notebooks
- Own folder structure
- Own security perimeter

6.2 Workspace Organization Strategies

1. **Traditional (Simple):** Dev workspace, Staging workspace, Production workspace
2. **By Line of Business:** Marketing workspace, Sales workspace, Finance workspace
3. **By Project:** Each major project gets its own workspace
4. **Hybrid:** Core data in shared workspace, domain-specific in dedicated workspaces

Key Information

Enterprise Scale:
Mature organizations may have **hundreds to thousands** of workspaces. This is why IaC and standardized provisioning become essential—you can't manually configure 1,000 workspaces.

6.3 Data Mesh Architecture

Definition:

Data Mesh A decentralized approach to data architecture where:

1. **Domain Ownership:** Each business domain (Marketing, Sales) owns its data
2. **Data as Product:** Data is treated as a product with SLAs and documentation
3. **Self-Service Platform:** Domains can provision their own infrastructure
4. **Federated Governance:** Central policies, decentralized execution

Unity Catalog's Role: Even with decentralized domains, Unity Catalog provides the "glue" that enables data discovery, lineage tracking, and consistent governance across all domains.

7 Observability and Monitoring

7.1 System Tables for Platform Monitoring

Databricks provides **system tables** in a special `system` catalog that contain:

Schema	**Contents**
`system.billing`	Cost and usage data (SKUs, usage quantities, custom tags)
`system.compute`	Cluster information (who owns it, instance types, CPU usage)
`system.access`	Audit logs (who accessed what data, when)

Table 4: *Key System Tables*

```sql
-- Find top 10 most expensive jobs this month
SELECT
    custom_tags:project AS project,
    SUM(usage_quantity) AS total_dbus,
    SUM(usage_quantity * list_price) AS estimated_cost
FROM system.billing.usage
WHERE usage_date >= DATE_TRUNC('month', CURRENT_DATE())
GROUP BY custom_tags:project
ORDER BY estimated_cost DESC
LIMIT 10;
```

Listing 3: Querying System Tables for Cost Analysis

Important:

Tag Your Resources! Custom tags are essential for cost allocation. Tag all clusters and jobs with:

- Project name
- Team/department
- Environment (dev/staging/prod)
- Cost center

Without tags, you cannot answer ”Which project is costing us the most?”

7.2 Data Quality Monitoring

Data teams often have SLAs on **delivery** (get data by 8 AM) but users expect **quality**. When quality issues arise, teams are reactive rather than proactive.

7.2.1 Anomaly Detection

One-click setup that uses AI to detect:

- **Freshness:** Data hasn't been updated when expected
- **Completeness:** Sudden drops in row counts

The system learns patterns automatically and alerts when deviations occur.

7.2.2 Lakehouse Monitoring

More detailed monitoring at the table level:

- **Profile Metrics:** Min, max, mean, nulls, distinct counts
- **Drift Detection:** Statistical tests (KS test, chi-square) to detect distribution changes
- **Automatic Dashboards:** Visual representation of data quality over time

Example:

Drift Detection Use Case **Scenario:** A column `preferred_payment_method` suddenly has 90% NULL values (normally 5%).
Without Monitoring: You discover this 2 weeks later when a downstream ML model starts making bad predictions.
With Lakehouse Monitoring: Alert fires within hours. Investigation reveals an upstream API change that stopped sending payment data. Fix deployed before ML model is affected.

7.3 Lineage for Root Cause Analysis

When something breaks, lineage helps you answer:

- **What upstream process caused this?** (trace back to source)
- **What downstream systems are affected?** (impact analysis)

```
-- This table is used by these downstream tables/dashboards
-- Visible in Unity Catalog UI under "Lineage" tab
-- Shows: source tables -> transformations -> target tables -> dashboards
```

Listing 4: Using Lineage for Impact Analysis

7.4 DQX: Code-Based Quality Rules

For more explicit quality rules, use DQX (Data Quality X):

```python
# Define quality rules in YAML
rules = """
checks:
  - column: id
    rule: not_null_and_not_empty
    criticality: error
  - column: age
    rule: value_in_range
    params: {min: 18, max: 120}
    criticality: warning
  - column: country
    rule: is_in_list
    params: {allowed: [Germany, France, USA]}
    criticality: warning
"""

# Apply rules to DataFrame
from dqx import DQEngine
engine = DQEngine()
valid_df, invalid_df, errors = engine.apply_checks(df, rules)

# valid_df contains rows passing all checks
# invalid_df contains rows that failed
# errors contains detailed failure information
```

Listing 5: DQX Quality Rules Example

8 Databricks Asset Bundles (DABs)

8.1 The Problem DABs Solve

Moving projects between environments traditionally required:

- Manual export/import of notebooks
- Recreating jobs with different configurations
- Updating cluster references
- Lots of API scripting

DABs package everything together in a single, portable bundle.

Definition:

Databricks Asset Bundle A **DAB** is a project-level packaging format that includes:

- Notebooks and Python files
- Job definitions

- Cluster configurations
- Variables that differ by environment
- Target environments (dev, staging, prod)

One command deploys everything: `databricks bundle deploy`

8.2 DAB Structure

```yaml
bundle:
  name: my_project

variables:
  catalog:
    default: dev_catalog
  cluster_id:
    lookup:
      cluster: my-cluster

resources:
  jobs:
    etl_pipeline:
      name: "ETL Pipeline"
      tasks:
        - task_key: bronze
          notebook_task:
            notebook_path: ./src/create_bronze.py
        - task_key: silver
          depends_on:
            - task_key: bronze
          notebook_task:
            notebook_path: ./src/create_silver.py

targets:
  development:
    mode: development
    default: true
    workspace:
      host: https://dev.cloud.databricks.com

  production:
    mode: production
    workspace:
      host: https://prod.cloud.databricks.com
    variables:
      catalog: prod_catalog
    resources:
      jobs:
        etl_pipeline:
          name: "Production ETL Pipeline"
```

Listing 6: Example databricks.yml

8.3 DAB Commands

```
# Initialize a new bundle from template
databricks bundle init

# Validate bundle configuration (catch errors before deploy)
databricks bundle validate

# Deploy to development environment
databricks bundle deploy -t development

# Run a job in development
databricks bundle run -t development etl_pipeline

# Deploy to production (uses production overrides)
databricks bundle deploy -t production
```

Listing 7: DAB CLI Commands

Key Information

DAB vs Terraform:

- **Terraform:** Creates the *infrastructure* (workspace, networks, storage)
- **DAB:** Deploys your *project* within that infrastructure (notebooks, jobs, pipelines)

Both are complementary. Use Terraform to create workspaces, then DAB to deploy code to them.

8.4 DAB in CI/CD Pipelines

DABs integrate seamlessly into CI/CD:

```
# .azure-pipelines.yml
trigger:
  - main

stages:
  - stage: Test
    jobs:
      - job: ValidateBundle
        steps:
          - script: pip install databricks-cli
          - script: databricks bundle validate

  - stage: DeployDev
    jobs:
      - job: DeployToDevEnvironment
        steps:
          - script: databricks bundle deploy -t development
          - script: databricks bundle run -t development etl_pipeline

  - stage: DeployProd
    condition: succeeded()
    jobs:
      - deployment: DeployToProduction
        environment: production
        strategy:
          runOnce:
            deploy:
              steps:
                - script: databricks bundle deploy -t production
```

Listing 8: Azure DevOps Pipeline with DABs

9 MLOps: Special Considerations for Machine Learning

9.1 Deploy Model vs Deploy Code

Approach	**Deploy Model**	**Deploy Code**
What Moves	Trained model artifact	Training code
Training Location	Development environment	Each environment
Data Used	Dev data	Environment-specific data
Best For	Simple models, same data everywhere	Models that need environment-specific training

Table 5: *MLOps Deployment Strategies*

Warning

Data Differences Matter:
A model trained on development data (which may be sampled or anonymized) might perform differently on production data. Consider retraining in production with full data.

10 Cost Optimization Strategies

1. **Use Spot Instances for Development**
   - 60-90% cheaper than on-demand
   - May be reclaimed by cloud provider
   - Never use for production-critical workloads
2. **Enable Auto-Termination**
   - Clusters shut down after N minutes of inactivity (e.g., 30 minutes)
   - Prevents ”weekend clusters” that run unused for 48 hours
3. **Use Autoscaling**

- Scale up when workload increases
- Scale down (even to 0) when idle

4. **Move to Serverless**
   - Pay only for what you use
   - No cluster management overhead
   - Faster startup (seconds vs minutes)
5. **Write Efficient Code**
   - Use DataFrame APIs over UDFs
   - Use binary formats (Delta/Parquet) over CSV
   - Leverage caching appropriately
6. **Monitor and Alert**
   - Set up alerts for unusual spending
   - Review system tables weekly
   - Tag resources for attribution

11 Quick Summary: One-Page Review

Key Summary

Key Takeaways from Lecture 13:

1. **SDLC Environments:**
   - Development → Staging → Production
   - Use service principals (not personal accounts) in production
   - Production should be fully automated
2. **Infrastructure as Code (IaC):**
   - Use Terraform for reproducible infrastructure
   - Version control all infrastructure code
   - Immutable infrastructure: replace, don't modify
3. **CI/CD:**
   - Automate testing on every commit
   - Deploy through pipelines, not manually
   - Always have a rollback plan
4. **Observability:**
   - System tables for cost and audit monitoring
   - Anomaly detection for freshness/completeness
   - Lakehouse monitoring for detailed quality metrics
   - Lineage for root cause and impact analysis
5. **Databricks Asset Bundles:**
   - Package entire projects (notebooks, jobs, configs)

- Environment-specific overrides via targets
- `databricks bundle deploy` for one-command deployment

6. **Cost Optimization:**
   - Spot instances for dev, auto-termination, autoscaling
   - Tag everything for chargeback
   - Consider serverless for variable workloads

12 Glossary

Term	**Definition**
Service Principal	A ”robot account” for automated processes
Spot Instance	Discounted compute that can be reclaimed by the cloud provider
RTO	Recovery Time Objective: how quickly to restore after failure
RPO	Recovery Point Objective: maximum acceptable data loss
IaC	Infrastructure as Code: defining infrastructure in version-controlled files
CI/CD	Continuous Integration / Continuous Delivery: automated testing and deployment
DAB	Databricks Asset Bundle: project packaging for cross-environment deployment
Data Mesh	Decentralized data architecture with domain ownership
Drift	Changes in data distributions over time
Lineage	Tracking data’s journey from source to destination

Table 6: *Key Terms*

■ **Course:** CSCI E-103: Reproducible Machine Learning

■ **Lecture:** Lecture 14 – Databricks Roadmap & Emerging Features

■ **Instructors:** Anindita Mahapatra & Eric Gieseke

■ **Objective:** Explore Databricks' latest platform features including Lakebase, AI/BI, Agent Bricks, MCP integration, and review key concepts from Quiz 2 and Assignment 3

Contents

1 Introduction: The Databricks Data Intelligence Platform

Lecture Overview

This lecture explores the cutting-edge features in the Databricks roadmap—technologies that are either in public preview or will be released soon. These innovations represent the platform's evolution toward a comprehensive **Data Intelligence Platform** that handles everything from data storage to AI agent deployment.

Key Learning Objectives:

- Understand the new Databricks One unified experience
- Learn about Lakebase and OLTP capabilities within the Lakehouse
- Explore Agent Bricks and AI/BI features (Genie, Research Mode)
- Understand Lakehouse Apps for building secure data applications
- Review Quiz 2 concepts (Spark optimization, Delta Lake internals)
- Review Assignment 3 concepts (Streaming, SCD, Kafka/Kinesis)

1.1 The Data Intelligence Platform Architecture

The modern Databricks platform is built on a layered architecture that serves diverse workloads:

Layer	Components	Purpose
Foundation	Cloud Storage	Object storage (S3, ADLS, GCS) for all data
Format Layer	Delta, Iceberg	Open table formats ensuring data reliability and ACID compliance
Governance	Unity Catalog	Centralized governance, security, and lineage tracking
Compute	Serverless SQL, Clusters	Scalable compute for all workload types
Applications	Agent Bricks, DBSQL, Apps	User-facing applications and AI capabilities

Table 1: *Databricks Data Intelligence Platform Layers*

Key Information

Key Investment Areas in Databricks:

1. **Data Intelligence Capabilities**: Making the platform smarter so data professionals do less manual work
2. **Built-in Governance & Security**: Data sharing tools with ABAC and fine-grained access control
3. **Tools to Build AI Agents**: Agent Bricks, MCP servers, model serving
4. **Data Engineering & Migration**: LakeFlow, connectors, zero-copy ETL

2 Databricks One: The Unified Experience

Definition:

Databricks One **Databricks One** is a new unified portal experience designed for business users. It provides account-level access to dashboards, Genie spaces, and apps across all workspaces in a single, intuitive interface.

2.1 Portal Switcher: Three Experiences

When you log into Databricks, you'll see a portal switcher with three options:

Portal	Target Users	Purpose
Lakehouse	Data Engineers, Data Scientists	Full workspace with SQL, ML, Data Engineering features. This is the traditional Databricks experience.
Databricks One	Business Users, Analysts	Simplified view showing dashboards, Genie, and apps across all workspaces. No technical complexity.
Lakebase Postgres	Application Developers	OLTP database interface for transactional workloads. PostgreSQL-compatible.

Table 2: *Databricks Portal Experiences*

2.2 Databricks One Features

- **Account-Level Access**: View assets from all workspaces in one place
- **Vanity URL**: Companies can use their own domain (e.g., `analytics.yourcompany.com`)
- **Domain Integration**: Organize content by business domain (Marketing, Finance, Tech)
- **Intuitive Navigation**: Search and discover dashboards, Genie spaces, and apps easily

Example:

Why Databricks One Matters Imagine a large enterprise with 50+ workspaces. Without Databricks One, a marketing analyst would need to know which specific workspace contains their dashboard. With Databricks One, they simply log in, see all marketing-related assets, and get to work—regardless of which workspace hosts the asset.

3 Lakebase: Bringing OLTP to the Lakehouse

Definition:

Lakebase **Lakebase** is Databricks' managed PostgreSQL implementation that brings **OLTP (Online Transaction Processing)** capabilities to the Lakehouse platform. It enables transactional workloads alongside analytical workloads, all within the same governance framework.

3.1 Understanding OLTP vs OLAP

Aspect	OLTP (Transactional)	OLAP (Analytical)
Purpose	Handle real-time transactions	Analyze historical data
Operations	INSERT, UPDATE, DELETE (many small)	SELECT (few large queries)
Data Volume	Current operational data	Historical aggregated data
Users	Applications, end users	Analysts, data scientists
Examples	Banking transactions, order processing	Sales reports, trend analysis
Traditional Tools	Oracle, SQL Server, PostgreSQL	Data warehouses, Databricks

Table 3: *OLTP vs OLAP Comparison*

3.2 Why Lakebase is Revolutionary

Important:

The Storage-Compute Separation Paradigm Shift Traditional databases (Oracle, SQL Server) have compute and storage tightly coupled. If you buy a database server, it runs 24/7 whether you have one row or one billion rows.

Lakebase changes this fundamentally:

- Storage and compute are completely separated
- Compute can scale up as load increases
- Compute can scale **down to zero** when idle—you pay nothing
- Sub-second startup time means instant readiness when needed

This was possible for data lakes and warehouses, but **never before for transactional databases**. Lakebase is the first to achieve this for OLTP workloads.

3.3 Lakebase Architecture

```
-- Connection string format
psql "postgres://your-lakebase-instance.databricks.com:5432/database_name"

-- Use OAuth token for authentication
-- Default role is your Databricks username
```

Listing 1: Lakebase PostgreSQL Connection

3.3.1 Key Features

1. **PostgreSQL Compatibility**: Standard PostgreSQL syntax and tools work seamlessly
2. **Version Selection**: Choose your PostgreSQL version (e.g., PostgreSQL 17)
3. **Branch-Based Development**:
   - Production branch (for live data)
   - Development branch (for testing)
   - Git-style workflow: Test safely without touching production
4. **Backup and Recovery**:
   - Scheduled snapshots
   - Point-in-time recovery
   - Instant recovery capabilities
5. **SQL Editor**: Built-in query interface, monitoring, and table management

3.4 Catalog Types in Unity Catalog

When creating a catalog in Unity Catalog, you now have four options:

Catalog Type	Use Case	Description
Standard	Default projects	Typical catalog for tables, views, models
Foreign	Data federation	Connect to external data stores (Snowflake, Redshift, Teradata) without moving data
Shared	Delta Sharing	Zero-copy data sharing with external parties
Lakebase PostgreSQL	OLTP workloads	Connect to your Lakebase databases for transactional access

Table 4: *Unity Catalog Types*

3.5 Bidirectional Sync: Lakehouse ↔ Lakebase

Example:

Data Flow Between Lakehouse and Lakebase **Forward Sync (Lakehouse → Lakebase):**

- Sync a subset of your analytical data to Lakebase
- Enables low-latency reads for applications

- Useful for ML inference, real-time dashboards

Reverse Sync (Lakebase → Lakehouse):

- Business users make small corrections in Lakebase
- Changes sync back to the Lakehouse
- Maintains a single source of truth

```sql
-- Create table in Unity Catalog
CREATE TABLE catalog.schema.user_segments (
    user_id INT,
    segment STRING,
    engagement STRING
);

-- Sync to Lakebase PostgreSQL
-- (Done through UI or sync configuration)

-- Query in Lakebase
SELECT * FROM default.user_segment_synced
WHERE engagement = 'high';
```

4 AI/BI: Genie and Research Mode

Definition:

AI/BI **AI/BI** is Databricks' AI-powered business intelligence solution consisting of two main components:

1. **Dashboards**: Interactive visualizations with drill-down capabilities
2. **Genie**: Natural language interface for querying data

4.1 Genie: Answering "What" Questions

Genie transforms natural language questions into SQL queries against your structured data:

- **Factual Questions**: "What were total sales last quarter?"
- **SQL Generation**: Automatically generates optimized SQL
- **Custom Instructions**: Add context, join logic, and example queries
- **Consistent Responses**: Trained on your data semantics

4.2 Research Mode: Answering "Why" Questions

Important:

Research Mode vs Regular Genie **Regular Genie**: Answers *what* questions with direct SQL queries.

- "What were sales in Q4?" → Returns a number
- Fast response, factual answer

Research Mode: Answers *why* questions with hypothesis-driven analysis.

- "Why did sales drop in Q4?" → Explores multiple factors
- Uses **Chain of Thought** reasoning
- Takes longer (several minutes) but provides deeper insights
- Shows the reasoning paths explored

Example:

Research Mode in Action **Question:** "What would happen to my portfolio if oil prices increase by 20%?"

Research Mode Process:

1. Identifies relevant data (portfolio holdings, sector exposure)
2. Generates hypotheses about impact paths
3. Explores correlations with oil-sensitive sectors
4. Analyzes historical patterns during price spikes
5. Synthesizes findings into actionable insights

Output: A detailed analysis showing multiple scenarios, not just a simple number.

4.3 Enhanced AI/BI Features

- **Document Upload**: Attach CSVs, Excel files, or PDFs to combine with structured data
- **Drill-Down for Dashboards**: Click on visualizations to explore hierarchies (Region → Country → State → City)
- **Slack/Teams Integration**: Subscribe to dashboard alerts and notifications
- **Multi-Page Subscriptions**: Schedule reports across multiple dashboard pages

5 Agent Bricks and AI Agents

Definition:

Agent Bricks **Agent Bricks** is Databricks' framework for creating AI agents with minimal code. Similar to AutoML, you point it to data and specify what you want—the framework handles the scaffolding, evaluation, and deployment.

5.1 Types of Agents Available

Agent Type	**Purpose**
Information Extraction	Extract structured data from unstructured documents
Knowledge Assistant	Answer questions using your organization's knowledge base
AI/BI Genie	Natural language querying of structured data
Multi-Agent Supervisor	Coordinate multiple specialized agents
Custom LLM	Build agents using your own fine-tuned models
Custom Agent	Full control over agent logic and behavior

Table 5: *Agent Bricks Agent Types*

5.2 Creating an Agent

Example:

Information Extraction Agent Creating an information extraction agent:

1. **Name**: Give your agent a descriptive name
2. **Data Location**: Point to your documents (PDFs, emails, forms)
3. **Fields to Extract**: Specify what information to pull (company name, address, dates)
4. **Deploy**: Agent is ready in minutes

No coding required—similar to the AutoML experience for machine learning.

5.3 MCP (Model Context Protocol)

Definition:

MCP **Model Context Protocol (MCP)** is an open standard that enables AI agents to connect to external tools, data sources, and services. MCP servers act as bridges between agents and the resources they need.

- **MCP in Marketplace**: Discover and use pre-built MCP servers

- **Governance**: MCP servers are governed by Unity Catalog
- **Custom MCP**: Build your own MCP servers for proprietary tools
- **Serverless GPU**: MCP servers can leverage GPU compute for AI tasks

6 Lakehouse Apps

Definition:

Lakehouse Apps **Lakehouse Apps** allow you to build and deploy web applications directly on the Databricks platform, where your data lives. Apps can read and write data securely, interact with models, and provide user-friendly interfaces for non-technical users.

6.1 Why Apps vs Dashboards?

Aspect	Dashboards	Apps
Interaction	Read-only visualization	Bidirectional (read & write)
Data Modification	Not possible	Users can update data
Use Case	Reporting, monitoring	Data correction, input forms, chatbots
User Experience	Pre-defined views	Custom, interactive UI

Table 6: *Dashboards vs Apps Comparison*

6.2 Supported Frameworks

- **Python**: Streamlit, Dash, Flask, Gradio, Shiny
- **JavaScript**: Node.js, React
- **Templates**: Data apps, chatbots, model serving interfaces

6.3 App Architecture

```python
import streamlit as st
from databricks import sql

# Connect to SQL warehouse
connection = sql.connect(
    server_hostname="your-workspace.databricks.com",
    http_path="/sql/1.0/warehouses/abc123"
)

# Query data
cursor = connection.cursor()
cursor.execute("SELECT * FROM gold.customer_metrics LIMIT 100")
data = cursor.fetchall()

# Display in Streamlit
st.title("Customer Metrics Dashboard")
st.dataframe(data)

# Allow user to update data
if st.button("Mark as Reviewed"):
    cursor.execute("UPDATE gold.customer_metrics SET reviewed = TRUE")
    st.success("Data updated!")
```

Listing 2: Simple Streamlit App Example

Key Information

Key Benefits of Lakehouse Apps:

- Data stays in place—no copying to external systems
- Security inherited from Unity Catalog
- Horizontal scaling for many users
- Easy deployment from Git or SDK
- Support for vibe coding tools (Claude, Cursor)

7 Data Intelligence Features

7.1 AI Parse Document

```sql
-- Extract structured data from PDFs
SELECT AI_PARSE_DOCUMENT(
    file_path,
    'company_name, address, invoice_date, total_amount'
)
FROM volumes.documents.invoices;

-- Result: Structured columns extracted from unstructured PDFs
```

Listing 3: AI Parse Document Example

Use Cases:

- Extract data from scanned forms and invoices
- Parse regulatory documents
- Convert unstructured PDFs to structured tables

7.2 AI Query with Multimodal Support

```sql
-- Filter product catalog for white shoes using AI
SELECT AI_QUERY(
    'Filter our catalog for white shoes',
    image_column
)
FROM product_catalog;

-- AI understands both the query and the images
```

Listing 4: Multimodal AI Query

7.3 Spatial SQL

Databricks now supports spatial SQL for geographic analysis:

- **Geometry/Geography Types**: Native support for spatial data
- **Spatial Functions**: Intersection, distance, containment
- **Use Cases**: Disaster impact analysis, location-based services

8 Data Engineering: LakeFlow and Spark Declarative Pipelines

8.1 LakeFlow Connect

Definition:

LakeFlow Connect **LakeFlow Connect** provides a unified way to ingest data from various sources into the Lakehouse. Previously, Databricks relied on data already being in cloud storage—now it actively pulls data from source systems.

Supported Connectors:

- **File Sources**: Excel, SFTP
- **Databases**: PostgreSQL, MySQL, Oracle
- **Marketing**: Google Ads, Salesforce Marketing Cloud, TikTok Ads
- **Industry**: Customer 360, Financial Services, Insurance

8.2 LakeFlow Designer

A visual interface for building ETL pipelines:

1. Drag and drop data sources from your catalog
2. Add operators (Filter, Join, Aggregate, AI Functions)
3. Visual representation of data flow
4. Behind the scenes: Generates Spark Declarative Pipeline code

8.3 Spark Declarative Pipelines

Key Information

Evolution of Delta Live Tables (DLT):

- **Original Name**: Delta Live Tables (DLT)
- **Renamed To**: Live Tables Pipeline (LTP)
- **Current Name**: Spark Declarative Pipelines
- **Status**: Now fully open-sourced and Generally Available

9 Feature Release Lifecycle

Understanding when features are safe to use in production:

Stage	**Access**	**Production Ready?**
Beta	Very early, limited	No—experimental only
Private Preview	Selected customers	No—for evaluation
Public Preview	Available in documentation, anyone can try	Caution—some enterprises wait
Generally Available (GA)	Production-ready	Yes—enterprise use

Table 7: *Feature Release Stages*

10 Quiz 2 Review: Spark and Delta Lake Concepts

Key Summary

This section reviews key concepts from Quiz 2 that students found challenging.

10.1 Spark Optimization: The Four S's

Problem	Description	Solution
Data Skew	Uneven data distribution across partitions	Salting, broadcast joins, AQE
Shuffle	Data movement between nodes	Minimize wide transformations, optimize partitioning
Spill	Memory overflow to disk	Increase executor memory, reduce data size
Small Files	Too many tiny files	Use OPTIMIZE, predictive optimization

Table 8: *Spark Performance Problems and Solutions*

10.2 Broadcast Variables

Warning

Common Misconception: Broadcast variables are *not* shared across the cluster.

Correct Understanding:

- Broadcast variables are **immutable**—cannot be changed after creation
- They are **local to each worker node**—each node gets its own copy
- Used for small lookup tables that would otherwise require shuffle

10.3 Repartition vs Coalesce

```python
# INCREASING partitions (12 -> 24): Use repartition
df_more = df.repartition(24)  # Full shuffle

# DECREASING partitions (24 -> 12): Use coalesce
df_less = df.coalesce(12)  # No shuffle, more efficient

# ERROR: coalesce(24) when you have 12 partitions
# This won't increase partitions!
```

Listing 5: Partition Operations

10.4 collect() vs take()

Important:

The Danger of collect() **collect()** brings *all* data to the driver. This causes Out of Memory (OOM) errors with large datasets.

Best Practices:

- Use `take(n)` to retrieve only what you need
- Use `collect()` only for debugging small datasets
- Always be suspicious of `collect()` in production code

```python
# DANGEROUS for large datasets
all_data = df.collect()  # OOM risk!

# SAFE alternative
sample = df.take(5)  # Returns only 5 rows
```

10.5 Delta Lake Transaction Log

Definition:

Delta Log Structure Every Delta table has an `_delta_log` folder containing:

- **JSON files**: One per transaction (000000000000.json, 000000000001.json, ...)
- **Checkpoint files**: Parquet files created every 10 transactions
- This enables **time travel** without separate copies

Warning

Quiz Correction: The question asked about Delta log structure. The correct answer is that each transaction creates a **separate JSON file** (not a single JSON). If you marked “single JSON” as correct, please reach out to have your grade adjusted.

10.6 DataFrame Operations

```python
# EXPLODE is NOT a DataFrame method
# WRONG:
df.explode("array_column")  # This doesn't exist!

# CORRECT - use inside select:
from pyspark.sql.functions import explode
df.select("id", explode("array_column").alias("item"))

# DROP DUPLICATES - use a list for columns
# WRONG:
df.dropDuplicates("col1", "col2")  # Incorrect syntax

# CORRECT:
```

```
df.dropDuplicates(["col1", "col2"])  # Pass columns as a list
```

Listing 6: Common DataFrame Gotchas

10.7 Generated Columns

```sql
CREATE TABLE transactions (
    transaction_id BIGINT,
    transaction_timestamp TIMESTAMP,
    -- Generated column: automatically computed
    transaction_date DATE GENERATED ALWAYS AS (
        CAST(transaction_timestamp AS DATE)
    )
) PARTITIONED BY (transaction_date);

-- Benefit: Filtering by timestamp automatically
-- resolves the partition pruning on transaction_date
```

Listing 7: Generated Always As

11 Assignment 3 Review: Streaming and Data Architecture

11.1 Kafka vs Kinesis

Aspect	Apache Kafka	AWS Kinesis
Management	Self-managed or Confluent Cloud	Fully managed by AWS
Cloud Agnostic	Yes—runs anywhere	No—AWS only
Scaling Unit	Partitions	Shards
Pricing	Confluent fees or infrastructure	Per-shard pricing
Data Distribution	Round-robin or key-based	Partition key
Ecosystem	KSQL, Kafka Connect, Schema Registry	Kinesis Data Analytics, Firehose

Table 9: *Kafka vs Kinesis Comparison*

Key Information

Other Cloud Equivalents:

- **Azure**: Event Hubs
- **Google Cloud**: Pub/Sub

11.2 Slowly Changing Dimensions (SCD)

Type	Behavior	History	Use Case
Type 1	Overwrite the row	No history kept	Correction of errors
Type 2	Add new row with version	Full history with dates/flags	Customer address changes
Type 3	Add column for previous value	Limited history	Only need previous value

Table 10: *SCD Types Summary*

```sql
MERGE INTO target_table t
USING source_table s
ON t.id = s.id
-- Only update if city actually changed
WHEN MATCHED AND t.city <> s.city THEN
    UPDATE SET t.city = s.city, t.updated_at = CURRENT_TIMESTAMP
WHEN NOT MATCHED THEN
    INSERT (id, name, city, updated_at)
    VALUES (s.id, s.name, s.city, CURRENT_TIMESTAMP);
```

Listing 8: MERGE for SCD Type 1 with Condition

11.3 Spark Streaming Key Concepts

11.3.1 Trigger Options

```
# DEPRECATED - reads all available data at once
stream.writeStream \
    .trigger(once=True) \  # Don't use this!
    ...

# RECOMMENDED - micro-batch processing
stream.writeStream \
    .trigger(availableNow=True) \  # Use this instead
    ...

# CONTINUOUS - for low-latency (sub-100ms)
stream.writeStream \
    .trigger(processingTime='10 seconds') \
    ...
```

Listing 9: Streaming Trigger Options

11.3.2 Checkpointing and Exactly-Once Processing

```
stream = (
    spark.readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "server:9092")
    .option("subscribe", "sensor_data")
    .option("startingOffsets", "earliest")  # or "latest"
    .load()
)

# Checkpoint location enables exactly-once processing
stream.writeStream \
    .format("delta") \
    .outputMode("append") \
    .option("checkpointLocation", "/checkpoints/sensor_stream") \
    .table("bronze.sensor_readings")
```

Listing 10: Streaming with Checkpoint

Definition:

Exactly-Once Processing Achieved through the combination of:

1. **Checkpointing**: Tracks progress through the stream
2. **Idempotent Sinks**: Ensure operations can be safely replayed

3. **WAL (Write-Ahead Log)**: Records operations before execution

11.3.3 Reading from Kafka

```python
from pyspark.sql.functions import from_json, col
from pyspark.sql.types import StructType, StringType, IntegerType

# Define schema
schema = StructType() \
    .add("user_id", StringType()) \
    .add("device_id", StringType()) \
    .add("steps", IntegerType())

# Read and parse Kafka stream
df = (
    spark.readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "server:9092")
    .option("subscribe", "fitness_data")
    .load()
    # Kafka value is binary, cast to string
    .selectExpr("CAST(value AS STRING)")
    # Parse JSON
    .select(from_json(col("value"), schema).alias("data"))
    # Flatten
    .select("data.*")
    # Deduplicate
    .dropDuplicates(["user_id", "device_id"])
)
```

Listing 11: Kafka Stream Processing

11.4 AutoLoader (Cloud Files)

```python
# Read incrementally from cloud storage
df = (
    spark.readStream
    .format("cloudFiles")  # AutoLoader format
    .option("cloudFiles.format", "csv")
    .option("cloudFiles.schemaLocation", "/schema/sales")
    .option("header", "true")
    .load("/data/sales/")
)

# Write to Delta table
df.writeStream \
    .format("delta") \
    .option("checkpointLocation", "/checkpoints/sales") \
    .option("mergeSchema", "true") \
    .outputMode("append") \
    .table("bronze.sales")
```

Listing 12: AutoLoader Example

Key Information

AutoLoader SQL Equivalent: `COPY INTO`

```
COPY INTO bronze.sales
FROM '/data/sales/'
FILEFORMAT = CSV
FORMAT_OPTIONS ('header' = 'true')
COPY_OPTIONS ('mergeSchema' = 'true');
```

12 Machine Learning Review (Quiz 2)

12.1 Hyperparameter Optimization

Warning

Hyperopt Deprecation: The `hyperopt` library has stopped development. Use `Optuna` instead for hyperparameter optimization in new projects.

```python
import optuna
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

def objective(trial):
    # Define hyperparameters to tune
    n_estimators = trial.suggest_int('n_estimators', 10, 100)
    max_depth = trial.suggest_int('max_depth', 2, 32)

    model = RandomForestClassifier(
        n_estimators=n_estimators,
        max_depth=max_depth
    )

    score = cross_val_score(model, X, y, cv=5).mean()
    return score  # Optuna maximizes by default

study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=100)
```

Listing 13: Optuna for Hyperparameter Tuning

12.2 MLflow Model Registry

- **Purpose**: Store, version, and govern ML models
- **Aliases**: Champion, Challenger, Production, Staging
- **Integration**: Now part of Unity Catalog
- **Auto-logging**: Use `mlflow.autolog()` for automatic metric capture

12.3 Handling Imbalanced Data

Technique	Description
Oversampling	Duplicate minority class examples
Undersampling	Remove majority class examples
SMOTE	Synthetic Minority Over-sampling Technique—create synthetic minority examples
Class Weights	Adjust loss function to penalize minority misclassification more

Table 11: *Techniques for Imbalanced Data*

13 Advanced Sharing: ABAC with Delta Sharing

Definition:

ABAC in Delta Sharing **Attribute-Based Access Control (ABAC)** policies now carry over during Delta Sharing. When a provider shares a table with ABAC policies (e.g., PII masking), the recipient automatically inherits those policies.

Key Information

Benefits:

- No separate policy configuration needed on recipient side
- Consistent data protection across organizations
- Share tables, views, and materialized views with embedded policies
- Row-level security and column masking preserved

13.1 Sharing to Iceberg Clients

Through **UniForm** (Delta with Iceberg metadata) or **Managed Iceberg**:

- Delta tables generate both Delta and Iceberg metadata
- External tools that only understand Iceberg can read the data
- Delta Sharing protocol works for both formats
- Zero-copy sharing—no data duplication

14 Course Summary and Next Steps

Key Summary

Key Takeaways from Lecture 14:

1. **Databricks One**: Unified experience for business users across workspaces
2. **Lakebase**: Revolutionary OLTP with scale-to-zero compute
3. **AI/BI**: Genie for "what" questions, Research Mode for "why"
4. **Agent Bricks**: Low-code AI agent creation
5. **Lakehouse Apps**: Build secure, interactive applications on data
6. **MCP**: Connect agents to external tools and data
7. **Quiz Review**: Broadcast variables are immutable and local to workers
8. **Streaming Review**: Use `availableNow` instead of deprecated `once`

14.1 Upcoming Events

- **Next Lecture**: Guest speakers from industry—highly recommended attendance
- **Assignment 4**: Review the final assignment requirements
- **Final Project**: Consider building a Lakehouse App for extra credit

Term	Definition
OLTP	Online Transaction Processing—real-time transactional systems
OLAP	Online Analytical Processing—analytical data warehousing
Lakebase	Databricks' managed PostgreSQL for OLTP workloads
Databricks One	Unified business user experience across workspaces
Genie	Natural language interface for data querying
Research Mode	Hypothesis-driven deep analysis mode in Genie
Agent Bricks	Low-code framework for building AI agents
MCP	Model Context Protocol—standard for agent-tool integration
Lakehouse Apps	Web applications built on the Databricks platform
UniForm	Delta tables with Iceberg metadata compatibility
ABAC	Attribute-Based Access Control
SCD	Slowly Changing Dimensions—dimension management pattern
AQE	Adaptive Query Execution—Spark runtime optimization
Checkpoint	Streaming progress tracking for fault tolerance

Glossary

One-Page Summary

CSCI E-103 Lecture 14: Databricks Roadmap & Emerging Features

1. Databricks One & Portal Experiences

- Three portals: Lakehouse (data professionals), Databricks One (business users), Lakebase (OLTP)
- Account-level access with vanity URLs and domain organization

2. Lakebase: OLTP Revolution

- Managed PostgreSQL with storage-compute separation
- Scale-to-zero: Pay only when compute is used
- Production/Development branching like Git
- Bidirectional sync with Lakehouse

3. AI/BI Enhancements

- Genie: "What" questions → SQL → Answers
- Research Mode: "Why" questions → Hypothesis exploration
- Document upload: Combine PDFs/CSVs with structured data

4. Agent Bricks & MCP

- AutoML-like agent creation: Point to data, specify task
- MCP servers in marketplace for agent integrations
- Governed by Unity Catalog

5. Quiz 2 Key Points

- Broadcast variables: Immutable + Local to each worker
- Repartition (increase) vs Coalesce (decrease)