

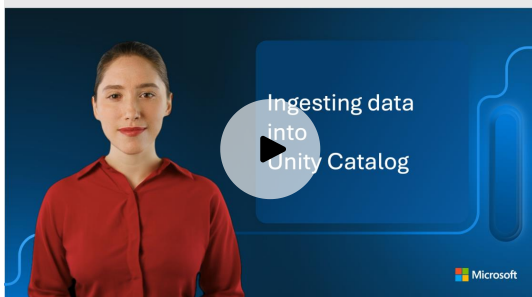
Ingest data into Unity Catalog

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/1-introduction>

Introduction

Completed



Data engineers face a fundamental challenge: getting data from diverse sources into a unified analytics platform efficiently and reliably. Whether your organization works with batch files in cloud storage, streaming events from Kafka, or transactional data from operational databases, you need ingestion methods that handle each scenario while maintaining data quality and governance. **Unity Catalog** in Azure Databricks provides the foundation for this unified approach, offering centralized governance for all your ingested data.

Azure Databricks offers multiple ingestion techniques, each optimized for specific data patterns and requirements. **Lakeflow Connect** provides managed connectors for common enterprise sources like SQL Server and Salesforce, eliminating the need for custom extraction code. **Notebooks** give you full control over ingestion logic using Python, SQL, or Scala. **SQL commands** like `COPY INTO` and `CREATE TABLE AS SELECT` offer declarative approaches for file-based ingestion. For real-time workloads, **Spark Structured Streaming** and **Auto Loader** process data continuously as it arrives, with exactly-once processing guarantees.

Understanding when to apply each technique enables you to build robust data pipelines that scale with your organization. You'll learn how to configure managed connectors for enterprise data sources, write notebook code for custom ingestion scenarios, use SQL commands for file-based batch loads, process change data capture feeds incrementally, stream data from message buses like Kafka and Event Hubs, configure Auto Loader for automatic file detection, and orchestrate ingestion workflows with Lakeflow Spark Declarative Pipelines.

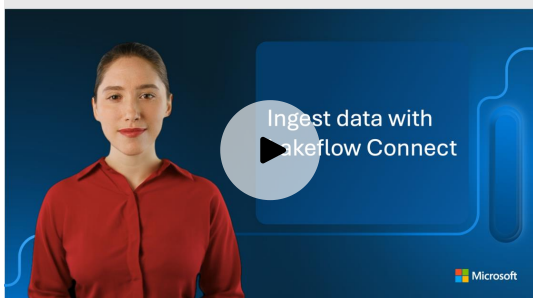
By the end of this module, you'll be equipped to choose and implement the right ingestion approach for any data source, ensuring your data lands in Unity Catalog tables with proper governance and optimal performance.

2. Ingest data with Lakeflow Connect

<https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/2-ingest-data-lakeflow-connect>

Ingest data with Lakeflow Connect

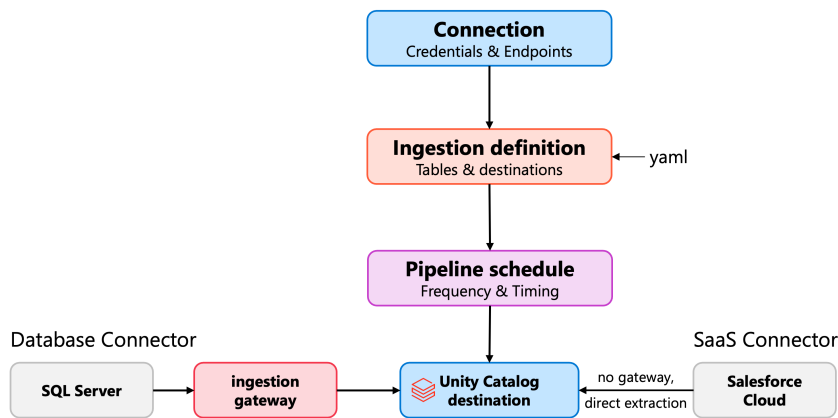
Completed



Data engineers face a common challenge: getting data from diverse sources into analytics platforms efficiently and reliably. Whether your data lives in SQL Server, Salesforce, SharePoint, or cloud storage, you need consistent, governed pipelines that minimize manual effort. **Lakeflow Connect** addresses this challenge by providing managed ingestion connectors that bring data directly into Unity Catalog tables.

Understand ingestion pipelines

Lakeflow Connect is a collection of managed connectors in Azure Databricks that simplify data ingestion from external sources. Rather than writing custom extraction code, you configure pipelines through either a graphical interface or declarative definitions.



Each pipeline consists of three core components:

- **Connection:** Stores credentials and endpoint information for the source system. You create connections once and reuse them across multiple pipelines.
- **Ingestion definition:** Specifies which tables or objects to extract, along with their destination catalogs and schemas in Unity Catalog.
- **Pipeline schedule:** Controls when and how frequently data flows from source to destination.

When you create an ingestion pipeline for a database connector like SQL Server, Lakeflow Connect also creates an **ingestion gateway**. This gateway continuously extracts change data from the source database and stages it for processing. For SaaS connectors like Salesforce, the connector handles extraction directly without a separate gateway.

Configure ingestion pipelines

You can create ingestion pipelines through the Databricks UI, Asset Bundles, notebooks, or the CLI. The UI provides the simplest approach for initial setup.

Consider a scenario where you need to ingest customer order data from SQL Server. In the Azure Databricks workspace, you navigate to **Data Ingestion** and select **SQL Server**. The wizard guides you through creating an ingestion gateway, selecting the connection, and choosing the tables to ingest. You specify a destination catalog and schema where the ingested data lands as streaming tables.

```

resources:
  pipelines:
    pipeline_sqlserver:
      name: customer-orders-pipeline
      catalog: sales
      schema: bronze
      ingestion_definition:
        ingestion_gateway_id: ${resources.pipelines.gateway.id}
      objects:
        - table:
            source_schema: dbo
            source_table: orders
  
```

```
destination_catalog: sales
destination_schema: bronze
```

With this configuration, the pipeline extracts the `orders` table from the source database and loads it into `sales.bronze.orders` in Unity Catalog.

Control data extraction behavior

Lakeflow Connect offers several options for controlling how data flows from source to destination. The most important choice is between **snapshot** and **incremental** extraction patterns.

For database connectors, the gateway uses **change data capture (CDC)** to track modifications at the source. When CDC is enabled on the source database, the gateway captures inserts, updates, and deletes incrementally. This means subsequent pipeline runs process only changed records rather than reloading entire tables.

The **SCD type** setting determines how the destination table handles changes:

- **SCD type 1:** Overwrites existing records when updates occur. The destination table always reflects the current state of the source data.
- **SCD type 2:** Preserves historical versions of records. When a row changes, the pipeline marks the previous version as inactive and adds the new version. This approach tracks how data evolves over time.

```
table_configuration:
  scd_type: SCD_TYPE_2
  sequence_by: last_modified
```

When you enable SCD type 2, the destination table includes `__START_AT` and `__END_AT` columns that record each version's active period.

Optimize column selection and refresh strategies

Not every column in your source tables may be relevant for analytics. Lakeflow Connect lets you specify exactly which columns to include or exclude from ingestion.

```
table_configuration:
  include_columns:
    - customer_id
    - order_date
    - total_amount
```

With explicit column selection, you reduce data transfer volumes and storage costs. If you use `include_columns`, newly added source columns won't automatically appear in the destination. With `exclude_columns`, new columns are included by default.

For situations where incremental processing can't recover from schema changes or data inconsistencies, you can trigger a **full refresh**. This operation clears the destination table and reloads all records from scratch. Lakeflow Connect minimizes downtime during full refresh by maintaining the existing data until the new snapshot completes.

You can also configure a **full refresh window** to schedule when these operations occur, avoiding disruption during peak business hours.

Scale with multiple destinations

A single ingestion pipeline can write to multiple destination catalogs and schemas. This capability proves useful when different teams need isolated copies of the same source data.

```
objects:
  - table:
      source_table: customers
      destination_catalog: sales
      destination_schema: analytics
  - table:
      source_table: customers
      destination_catalog: marketing
      destination_schema: campaigns
```

You can even ingest the same source table to multiple destinations within the same schema by specifying a custom `destination_table` name.

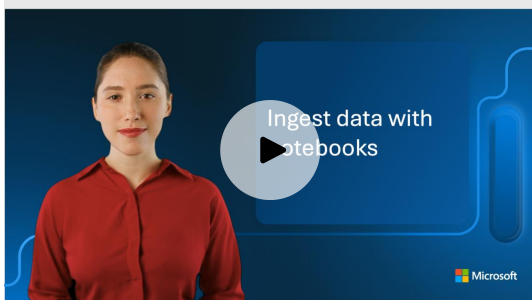
Once your pipelines are running, you can monitor ingestion progress, set up failure notifications, and adjust schedules through the pipeline details page. Lakeflow Connect automatically creates jobs for each schedule you define, allowing you to add additional processing tasks after ingestion completes.

3. Ingest data with notebooks

<https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/3-ingest-data-notebooks>

Ingest data with notebooks

Completed



Notebooks in Azure Databricks provide a flexible, code-driven approach for ingesting data from various sources. When graphical tools don't meet your requirements, or when you need custom logic for data transformation, notebooks give you complete control over the ingestion process. You can use Python (PySpark), SQL, Scala, or R to connect to data sources, apply transformations, and load data into Unity Catalog tables.

In this unit, you learn how to ingest data using notebooks for both batch and streaming scenarios.

Ingest batch data with DataFrames

DataFrames are the foundation for batch data ingestion in Databricks. A DataFrame is a distributed collection of data organized into named columns, similar to a table in a relational database. You use the `spark.read` API to load data from various sources into a DataFrame.

The following example reads a CSV file into a DataFrame:

```
df = (spark.read
      .format("csv")
      .option("header", "true")
      .option("inferSchema", "true")
      .load("/Volumes/catalog/schema/volume/data.csv"))
```

The `format()` method specifies the file type, while `option()` methods configure how the data is parsed. The `header` option tells Spark to use the first row as column names, and `inferSchema` automatically detects data types.

Read from different file formats

Azure Databricks supports multiple file formats. You can adjust the format and options based on your data source:

```
# Read JSON files
df_json = spark.read.format("json").load("/path/to/data.json")

# Read Parquet files (columnar format, efficient for analytics)
df_parquet = spark.read.format("parquet").load("/path/to/data.parquet")

# Read XML files (requires rowTag option)
df_xml = (spark.read
```

```
.format("xml")
.option("rowTag", "record")
.load("/path/to/data.xml"))
```

Note

The `rowTag` option specifies which XML element represents a row. For example, given this XML structure:

```
<data>
  <record><id>1</id><name>Alice</name></record>
  <record><id>2</id><name>Bob</name></record>
</data>
```

Setting `rowTag` to `"record"` tells Spark to treat each `<record>` element as a single row in the DataFrame.

Parquet files store data in a columnar format, which improves query performance for analytical workloads. JSON files support nested structures and are common for API responses. XML files require the `rowTag` option to specify which element maps to each row.

Connect to external databases

For data stored in relational databases, use JDBC connections to read directly from the source:

```
df_jdbc = (spark.read
  .format("jdbc")
  .option("url", "jdbc:sqlserver://server.database.windows.net:1433;database=mydb")
  .option("dbtable", "schema.tablename")
  .option("user", dbutils.secrets.get(scope="jdbc", key="username"))
  .option("password", dbutils.secrets.get(scope="jdbc", key="password"))
  .load())
```

Store credentials in Databricks secrets rather than hardcoding them in notebooks. You can also push queries down to the database by using a subquery in the `dbtable` option:

```
pushdown_query = "(SELECT id, name, amount FROM orders WHERE order_date > '2024-01-01'"
df = spark.read.format("jdbc").option("dbtable", pushdown_query).load()
```

Write data to Unity Catalog tables

After loading and transforming your data, write it to a Unity Catalog table. Azure Databricks uses Delta Lake format by default, which provides ACID transactions and versioning:

```
df.write.mode("overwrite").saveAsTable("catalog.schema.table_name")
```

The `mode()` option controls how existing data is handled:

- **overwrite**: Replaces the table contents entirely
- **append**: Adds rows to the existing table
- **ignore**: Does nothing if the table already exists
- **error**: Throws an exception if the table exists (default)

Ingest streaming data with Structured Streaming

Structured Streaming enables you to process data continuously as it arrives. Instead of `spark.read`, you use `spark.readStream` to create a streaming DataFrame that processes new data incrementally.

The following example reads streaming data from Kafka:

```
df_stream = (spark.readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "broker:9092")
    .option("subscribe", "topic-name")
    .option("startingOffsets", "latest")
    .load())
```

The `startingOffsets` option determines where to begin reading when starting a new query. Use `latest` for new data only or `earliest` to read from the beginning of the topic. Once you configure a checkpoint location in the write stream, subsequent runs automatically resume from the last processed offset, ignoring `startingOffsets`.

Process file-based streams

You can also stream from file sources as new files arrive in a directory:

```
df_file_stream = (spark.readStream
    .format("csv")
    .option("header", "true")
    .schema(defined_schema)
    .load("/Volumes/catalog/schema/volume/incoming/"))
```

For streaming file sources, you should define the schema explicitly rather than using `inferSchema`. Schema inference isn't supported for streaming sources because it requires reading the entire dataset.

Write streaming data to tables

Use `writeStream` to persist streaming data to Unity Catalog tables:

```
(df_stream
    .writeStream
    .format("delta"))
```

```
.outputMode("append")  
.option("checkpointLocation", "/checkpoints/my-stream")  
.toTable("catalog.schema.streaming_table"))
```

The output mode determines which records are written:

- **append**: Only new rows are written (default for most operations)
- **update**: Changed rows are written
- **complete**: All rows are rewritten (used with aggregations)

The checkpoint location stores progress information, allowing the stream to resume from where it stopped if interrupted.

Handle semi-structured data

When working with semi-structured data like JSON, the VARIANT data type provides flexibility for storing and querying nested structures:

```
from pyspark.sql.functions import parse_json, variant_get, col  
  
df = spark.read.text("/path/to/data.json")  
df_variant = df.select(parse_json(col("value")).alias("data"))
```

You can then query specific fields using the `variant_get` function or SQL syntax:

```
df_variant.select(variant_get(col("data"), "$.customer.name", "string")).display()
```

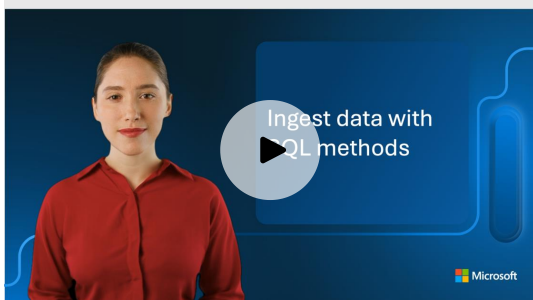
This approach works well when the JSON schema varies between records or evolves over time.

4. Ingest data with SQL methods

<https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/4-ingest-data-sql-methods>

Ingest data with SQL methods

Completed



SQL commands provide a declarative approach to data ingestion in Azure Databricks. If you're already comfortable with SQL syntax, these methods let you create and populate tables without writing procedural code. The three primary SQL ingestion techniques— `CREATE TABLE AS SELECT` (CTAS), `CREATE OR REPLACE TABLE`, and `COPY INTO`—each address different ingestion scenarios while maintaining full compatibility with Unity Catalog.

Create tables from queries with CTAS

The `CREATE TABLE AS SELECT` statement combines table creation and data population into a single operation. You define a new table based on the results of a `SELECT` query, making it ideal for transforming data during ingestion.

Consider a scenario where you need to ingest customer data from an external source and apply transformations. With CTAS, you write a query that reads from the source, applies your transformations, and saves the results to a new managed table:

```
CREATE TABLE catalog.schema.customers AS
SELECT
    customer_id,
    UPPER(customer_name) AS customer_name,
    email,
    created_date
FROM external_staging.raw_customers
WHERE customer_status = 'active';
```

The table schema is automatically derived from the query results. Azure Databricks creates the table using Delta format by default, which provides ACID transactions, time travel, and optimized performance.

CTAS works well for initial data loads and one-time migrations. When you need to read data from files, combine CTAS with the `read_files` table-valued function:

```
CREATE TABLE catalog.schema.sales_data AS
SELECT * FROM read_files(
    '/Volumes/catalog/schema/volume/sales/*.parquet',
    format => 'parquet'
);
```

Note

CTAS creates a new table each time it runs. If the table already exists, the command fails unless you use the `IF NOT EXISTS` clause—but that clause skips execution entirely rather than updating the table.

Refresh tables with `CREATE OR REPLACE TABLE`

When you need to fully refresh a table's contents, `CREATE OR REPLACE TABLE` provides a clean solution. This command either creates a new table or completely replaces an existing one, preserving table history, granted privileges, and any row filters or column masks you've configured.

This approach proves valuable for periodic data refreshes where you want to replace all existing data:

```
CREATE OR REPLACE TABLE catalog.schema.daily_metrics AS
SELECT
    report_date,
    SUM(revenue) AS total_revenue,
    COUNT(DISTINCT customer_id) AS unique_customers
FROM catalog.schema.transactions
WHERE report_date >= CURRENT_DATE - INTERVAL 30 DAYS
GROUP BY report_date;
```

Unlike dropping and recreating a table, `CREATE OR REPLACE` maintains the table's metadata and permissions. Downstream users and applications continue to access the table without reconfiguration.

You can also use this command to load data directly from files. The schema is automatically inferred from the query results:

```
CREATE OR REPLACE TABLE catalog.schema.products AS
SELECT * FROM read_files(
    '/Volumes/catalog/schema/volume/products.csv',
    format => 'csv',
    header => true
);
```

Important

`CREATE OR REPLACE` performs a full table replacement. For incremental updates where you only want to add new records, use `COPY INTO` instead.

Load files incrementally with `COPY INTO`

`COPY INTO` addresses a common ingestion challenge: loading files from cloud storage in a reliable, repeatable manner. Unlike CTAS, which runs once and creates a table, `COPY INTO` is designed for ongoing ingestion workflows where new files arrive regularly.

The command reads files from a specified location and appends them to an existing Delta table. Its key feature is idempotency—files that have already been loaded are automatically skipped, even across multiple executions:

```
COPY INTO catalog.schema.events
FROM '/Volumes/catalog/schema/volume/events/'
FILEFORMAT = JSON
FORMAT_OPTIONS ('multiline' = 'true');
```

Before running `COPY INTO`, the target table must already exist. Create it with the appropriate schema:

```
CREATE TABLE IF NOT EXISTS catalog.schema.events (
  event_id STRING,
  event_type STRING,
  event_timestamp TIMESTAMP,
  payload STRING
);
```

Configure file selection

When your source directory contains files with different naming patterns or you need to load specific files, use the `PATTERN` or `FILES` options:

```
-- Load only files matching a pattern
COPY INTO catalog.schema.orders
FROM '/Volumes/catalog/schema/volume/orders/'
FILEFORMAT = PARQUET
PATTERN = 'orders_2024*.parquet';

-- Load specific files by name
COPY INTO catalog.schema.orders
FROM '/Volumes/catalog/schema/volume/orders/'
FILEFORMAT = PARQUET
FILES = ('orders_001.parquet', 'orders_002.parquet');
```

Handle schema and data quality

`COPY INTO` provides options for handling schema changes and validating data before loading:

```
COPY INTO catalog.schema.sensor_data
FROM '/Volumes/catalog/schema/volume/sensors/'
FILEFORMAT = CSV
FORMAT_OPTIONS (
  'header' = 'true',
  'inferSchema' = 'true'
)
COPY_OPTIONS ('mergeSchema' = 'true');
```

The `mergeSchema` option allows the table schema to evolve as new columns appear in source files. For validating data without loading it, add the `VALIDATE` clause:

```
COPY INTO catalog.schema.sensor_data
FROM '/Volumes/catalog/schema/volume/sensors/'
FILEFORMAT = CSV
VALIDATE ALL;
```

This validation checks whether data can be parsed, matches the table schema, and satisfies nullability and check constraints.

Choose the right method

Each SQL ingestion method serves distinct purposes in your data engineering workflows:

Method	Best for	Behavior
CTAS	Initial data loads, one-time migrations, creating tables from queries	Creates a new table; fails if table exists
CREATE OR REPLACE	Periodic full refreshes, replacing staging tables	Replaces entire table; preserves permissions
COPY INTO	Ongoing file ingestion, incremental loads	Appends to existing table; skips loaded files

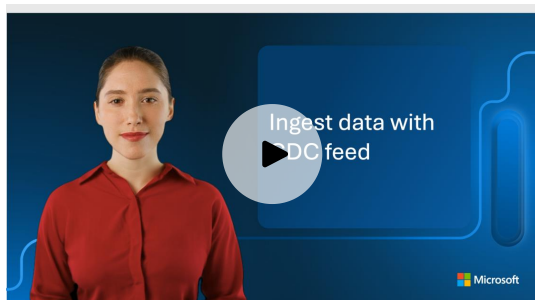
For file-based ingestion that requires automatic schema inference, file notifications, or exactly-once guarantees, consider using Auto Loader as a complementary approach. When your ingestion needs are straightforward and you prefer declarative SQL over procedural code, these three methods provide a complete toolkit for managing data flow into Unity Catalog.

5. Ingest data with CDC feed

<https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/5-ingest-data-change-data-capture-feed>

Ingest data with CDC feed

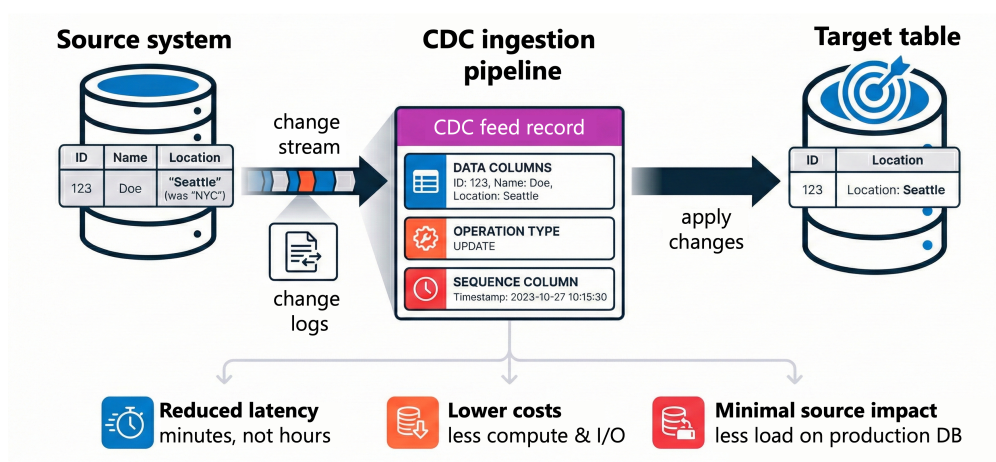
Completed



When source systems generate millions of transactions daily, reloading entire tables becomes impractical. You need a way to capture only the changes—inserts, updates, and deletes—and apply them efficiently to your destination tables. This pattern, known as change data capture (CDC), forms the foundation of modern incremental data pipelines.

Understand CDC ingestion patterns

Change data capture ingestion involves reading a stream of change records from a source system and applying those changes to a target table. Unlike full table reloads, CDC processes only what has changed since the last sync, reducing both processing time and resource consumption.



A typical CDC feed contains records with the following structure:

- **Data columns:** The actual values for each field in the source table
- **Operation type:** Indicates whether the record is an INSERT, UPDATE, or DELETE
- **Sequence column:** A timestamp or sequence number that determines the order of changes

Consider an employee table that tracks name and location. When you update an employee's city, the CDC feed doesn't send the entire table—it sends a single record with the new city value, the operation type UPDATE, and a sequence number. Your ingestion pipeline reads this record and applies the change to the corresponding row in the destination table.

This approach offers several advantages for data engineers:

- **Reduced latency:** Changes flow to the destination within minutes rather than hours
- **Lower costs:** Processing fewer records means less compute time and storage I/O

- **Minimal source impact:** Reading change logs puts less load on production databases than full table scans

Process CDC with the AUTO CDC API

Azure Databricks provides the AUTO CDC API in Lakeflow Spark Declarative Pipelines to simplify change data processing. This API handles the complexity of out-of-order records, deduplication, and change application automatically.

To implement CDC processing, you first create a streaming table as the target. Then you define a flow that reads change records from your source and applies them using the appropriate slowly changing dimension (SCD) type.

SCD Type 1 keeps only the current version of each record:

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

@dp.view
def employees_cdf():
    return spark.readStream.table("bronze.employee_changes")

dp.create_streaming_table("employees")

dp.create_auto_cdc_flow(
    target="employees",
    source="employees_cdf",
    keys=["employee_id"],
    sequence_by=col("sequence_num"),
    apply_as_deletes=expr("operation = 'DELETE'"),
    except_column_list=["operation", "sequence_num"],
    stored_as_scd_type=1
)
```

With SCD Type 1, when an employee moves from Seattle to Portland, the destination table updates the city column directly. The Seattle record disappears—only Portland remains.

SCD Type 2 preserves the complete history of changes:

```
dp.create_streaming_table("employees_history")

dp.create_auto_cdc_flow(
    target="employees_history",
    source="employees_cdf",
    keys=["employee_id"],
    sequence_by=col("sequence_num"),
    apply_as_deletes=expr("operation = 'DELETE'"),
    except_column_list=["operation", "sequence_num"],
    stored_as_scd_type=2
)
```

SCD Type 2 adds `__START_AT` and `__END_AT` columns to track when each version was active. The Seattle record remains with an end date, while a new Portland record appears with no end date. Analysts can query the table to see exactly when and how employee data changed over time.

Configure CDC flows in SQL

The same CDC patterns work in SQL using declarative syntax. SQL flows integrate naturally with medallion architecture pipelines where you define bronze, silver, and gold layers.

```
CREATE OR REFRESH STREAMING TABLE employees;

CREATE FLOW employees_cdc_flow
AS AUTO CDC INTO employees
FROM stream(bronze.employee_changes)
KEYS (employee_id)
APPLY AS DELETE WHEN operation = 'DELETE'
SEQUENCE BY sequence_num
COLUMNS * EXCEPT (operation, sequence_num)
STORED AS SCD TYPE 1;
```

The SQL syntax mirrors the Python API. You specify the target table, source stream, key columns, delete condition, sequencing column, and SCD type. Columns you don't need in the destination—like the operation and sequence columns—get excluded with the `EXCEPT` clause.

For historical tracking with SCD Type 2, you can also limit which columns trigger new history records:

```
CREATE FLOW employees_history_flow
AS AUTO CDC INTO employees_history
FROM stream(bronze.employee_changes)
KEYS (employee_id)
SEQUENCE BY sequence_num
COLUMNS * EXCEPT (operation, sequence_num)
STORED AS SCD TYPE 2
TRACK HISTORY ON * EXCEPT (last_login);
```

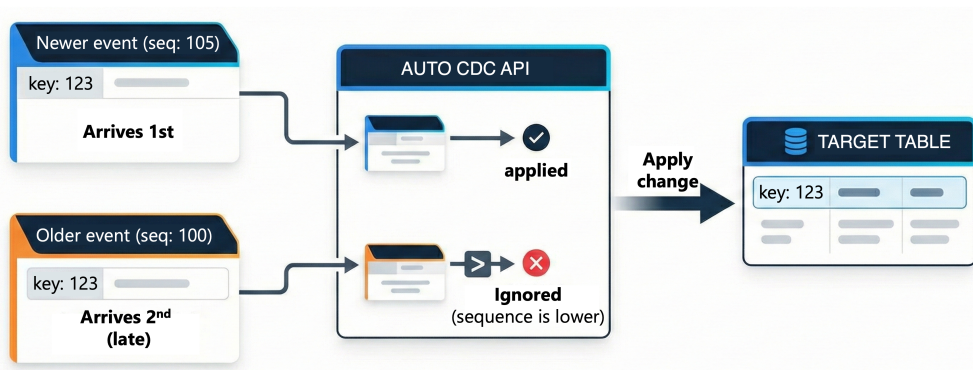
With this configuration, changes to the `last_login` column update the existing record in place. Only changes to other columns—like name or department—create new historical versions.

Handle out-of-order events

Distributed systems often deliver events out of order. A network delay might cause an older update to arrive after a newer one. The AUTO CDC API handles this scenario automatically using the sequence column you specify.

When two updates arrive for the same key, the API compares their sequence values. If the newer update (higher sequence number) has already been applied and an older update arrives late, the API

ignores the late-arriving record. This ensures your destination table always reflects the correct final state.



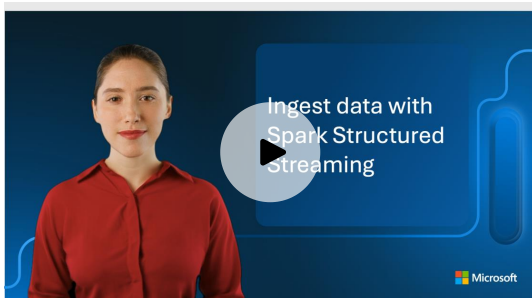
Without this built-in handling, you would need to write complex merge logic that checks timestamps, compares versions, and conditionally applies updates. The AUTO CDC API encapsulates all this logic, letting you focus on your business requirements rather than edge case handling.

6. Ingest data with Spark Structured Streaming

<https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/6-ingest-data-spark-structured-streaming>

Ingest data with Spark Structured Streaming

Completed



Real-time data ingestion allows you to process events as they occur rather than waiting for batch intervals. When your organization needs to capture IoT sensor readings, clickstream data, or financial transactions as they happen, Spark Structured Streaming provides the foundation for continuous data ingestion into your lakehouse.

In this unit, you learn how to configure Structured Streaming jobs that read from streaming sources and write to Unity Catalog tables.

Understand the Structured Streaming model

Structured Streaming treats incoming data as an unbounded table that grows continuously as new data arrives. You use the same DataFrame API you already know, but instead of `spark.read`, you use `spark.readStream` to create a streaming DataFrame that processes data incrementally.

Consider a scenario where sensor data arrives continuously from manufacturing equipment. Rather than scheduling hourly batch jobs to ingest this data, you can configure a streaming job that processes each reading within seconds of its arrival. The streaming engine tracks which records have been processed using **checkpoints**, so if the job restarts, it resumes from where it left off without reprocessing or missing data.

The fundamental pattern for streaming ingestion involves three steps:

1. **Configure the source** using `spark.readStream` with connection details
2. **Apply transformations** to shape the data as needed
3. **Write to a sink** using `writeStream` with a checkpoint location

Connect to Kafka and Event Hubs

Azure Event Hubs provides an Apache Kafka-compatible endpoint, which means you can use the same Kafka connector for both services. This unified approach simplifies your code when working with different message buses.

The following example reads messages from a Kafka topic:

```
df_stream = (spark.readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "broker-server:9092")
    .option("subscribe", "sensor-readings")
    .option("startingOffsets", "latest")
    .load())
```

For Azure Event Hubs, you connect through its Kafka-compatible interface. The key difference is the authentication configuration, which uses SASL with your Event Hubs connection string:

```
eh_namespace = "your-namespace"
eh_name = "your-eventhub"
connection_string = dbutils.secrets.get(scope="eventhubs", key="connection-string")

kafka_options = {
    "kafka.bootstrap.servers": f"{eh_namespace}.servicebus.windows.net:9093",
    "subscribe": eh_name,
    "kafka.sasl.mechanism": "PLAIN",
    "kafka.security.protocol": "SASL_SSL",
    "kafka.sasl.jaas.config": f'kafkashaded.org.apache.kafka.common.security.plain
}

df_stream = (spark.readStream
```

```
.format("kafka")
.options(**kafka_options)
.load()
```

When reading from Kafka or Event Hubs through the Kafka-compatible interface, you receive a standard schema with `key`, `value`, `topic`, `partition`, `offset`, and `timestamp` columns. The `key` and `value` columns contain binary data, so you typically cast them to string and parse the content:

```
from pyspark.sql.functions import col, from_json
from pyspark.sql.types import StructType, StringType, DoubleType

schema = StructType() \
    .add("device_id", StringType()) \
    .add("temperature", DoubleType()) \
    .add("timestamp", StringType())

parsed_df = (df_stream
    .select(col("value").cast("string").alias("json_value"))
    .select(from_json(col("json_value"), schema).alias("data"))
    .select("data.*"))
```

Use the native Event Hubs connector

Azure Databricks also provides a native Event Hubs connector that simplifies authentication and provides direct access to Event Hubs-specific features. This connector uses the `eventhubs` format:

```
connection_string = dbutils.secrets.get(scope="eventhubs", key="connection-string")

events = (spark.readStream
    .format("eventhubs")
    .option("eventhubs.connectionString", connection_string)
    .load())
```

The native Event Hubs connector returns a different schema than the Kafka-compatible interface. The event payload is stored in the `body` column (instead of `value`), along with additional Event Hubs-specific metadata columns:

```
# Extract and parse the event payload
payload = events.selectExpr("CAST(body AS STRING) AS payload")
```

You can configure additional Event Hubs options such as consumer group and starting position:

```
events = (spark.readStream
    .format("eventhubs")
    .option("eventhubs.connectionString", connection_string)
    .option("eventhubs.consumerGroup", "$Default")
    .option("eventhubs.startingPosition", '{"offset": "-1", "seqNo": -1, "enqueued'
    .load())
```

The `startingPosition` option controls where to begin reading events. Use `-1` for the latest events or configure a specific offset, sequence number, or enqueued time for your ingestion needs.

Configure checkpoint locations

Checkpoints are the mechanism that makes Structured Streaming fault-tolerant. The checkpoint directory stores the offset information that tracks exactly which records have been processed. If your streaming job stops unexpectedly, it can restart and continue from the last committed checkpoint.

Important

Each streaming query must have its own unique checkpoint location. Sharing checkpoints between different queries causes data processing errors.

When you start a streaming write, specify the checkpoint location:

```
checkpoint_path = "/checkpoints/sensor-ingestion"

query = (parsed_df.writeStream
    .format("delta")
    .option("checkpointLocation", checkpoint_path)
    .toTable("catalog.schema.sensor_readings"))
```

Store your checkpoints in a reliable, durable location like Azure Data Lake Storage. The checkpoint files are small but critical for recovery, so configure your storage location with appropriate backup and retention policies.

Write streaming data to Unity Catalog tables

Writing to Unity Catalog tables uses Delta Lake format, which supports both streaming writes and ACID transactions. The `writeStream` API mirrors the batch `write` API but includes options for continuous processing:

```
(parsed_df.writeStream
    .format("delta")
    .outputMode("append")
    .option("checkpointLocation", "/checkpoints/my-stream")
    .toTable("bronze.iot.sensor_readings"))
```

The **output mode** determines how results are written:

- **append**: New rows are added to the table. Use this mode for most ingestion workloads.
- **update**: Changed rows are updated. Use with aggregations that produce updates.
- **complete**: All rows are rewritten. Use with aggregations that require complete recalculation.

For ingestion scenarios where you're appending raw data, the default `append` mode is appropriate.

Control processing with triggers

Triggers determine when and how frequently the streaming engine processes data. Azure Databricks supports several trigger options:

Trigger	Behavior
Default (no trigger)	Processes the next batch as soon as the previous batch completes
<code>processingTime="10 seconds"</code>	Processes in fixed intervals
<code>availableNow=True</code>	Processes all available data, then stops
<code>once=True</code>	Processes one micro-batch, then stops (deprecated)

For continuous ingestion that runs indefinitely, omit the trigger option or use a processing time interval. For incremental batch patterns where you want to process all available data on a schedule, use `availableNow`:

```
(parsed_df.writeStream
  .format("delta")
  .option("checkpointLocation", checkpoint_path)
  .trigger(availableNow=True)
  .toTable("catalog.schema.table_name"))
```

The `availableNow` trigger processes all unprocessed data since the last checkpoint, then terminates. You can schedule this pattern using Databricks Jobs to run at regular intervals, combining the efficiency of streaming (only processing new data) with the predictability of scheduled batch jobs.

Monitor streaming queries

Active streaming queries appear in the Spark UI under the **Structured Streaming** tab. You can also access query status programmatically:

```
query = parsed_df.writeStream.format("delta").option("checkpointLocation", checkpoint_path).start()

# Check if the query is still running
print(query.isActive)

# Get the current status
print(query.status)

# View recent progress
print(query.recentProgress)
```

Use the `awaitTermination()` method to block execution until the stream completes or fails. For production workloads, implement error handling to capture and respond to streaming failures:

```
try:
    query.awaitTermination()
except Exception as e:
    print(f"Stream failed: {e}")
```

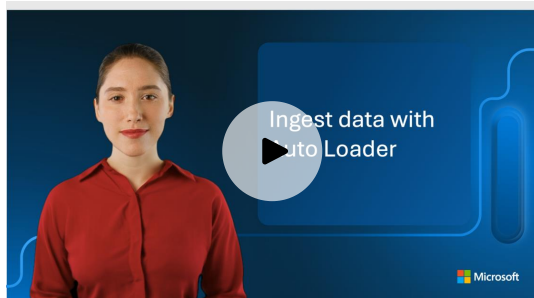
In production environments, configure monitoring and alerts to detect when streaming jobs fall behind or encounter errors. Streaming metrics include lag information that helps you understand whether your processing keeps pace with incoming data.

7. Ingest data with Auto Loader

<https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/7-ingest-data-auto-loader>

Ingest data with Auto Loader

Completed

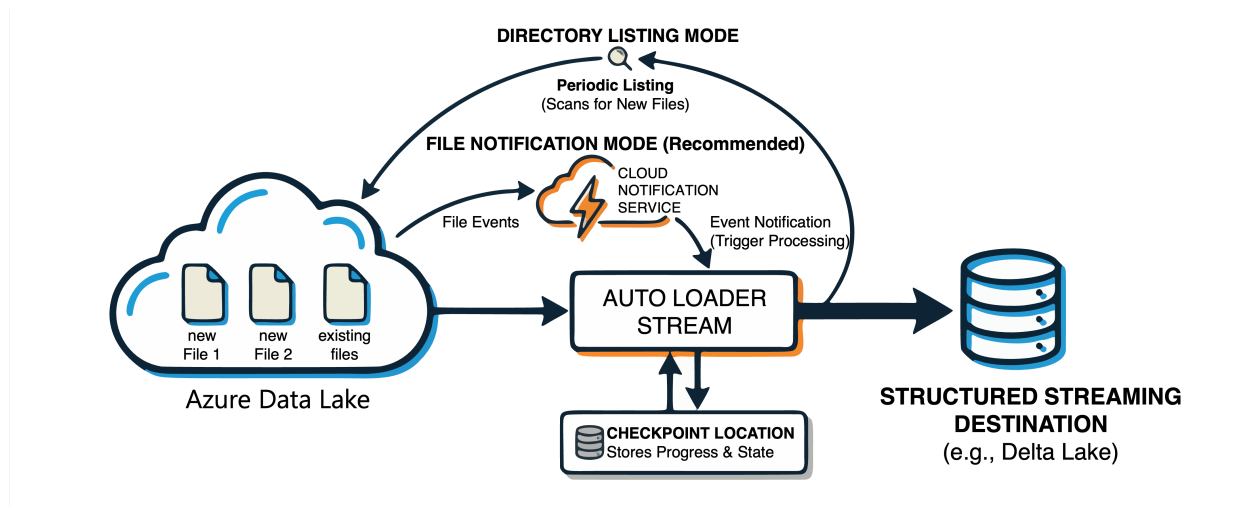


When new data files arrive continuously in cloud storage, you need an efficient way to process them without manually tracking which files have been ingested. **Auto Loader** solves this challenge by automatically detecting and ingesting new files as they appear, providing exactly-once guarantees without requiring you to manage state or checkpoints manually.

Understand how Auto Loader works

Auto Loader is a Structured Streaming source that monitors a cloud storage location and incrementally processes new files. Rather than scanning entire directories repeatedly, Auto Loader maintains state about which files it has already processed, ensuring each file is ingested exactly once.

When you start an Auto Loader stream, it can process existing files in the directory and then continuously watch for new arrivals. The stream stores progress information in a checkpoint location, which allows it to resume from exactly where it stopped if interrupted.



Auto Loader detects new files using one of two modes:

- **Directory listing mode:** Auto Loader periodically lists the input directory to discover new files. This approach requires no additional configuration beyond storage access.
- **File notification mode:** Auto Loader uses cloud notification services to receive events when new files arrive. This mode is more efficient for large-scale workloads because it avoids repeated directory listings.

For most production workloads, Databricks recommends file notification mode with file events enabled on your external location in Unity Catalog. With file events, Auto Loader receives notifications directly when files arrive, which reduces latency and cloud API costs.

Configure Auto Loader for file ingestion

To ingest data with Auto Loader, you use the `cloudFiles` format with `spark.readStream`. The following example reads JSON files from Azure Data Lake Storage and writes them to a Unity Catalog table:

```
base_path = "abfss://container@storage.dfs.core.windows.net/autoloader/orders"
schema_path = f"{base_path}/schema"
checkpoint_path = f"{base_path}/checkpoint"

(spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "json")
  .option("cloudFiles.schemaLocation", schema_path)
  .load("abfss://container@storage.dfs.core.windows.net/incoming/orders/")
  .writeStream
  .option("checkpointLocation", checkpoint_path)
  .trigger(availableNow=True)
  .toTable("sales.bronze.orders"))
```

The `cloudFiles.format` option specifies the source file format. Auto Loader supports JSON, CSV, Parquet, Avro, ORC, XML, text, and binary files. The `cloudFiles.schemaLocation` stores the

inferred schema, which enables Auto Loader to track schema changes over time. While the schema location and checkpoint location can be the same, keeping them separate allows you to reset a stream without losing the inferred schema.

The `trigger(availableNow=True)` setting processes all currently available files and then stops. This pattern works well for scheduled batch jobs. For continuous processing, you can omit the trigger or use `trigger(processingTime="1 minute")` to process files in intervals.

Use SQL syntax with `read_files`

Auto Loader also supports SQL syntax through the `read_files` table-valued function. This approach is useful when you prefer SQL or when working in SQL notebooks:

```
SELECT * FROM read_files(  
  'abfss://container@storage.dfs.core.windows.net/incoming/orders/',  
  format => 'json',  
  schemaHints => 'order_id INT, amount DECIMAL(10,2)'  
)
```

When using `read_files` with streaming tables in Lakeflow Spark Declarative Pipelines, Auto Loader capabilities are automatically enabled. Use the `STREAM` keyword to enable incremental processing:

```
CREATE OR REFRESH STREAMING TABLE bronze_orders  
AS SELECT * FROM STREAM read_files(  
  'abfss://container@storage.dfs.core.windows.net/incoming/orders/',  
  format => 'json'  
)
```

In this context, the pipeline manages checkpointing and schema evolution automatically.

Tip

When working with Unity Catalog, store your checkpoint and schema locations in a managed storage location. This ensures consistent governance and avoids permission issues with nested paths.

Handle schema inference and evolution

Auto Loader can automatically detect the schema of your source files, eliminating the need to define schemas manually. When schema inference is enabled, Auto Loader samples files to determine column names and data types.

For JSON and CSV files, Auto Loader infers all columns as strings by default. This conservative approach prevents type mismatches that could cause data loss. To enable automatic type detection, set the `cloudFiles.inferColumnTypes` option:

```
(spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "json")
  .option("cloudFiles.inferColumnTypes", "true")
  .option("cloudFiles.schemaLocation", checkpoint_path)
  .load(source_path))
```

When your source data introduces new columns, Auto Loader detects the change and can evolve the target schema automatically. The `cloudFiles.schemaEvolutionMode` option controls this behavior:

Mode	Behavior
<code>addNewColumns</code>	Stream fails. New columns are added to the schema. On restart, stream continues with the updated schema. This is the default when no schema is provided.
<code>rescue</code>	Schema is never evolved and stream does not fail. All new columns are captured in the <code>_rescued_data</code> column.
<code>failOnNewColumns</code>	Stream fails. Stream does not restart unless the provided schema is updated or the offending data file is removed.
<code>none</code>	Schema is not evolved, new columns are ignored, and data is not rescued unless the <code>rescuedDataColumn</code> option is set. This is the default when a schema is provided.

The **rescued data column** (`_rescued_data`) captures any data that doesn't match the expected schema, including type mismatches and unexpected columns. This feature prevents data loss when source data doesn't conform to expectations. Auto Loader automatically adds this column when inferring the schema.

```
(spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "csv")
  .option("cloudFiles.schemaEvolutionMode", "rescue")
  .schema(expected_schema)
  .load(source_path))
```

To rename the rescued data column, use the `rescuedDataColumn` option:

```
.option("rescuedDataColumn", "my_rescued_data")
```

If you know certain columns should have specific types, use **schema hints** to override the inferred types:

```
.option("cloudFiles.schemaHints", "order_date DATE, amount DECIMAL(10,2)")
```

Monitor and optimize Auto Loader streams

For production workloads, monitoring helps you track ingestion progress and identify issues. Auto Loader provides a SQL function to query the state of discovered files:

```
SELECT * FROM cloud_files_state('/path/to/checkpoint');
```

This query returns metadata about files that Auto Loader has discovered, including their processing status. You can use this information to verify that files are being ingested as expected.

Auto Loader reports metrics through the Structured Streaming progress dashboard, including:

- `numFilesOutstanding` : Files discovered but not yet processed
- `numBytesOutstanding` : Total bytes waiting to be processed

For high-volume workloads, control the processing rate using `cloudFiles.maxFilesPerTrigger` or `cloudFiles.maxBytesPerTrigger`. These settings prevent any single micro-batch from becoming too large:

```
.option("cloudFiles.maxFilesPerTrigger", "1000")  
.option("cloudFiles.maxBytesPerTrigger", "1g")
```

Important

The checkpoint location stores critical state information. Avoid applying cloud storage lifecycle policies that might delete checkpoint files, as this corrupts the stream state and requires restarting from scratch.

To access file-level metadata in your ingested data, select the `_metadata` column. This hidden column contains information about each source file, including `file_path`, `file_name`, `file_size`, and `file_modification_time`:

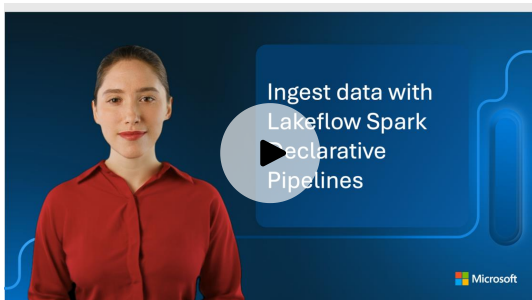
```
(spark.readStream  
  .format("cloudFiles")  
  .option("cloudFiles.format", "parquet")  
  .load(source_path)  
  .select("*", "_metadata.file_path", "_metadata.file_modification_time"))
```

Auto Loader integrates seamlessly with Lakeflow Spark Declarative Pipelines, where it handles checkpointing and schema management automatically. In the next unit, you learn how to use Auto Loader within declarative pipeline definitions for end-to-end data ingestion workflows.

8. Ingest data with Lakeflow Spark Declarative Pipelines

Ingest data with Lakeflow Spark Declarative Pipelines

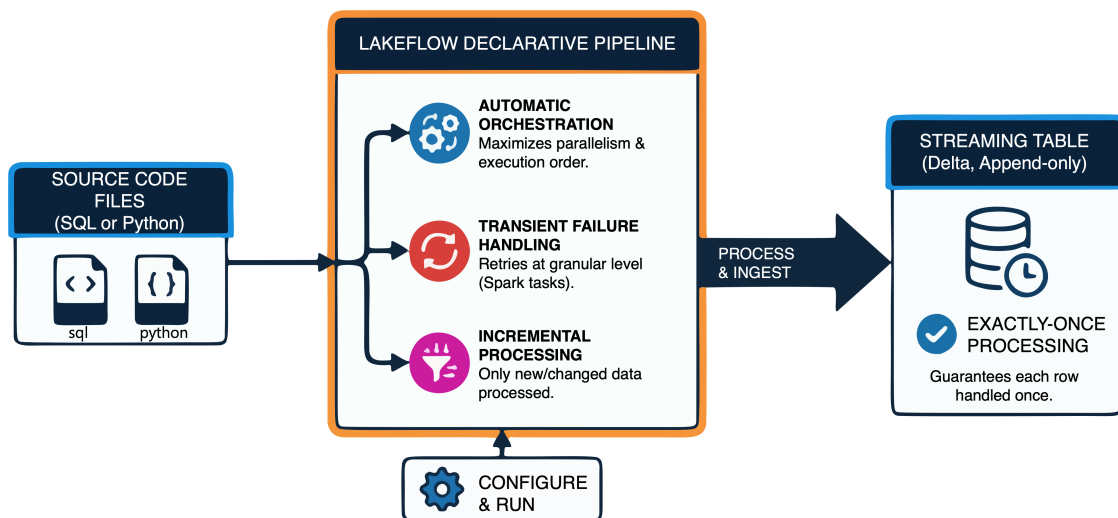
Completed



When you need to ingest data from multiple sources with automated orchestration, schema management, and exactly-once processing guarantees, Lakeflow Spark Declarative Pipelines provides a declarative framework for the entire process. Rather than writing complex imperative code to manage streaming state and dependencies, you define what you want, and the pipeline handles how to execute it.

Understand Lakeflow Spark Declarative Pipelines for ingestion

Lakeflow Spark Declarative Pipelines is a framework for building batch and streaming data pipelines using SQL or Python. You declare the structure of your data transformations, and the pipeline automatically handles orchestration, retries, and incremental processing.



For data ingestion, you typically create **streaming tables** as targets. A streaming table is a Delta table with built-in support for streaming data. Each row from the source is processed exactly once, making streaming tables ideal for append-only ingestion workloads where data continuously arrives.

The pipeline framework provides several benefits for ingestion:

- **Automatic orchestration:** The pipeline determines the correct execution order and maximizes parallelism.
- **Transient failure handling:** If a task fails, the pipeline retries at the most granular level possible, starting with individual Spark tasks.
- **Exactly-once processing:** Streaming tables guarantee that each input row is handled only once.
- **Incremental processing:** Only new or changed data is processed during each update.

To use pipelines, you define streaming tables and flows in source code files (SQL or Python), then configure and run a pipeline that processes these definitions.

Create streaming tables for ingestion

A streaming table acts as the target for your ingestion flow. You define what data to read and how to transform it, and the pipeline manages the execution.

The following SQL example creates a streaming table that reads JSON files from cloud storage:

```
CREATE OR REFRESH STREAMING TABLE orders_bronze
AS SELECT *
FROM STREAM read_files(
  'abfss://container@storage.dfs.core.windows.net/incoming/orders/',
  format => 'json'
)
```

The `STREAM` keyword indicates that the source should be processed incrementally. The `read_files` function reads files from the specified path with automatic format detection. When used with `STREAM`, it enables Auto Loader capabilities for incremental file detection.

In Python, you use the `@dp.table` decorator to define a streaming table:

```
from pyspark import pipelines as dp

@dp.table
def orders_bronze():
    return (
        spark.readStream.format("cloudFiles")
            .option("cloudFiles.format", "json")
            .load("abfss://container@storage.dfs.core.windows.net/incoming/orders/")
    )
```

Both examples create a streaming table named `orders_bronze`. When the pipeline runs, it processes any new files that arrive in the source location. The pipeline manages checkpoints internally, so you don't need to configure checkpoint locations manually.

Tip

Store your pipeline source code in a Git folder for version control. The Lakeflow Pipelines Editor supports a default folder structure with `transformations` for source code and `explorations` for ad hoc analysis.

Configure flows for multiple sources

A **flow** defines how data moves from a source to a target. When you create a streaming table with a query, a default flow is created automatically. However, you can also define flows explicitly to handle more complex scenarios, such as ingesting from multiple sources into a single table.

Consider a scenario where you receive order data from different regional systems. You want to combine them into a single `orders_us` table without using a `UNION` clause. Explicit flows enable this pattern:

```
-- Create the target streaming table
CREATE OR REFRESH STREAMING TABLE orders_us;

-- Add a flow from the west region
CREATE FLOW orders_west_flow
AS INSERT INTO orders_us BY NAME
SELECT * FROM STREAM(orders_west);

-- Add a flow from the east region
CREATE FLOW orders_east_flow
AS INSERT INTO orders_us BY NAME
SELECT * FROM STREAM(orders_east);
```

Each flow appends data to the same target table independently. This approach is more flexible than `UNION` because you can add new source flows without modifying existing ones or triggering a full refresh.

In Python, you use the `@dp.append_flow` decorator:

```
from pyspark import pipelines as dp

# Create the target streaming table
dp.create_streaming_table("orders_us")

# Add flow from west region
@dp.append_flow(target="orders_us")
def orders_west_flow():
    return spark.readStream.table("orders_west")
```

```
# Add flow from east region
@dp.append_flow(target="orders_us")
def orders_east_flow():
    return spark.readStream.table("orders_east")
```

Important

Flow names are used to identify streaming checkpoints. If you rename a flow, the checkpoint doesn't carry over, and the renamed flow starts processing from the beginning as a new flow.

Ingest from message buses

Lakeflow Spark Declarative Pipelines supports ingestion from message buses like Apache Kafka, Azure Event Hubs, Amazon Kinesis, and Google Pub/Sub. For Azure environments, Event Hubs is commonly used and provides a Kafka-compatible endpoint.

The following example reads from a Kafka topic:

```
CREATE OR REFRESH STREAMING TABLE events_raw
AS SELECT *
FROM STREAM read_kafka(
    bootstrapServers => 'kafka_server:9092',
    subscribe => 'events_topic'
)
```

For Azure Event Hubs, you use the Kafka-compatible interface. Event Hubs requires authentication using a connection string stored securely in Azure Databricks secrets:

```
from pyspark import pipelines as dp

EH_NAMESPACE = spark.conf.get("eh.namespace")
EH_NAME = spark.conf.get("eh.name")
EH_CONN_STR = dbutils.secrets.get(scope="eh-secrets", key="connection-string")

KAFKA_OPTIONS = {
    "kafka.bootstrap.servers": f"{EH_NAMESPACE}.servicebus.windows.net:9093",
    "subscribe": EH_NAME,
    "kafka.sasl.mechanism": "PLAIN",
    "kafka.security.protocol": "SASL_SSL",
    "kafka.sasl.jaas.config": f'kafkashaded.org.apache.kafka.common.security.plain
}

@dp.table
def events_bronze():
    return (
        spark.readStream
            .format("kafka")
            .options(**KAFKA_OPTIONS)
```

```
)  
    .load()
```

Store sensitive connection information in Azure Databricks secrets rather than hardcoding values in your pipeline code. Use pipeline parameters (configured in settings) to pass non-sensitive configuration values like namespace names.

Handle schema inference and evolution

When ingesting semi-structured data like JSON, you often don't know the exact schema in advance. Lakeflow Spark Declarative Pipelines provides automatic schema inference and evolution through the `from_json` function.

```
CREATE STREAMING TABLE events_parsed AS  
SELECT  
    from_json(value, NULL,  
        map('schemaLocationKey', 'events_schema',  
            'schemaEvolutionMode', 'addNewColumns')) AS parsed  
FROM STREAM read_kafka(  
    bootstrapServers => 'kafka_server:9092',  
    subscribe => 'events_topic'  
)
```

Setting the schema to `NULL` enables automatic inference. The `schemaLocationKey` uniquely identifies where the schema is stored. When new fields appear in the source data, the pipeline detects them and can automatically add them to the schema.

The `schemaEvolutionMode` controls how the pipeline responds to schema changes:

Mode	Behavior
<code>addNewColumns</code>	New columns are added to the schema automatically. This is the default.
<code>rescue</code>	New columns are captured in a <code>_rescued_data</code> column instead of being added.
<code>failOnNewColumns</code>	The pipeline fails when new columns are detected.
<code>none</code>	New columns are ignored and not rescued.

For columns that don't match the inferred schema (type mismatches or unexpected fields), a `_rescued_data` column captures the data to prevent loss.

Note

If you need strict control over the schema, provide explicit schema hints instead of relying on automatic inference: `map('schemaHints', 'order_id INT, amount DECIMAL(10,2)')`

When your ingestion pipeline is running, you can monitor progress through the pipeline event log and the Lakeflow Pipelines Editor. The editor provides execution insights, data previews, and a directed acyclic graph (DAG) showing table dependencies. For production workloads, configure notifications in the pipeline settings to receive alerts when issues occur.

9. Exercise - Ingest Data into Unity Catalog

<https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/9-exercise-ingest-data-into-unity-catalog>

Exercise - Ingest Data into Unity Catalog

Completed

Now it's your chance to ingest data into Unity Catalog. In this lab, you practise the core data ingestion techniques available in Azure Databricks. You load CSV files from a Unity Catalog managed volume into Delta tables using PySpark DataFrames, `SQL COPY INTO`, and `CREATE TABLE AS SELECT`. You also configure Auto Loader to automatically detect and process new files from cloud storage, demonstrating exactly-once ingestion for continuously arriving data.

Note

To complete this lab, you need an [Azure subscription](#) in which you have administrative access.

Launch the exercise and follow the instructions.

[Launch Exercise](#)

10. Module assessment

<https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/10-knowledge-check>

Module assessment

Completed

11. Summary

<https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/11-summary>

Summary

Completed

Throughout this module, you explored the comprehensive set of data ingestion techniques available in Azure Databricks for loading data into **Unity Catalog**. From managed connectors to custom notebook code, from declarative SQL commands to real-time streaming, each approach addresses specific data patterns and organizational requirements.

You learned how **Lakeflow Connect** simplifies ingestion from enterprise sources by providing managed connectors with built-in change data capture and SCD support. You explored **notebook-based ingestion** using DataFrames and the Spark Structured Streaming API for full control over your ingestion logic. You discovered how **SQL commands** like `COPY INTO` and `CREATE TABLE AS SELECT` provide declarative options for file-based batch loads with automatic file tracking.

For incremental processing, you implemented **CDC flows** using the AUTO CDC API to efficiently apply inserts, updates, and deletes to destination tables. You configured **Spark Structured Streaming** to process events from Kafka and Event Hubs in real time with exactly-once guarantees. You set up **Auto Loader** to automatically detect and ingest new files with schema inference and evolution capabilities. Finally, you used **Lakeflow Spark Declarative Pipelines** to orchestrate end-to-end ingestion workflows with automatic orchestration and error handling.

As you build data pipelines for your organization, consider which ingestion method best matches each source system's characteristics. Use managed connectors when available for common enterprise sources. Choose notebooks for complex transformations or sources that require custom logic. Apply Auto Loader for file-based streaming with automatic schema handling. Orchestrate your ingestion flows with Lakeflow Spark Declarative Pipelines to benefit from built-in reliability features. With these techniques, you can build robust ingestion pipelines that deliver high-quality data to your lakehouse efficiently and reliably.