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 → Silver transformations

2. Materialized Views:

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

```
-- 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
```

```
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.

```
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