databricks

Build Data Pipelines with Delta Live Tables

Module Agenda

Build Data Pipelines with Delta Live Tables

Introduction to Delta Live Tables

DE 5.1 – DLT UI Walkthrough

DE 5.1A – SQL Pipelines

DE 5.1B – Python Pipelines

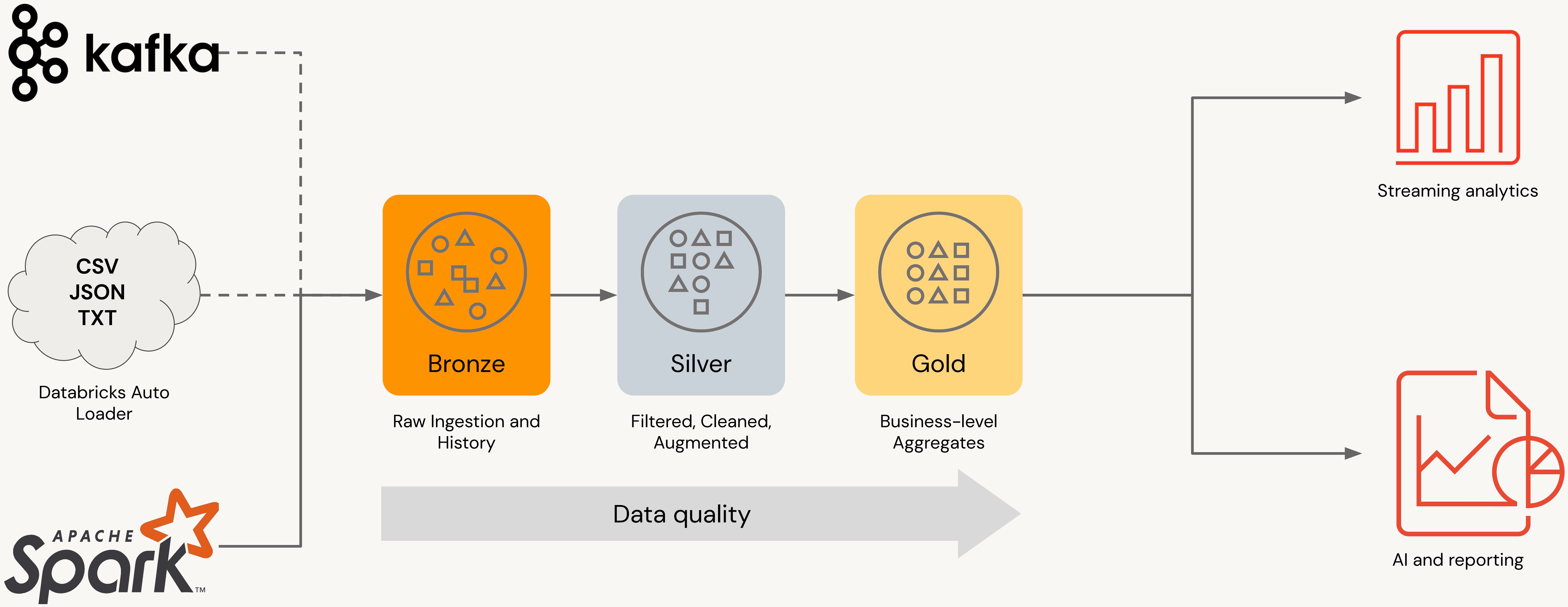
DE 5.2 – Python vs SQL

DE 5.3 – Pipeline Results

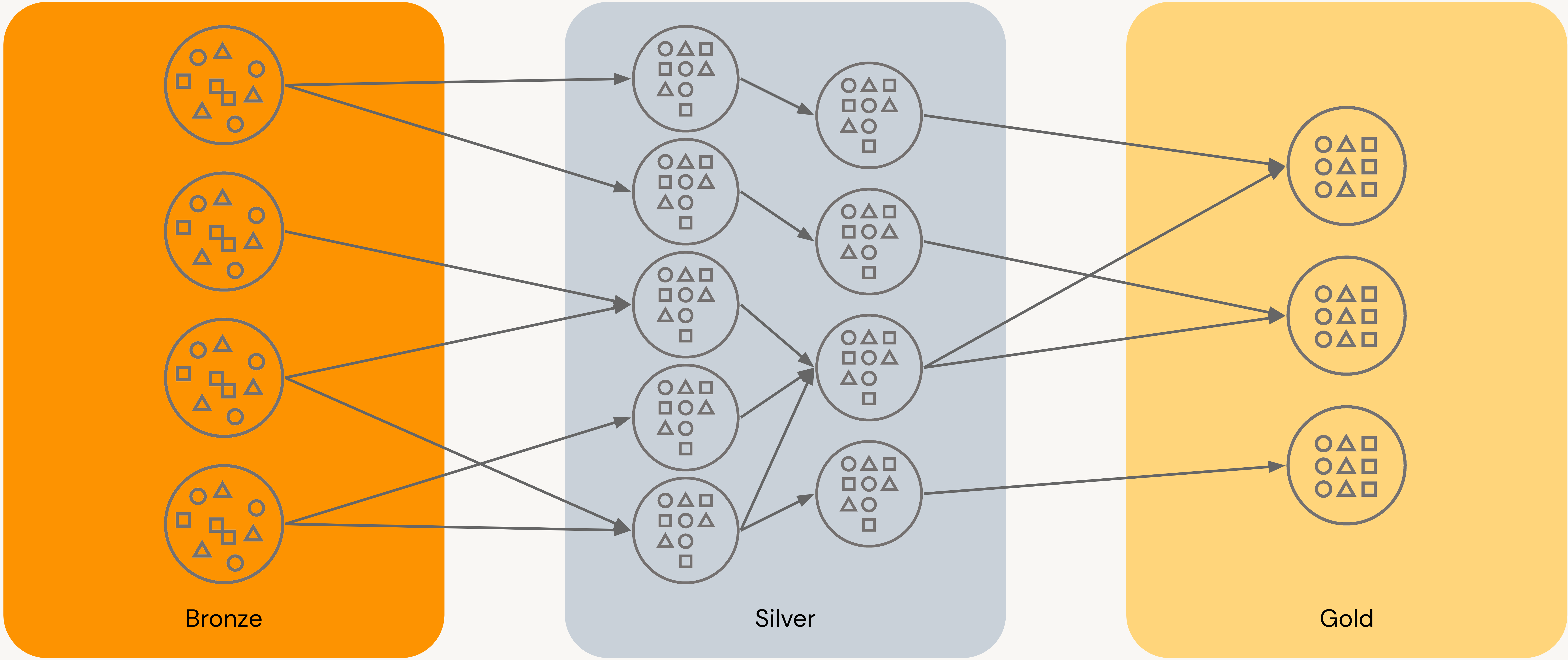
DE 5.4 – Pipeline Event Logs

Introduction to Delta Live Tables

Multi-Hop in the Lakehouse



The Reality is Not so Simple



Large scale ETL is complex and brittle

Complex pipeline development

Hard to build and maintain table **dependencies**

Difficult to switch between **batch** and **stream** processing

Data quality and governance

Difficult to monitor and enforce **data quality**

Impossible to trace data **lineage**

Difficult pipeline operations

Poor **observability** at granular, data level

Error handling and **recovery** is laborious

Introducing Delta Live Tables

Make reliable ETL easy on Delta Lake

Operate with agility

Declarative tools to build batch and streaming data pipelines

Trust your data

DLT has built-in declarative quality controls

Declare quality expectations and actions to take

Scale with reliability

Easily scale infrastructure alongside your data

What is a LIVE TABLE?

What is a Live Table?

Live Tables are materialized views for the lakehouse.

A live table is:

- Defined by a SQL query
- Created and kept up-to-date by a pipeline

```
              LIVE
CREATE ~~OR REPLACE~~ TABLE report
AS SELECT sum(profit)
FROM prod.sales
```

Live tables provides tools to:

- Manage dependencies
- Control quality
- Automate operations
- Simplify collaboration
- Save costs
- Reduce latency

What is a Streaming Live Table?

Based on Spark™ Structured Streaming

A streaming live table is "stateful":

- Ensures exactly-once processing of input rows
- Inputs are only read once

```
CREATE STREAMING LIVE TABLE report
AS SELECT sum(profit)
FROM cloud_files(prod.sales)
```

- Streaming Live tables compute results over append-only streams such as Kafka, Kinesis, or Auto Loader (files on cloud storage)
- Streaming live tables allow you to reduce costs and latency by avoiding reprocessing of old data.

How do I use DLT?

Creating Your First Live Table Pipeline

SQL to DLT in three easy steps…

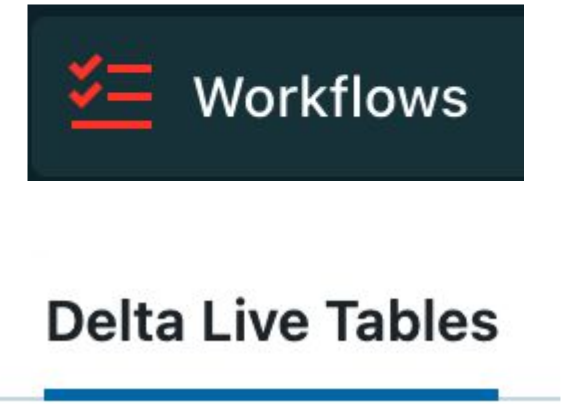
Write create live table

- Table definitions are written (but not run) in notebooks
- Databricks Repos allow you to version control your table definitions.

```sql
CREATE LIVE TABLE daily_stats
AS SELECT sum(rev) - sum(costs) AS profits
FROM prod_data.transactions
GROUP BY day
```

Create a pipeline

- A Pipeline picks one or more notebooks of table definitions, as well as any configuration required.



Click start

- DLT will create or update all the tables in the pipelines.

BEST PRACTICE

Development vs Production

Fast iteration or enterprise grade reliability

Development Mode

- Reuses a **long-running cluster** running for fast iteration.
- **No retries** on errors enabling faster debugging.

Production Mode

- Cuts costs by **turning off clusters** as soon as they are done (within 5 minutes)
- **Escalating retries**, including cluster restarts, ensure reliability in the face of transient issues.

In the Pipelines UI: Development | Production

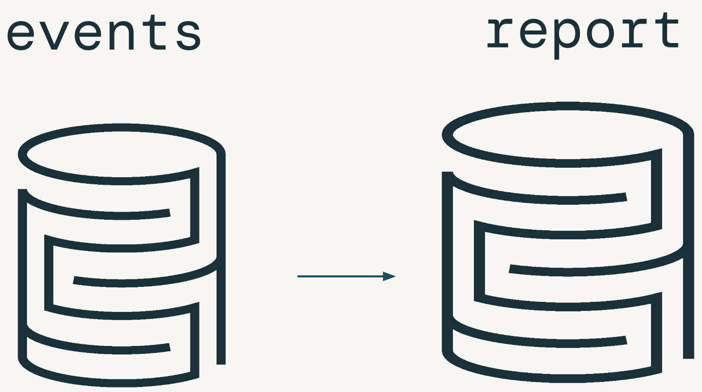
What if I have many tables?

Declare LIVE Dependencies

Using the LIVE virtual schema.

```
CREATE LIVE TABLE events
AS SELECT … FROM prod.raw_data


CREATE LIVE TABLE report
AS SELECT … FROM LIVE.events
```



- Dependencies owned by other producers are just read from the catalog or spark data source as normal.
- LIVE dependencies, from the **same pipeline**, are read from the LIVE schema.
- DLT detects LIVE dependencies and executes all operations in **correct order**.
- DLT handles parallelism and captures the lineage of the data.

How do I know my results are correct?

BEST PRACTICE

Ensure correctness with Expectations

Expectations are tests that ensure data quality in production

```
CONSTRAINT valid_timestamp
EXPECT (timestamp > '2012-01-01')
ON VIOLATION DROP
```

```
@dlt.expect_or_drop(
  "valid_timestamp",
  col("timestamp") > '2012-01-01')
```

Expectations are true/false expressions that are used to validate each row during processing.

DLT offers flexible policies on how to handle records that violate expectations:

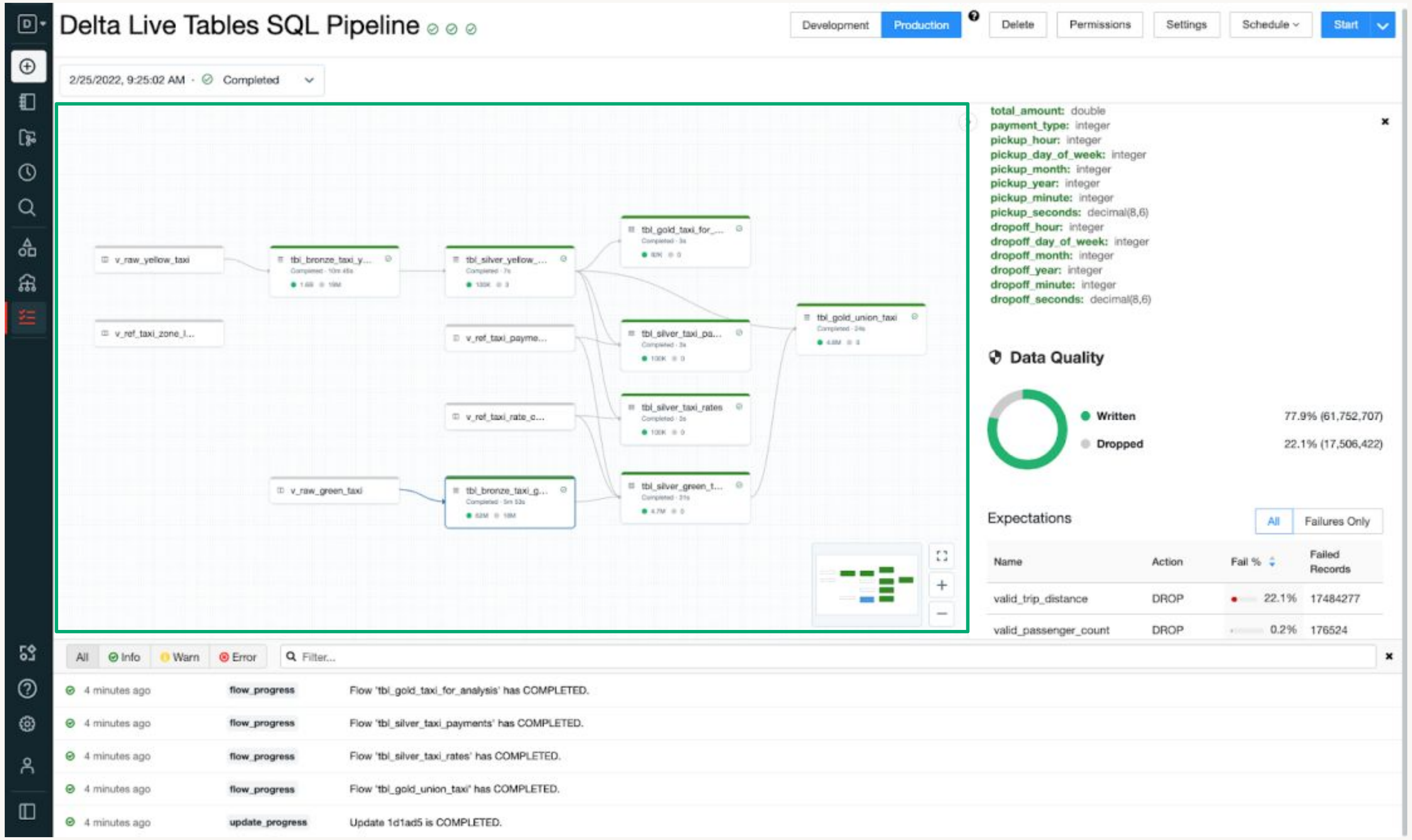
- Track number of bad records
- Drop bad records
- Abort processing for a single bad record

What about operations?

Pipelines UI

A one stop shop for ETL debugging and operations

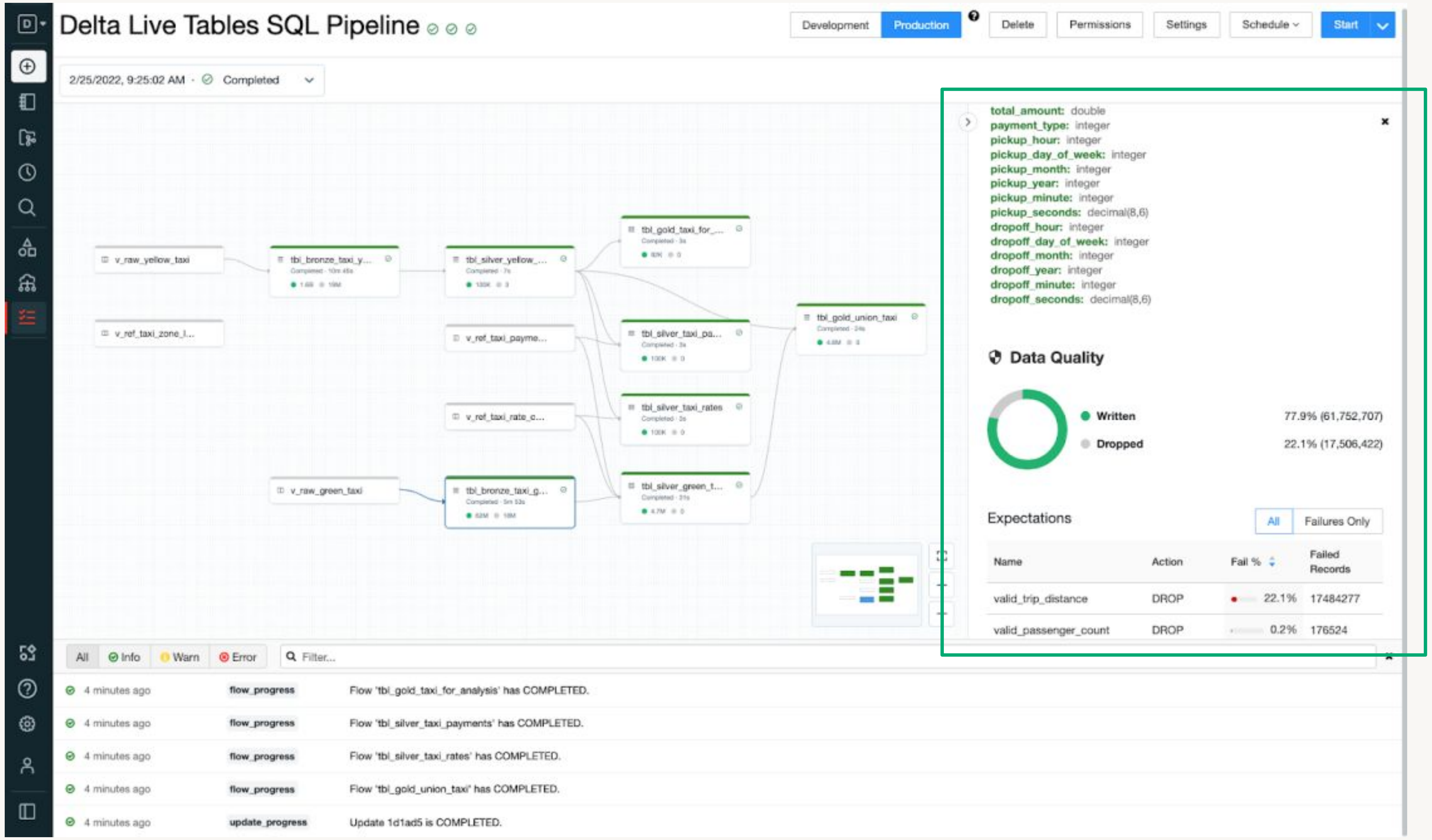
- **Visualize** data flows between tables



Pipelines UI

A one stop shop for ETL debugging and operations

- Visualize data flows between tables
- Discover metadata and quality of each table



Pipelines UI

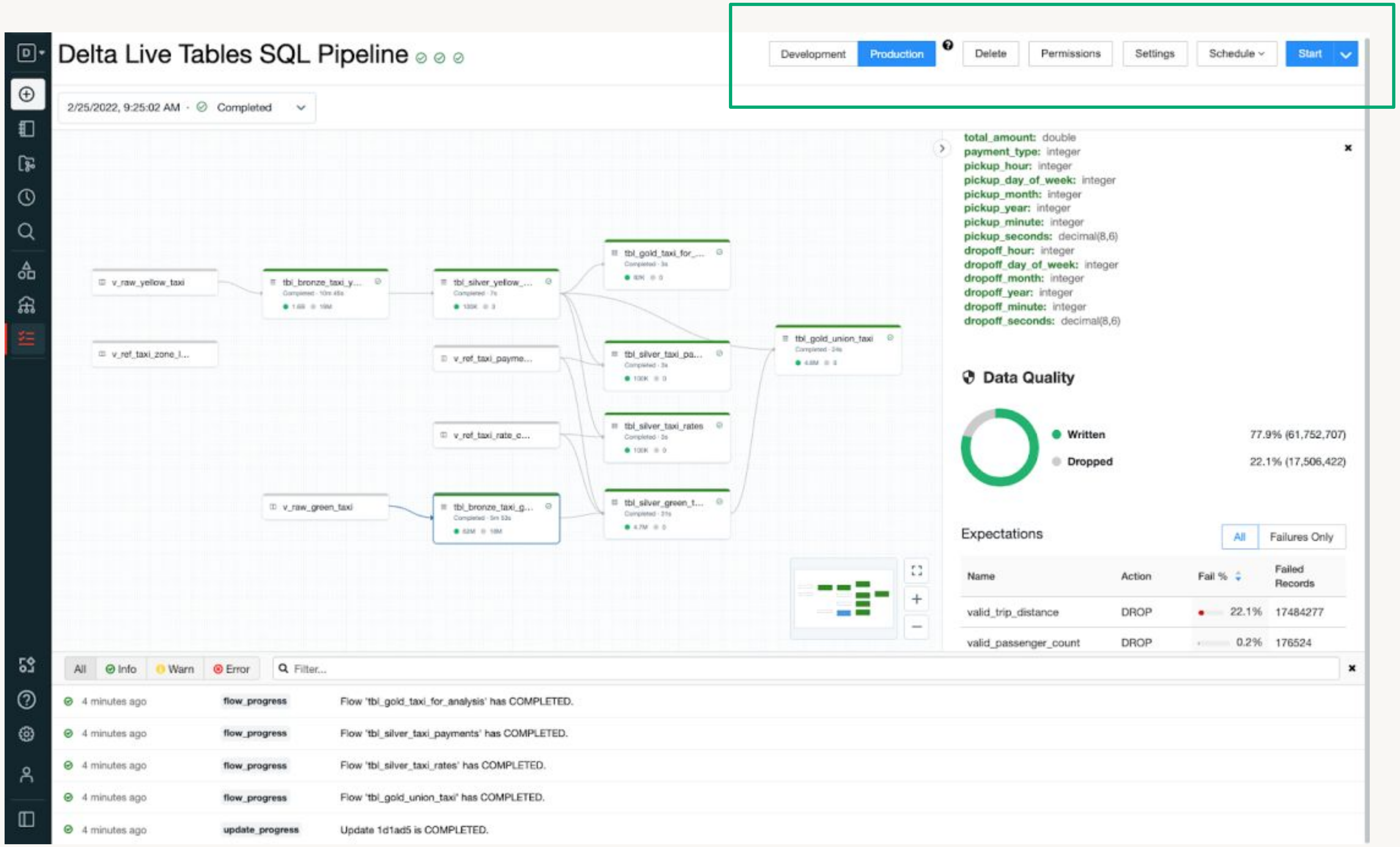
A one stop shop for ETL debugging and operations

- **Visualize** data flows between tables
- **Discover** metadata and quality of each table
- **Access** to historical updates

Pipelines UI

A one stop shop for ETL debugging and operations

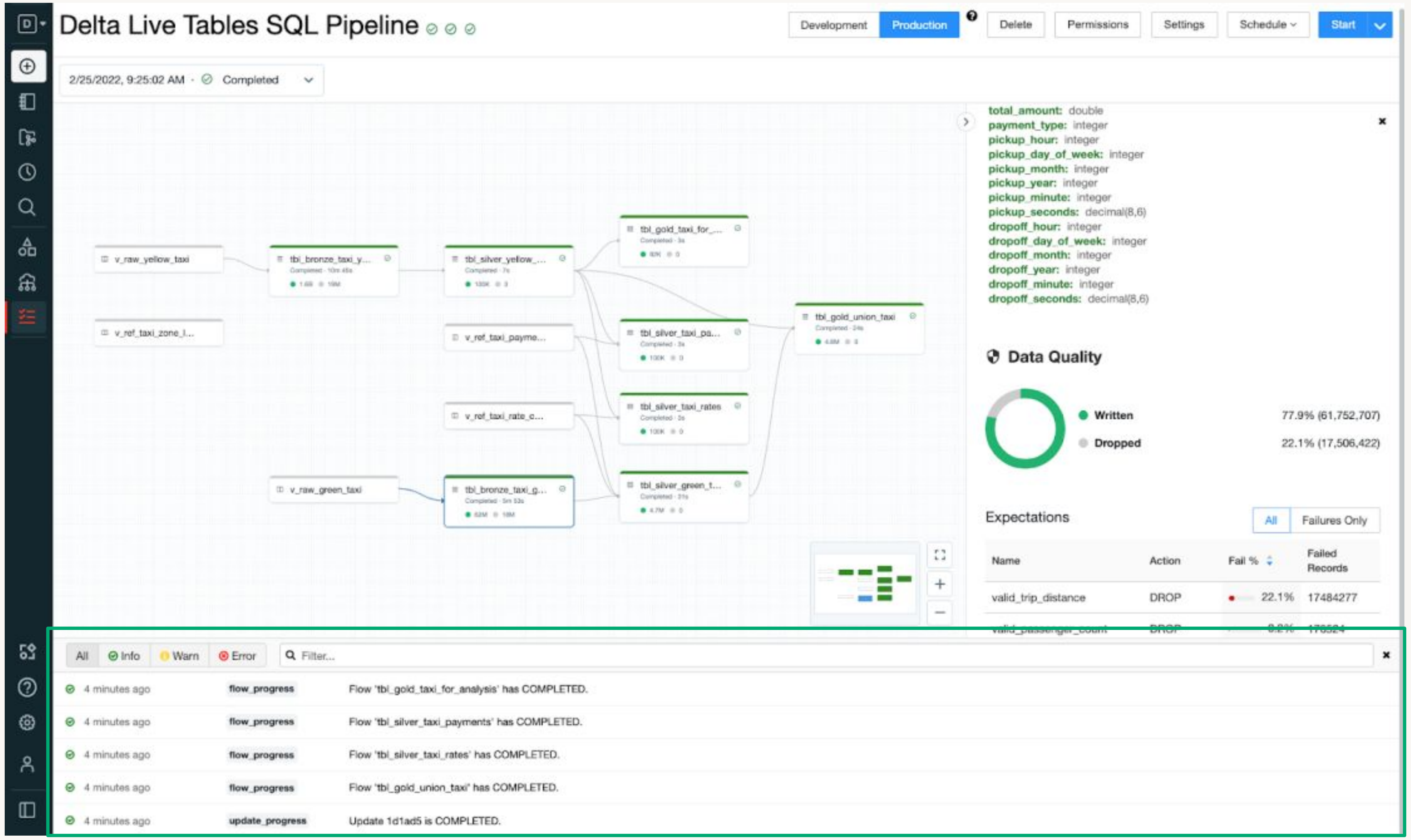
- Visualize data flows between tables
- Discover metadata and quality of each table
- Access to historical updates
- Control operations



Pipelines UI

A one stop shop for ETL debugging and operations

- Visualize data flows between tables
- Discover metadata and quality of each table
- Access to historical updates
- Control operations
- Dive deep into events



The Event Log

The event log automatically records all pipelines operations.

Operational Statistics

Time and current status, for all operations

Pipeline and cluster configurations

Row counts

Provenance

Table schemas, definitions, and declared properties

Table-level lineage

Query plans used to update tables

Data Quality

Expectation pass / failure / drop statistics

Input/Output rows that caused expectation failures

When should I use streaming?

Using Spark™ Structured Streaming for ingestion

Easily ingest files from cloud storage as they are uploaded

```
CREATE STREAMING LIVE TABLE raw_data
AS SELECT *
FROM cloud_files("/data", "json")
```

This example creates a table with all the json data stored in "/data":

- `cloud_files` keeps track of which files have been read to avoid duplication and wasted work
- Supports both listing and notifications for arbitrary scale
- Configurable schema inference and schema evolution

Using the SQL STREAM() function

Stream data from any Delta table

```
CREATE STREAMING LIVE TABLE mystream
  AS SELECT *
  FROM STREAM(my_table)
```

Pitfall: `my_table` must be an append-only source.

e.g. it may not:

- be the target of APPLY CHANGES INTO
- define an aggregate function
- be a table on which you've executed DML to delete/update a row (see GDPR section)

- `STREAM(my_table)` reads a stream of new records, instead of a snapshot
- `Streaming` tables must be an append-only table
- Any append-only delta table can be read as a stream (i.e. from the live schema, from the catalog, or just from a path).

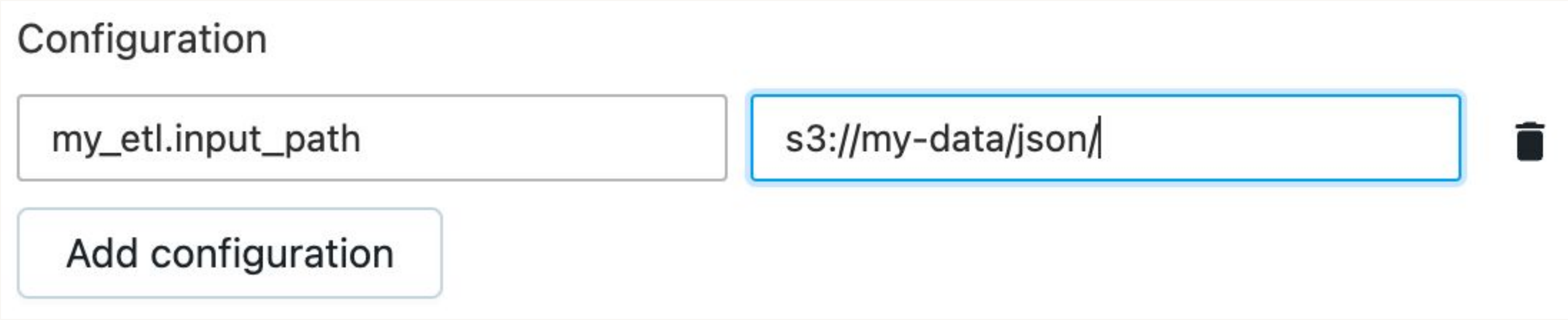
How can I use parameters?

Modularize your code with configuration

Avoid hard coding paths, topic names, and other constants in your code.

A pipeline's configuration is a map of key value pairs that can be used to parameterize your code:

- Improve code readability/maintainability
- Reuse code in multiple pipelines for different data




```
CREATE STREAMING LIVE TABLE data AS
SELECT * FROM cloud_files("${my_etl.input_path}", "json")
```

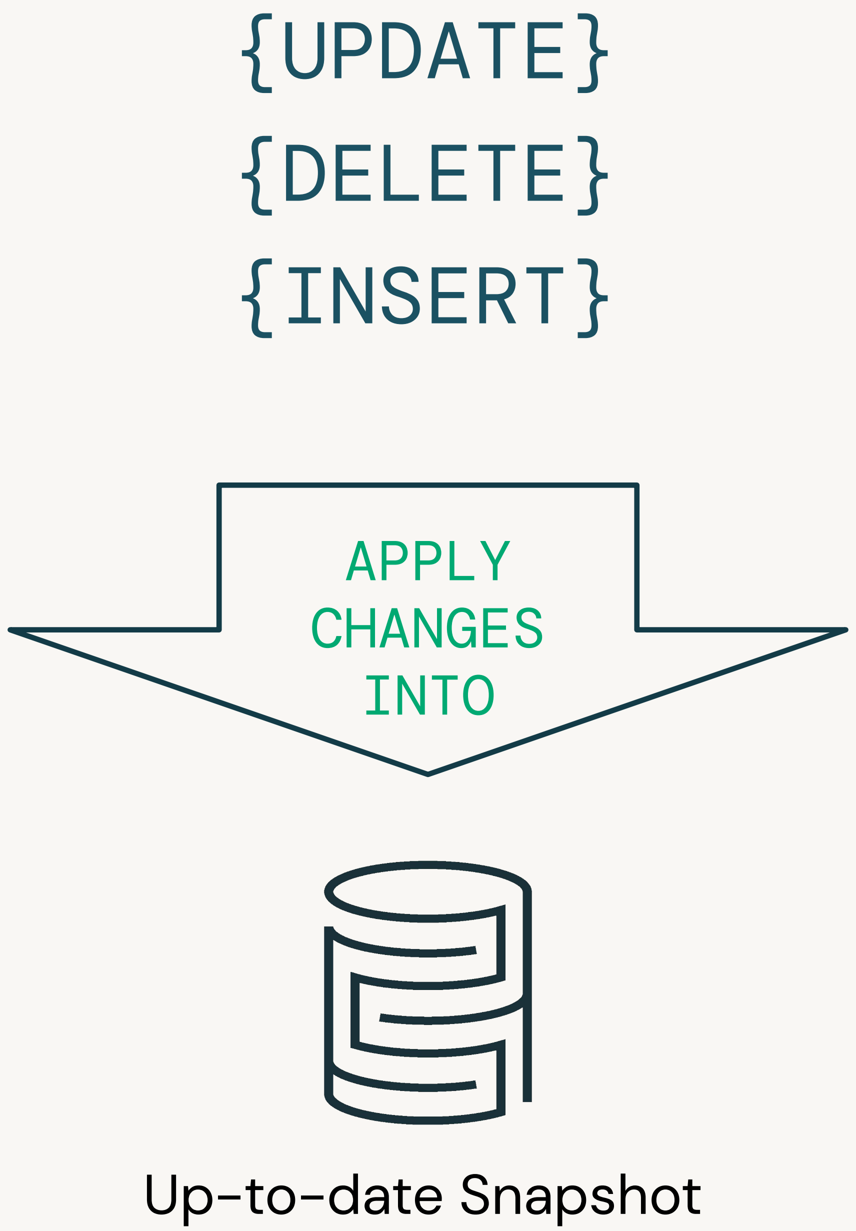
```
@dlt.table
def data():
  input_path = spark.conf.get("my_etl.input_path”)
  spark.readStream.format("cloud_files”).load(input_path)
```

How can I do change data capture (CDC)?

APPLY CHANGES INTO for CDC

Maintain an up-to-date replica of a table stored elsewhere

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(LIVE.city_updates)
KEYS (id)
SEQUENCE BY ts
```



APPLY CHANGES INTO for CDC

Maintain an up-to-date replica of a table stored elsewhere

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(LIVE.city_updates)
KEYS (id)
SEQUENCE BY ts
```

A source of changes, currently this has to be a stream.

city_updates

```
{"id": 1, "ts": 1, "city": "Bekerly, CA"}
```

APPLY CHANGES INTO for CDC

Maintain an up-to-date replica of a table stored elsewhere

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(LIVE.city_updates)
KEYS (id)
SEQUENCE BY ts
```

A target for the changes to be applied to.

city_updates

{"id": 1, "ts": 1, "city": "Bekerly, CA"}

cities

id	city

APPLY CHANGES INTO for CDC

Maintain an up-to-date replica of a table stored elsewhere

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(LIVE.city_updates)
KEYS (id)
SEQUENCE BY ts
```

A unique key that can be used to identify a given row.

city_updates

```
{"id": 1, "ts": 1, "city": "Bekerly, CA"}
```

cities

id	city

APPLY CHANGES INTO for CDC

Maintain an up-to-date replica of a table stored elsewhere

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(LIVE.city_updates)
KEYS (id)
SEQUENCE BY ts
```

A sequence that can be used to order changes:

- Log sequence number (lsn)
- Timestamp
- Ingestion time

city_updates

{"id": 1, "ts": 100, "city": "Bekerly, CA"}

cities

id	city

APPLY CHANGES INTO for CDC

Maintain an up-to-date replica of a table stored elsewhere

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(LIVE.city_updates)
KEYS (id)
SEQUENCE BY ts
```

city_updates

```
{"id": 1, "ts": 100, "city": "Bekerly, CA"}
{"id": 1, "ts": 200, "city": "Berkeley, CA"}
```

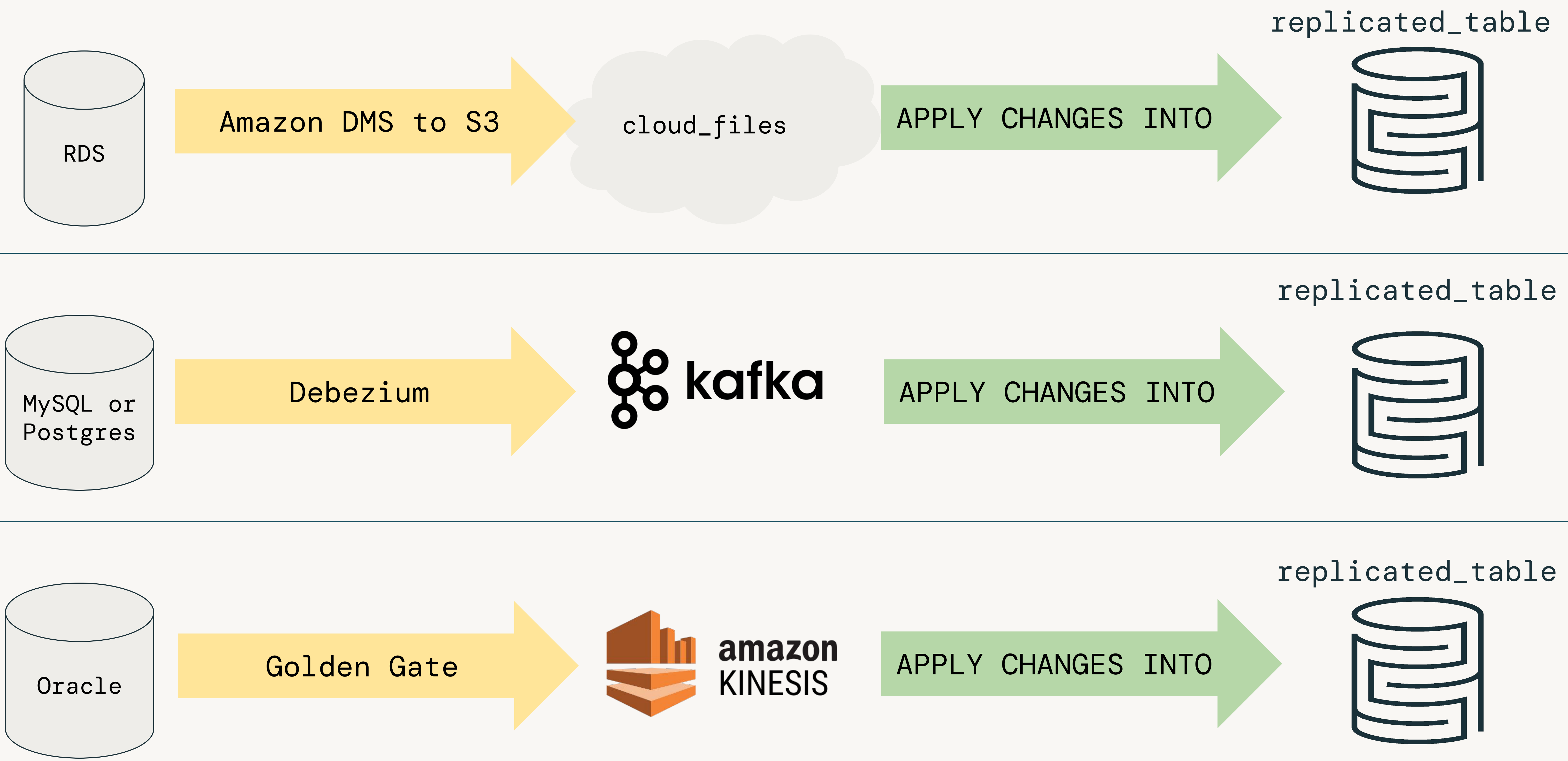
cities

id	city	
1	~~Bekerly, CA~~	Berkeley, CA

REFERENCE ARCHITECTURE

Change Data Capture (CDC) from RDBMS

A variety of 3rd party tools can provide a streaming change feed



What do I *no longer* need to manage with DLT?

Automated Data Management

DLT automatically optimizes data for performance & ease-of-use

Best Practices

What:

DLT encodes Delta best practices automatically when creating DLT tables.

How:

DLT sets the following properties:

- `optimizeWrite`
- `autoCompact`
- `tuneFileSizesForRewrites`

Physical Data

What:

DLT automatically manages your physical data to minimize cost and optimize performance.

How:

- runs vacuum daily
- runs optimize daily

You still can tell us how you want it organized (ie ZORDER)

Schema Evolution

What:

Schema evolution is handled for you

How:

Modifying a live table transformation to add/remove/rename a column will automatically do the right thing.

When removing a column in a streaming live table, old values are preserved.