Databricks Lakeflow

What's new in pipelines and how it's changing Data Engineering

Agenda

Introduction to Lakeflow Pipelines

Definition and declarative approach

Syntax and fundamentals

CREATE OR REFRESH, Streaming Tables vs Materialized Views

Expectations

Data quality, reaction types, best practices

Comparison with other solutions

Lakeflow vs Standard Spark, Automation

Compute

Serverless vs Classic Clusters

Lakeflow UI

DAG View, Monitoring

Demo

Practical application

Microsoft CERTIFIED TRAINER MCT

Microsoft CERTIFIED AZURE AI ENGINEER ASSOCIATE

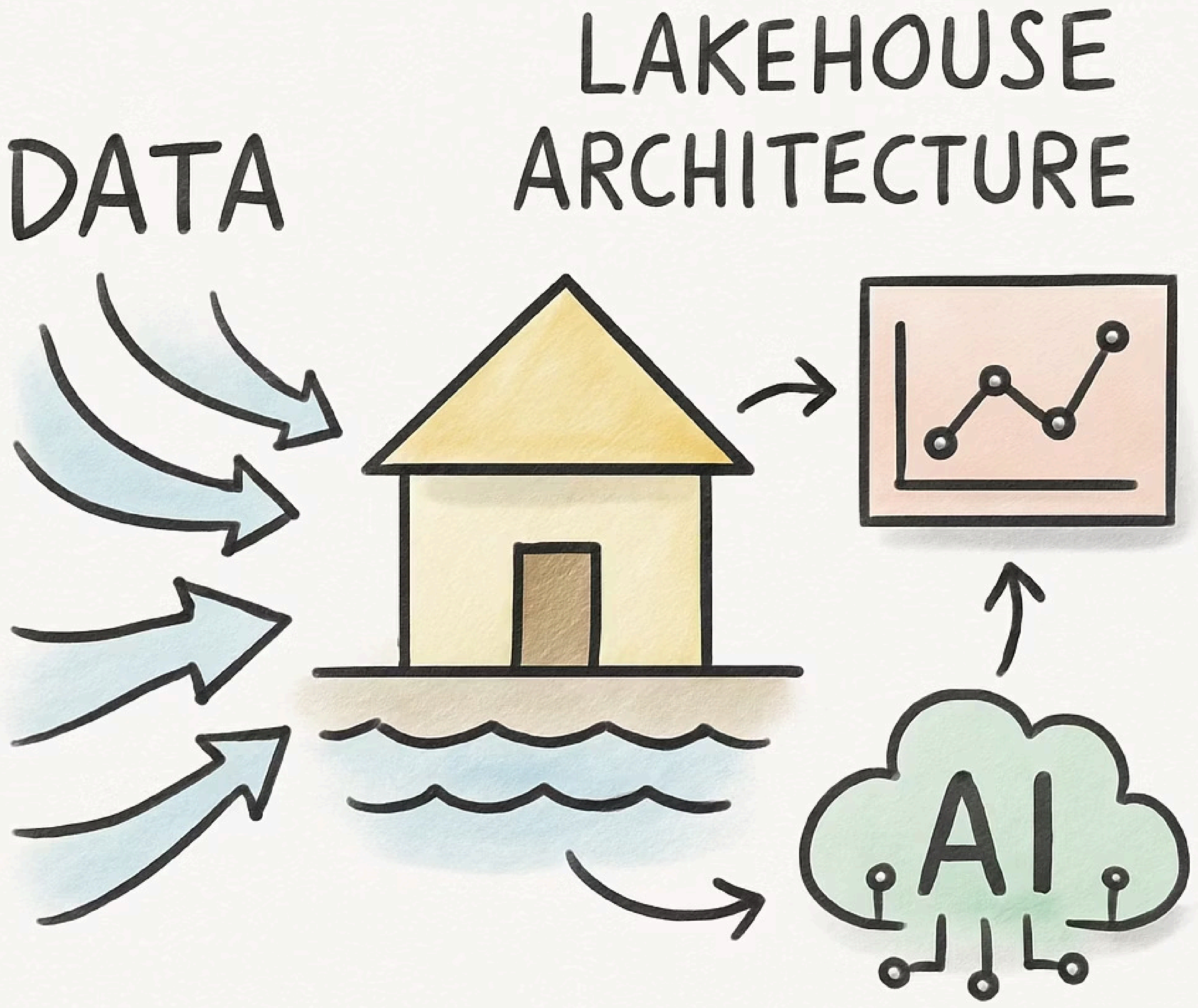
Microsoft CERTIFIED FABRIC DATA ENGINEER ASSOCIATE

databricks CERTIFIED Data Engineer Professional

Krzysztof Burejza

Azure Data Engineer

Data engineer with 10 years of experience working with data, from SQL Server to cloud platforms such as Azure, Databricks, and Microsoft Fabric. He focuses on transforming raw data into valuable solutions, building pipelines, data warehouses, and lakehouses. An enthusiast of new technologies at the intersection of data and AI, which make data engineering even more fascinating.


LAKEHOUSE ARCHITECTURE
DATA
AI

What are Lakeflow Pipelines?

Lakeflow (formerly Delta Live Tables) is a framework for **declaratively building data pipelines** in Databricks. Instead of writing orchestration code, we define the **final outcome** – the framework manages execution, monitoring, and data quality.

Traditional Approach

- Manual state management
- Custom retry logic
- Custom checkpoints
- Manual dependency tracking
- Separate quality control tools

Lakeflow Pipelines

- Declarative table definitions
- Automatic retries and error handling
- Built-in checkpointing
- Automatic lineage graph
- Built-in Expectations (quality rules)

What happened to Delta Live Tables (DLT)?

Delta Live Tables (DLT) has been updated to **Lakeflow Spark Declarative Pipelines (SDP)**. Existing DLT code is fully compatible, but it is recommended to update to the new API names for future features and compatibility with Apache Spark™ Declarative Pipelines (from Spark 4.1).

Key changes in the Python API:

~~@dlt~~ → @dp

Replacing the library reference.

~~@table~~ (general) → @table (streaming) / @materialized_view

A dedicated decorator for materialized views has been introduced, increasing readability.

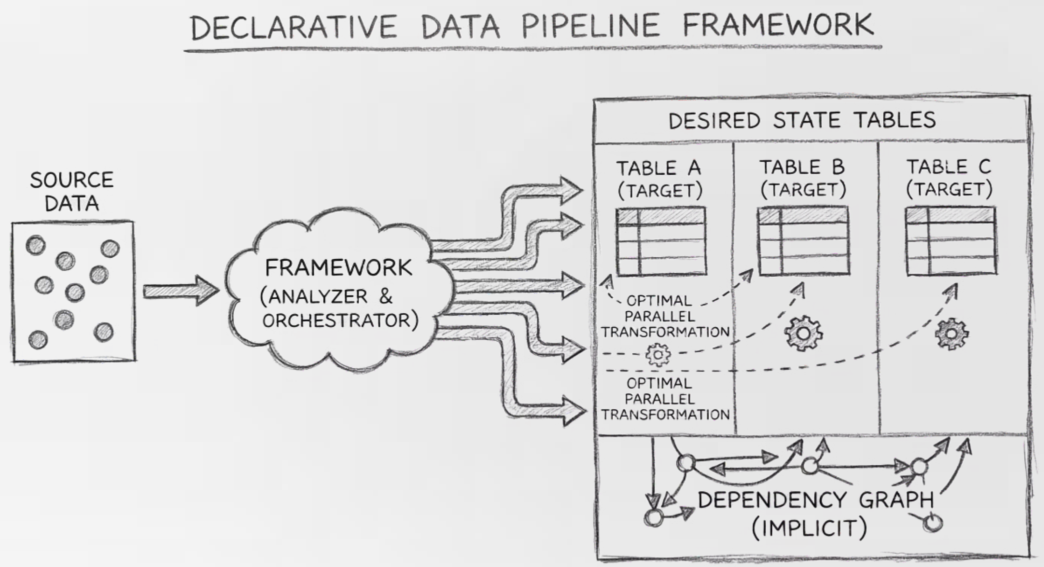
~~@view~~ → @temporary_view

Clear distinction for temporary views.

Note: You may still encounter references to the DLT name in some places in Databricks (e.g., classic SKUs, event log schemas). The existing Python API with DLT names is still supported, but transitioning to the new naming convention is recommended.

Declarative Approach - "WHAT" instead of "HOW"

In the declarative approach, we describe the desired state of the data (what we want to achieve), instead of imperative steps (how to do it). Lakeflow independently analyzes dependencies and optimizes the execution of transformations.


DECLARATIVE DATA PIPELINE FRAMEWORK
SOURCE DATA
FRAMEWORK (ANALYZER & ORCHESTRATOR)
DESIRED STATE TABLES
TABLE A (TARGET)
TABLE B (TARGET)
TABLE C (TARGET)
OPTIMAL PARALLEL TRANSFORMATION
OPTIMAL PARALLEL TRANSFORMATION
DEPENDENCY GRAPH (IMPLICIT)

What Makes Up Lakeflow Pipelines?

Lakeflow Pipelines is a framework integrating key aspects of data engineering, enabling the building of reliable and efficient pipelines based on the Lakehouse paradigm.

Declarative Definitions

Focus on **"WHAT"** you want to achieve with data, not **"HOW"** to implement it. Lakeflow automatically manages execution logic.

Automatic Orchestration

The framework independently manages task order, dependencies, retries in case of errors, and progress monitoring.

Built-in Data Quality

Define data quality rules directly in the pipeline code, ensuring automatic validation and control over data.

Streaming and Materialized Views

A unified approach to processing streaming and batch data within a single pipeline.

Simplified Compute

Serverless options minimize the need for infrastructure management, automatically scaling resources based on load.

CREATE OR REFRESH Syntax

Streaming Table

For append-only sources (logs, events, transactions):

```
CREATE OR REFRESH STREAMING TABLE bronze_orders
AS SELECT *, current_timestamp() as ingestion_time
FROM read_files(
  "/Volumes/raw/orders/", format=>"csv", header=>true,
inferSchema=>true
)
```

A streaming table automatically processes new files.

Materialized View

For full refresh or incremental update transformations:

```
CREATE OR REFRESH MATERIALIZED VIEW silver_orders
AS SELECT
  order_id, customer_id, order_date, total_amount, items
FROM STREAM(bronze_orders)
WHERE order_status != 'cancelled'
```

`STREAM()` indicates incremental processing.

When to use Streaming vs. Materialized?

Streaming Table

- Data is append-only (events, logs)
- Exactly-once semantics required
- Data arrives in real-time
- Bronze/Raw Layer - retaining all data
- CDC from transactional databases

Examples: application logs, Kafka streams, CDC from PostgreSQL

Materialized View

- UPDATE/DELETE required (SCD2, deduplication)
- Aggregations and complex joins
- Gold Layer - business metrics
- Snapshot Tables - current state only
- Feature Tables for ML

Examples: Calculating CLV, daily active users, product recommendation features

Data Flows in Lakeflow Pipelines

A flow is a combination of a query and a target table, processing data in batch or streaming mode, incrementally in a Lakeflow Spark Declarative Pipelines pipeline.

Definition and Types of Flows

- Flows are created **automatically** when defining a table or view (e.g., `CREATE OR REFRESH STREAMING TABLE`).
- They can be defined **explicitly** (`CREATE FLOW` / `@dp.append_flow`) for complex scenarios.
- Standard flows (append flows) **add new rows**, ideal for data ingestion and transformation.
- **Auto CDC** flows are used to ingest data with changes (Change Data Capture).

Applications and Benefits

- Every pipeline update triggers a flow that refreshes the tables.
- This can be an **incremental refresh** (only new records) or a **full refresh**.
- Enables **combining data from multiple sources** into one target table (e.g., adding new regions) without a full refresh.
- Allows **backfilling historical data** using the `INSERT INTO ONCE` syntax.

Multiple Flows to a Single Target

Creating a table and assigning a flow

You can first create a streaming table and then define a flow to it. The example below shows how to independently create a table and assign a single flow, achieving the same results as with default flow creation.

```
-- utwórz tabelę strumieniową
CREATE OR REFRESH STREAMING TABLE customers_silver;

-- dodaj przepływ do tabeli customers_silver
CREATE FLOW customers_silver_flow
AS INSERT INTO customers_silver BY NAME
SELECT * FROM STREAM(customers_bronze);
```

Combining data from multiple sources

Using multiple append flows (`append_flow` in Python or `CREATE FLOW...INSERT INTO` in SQL) is ideal for combining data from different sources into a single streaming table. This ensures incremental updates, which are more efficient than using a `UNION` clause.

```
-- utwórz tabelę strumieniową
CREATE OR REFRESH STREAMING TABLE customers_us;

-- dodaj pierwszy przepływ dopisujący z danych z zachodniego USA
CREATE FLOW append_us_west
AS INSERT INTO customers_us BY NAME
SELECT * FROM STREAM(customers_us_west);

-- dodaj drugi przepływ dopisujący z danych ze wschodniego USA
CREATE FLOW append_us_east
AS INSERT INTO customers_us BY NAME
SELECT * FROM STREAM(customers_us_east);
```

Important: Data quality expectations should be defined on the **target table** (e.g., `customers_us`), not in the `@dp.append_flow` flow definition.

Flows are identified by name, used to manage streaming checkpoints. Renaming an existing flow is treated as creating a new one, resetting its checkpoints. Additionally, the same flow name cannot be reused within a single pipeline.

Backfilling Historical Data

Lakeflow Pipelines makes it easy to backfill historical data in existing pipelines. This is crucial when you need to process data from periods not included in the initial pipeline configuration.

Defining the Scope

Create a new flow, for example, by removing or changing `modifiedAfter`.

One-time run

Use the `INSERT INTO ONCE` syntax for a one-time execution and to avoid duplicates.

Integration

New historical data will be seamlessly integrated into the target table.

```
-- create the streaming table
CREATE OR REFRESH STREAMING TABLE registration_events_raw;

-- original incremental flow (e.g., from 2025 onwards)
CREATE FLOW registration_events_raw_incremental
AS INSERT INTO registration_events_raw BY NAME
SELECT * FROM STREAM read_files(
  "/Volumes/gc/demo/apps_raw/event_registration/*",
  format => "json",
  modifiedAfter => "2024-12-31T23:59:59.999+00:00"
);

-- one-time backfill for 2024
CREATE FLOW registration_events_raw_backfill_2024
AS INSERT INTO ONCE registration_events_raw BY NAME
SELECT * FROM read_files(
  "/Volumes/gc/demo/apps_raw/event_registration/year=2024/*",
  format => "json"
);
```

CDC APPLY in Lakeflow Pipelines (SCD Type 1 and 2)

What is AUTO CDC INTO?

`AUTO CDC INTO` automates data change management, replacing manual `MERGE INTO` logic. It simplifies data integration from transactional systems where tracking record evolution is crucial.

`KEYS (column)`

Specifies one or more columns that form a unique record key.

`APPLY AS DELETE WHEN ...`

Defines a logical condition for when a source record should be treated as a deletion.

`APPLY AS TRUNCATE WHEN ...`

Defines a condition that causes all records in the target table to be deleted before applying changes.

`SEQUENCE BY column`

Used to handle out-of-order events by specifying the latest change.

`COLUMNS * EXCEPT (columns)`

Indicates all source columns to include, except for those listed.

`STORED AS SCD TYPE 1`

Changes update records, overwriting old values (only the latest version).

`STORED AS SCD TYPE 2`

Changes are stored as new records with `__START_AT` and `__END_AT` columns to track history.

SCD Type 1: Retaining the Latest Data

In SCD Type 1, data updates overwrite existing records in the target table, always ensuring the latest version of the data. Lakeflow Pipelines automatically manages this logic.

```
CREATE OR REFRESH STREAMING TABLE target;

CREATE FLOW flowname AS AUTO CDC INTO target
  FROM stream(cdc_data.users)
  KEYS (userId)
  APPLY AS DELETE WHEN operation = "DELETE"
  APPLY AS TRUNCATE WHEN operation = "TRUNCATE"
  SEQUENCE BY sequenceNum
  COLUMNS * EXCEPT (operation, sequenceNum)
  STORED AS SCD TYPE 1;
```

After applying SCD Type 1 updates, the target table contains the latest records:

userId	name	city
124	Raul	Oaxaca
125	Mercedes	Guadalajara
126	Lily	Cancun

SCD Type 2: Tracking Historical Changes

SCD Type 2 tracks all changes, preserving historical versions of records. Each change creates a new record with a new validity period (`__START_AT` and `__END_AT`), which is crucial for analyzing changes over time.

```
CREATE OR REFRESH STREAMING TABLE target;

CREATE FLOW target_flow AS AUTO CDC INTO target
  FROM stream(cdc_data.users)
  KEYS (userId)
  APPLY AS DELETE WHEN operation = "DELETE"
  SEQUENCE BY sequenceNum
  COLUMNS * EXCEPT (operation, sequenceNum)
  STORED AS SCD TYPE 2;
```

After applying SCD Type 2 updates, the target table contains historical records:

userId	name	city	__START_AT	__END_AT
123	Isabel	Monterrey	1	5
123	Isabel	Chihuahua	5	6

Automated Maintenance Tasks

Lakeflow Pipelines autonomously performs key maintenance tasks on managed tables, utilizing advanced predictive optimization. This ensures better query performance and cost reduction by removing stale data and organizing it.

Key Operations

The pipelines perform **OPTIMIZE** (file compaction) and **VACUUM** (removal of old data) operations, ensuring optimal storage utilization.

Intelligent Scheduling

Maintenance tasks are scheduled by the predictive optimization mechanism and run only after a pipeline update.

Benefits

Automatic maintenance keeps tables in optimal condition, resulting in increased query performance, reduced resource consumption, and lower storage costs.

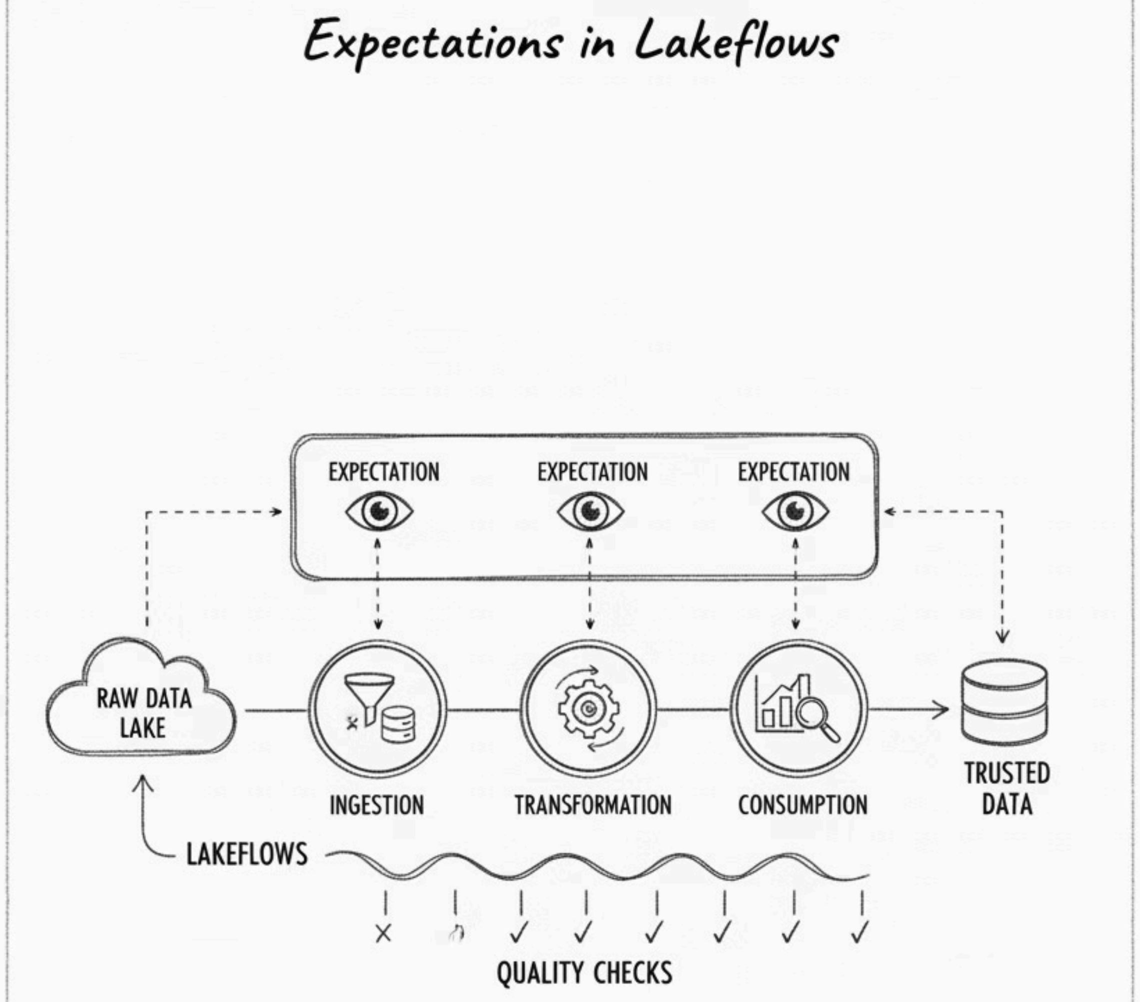
To learn more about the frequency of predictive optimization and associated costs, refer to the **predictive optimization system table**.

Expectations

In the context of Lakeflow Pipelines, "Expectations" define data quality rules and tests that are automatically verified during processing. They ensure data reliability and early anomaly detection.

Data Quality - Expectation

Expectations are declarative data quality rules, built directly into the pipeline. They act like unit tests for data, automatically verifying conditions with each processing. They block the propagation of erroneous data to downstream tables.



Types of Violation Reactions

DELETE ROW

Rows that do not meet the condition are automatically skipped. Use when erroneous data can be safely ignored.

```
CONSTRAINT valid_email
  EXPECT (email LIKE '%@%')
  ON VIOLATION DROP ROW
```

CANCEL UPDATE

The entire batch is rejected if any row violates the condition. Apply to critical business rules.

```
CONSTRAINT positive_amount
  EXPECT (amount > 0)
  ON VIOLATION FAIL UPDATE
```

TRACK ONLY (DEFAULT)

The violation is recorded in metrics, but data continues to flow. Ideal for monitoring quality without blocking the pipeline.

```
CONSTRAINT recent_data
  EXPECT (
    event_date >= current_date() - 7
  )
```

Metrics: All violations are automatically recorded in the pipeline event log and are available in the user interface as metrics for each table.

Best Practices for Expectations

Why is this important?

Good practices facilitate the application, updating, and auditing of expectations rules across multiple datasets and pipelines, without modifying code. They increase validation consistency, reduce maintenance costs, and improve solution portability.

Definition Separation

Store expectation definitions separately from pipeline logic

Tagging

Add custom tags to group related expectations

Reusability

Apply the same expectations across multiple pipelines

Centralized Expectations Repository: Delta Table

The rules table centralizes validation rules, storing the name, SQL constraint, and a tag for filtering.

```
CREATE OR REPLACE TABLE rules AS
SELECT col1 AS name, col2 AS constraint, col3 AS tag
FROM (
  VALUES
    ("website_not_null",
     "Website IS NOT NULL",
     "validity"),
    ("fresh_data",
     "to_date(updateTime,'M/d/yyyy h:m:s a') > '2010-01-01'",
     "maintained"),
    ("social_media_access",
     "NOT(Facebook IS NULL AND Twitter IS NULL AND Youtube IS NULL)",
     "maintained")
)
```

It simplifies rule management via SQL, allowing easy addition of new constraints and categorization by business tags.

Applying rules in a Python pipeline

The `get_rules()` function retrieves validation rules from the table, matching them by tag. The `@dp.expect_all_or_drop()` decorator automatically removes records that do not meet these criteria.

```
from pyspark import pipelines as dp
from pyspark.sql.functions import expr, col

def get_rules(tag):
 df = spark.read.table("rules").filter(col("tag") == tag).collect()
 return {row['name']: row['constraint'] for row in df}

@dp.table
@dp.expect_all_or_drop(get_rules('validity'))
def raw_farmers_market():
 return (
 spark.read.format('csv').option("header", "true")
 .load('/databricks-datasets/data.gov/farmers_markets_geographic_data/')
 )

@dp.table
@dp.expect_all_or_drop(get_rules('maintained'))
def organic_farmers_market():
 return spark.read.table("raw_farmers_market").filter(expr("Organic = 'Y'"))
```

Schema Evolution

This pattern enables the migration of data sources and the management of multiple versions, ensuring backward compatibility and data quality.

```
CREATE OR REFRESH MATERIALIZED VIEW evolving_table(
  CONSTRAINT valid_migrated_data EXPECT (
    (col1 IS NOT NULL AND col2 IS NOT NULL) AND
    (CASE WHEN col3 IS NOT NULL THEN col3 > 0 ELSE TRUE END)
  ) ON VIOLATION FAIL UPDATE
) AS
SELECT * FROM new_source
UNION
SELECT *, NULL as col3 FROM legacy_source;
```

`UNION` combines new and old sources. `CASE` handles optional columns. `ON VIOLATION FAIL UPDATE` stops the pipeline on critical errors.

Row Count Verification

Monitoring the number of rows in materialized views using `COUNT(*)` allows for quick record verification and anomaly identification, helping to maintain data quality.

```
CREATE OR REFRESH MATERIALIZED VIEW
count_verification(
  CONSTRAINT no_rows_dropped
  EXPECT (a_count == b_count)
) AS
SELECT * FROM
  (SELECT COUNT(*) AS a_count FROM table_a),
  (SELECT COUNT(*) AS b_count FROM table_b)
```

The constraint `EXPECT (a_count == b_count)` automatically detects discrepancies in row counts between source tables.

Detecting Missing Records and Key Uniqueness

Missing Record Detection

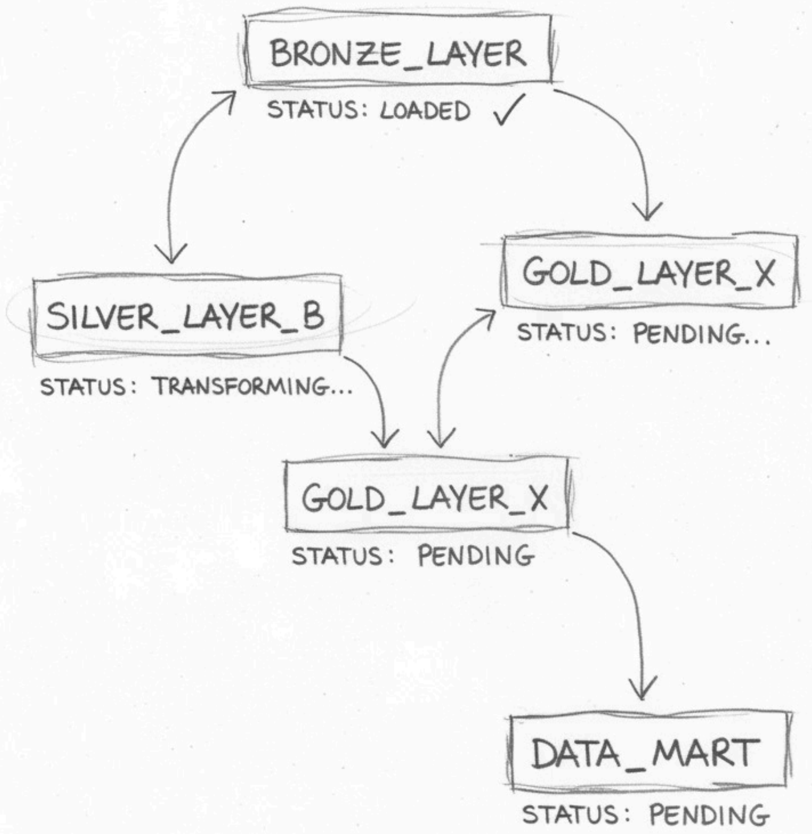
```
CREATE OR REFRESH MATERIALIZED VIEW
report_compare_tests(
  CONSTRAINT no_missing_records
  EXPECT (r_key IS NOT NULL)
) AS
SELECT v.*, r.key as r_key
FROM validation_copy v
LEFT OUTER JOIN report r
ON v.key = r.key
```

LEFT JOIN reveals records present in validation but missing in the report.

Primary Key Validation

```
CREATE OR REFRESH MATERIALIZED VIEW
report_pk_tests(
  CONSTRAINT unique_pk
  EXPECT (num_entries = 1)
) AS
SELECT pk, count(*) as num_entries
FROM report
GROUP BY pk
```

GROUP BY detects primary key duplicates by counting occurrences.



DATABRICKS DATA PIPLINE
TABLE DEPENDENIES & STATUS

DAG View in Lakeflow Interface

The Databricks UI automatically generates a visual DAG (Directed Acyclic Graph) of all tables in the pipeline. Each node shows status (running/success/failure), processing metrics, and data quality violations. You can click on a node to see the definition code, data preview, and detailed logs.

What the DAG shows

- Table dependencies (upstream → downstream)
- Real-time status of each table
- Row count and data volume
- Processing duration
- Summary of expectation violations

Interaction

- Clicking a node → table details
- Hovering over → quick metrics preview
- Timeline view → historical runs
- Detailed error analysis
- Schema evolution tracking

Standard Spark vs Lakeflow Pipelines

Feature	Standard Spark SQL/Python	Lakeflow Pipelines (DLT)
Programming model	SQL/DataFrame API, more boilerplate code.	Extended SQL/Python with DLT (`CREATE OR REFRESH`, `EXPECT`).
Infrastructure automation	Manual cluster configuration, checkpoint management.	Automatic cluster management, checkpoint management, refresh.
Data quality	Custom implementation of data quality rules.	Built-in expectations (`EXPECT`) with configurable reactions.
Monitoring and observability	Integration with external tools or custom implementation.	Automatic DAG, metrics, logs, and statuses in UI.
Dependency management	External orchestrator (e.g., Airflow).	Automatic detection and management of table dependencies.
Performance optimization	Manual optimization (e.g., partitioning, indexing).	Automatic Photon optimizations, caching, and scaling.

Automation – what do we get for free?

Data Lineage

Automatic table dependency graph indicates sources and consumers, propagating changes upstream.

State Management

Automatic checkpoint management and state processing, eliminating manual configuration and file tracking.

Incremental Refresh

The pipeline processes only changed data (deltas), saving resources and shortening execution time by 10-100 times.

Error Handling

Automatic retries and graceful degradation in case of errors, preventing pipeline hangs due to faulty records.

Observability

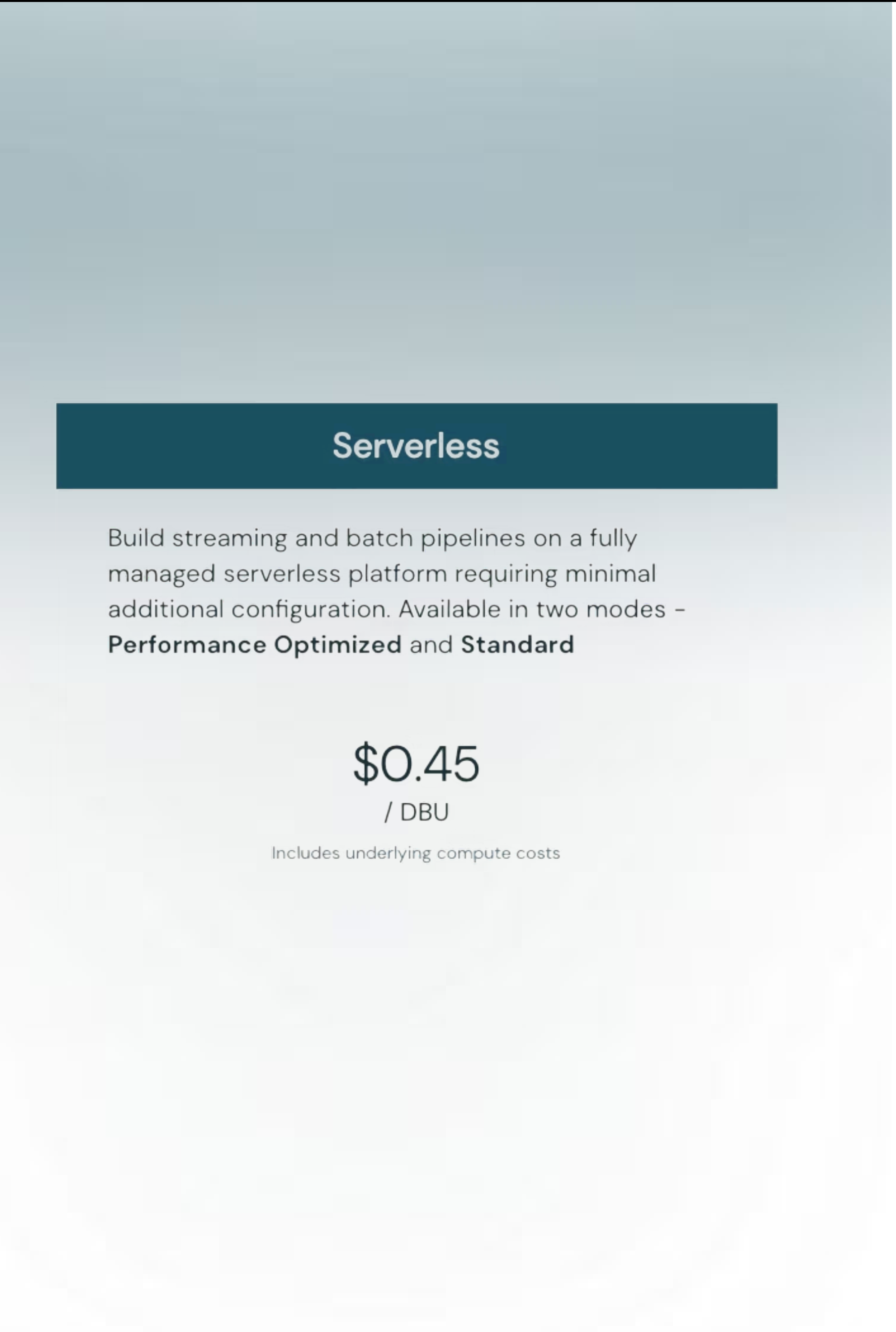
Built-in metrics (row count, processing time, data quality) available in UI, without additional code instrumentation.

Optimization

Automatic execution plan optimization and result caching, ensuring performance without manual intervention.

Compute

Lakeflow Pipelines offers two compute options, tailored to different data workloads.



Serverless

Build streaming and batch pipelines on a fully managed serverless platform requiring minimal additional configuration. Available in two modes - **Performance Optimized** and **Standard**

$0.45

/ DBU

Includes underlying compute costs

Serverless

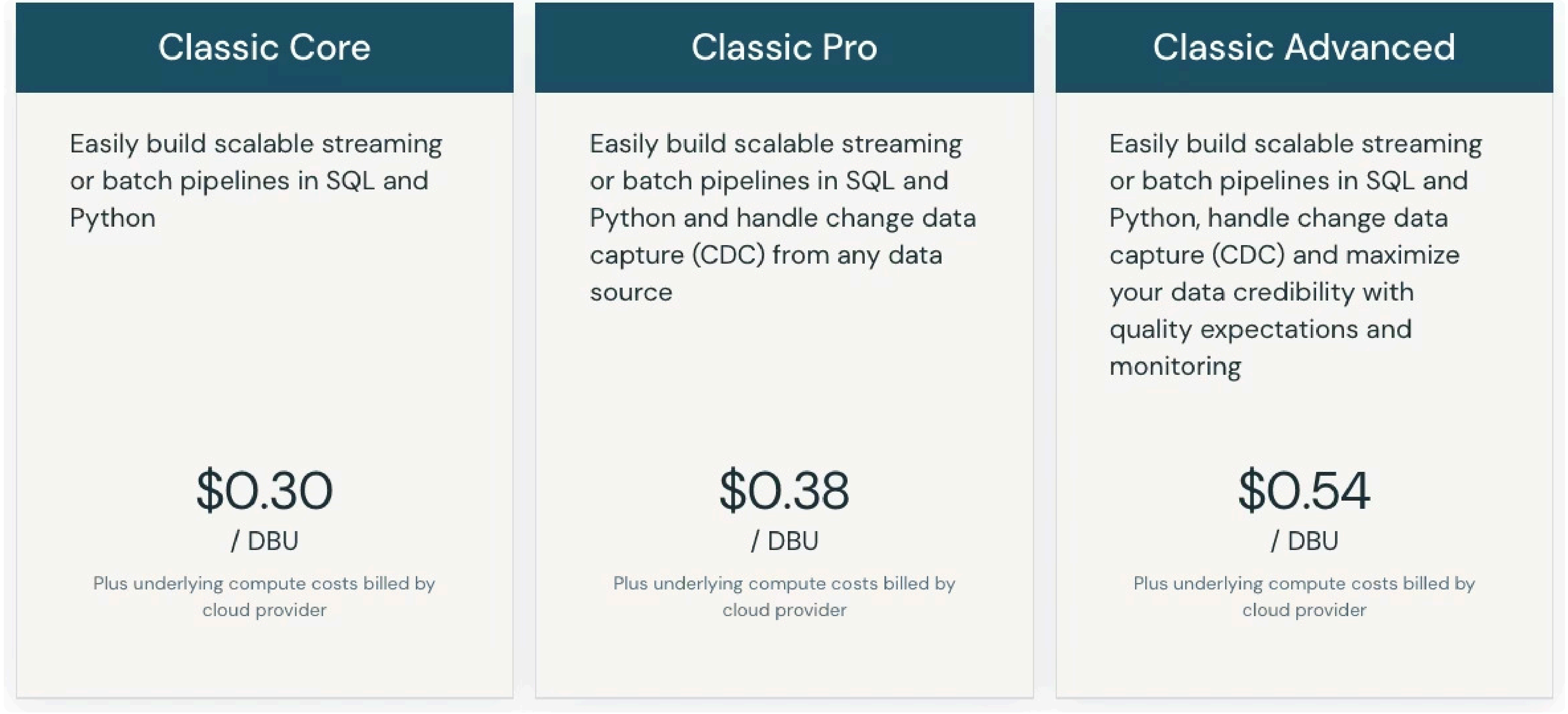
Serverless for Lakeflow Pipelines is a fully managed option. It eliminates cluster configuration and the price is included in the DBU cost, with no additional cloud infrastructure fees. Two variants are available:

- **Standard:** up to 70% cheaper, start-up time 4-6 minutes.
- **Performance Optimized:** faster start-up (<1 minute), with Enzyme (incremental refresh), Stream Pipelining, and automatic scaling preventing OOM issues.

Both variants have the Photon engine automatically enabled, ensure automatic instance selection and scaling. They also support advanced features: CDC, SCD Type 2, and data quality rules.

Classic Clusters

Classic mode for Lakeflow Pipelines uses advanced clusters that require manual configuration. This variant involves separate payment for cloud instances and the cost of Lakeflow Pipelines DBU. Cluster startup time ranges from 1 to 6 minutes, and the Photon engine is optional. Please note that advanced features such as CDC, SCD Type 2, and data quality rules are unavailable in Classic Core.



Classic Core	Classic Pro	Classic Advanced
Easily build scalable streaming or batch pipelines in SQL and Python	Easily build scalable streaming or batch pipelines in SQL and Python and handle change data capture (CDC) from any data source	Easily build scalable streaming or batch pipelines in SQL and Python, handle change data capture (CDC) and maximize your data credibility with quality expectations and monitoring
$0.30 / DBU	$0.38 / DBU	$0.54 / DBU
Plus underlying compute costs billed by cloud provider	Plus underlying compute costs billed by cloud provider	Plus underlying compute costs billed by cloud provider

Comparison: Classic vs Serverless

	Classic	Serverless Standard	Serverless Performance Optimized
Typical Use Case	Traditional workloads, full control	General data pipelines, lower costs	High-performance pipelines, fast start
Change Data Capture (CDC)	Not Available	Yes	Yes
Slowly Changing Dimensions (SCD) Type 2	Not Available	Yes	Yes
Data Quality Expectation Rules	Not Available	Yes	Yes
Photon Engine	Optional (2.5X DBU)	Automatically	Automatically
Automatic Instance Selection	Manual	Automatic	Automatic
Auto-scaling (OOM)	No	No	Yes
Enzym (Incremental Refresh)	No	No	Yes
Stream Pipelines	No	No	Yes
Startup Time	4-6 min	4-6 min	<1 min
Total Cost	Instances paid separately	Included in price (up to 70% cheaper)	Included in price

Refresh Techniques in Lakeflow Pipelines

Lakeflow Pipelines automatically manages the refreshing of materialized views. Below are the available refresh methods and their characteristics:

Technique	Incremental?	Description
`FULL_RECOMPUTE`	No	View fully recomputed from scratch.
`NO_OP`	N/A	View not updated (no changes in the base table).
`ROW_BASED`, `PARTITION_OVERWRITE`, `WINDOW_FUNCTION`, `APPEND_ONLY`, `GROUP_AGGREGATE`, `GENERIC_AGGREGATE`	Yes	View refreshed incrementally, minimizing resource usage and time.

Lakeflow Pipelines vs MLV (Fabric)

In the context of modern lakehouse architectures, Materialized Lake Views (MLV) and Lakeflow serve to automate data pipelines, but they differ in scope and platform.

Feature	Lakeflow Pipelines (Databricks)	Microsoft Fabric MLV
Ecosystem	Databricks (Unified Platform)	Microsoft Fabric (SaaS)
Scope	Entire data lifecycle (ingest, ETL, orchestration)	Single object/transformation
Language	SQL and Python	SQL (mainly)
Main Goal	Scalable, professional data pipelines	Fast data delivery, simplification of Medallion architecture
Mechanism	Automatic orchestration, incremental processing, autoscaling, real-time	Declarative approach – system manages refresh
Object Types	Streaming Tables and Materialized Views	Materialized Lake Views (Delta format in OneLake)
Data Quality	Built-in EXPECT (configurable reactions)	Built-in monitoring (e.g., CONSTRAINT ON MISMATCH)
CDC / SCD Type 2	Built-in syntax (APPLY CHANGES INTO)	Requires manual implementation
Monitoring	Automatic DAG, metrics, logs	Automatic lineage visualization, monitoring in Fabric

DEMO