Delta Live Table

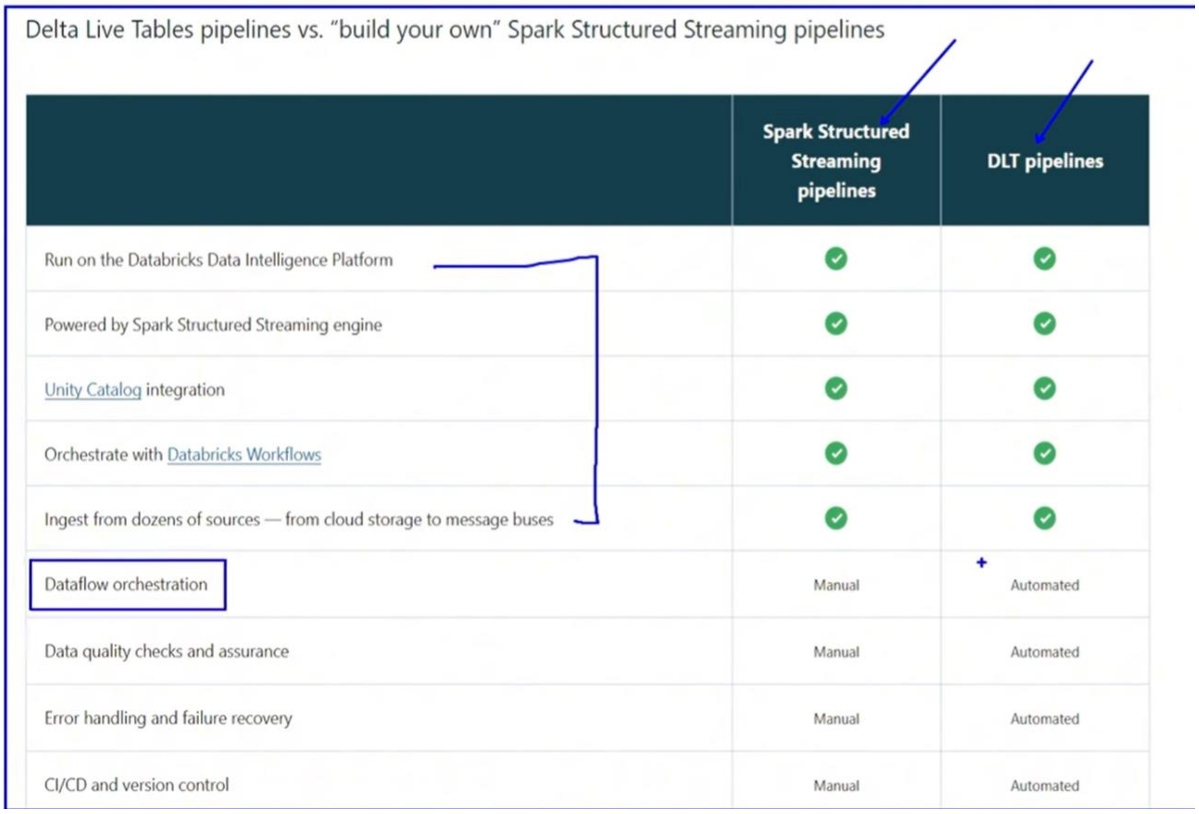
[Lakeflow Declarative Pipelines Python language reference | Databricks on AWS]

[Lakeflow Declarative Pipelines concepts | Databricks on AWS]

What is DLT?

Delta Live Tables or DLT is one of the best way to do ETL process in Lakehouse. It's responsible for creating, maintaining and testing the data flow pipelines using a declarative approach, which helps data engineer to focus more on the getting the value out of data rather than focusing on tools.

Simply define the transformations to perform on your data and let DLT pipelines **automatically manage task orchestration, cluster management, monitoring, data quality and error handling**.



Delta Live Tables pipelines vs. "build your own" Spark Structured Streaming pipelines

	Spark Structured Streaming pipelines	DLT pipelines
Run on the Databricks Data Intelligence Platform	✓	✓
Powered by Spark Structured Streaming engine	✓	✓
Unity Catalog integration	✓	✓
Orchestrate with Databricks Workflows	✓	✓
Ingest from dozens of sources — from cloud storage to message buses	✓	✓
Dataflow orchestration	Manual	Automated
Data quality checks and assurance	Manual	Automated
Error handling and failure recovery	Manual	Automated
CI/CD and version control	Manual	Automated

In DLT we have two option to use – python and SQL

Benefits:

Quality check: Built-in DLT something called Expectations where you can define rues (like "this column should never be null"). If a record breaks the rule, DLT will drop it or send it to quarantine – and you don't have to write extra code to handle it (data quality check).

Automatic Dependency management: you don't need to worry about the order of transformations. Just define you tables and DLT figures out which table depends on which and runs things in the correct order.

Incremental processing: DLT is smart – it only processes new or changed data using something called **Change Data Capture (CDC).**

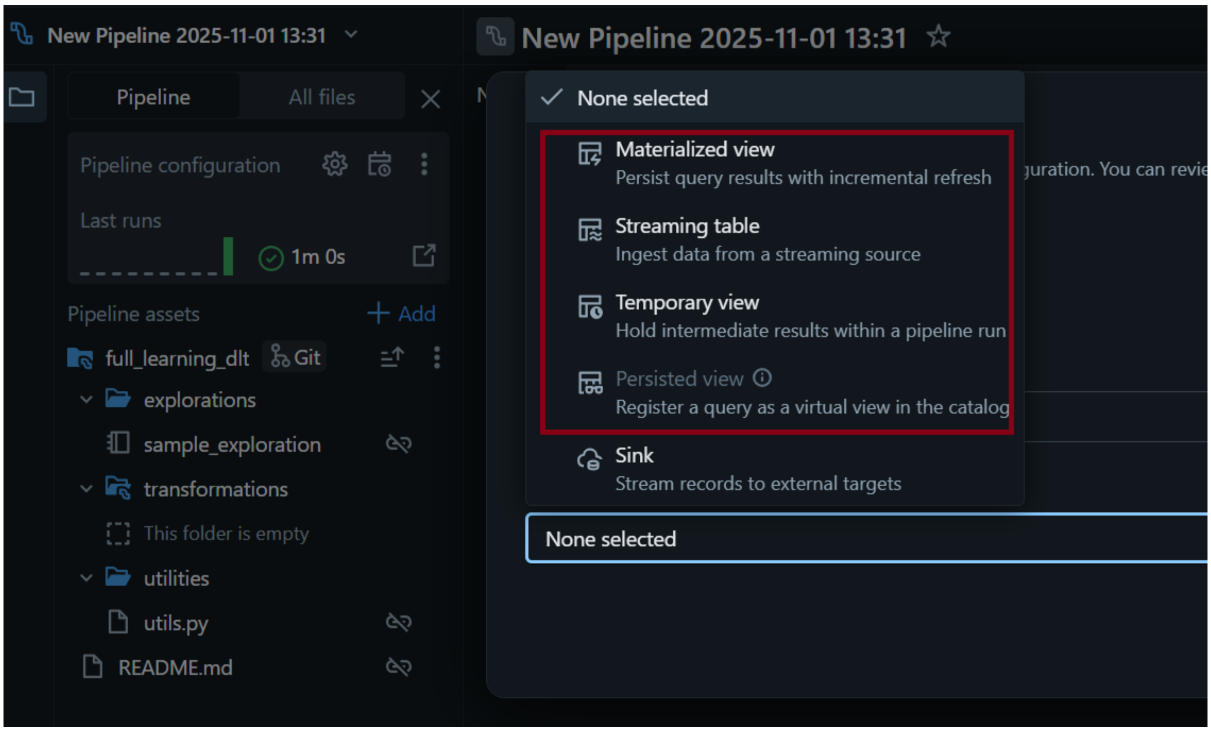
Unified batch and streaming: normally, batch and streaming pipelines are built and maintained separately. But with DLT, you can use the same code for both. It detects whether your data source is streaming and handles it accordingly.

Benefits

1. Automatic orchestration
2. Declarative processing
3. Unified batch + streaming
4. Built-in data quality
5. Resiliency & reliability
6. Governanace & lineage
7. Productivity boost

Lakeflow Declarative Pipelines Python functions are defined in the **pyspark.pipelines** module (imported as dp). Your pipelines implemented with the Python API must import this module:

from pyspark import pipelines as dp



Three major components in DLT (main building blocks)

Streaming Table

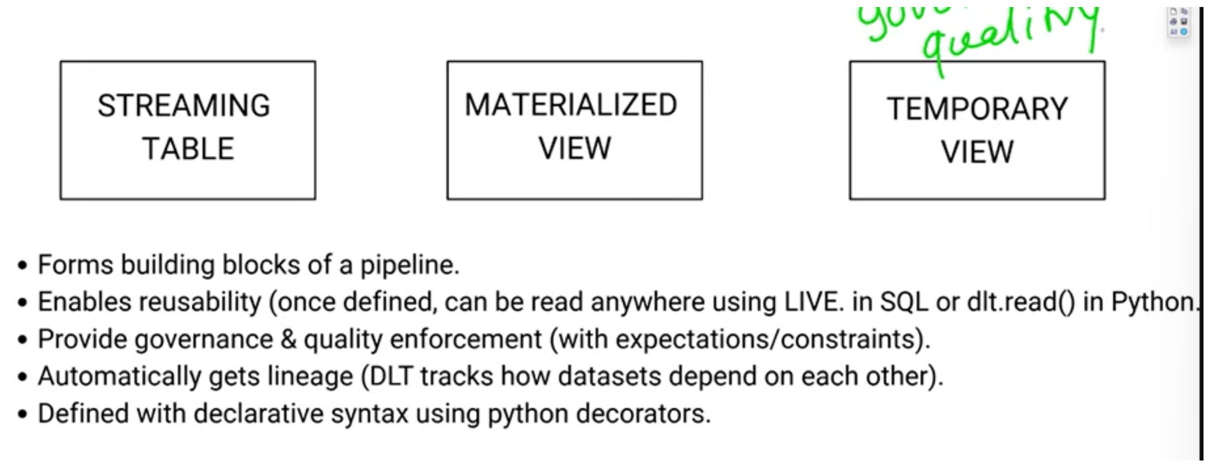
If we are going to work with streaming data (incremental data) go for this. (it better than the auto loader).

Materialized View

If you are writing some query (select * from catalog.schema.table), then it gives some result. That result is stored in the table.

(used in gold layer) [dp.table + spark.read ➔ materialized view]

Is a database object that stores the results of a query physically in the database.

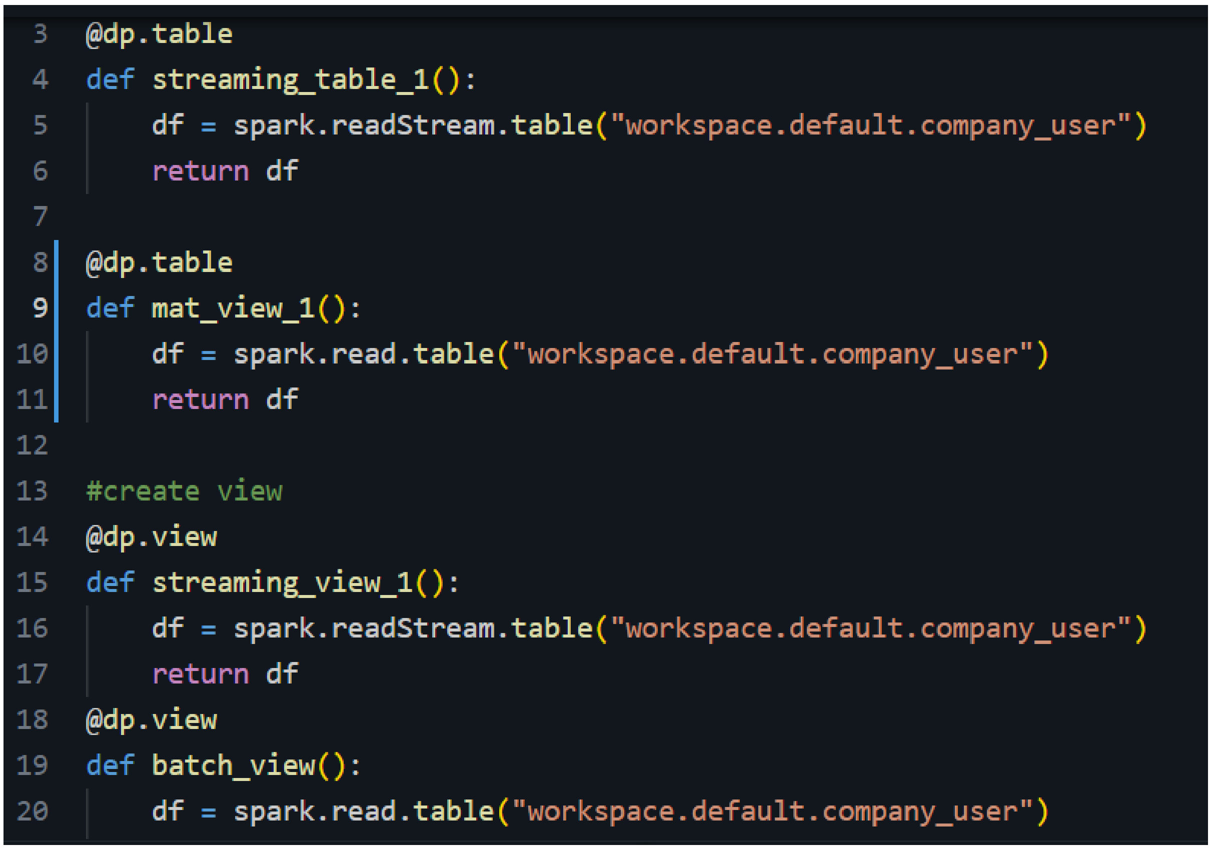


- Forms building blocks of a pipeline.
- Enables reusability (once defined, can be read anywhere using LIVE. in SQL or dlt.read() in Python.
- Provide governance & quality enforcement (with expectations/constraints).
- Automatically gets lineage (DLT tracks how datasets depend on each other).
- Defined with declarative syntax using python decorators.

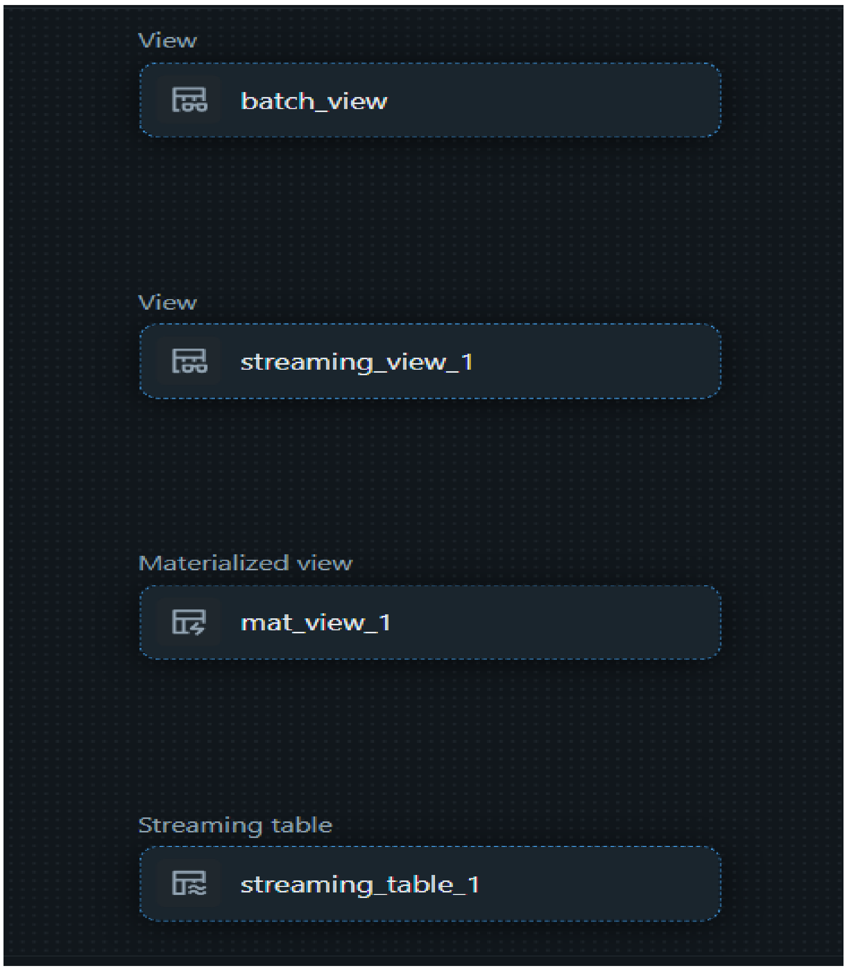
View:

If you want to do transformation (silver layer) and join two datasets then go with views.

Batch view and streaming views are in DLT



```python
@dp.table
def streaming_table_1():
    df = spark.readStream.table("workspace.default.company_user")
    return df

@dp.table
def mat_view_1():
    df = spark.read.table("workspace.default.company_user")
    return df

#create view
@dp.view
def streaming_view_1():
    df = spark.readStream.table("workspace.default.company_user")
    return df
@dp.view
def batch_view():
    df = spark.read.table("workspace.default.company_user")
```


View

batch_view

View

streaming_view_1

Materialized view

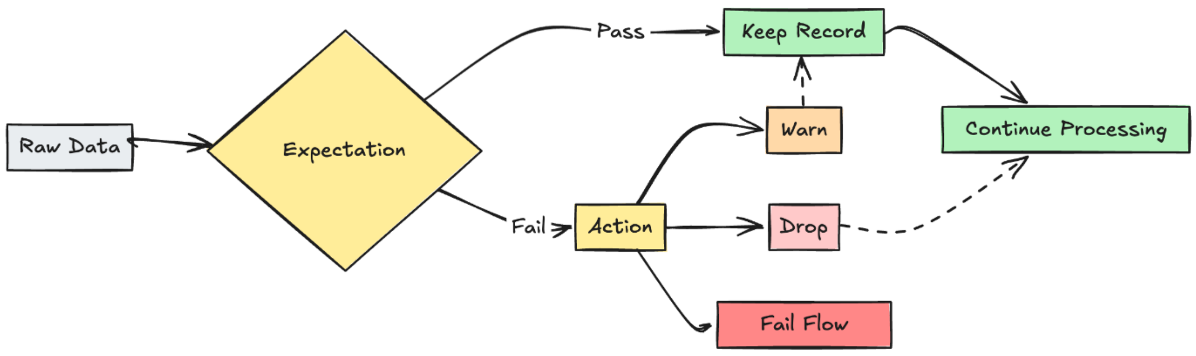
mat_view_1

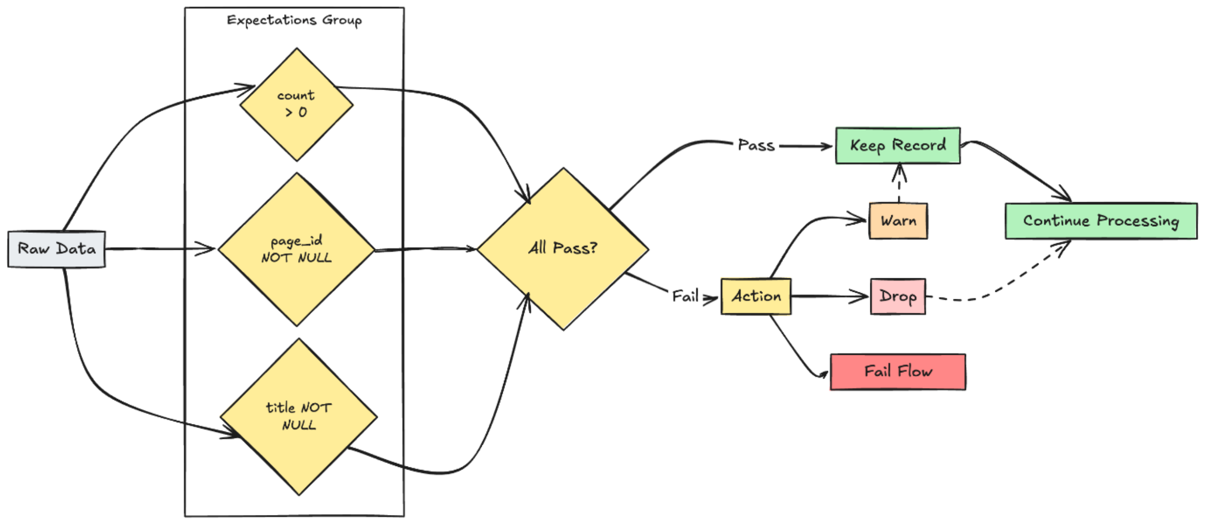
Streaming table

streaming_table_1

Expectations

Optional clauses in pipeline materialized view, streaming table or view creation statements that apply data quality checks on each record passing through a query.





Multiple expectations management

You can group multiple expectations together and specify collective actions using the functions **expect_all, expect_all_or_drop, and expect_all_or_fail**

Action	SQL syntax	Python syntax	Result
warn (default)	`EXPECT`	`dp.expect`	Invalid records are written to the target.
drop	`EXPECT ... ON VIOLATION DROP ROW`	`dp.expect_or_drop`	Invalid records are dropped before data is written to the target. The count of dropped records is logged alongside other dataset metrics.
fail	`EXPECT ... ON VIOLATION FAIL UPDATE`	`dp.expect_or_fail`	Invalid records prevent the update from succeeding. Manual intervention is required before reprocessing. This expectation causes a failure of a single flow and does not cause other flows in your pipeline to fail.

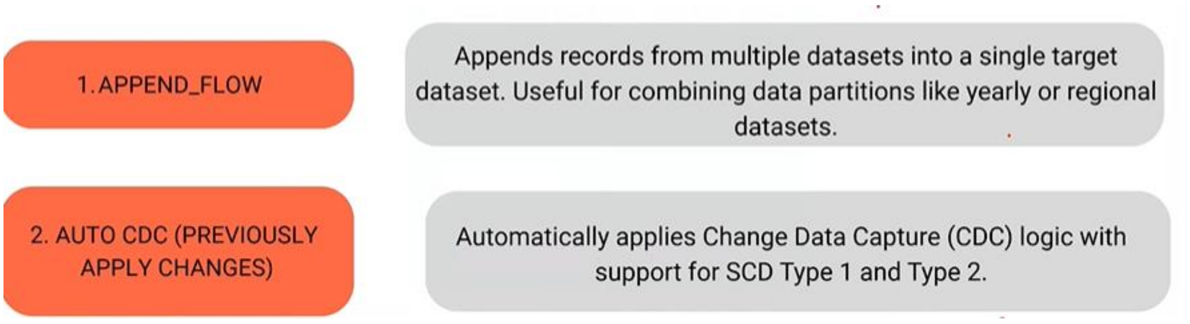
Flow

Load and process data incrementally with Lakeflow Declarative Pipelines flows

Data is processed on lakeflow declarative pipelines through flows.

Each flow consists of a query, and typically a target. The flow processes the query, either as batch or incrementally as a stream of data into the target. A flow lives within an ETL pipeline in databricks.

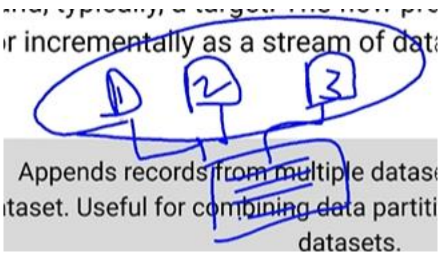
Typically, flows are defined automatically when you create a query in lakeflow declarative pipeline that updates a target, but you can also explicitly define additional flows for more complex processing, such as appending to a single target from multiple sources.



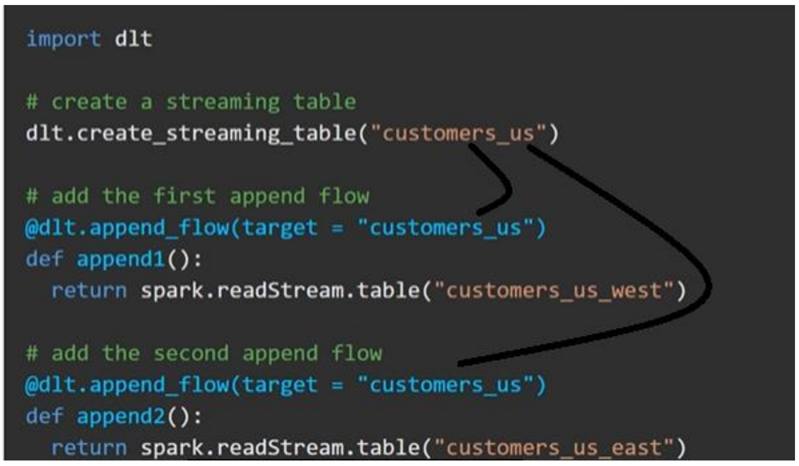
1. APPEND_FLOW	Appends records from multiple datasets into a single target dataset. Useful for combining data partitions like yearly or regional datasets.
2. AUTO CDC (PREVIOUSLY APPLY CHANGES)	Automatically applies Change Data Capture (CDC) logic with support for SCD Type 1 and Type 2.

Append flow

The @dlt.append_flow decorator creates append flows or backfills for your Lakeflow Declarative Pipelines tables.


r incrementally as a stream of dat
1
2
3
Appends records from multiple datas
taset. Useful for combining data partiti
datasets.

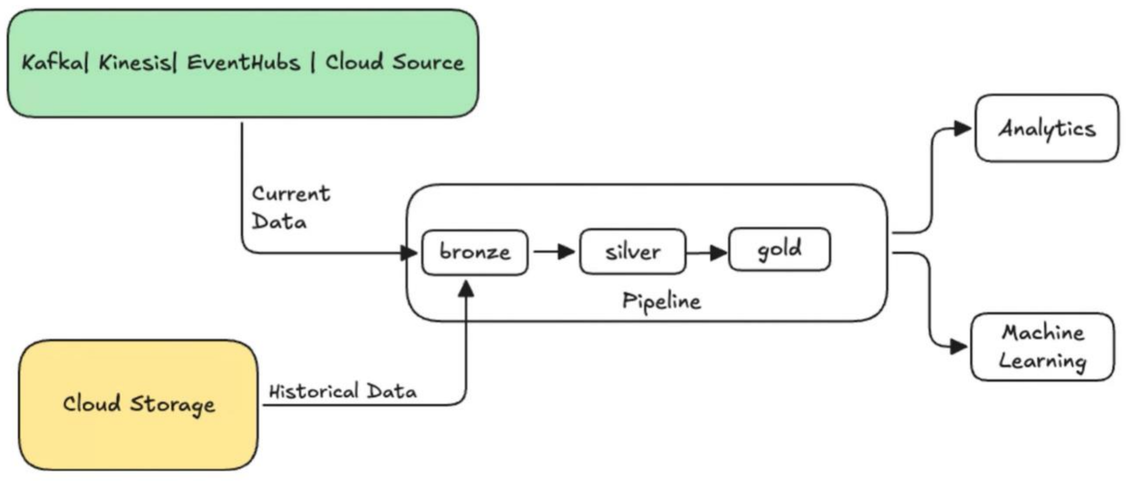
let's say we have data in source 1, source 2, source 3 all the data is collected and stored in single source. [that is append flow].



```
import dlt

# create a streaming table
dlt.create_streaming_table("customers_us")

# add the first append flow
@dlt.append_flow(target = "customers_us")
def append1():
  return spark.readStream.table("customers_us_west")

# add the second append flow
@dlt.append_flow(target = "customers_us")
def append2():
  return spark.readStream.table("customers_us_east")
```

Backfilling

BACKFILLING HISTORICAL DATA



Data backfilling is the process of filling in missing or incomplete data in a dataset or pipeline to ensure data completeness and accuracy.

Backfilling means processing the historical data that already exist. It’s not the data arrived today, processing historical data too.

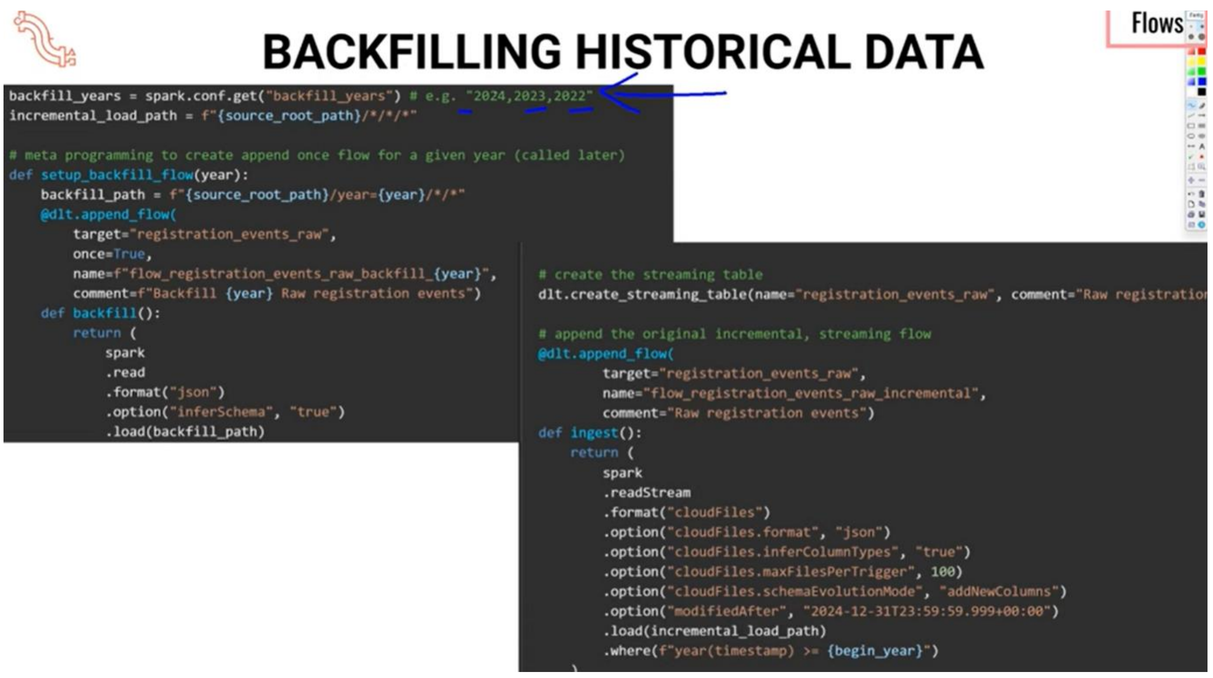
Example:

Imagine set up a pipeline for the month of March, but you also have the files from month of JAN and FEB, if you run the pipeline normally it may start from month of March onwards.

Here only the backfilling comes, backfilling concept tells this pipeline that go back and process the older data (JAN and FEB) month also.

Why do we need to use backfill?

1. First load
2. Logic changes into your pipeline (once you changed the logic then you need to process all the historical data along with current data with this new logic).
3. Late arriving data

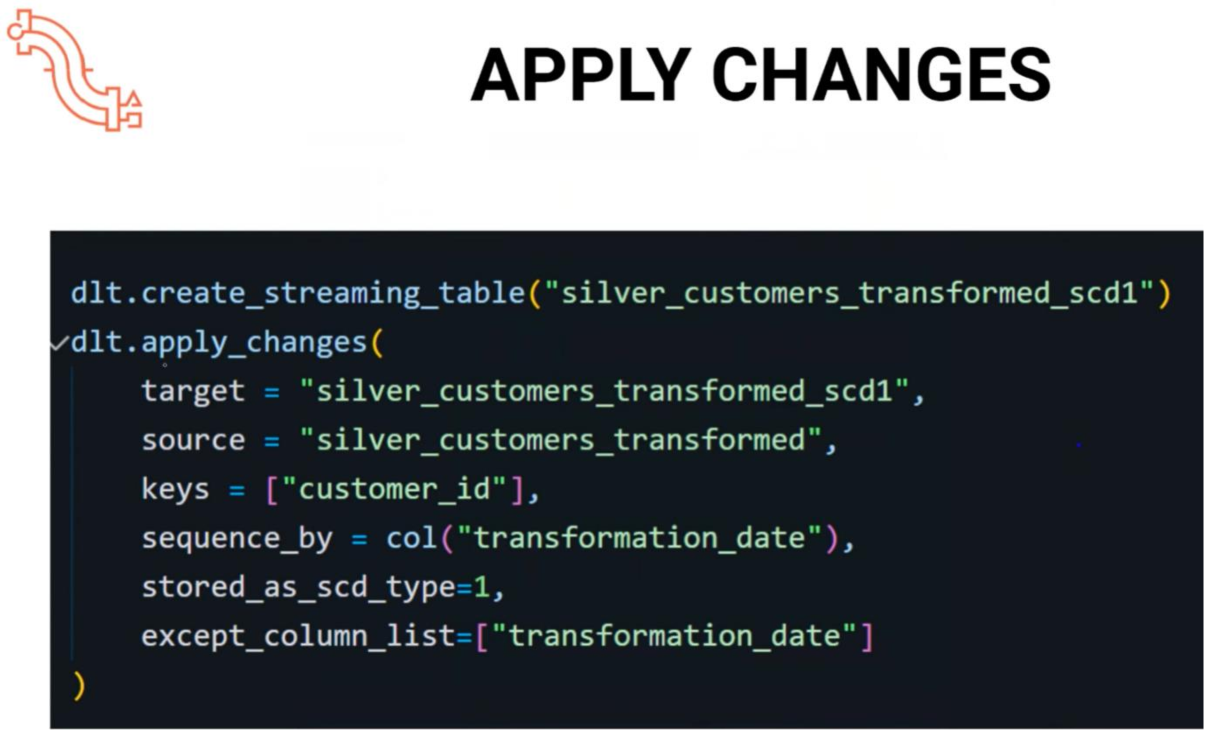

BACKFILLING HISTORICAL DATA

```python
backfill_years = spark.conf.get("backfill_years") # e.g. "2024,2023,2022"
incremental_load_path = f"{source_root_path}/*/*/*"

# meta programming to create append once flow for a given year (called later)
def setup_backfill_flow(year):
    backfill_path = f"{source_root_path}/year={year}/*/*"
    @dlt.append_flow(
        target="registration_events_raw",
        once=True,
        name=f"flow_registration_events_raw_backfill_{year}",
        comment=f"Backfill {year} Raw registration events")
    def backfill():
        return (
            spark
            .read
            .format("json")
            .option("inferSchema", "true")
            .load(backfill_path)
```

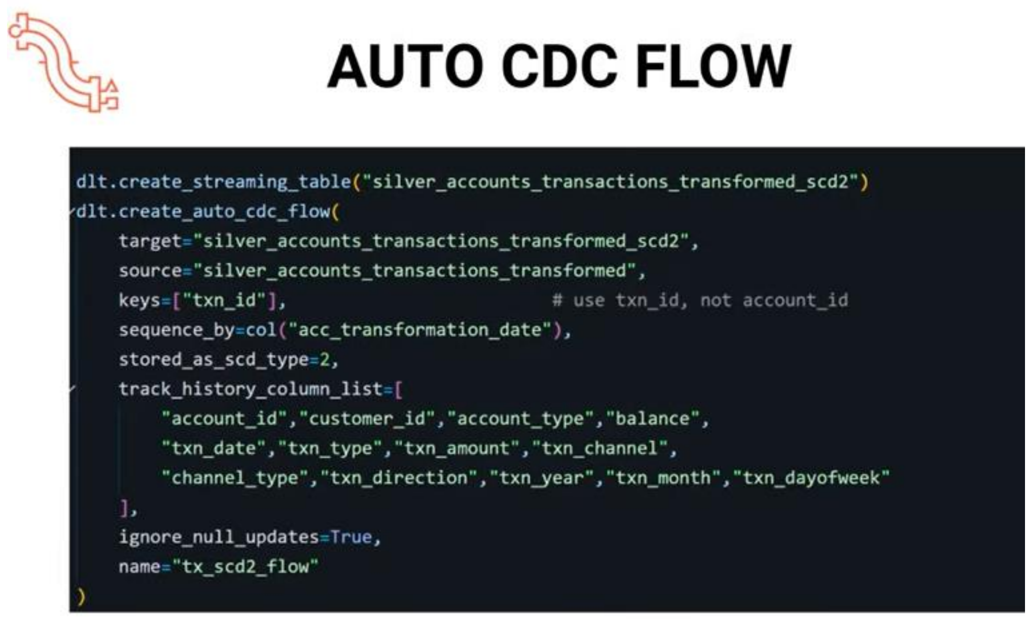
```python
# create the streaming table
dlt.create_streaming_table(name="registration_events_raw", comment="Raw registratio

# append the original incremental, streaming flow
@dlt.append_flow(
        target="registration_events_raw",
        name="flow_registration_events_raw_incremental",
        comment="Raw registration events")
def ingest():
    return (
        spark
        .readStream
        .format("cloudFiles")
        .option("cloudFiles.format", "json")
        .option("cloudFiles.inferColumnTypes", "true")
        .option("cloudFiles.maxFilesPerTrigger", 100)
        .option("cloudFiles.schemaEvolutionMode", "addNewColumns")
        .option("modifiedAfter", "2024-12-31T23:59:59.999+00:00")
        .load(incremental_load_path)
        .where(f"year(timestamp) >= {begin_year}")
```

Apply changes

Syntax for AUTO CDC and apply changes are same


APPLY CHANGES

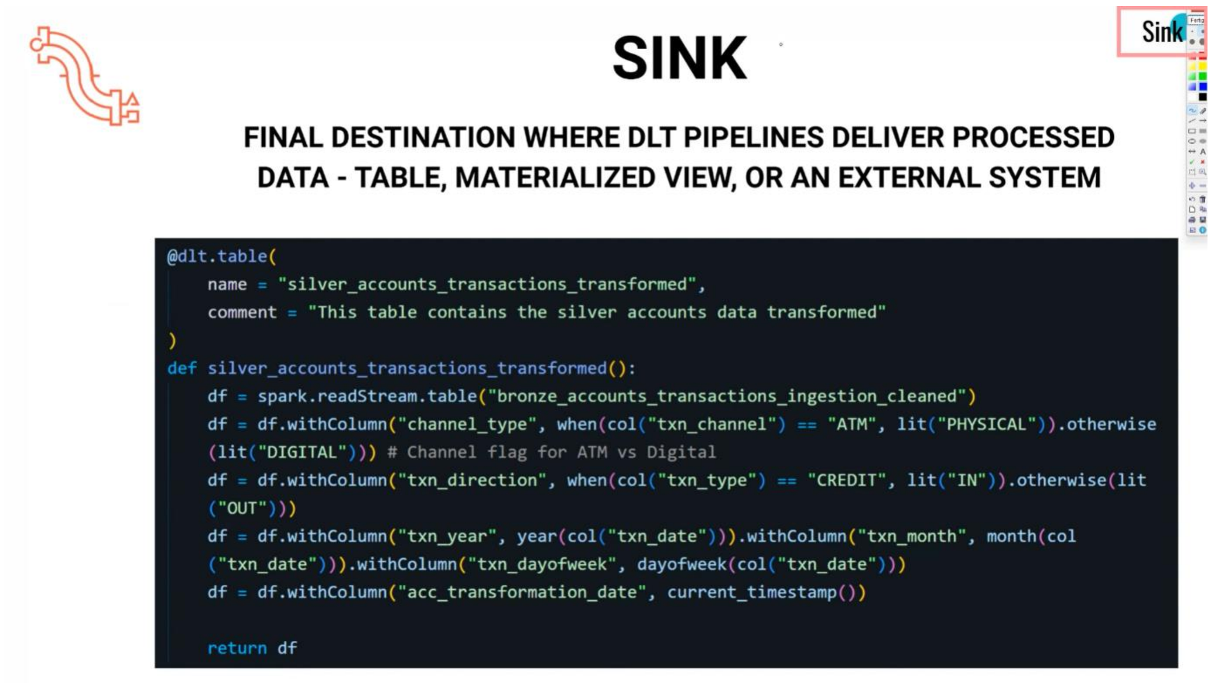
```python
dlt.create_streaming_table("silver_customers_transformed_scd1")
dlt.apply_changes(
    target = "silver_customers_transformed_scd1",
    source = "silver_customers_transformed",
    keys = ["customer_id"],
    sequence_by = col("transformation_date"),
    stored_as_scd_type=1,
    except_column_list=["transformation_date"]
)
```



AUTO CDC FLOW

```
dlt.create_streaming_table("silver_accounts_transactions_transformed_scd2")
dlt.create_auto_cdc_flow(
    target="silver_accounts_transactions_transformed_scd2",
    source="silver_accounts_transactions_transformed",
    keys=["txn_id"],                    # use txn_id, not account_id
    sequence_by=col("acc_transformation_date"),
    stored_as_scd_type=2,
    track_history_column_list=[
        "account_id","customer_id","account_type","balance",
        "txn_date","txn_type","txn_amount","txn_channel",
        "channel_type","txn_direction","txn_year","txn_month","txn_dayofweek"
    ],
    ignore_null_updates=True,
    name="tx_scd2_flow"
)
```

SINK



SINK

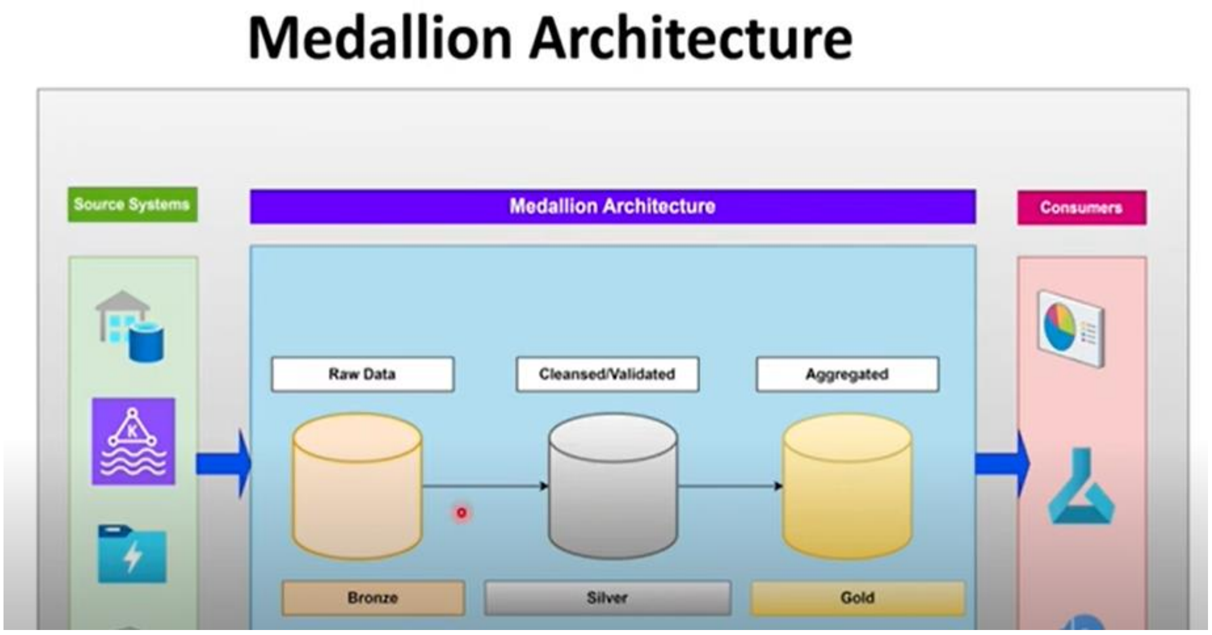
Sink

FINAL DESTINATION WHERE DLT PIPELINES DELIVER PROCESSED DATA - TABLE, MATERIALIZED VIEW, OR AN EXTERNAL SYSTEM

```
@dlt.table(
    name = "silver_accounts_transactions_transformed",
    comment = "This table contains the silver accounts data transformed"
)
def silver_accounts_transactions_transformed():
    df = spark.readStream.table("bronze_accounts_transactions_ingestion_cleaned")
    df = df.withColumn("channel_type", when(col("txn_channel") == "ATM", lit("PHYSICAL")).otherwise
    (lit("DIGITAL"))) # Channel flag for ATM vs Digital
    df = df.withColumn("txn_direction", when(col("txn_type") == "CREDIT", lit("IN")).otherwise(lit
    ("OUT")))
    df = df.withColumn("txn_year", year(col("txn_date"))).withColumn("txn_month", month(col
    ("txn_date"))).withColumn("txn_dayofweek", dayofweek(col("txn_date")))
    df = df.withColumn("acc_transformation_date", current_timestamp())

    return df
```

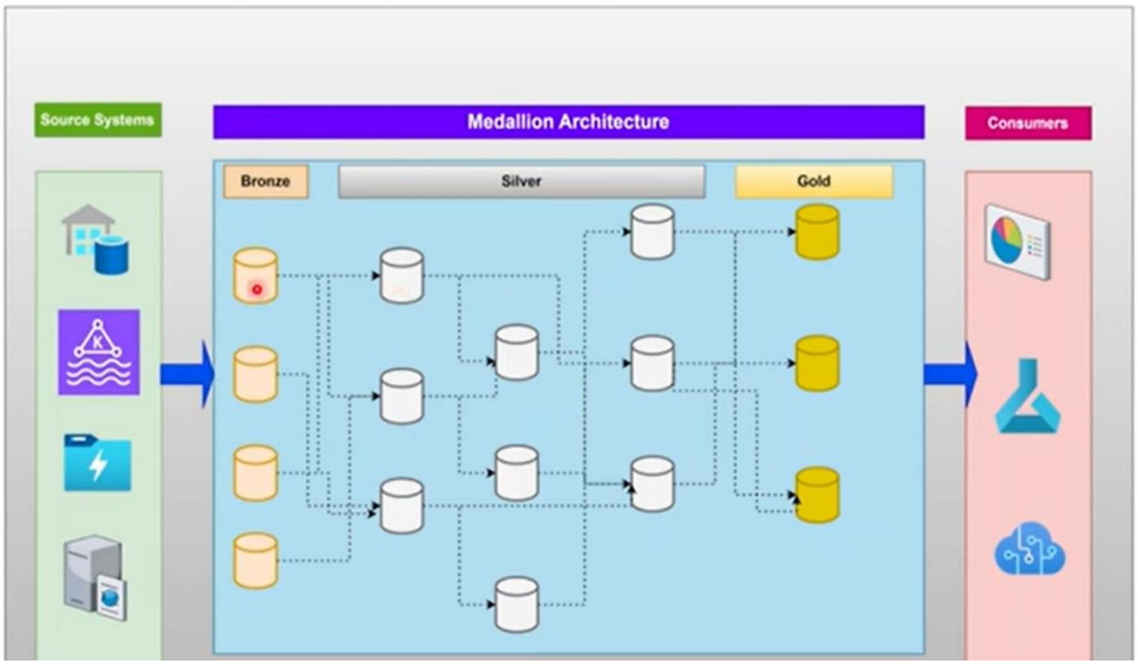
Create a streaming table from Kafka -- [Infer and evolve the schema using from_json in Lakeflow Declarative Pipelines | Databricks on AWS]

```
@dp.table(comment="from_json kafka example")
def bronze():
 return (
   spark.readStream
        .format("kafka")
        .option("kafka.bootstrap.servers", "<server:ip>")
        .option("subscribe", "<topic>")
        .option("startingOffsets", "latest")
        .load()
        .select(col("value"), from_json(col("value"), None, {"schemaLocationKey":
"keyX"}).alias("jsonCol"))
)
```



There is few problems in Medallion architecture if the component grows

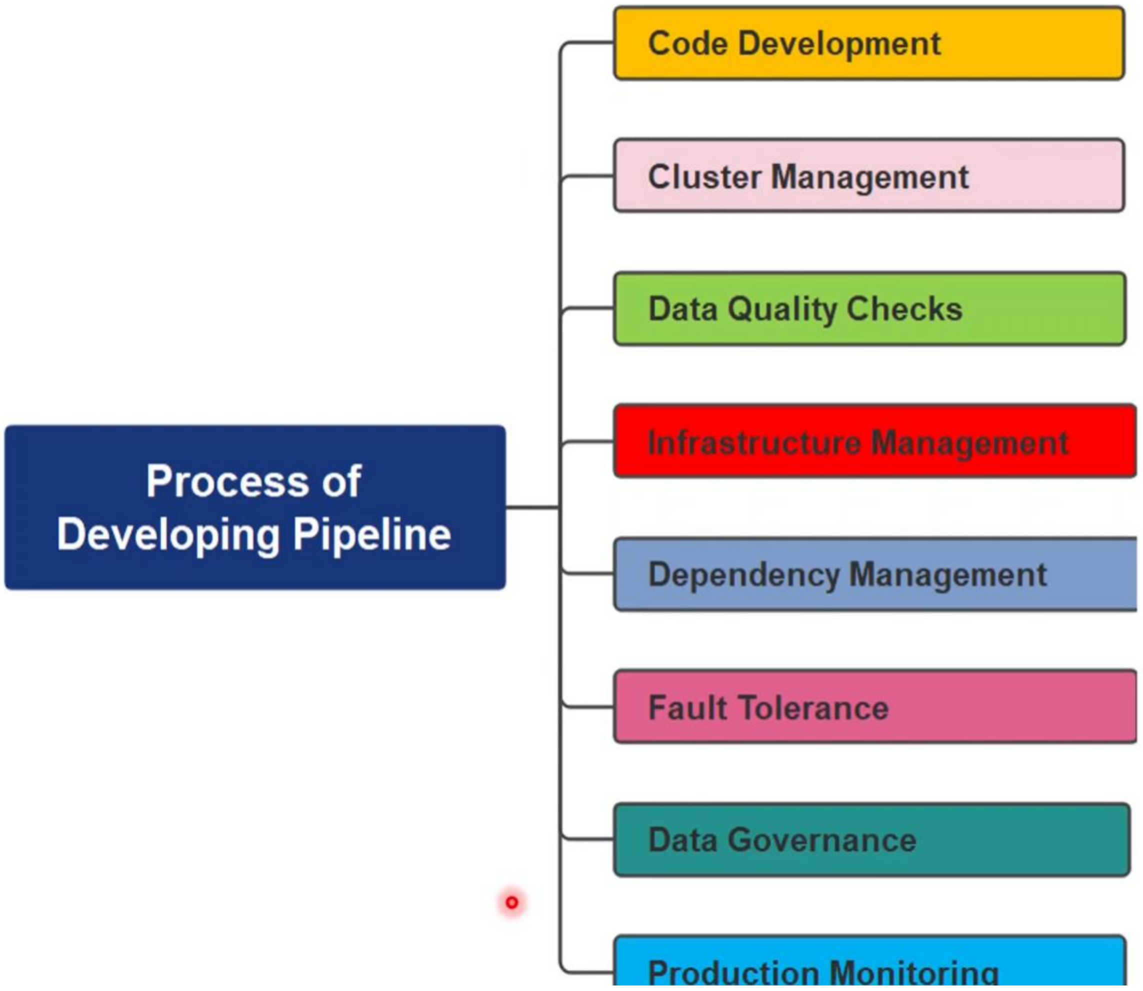
Complexity of Medallion Architecture



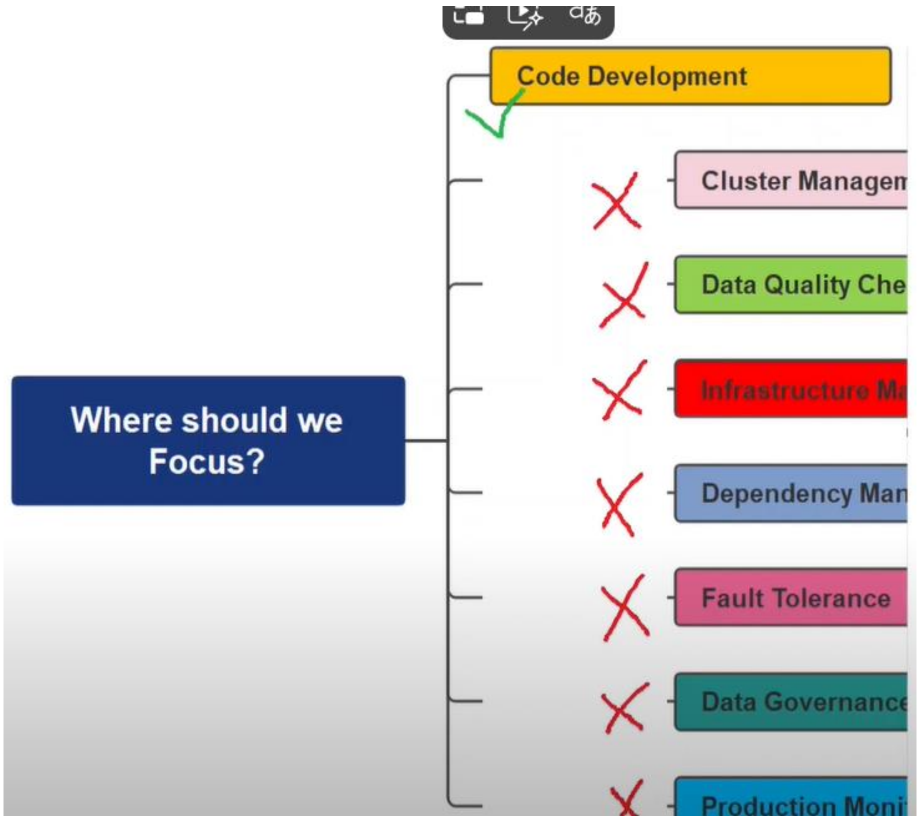
Challenges with current solution approach

- Troubleshooting and debugging is challenging, Time-consuming and laborious
- Data quality is poor and need to handle data validation manually.
- Data lineage is not available.
- Visibility/monitoring at more granular and logic level is not available. Status available only at pipeline level.
- Challenging to combine batch and streaming solution in a single pipeline.

Process of developing the ETL pipeline



By the developer the code development is only more focused.



What is Delta live Table?

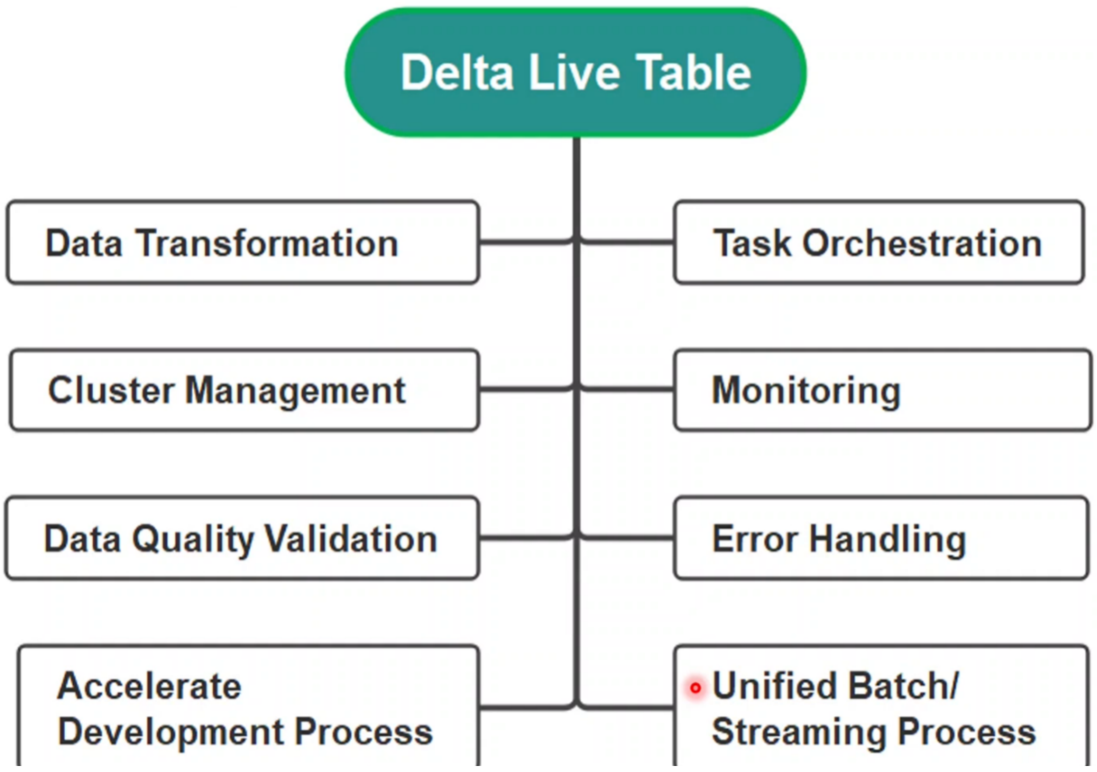
DLT is a development framework provided by databricks which simplifies and accelerates ETL pipeline building process.

DLT follows declarative approach to build pipeline instead of procedural approach.

DLT framework automatically manages the infrastructure at scale.

With the help of DLT, engineers can focus more on only developing solution than spending time on tooling.

Delta Live Tables (DLT) is a declarative framework on Databricks for building **reliable, maintainable, and testable data processing pipelines**. It simplifies the Extract, Transform, Load (ETL) process by allowing data engineers and analysts to focus on defining the data transformations using SQL or Python, while DLT automatically manages the operational aspects.



Data Transformation

It will perform the data transformation internally (it will manage all the transformation internally)

We must just declare what kind of transformation (what kind of output we need) we need, will think internally (delta engine) provide the best approach and apply transformation.

Cluster management

We don't need to create or manage any cluster we have to just create the delta live table workflow internally it will manage the cluster management.

Data Quality and Validation

We have to use different logics to handle null or duplicate for data quality checks or data validation, we have to perform manually through coding.

Coming to this declarative approach we can just tell what kind of data quality we are expecting.

(for example: one of the particular column that should not accept null value or one column it should not accept any value greater than certain limit)

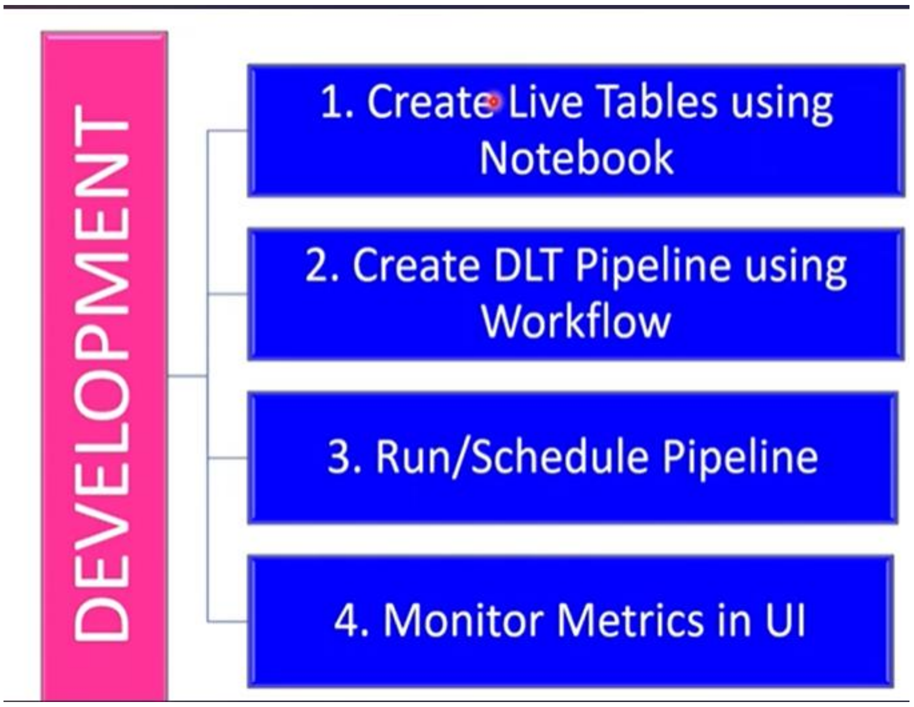
Monitoring

It is important component of DLT. Monitor through UI.

We can monitor each component (entire pipeline) in the UI.

Error handling: Some options are their like, retry (in procedural approach in case one of the particular pipeline failed then we have to rerun the entire logic from start). But DLT gives the efficient options to handle the error.

Step to follow DLT



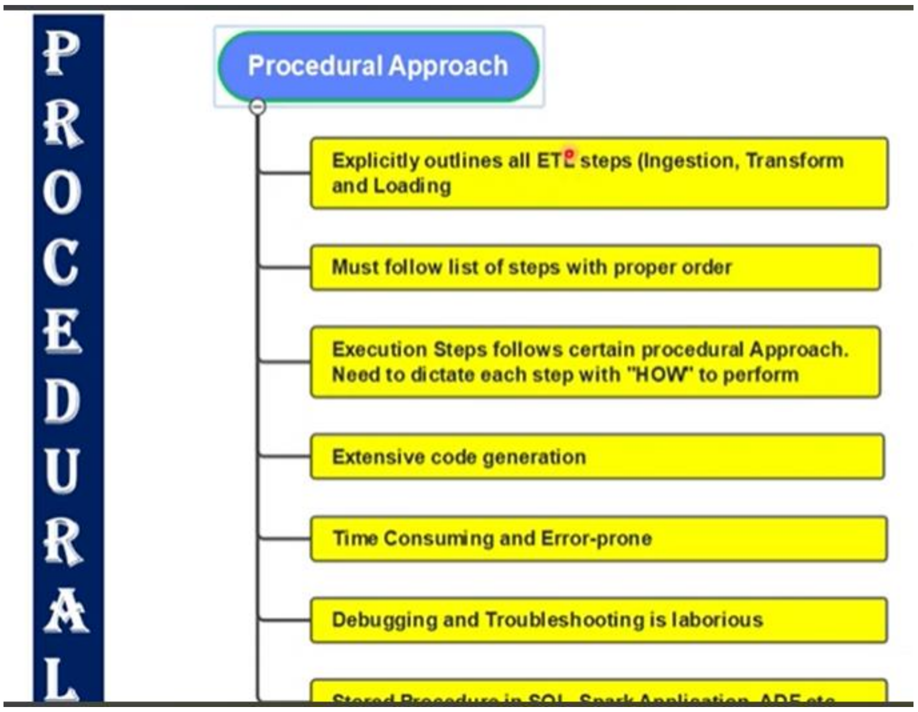
DEVELOPMENT

1. Create Live Tables using Notebook
2. Create DLT Pipeline using Workflow
3. Run/Schedule Pipeline
4. Monitor Metrics in UI

Procedural VS Declarative

DLT uses Declarative approach.

<u>Procedural approach</u>



PROCEDURAL

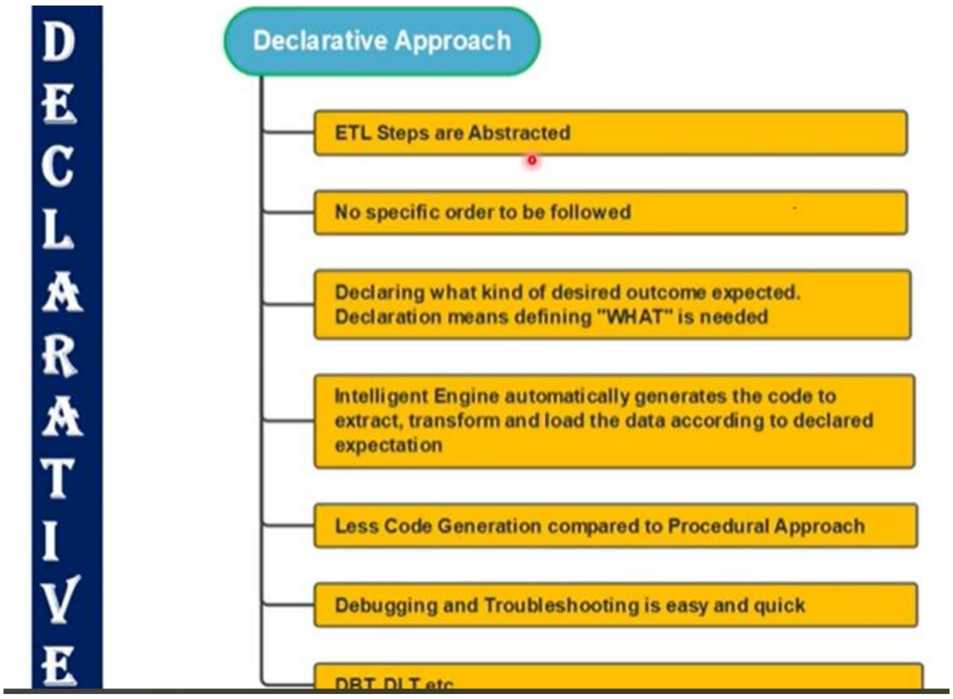
Procedural Approach

- Explicitly outlines all ETL steps (Ingestion, Transform and Loading
- Must follow list of steps with proper order
- Execution Steps follows certain procedural Approach. Need to dictate each step with "HOW" to perform
- Extensive code generation
- Time Consuming and Error-prone
- Debugging and Troubleshooting is laborious
- Stored Procedure in SQL, Spark Application, ADF etc

Declarative approach

We declare what kind of final output we are looking for; all other step is hided.

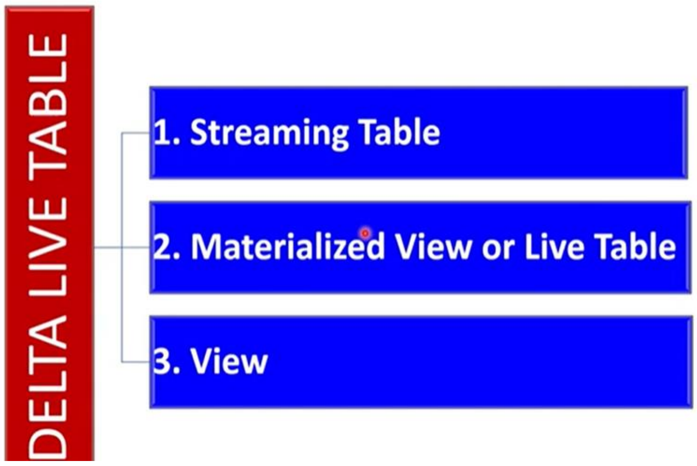
Engine will think on behalf, and it will come up with best plan, so all the ETL steps are abstracted and there is no specific order to be followed. Just we can declare what are the items we need.


DECLARATIVE
Declarative Approach
ETL Steps are Abstracted
No specific order to be followed
Declaring what kind of desired outcome expected. Declaration means defining "WHAT" is needed
Intelligent Engine automatically generates the code to extract, transform and load the data according to declared expectation
Less Code Generation compared to Procedural Approach
Debugging and Troubleshooting is easy and quick
DBT DLT etc

DLT Dataset: Tables and views

DLT table

1. Streaming table
2. Materialized view or live table
3. View



Streaming table

Supporting incremental and streaming ingestion from append-only sources.

Each record is processed exactly only once.

Auto loader is used to automatically detect and incrementally process new files as they arrive.

Cloud storage and message queues are used in streaming ingestion with auto loader.

When to use streaming table:

High volume data of rapid growth with low latency.

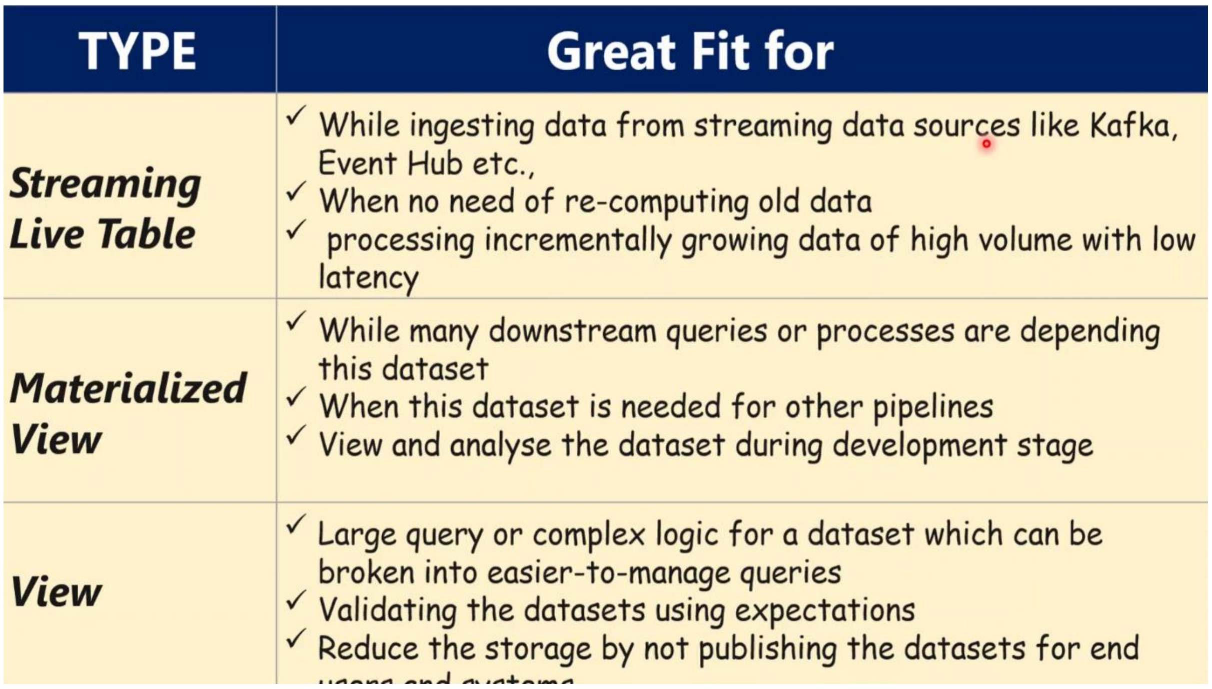
No re-computation of old data needed.

Suitable for extraction stages.

<u>Materialized view</u>

What is materialized view?

When we are defining one query



TYPE	Great Fit for
Streaming Live Table	✓ While ingesting data from streaming data sources like Kafka, Event Hub etc., ✓ When no need of re-computing old data ✓ processing incrementally growing data of high volume with low latency
Materialized View	✓ While many downstream queries or processes are depending this dataset ✓ When this dataset is needed for other pipelines ✓ View and analyse the dataset during development stage
View	✓ Large query or complex logic for a dataset which can be broken into easier-to-manage queries ✓ Validating the datasets using expectations ✓ Reduce the storage by not publishing the datasets for end

https://youtu.be/PIFL7W3DmaY?si=Gf3N_N-3fPdujmdK