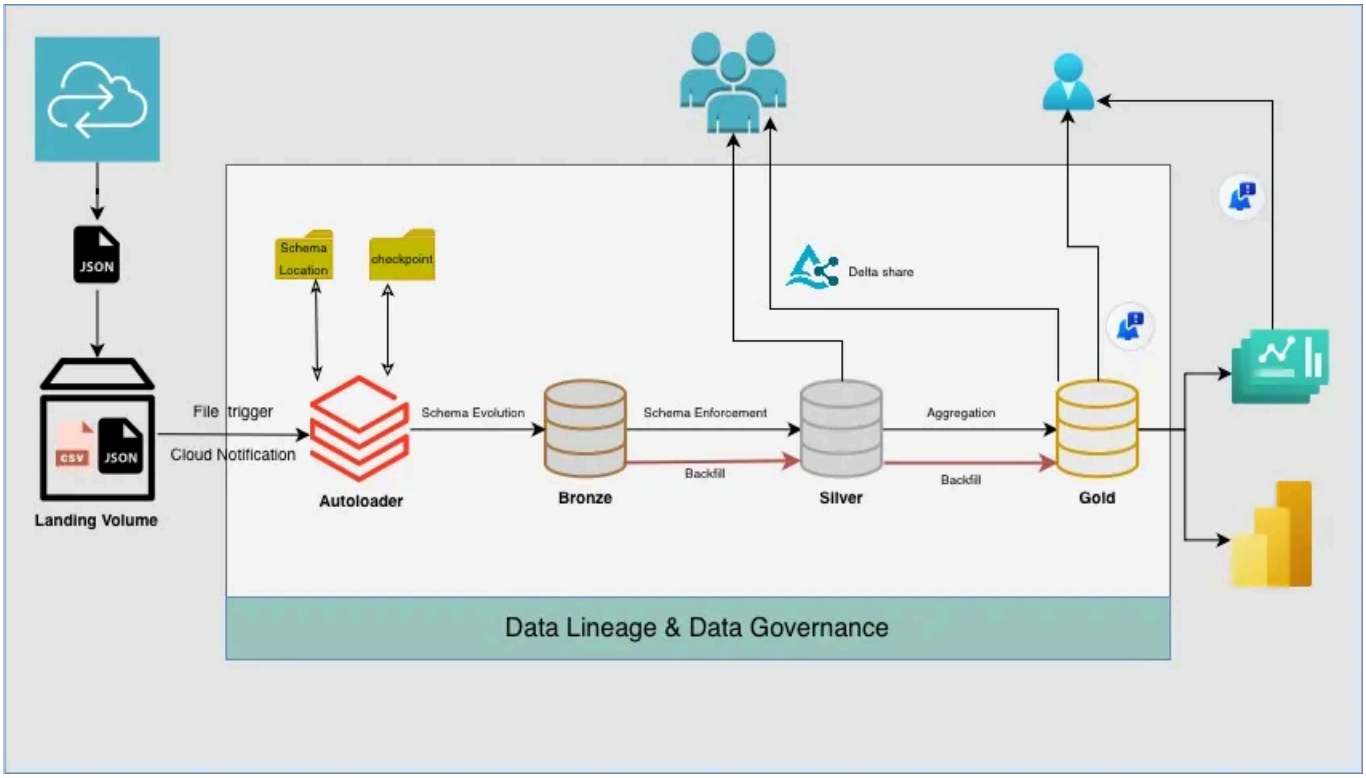
Databricks Auto Loader: Best Practices for Reliable and Scalable Data Ingestion

Asindu Gayangana Follow 15 min read · Dec 25, 2025



This article provides a comprehensive overview of **Databricks Auto Loader** and its role in building scalable, reliable, and production-ready data pipelines. It covers the core capabilities of Auto Loader, including **different trigger types**, **schema evolution strategies**, and **best practices for incremental/streaming data ingestion**, duplicate handling, and JSON parsing using real-world use cases.

The article also explains **watermarking for streaming pipelines**, **backfill concepts**, and how **parallelism works during backfills**. It demonstrates how to **parameterize notebooks and jobs** to support both automated incremental runs and historical backfills without code changes. In addition, it covers important operational aspects such as **retry policies**, **runtime thresholds**, **notifications**, and **orchestrating downstream actions** like refreshing **dbt models** and **dashboards** as part of the same pipeline.

Beyond Auto Loader, **Databricks is a powerful unified data platform**. Databricks integrates seamlessly with all three major cloud providers — **AWS, Azure, and GCP** — and is built for large-scale data processing using **Delta Lake and the Medallion (Bronze–Silver–Gold) architecture**. The platform enables reliable batch and streaming workloads, strong governance through Unity Catalog, built-in data lineage, and smooth integration with **machine learning and AI workflows**.

With features such as **serverless compute**, **job orchestration**, **native dashboarding**, and **AI-assisted development**, Databricks significantly simplifies data engineering workflows while maintaining enterprise-grade scalability, reliability, and performance. This article aims to provide both conceptual understanding and practical guidance for implementing robust end-to-end pipelines using Databricks Auto Loader and the Lakehouse architecture.

Databricks Job Trigger Types

Schedule Trigger

- Runs on a **specific time or date**
- Common for **daily/hourly batch jobs**

File Arrival Trigger (Auto Loader)

- Automatically runs when **new files arrive**
- **Same file is never processed again**

Table Update Trigger

- Watches a **Delta table**
- Job runs whenever the table is **updated**
- Useful for **downstream dependencies**

Continuous Trigger (Streaming)

- Used for **real-time streaming**
- Job runs continuously
- Suitable for Kafka, Event Hubs, etc.

Auto Loader — Schema Evolution Behavior

Databricks Auto Loader provides **idempotent file ingestion**, meaning the **same file is never processed more than once.**
Auto Loader tracks processed files using **checkpoint metadata** and stores inferred or evolving schemas in a **schema location.**

Because of this:

- Already processed files are not re-read
- Schema changes are tracked separately from processing state

Both schemaLocation and checkpointLocation are mandatory when using Auto Loader in streaming mode.

Supported Schema Evolution Scenarios

- Adding new columns
- Column renames (treated as new columns)
- Dropped columns (values appear as NULL for new records)

Not Supported

- Column data type changes
  (These cause failures unless handled explicitly using rescue mode or raw ingestion)

Schema Evolution Scenarios

Adding New Columns (Best for Structured Files)

This approach works **best for structured formats such as CSV, Parquet, and Avro**, where Spark can reliably parse records even when new columns appear.

Schema evolution must be enabled:

- At **read time** (Auto Loader)
- At **write time** (Delta Lake)

```
df = (
spark.readStream
.format("cloudFiles")
.option("cloudFiles.format", "csv")
.option("cloudFiles.schemaEvolutionMode", "addNewColumns")
.option("cloudFiles.schemaLocation", schema)
.load("/Volumes/dev/landing/source_data/")
)
df.writeStream \
.format("delta") \
.option("checkpointLocation", checkpoint) \
.option("mergeSchema", "true") \
.trigger(once=True) \
.table("dev.bronze.api_data")
```

addNewColumns does **not** protect against schema drift at JSON parsing time. It primarily applies during Delta writes and structured formats.

JSON Ingestion and Schema Drift

When reading JSON files using Auto Loader, ingestion **can fail** if the source schema evolves unexpectedly, such as:

- New or missing fields
- Nested structure changes
- Array or struct shape changes
- Data type changes

This happens because Spark must successfully **parse JSON before schema evolution can be applied.**

Rescue Mode for JSON (Recommended When Parsing in Bronze)

To prevent pipeline failures, Auto Loader provides **rescue mode**, which captures all fields that do not match the inferred schema into a single column (for example, _unprocessed_column).

This allows ingestion to continue without errors.

```
df = (
 spark.readStream
   .format("cloudFiles")
   .option("cloudFiles.format", "json")
   .option("cloudFiles.schemaEvolutionMode", "rescue")
   .option("cloudFiles.rescuedDataColumn", "_unprocessed_column")
   .option("cloudFiles.schemaLocation", schema)
   .load("/Volumes/dev/landing/test/")
)


df.writeStream \
 .format("delta") \
 .option("checkpointLocation", checkpoint) \
 .trigger(once=True) \
 .table("dev.bronze.coins_price_best_practice")
```

Bronze Layer Best Practice for JSON (Strong Recommendation)

According to Lakehouse best practices, the **Bronze layer should preserve source data exactly as received**, without enforcing schemas or applying transformations.

For JSON data, the **safest and most reliable approach** is:

- Ingest JSON as **raw text** in Bronze
- Parse, explode, and enforce schema in **Silver**

This guarantees:

- Zero ingestion failures due to schema drift
- Full traceability and reprocessing capability
- Clean separation of ingestion and transformation concerns

Schema Change Behavior Summary

Dropping Columns

- Dropped columns appear as NULL in new records
- No ingestion failure occurs

Renaming Columns

- Treated as a new column
- Old column retains historical values
- New column contains values only after the rename

Fail on New Columns

To explicitly **fail ingestion when schema changes** are detected:

```
.option("cloudFiles.schemaEvolutionMode", "failOnNewColumn")
```

This is useful in tightly controlled pipelines where schema drift must be blocked.

Schema Hints

Schema hints allow Auto Loader to **avoid schema inference** and enforce column data types upfront.

```
.option(
"cloudFiles.schemaHints",
"order_id INT, order_date DATE, customer_id INT, product_id INT, quantity INT, u
)
```

File Discovery Modes

Default Directory Listing

Best when:

- Many directories exist
- Deeply partitioned folder structures (for example: year=2024/month=01)

useNotifications (Recommended for Large File Volumes)

Faster
More cost-efficient
Lower file-listing overhead

Best when:

- Few directories
- Many incoming files

Uses cloud-native event systems such as:

- AWS SNS / SQS
- Azure Event Grid
- GCP Pub/Sub

Auto Loader Performance Options

Control ingestion rate to protect downstream systems.

Maximum files per trigger:

```
.option("maxFilesPerTrigger", 100)
```

Maximum bytes per trigger:

```
.option("maxBytesPerTrigger", "1g")
```

JSON Formats

NDJSON (Newline-Delimited JSON)

Each JSON object is written on a single line (one record per row).
This format can be ingested efficiently **without multiline mode**.

```
spark.read.format("json").load(path)
```

Multiline / Complex JSON

When a single JSON object or an array spans multiple lines, the reader must be configured explicitly:

```
.option("multiline", "true")
```

Compute Selection for Job Runs

Serverless vs. Classic

- **Serverless Jobs are recommended** for most ETL workloads
- Instant startup

- No cluster configuration
- Lower operational overhead

Job Clusters vs. All-Purpose Clusters

- Production jobs should **never run on All-Purpose clusters**
- All-Purpose clusters are approximately **2× more expensive**
- Always use **Job clusters** for scheduled and production workloads

Notebook Parameterization

Why Dynamic Parameterization?

Dynamic parameterization enables the reusability of the same notebook without hardcoding values. It allows notebooks to be executed for specific time windows, making them flexible for different operational scenarios. By parameterizing inputs, the same notebook can support daily scheduled runs as well as historical backfill runs, reducing code duplication and improving maintainability.

When to Use?

In my use case, a scheduled Python script runs at predefined intervals and uploads JSON files into a landing bucket. Once a new file arrives, Databricks Auto Loader detects it and triggers the downstream pipeline.

The pipeline consists of three tasks:

1. **Bronze ingestion** — The newly arrived file is read as a text file from the landing bucket and loaded into a Bronze table.

2. **Silver transformation** — A parameterized notebook is triggered to process data from the Bronze layer to the Silver layer.
3. **Incremental processing logic** — The Silver notebook does not load the entire Bronze dataset. Instead, it applies dynamic filtering based on parameters.

For regular scheduled runs, no parameters are explicitly passed. In this case, the notebook automatically processes only the last four hours of data and performs a merge operation to avoid duplicates.

For backfill scenarios, date/time parameters are passed through Databricks job parameters. This allows the same notebook to process historical data for a specific time range without any code changes.

This approach ensures flexibility, supports both incremental and backfill processing, and promotes clean, production-ready notebook design.

```
#set widgets to pass values
dbutils.widgets.text("start_date", "")
dbutils.widgets.text("end_date", "")

#extract values from widgets
raw_start = dbutils.widgets.get("start_date").strip()
raw_end = dbutils.widgets.get("end_date").strip()


# Backfill Mode: Triggered when parameters are provided in the Job UI
if len(raw_start) > 0 and len(raw_end) > 0:
start_date = datetime.strptime(raw_start, "%Y-%m-%d") - timedelta(hours=4)
end_date = datetime.strptime(raw_end, "%Y-%m-%d") + timedelta(hours=1)
print(f"Running in BACKFILL mode for range: {start_date} to {end_date}")

# Automated Mode: Default behavior for File Arrival triggers (last 4 hours)
else:
start_date = datetime.now() - timedelta(hours=4)
```

```
end_date = datetime.now() + timedelta(hours=1)
print(f"Running in AUTOMATED mode for range: {start_date} to {end_date}")

# 3. Extract incremental data from Bronze
silver_df = (
spark.read.format("delta")
.table("dev.bronze.coins_price_best_practice")
.filter(f"LoadTime BETWEEN '{start_date}' AND '{end_date}'")
)
```

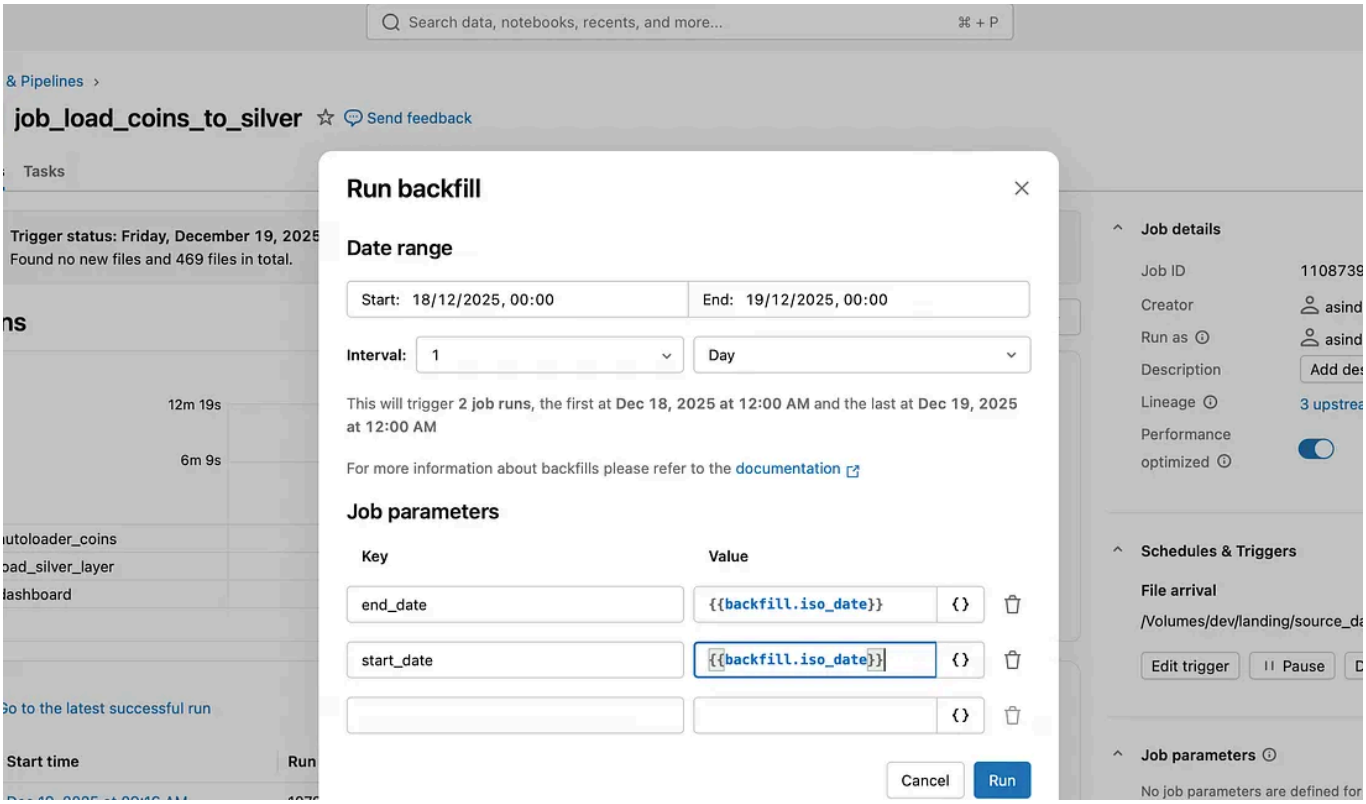
When the notebook is configured as a task within a Databricks job, two parameters are defined at the job level. These parameters are populated during backfill runs to pass the required date or time range, while regular runs rely on current timestamps.

As shown in the screenshot above, two parameters are defined at the **Databricks job level**. These parameters are exposed when the job is triggered using the **Backfill** option, as shown in the screenshot below.

During backfill execution, Databricks creates **separate job runs for each backfill interval**. In this example, two concurrent runs are generated for **18th December** and **19th December**. For each run, Databricks automatically injects the corresponding parameter values based on the backfill configuration.

This approach ensures that each backfill run processes its assigned date range independently, while reusing the same parameterized notebook logic

without any code changes.



Best Practice: Extract JSON into lakehouse

Recommended Approach

- Read JSON files **as text**
- Store the **entire file content as a single string column**
- Add ingestion metadata (filename, file_path, ingestion_time, source_system)

Why Read JSON as Text in Bronze?

Reading JSON as text **never fails**, regardless of:

- Schema evolution
- Nested structure changes

- New or missing fields
- Data type changes
- Partially malformed JSON

JSON parsing happens **before schema evolution.**
If parsing fails, schema evolution **cannot help.**

By storing raw JSON as text:

- You preserve the **exact source payload**
- No data loss occurs
- Bronze remains **schema-agnostic**
- The pipeline is resilient to source changes

If parsing fails later in Silver:

- You can reprocess from Bronze
- No need to re-extract from source systems
- Full auditability is preserved

Bronze Layer Principles

- Append-only
- No transformations
- No schema enforcement
- No deduplication
- No business logic

Bronze represents the **system of record.**

Silver Layer Responsibilities

All structure and quality enforcement happens here:

- Parse JSON using a **defined schema**
- Flatten nested structures
- Explode arrays
- Deduplicate using window functions
- Apply data quality rules
- Drop or isolate invalid records
- Enforce schema evolution deliberately

Error Handling & Dead Letter Queue (DLQ)

_rescued_data (Auto Loader Only)

- Captures fields that **do not match the schema**
- Works only with **Auto Loader** & **COPY INTO**

- Does not work with from_json / parse_json
- Does not protect against malformed JSON

Use this when:

- You must parse JSON in Bronze
- You want limited schema drift tolerance

Validation in Silver

- Apply explicit validation rules: isNotNull, Range checks, Referential checks
- Invalid records can be: Dropped, Redirected to a DLQ table

This keeps Silver clean and trusted.

Streaming Considerations (Silver Layer)

Watermarking

Watermarking is required **only when Silver is running as a streaming job**.

Without a watermark:

- Spark keeps the state forever
- Deduplication windows never close
- A hanging job can grow endlessly
- Jobs can run for days and consume memory

With a watermark:

- Spark knows **how long to wait for late data**
- Old state is safely cleaned up
- Streaming jobs remain stable and bounded

Watermarking (Silver Layer — Streaming Jobs)

When the Silver layer is implemented as a streaming job, watermarking must be applied on an **event-time column** (for example, last_updated).

A watermark defines how long Spark should wait for late-arriving data before discarding old state. This is especially important for:

- Stateful deduplication
- Aggregations
- Preventing unbounded state growth

If watermarking is not set, a streaming job can retain state indefinitely and may run for days if the pipeline hangs or receives delayed data.

```
from pyspark.sql.functions import col
df_structured = df_structured \
.withWatermark("last_updated", "5 minutes")
```

Backfilling Strategy

- Parameterize notebooks (date range, interval)

- Run parallel backfill jobs by partition
- Backfill reads from Bronze, not source
- Can run alongside regular ingestion safely

Trigger Types (Corrected)

trigger(availableNow=True)

- Processes all available data
- Uses multiple micro-batches
- Recommended for large backfills

trigger(once=True)

- Processes all data in a single batch
- Simpler, but less scalable

How to load Json from landing bucket into Bronze Ingestion (JSON as text)

```
from pyspark.sql.functions import col, current_timestamp
checkpoint = "/Volumes/dev/bronze/checkpoint/coins_price_best_practice/"
schema = "/Volumes/dev/bronze/schema/coins_price_best_practice/"
df = (
spark.readStream
.format("cloudFiles")
.option("cloudFiles.format", "text")
.option("cloudFiles.schemaLocation", schema)
.load("/Volumes/dev/landing/source_data/")
)
df = df.select(
col("value").alias("jsonArray"),
col("_metadata.file_name").alias("file_name"),
col("_metadata.file_path").alias("file_path"),
```

```
current_timestamp().alias("LoadTime")
)
df.writeStream \
.format("delta") \
.option("checkpointLocation", checkpoint) \
.trigger(once=True) \
.table("dev.bronze.coins_price_best_practice")
```

Writing into the Silver Layer

In the Silver layer, the goal is to **transform raw JSON data into a structured, reliable, and analytics-ready format.** Unlike the Bronze layer, where data is stored as-is, the Silver layer is where **schema enforcement and validation are intentionally applied.**

Define and Enforce the Schema

Rather than relying on schema inference or automatic schema evolution, the recommended approach in the Silver layer is to **explicitly define the expected JSON schema.** This ensures:

- Consistent data types across all records.
- Early detection of schema drift.
- Predictable downstream behavior for analytics and reporting.

```
from pyspark.sql.functions import *
from pyspark.sql.types import *
schema = StructType([StructField("id", StringType(), True),
StructField("symbol", StringType(), True),
StructField("name", StringType(), True),
StructField("image", StringType(), True),
StructField("current_price", FloatType(), True),
StructField("market_cap", FloatType(), True),
StructField("market_cap_rank", IntegerType(), True),
```

```
    StructField("fully_diluted_valuation", FloatType(), True),
    StructField("total_volume", FloatType(), True),
    StructField("high_24h", FloatType(), True),
    StructField("low_24h", FloatType(), True),
    StructField("price_change_24h", FloatType(), True),
    StructField("price_change_percentage_24h", FloatType(), True),
    StructField("market_cap_change_24h", FloatType(), True),
    StructField("market_cap_change_percentage_24h", FloatType(), True),
    StructField("circulating_supply", FloatType(), True),
    StructField("total_supply", FloatType(), True),
    StructField("max_supply", FloatType(), True),
    StructField("ath", FloatType(), True),
    StructField("ath_change_percentage", FloatType(), True),
    StructField("ath_date", TimestampType(), True),
    StructField("atl", FloatType(), True),
    StructField("atl_change_percentage", FloatType(), True),
    StructField("atl_date", TimestampType(), True),
    StructField("roi", StructType([
                StructField("times", DoubleType()),
                StructField("currency", StringType()),
                StructField("percentage", DoubleType())
                ])),
    StructField("last_updated", TimestampType(), True)
    ])
```

In the code below, data is read from the Bronze table using a parameter-driven approach.
If no parameters are provided, the notebook performs a **regular incremental load**, extracting only the last four hours of data from the Bronze layer.
When parameters are supplied (for example, during a backfill), the same logic dynamically switches to load a **specific date or time range** based on the provided values.

This design makes the notebook **fully reusable** for both daily incremental processing and historical backfills without any code changes.
Parameterization is therefore a key practice for building scalable, reliable, and maintainable incremental data pipelines.

```
from datetime import datetime, timedelta

if len(start_date) > 0 and len(end_date) > 0:
start_date = datetime.strptime(start_date, "%Y-%m-%d") - timedelta(hours=4)
end_date = datetime.strptime(end_date, "%Y-%m-%d") + timedelta(hours=1)
else:
start_date = datetime.now() - timedelta(hours=4)
end_date = datetime.now() + timedelta(hours=1)
silver_df = (
spark.read.format("delta")
.table("dev.bronze.coins_price_best_practice")
.filter(
f"LoadTime BETWEEN '{start_date}' AND '{end_date}'"
)
)
display(silver_df)
```

Code Explanation:

1. **Parse JSON:**
   The from_json function is used to convert the JSON string column (jsonArray) from the Bronze table into a structured array of JSON objects based on the defined schema.
2. **Explode JSON Array:**
   Since each row contains an array of JSON objects, explode is used to **split each array element into separate rows**, so that each JSON object becomes a separate row in the DataFrame.
3. **Select Structured Columns:**
   Finally, .select("jsonArray.*") flattens the JSON object into individual columns based on the schema.

```
from pyspark.sql.functions import from_json, col
from pyspark.sql.types import ArrayType
```

```
from pyspark.sql.functions import explode
df_parsed = silver_df.withColumn(
"jsonArray",
from_json(col("jsonArray"), ArrayType(schema))
)
df_structured = df_parsed.select(explode(col("jsonArray")).alias("jsonArray")).s
```

Then flatten the ROI column into the main dataframe

```
df_structured = df_structured\
.withColumn("roi_times", col("roi.times")) \
.withColumn("roi_currency", col("roi.currency")) \
.withColumn("roi_percentage", col("roi.percentage")) \
.withColumn("key",concat(col("id"),col("last_updated").cast("int")))\
.drop("roi")
```

Remove duplicates using window functions and load them into the temp table before merging into the main table.

```
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number
windowspec = Window.partitionBy("key").orderBy(col("last_updated").desc())
dedup = df_structured.withColumn("rn",row_number().over(windowspec)).filter(col(
dedup.write.mode("overwrite").option("mergeSchema", "true").format("delta").save
```

Finally runs the Merge query to ensure no duplicates in the final table.

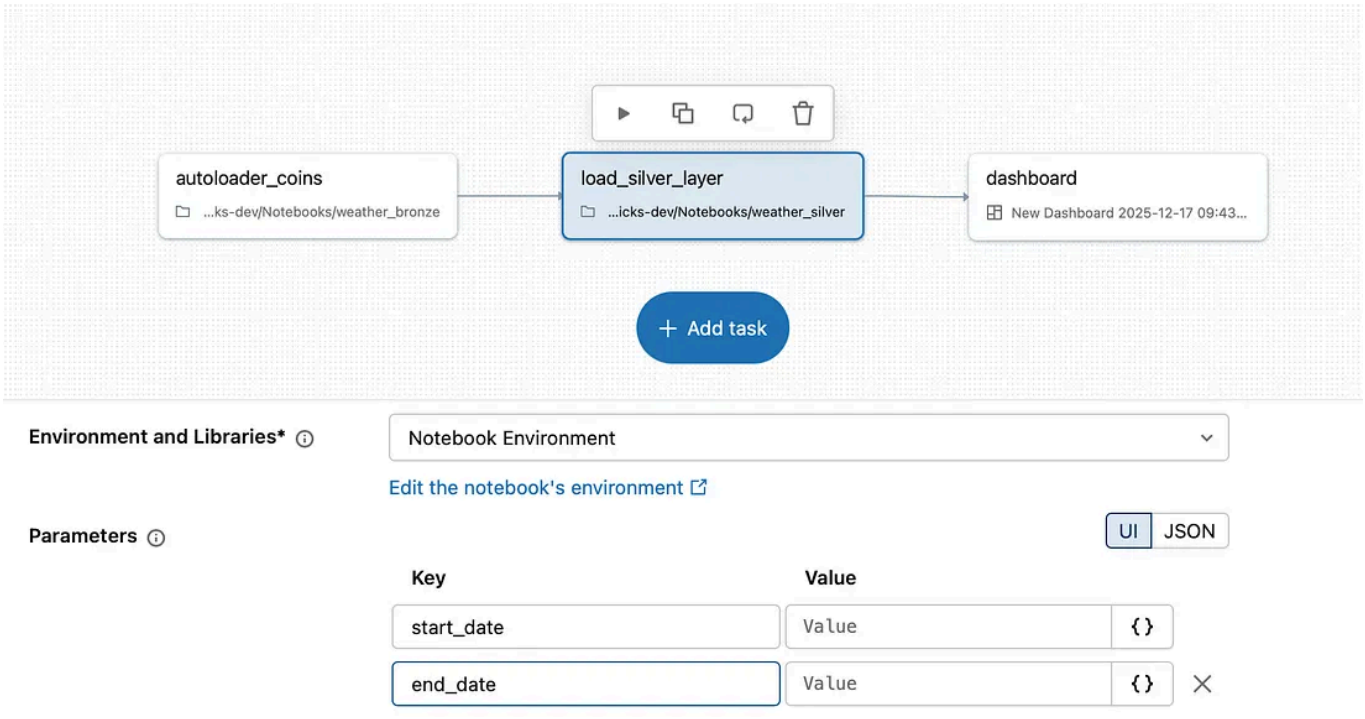
```
%sql
merge into dev.silver.coins_price_best_practice as t
using dev.silver.coins_price_best_practice_temp as s
on t.key = s.key
when matched then update set *
when not matched then insert *
```

Orchestrating Data Pipelines in Databricks

Databricks provides powerful orchestration capabilities that go beyond just running notebooks. You can design end-to-end pipelines that not only **load and transform data** but also **trigger downstream actions** automatically. For example:

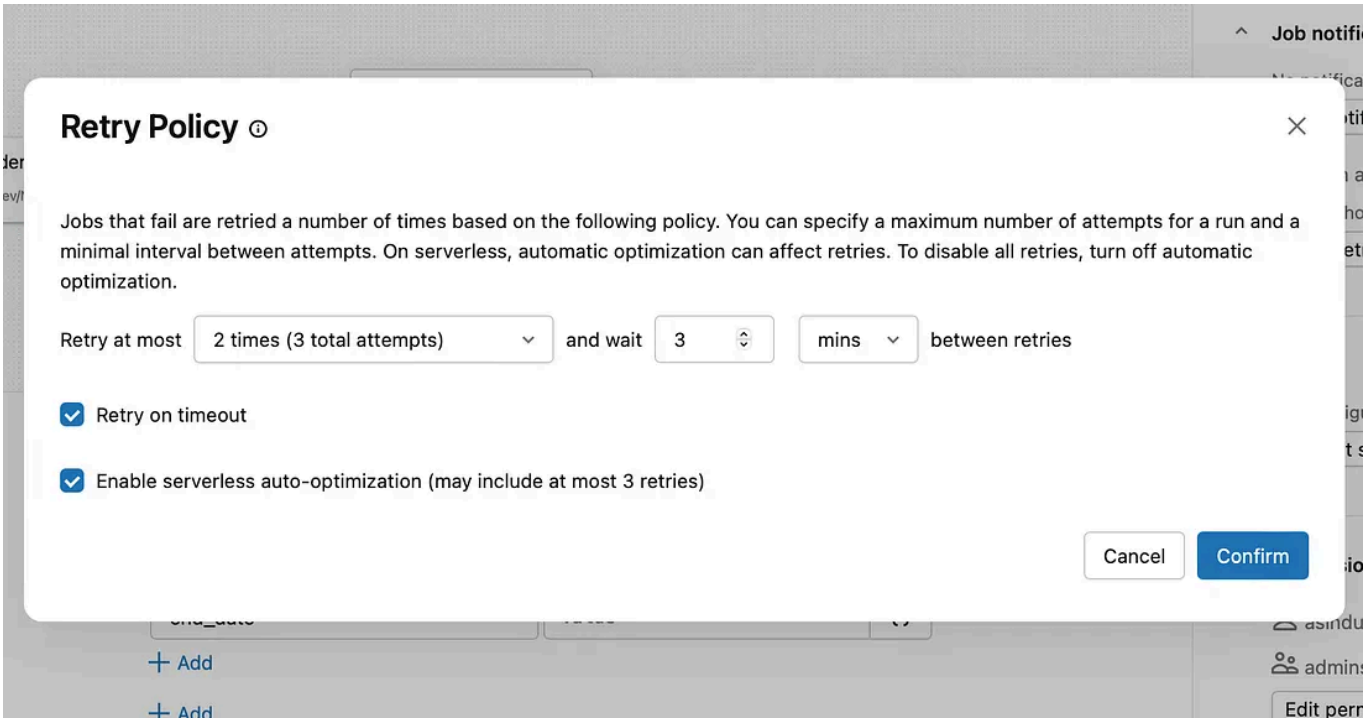
- **Dashboard Refresh:** Once data is loaded into tables, the pipeline can automatically refresh dashboards — such as **Databricks dashboards** or **Power BI dashboards** — to ensure the latest data is reflected.
- **Model Refresh:** You can also trigger **DBT models** or other analytical workflows as part of the same pipeline.

This makes Databricks pipelines fully integrated, allowing **data ingestion, transformation, and reporting to work seamlessly together**.



Retry Policy:

You can configure retry policies at the **job or notebook level**, so if a pipeline fails due to transient issues, it automatically reruns a set number of times. This ensures reliability without manual intervention.



Setting Notifications in Databricks

Databricks provides flexible options to configure notifications at multiple levels: **Notebook, Job, or Dashboard**, helping teams stay informed about pipeline and report status.

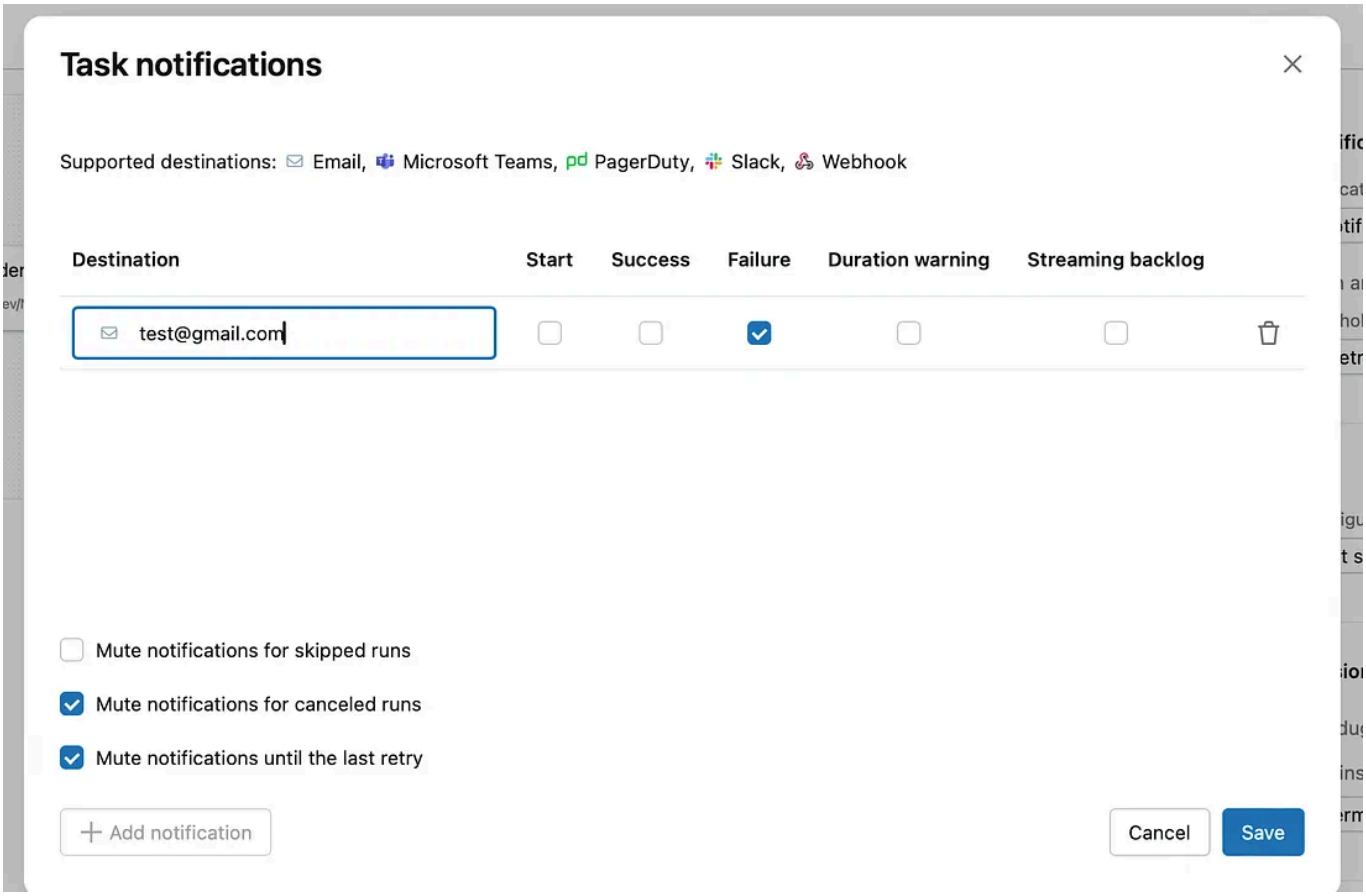
Notebook or Job Notifications:

- Configure notifications to trigger when a **Notebook or Job succeeds, fails, or starts running**.
- Alerts can be sent via **email or webhook**, allowing stakeholders to stay informed or trigger downstream processes automatically.

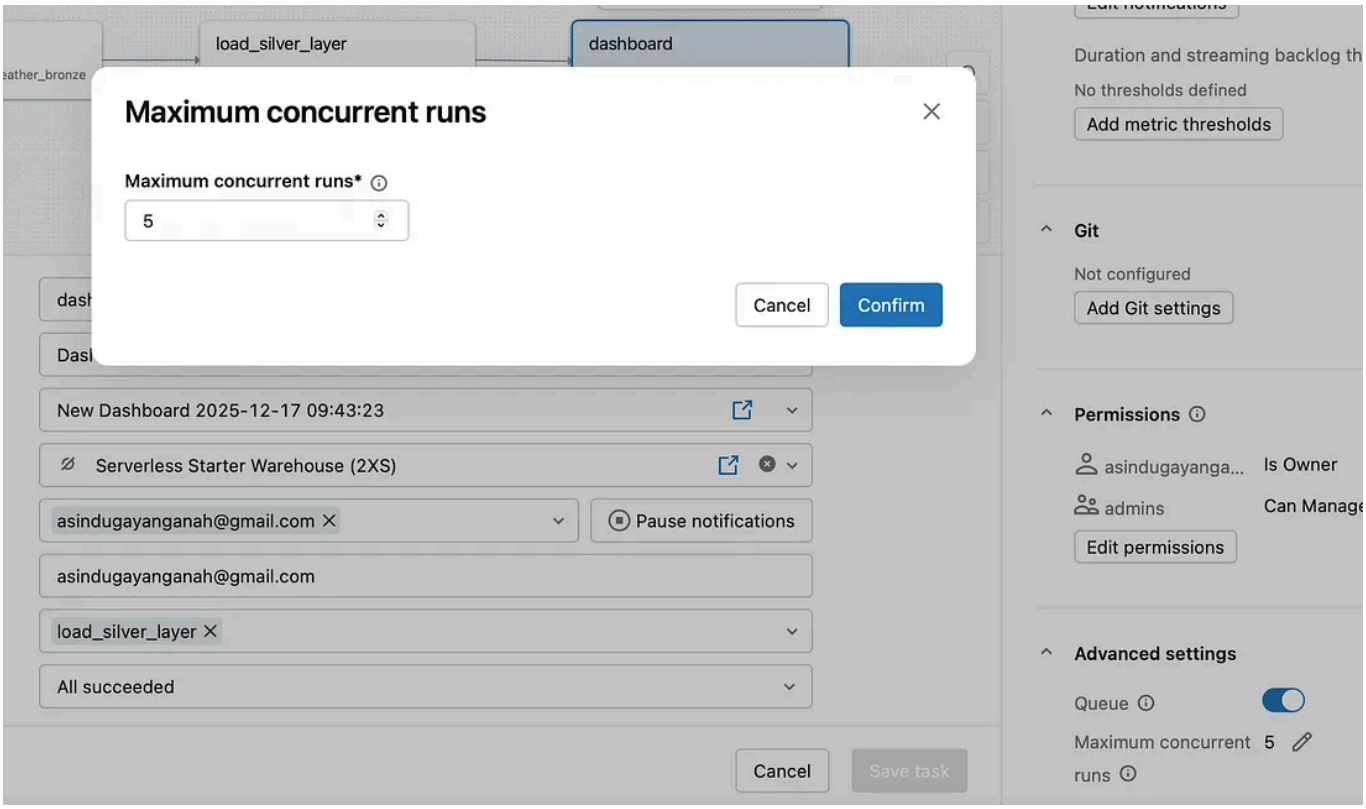
Dashboard Notifications:

- You can also set notifications on **Databricks dashboards**.
- Whenever a dashboard is refreshed, notifications can automatically **send updated dashboard pages** to your selected destinations, such as **email, Teams, or Slack**.
- This ensures that stakeholders always have the **latest view** without manually checking the dashboard, making it easier to track and act on updated data.

By effectively combining notifications with pipeline orchestration, you **increase reliability, reduce manual monitoring**, and provide a seamless way for teams to stay up-to-date with both **pipeline status and refreshed dashboard data**.



Maximum Concurrent Runs



Databricks allows setting the maximum number of concurrent runs per job in the advanced settings. This helps scale jobs efficiently, avoid overlapping runs, and ensures reliable execution when multiple triggers or backfills occur.

Thresholds
Databricks allows setting thresholds at the job or notebook level to prevent long-running or stuck jobs. Thresholds are important for reliability, cost control, and operational safety.

Importance:
If thresholds are not set, a stuck or hung job can continue running indefinitely, potentially for days, consuming resources and delaying downstream processes. Setting thresholds ensures that long-running jobs are detected, stopped, and retried if needed, keeping pipelines reliable and efficient.

There are two types of thresholds:

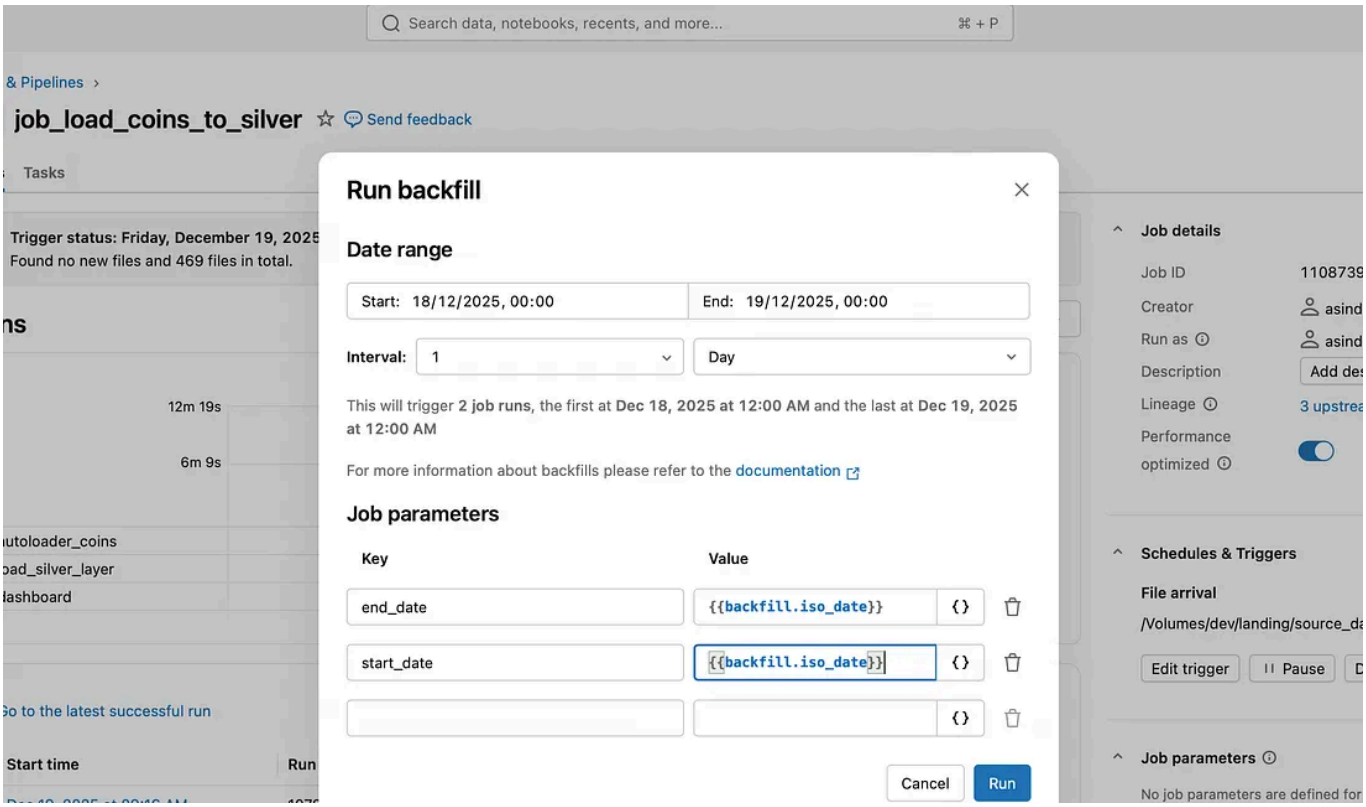
1. **Warning Threshold** — Sends an alert if the job exceeds a specified run time, allowing you to monitor potential issues.
2. **Timeout Threshold** — Automatically fails the job if it runs longer than the set limit. If a retry policy is configured, the job will rerun according to the defined rules.

Backfill with Databricks Jobs

Backfills should always be **controlled through job parameters** rather than code changes. By parameterizing notebooks (for example, start date, end

date, and interval), the same job can be reused for both regular incremental runs and historical backfills.

When a backfill is triggered, Databricks **creates multiple concurrent job runs** based on the defined backfill interval.
For example, if the backfill range includes **18th and 19th December**, Databricks will create **two parallel runs**, each processing its own date range. The level of parallelism is governed by the **Maximum Concurrent Runs** setting (for example, up to 100 concurrent runs).



Backfills are ideally executed **from the Bronze layer**, since Bronze contains all source data preserved as-is. This approach avoids re-extracting data from the source system and ensures consistency and auditability.

If data must be reloaded from the source system, the recommended approach is to ingest the updated data as **new files** into the landing zone.

Because Databricks Auto Loader tracks processed files using checkpoints, **the same file cannot be reprocessed from the landing bucket**. Reprocessing is therefore only possible by either:

- Loading newly ingested files, or
- Replaying data from the Bronze layer.

This design ensures reliable, scalable, and repeatable backfills without compromising idempotency or data integrity.

Databricks Dashboards

To create and run **Databricks dashboards**, a **SQL Warehouse** is required as the compute engine. Dashboards execute SQL queries directly against the warehouse and are tightly integrated with Databricks tables.

Databricks dashboards support **parameterized queries**, allowing users to pass values from dashboard controls (such as dropdowns or date pickers) to dynamically filter datasets. To enable this, the underlying SQL queries must be written using parameters.

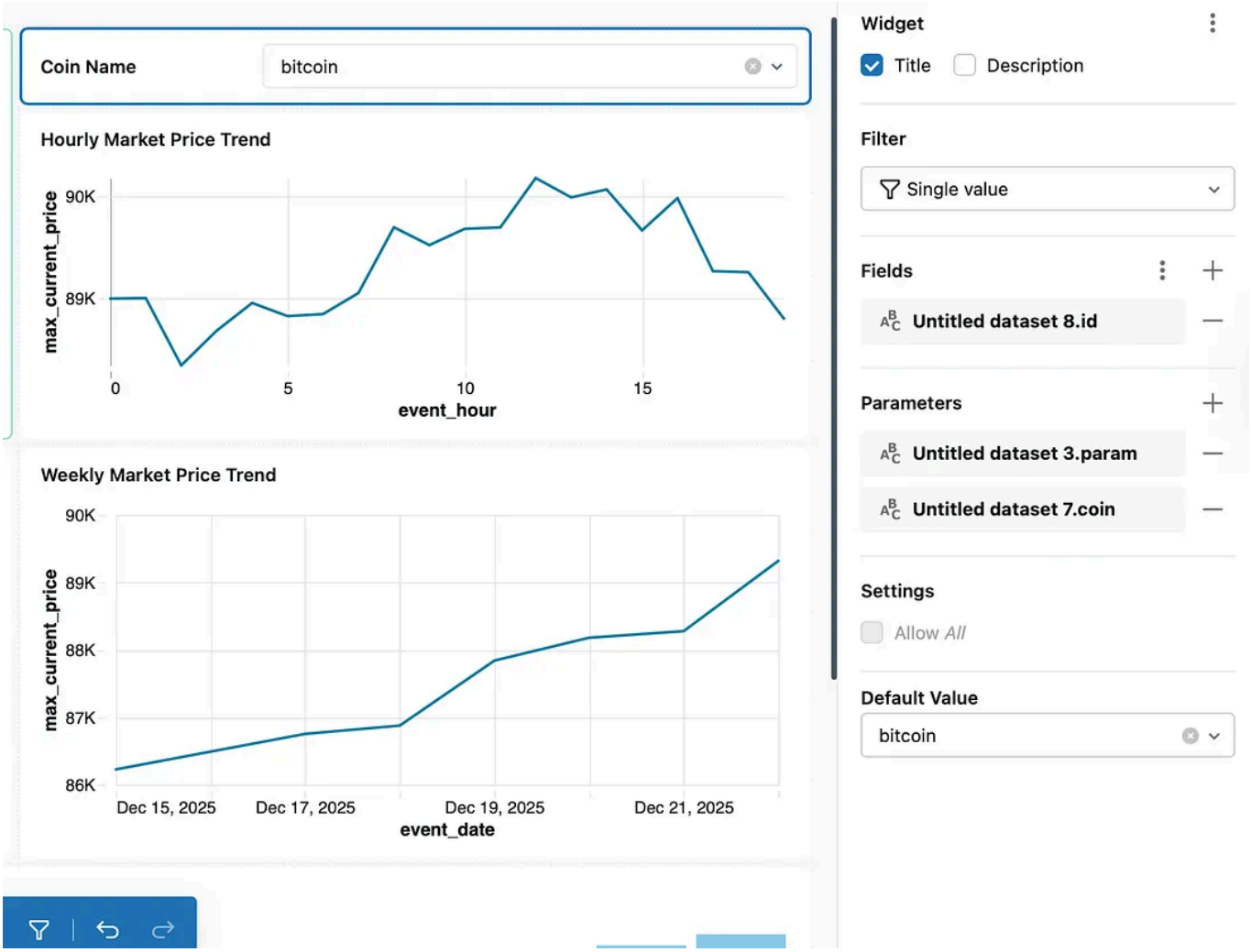
While Databricks dashboards are well-suited for **operational analytics and lightweight reporting**, their visualization capabilities are more limited compared to tools like **Power BI**, which offer richer visuals and advanced formatting options.

Passing Parameters from Dashboards to Queries

Dashboard input controls are mapped to SQL query parameters. When a user selects a value in the dashboard, that value is automatically injected into the parameterized query, and the dashboard refreshes accordingly.

This approach enables:

- Dynamic filtering without modifying SQL code
- Reusable queries across multiple dashboard views
- Interactive analysis directly within Databricks



Data Lineage

Databricks automatically captures **end-to-end data lineage** when Unity Catalog is enabled. Lineage is tracked regardless of whether transformations are written using **PySpark, Spark SQL, or Delta Live Tables.**

Lineage is recorded across the full lakehouse stack, including:

- **Source volumes / cloud storage locations**
- **Bronze, Silver, and Gold Delta tables**
- **Views and materialized views**
- **Jobs, notebooks, and SQL queries**

- **Dashboards and downstream consumers**

This lineage is available directly in the **Unity Catalog UI**, without any additional configuration.

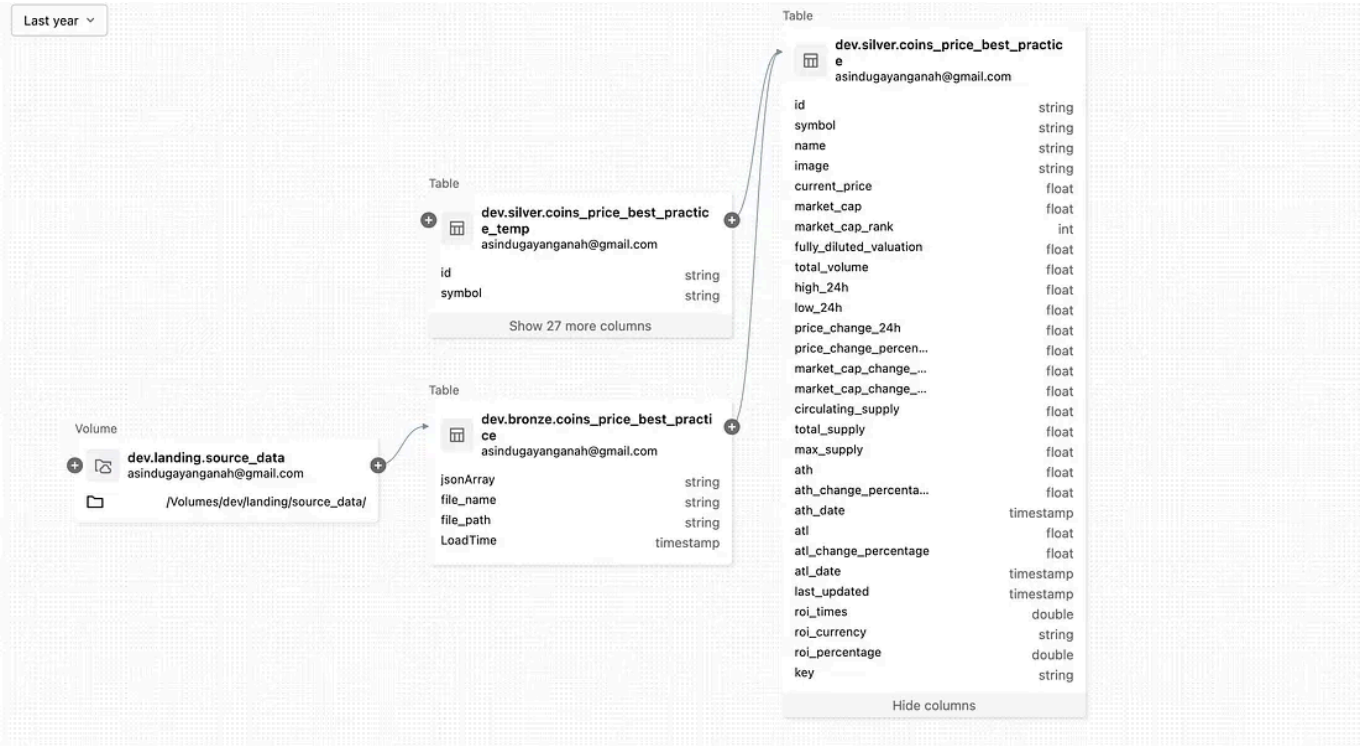
What Databricks Lineage Shows

Databricks lineage allows you to:

- Trace **where the data originated** (volume, external location, or table)
- See **which notebooks, jobs, or SQL queries** produced a table
- Identify **downstream dependencies** such as dashboards and reports
- Perform **impact analysis** before changing schemas or logic

Lineage is captured at:

- **Table level**
- **Column level** (for supported operations)
- **Volume / external location level**



Why Data Lineage Is Important

- **Impact Analysis** — Understand what breaks before changing a table or column
- **Debugging** — Quickly trace data quality issues back to the source
- **Governance & Auditing** — Meet compliance and audit requirements
- **Operational Confidence** — Know exactly how data flows through the platform

Written by Asindu Gayangana

1 follower · 2 following

Data Engineer

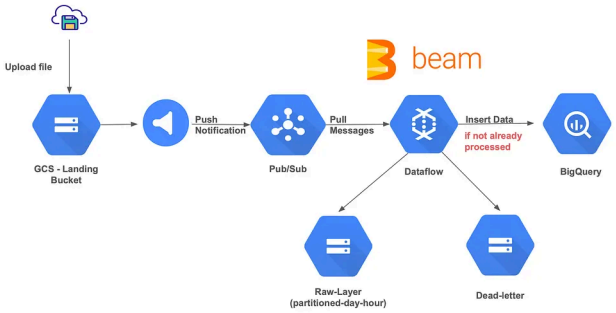
No responses yet

Write a response

What are your thoughts?

More from Asindu Gayangana

ronment	Resource Group Name	Synapse Workspace	ADLS Gen2 Account	File System Name	Key Va Name
elopment	RG_Development	workspace-development	developmentadls	developmentfilesystem	kv-development-0
uction	RG_Production	workspace-production	productionadls	productionfilesystem	kv-production-00



Asindu Gayangana

Best Practices for Azure Synapse CI/CD Deployment

This article describes how to develop a Continuous Integration and Continuous...

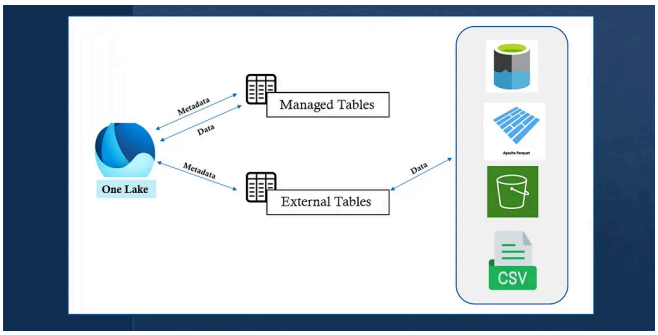
Oct 13, 2025

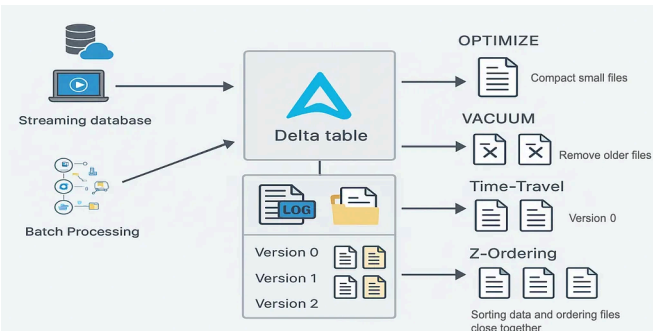
Asindu Gayangana

Building a Real-Time Data Pipeline on Google Cloud Platform (GCP)

I have designed and implemented a real-time data pipeline on Google Cloud Platform...

Nov 20, 2025





Asindu Gayangana

Managed vs External Tables in Spark / Fabric / Synapse

1. Introduction

Sep 4, 2025

Asindu Gayangana

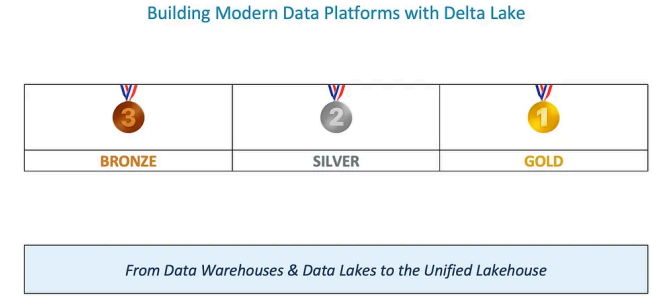
Delta Tables

A Delta Table is a modern table format built on top of Parquet files. It provides version...

Sep 10, 2025

See all from Asindu Gayangana

Recommended from Medium



In EndToEndData by Prem Vishnoi(cloudvala)

Reliable Data Engineering

The Complete Guide To Lakehouse Architecture

1.2 Data Lakes: Scale Without Trust

Dec 30, 2025 75

The Modern Data Stack in 2025: What Actually Won

A data-driven analysis of which tools dominate the modern data ecosystem—and...

6d ago 81 3

Cloud With Azeem

LinkedIn Is Replacing Kafka—Here's Why the Streaming Giant is...

Inside LinkedIn's Bold Move to a New Data Pipeline That Could Change the Future of...

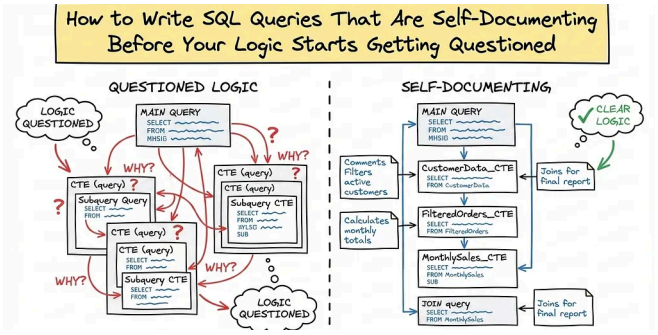
6d ago 55

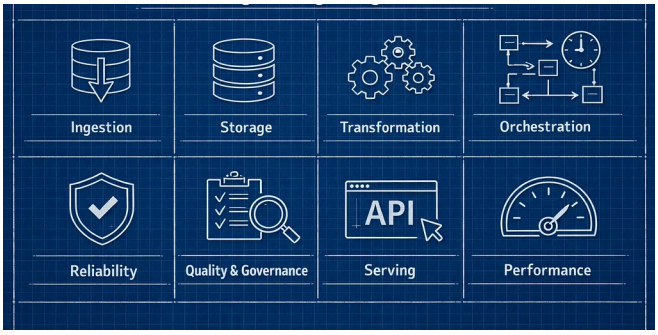
In itversity by Sarath Sagi

Data Engineering System Design

How to Build Systems That Scale, Recover, and Deliver—With Intentionality

Dec 5, 2025 17





Rohan Dutt

How to Write SQL Queries That Are Self-Documenting Before Your...

Structuring queries so intent, assumptions, and calculations explain themselves

Khushbu Shah

Data Engineering Design Patterns You Must Learn in 2026

These are the 8 data engineering design patterns every modern data stack is built on....

6d ago 161 4

4d ago 25 1

See more recommendations