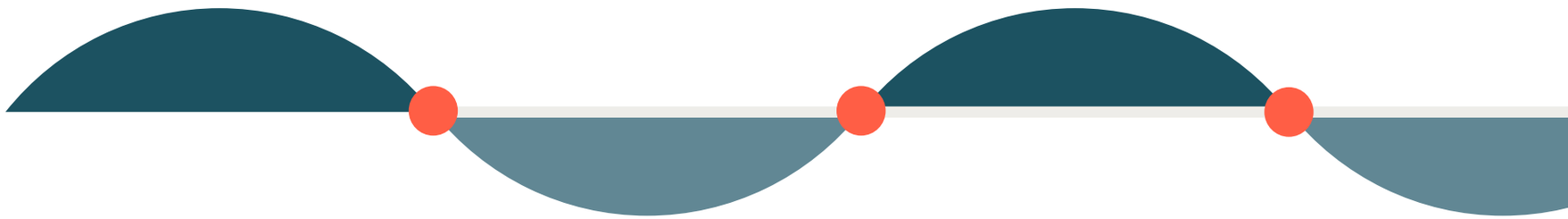
eBook

A Guide to Building ETL Pipelines With SQL

databricks

Contents

Contents



Introduction

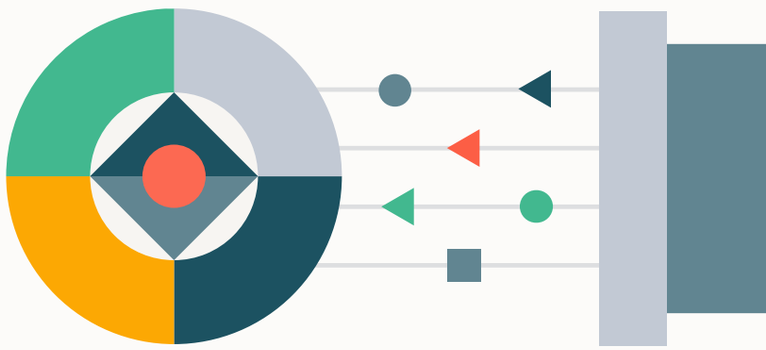
The evolving landscape of ETL

The needs for ETL (extract, transform, load) are changing rapidly. Data and AI teams are under pressure to reduce complexity, deliver data products faster and support real-time use cases — often with fewer engineering resources. As organizations face these mounting pressures, traditional approaches to ETL are showing their limits.

Analytics projects are multiplying. Data volumes are exploding. Business stakeholders want fresher insights, faster. Many teams still depend on specialized Apache Spark™ developers to bring logic into production, leaving SQL practitioners like analysts and analytics engineers stuck in a bottleneck of handoffs and rework. And even data engineers who prefer to work in SQL often find themselves constrained by legacy warehouse ETL tools — procedural, vendor-specific and difficult to maintain at scale.

The result is a growing gap between what organizations expect from their data stack and what teams can realistically deliver. Analysts are feeling this pressure firsthand. They're being asked to build more data products, answer more complex business questions and even take ownership of parts of the data pipeline that have traditionally been managed by engineers. Too often, the tooling doesn't meet them where they are.

Spark-based development remains powerful, especially for dynamic or large-scale workloads. But it comes with real barriers. PySpark and Scala require language shifts, custom environments and orchestration frameworks that create friction. For SQL-native users, these frameworks are often too complex and heavyweight for the kinds of transformations they need to perform.



The shift toward SQL for ETL

SQL is the most widely used language in data, and increasingly, it's the language of choice for modern ETL.

This shift is more than a preference. It's a response to real-world constraints: time, resources and collaboration. SQL has long been central to ETL — from stored procedures to modern data warehouse workflows — because it's familiar, readable and standardized. Analysts use it every day, and engineers can leverage it when they don't need the overhead of a full Python workflow. Its accessibility enables rapid prototyping: analysts can develop transformations in SQL, and those definitions can be hardened and optimized by engineers without rewriting them in another language. That shared foundation shrinks time to insight and makes SQL the common language that bridges different roles and skill sets.

More and more teams are using SQL not just to analyze data, but to transform and operationalize it. This SQL-first approach allows practitioners to move faster, reduce handoffs and build pipelines that are easier to maintain and debug. Increasingly, Python is reserved for more specialized use cases like machine learning, metadata-driven logic, advanced orchestration or custom libraries, while SQL becomes the default starting point.

While the appetite for SQL-based ETL is strong, many teams still hit a wall when trying to take SQL to production.

The production gap for SQL practitioners

SQL-first professionals should be able to move their transformations into production without depending on data engineering support.

SQL drives the day-to-day work of analysts and analytics engineers. Often though, deploying SQL-based pipelines — whether to refresh dashboards, aggregate logs or enable downstream analytics — requires extra engineering effort for orchestration, scaling and reliability. These handoffs slow innovation and fragment workflows.

According to the Economist Impact study, **nearly two-thirds of organizations are fully dependent on data engineers for every aspect of pipeline creation and management**. This dependency creates bottlenecks, slows iteration and blocks analytics teams from owning their workflows end to end.

This production gap stems from three core challenges. The first challenge is the language divide that slows development velocity. Analysts typically build their initial logic in SQL, while engineers may maintain production pipelines in Python or Scala. This leads to translation errors, deployment delays and duplicate logic across teams. It also makes collaboration on shared pipelines more difficult, which ultimately slows down iteration and innovation cycles.

The second challenge is that development with imperative SQL frameworks — whether it's vendor-specific stored procedures, tools like Teradata BTEQ or hand-coded scripts — is often slow and fragile. Code becomes verbose and error-prone, with little built-in observability — no logs, metrics or operational visibility unless you build them yourself. Debugging and maintaining these tightly coupled systems is difficult, and incremental patterns like CDC and slowly changing dimensions often end up hard-coded into ETL logic. For both specialized data warehouse engineers and big data teams, this adds significant complexity and slows delivery — a recurring pain point across industries.

The third challenge is that streaming remains too difficult for most teams to implement effectively. Most tools force an upfront choice between batch and streaming, leading to rigid architectures or complex lambda setups with separate pipelines that drift out of sync. If needs change later, teams often face a costly rebuild instead of a seamless upgrade to streaming.

Taken together, these limitations lead to slower iteration cycles, fractured workflows between team members and a more difficult path toward implementing real-time analytics capabilities.

Bridging the gap: Declarative SQL for ETL

For SQL-native practitioners, this insight underscores the need for tools that allow **writing and running production-grade ETL directly in SQL.**

The solution to these challenges is a fully declarative, unified approach to SQL-based ETL — one that eliminates the brittleness of imperative logic and supports both batch and streaming processing within a single model.

Declarative SQL for ETL refers to pipelines defined through intention-driven SQL statements that let the underlying system handle execution details like orchestration, performance optimization, refresh logic and operational observability. This approach handles the operational complexity while maintaining the power and flexibility that teams need.

This means that anyone who knows SQL — whether they're an analyst, analytics engineer or data engineer — can build production-grade pipelines without additional training or tooling complexity. There's no need to choose between batch and streaming processing up front. The system handles both modes transparently, allowing the evolution from one approach to the other without rewriting the logic.

Imagine a data analyst tasked with building a new pipeline for customer event data. They're comfortable with SQL, but building the pipeline in Python would mean learning PySpark, setting up a development environment and debugging potentially flaky orchestration systems. With SQL-first ETL on Databricks, that same analyst can stand up a real-time pipeline in hours rather than days, using tools they already know.

With declarative SQL ETL, building data pipelines is fast, collaborative and scalable, with significantly less friction than traditional approaches.

WHAT THIS GUIDE COVERS

This resource is designed specifically for data professionals, particularly SQL-native data engineers and analytics engineers who are being asked to take on expanded ETL responsibilities as their organizations grow.

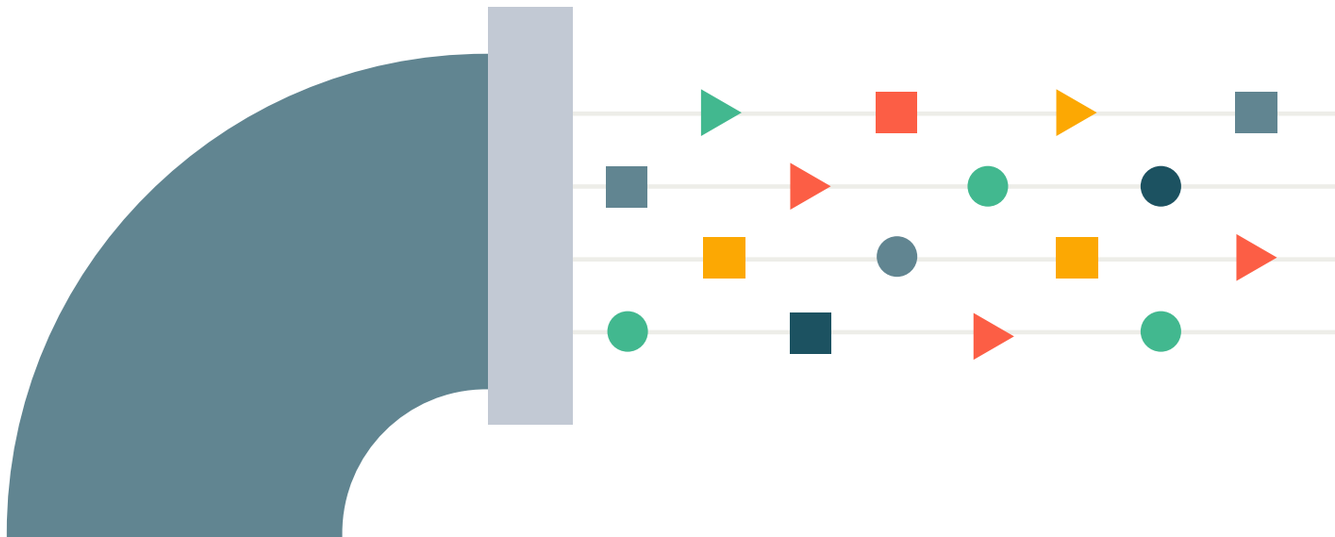
Throughout the following chapters, you'll learn how data teams can build scalable, production-grade ETL pipelines using SQL alone. This guide introduces two foundational components of the Databricks declarative approach: streaming tables and materialized views. Together, these components simplify ingestion, transformation and aggregation while supporting both batch and streaming use cases.

Other key topics include the broader shift from imperative to declarative programming, practical strategies for implementing incremental refresh patterns, simplified change data capture for tracking updates and maintaining historical records and the operational benefits of the Databricks integrated approach to orchestration.

By the end, you'll understand how to build fast, reliable and collaborative ETL pipelines using just SQL — without requiring Python expertise, separate orchestration tools, stored procedures or deep Spark knowledge.

A NOTE ABOUT NO CODE

This guide isn't about no code. It's for SQL-native professionals — analysts, analytics engineers, data engineers and others — who already work in code but get stuck waiting for someone else to productionize their work. Many existing no-code tools focus on building pipelines quickly, but stop short of giving you the control and reliability you need to run them in production. That's where handoffs to data engineering teams slow everything down. The solution isn't abandoning SQL or switching tools. It's removing the friction that keeps SQL practitioners from taking their work all the way to production themselves.



From Procedural to Declarative ETL

Data teams are stuck. Analysts can write the SQL code for an insight in minutes, but turning that logic into a production pipeline often takes days or weeks. That's true whether the code is in Python or Scala, or buried in 100,000 lines of vendor-specific procedural SQL — like stored procedures or Teradata BTEQ scripts. These approaches are verbose, brittle and heavily dependent on specialized engineering skills.

Declarative, SQL-based ETL closes this gap. It's not just "ETL in SQL" — it's a different way of working that focuses on *what* you want to happen, not *how* to make it happen. You can take the same SQL you already know for analysis and use it directly to build production-ready pipelines — without engineering handoffs or translating logic into other languages.

This is more than a syntax change — it's a fundamental shift in how pipelines are built and maintained. But what does "declarative" actually mean in practice? And how does it differ from the procedural approaches that have dominated ETL for decades?

Imperative vs. declarative data processing

Most data teams face a fundamental choice when building pipelines: imperative or declarative approaches. This decision shapes everything from development speed to team collaboration.

Imperative pipelines follow explicit, step-by-step instructions. You define exactly how data should be processed, when each transformation should run and how the system should handle every edge case.

This imperative approach — often called *procedural* in Databricks contexts — assumes the system needs constant guidance. You spell out every operation, manage state manually and orchestrate each step in sequence. Think of it like writing a cooking recipe that specifies not just the ingredients, but exactly how to hold the spoon and which direction to stir.

It's how most traditional pipelines work today. Line-by-line logic, explicit state management and procedural flow give you complete control over execution — but at a cost. Code becomes brittle, verbose and difficult to maintain. Small changes often require rewriting entire sections.

Consider this example of calculating a running total. The imperative approach requires you to manage cursors, track fetch status and manually accumulate values:

```sql
WHILE @@FETCH_STATUS = 0
BEGIN
    SET @Total = @Total + @Value;
    FETCH NEXT FROM cur INTO @Value;
END
SELECT @Total;
```

Declarative programming takes the opposite approach. Instead of spelling out every step, you describe the outcome you want. The system figures out how to get there.

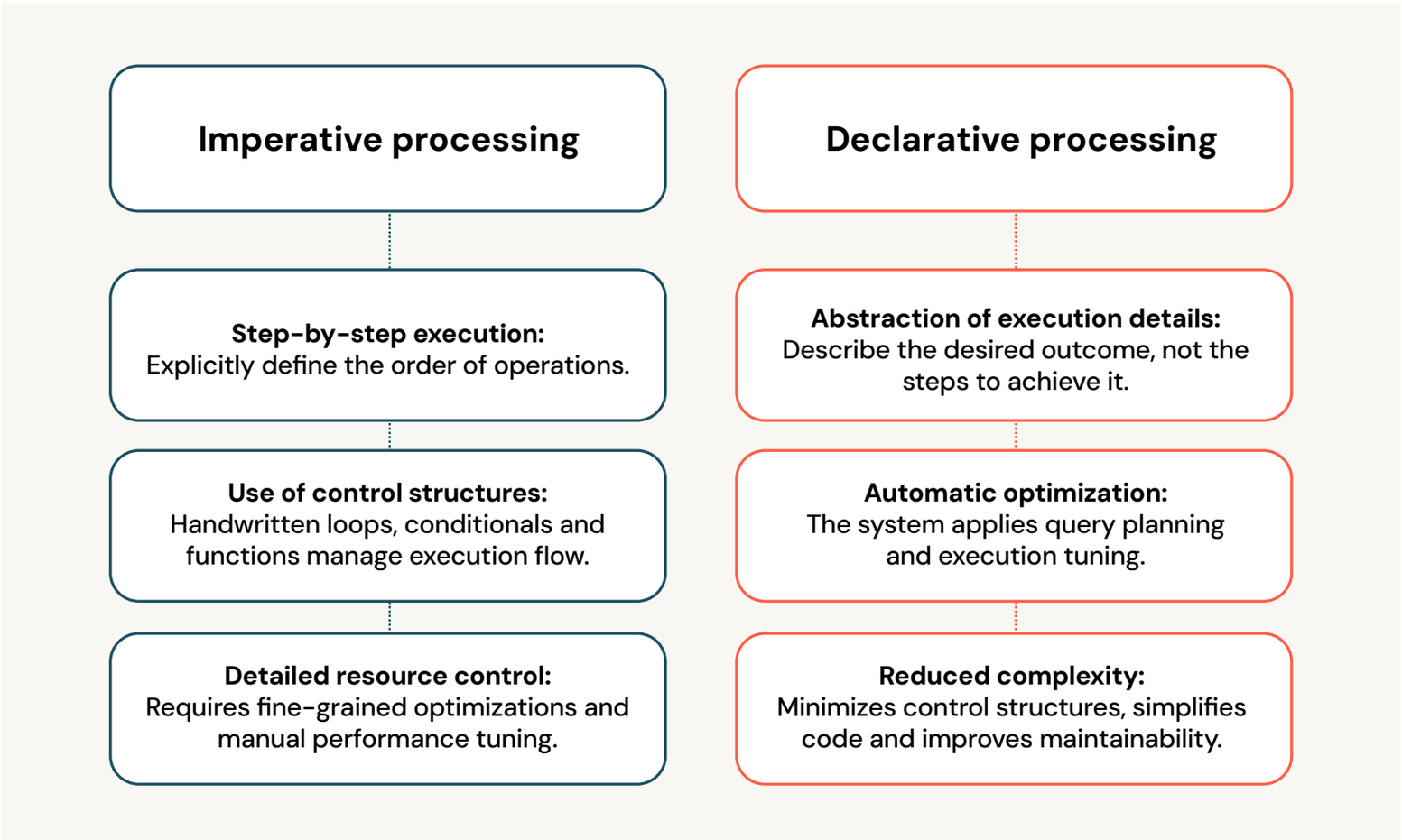
See the declarative version of the same task:

```sql
SELECT
    sum(1)
from
    <source>
```

Now imagine you're doing this to calculate daily sales across billions of transactions from hundreds of stores — the gains in simplicity and maintainability are dramatic.

Imperative code dictates "loop through each row, check this condition, accumulate that value." A declarative approach asks "give me the sum." Both get the same result, but one trusts the system to optimize the path.

This isn't just about writing less code — it's about writing more maintainable code. When business logic changes, you update the desired outcome, not the implementation details. The system handles the rest.



But what about control? What about performance? What about those edge cases that require custom logic? What about logging and monitoring? These are valid concerns, and declarative approaches have evolved to address them without sacrificing the simplicity that makes SQL so powerful. Declarative SQL lets you write less code, make fewer mistakes and rely on the platform to optimize and operate under the hood.

Now let's look at how declarative SQL powers the medallion architecture without procedural overhead.

Declarative SQL pipelines: From Bronze to Gold

"SQL is the language of data. We're now a SQL-first organization, and utilizing materialized views, we're able to build real-time, production-grade pipelines. It's faster to develop, performs much better than our legacy Python pipelines and is dramatically more efficient. By changing our focus to declarative SQL materialized views and away from heavy, procedural Python dataframe-based code, we can process data incrementally in minutes as opposed to hours."

— Sharon Bjeletich, Senior Principal Data Architect, Insulet

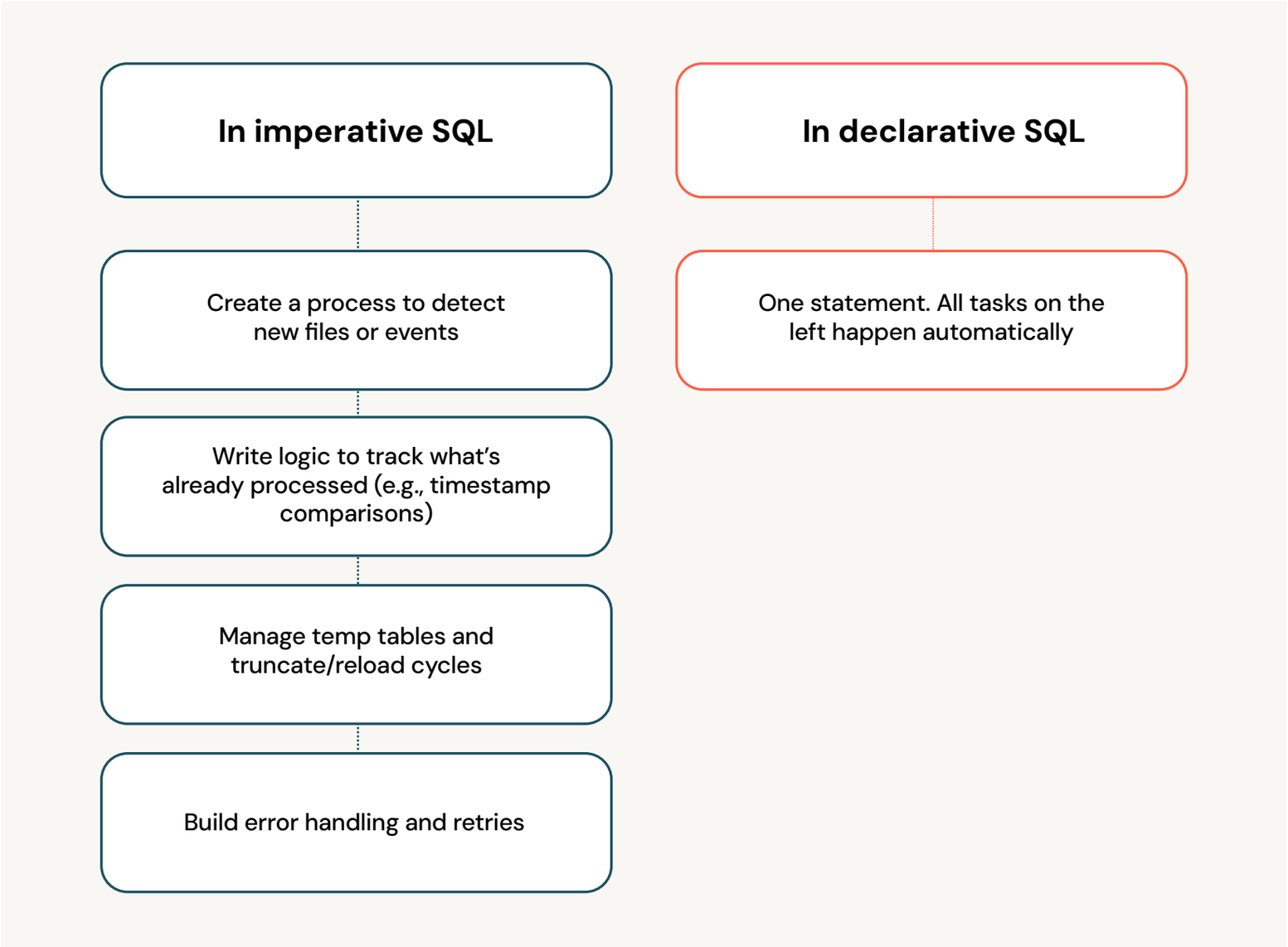
This declarative approach transforms how you build and maintain production pipelines. Instead of switching between SQL for analysis and Python for production, you can now manage complex workflows using the same SQL skills you already have — no new languages, no orchestration tools, no handoffs required.

Here's a simple example of what this looks like in practice. You write two statements. The first creates a streaming table that ingests JSON files from cloud storage — automatically handling event detection, exactly-once processing and state management. No separate listener jobs, no temp tables to truncate and reload, no custom timestamp logic to track what's new — it's all built in:

SQL

```sql
CREATE OR REFRESH STREAMING TABLE raw_data
AS SELECT *
FROM STREAM read_files('/raw_data', format =>'json')
```

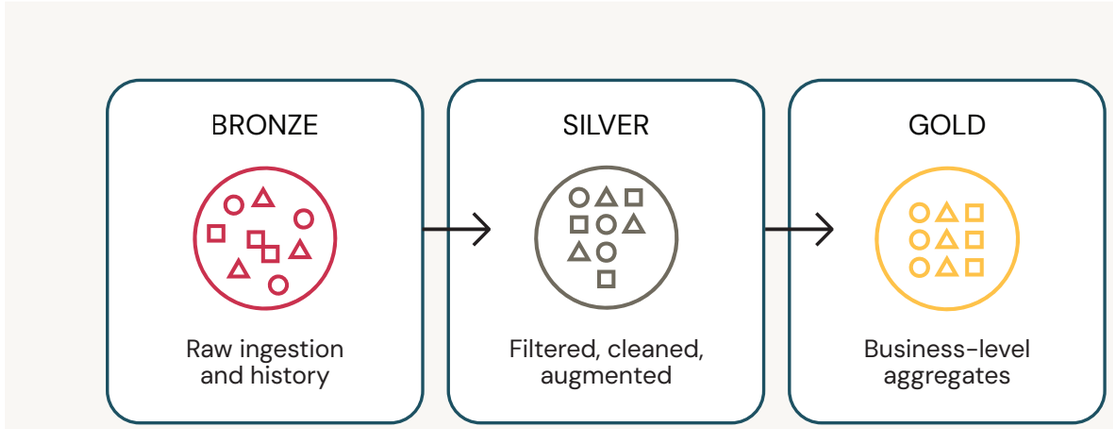
Consider how much complexity this statement eliminates.



The second statement defines a materialized view to transform that raw data. Both use familiar SQL syntax:

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

This maps naturally to how most teams already think about data: Raw files become clean tables, clean tables become business-ready views. The medallion architecture gives this progression a name — Bronze for raw ingestion, Silver for cleaned data, Gold for business aggregates.



But here's what's different: Each layer runs automatically. Your streaming table ingests new files as they arrive. Your materialized view refreshes when underlying data changes. The system handles scheduling, error recovery and optimization.

This simplicity extends to maintenance and collaboration. Each layer is declaratively defined in SQL, making the entire pipeline accessible to anyone fluent in SQL. Anyone who knows SQL — whether they're an analyst or an engineer — can build and own production-grade pipelines. No translation overhead between roles. Logic changes mean updating your queries, not your infrastructure. Source changes mean pointing to new tables, not rebuilding pipelines.

The approach is also flexible by design. Traditional pipelines force you to choose batch or streaming up front. Declarative SQL handles both transparently. Start with batch processing, scale to streaming later — no code changes required.

You might wonder: What about performance? Declarative SQL approaches can deliver intelligent optimizations — from vectorized execution, adaptive caching and cost-aware query planning to parallel task scheduling and automatic file layout optimization — capabilities that once required extensive hand-tuning. Simplicity and performance no longer compete.

How Databricks enables declarative SQL

The declarative approach we've outlined becomes concrete through purpose-built components. Databricks enables this SQL-first approach with two building blocks designed specifically for declarative ETL: streaming tables and materialized views.

In the next chapter, you'll explore both tools, learning how streaming tables and materialized views work together to enable SQL-native engineers to build production-ready pipelines.

The Building Blocks of Declarative ETL

Declarative SQL bridges the production gap we explored in Chapter 1. But how does this actually work in practice? What makes it possible for SQL practitioners to build production-grade pipelines without switching languages or waiting for engineering handoffs?

The answer lies in two foundational components that work together seamlessly:

- **Streaming tables:** Handle ingestion and basic transformation of raw data, including real-time and incremental sources
- **Materialized views:** Perform more complex transformations, aggregations and serve as the optimized layer for consumption — powering dashboards, ML models and downstream analytics

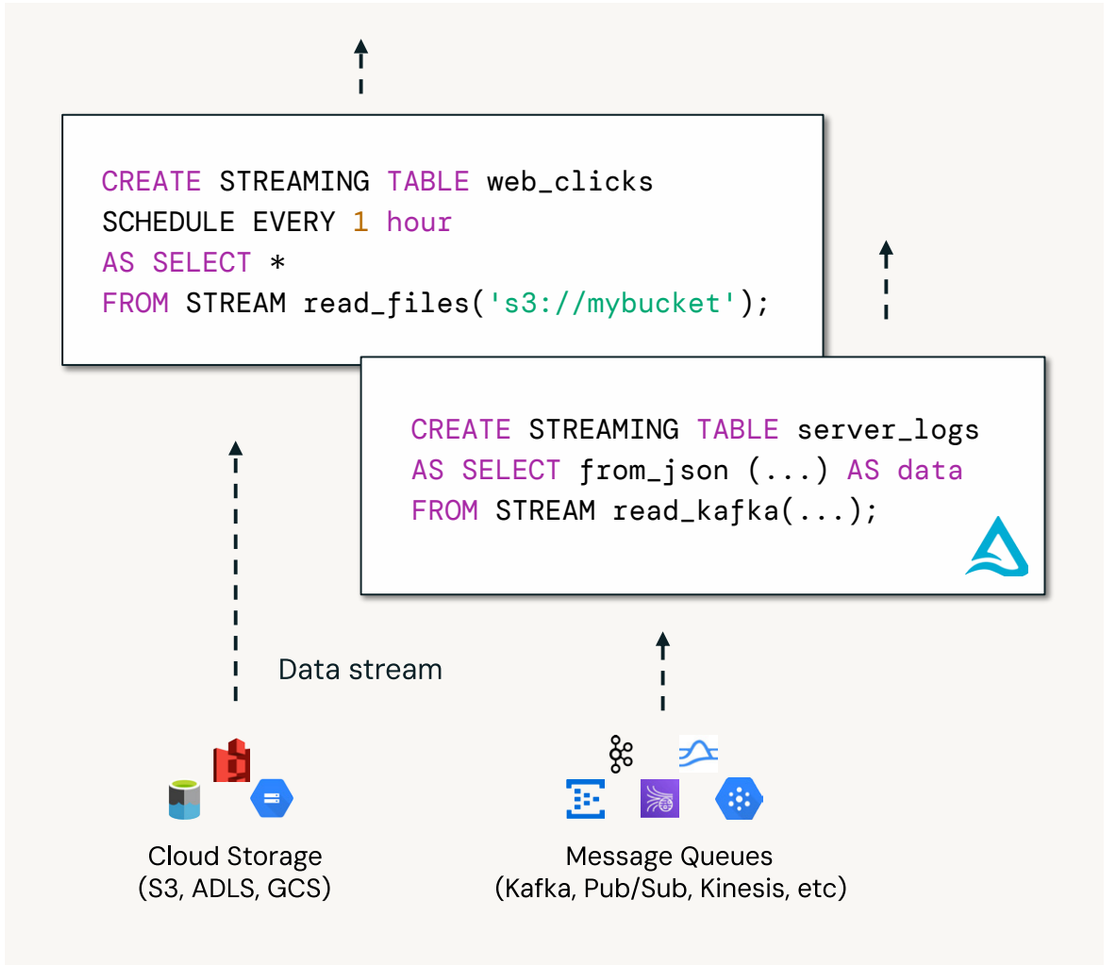
Together, these components enable you to build complete pipelines using only SQL. No language switching. No handoffs between teams. Just declarative statements that describe what you want, while the system handles the complexity of how to get there.

Let's start with streaming tables and how they make both incremental batch and real-time data processing accessible to every SQL practitioner.

Streaming tables: Making incremental batch and real-time data processing accessible

Real-time and incremental data processing has traditionally been the exclusive domain of specialized engineers. The complexity of managing streaming sources, handling back pressure and ensuring fault tolerance has kept most SQL practitioners in batch-only territory.

Streaming tables address these traditional barriers by making real-time processing accessible through familiar SQL. Whether you're pulling from cloud storage (Amazon S3, ADLS, GCS) or message queues like Kafka or Azure EventHub, it's just a `CREATE STREAMING TABLE` statement:



That's it. No Spark configuration. No checkpoint management. No complex error handling. No additional infrastructure to watch for new files or events. You describe what you want, and the system handles the rest — scheduling, checkpointing, incremental ingestion — all under the hood. In a traditional SQL warehouse, achieving the same result might mean polling storage for new data, loading into staging tables and running truncate/reload cycles — all orchestrated by external jobs. Here, it's a single statement.

This works across a wide range of sources, including Amazon S3, ADLS, GCS, Kafka, Pub/Sub and Kinesis. The syntax stays familiar, regardless of the underlying complexity.

Compare this to traditional streaming approaches, where the same functionality might require dozens of lines of PySpark code, custom error handling and deep knowledge of distributed systems. With streaming tables, all that complexity is handled automatically while you work with standard SQL.

KEY BENEFITS OF STREAMING TABLES

- **More practitioners can build streaming pipelines:** You no longer need to write complex Spark or Apache Flink code to build a real-time pipeline — just use the SQL you already know
- **Better scalability through incremental processing:** Streaming tables handle high volumes via incremental updates, delivering lower latency and reduced costs compared to large-batch processing
- **Real-time use cases are now within reach:** Power real-time dashboards, ML feature pipelines, or operational alerting — all without leaving SQL or requiring specialized expertise

Ingestion is only the first step. Once data flows into your lakehouse, you need to transform and optimize it for consumption. Materialized views handle this transformation layer efficiently.

Materialized views: Precomputing without the pain

"At Danske Spil we use materialized views to speed up the performance of our website tracking data. With this feature we avoid the creation of unnecessary tables and added complexity, while getting the speed of a persisted view that accelerates the end user reporting solution."

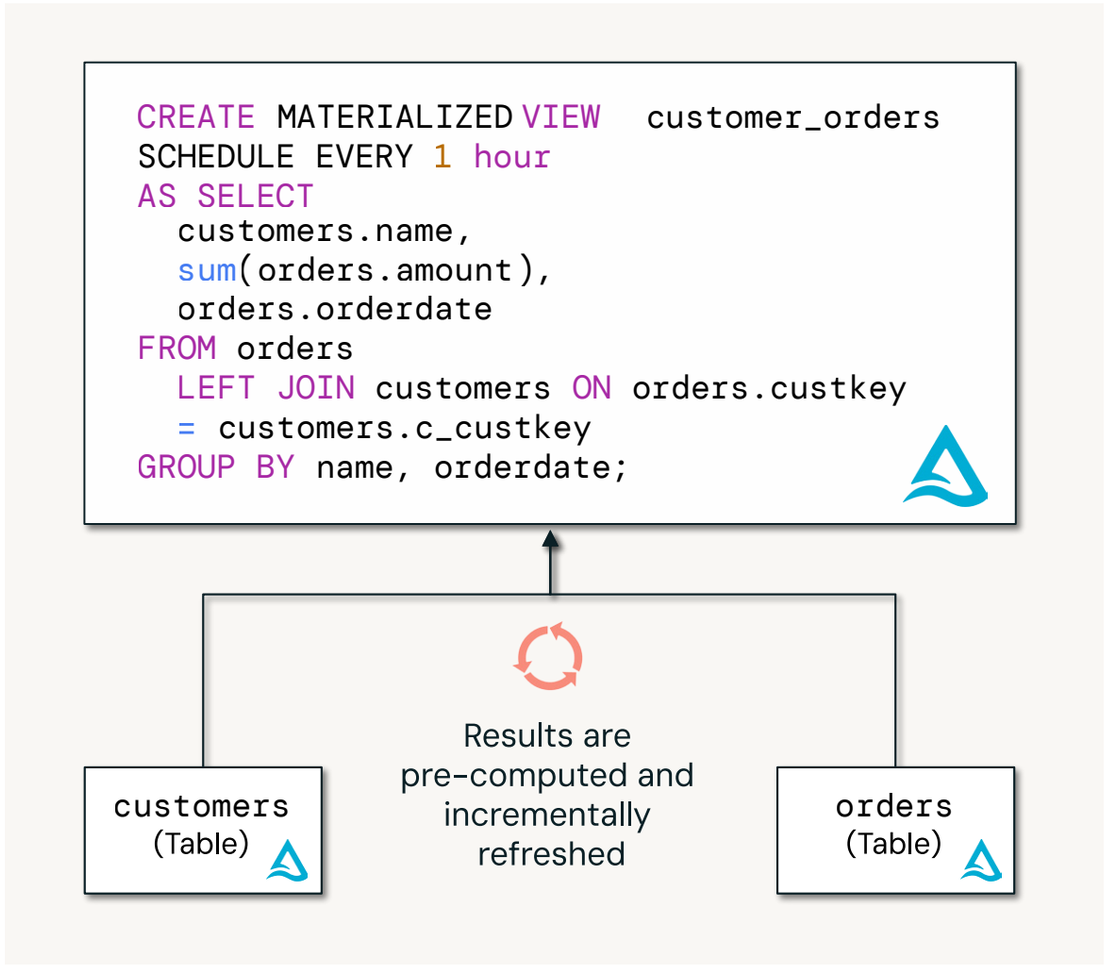
— Søren Klein, Data Engineering Team Lead, Danske Spil

Dashboard performance is a persistent pain point for SQL practitioners. Large BI applications query complex views over massive datasets, leading to slow response times for users and high compute costs from repeatedly scanning raw data.

Traditionally, teams have worked around this by manually creating and maintaining precomputed tables. But that approach brings its own challenges: stale data, fragile refresh logic and rising maintenance overhead as systems scale.

Materialized views solve these problems on multiple fronts. They accelerate queries by precomputing results, improve freshness through incremental refreshes that avoid full recomputes and simplify ETL by letting you define transformations declaratively in SQL — no pipelines or manual scheduling required.

Consider a typical scenario where you need to aggregate customer order data for a dashboard. You define a materialized view using SQL, join your tables, add transformations like aggregations and schedule it to refresh periodically:



This materialized view joins customers and orders, then computes total order amounts per customer per date. Instead of rerunning that aggregation every time someone opens a dashboard, the results are incrementally maintained and ready to go.

KEY BENEFITS OF MATERIALIZED VIEWS

- **Faster dashboards** — When querying precomputed data vs. base tables
- **Improved data freshness** — Through incremental refreshes that avoid time-consuming full recomputes
- **Simplified ETL** — With declarative SQL, scheduling and refresh logic are built into the definition, reducing the need for separate orchestration jobs

PROVEN PERFORMANCE IMPACT

"Utilizing materialized views on top of transaction tables has drastically improved query performance on our analytical layer, with the execution time decreasing up to 85% on a 500 million–fact table."

— Nikita Raje, Director of Data Engineering, DigiCert

"The conversion to materialized views has resulted in a drastic improvement in query performance, with the execution time decreasing from eight minutes to just three seconds. This enables our team to work more efficiently and make quicker decisions based on the insights gained from the data. Plus, the added cost savings have really helped."

— Karthik Venkatesan, Security Software Engineering Sr. Manager, Adobe

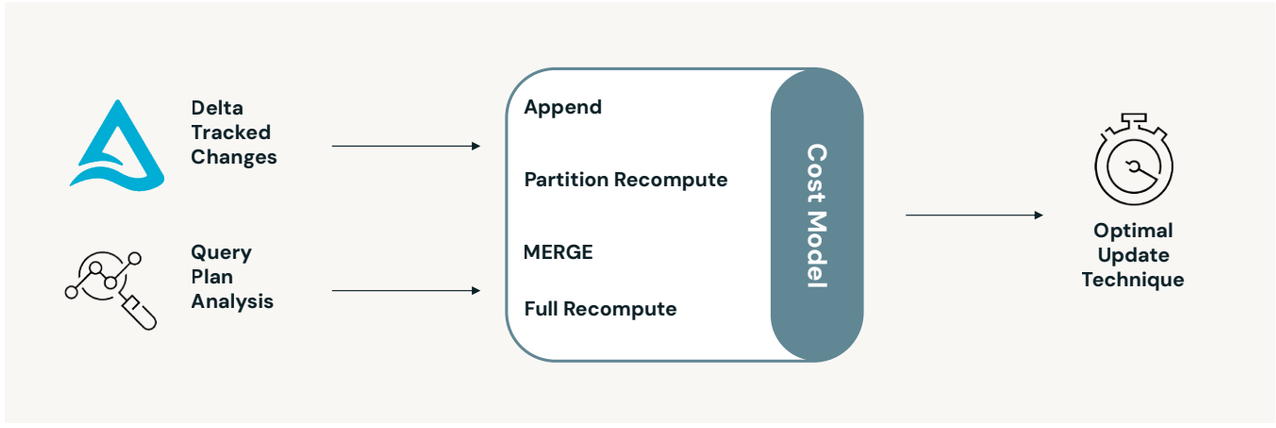
These performance improvements extend across different scales and use cases. At AnyClip, a visual intelligence company that processes millions of videos and terabytes of data, materialized views delivered even more dramatic results. “We’ve seen query performances improve by 98% with some of our tables that have several terabytes of data,” says Gal Doron, Head of Data at AnyClip. “Previously when users tried to run queries, it could take hours. With materialized views, it takes something between half a minute to three minutes.”

Behind these performance gains are the intelligent refresh capabilities built into materialized views. Understanding how these capabilities work helps explain why materialized views can deliver both performance and freshness without the traditional trade-offs.

Under the hood: Automatic incremental refresh of materialized views

Materialized views aren’t just simple — they’re smart. They understand your query, analyze your data and keep results fresh automatically, even for complex logic.

At the core of this capability is an intelligent cost model. It takes two inputs: the table’s change data feed a nd the query plan of your materialized view definition. With that information, the system automatically picks the most efficient refresh strategy — whether that’s a lightweight append, a partition-level recompute, a `MERGE` operation and so on.



The key insight? You don't have to choose. The system picks the optimal path for you.

This automatic optimization delivers two major benefits:

- **Fresher data**, because smaller updates can be applied more frequently
- **Lower costs**, because expensive full recomputes are avoided unless absolutely necessary

The entire process runs automatically — no triggers, scheduling logic or manual refresh scripts required.

To get the most from automatic incremental refresh, there are a few best practices worth following. These ensure that the system can optimize your queries effectively and refresh them incrementally whenever possible.

Best practices for incremental refresh

To maximize the effectiveness of automatic incremental refresh:

- **Use supported operators:** materialized views are automatically incrementally refreshed if their queries support it. If a query includes unsupported expressions, a full refresh will be done instead. Check the Databricks documentation for supported operations.
- **Enable row tracking on source tables:** This allows the system to identify which rows have changed since the last refresh
- **Enable deletion vectors on source tables:** This limits changeset size by efficiently tracking deletions
- **Proper clustering on grouping and join keys:** Well-clustered source tables reduce the amount of data that needs to be processed during refresh

The system can incrementally process a wide range of queries while still benefiting from automatic optimization. Supported operations include most common join types — left outer, full outer and inner joins — as well as window functions like row numbers and lag/lead. Even more complex patterns, such as a JOIN followed by an aggregation, are often eligible for incremental refresh.

Easily ingesting and transforming data with streaming tables and materialized views

Streaming tables and materialized views become transformative when they work together. This combination enables you to build continuous ingestion and transformation flows using only SQL.

Take a typical use case: Data arrives continuously in a cloud storage bucket, and you need to ingest and transform it in real time. The solution is straightforward. Create a streaming table to land incoming data into your lakehouse. Then add a materialized view to transform and aggregate that data — like counting ingested rows or calculating running totals.

Step 1: Create a streaming table to ingest data from a volume once an hour. The streaming table ensures exactly-once delivery of new data. Because streaming tables use serverless background compute, they automatically scale to handle spikes in data volume.

```sql
CREATE OR REFRESH STREAMING TABLE my_bronze
AS SELECT
    date,
    event_id
FROM STREAM read_files('/Volumes/bucket_name')
```

Step 2: Create a materialized view to transform data every hour. In this case, it's scheduled to refresh every hour, ideal for batching updates to balance cost and freshness. When possible, the system refreshes incrementally.

```sql
CREATE OR REPLACE MATERIALIZED VIEW my_silver
AS SELECT
    date,
    count(distinct event_id) AS event_count
FROM my_bronze
GROUP BY ALL;
```

This two-step process shows how streaming tables and materialized views integrate — connecting raw ingestion with transformation. It's the foundation of building full ETL pipelines, where additional layers refine data into trusted, business-ready results.

Setting the stage for production

You now have the two core components for SQL-first ETL, but having the right components is just the beginning. How do you actually implement these streaming tables and materialized views in production? What approaches work best for different team structures and use cases?

Two distinct implementation paths exist with different trade-offs: declarative pipelines for full-featured, production-grade workflows, and standalone streaming tables and materialized views in SQL warehouses.

Next, we'll explore both approaches through real-world examples. You'll see how teams choose between these options and implement them successfully across different organizational contexts.

Two Paths to Production: Building with Streaming Tables and Materialized Views

You understand the power of declarative SQL. You've seen how streaming tables and materialized views can transform your ETL workflows. Now comes the practical question: How do you actually implement this in your organization?

Databricks offers two distinct paths — each tailored to different team structures and maturity levels. Whether you need mission-critical ETL pipelines or just fast SQL transformations, you can choose the approach that fits your team.

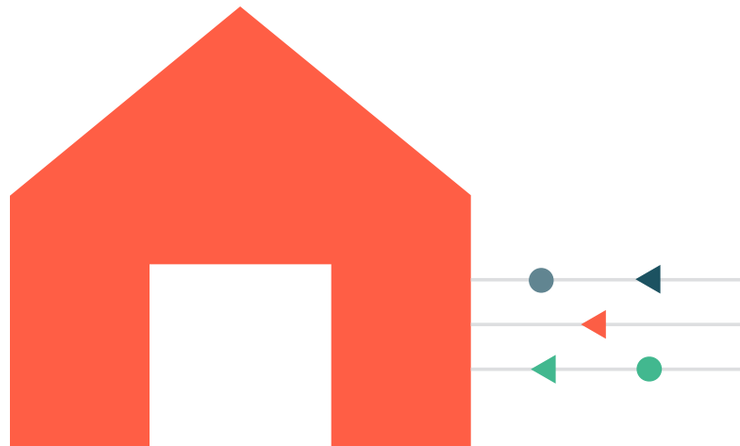
- **Lakeflow Declarative Pipelines** — A full-featured option for governed, end-to-end ETL workflows with comprehensive pipeline management and governance controls.
- **Standalone materialized views and streaming tables** — A lightweight approach that can be created and managed outside of ETL pipelines (for example, in the DBSQL warehouse), letting you work directly in SQL with no pipeline framework required.

Both paths use the same underlying engine. Both deliver the same performance benefits. The difference lies in operational complexity and governance requirements.

Full-scale ETL with Lakeflow Declarative Pipelines

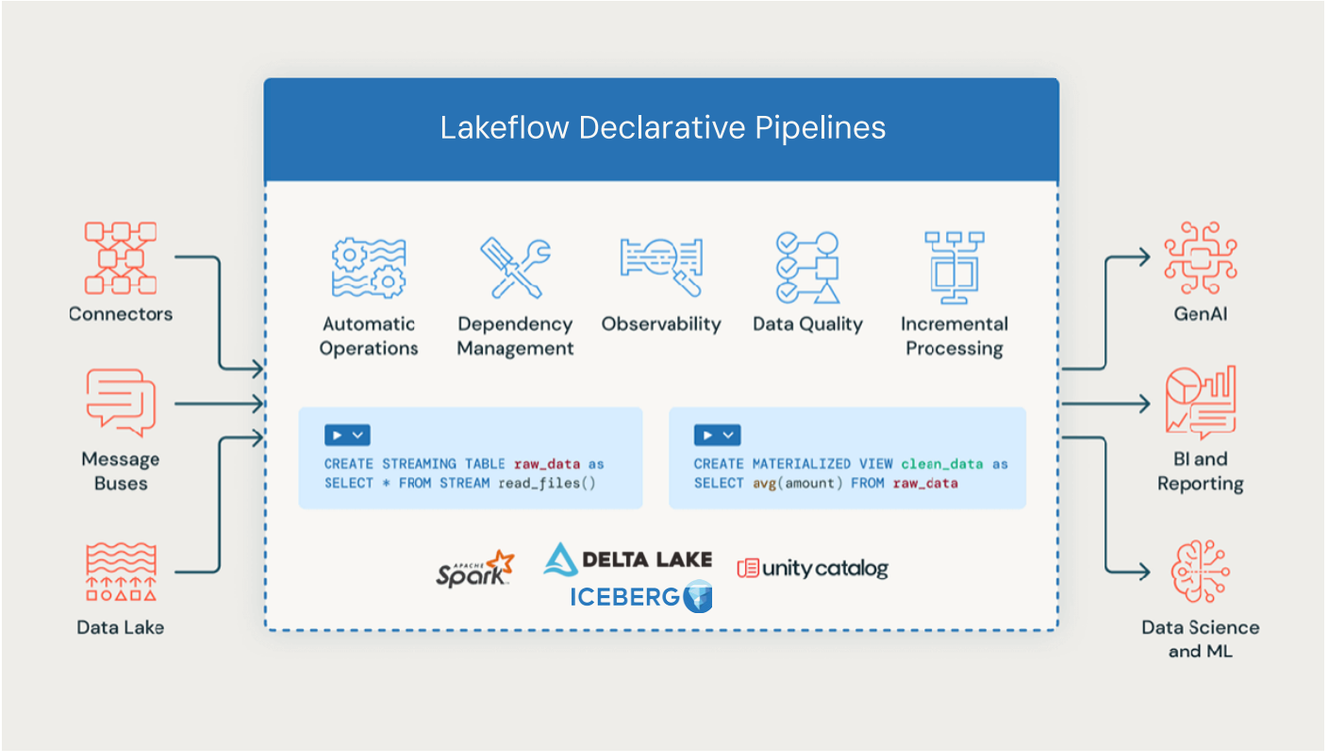
Lakeflow Declarative Pipelines brings production-grade automation to SQL-first ETL. You define transformations in familiar SQL syntax while the platform handles orchestration, dependencies and scaling.

Write your table definitions and transformation logic. The system automatically builds them into a DAG — managing streaming ingestion, incremental refresh and dependency tracking. No manual scheduling. No custom error handling. No infrastructure management.



This eliminates the operational overhead that traditionally required specialized engineering resources:

- **Automatic operations** — Such as retries, state recovery and checkpoint management
- **Built-in observability** — For tracing data flow and debugging issues quickly
- **Data quality checks** — That validate freshness and integrity at each stage
- **Incremental processing** — That scales efficiently as data volumes grow



The development pattern stays simple. You write a `CREATE STREAMING TABLE` to ingest from your data source. Then add a `CREATE MATERIALIZED VIEW` to transform that raw data. Declarative Pipelines manages the scheduling, optimization and orchestration behind the scenes.

Integration happens natively. Declarative Pipelines works directly with Delta Lake, Apache Iceberg™, Unity Catalog and Photon. The output feeds seamlessly into GenAI applications, BI dashboards or ML workflows — no extra data movement or duplication required.

This approach proves especially valuable for teams building governed pipelines, managing compliance requirements, or scaling real-time workloads across multiple data sources.

REAL-WORLD IMPACT: REMOVING ENGINEERING BOTTLENECKS

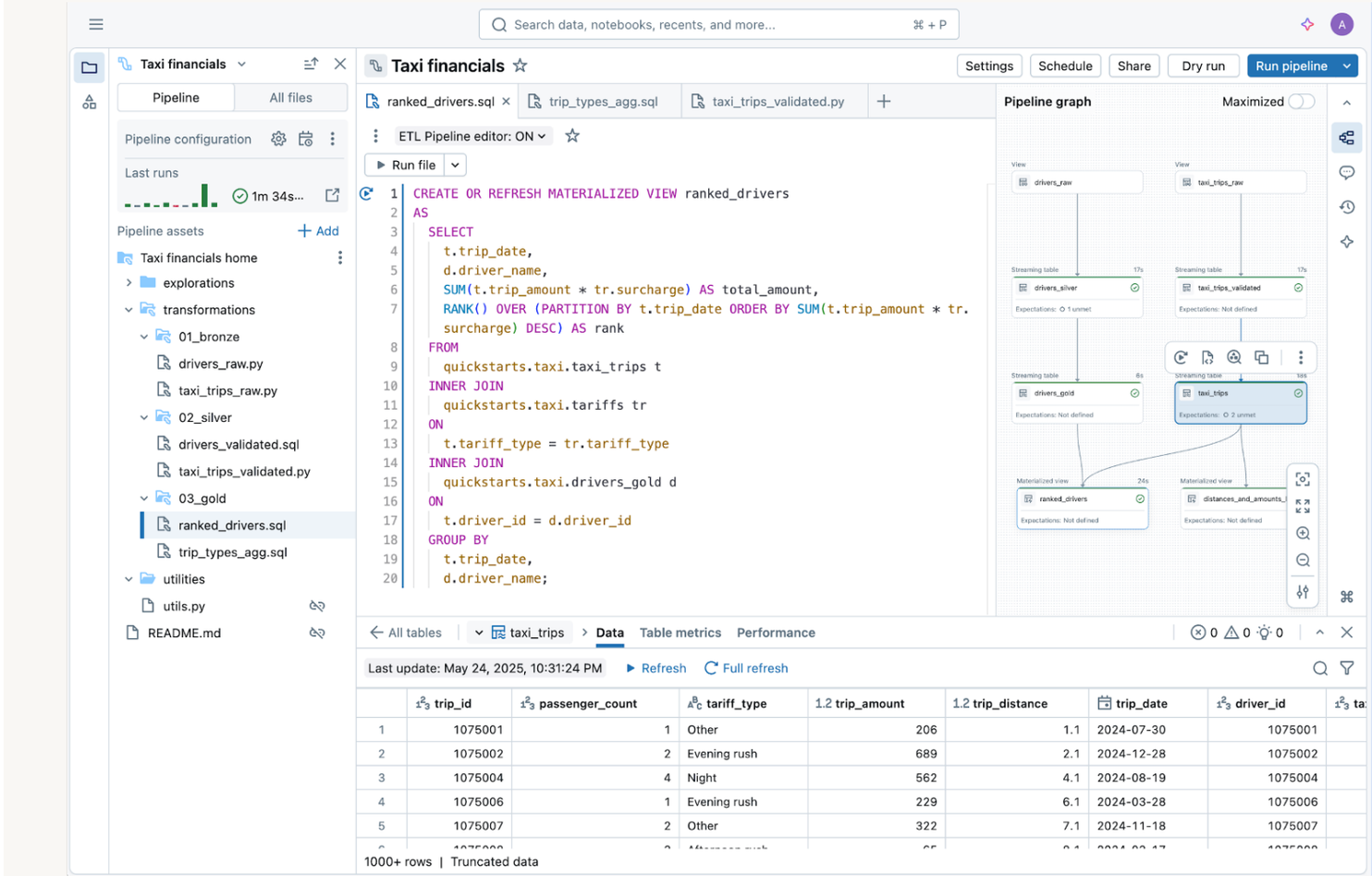
"[Lakeflow Declarative Pipelines] enables collaboration and removes data engineering resource blockers, allowing our analytics and BI teams to self-serve without needing to know Spark or Scala," explains Christina Taylor, Senior Data Engineer at Bread Finance. "In fact, one of our data analysts — with no prior Databricks or Spark experience — was able to build a declarative pipeline to turn file streams on Amazon S3 into usable exploratory datasets within a matter of hours using mostly SQL."

This is exactly what Declarative Pipelines enables — SQL practitioners building production pipelines without engineering dependencies. To help SQL users move faster, Databricks offers a purpose-built development environment.

Development tooling for simplified pipeline authoring

With the Lakeflow Declarative Pipelines editor, you get an authoring experience that brings together code, DAG, data previews and more. It's the IDE for data engineering.

The editor supports modular development — you can focus on one dataset at a time, preview transformations inline and iterate without rerunning entire pipelines. Features like automatic DAG generation and contextual error handling make it faster to move from initial logic to production deployment. Databricks Assistant is also fully integrated with the IDE, enabling you to use natural language prompts to quickly determine the right language and code.



The editor fits into your existing workflows instead of replacing them. Whether you're starting fresh or bringing over existing SQL, it makes development much faster.

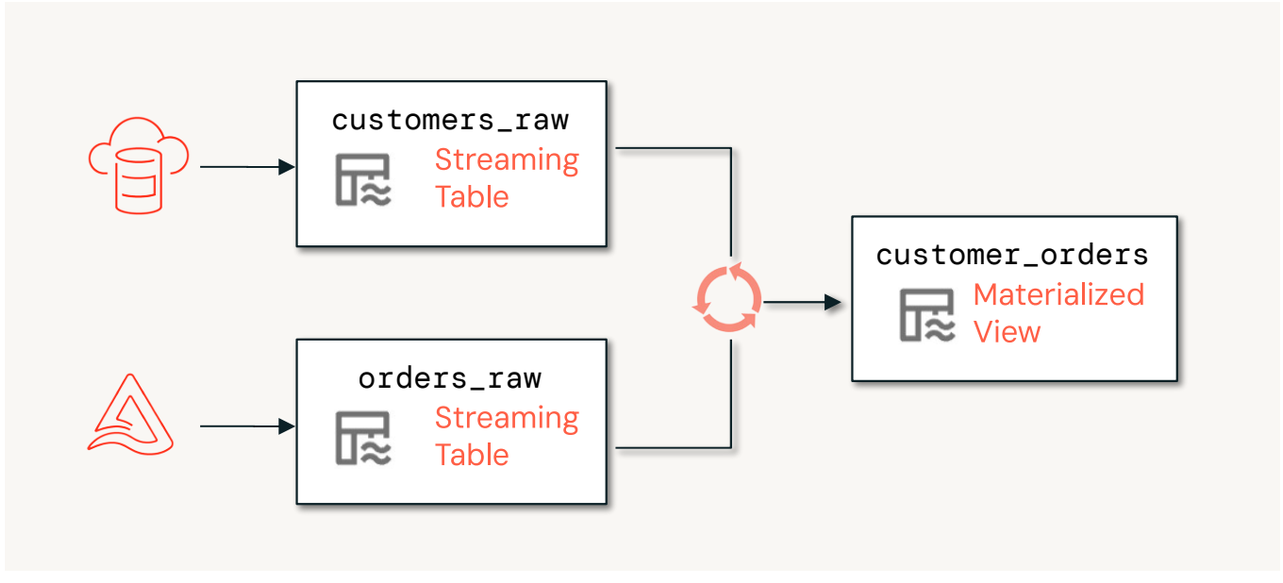
If you don't need full orchestration capabilities, there's a simpler way to leverage streaming tables and materialized views.

Standalone materialized views and streaming tables without pipelines

The second approach delivers speed and simplicity by reducing operational complexity. You define individual streaming tables and materialized views directly in DBSQL using standard SQL statements — no pipeline framework required.

This lightweight option delivers the same performance benefits as Declarative Pipelines but with less management overhead. There's no pipeline definition file, no DAG to manage and no separate orchestration layer — just SQL statements executed directly in DBSQL. You get the same intelligent refresh logic, the same incremental processing capabilities and the same automatic optimization. The difference is in scope and orchestration.

Take a typical use case: streaming customer and order data for real-time analytics. You create two streaming tables to ingest the raw data, then define a materialized view to join and aggregate results.



```sql
-- Ingest from external storage into a streaming table
CREATE OR REPLACE STREAMING TABLE customers_raw
SCHEDULE EVERY 10 MINUTES
AS SELECT * FROM STREAM read_files('s3://your-bucket/customers/', format => 'json');

-- Ingest another dataset
CREATE OR REPLACE STREAMING TABLE orders_raw
SCHEDULE EVERY 10 MINUTES
AS SELECT * FROM STREAM read_files('s3://your-bucket/orders/', format => 'json');

-- Materialized view joining the two
CREATE MATERIALIZED VIEW customer_orders
SCHEDULE EVERY 1 HOUR
AS SELECT o.*, c.* FROM orders_raw o
JOIN customers_raw c ON o.customer_id = c.id;
```

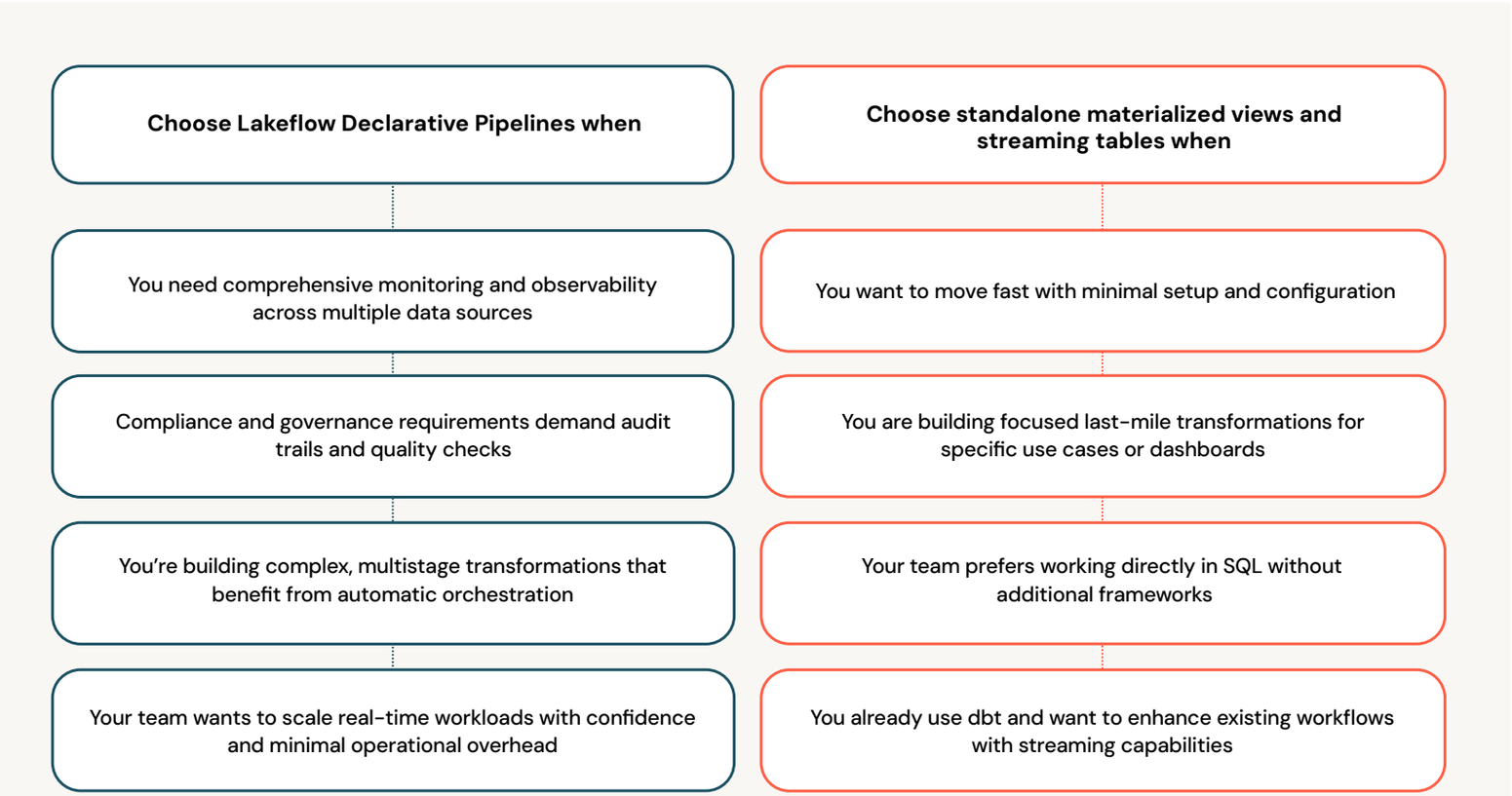
No pipeline orchestration. No dependency management. Just SQL that describes what you want, while the system handles incremental refresh and optimization automatically.

This approach works especially well for SQL analysts who want to build transformations quickly or power specific dashboards without managing complex pipelines. It's also popular with teams already using dbt — you can integrate these capabilities directly into existing workflows (see this blog for more information).

The standalone approach provides fast, flexible, SQL-only ETL without the overhead of a full pipeline framework.

When to use each approach

How do you decide between these approaches? The choice often comes down to governance requirements, team structure and operational complexity.



Teams may consider starting with the standalone approach for speed and simplicity, then migrating to Declarative Pipelines as their governance and orchestration needs grow. Since standalone materialized views and streaming tables are powered by serverless Lakeflow infrastructure, this evolution happens with minimal rewriting of transformation logic.

One important takeaway: You're not locked into either approach. Both deliver the same performance benefits and use similar SQL patterns. You can start where your team is most comfortable and evolve your approach as requirements change.

From implementation to advanced patterns

Whether you choose pipelines or standalone streaming tables and materialized views, the experience stays the same: It's all SQL, written in the same editor you already use. No new orchestration layer to learn, no pipeline framework to manage — just define the logic and let the system handle scheduling, incremental processing and optimization. These capabilities slot naturally into existing workflows, whether that's direct development in the Databricks Platform, SQL notebooks, or tools like dbt.

But what happens when you need to handle more complex data scenarios? Tracking changes over time, managing late-arriving updates and maintaining complete historical records for compliance — these change data capture patterns have traditionally required specialized engineering expertise.

The next chapter shows how declarative SQL transforms them into accessible tools for any SQL practitioner.

Simplifying Change Data Capture with Declarative SQL

You've chosen your path — either Lakeflow Declarative Pipelines for governed ETL or standalone streaming tables and materialized views for direct SQL work. Now you need to handle one of the most challenging aspects of data engineering: tracking changes over time.

For SQL practitioners, change data capture (CDC) has traditionally meant using a third-party tool or depending on engineering teams to build custom solutions. Most teams that attempt CDC find themselves writing hundreds of lines of complex code just to handle basic insert, update and delete operations. The complexity only compounds when dealing with out-of-order events, partial updates and schema changes.

Automatic CDC with Lakeflow Declarative Pipelines

Lakeflow Declarative Pipelines eliminates this complexity entirely. A typical Structured Streaming CDC implementation that once required 300+ lines of Scala code now becomes a single declarative SQL statement. Teams no longer need extensive engineering support to build working CDC solutions.

Declarative Pipelines provides built-in CDC functionality through familiar SQL syntax:

- **Declarative API for streaming CDC:** Apply changes to target Delta tables by specifying composite primary keys and deletion conditions using familiar SQL syntax
- **Automatic ordering:** Events arriving out of order are automatically reconciled based on user-specified sequencing columns, ensuring consistent final results
- **Partial update support:** Handle change records containing only modified fields, with NULL values representing unchanged columns. The system automatically coalesces partial updates into complete, current rows
- **Schema evolution:** Columns are automatically added to target tables, including complex nested schemas, without manual DDL management

Here's how this works in practice:

SQL

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

-- Apply CDC changes into the target table
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;
```

This declarative approach handles the entire CDC workflow — from ingesting change streams to maintaining current state — without procedural complexity.

Note: The APPLY CHANGES syntax shown here is being replaced by the new AUTO CDC syntax in Databricks. While APPLY CHANGES will continue to be supported for now, we recommend using AUTO CDC for all new workloads.

Slowly changing dimensions type 2

Slowly changing dimensions type 2 (SCD2) patterns maintain both current and historical record versions, preserving point-in-time analysis capabilities.

Traditional SCD2 implementations require complex logic for timestamp management, late-arriving data and referential integrity across historical snapshots.

Declarative Pipelines includes native SCD2 support in both streaming and batch CDC APIs:

SQL

```sql
-- Create streaming table
CREATE STREAMING TABLE cities

-- Apply CDC changes into the cities table
APPLY CHANGES INTO live.cities
FROM STREAM city_updates
KEYS (id)
SEQUENCE BY ts
COLUMNS * EXCEPT (operation)
STORED AS SCD TYPE 2;
```

city_updates

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

cities

id	city	__starts_at	__ends_at
1	Berkeley, 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`).

The system automatically manages historical versioning, timestamps and record lifecycle — turning a complex procedural process into a single declarative statement.

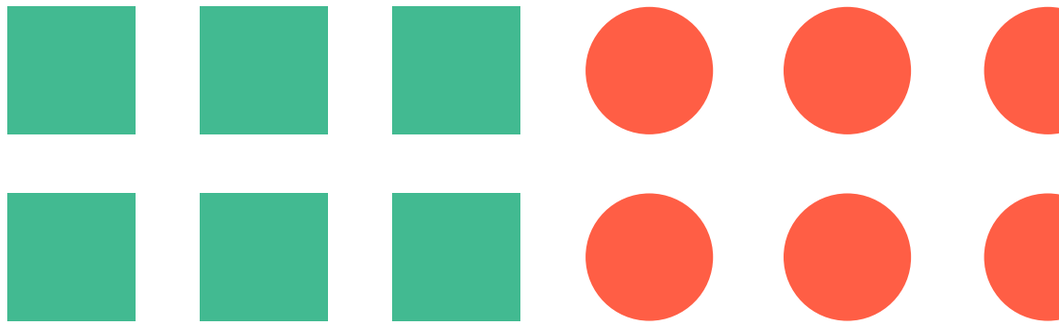
Whether you need real-time streaming CDC, snapshot-based processing or historical tracking through SCD2, Declarative Pipelines provides the same declarative simplicity. These patterns form the foundation for building robust data pipelines that can handle the full spectrum of change tracking requirements your organization faces.

Note: The APPLY CHANGES syntax shown here is being replaced by the new AUTO CDC syntax in Databricks. While APPLY CHANGES will continue to be supported for now, we recommend using AUTO CDC for all new workloads.

Taking SQL-first ETL forward

The production gap that separates SQL practitioners from modern data engineering is closing. The same SQL skills you use for analysis can now drive production pipelines at scale. This shift transforms how teams work together — no more waiting for engineering handoffs, no more translating logic between languages. You can build, maintain and evolve pipelines using tools you already know.

You now have everything you need to start building. Pick your approach: Declarative Pipelines for end-to-end workflows, standalone streaming tables and materialized views for targeted improvements. Both approaches support CDC patterns that turn complex change tracking into straightforward SQL operations. Start small with one pipeline, one model or one transformation. Measure development time, debugging effort and pipeline reliability. Let the data guide where to apply SQL-first approaches next.



Appendix

This appendix provides concrete examples and reference materials to help you implement declarative SQL ETL in your organization. You'll find code patterns for common scenarios and guidance on choosing the right approach for different use cases.

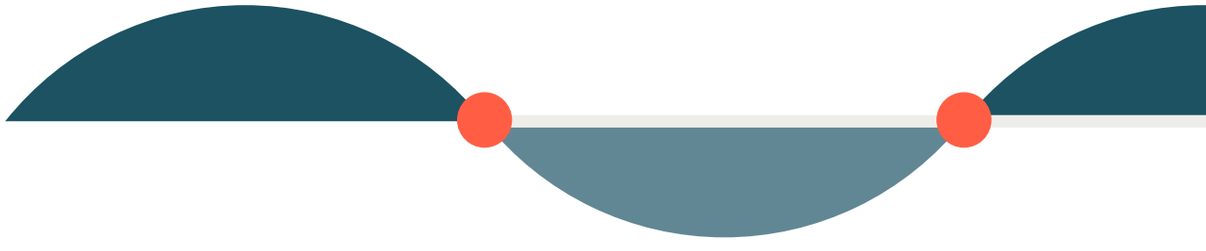
Bringing declarative SQL into existing dbt workflows

If your team already uses dbt, you can leverage streaming tables and materialized views immediately. They integrate directly into existing dbt projects, simplifying transformation logic while delivering better performance without workflow disruption.

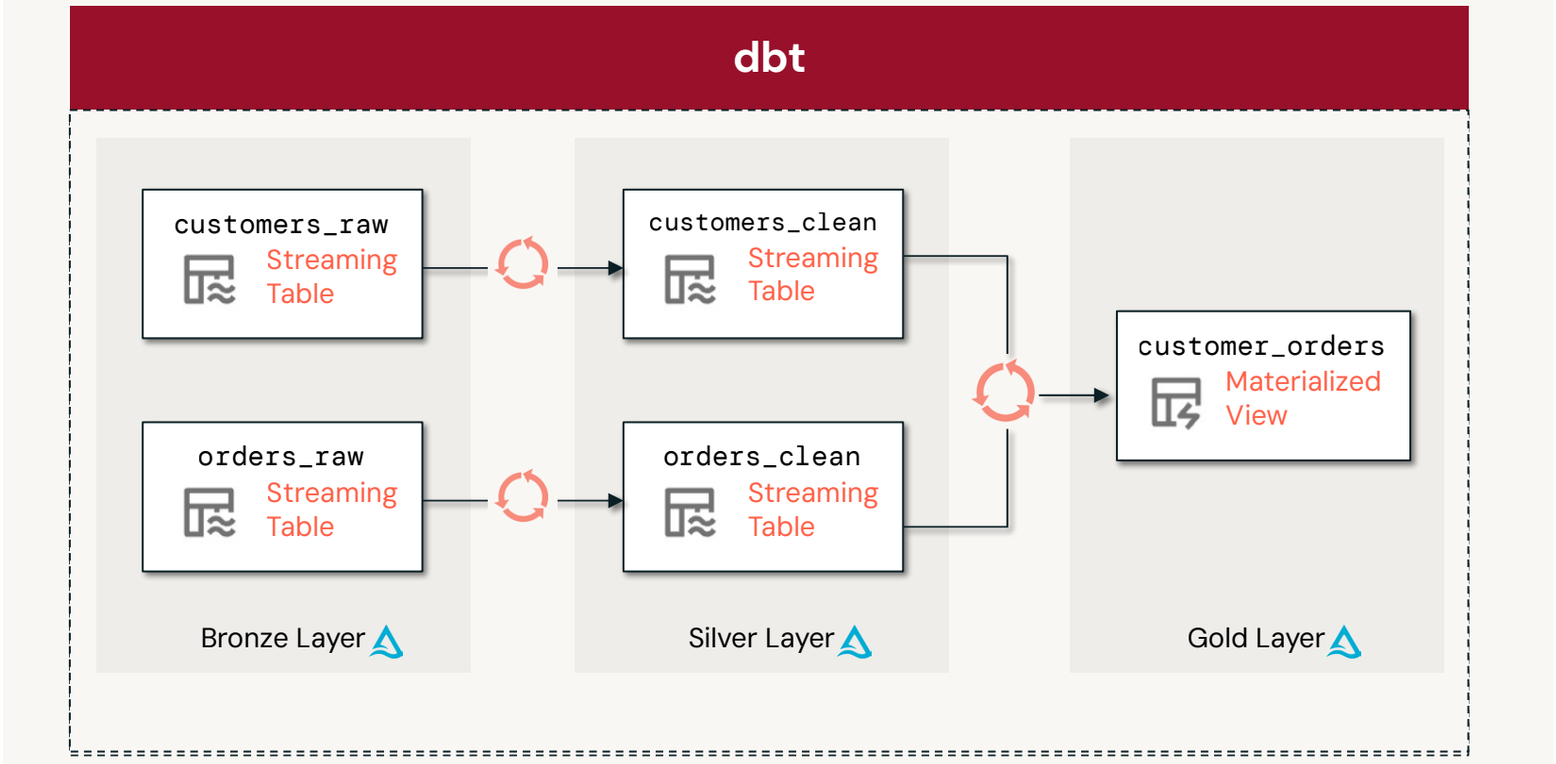
Both tools share the same philosophy: build production pipelines using SQL without learning new frameworks. This shared philosophy creates seamless compatibility — and makes them better as a pair.

NATURAL INTEGRATION: DBT USING STREAMING TABLES AND MATERIALIZED VIEWS

This alignment preserves your existing workflow. Your team keeps the same development patterns, testing processes and deployment commands. What changes is the performance and intelligence happening behind the scenes.



Consider a typical implementation using dbt. You define a pipeline with native streaming tables and materialized views — all in SQL, just like any other dbt model:



- **Bronze layer:** Ingest raw data as streaming tables
- **Silver layer:** Clean and enrich that data with additional streaming tables
- **Gold layer:** Aggregate and join in materialized views for consumption

All of this is orchestrated with dbt. The same `dbt run` commands. The same dependency management. The same testing and documentation capabilities you already rely on.

CORE BENEFITS FOR DBT USERS

This integration delivers three key advantages that address common pain points in dbt development.

- **Simpler code** — Means you write SQL while the platform handles state, partitions and refresh logic. Those intricate `is_incremental()` blocks and custom merge patterns disappear.
- **Native streaming capabilities** — Process data in real time by design. You can start with batch processing and scale to streaming without rewriting transformation logic.
- **Automatic incremental refresh** — Lets materialized views choose the optimal strategy based on your query and source data. The system handles cost optimization and scheduling automatically.

This approach enhances existing dbt pipelines with native streaming and incremental processing — without disrupting the workflow your team relies on.

Two practical examples demonstrate how these benefits translate into real dbt improvements.

MODERNIZING COPY INTO PATTERNS WITH STREAMING TABLES

`COPY INTO` is a common ingestion pattern in dbt projects, but legacy implementations create complexity that slows development. These implementations typically require Jinja macros, custom file format handling and manual schema merge options. Teams run them using `dbt run-operation` commands with multiple arguments passed in:

Before	After
Query definition	

`{% macro copy_into_target(table_name, source_path, file_format='csv') %}`
`  COPY INTO {{ table_name }}`
`  FROM {{ source_path }}`
`  FILEFORMAT = {{ file_format }}`
`  FORMAT_OPTIONS (`
`  'mergeSchema' = 'true',`
`  'header' = 'true'`
`  )`
`{% endmacro %}`
	`dbt model "airline_trips_bronze.sql"`
	`{{`
	`  config(`
	`  materialized='streaming_table'`
	`  )`
	`}}`
	
	`SELECT *`
	`FROM stream(`
	`  read_files(`
	`  's3://my-bucket/my-folder/',`
	`  format => 'json'`
	`  )`
	`)`

Run update	

`dbt run-operation copy_into_target \`
`--args '{`
`  "table_name": "raw.my_table",`
`  "source_path": [`
`  "s3://my-bucket/my-folder/"`
`  ],`
`  "file_format": "CSV"`
`}'`
	`dbt run --select airline_trips_bronze`

Streaming tables eliminate this complexity. You replace the macro with a standard dbt model, configure it as `materialized='streaming_table'` and use `read_files()` inside a SELECT statement. No macros, no operational commands.

This transformation delivers four key improvements to the pipeline:

- **Simpler operations** — Eliminate boilerplate, macros and operational commands. What remains is clean SQL that any team member can understand and maintain.
- **Faster processing** — Happens automatically as streaming tables ingest new files when they arrive, rather than reprocessing entire datasets on scheduled batch runs
- **Greater scalability** — Comes from serverless compute that scales based on data volume without capacity planning or resource management
- **Enhanced power** — Lets you filter and transform directly in the SELECT statement, reducing pipeline complexity by handling simple transformations during ingestion rather than in separate downstream models

You keep your dbt workflow while gaining real-time, declarative ingestion with minimal code changes. Materialized views offer similar simplification for transformation logic.

UPGRADING DBT INCREMENTAL MODELS WITH MATERIALIZED VIEWS

Traditional dbt incremental models create maintenance overhead. They require `materialized='incremental'` configuration, `unique_key` definitions and complex Jinja conditional logic with `is_incremental()` blocks. This approach works, but it's fragile, verbose and error-prone.

Before

Query definition

```
{{
    config(
        materialized='incremental',
        unique_key='order_id'
     )
}}

SELECT
    o.order_id,
    o.order_date,
    o.customer_id,
    c.customer_name,
    c.customer_email
FROM {{ ref('orders') }} o
JOIN {{ ref('customers') }} c
    ON o.customer_id = c.customer_id


{% if is_incremental() %}
   WHERE o.updated_at > (
       SELECT max(updated_at)
       FROM {{ this }}
    )
{% endif %}
```

After

```
{{
    config(
        materialized='incremental',
    )
}}

SELECT
     o.order_id,
     o.order_date,
     o.customer_id,
     c.customer_name,
     c.customer_email
FROM {{ ref('orders') }} o
JOIN {{ ref('customers') }} c
    ON o.customer_id = c.customer_id
```

Materialized views eliminate this complexity with a single configuration change: `materialized='materialized_view'`. No more `is_incremental()` blocks. No more manual `WHERE` clause construction. No risk of inconsistent logic across models.

This approach delivers three fundamental improvements:

- **Eliminates hand-coded logic:** Materialized views choose the optimal incremental strategy — merge, partition recompute or full refresh — based on cost and correctness analysis. Teams no longer need to write and maintain conditional refresh logic for each model.
- **Guarantees correctness:** Results remain consistent regardless of query complexity. The system analyzes transformation logic and source data changes to determine the optimal refresh approach, eliminating logic errors in manual incremental implementations.
- **Enhances scalability:** Materialized views run on serverless infrastructure that scales automatically. No capacity planning required. Costs optimize based on actual usage patterns.

You get faster refresh, simpler code and higher confidence with a single configuration change.

BUILDING COMPLETE PIPELINES

These individual improvements become transformative when streaming tables and materialized views work together in complete dbt projects. You can build end-to-end pipelines that handle ingestion, transformation and serving — all using familiar dbt patterns enhanced with declarative SQL capabilities.

Your dbt workflows remain unchanged while the system handles scheduling, optimization and dependency management automatically. Development stays familiar. Testing works the same way. Documentation generates normally.

The difference is performance and operational simplicity — benefits that compound as your dbt projects grow in complexity and scale.

Source evolution with FLOWS

Real-world data sources evolve over time. New data arrives from different systems, historical data needs backfilling and existing sources require corrections or schema changes.

Managing evolving data requirements — backfills, new sources and corrections — typically disrupts existing pipelines and requires extensive rework.

The FLOWS API solves this by enabling writes to existing streaming tables:

- **One-time data loads:** Backfill historical data into existing streaming tables without disrupting ongoing ingestion
- **Multiple streaming sources:** Add new data sources to existing tables as business requirements evolve
- **Corrections and adjustments:** Handle data quality issues by writing corrected records to production tables

Here's a practical example: A streaming table `raw_data` is created to store incoming data. Two different flows (`kafka_us_east` and `kafka_us_west`) are defined that insert data from two different Kafka topics or clusters into the same `raw_data` streaming table:

```sql
CREATE OR REPLACE STREAMING TABLE raw_data;

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

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

FLOWS enable streaming tables to evolve with your data landscape without requiring pipeline rewrites or migration downtime.

Technical resources and reference materials

SQL LANGUAGE REFERENCE FOR LAKEFLOW DECLARATIVE PIPELINES

The Lakeflow Declarative Pipelines SQL language reference provides detailed documentation for all operators available in Declarative Pipelines.

Key sections include:

- **CREATE STREAMING TABLE** — Syntax and options for ingesting data from various sources
- **CREATE MATERIALIZED VIEW** — Patterns for transformation and aggregation
- **APPLY CHANGES** — Operations for CDC and SCD implementations
- **FLOWS** — Syntax for source evolution and backfill scenarios

Each operator includes detailed examples and use cases specific to ETL and pipeline development scenarios.

DATABRICKS DATA ENGINEERING TUTORIALS

The Databricks Demo Center provides hands-on, interactive tutorials covering comprehensive data engineering scenarios on Databricks. You can install dbdemos directly from your Databricks Notebooks to explore sample implementations for various data engineering patterns.

Key data engineering tutorials include:

- **Declarative Pipelines implementations** with real-world datasets and use cases
- **CDC pipeline development** with Delta Lake integration
- **Streaming ETL patterns** for real-time data processing

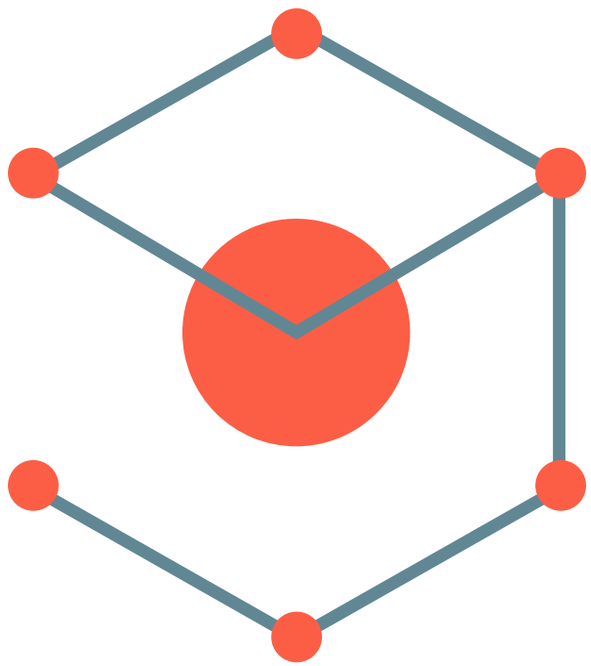
LAKEFLOW DECLARATIVE PIPELINES ONLINE COURSE

Build Data Pipelines with Lakeflow Declarative Pipelines is a free online course that teaches data engineers how to create robust data pipelines using Declarative Pipelines in Databricks. No prior Databricks experience required.

The course covers everything from basic concepts to production-ready implementations. You'll master the core components that power modern data architecture and learn when to use each one.

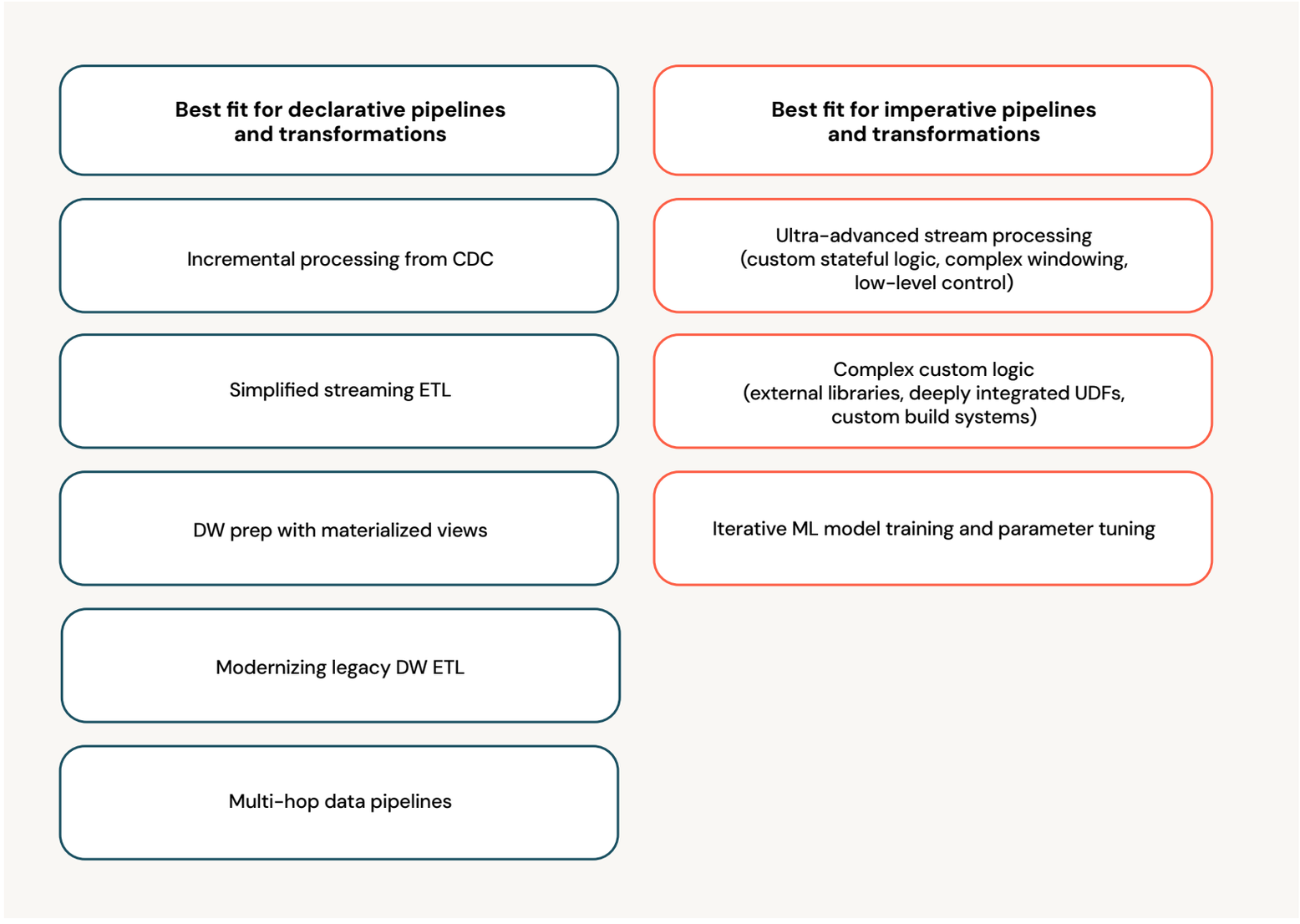
Key learning areas include:

- **Core Declarative Pipelines components** such as streaming tables, materialized views and temporary views, plus when and how to use each one effectively
- **Incremental data processing** and best practices for both batch and streaming data scenarios
- **Pipeline architecture design** and how to combine multiple streaming tables and materialized views for optimal performance
- **Component selection guidance**, including decision frameworks for choosing the right tools for each data challenge



When to use declarative vs. imperative approaches in ETL pipelines

While declarative approaches work for most ETL scenarios, some situations call for imperative capabilities. Use this guide to choose the right approach:



Declarative approaches are a great default for ETL in Databricks Lakeflow, with imperative approaches reserved for procedural logic or external libraries. This approach keeps pipelines accessible to declarative practitioners while enabling complex scenarios.

Modern ETL for SQL Users

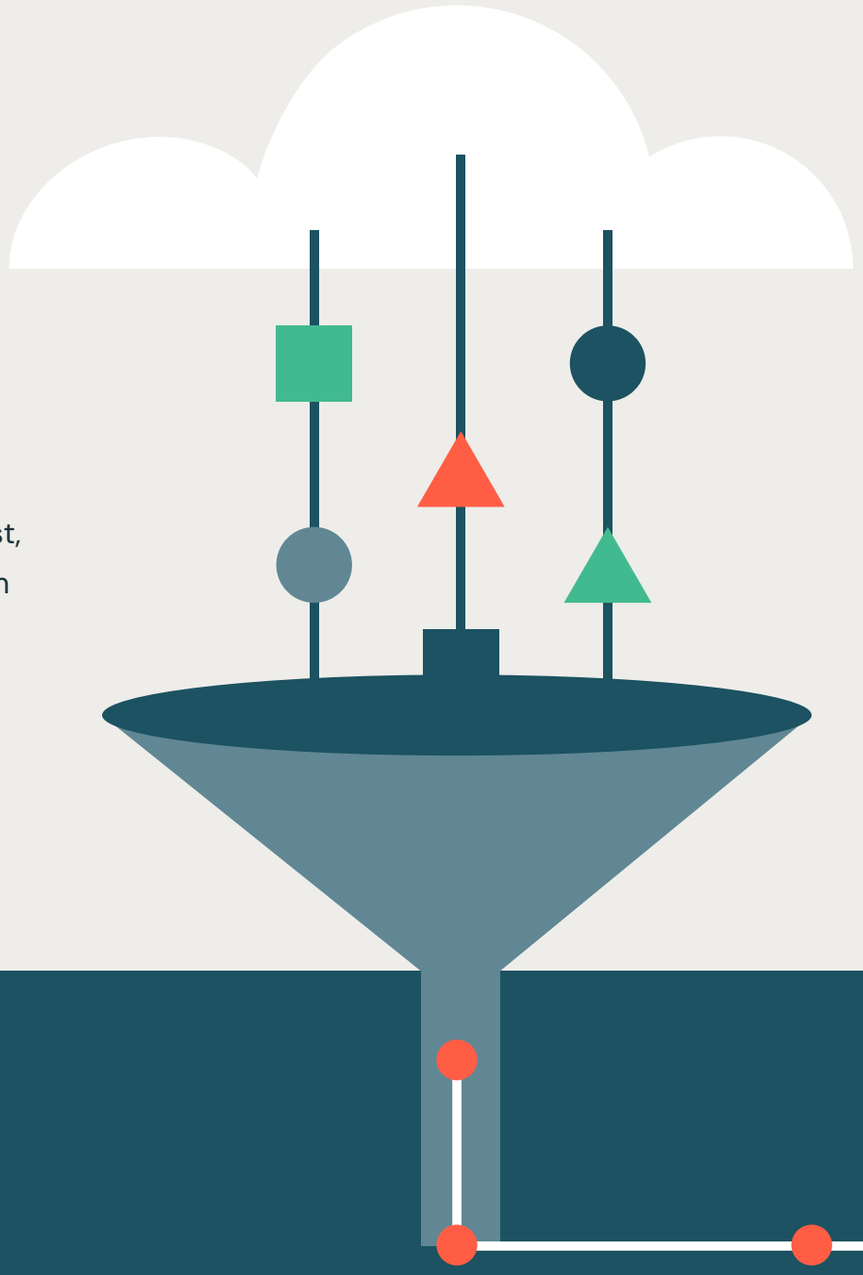
Declarative ETL makes it easier than ever to build fast, reliable data workflows—directly in SQL.

With Lakeflow Declarative Pipelines, you can ingest, transform, and serve data using the SQL you already know. Whether you're building dashboards, supporting real-time analytics or modernizing legacy models, Lakeflow brings powerful automation and intelligent optimization to every stage of your pipeline.

No orchestration frameworks. No complex refresh logic. Just declarative SQL that works.

LEARN MORE

TAKE THE PRODUCT TOUR



About Databricks

Databricks is the data and AI company. More than 15,000 organizations worldwide — including Block, Comcast, Condé Nast, Rivian, Shell and over 60% of the Fortune 500 — rely on the Databricks Data Intelligence Platform to take control of their data and put it to work with AI. Databricks is headquartered in San Francisco, with offices around the globe, and was founded by the original creators of Lakehouse, Apache Spark™, Delta Lake, MLflow, and Unity Catalog. To learn more, follow Databricks on LinkedIn, X and Facebook.

databricks