Delta Live Tables in Depth

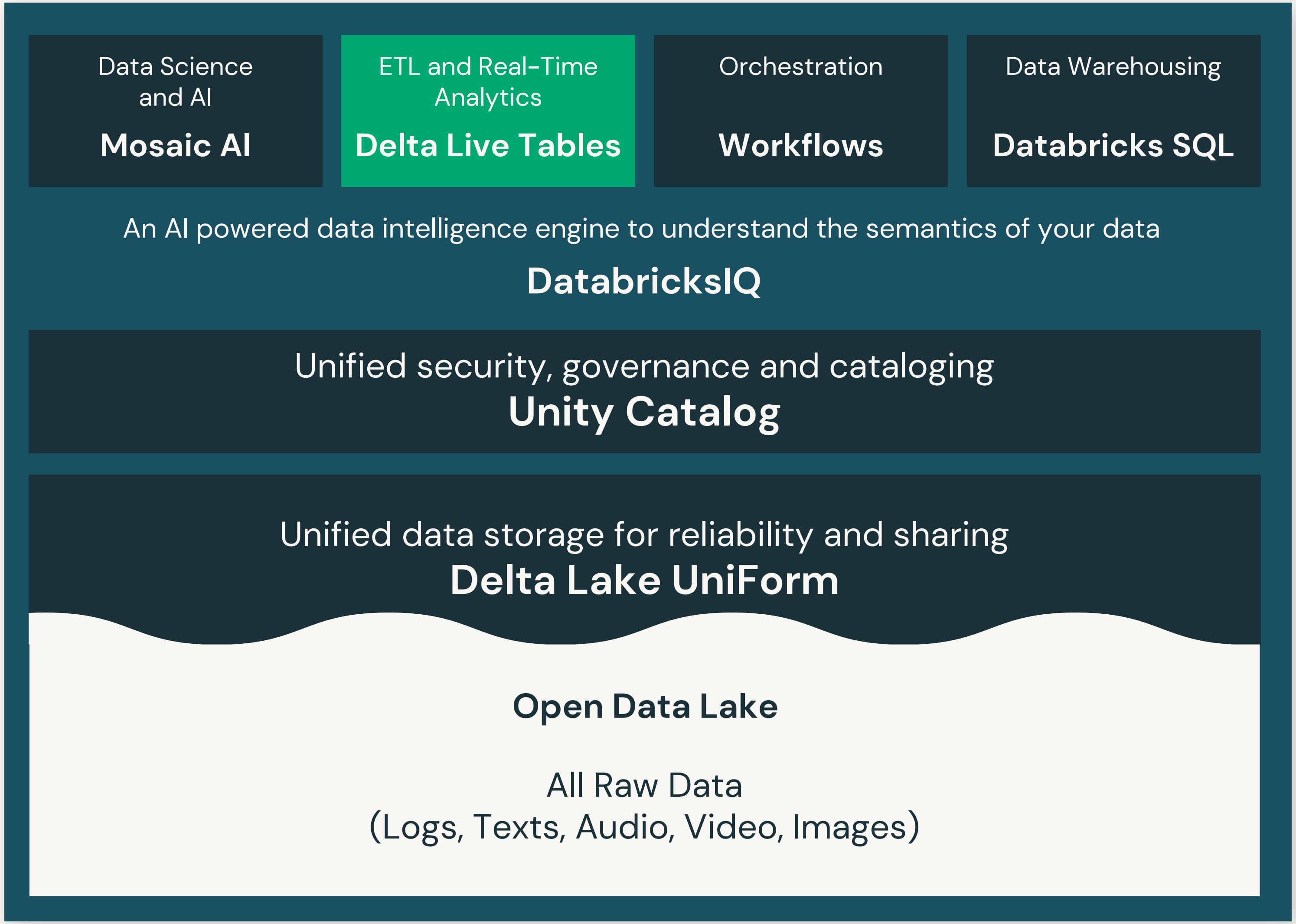
Data & AI Summit, June 2024

Michael Armbrust, Distinguished Engineer
Paul Lappas, Sr Staff Product Manager

Product safe harbor statement

This information is provided to outline Databricks' general product direction and is for **informational purposes only**. Customers who purchase Databricks services should make their purchase decisions relying solely upon services, features, and functions that are currently available. Unreleased features or functionality described in forward-looking statements are subject to change at Databricks discretion and may not be delivered as planned or at all

Reliable data pipelines require a unified platform with data intelligence



AI initiatives are top of mind…

By 2026, **over 80%** of enterprises will be using GenAI in production environments, up from **less than 5%** in 2023

—2023 Gartner Hype Cycle for Generative AI

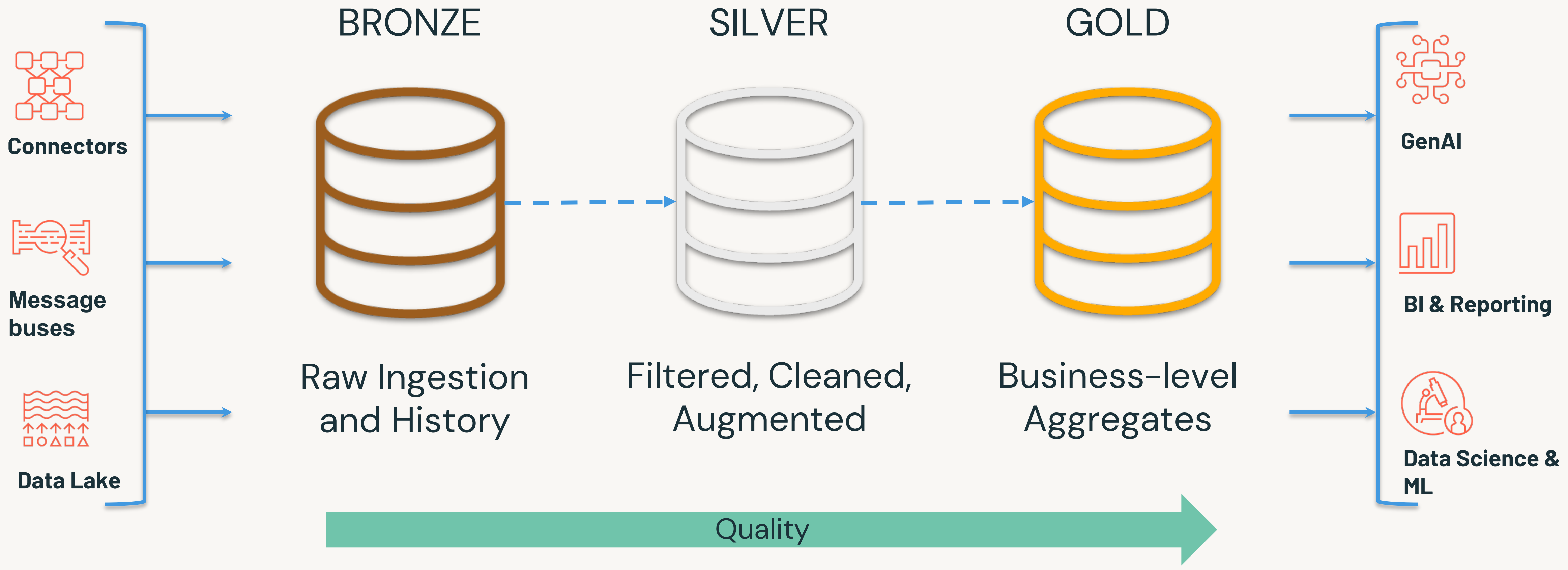
…but good models can't overcome bad data

"Data problems are **the most likely factor** to jeopardize our AI/ML goals"

—MIT Technology Review Insights survey, 2022

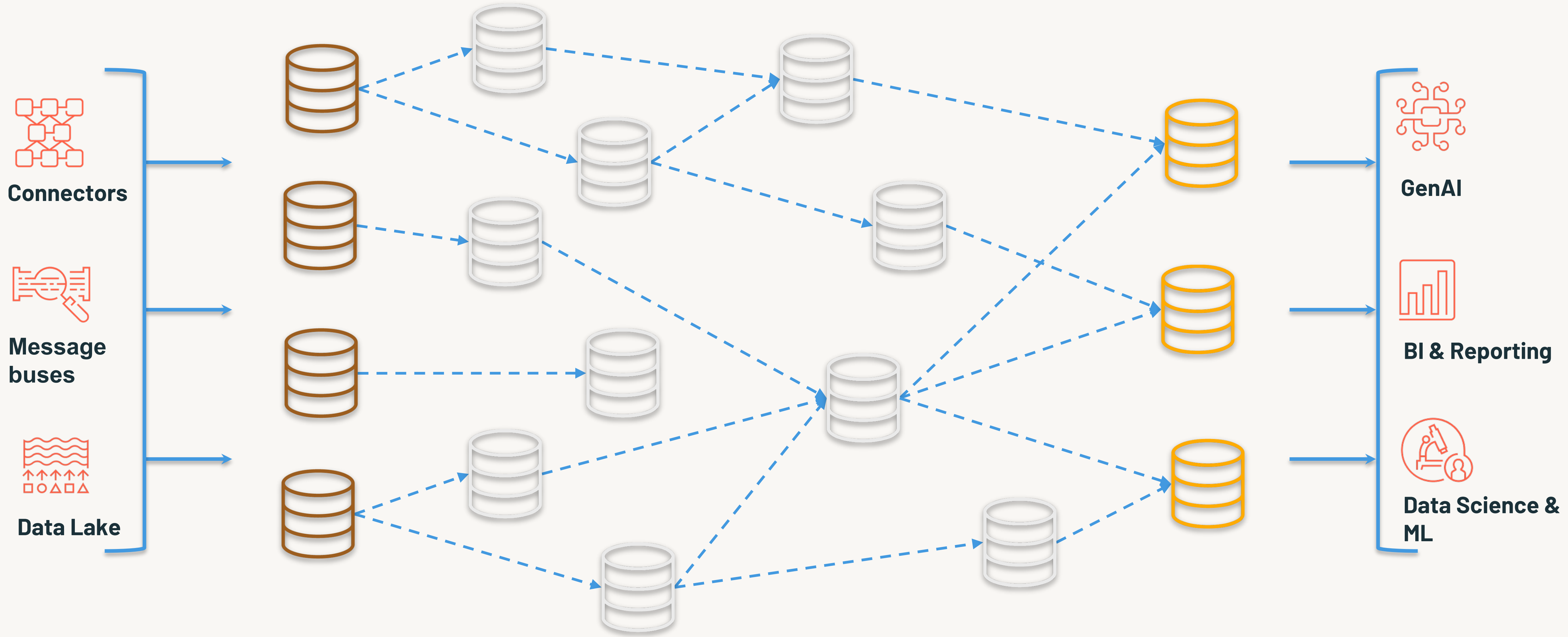
Good data is the foundation of a Lakehouse

All organizations need clean, fresh and reliable data.



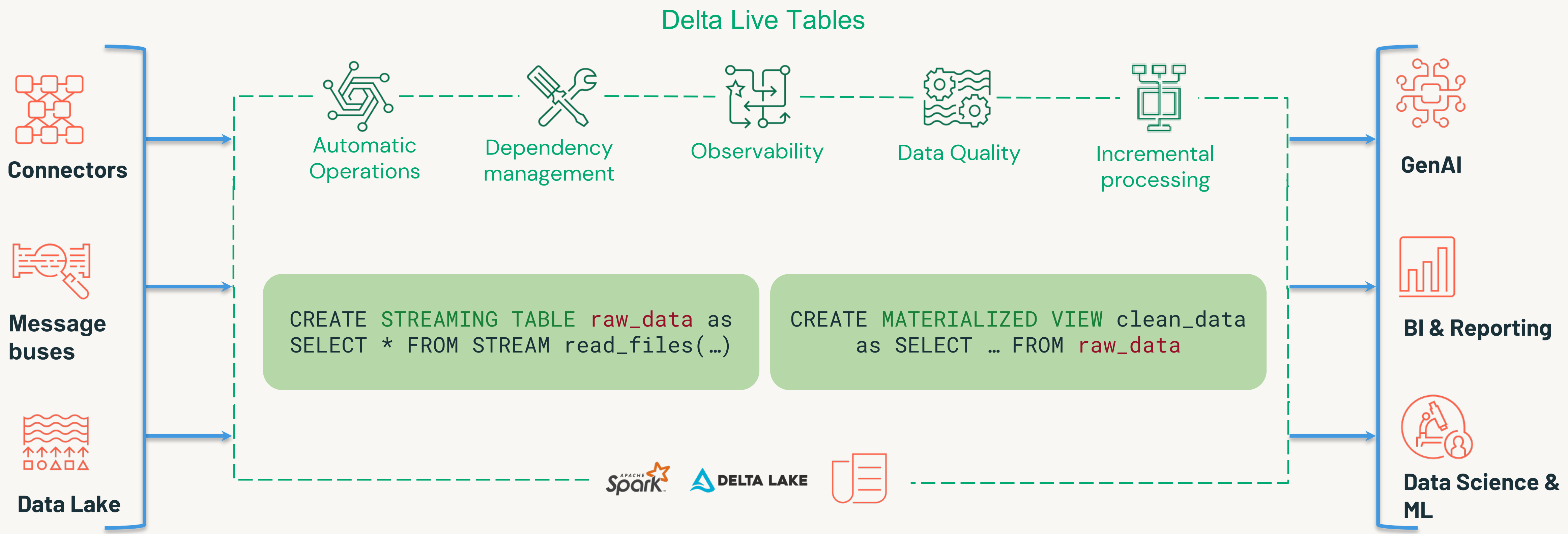
But the reality is not so simple

Data quality and reliability at scale is often complex and brittle



Delta Live Tables

From queries to production pipelines



What is declarative programming?

Imperative

Assumes the system is dumb.

```
total = 0
for i in range(1, 6)
    total += i
print(total)
```

Example: Step-by-step instructions

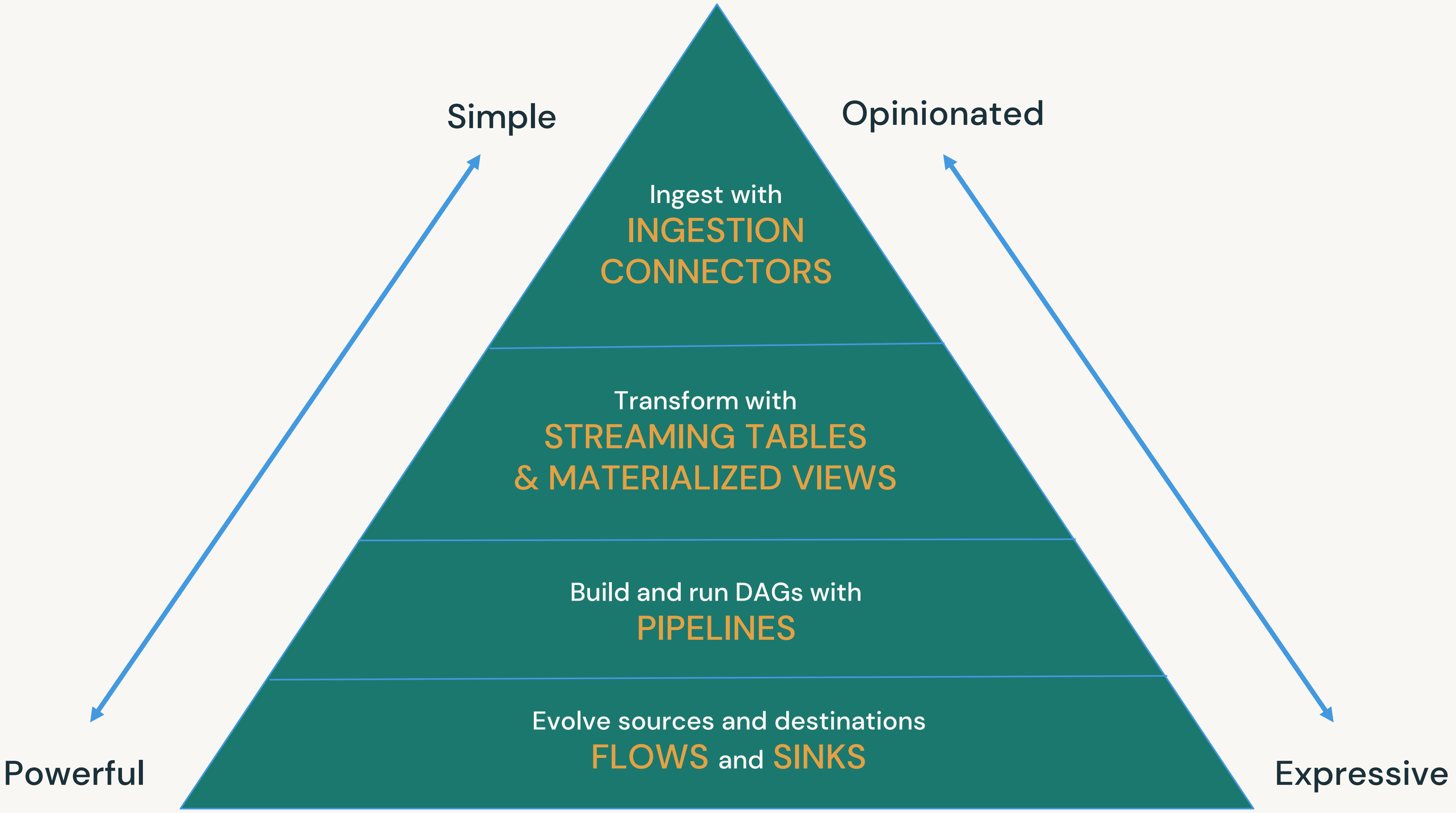
Declarative

Assumes the SYSTEM is smart.
(And you have better things to do!)

```
SELECT
    sum(i)
from
    <source>
```

Example: Simply describe the desired outcome

DLT powers declarative data engineering on the lakehouse



Streaming Table

```
CREATE STREAMING TABLE web_clicks
AS
SELECT *
FROM STREAM
  read_files('s3://mybucket')
```

```
CREATE STREAMING TABLE server_logs
AS
SELECT from_json(...) data
FROM STREAM
  read_kafka(...)
```

Data stream

Cloud Storage
(S3, ADLS, GCS)

Message Queues
(Kafka, Pub/Sub, Kinesis, etc)

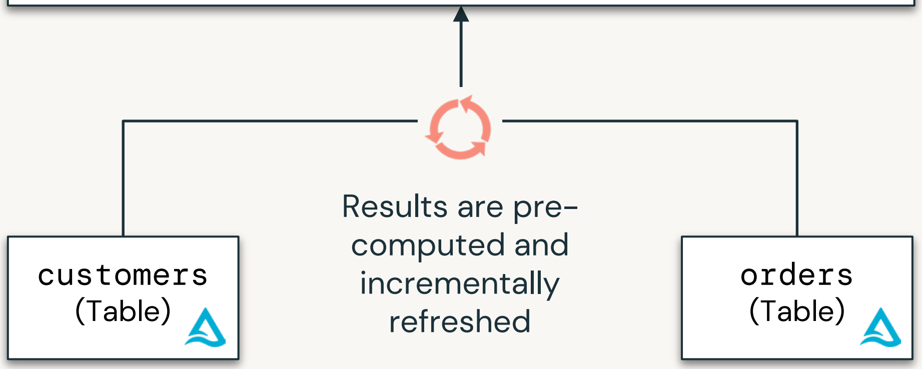
A simple way to stream and perform low-latency transformations on data.

Benefits:

1. Enable more practitioners. Simple SQL syntax makes data streaming accessible to all data engineers and analysts.
2. Better scalability. More efficiently handle high volumes of data via incremental processing vs large batches.
3. Unlock real-time use cases. Ability to support real-time analytics/BI, machine learning and operational use cases with streaming data.

Materialized View

```
CREATE MATERIALIZED VIEW customer_orders
AS
SELECT
  customers.name,
  sum(orders.amount),
  orders.orderdate
FROM orders
  LEFT JOIN customers ON
    orders.custkey = customers.c_custkey
GROUP BY
  name,
  orderdate;
```



Perform complex transformations for ETL and accelerate end-user queries for dashboards/BI.

Benefits:

1. Accelerate queries / dashboards. Much faster to query data that is pre-computed vs querying base tables.
2. Improve data freshness. MVs can be incrementally refreshed when new data arrives, avoiding time-consuming full recomputes
3. Simple ETL. Transform and process data in a declarative way.

Views in DLT

Modularizing your code

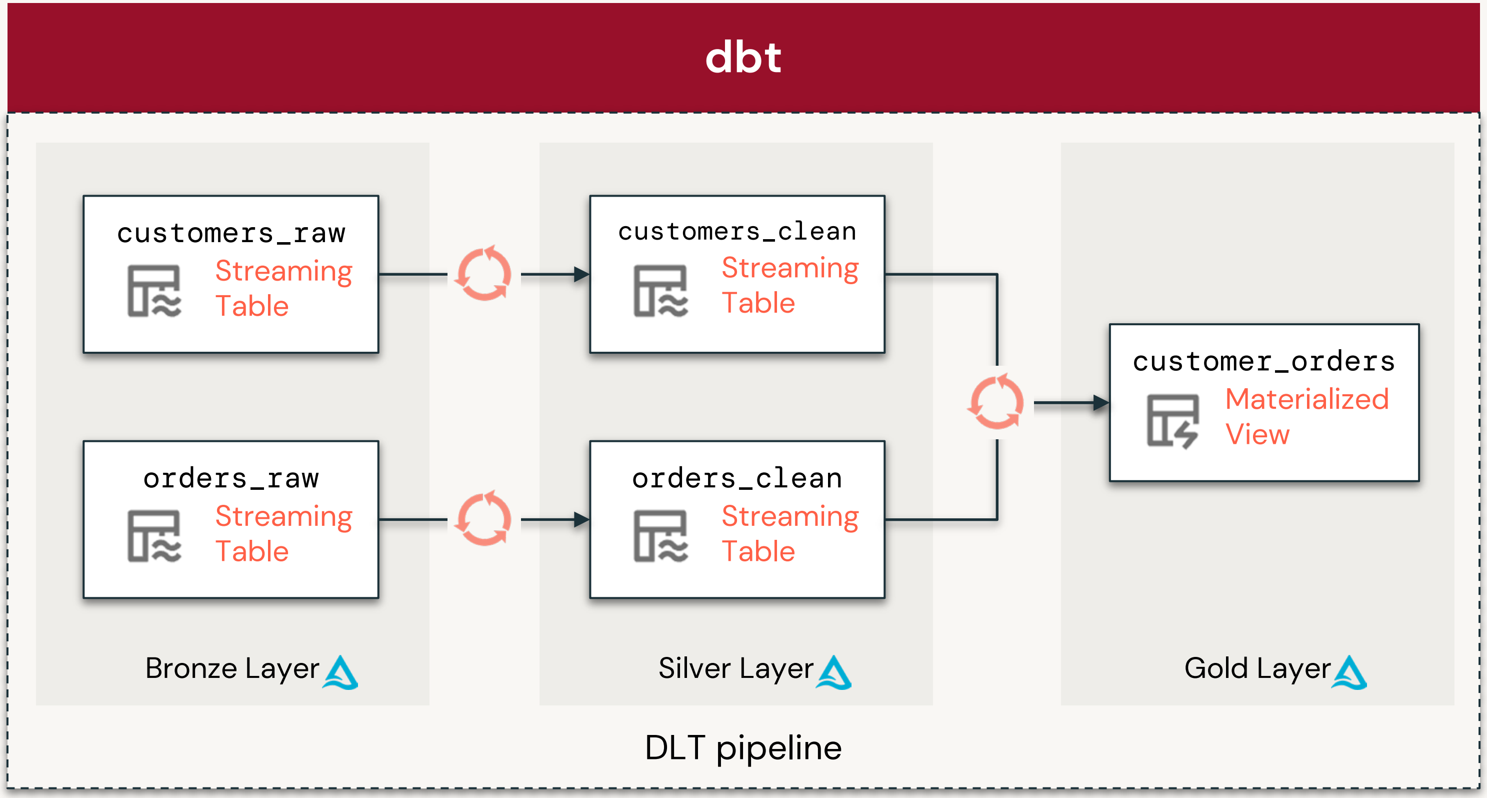
Views are, a name that is substituted with a query

- Use views to break up large/complex queries
- Expectations on views can validate correctness of intermediate results
- Views are recomputed every time they are queried

When multiple tables need the same result, consider a streaming table or materialized view instead.

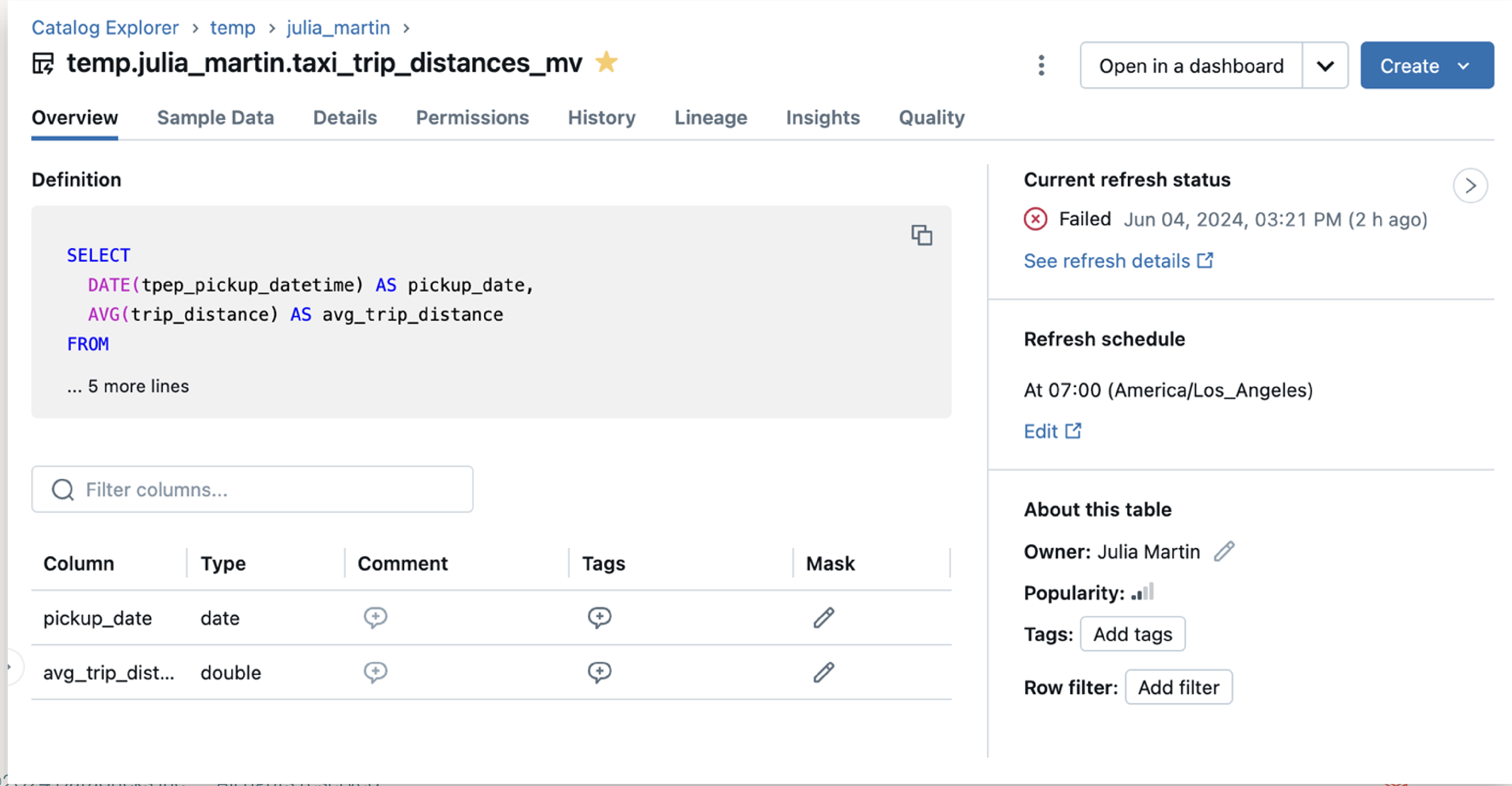
dbt Pipelines using MV’s and ST’s in DBSQL

Native streaming ingestion and automatic incremental refresh of models

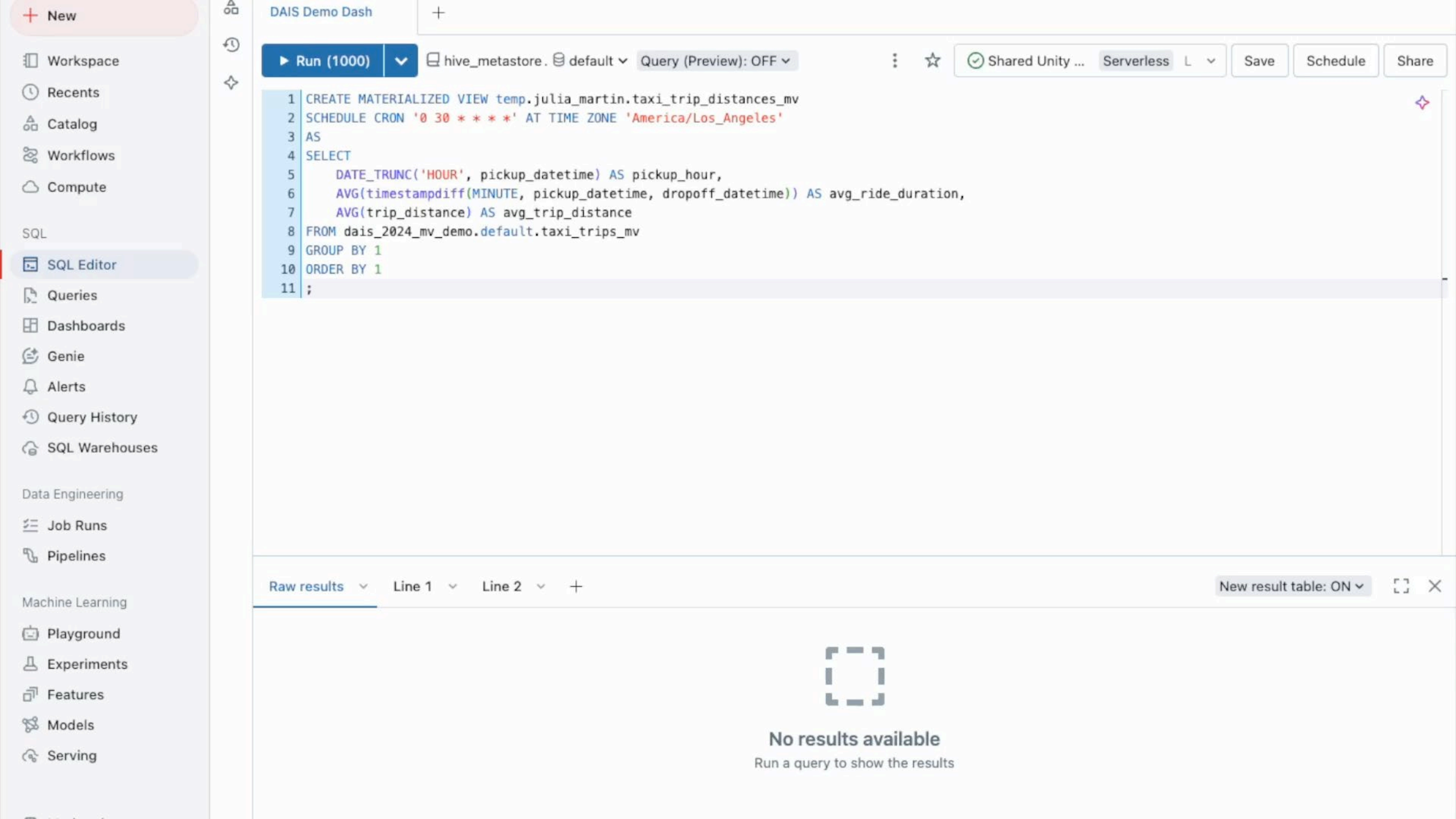


dbt-core v1.8 includes full support for Databricks’ streaming tables and materialized views

New MV/ST observability features

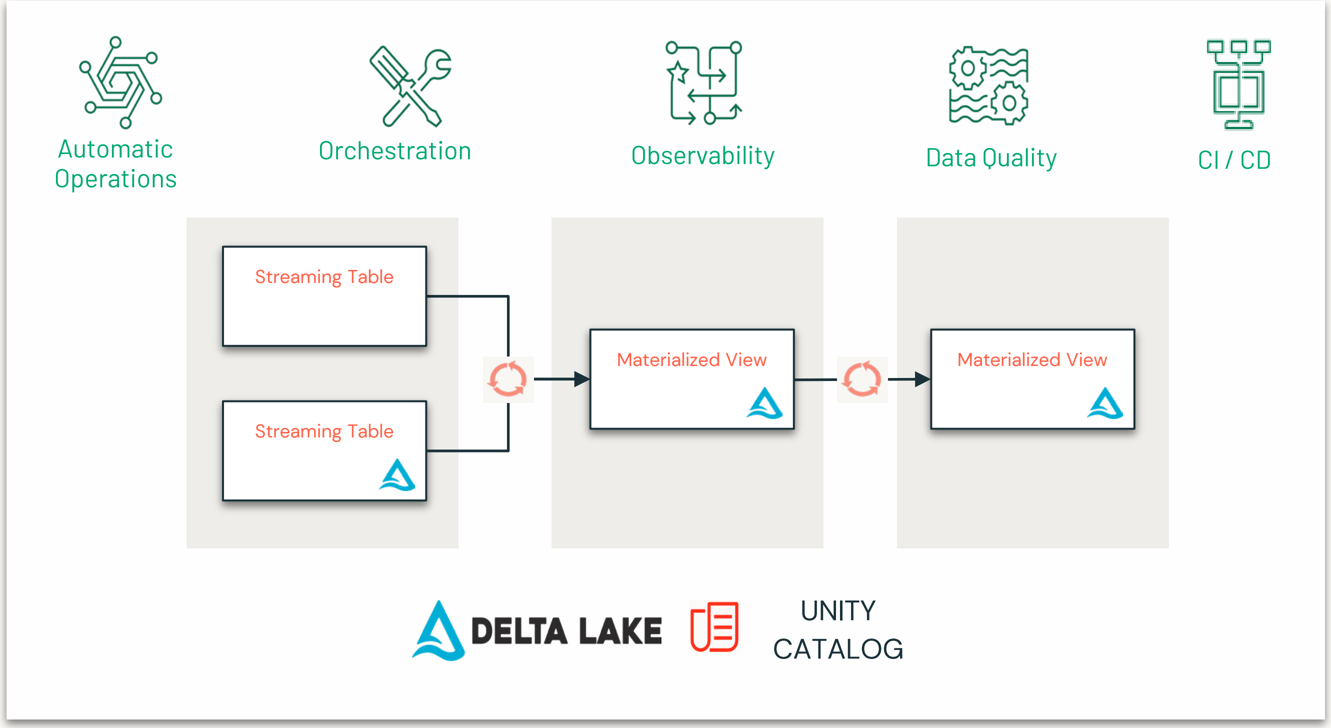


Demo: Materialized Views Observability


New
Workspace
Recents
Catalog
Workflows
Compute
SQL
SQL Editor
Queries
Dashboards
Genie
Alerts
Query History
SQL Warehouses
Data Engineering
Job Runs
Pipelines
Machine Learning
Playground
Experiments
Features
Models
Serving
DAIS Demo Dash
Run (1000)
hive_metastore . default
Query (Preview): OFF
Shared Unity ...
Serverless
L
Save
Schedule
Share
CREATE MATERIALIZED VIEW temp.julia_martin.taxi_trip_distances_mv
SCHEDULE CRON '0 30 * * * *' AT TIME ZONE 'America/Los_Angeles'
AS
SELECT
DATE_TRUNC('HOUR', pickup_datetime) AS pickup_hour,
AVG(timestampdiff(MINUTE, pickup_datetime, dropoff_datetime)) AS avg_ride_duration,
AVG(trip_distance) AS avg_trip_distance
FROM dais_2024_mv_demo.default.taxi_trips_mv
GROUP BY 1
ORDER BY 1
;
Raw results
Line 1
Line 2
New result table: ON
No results available
Run a query to show the results

Pipeline

Consists of source code, data location, and configuration



- Automated DAG resolution
- Environment isolation
- Automated recovery, upgrades, scaling and optimization
- Declarative APIs for data quality and CDC

Flow

Easiest way to do structured streaming

CDC, source evolutions, backfills, and initial hydration of streaming tables

Change data capture

Backfill data in streaming tables

Add and remove data sources without a full refresh

```
# APPLY CHANGES from different
streams
apply_changes(
  flow_name = "silver_data_main",
  target = "silver_data",
  source = "bronze_change_data",
  keys = ["id"],
  ignore_null_updates = True,
  stored_as_scd_type = "1",
  sequence_by = "seq",
  apply_as_deletes = "op = 'DELETE'"
)
```

```
CREATE STREAMING TABLE raw_data


CREATE FLOW kafka_us_east
AS INSERT INTO LIVE.raw_data BY NAME
SELECT * FROM kafka(…)


CREATE FLOW kafka_us_west
AS INSERT INTO LIVE.raw_data BY NAME
SELECT * FROM kafka(…)
```

Sink

Write to any location

Sinks define a target for a FLOW to send data

Supports operational and reverse ETL use cases

Reuse connection information stored in UC

File sink, unmanaged Delta tables, ForEachBatch, Kafka, and custom sinks (Data Source v2, Python Data Source APIs)

Private Preview

```
create_sink(
  name = "my_kafka_sink",
  format = "kafka",
  options = {
   "bootstrapServer": "hostA",
  },
)
```

```
CREATE SINK real_time
FORMAT kafka
OPTIONS (...)
```

Data Pipelines Made Simple with DLT

Simple

- **Simple development**: Declarative programming for batch and streaming pipelines including ingestion, transformation, CDC/SCD and data quality expectations
- **Simple operations**: Serverless infrastructure for vertical/horizontal autoscaling, automated orchestration and fast startup & retries

Performant

- Rapid infrastructure scale-up
- Continuous mode for streaming
- Stream pipelining for fast ingestion and task parallelization
- Fast incremental transformation with Enzyme

Low TCO

- **Efficient data processing:** Incremental ingestion and transformation
- **Efficient billing:** Only pay for what you use

Delta Live Tables

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

CREATE MATERIALIZED VIEW clean_data
AS SELECT …
FROM raw_data
```

DLT with serverless compute

The simplest way to build data pipelines

Simplicity	• Simple development • Simple operations
Performance	• End-to-end incremental processing • Parallelized ingestion
Low TCO	• Serverless metering • Efficient data processing

DLT with serverless compute

The simplest way to build data pipelines

Simplicity
- Simple development
- Simple operations

Performance
- End-to-end incremental processing
- Parallelized ingestion

Low TCO
- Serverless metering
- Efficient data processing

Ensure correctness with expectations in DLT

Manage and monitor data quality in real-time

```
%sql
CREATE OR REPLACE LIVE TABLE orders_valid
(CONSTRAINT valid_timestamp EXPECT (timestamp > '2020-01-01'))
SELECT * FROM LIVE.orders_vw

%python
@dlt.expect("valid timestamp", "timestamp > '2012-01-01'")
```

Retain Invalid Records

Records violate expectations are
- Added to target table
- Flagged in invalid in event log
- Pipeline continues

```
%sql
CREATE OR REPLACE LIVE TABLE orders_valid
(CONSTRAINT valid_timestamp EXPECT (timestamp > '2020-01-01') ON VIOLATION DROP ROW)
SELECT * FROM LIVE.orders_vw

%python
@dlt.expect_or_drop("valid timestamp", "timestamp > '2012-01-01'")
```

Drop Invalid Records

Records violate expectation are
- Dropped from target table
- Flagged invalid in event log
- Pipeline Continues

```
%sql
CREATE OR REPLACE LIVE TABLE orders_valid
(CONSTRAINT valid_timestamp EXPECT (timestamp > '2020-01-01') ON VIOLATION FAIL)
SELECT * FROM LIVE.orders_vw

%python
@dlt.expect_or_fail("valid timestamp", "timestamp > '2012-01-01'")
```

Fail on Invalid Records

Records violate expectation
- Cause the job to fail

Data Quality

Records Processed
79,259,129

- **Written** 77.9% (61,752,707)
- **Dropped** 22.1% (17,506,422)

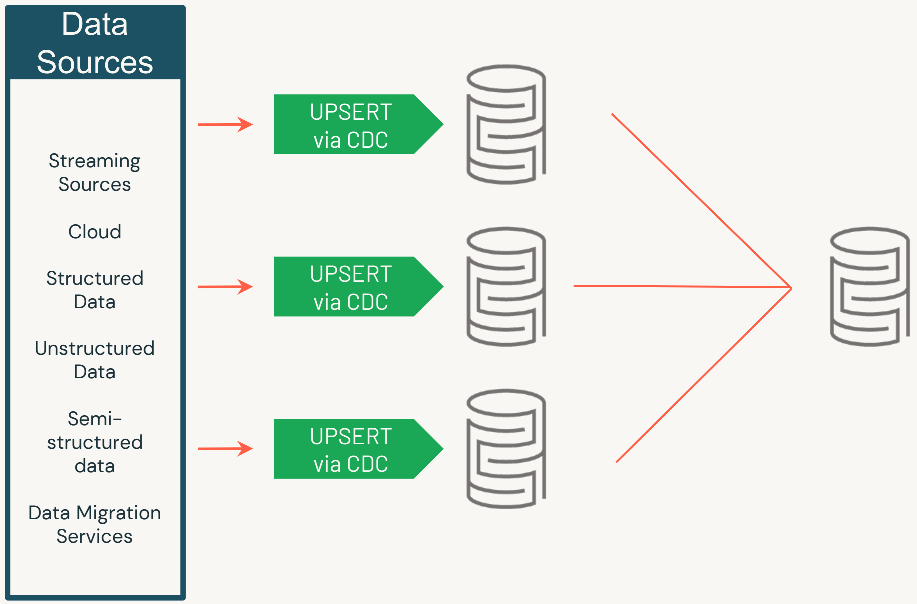
Expectations

Failures Only | All

Name	Action	Fail %	Failed Records
valid_trip_distance	DROP	22.1%	17484277
valid_passenger_count	DROP	0.2%	176524
valid_pickup_time	DROP	0.1%	60556
valid_dollar_amount	DROP	0%	1

Streaming CDC API

Process change records from a streaming change-data-feed




```sql
-- Create and populate the target table.
CREATE STREAMING TABLE target;

APPLY CHANGES INTO
  live.target
FROM
  stream(cdc_data.users)
KEYS
  (userId)
APPLY AS DELETE WHEN
  operation = "DELETE"
SEQUENCE BY
  sequenceNum
STORED AS
  SCD TYPE 2;
```

Simple, declarative API

Supports SCD1 or SCD2 storage formats

Python API support

Batch CDC API

Use DLT to process changes from full snapshots

Synchronize data from any source when you have access to full snapshots

Simple, declarative API

Supports SCD1 or SCD2 storage formats

Python API support

```
def apply_changes_from_snapshot(
  target,
  snapshot_and_version,
  keys,
  stored_as_scd_type,
  track_history_column_list = None,
  track_history_except_column_list =
None) -> None
```

Slowly Changing Dimensions Type 2

Keep a record of how values changed over time.

```
CREATE STREAMING TABLE cities

APPLY CHANGES INTO LIVE.cities
FROM STREAM(city_updates)
KEYS (id)
SEQUENCE BY ts
STORED AS SCD TYPE 2
```

SCD2 is supported for both batch and streaming CDC APIs

city_updates

```
{"id": 1, "ts": 01:00, "city": "Berkerly, CA"}
{"id": 1, "ts": 02:00, "city": "Berkeley, CA"}
```

cities

id	city	__starts_at	__ends_at
1	Bekerly, CA	01:00	02:00
1	Berkeley, CA	02:00	null

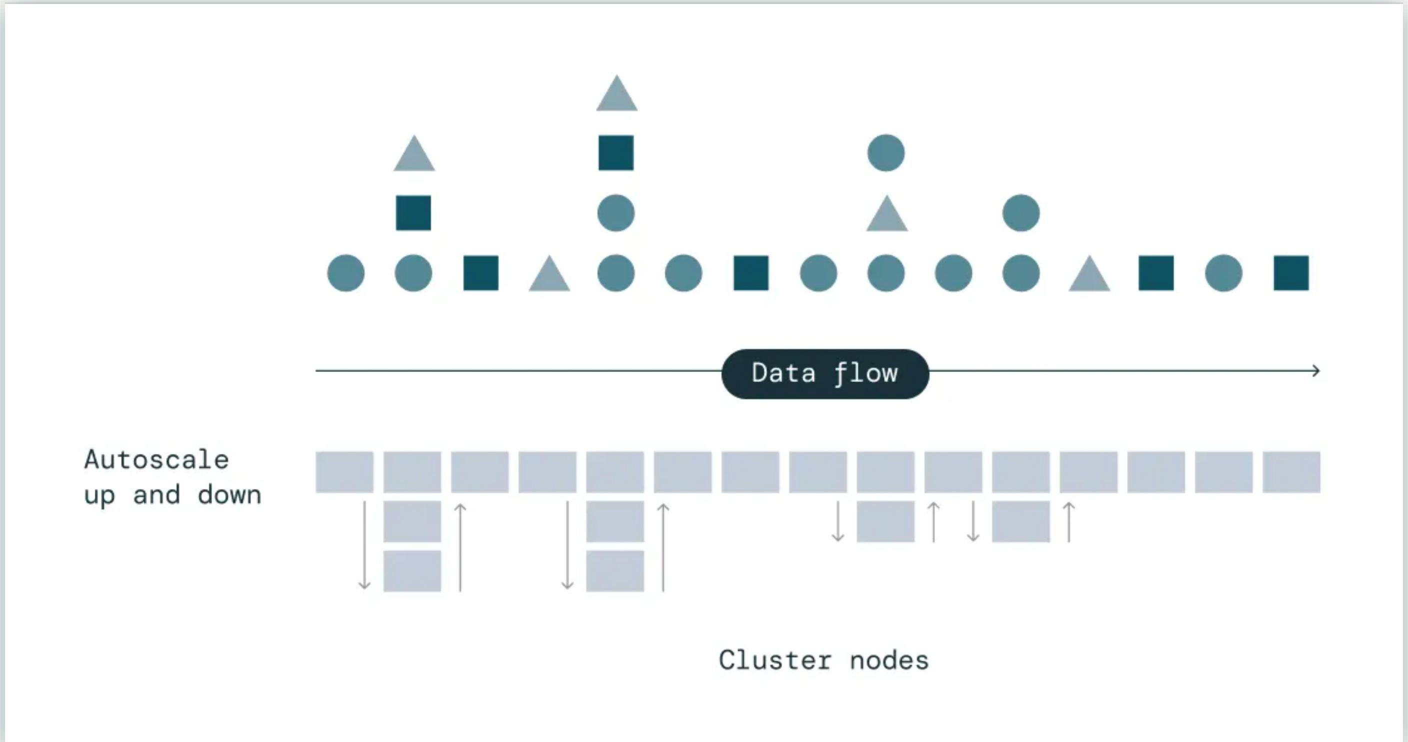
`__starts_at` and `__ends_at` will have the type of the `SEQUENCE BY` field (`ts`).

Horizontal Autoscaling

Automatically scale compute to handle high number of concurrent tasks

Enhanced autoscaling optimizes compute utilization while ensuring maximum concurrency

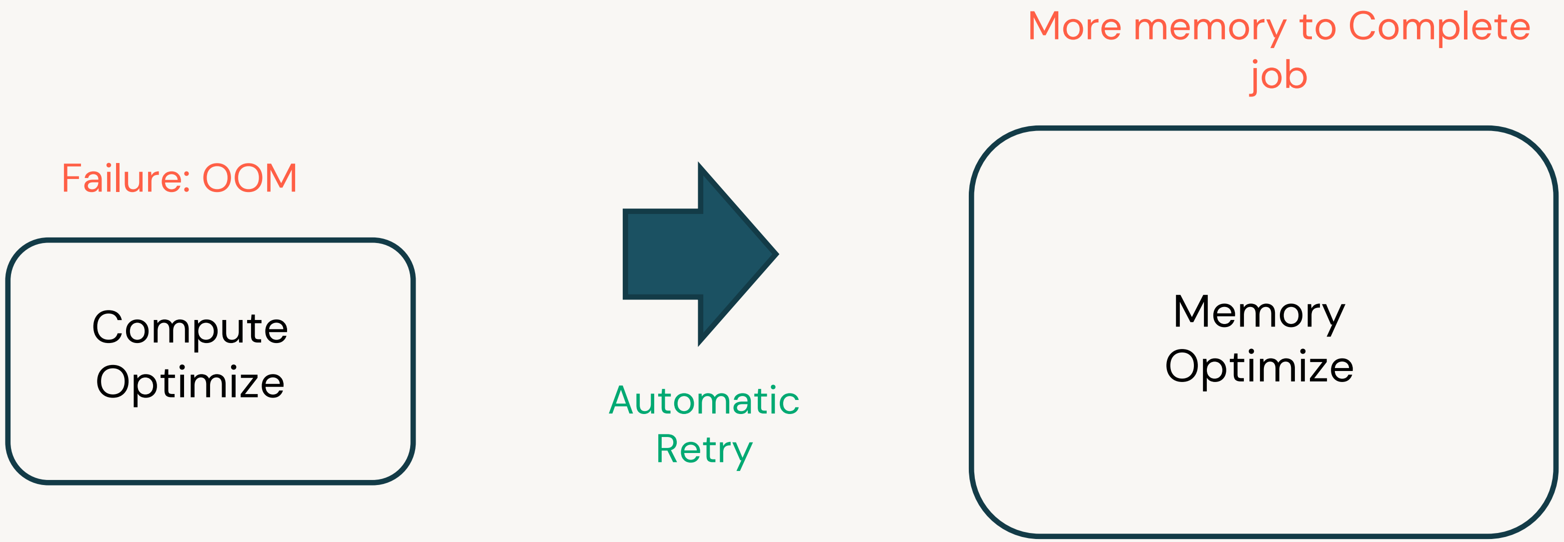
- Only scaling up to the necessary compute required
- Gracefully shuts down computed when utilization is low to avoid unnecessary spend



Vertical Autoscaling

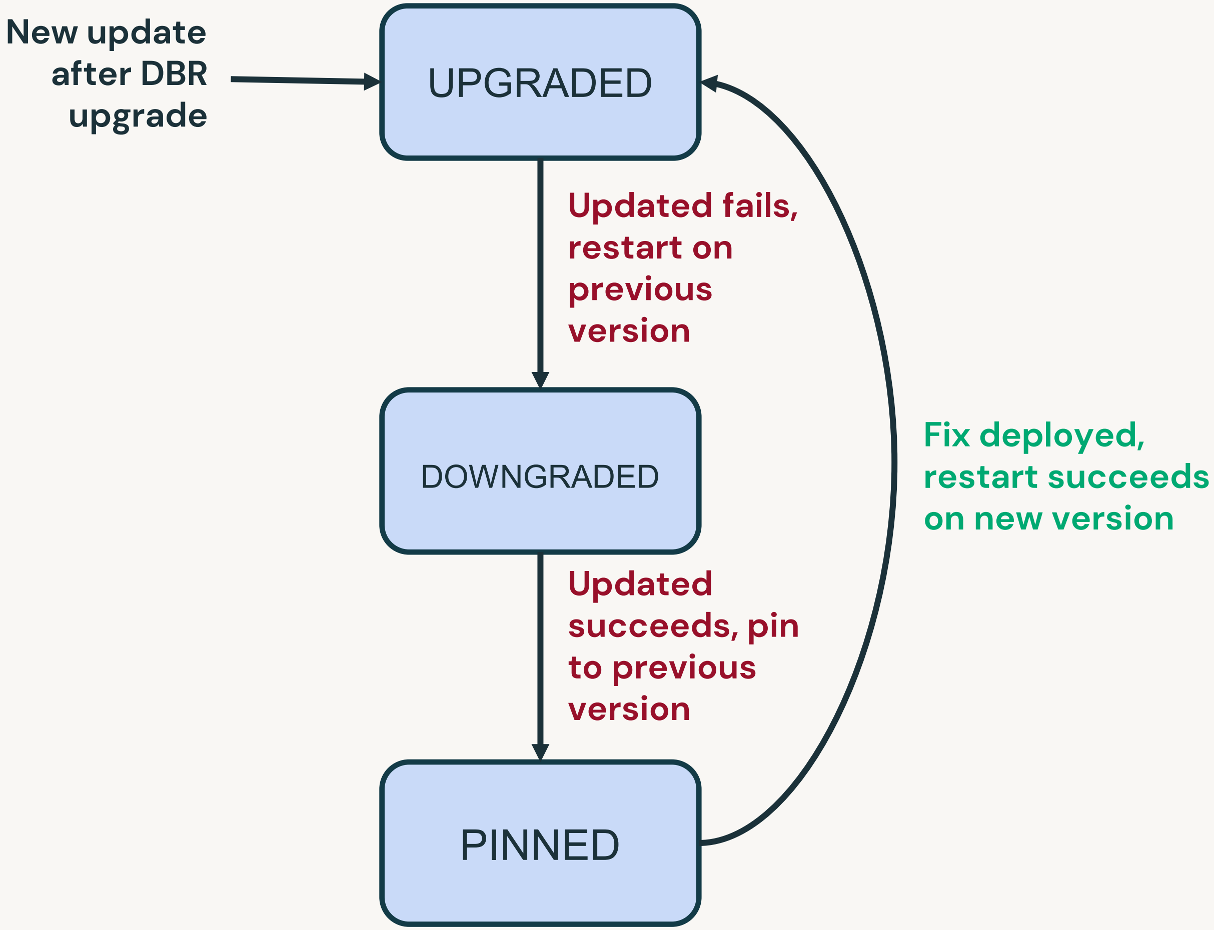
Automatically scale memory to handle complex workloads

- Horizontal helps, but may not be the most efficient with memory pressure.
- Automatic Vertical Scaling when Job runs into OOMs
- Scales down when larger instances are no longer needed



Automated service upgrades

Health mediated upgrade process maximizes uptime for production pipelines



Best practices for production pipelines:

- Use CURRENT channel and ‘production’ mode
- Configure restart notifications with DLT settings UI
- Continuously test production pipelines against the PREVIEW channel

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 CLUSTER BY)

Schema Evolution

What:

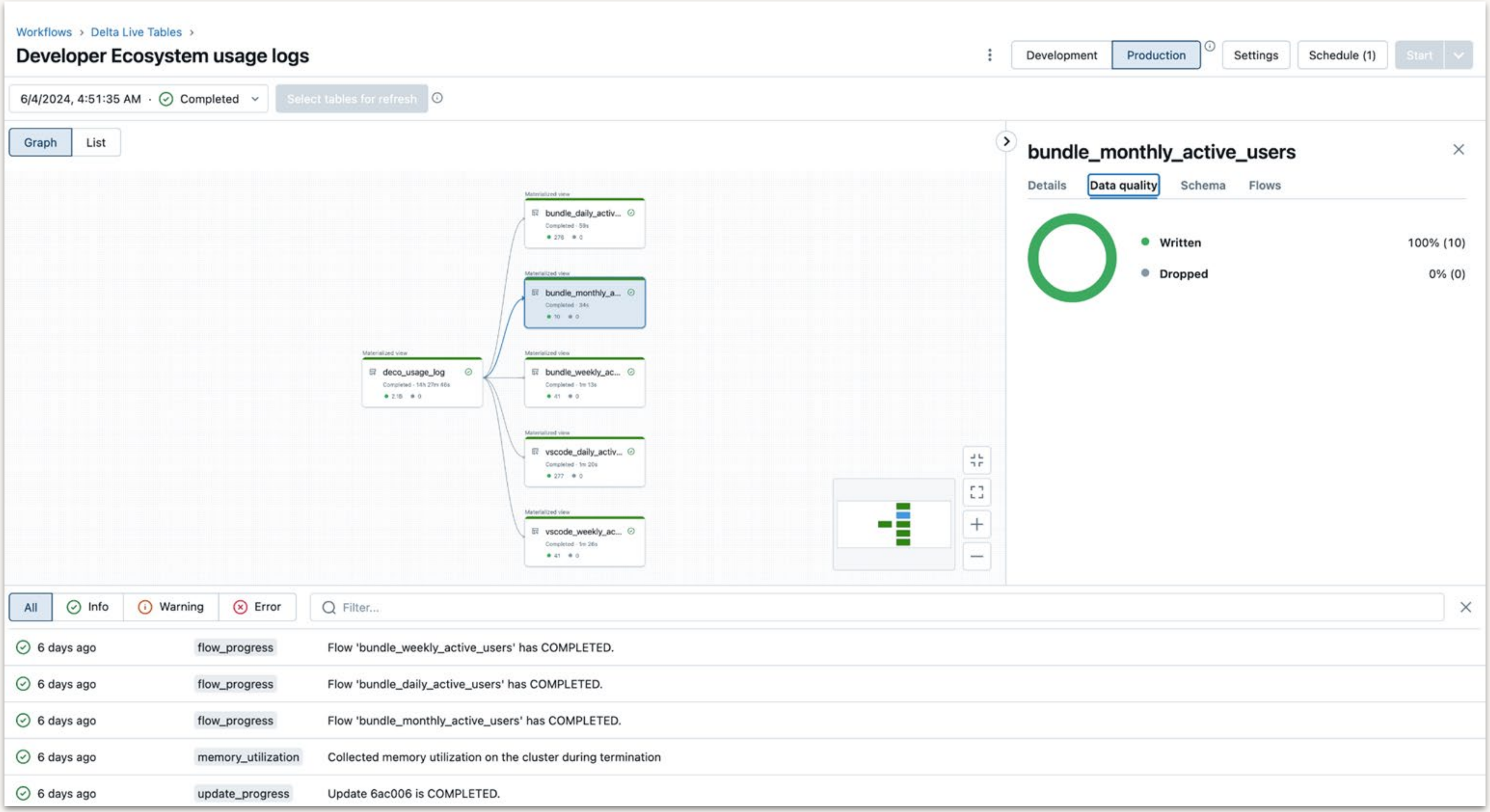
Schema evolution is handled for you

How:

Adding/removing/renaming a column in a materialized view will automatically do the right thing.

Old values are preserved with removing a column from a streaming table. Adding a column will add a new column with NULL values for old data.

DLT operational dashboard



The Event Log

Real-time log of pipeline operations

Telemetry

Time-series pipeline operations

Configurations and settings

Rows processed

Incremental refresh status

Lineage

Table schemas, definitions, and declared properties

Table-level lineage

Query plans used to update tables

Data Quality

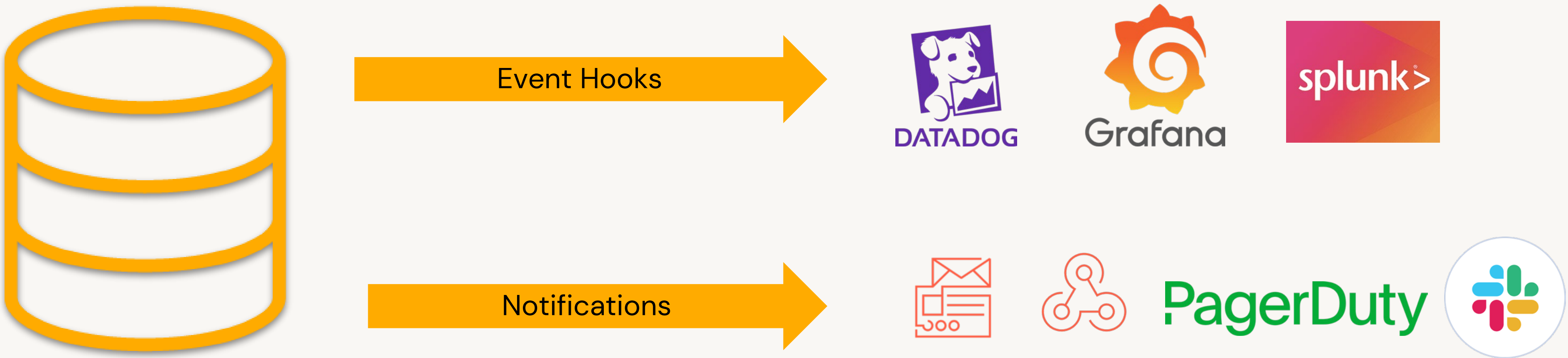
Expectation pass / failure / drop statistics

Input/Output rows that caused expectation failures

DLT Notifications and Monitoring

Get immediate notifications and ship the event log to your favorite tool

The event log is a table created for each pipeline



DLT Event Log

DLT with serverless compute

The simplest way to build data pipelines

Simplicity
- Simple development
- Simple operations

Performance
- End-to-end incremental processing
- Parallelized ingestion

Low TCO
- Serverless metering
- Efficient data processing

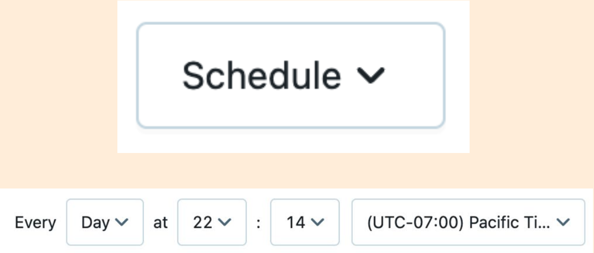
Scheduling pipelines

Controlling data freshness versus cost

TRIGGERED MODE

Costs: varies depending on schedule

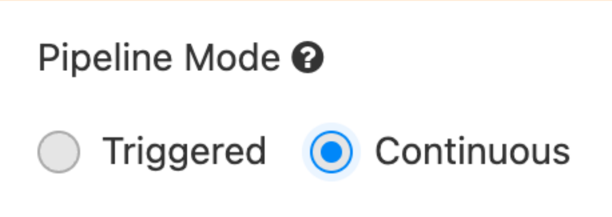
Latency: 10 minutes to months



CONTINUOUS MODE

Costs: highest

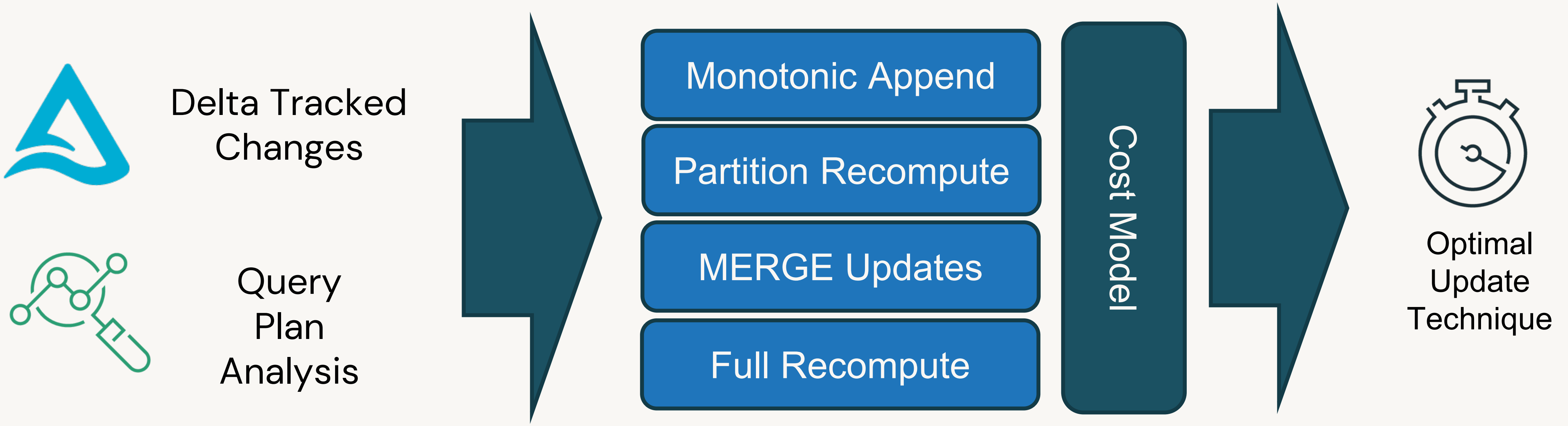
Latency: minutes to seconds



SERVERLESS EXCLUSIVE

Incremental Refresh for MVs

Cost based optimization powered by Enzyme



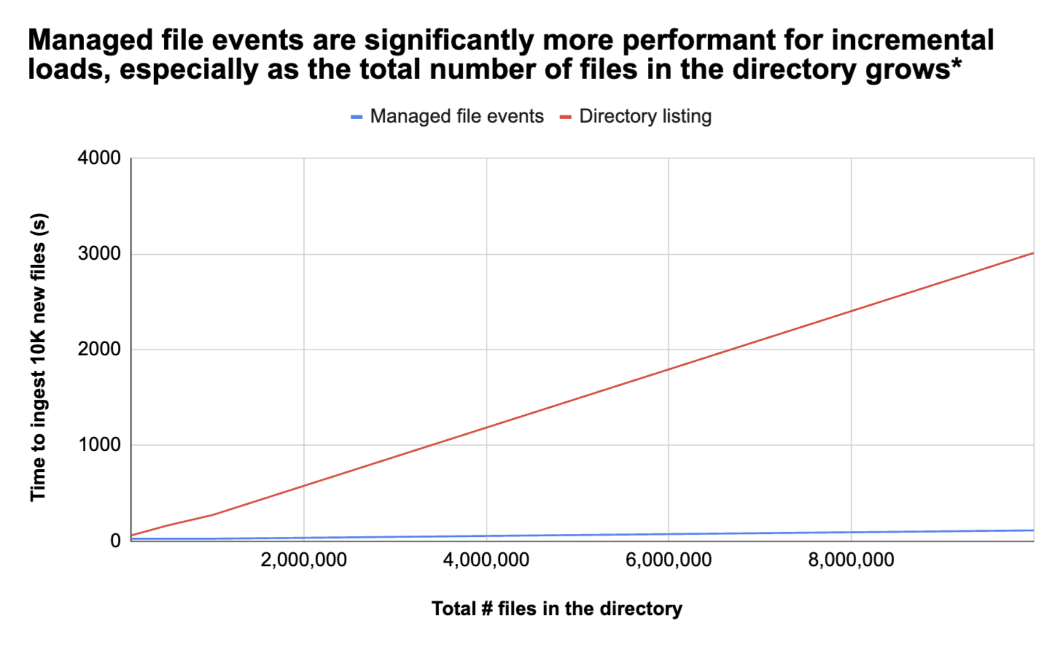
SERVERLESS EXCLUSIVE

Streaming tables with managed file events

Simple, high performance ingestion from external volumes

Simplifying file notifications

- **Single queue** for all streams
- **Lower risk** of hitting cloud notification limits
- **Simpler and faster** than directory listing
- Enabled for **serverless DLT or Jobs**

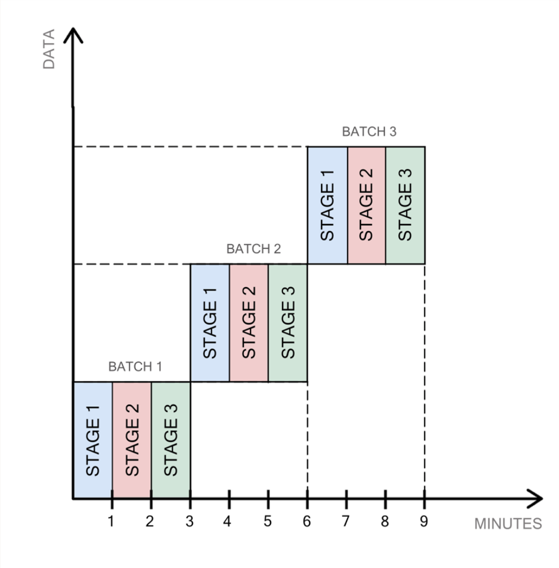


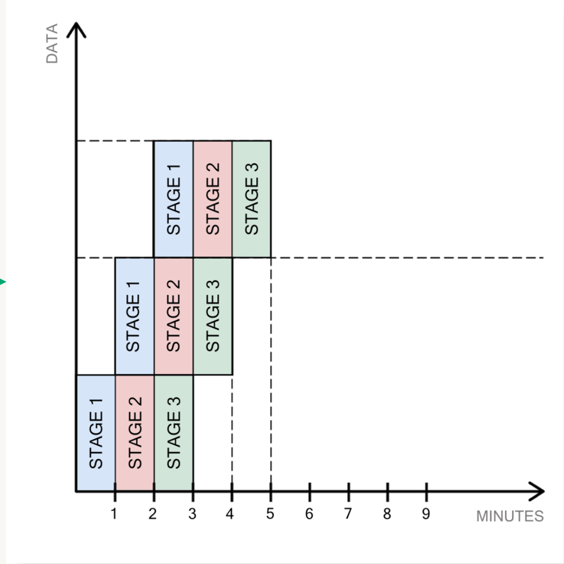
** Preliminary testing. Not a formal benchmark.*

```
autoLoaderStream = (spark.readStream
    .format("cloudFiles")
    ...
    .options("cloudFiles.useManagedFileEvents", True)
    ...)
```

Stream Pipelining

Concurrent batches allows for higher throughput and lower ingestion latency





Improve performance

Reduces latency

Works for both stateless and stateful streaming queries

DLT with serverless compute

The simplest way to build data pipelines

Simplicity	Simple development Simple operations
Performance	End-to-end incremental processing Parallelized ingestion
Low TCO	Serverless metering Efficient data processing

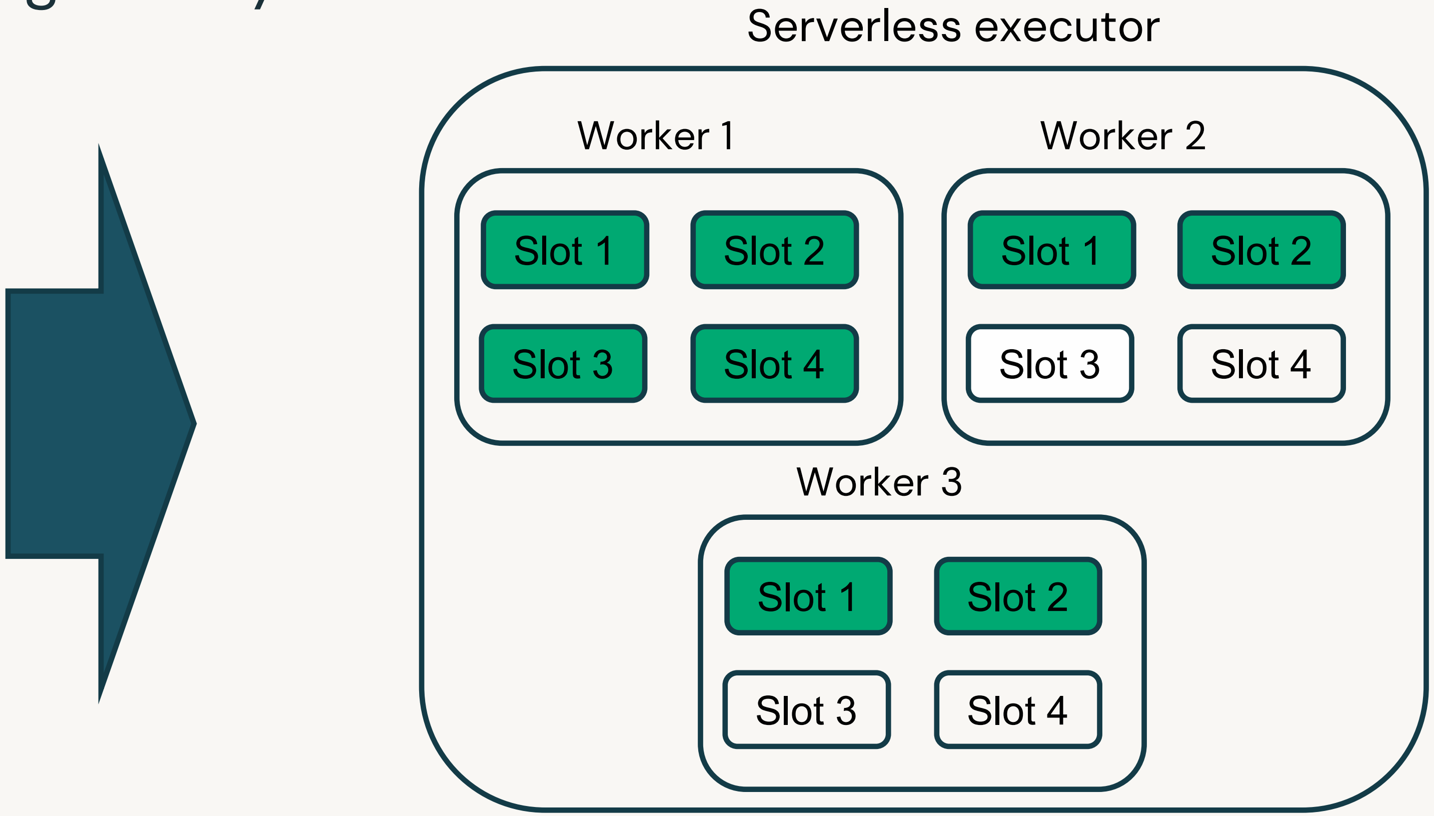
SERVERLESS EXCLUSIVE

Serverless metering

Consumption based pricing for any workload

In **Serverless** you only pay for utilized cores (slots).

DLT DBUs are the same price as Serverless Jobs and Serverless Interactive Notebooks.



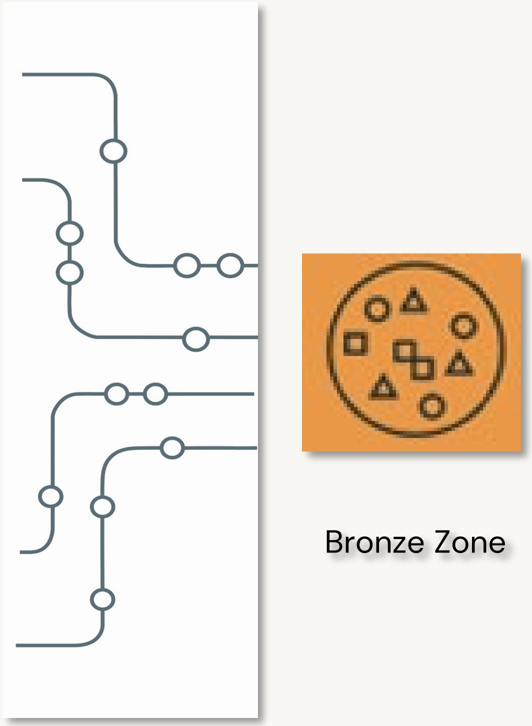
DLT Price-Performance

Streaming ingestion benchmark

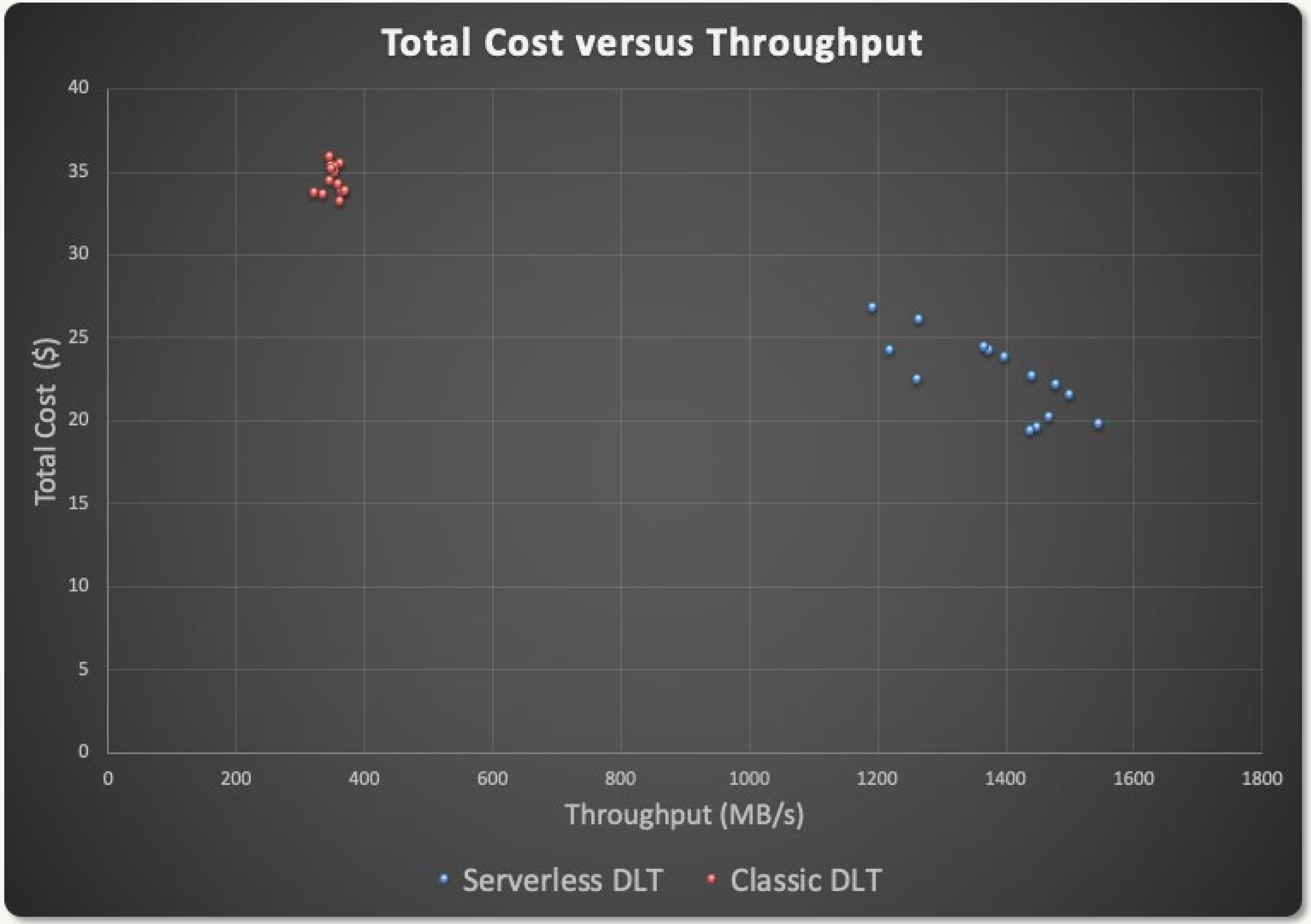
Measure price-performance of loading 100K JSON files into a streaming table

Classic compute configuration

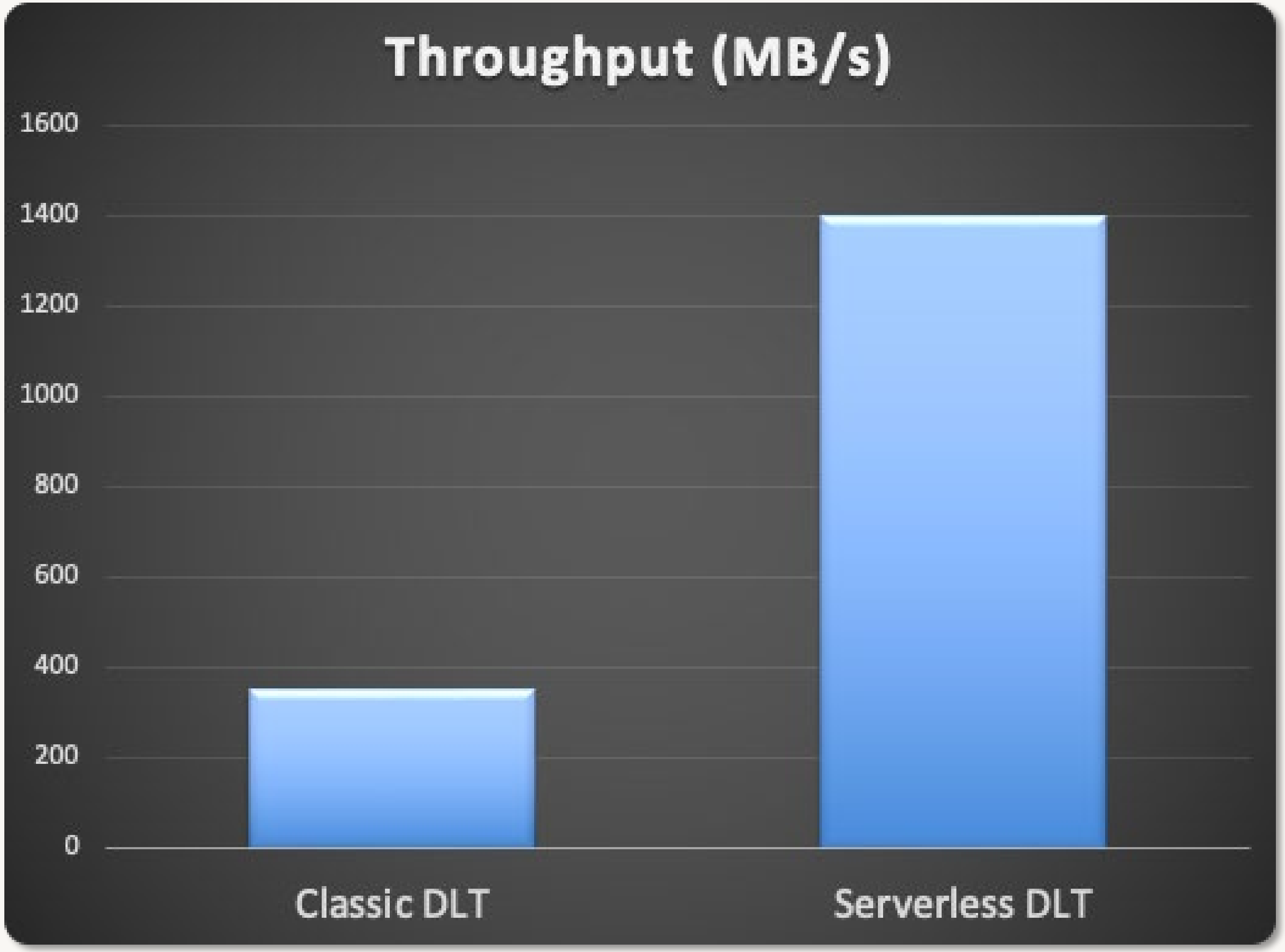
- Default instance type
- Enhanced autoscaling with max 64 workers
- Photon OFF



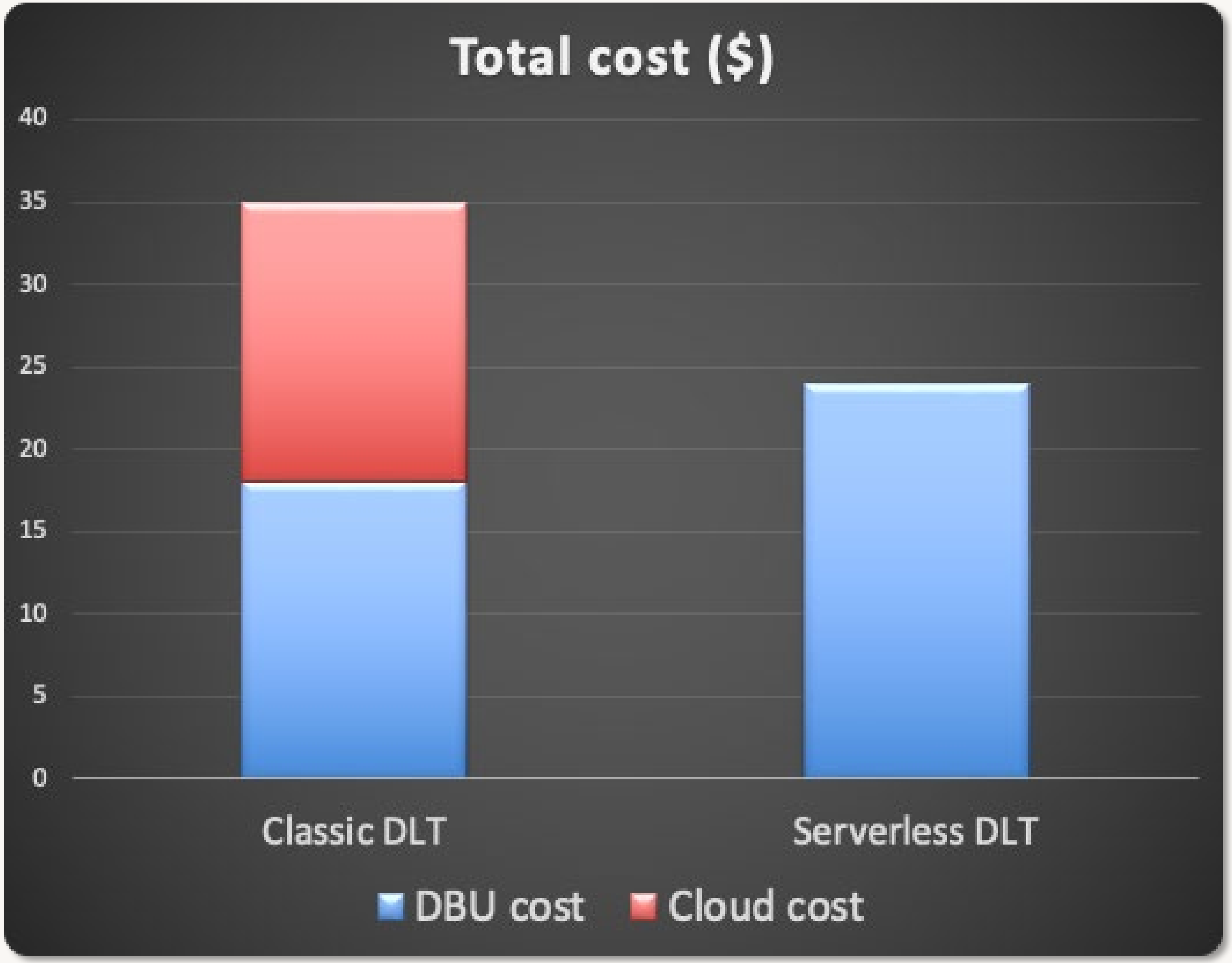
Raw experiment results



Serverless DLT provides 4x better throughput



With 32% less TCO



Overall 5x better price performance than DLT classic

MV refresh benchmark

Measure price performance of aggregating a 200B-row Delta table w/ 2B unique keys

We ran 4 total updates

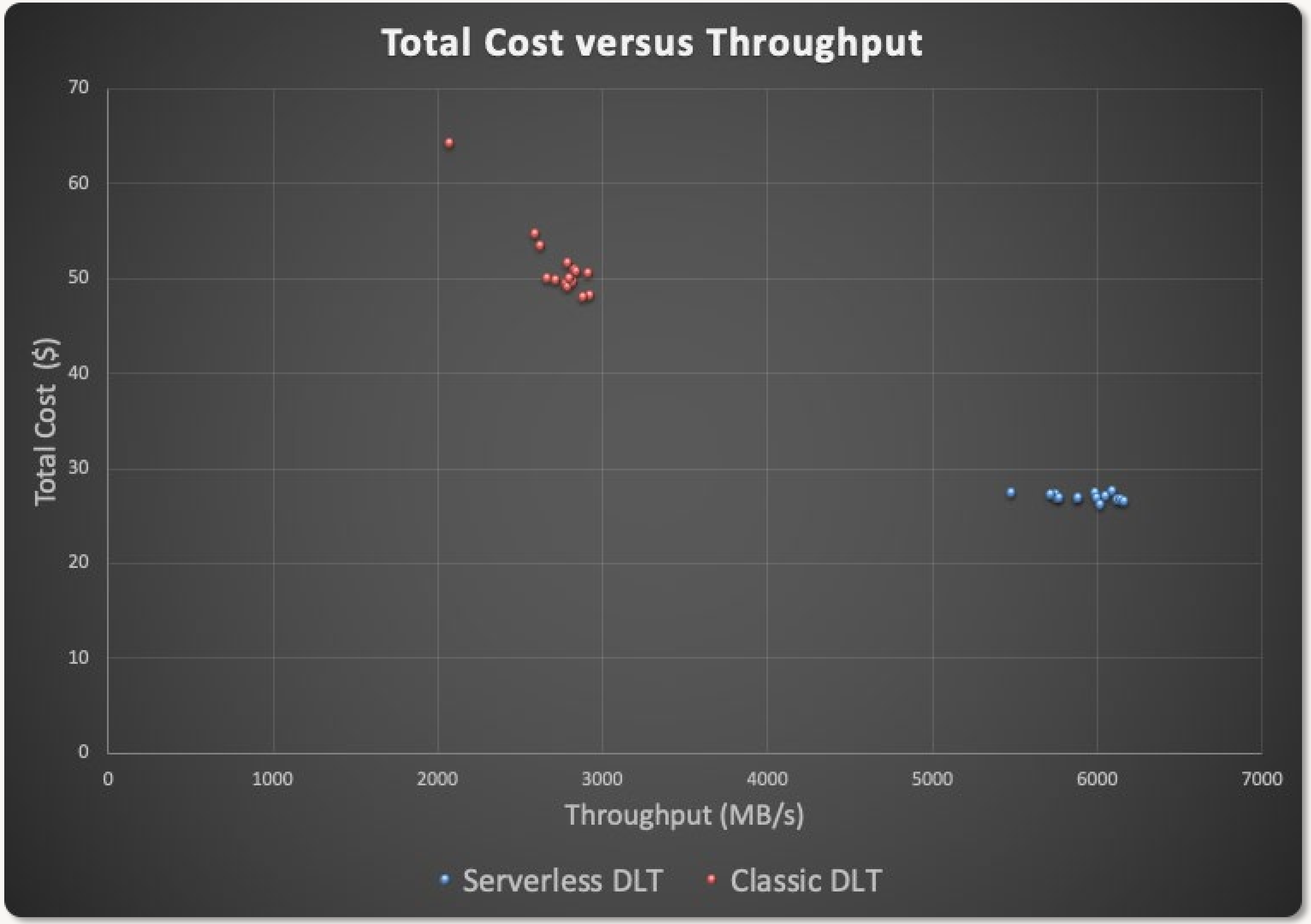
- First Update: initial load (ie CREATE)
- Subsequent Updates 2,3 and 4: update after inserting 1000 rows

DLT Classic compute configuration

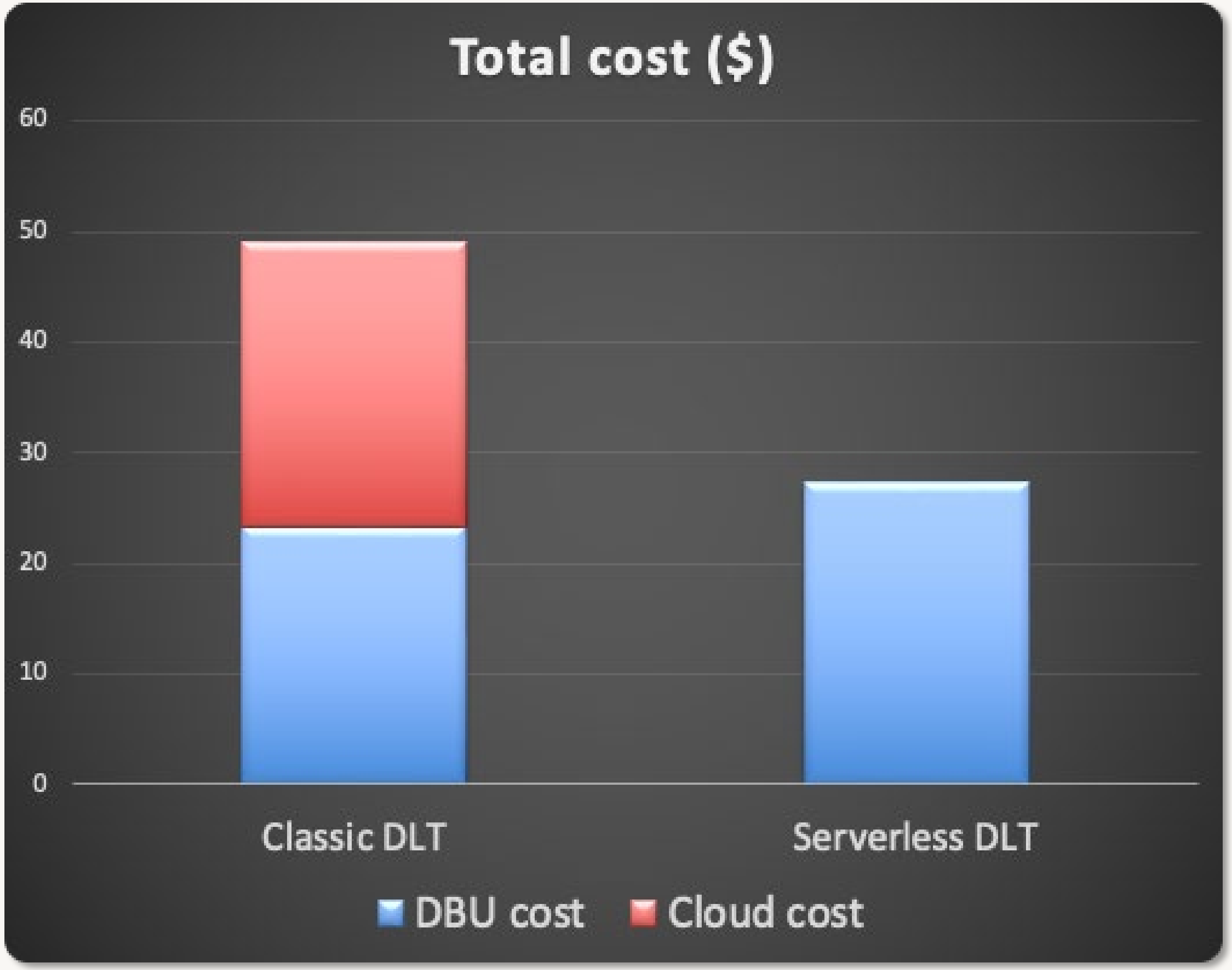
- Default instance type
- Enhanced autoscaling with 64 max clusters
- Photon OFF

```
# Create materialized view

CREATE MATERIALIZED VIEW <mv_experiment>
        AS
SELECT
    customer_id,
    min(amount) AS min_amount,
    max(amount) AS max_amount,
    avg(amount) AS avg_amount,
    sum(amount) AS total_amount
  FROM
        {bronze_table}
  GROUP BY 1
```

Raw results (Initial loading)



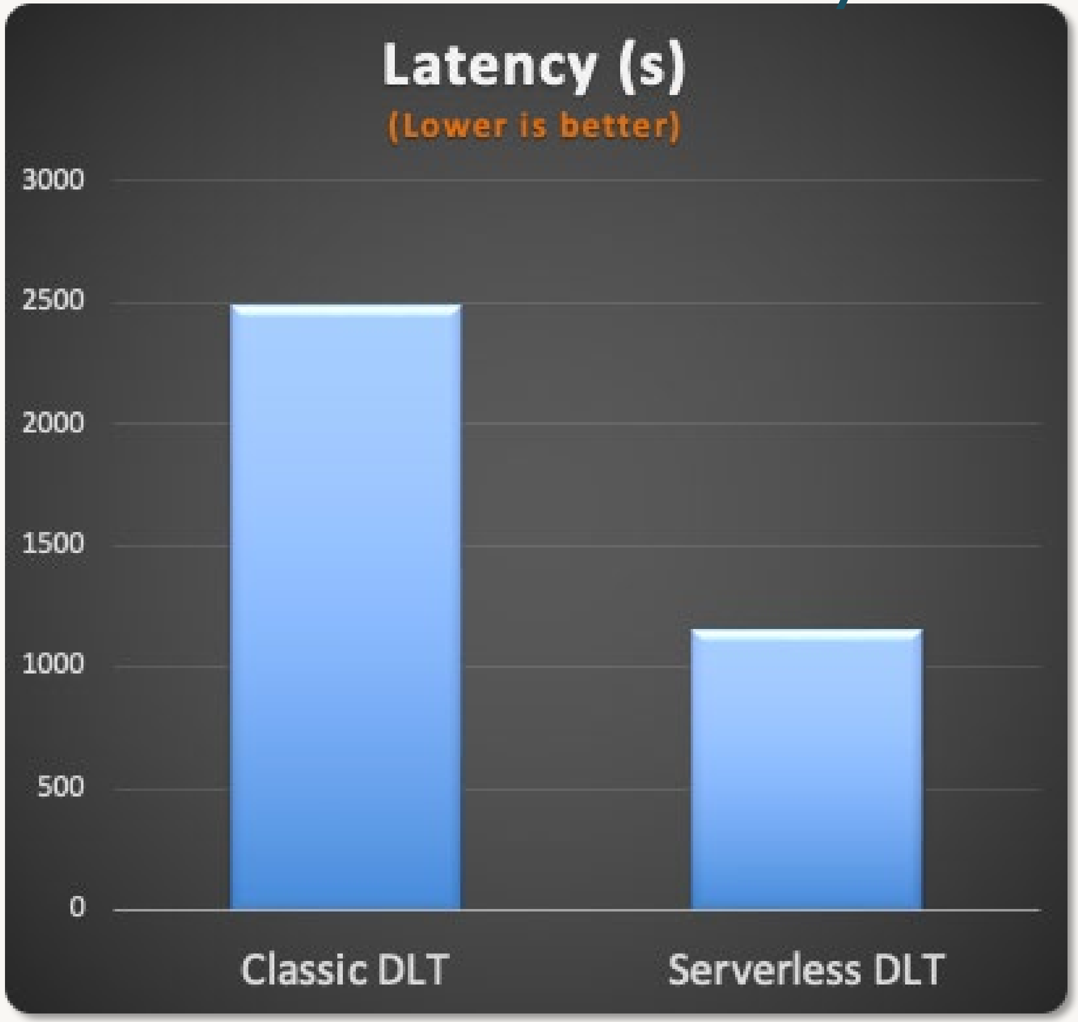
45% less TCO with serverless on the initial loading



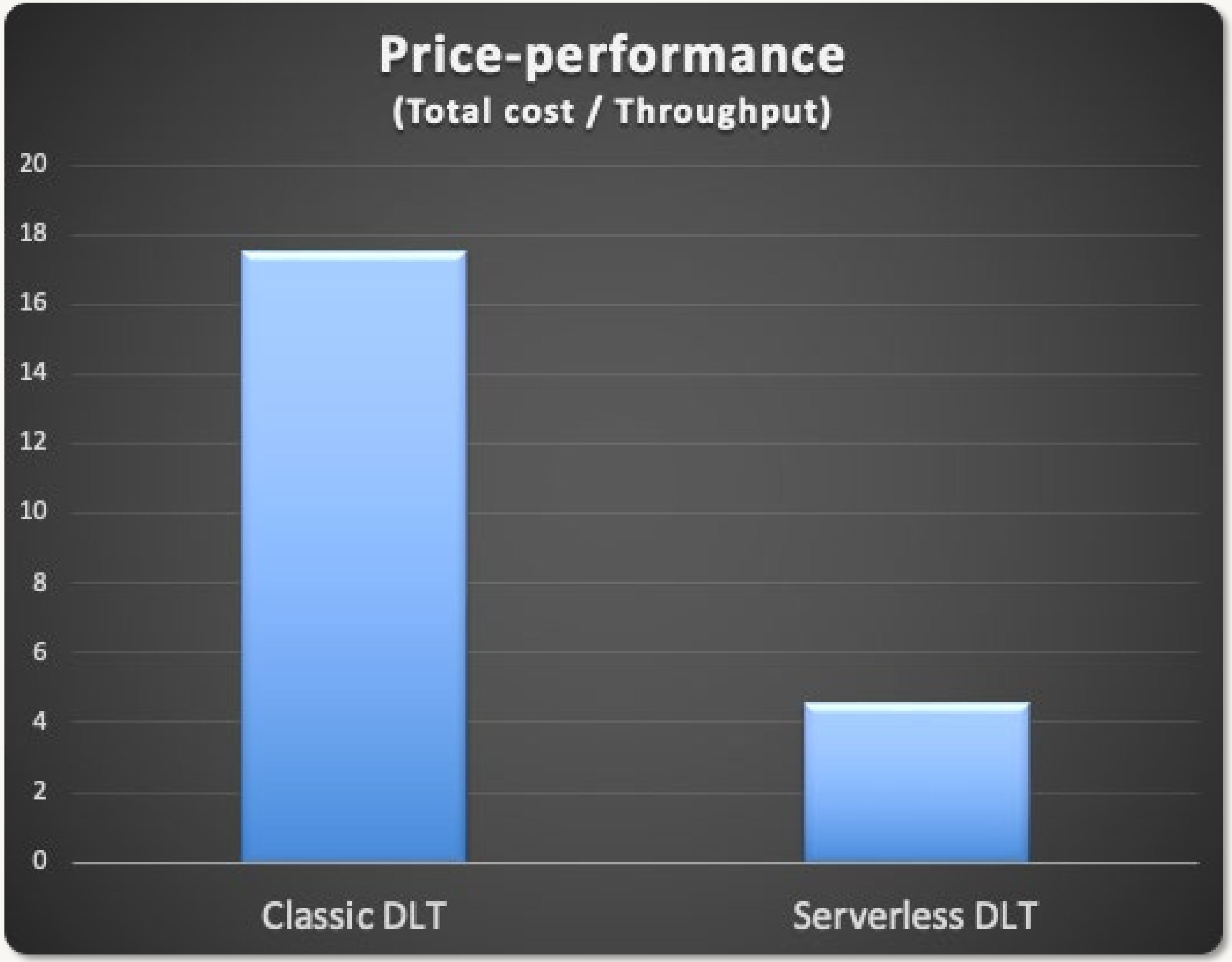
On the initial load, Serverless DLT provides

2x better throughput

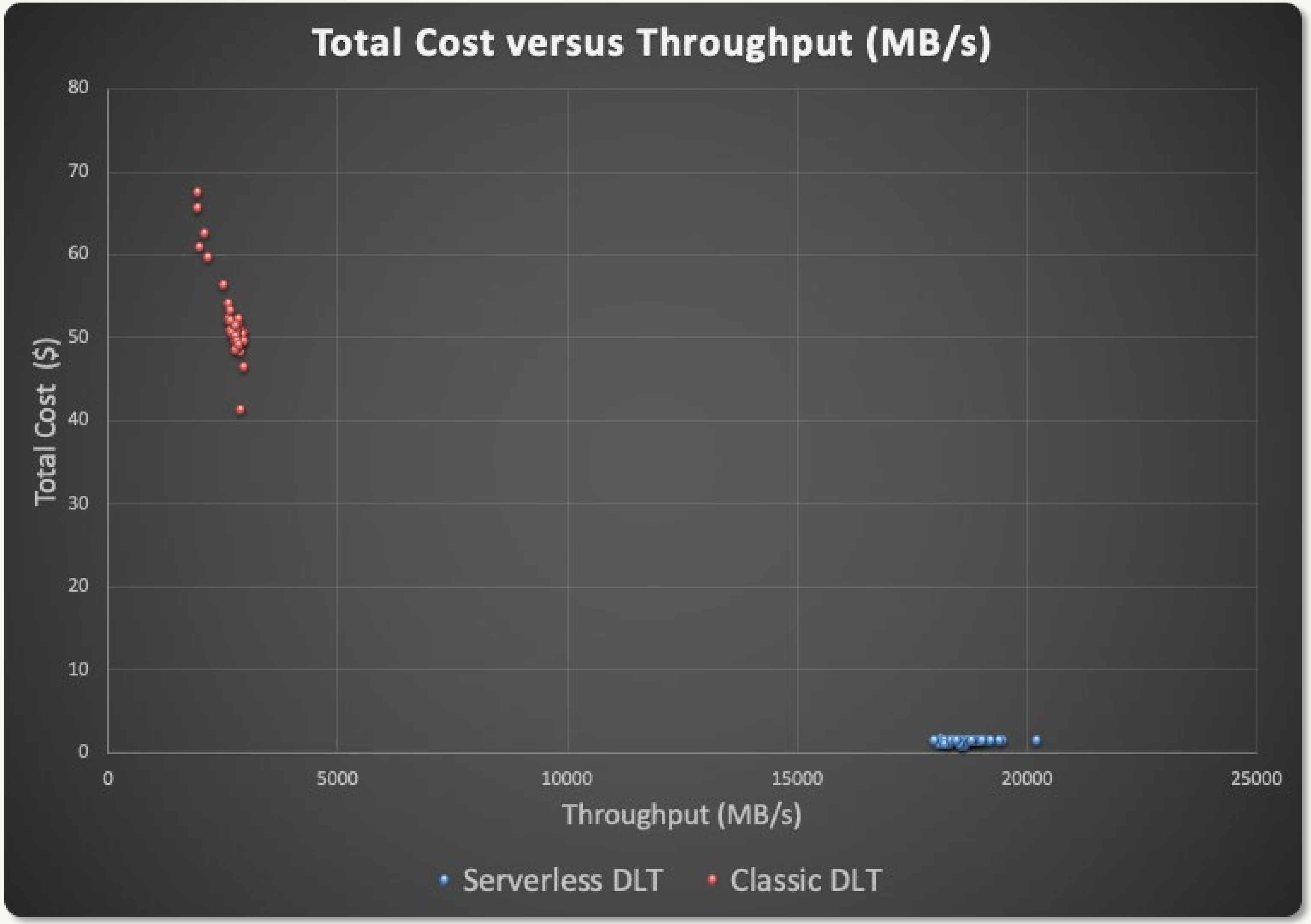
50% lower latency



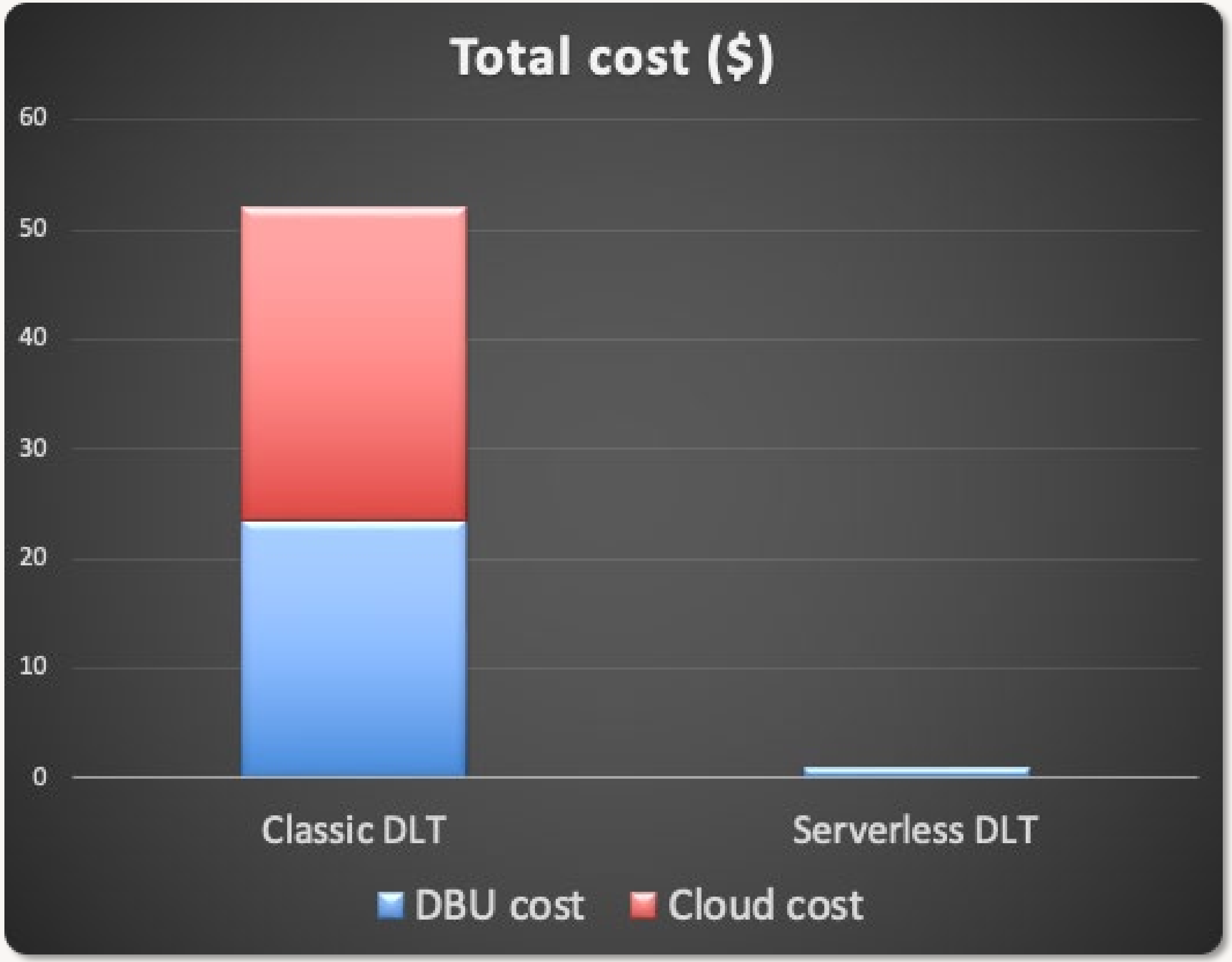
3.8x better price performance of initial load on serverless DLT



Raw results (Subsequent updates)

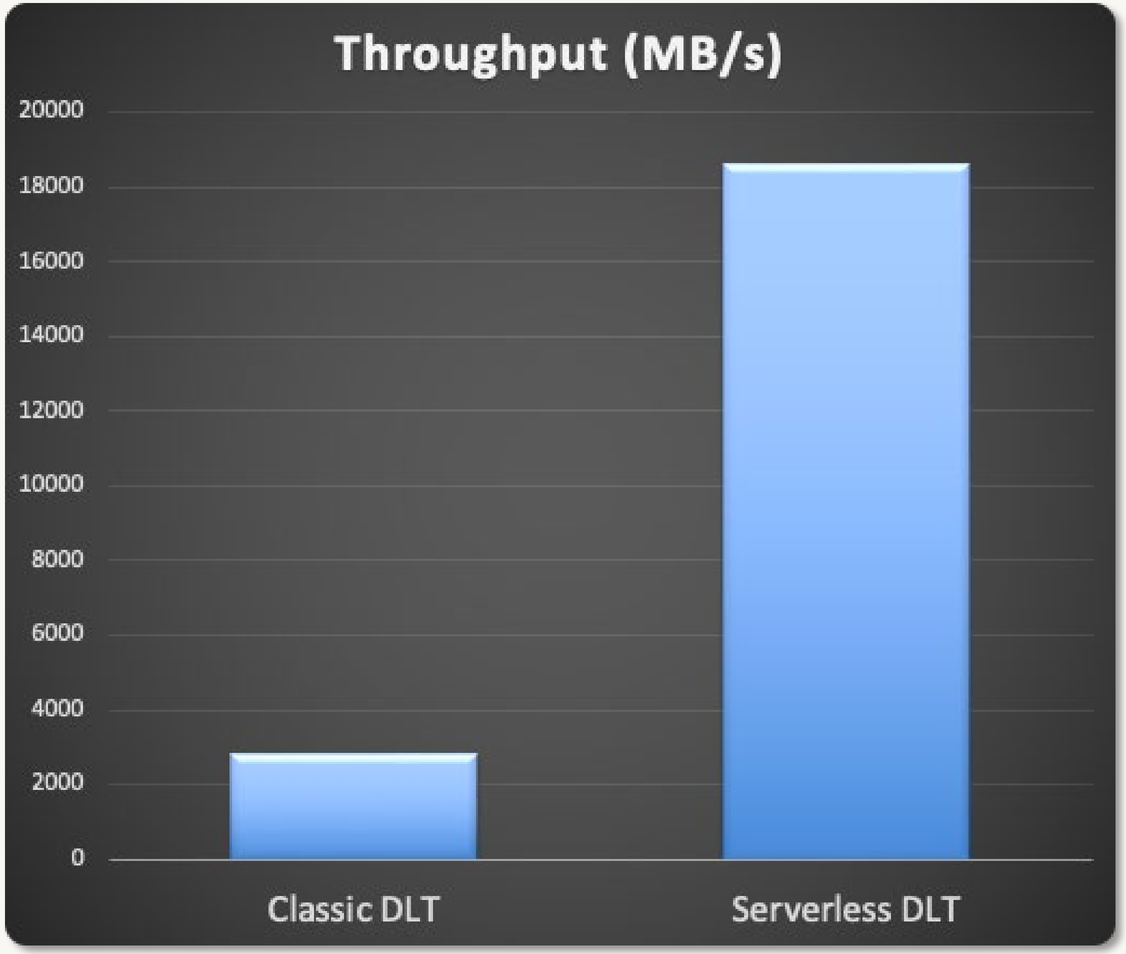


Incremental refresh resulted in 98% cost savings

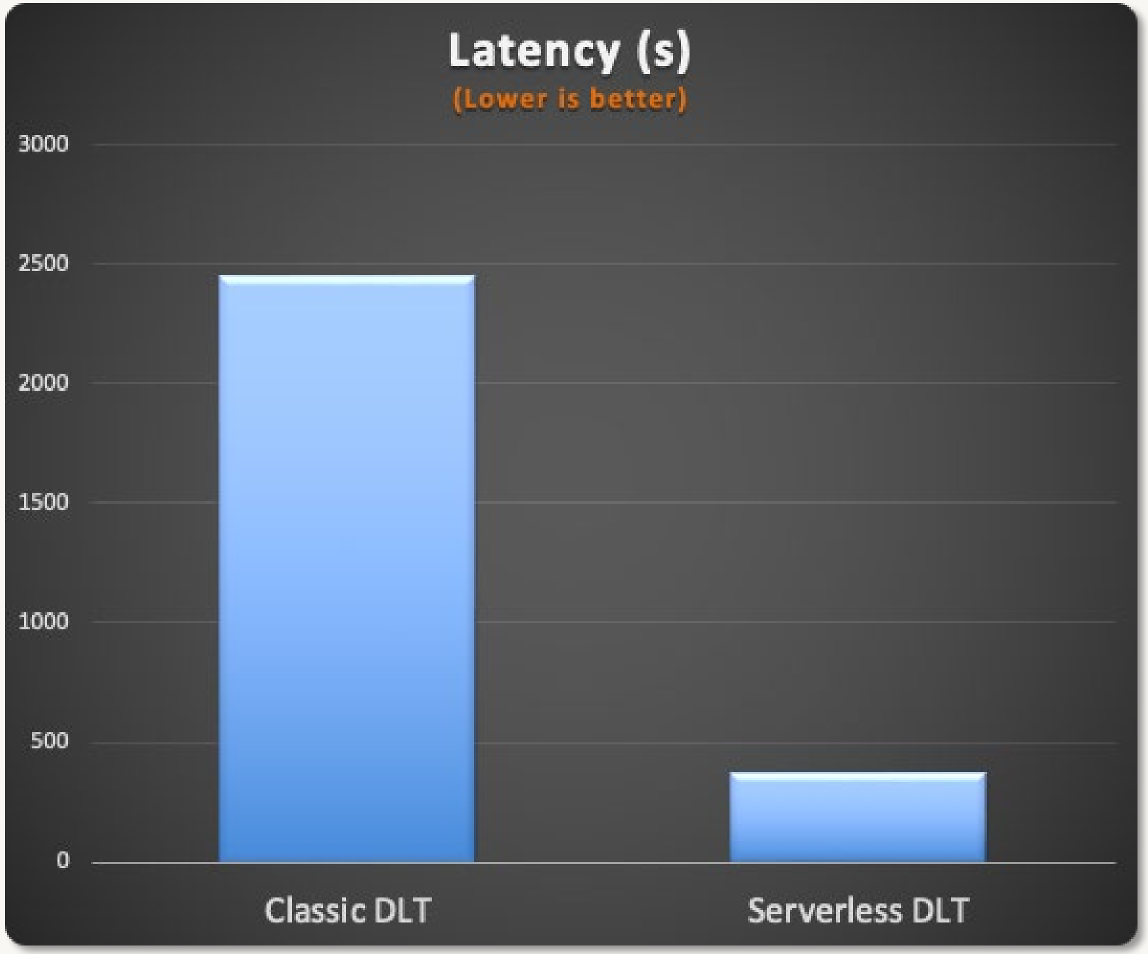


On the subsequent updates, Serverless DLT provides

6.5x better throughput



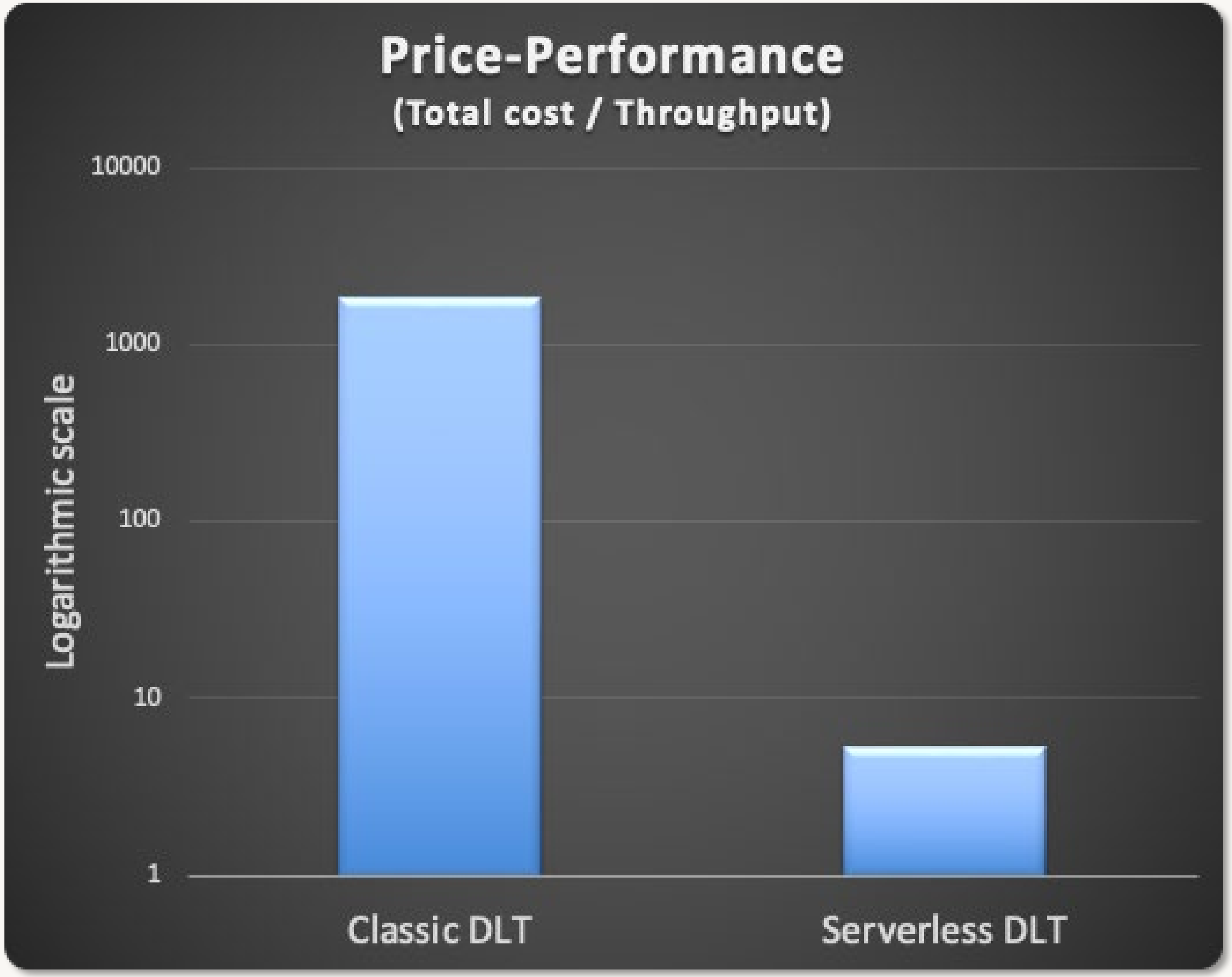
85% lower latency



Serverless DLT provides over 340x better price-performance over Classic DLT

We get these results because:

- MVs are incrementally refreshed in serverless
- Classic DLT MVs are always fully refreshed (subsequent MV refreshes are equivalent to initial load)



Run in your own environment

DLT Ingestion and MV transformation benchmarks

http://bit.ly/dlt_serverless_tco

Developing with DLT

Creating A Pipeline

How to create a pipeline from the databricks UI

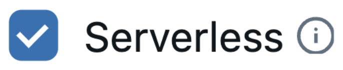
Write `CREATE ST/MV` statements

- Table definitions are written in files (or notebooks)
- Python or SQL

```
# Create materialized view
CREATE MATERIALIZED VIEW
<name>
AS….a
```

Create a pipeline

- A Pipeline combines all source code files



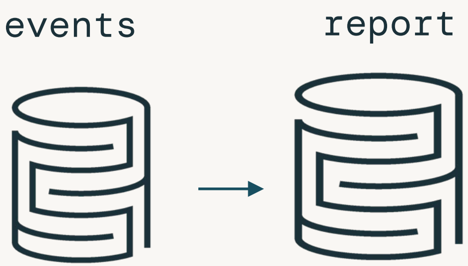
Click start

- DLT will create / update the tables and execute them in the correct order.

Automated dependency resolution

DLT detects dependencies and executes all operations in correct order

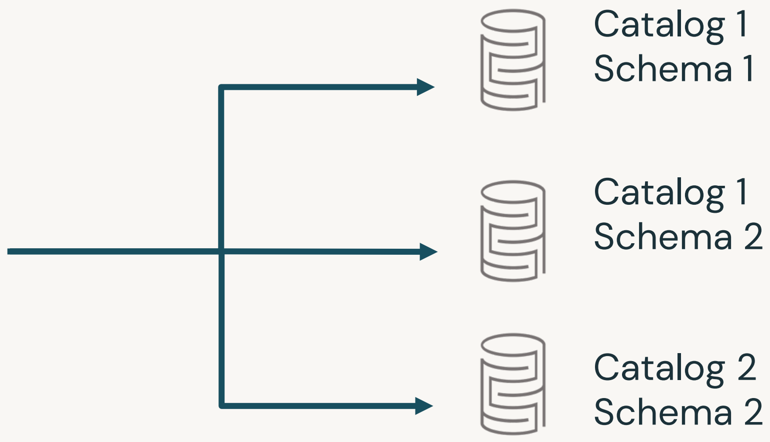
```
CREATE STREAMING TABLE events
AS SELECT … FROM prod.raw_data

CREATE MATERIALIZED VIEW report
AS SELECT … FROM LIVE.events
```



- Dependencies owned by other producers are just read from the managed or external data sources as normal.
- Dependencies from the **same pipeline** are read from the LIVE schema
- DLT handles parallelism and captures the lineage of the data.

Direct Publishing Mode in DLT

Publish tables to arbitrary catalogs and schemas from a single pipeline



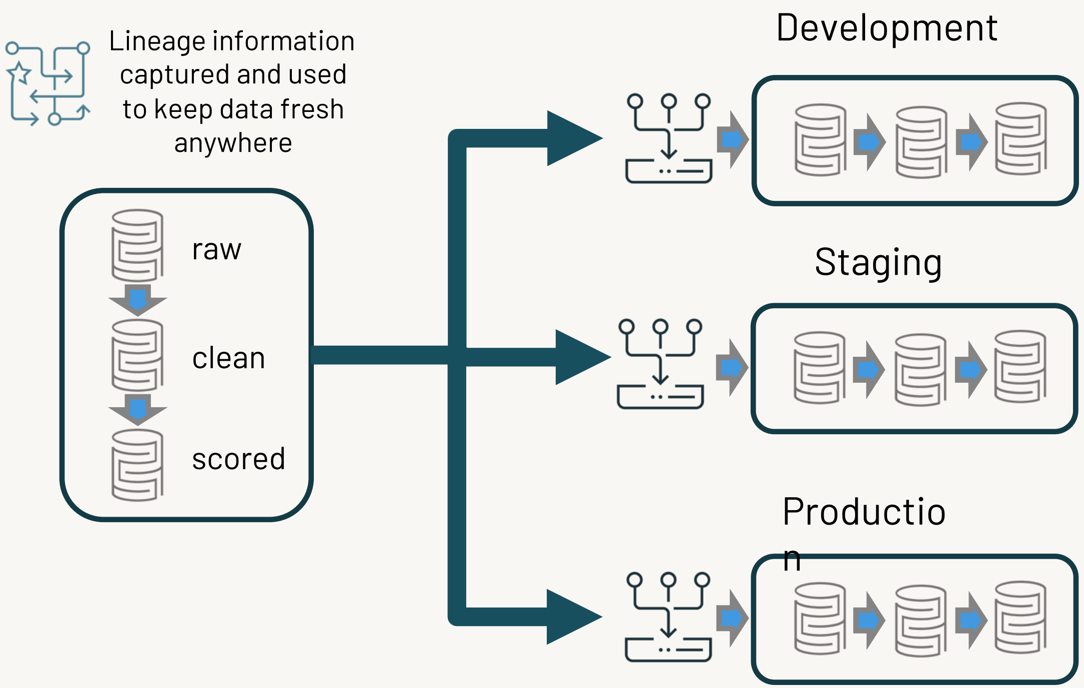

```
CREATE STREAMING TABLE catalog1.schema1.table1
     AS
SELECT
     *
FROM
     cloud_files(`path`)


CREATE MATERIALIZED VIEW catalog2.schema2.table2
     AS
SELECT
     day, sum(sales)
FROM
     catalog1.schema1.table1
GROUP BY day
```

DLT detects dependencies automatically (LIVE keyword no longer required)

Building reliable pipelines

Pipelines let you use software development best practices

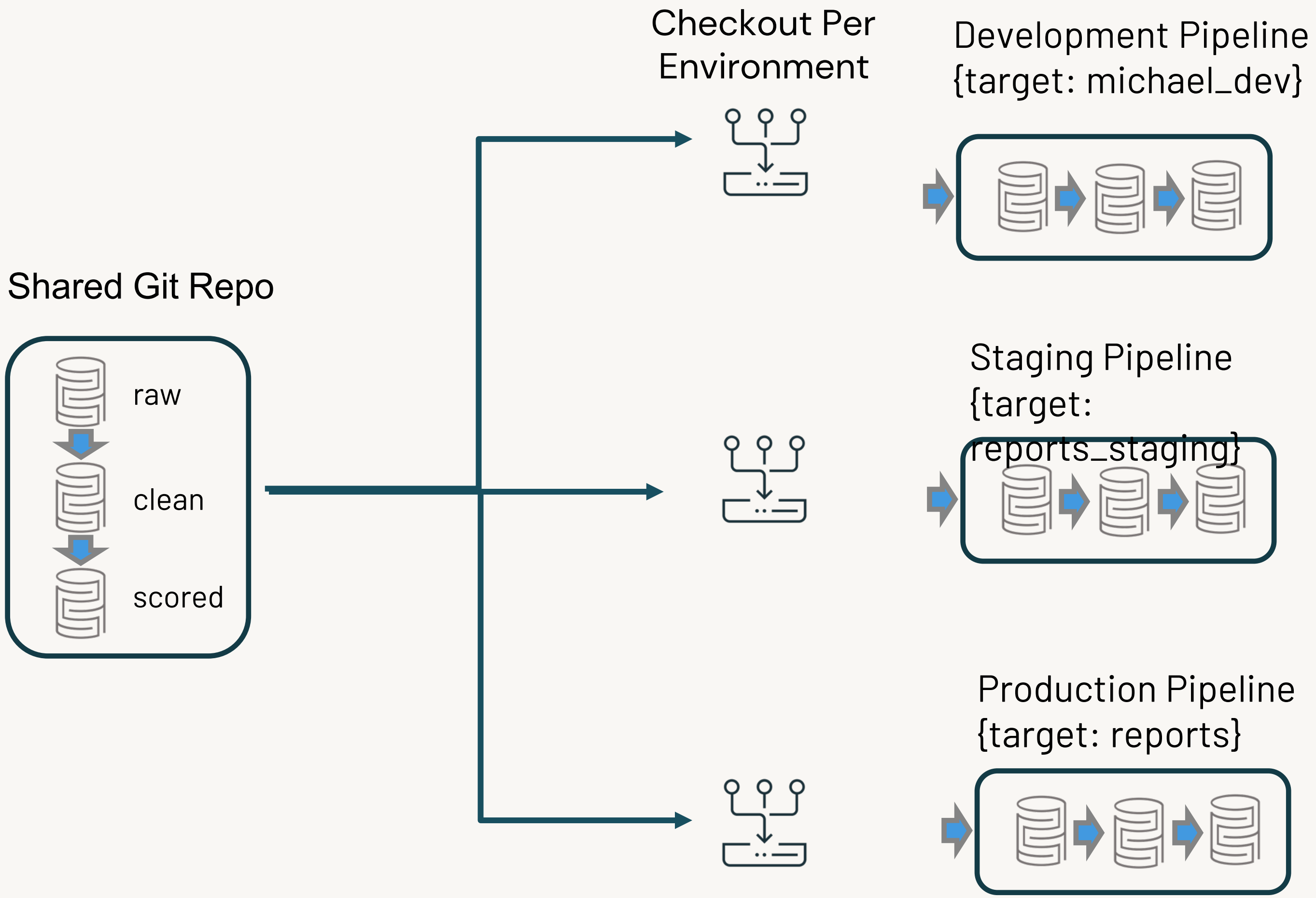
- Develop in environment(s) separate from production with the ability to easily test it before deploying - entirely in SQL
- Deploy and manage environments using parameterization
- Unit testing and documentation
- Enables metadata-driven ability to programatically scale to 100s of tables/pipelines dynamically



Databricks Asset Bundles (DABs) for development

Single configuration for all Databricks assets, including DLT

- Creating separate checkouts / pipelines is onerous / error prone
- DABs allow you to version control source code and pipeline configuration
- Automate the creation of multiple environments



Demo: DLT development in the notebook

exoplanets - E2 Dogfood

e2-dogfood.staging.cloud.databricks.com/?o=6051921418418893

E2 Dogfood

E2 Dogfood

D

+ New

Workspace

Recents

Catalog

Workflows

Compute

SQL

SQL Editor

Queries

Dashboards

Genie

Alerts

Query History

SQL Warehouses

Data Engineering

Job Runs

Pipelines

Machine Learning

Playground

Experiments

Features

Models

Serving

Marketplace

Partner Connect

E2 Dogfood (confidential)

powered by databricks

Search data, notebooks, recents, and more... ⌘ + P

For you | Mosaic AI | What's new

Quick access

Recents | Favorites | Popular

ipac nasa	15 hours ago	Schema

ps://e2-dogfood.staging.cloud.databricks.com/explore/data/nasa/ipac?o=6051921418418893

Reference architectures

Streaming tables for high speed, simple transformations

`cloud_files`

kafka

amazon KINESIS

Can have a short retention period in your source to reduce costs

Land data as-is into Delta. Avoid complex transformation that could introduce data quality problems

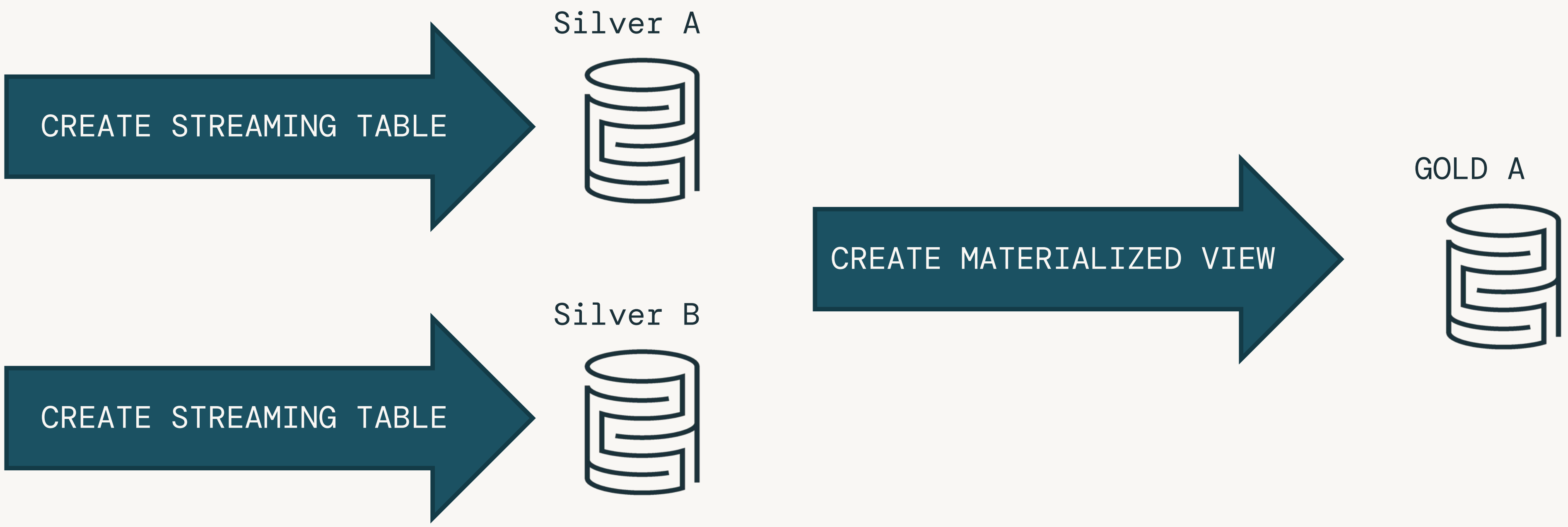
Retain infinite history in Delta

Easy to perform GDPR and other compliance tasks

Perform simple joins and aggregations, or CDC with APPLY CHANGES

Streaming table don't propagate changes, and require watermarks or time intervals to control memory utilization

Materialized views for complex transformations and modeling



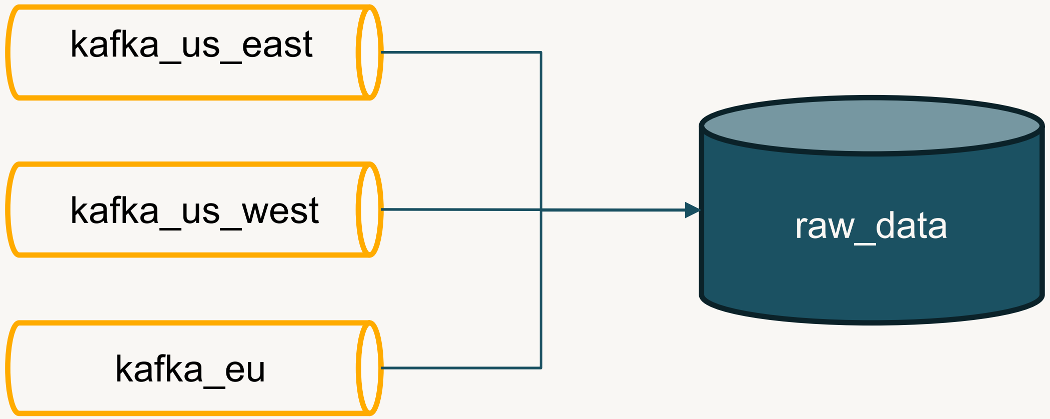
MVs can be created on any query, are always correct, and automatically propagate updates and deletes.

MVs are incrementally refreshed (when possible) in serverless

Seamlessly evolve streaming sources with Flows

Without a full refresh

```
CREATE STREAMING TABLE raw_data


CREATE FLOW kafka_us_east
AS INSERT INTO LIVE.raw_data BY NAME
SELECT * FROM kafka(…)


CREATE FLOW kafka_us_west
AS INSERT INTO LIVE.raw_data BY NAME
SELECT * FROM kafka(…)


CREATE FLOW kafka_eu
AS INSERT INTO LIVE.raw_data BY NAME
SELECT * FROM kafka(…)
```



Useful for backfills, corrections, and initial hydration of RDBMS sources