databricks

Big Book of Data Engineering

Contents

01

Introduction to Data Engineering on Databricks

Introduction to Data Engineering on Databricks

As data volume continues to expand and the complexity of data sources and distribution intensifies, organizations struggle to maintain control of their data and effectively benefit from it. According to a recent IDC Data Valuation Survey, 41% of organizations responded that data is changing faster than they can keep up with. Combined with an increased pressure to capitalize on AI, organizations across industries are facing new data-driven challenges they have to overcome to remain competitive in their field. Many of them running the data race encounter the same hard truth:

Everything, from AI to analytics to BI to applications, starts with good data.

This reality emphasizes the importance of building reliable data pipelines that can ingest or stream vast amounts of data efficiently and ensure high data quality reliably and at scale. A unified platform and good data engineering are essential components of success in every analytics, business intelligence (BI) and AI initiative. With the right data engineering solution, organizations can turn their proprietary data into real business assets and, in reverse, leverage those business insights to build their applications as well as their agentic workflows.

This book uses practical guidance, useful patterns, best practices and real-world examples to help you understand how data engineers can meet the challenges of this new era and how the Databricks Data Intelligence Platform can support you on your journey.

What is data engineering?

Data engineering is the practice of taking raw data from a data source and processing it so it's stored and organized for a downstream use case, such as data analytics, BI or machine learning (ML) model training. In other words, it's the process of preparing data so that value can be extracted from it.

A useful way of thinking about data engineering is by using the following framework, which includes three main parts:

1. **Ingest**
   Data ingestion is the process of bringing data from one or more data sources into a data platform. These data sources can be files stored on-premises or in cloud storage services, databases, applications and, increasingly, data streams that produce real-time events.
2. **Transform**
   Data transformation takes raw ingested data and uses a series of steps (referred to as "transformations") to filter, standardize, clean and finally aggregate it so it's stored in a usable way. A popular pattern is the medallion architecture, which defines three stages in the process — Bronze, Silver and Gold.
3. **Orchestrate**
   Data orchestration refers to the way a data pipeline that performs ingestion and transformation is scheduled and monitored, as well as the control of the various pipeline steps and how it handles failures (e.g., by executing a retry run).

Challenges of data engineering in the AI era

As previously mentioned, data engineering is key to ensuring reliable data for AI, analytics and BI initiatives. Data engineers who build and maintain ETL pipelines and the data infrastructure that underpins analytics and AI workloads face specific challenges in this fast-moving landscape.

- **Disparate data sources challenge most organizations:** ISG predicts that by 2026, 8 in 10 enterprises will have their data spread across multiple cloud providers and on-premises data centers that span multiple locations. This decentralization creates a dependency on specialized, siloed teams, inefficient pipelines, development with high costs and slow time to value. As a result, data usage is limited and innovation is blocked.
- **Fragmented tooling:** Most companies rely on multiple external tools and services or custom in-house solutions to manage their ETL processes. These tools are often disjointed with different architectures, interfaces and outputs, which adds another layer of complexity to the data engineering experience and leads to bottlenecked teams.
- **Handling real-time data:** From mobile applications to sensor data on factory floors, more and more data is created and streamed in real time and requires low-latency processing so it can be used in real-time decision-making.
- **Scaling data pipelines reliably:** With data coming in large quantities and often in real time, scaling the compute infrastructure that runs data pipelines is challenging, especially when trying to keep costs low and performance high. Running data pipelines reliably, monitoring them and troubleshooting when failures occur are some of the most important responsibilities of data engineers.
- **Data quality:** "Garbage in, garbage out." High data quality is essential to training high-quality models and gaining actionable insights from data. Ensuring data quality is a key challenge for data engineers.
- **Governance and security:** Data governance is becoming a key challenge for organizations that find their data spread across multiple systems, with increasingly larger numbers of internal teams looking to access and utilize it for different purposes. Securing and governing data is also an important regulatory concern that many organizations face, especially in highly regulated industries.

These challenges stress the importance of choosing the right data platform for navigating constantly evolving data needs, particularly in the age of AI. But a data platform in this age can also go beyond addressing just the challenges of building AI solutions. The right platform can improve the experience and productivity of data practitioners, including data engineers, by infusing intelligence and using AI to assist with daily engineering tasks.

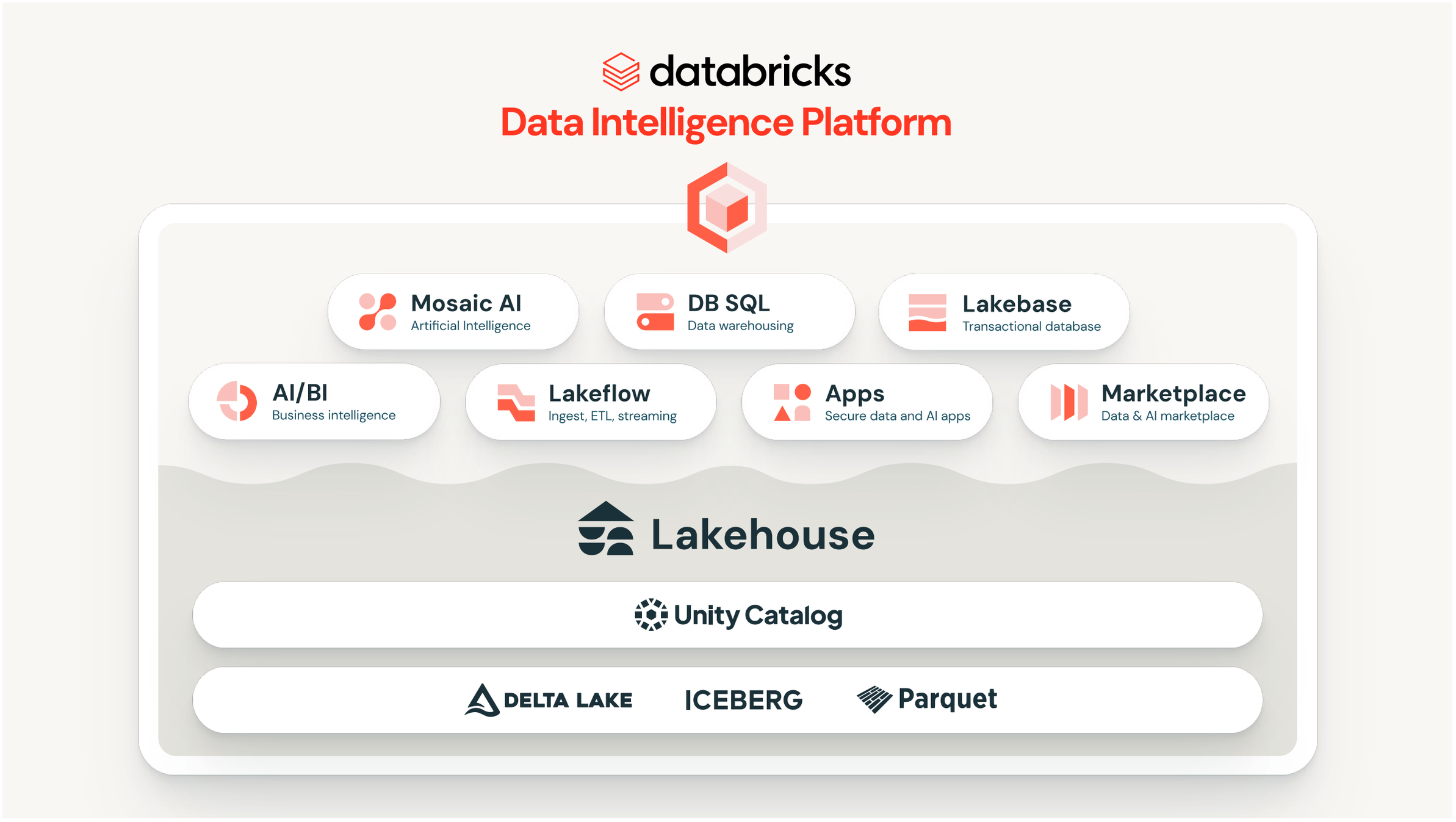
In other words, the new data platform is a data *intelligence* platform.

The Databricks Data Intelligence Platform

The Databricks mission is to democratize data and AI, allowing organizations to use their unique data to build or fine-tune their own machine learning and generative AI models to produce new insights that lead to business innovation.

The Databricks Data Intelligence Platform is built on lakehouse architecture to provide an open, unified foundation for all data and governance, and is powered by a Data Intelligence Engine that understands the uniqueness of your data. With these capabilities at its foundation, the Data Intelligence Platform lets Databricks customers run a variety of workloads, from business intelligence and data warehousing to AI and data science.

To get a better understanding of the Databricks Platform, the next page shows an overview of the different parts of the architecture as it relates to data engineering. The Databricks Data Intelligence Platform enables you to execute all your data, analytics, BI and AI initiatives. As a 100% serverless platform, it provides you with built-in features such as disaster recovery, cost controls and enterprise security. Key components feature Mosaic AI with end-to-end AI for both generative and classical AI; Databricks SQL, a serverless intelligent data warehouse; unified data engineering with Lakeflow, a built-in database with Lakebase; and AI/BI that integrates deeply with Databricks SQL to easily extend business intelligence across your business.


databricks
Data Intelligence Platform
Mosaic AI
Artificial Intelligence
DB SQL
Data warehousing
Lakebase
Transactional database
AI/BI
Business intelligence
Lakeflow
Ingest, ETL, streaming
Apps
Secure data and AI apps
Marketplace
Data & AI marketplace
Lakehouse
Unity Catalog
DELTA LAKE
ICEBERG
Parquet

Several elements of the Data Intelligence Platform are particularly important for data engineers and are worth a closer look. The next section will dive into Lakeflow — the unified data engineering solution for ingestion, transformation and orchestration. But first, it is worth understanding the foundational pieces of the platform that provide important value to data engineers:

UNIFIED AND OPEN GOVERNANCE WITH DATABRICKS UNITY CATALOG

Unity Catalog (UC) provides a single, centralized data catalog for managing permissions, auditing and lineage across all data and AI assets in the lakehouse. It supports all major open formats, including Delta Lake, Apache Iceberg™, Hudi and Parquet, so data engineers can choose the right format for their workloads without lock-in. With built-in access controls, data quality monitoring and column-level lineage, Unity Catalog makes it easier to enforce compliance and trust while accelerating development. Interoperability across formats and secure data sharing through Delta Sharing ensure engineers can collaborate across teams, platforms and clouds while maintaining consistent governance. Unity Catalog reduces complexity, scales with the needs of modern data platforms and keeps openness and reliability at the foundation of every data engineering workflow.

ACCELERATING PRODUCTIVITY WITH THE DATABRICKS ASSISTANT

Databricks combines generative AI with the unification benefits of a lakehouse to power a Data Intelligence Engine that understands your data's unique semantics. This allows the Databricks Platform to automatically optimize performance and manage infrastructure in ways unique to your business. Built on this engine is the Databricks Assistant — a context-aware AI assistant that offers a conversational API to query data, generate code, explain code queries and even fix issues. The Databricks Assistant helps practitioners, including data engineers, become more productive by cutting down the time it takes to build new data pipelines and to troubleshoot pipeline issues.

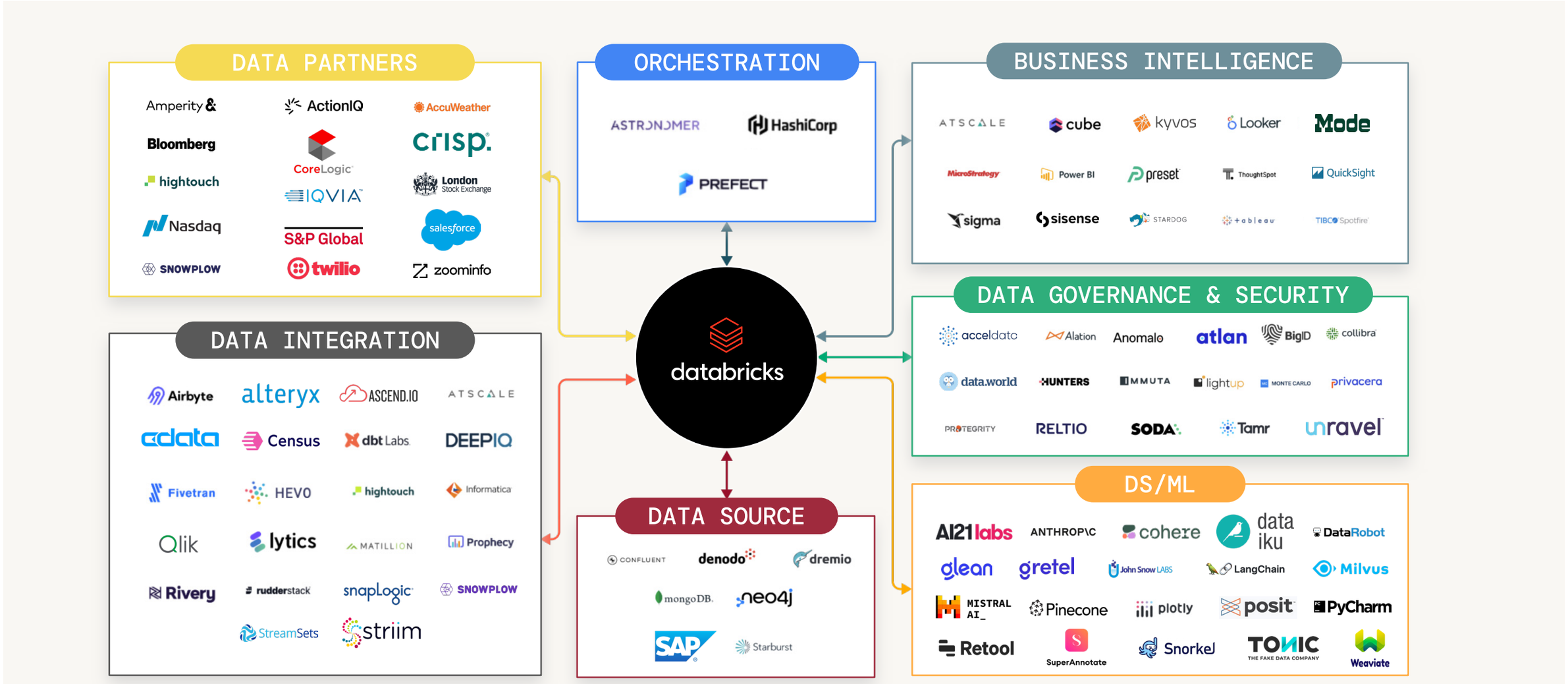
BUILT-IN AND AI-READY DATABASE WITH LAKEBASE

Every application runs on a database and Postgres is the database of choice for most application developers. Databricks Lakebase is a fully managed Postgres database deeply integrated in the lakehouse to make it easy to build data and AI applications. Lakebase automatically syncs lakehouse data and features/models into an operational database for low-latency applications, dashboards and CRM systems. With Lakebase, engineering teams can focus on delivering insights to applications instead of building custom pipelines, managing infrastructure or administering databases.

A RICH ECOSYSTEM OF DATA SOLUTIONS

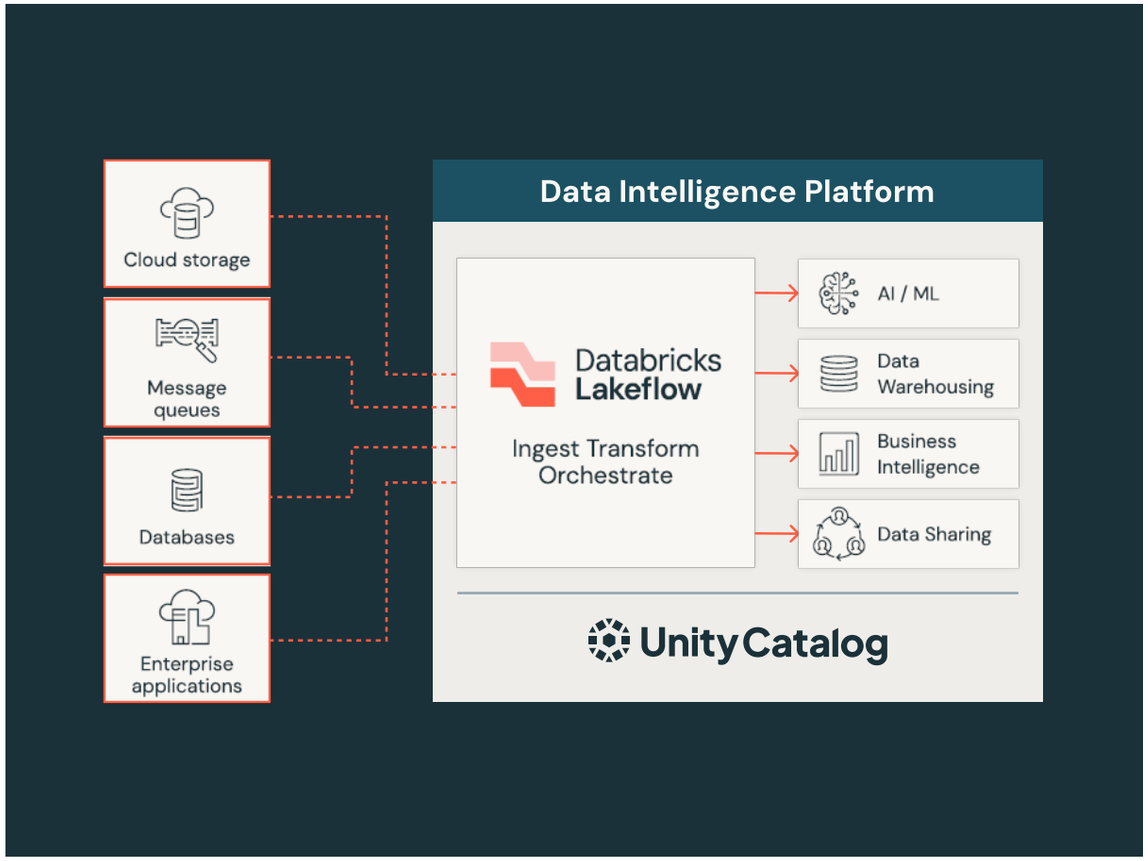
The Data Intelligence Platform is built on open source technologies and uses open standards, so leading data solutions can be leveraged with anything you build on the lakehouse.

A large collection of technology partners makes it easy and simple to integrate the technologies you rely on when migrating to Databricks — and you are not locked into a closed data technology stack.



Unified data engineering with Databricks Lakeflow

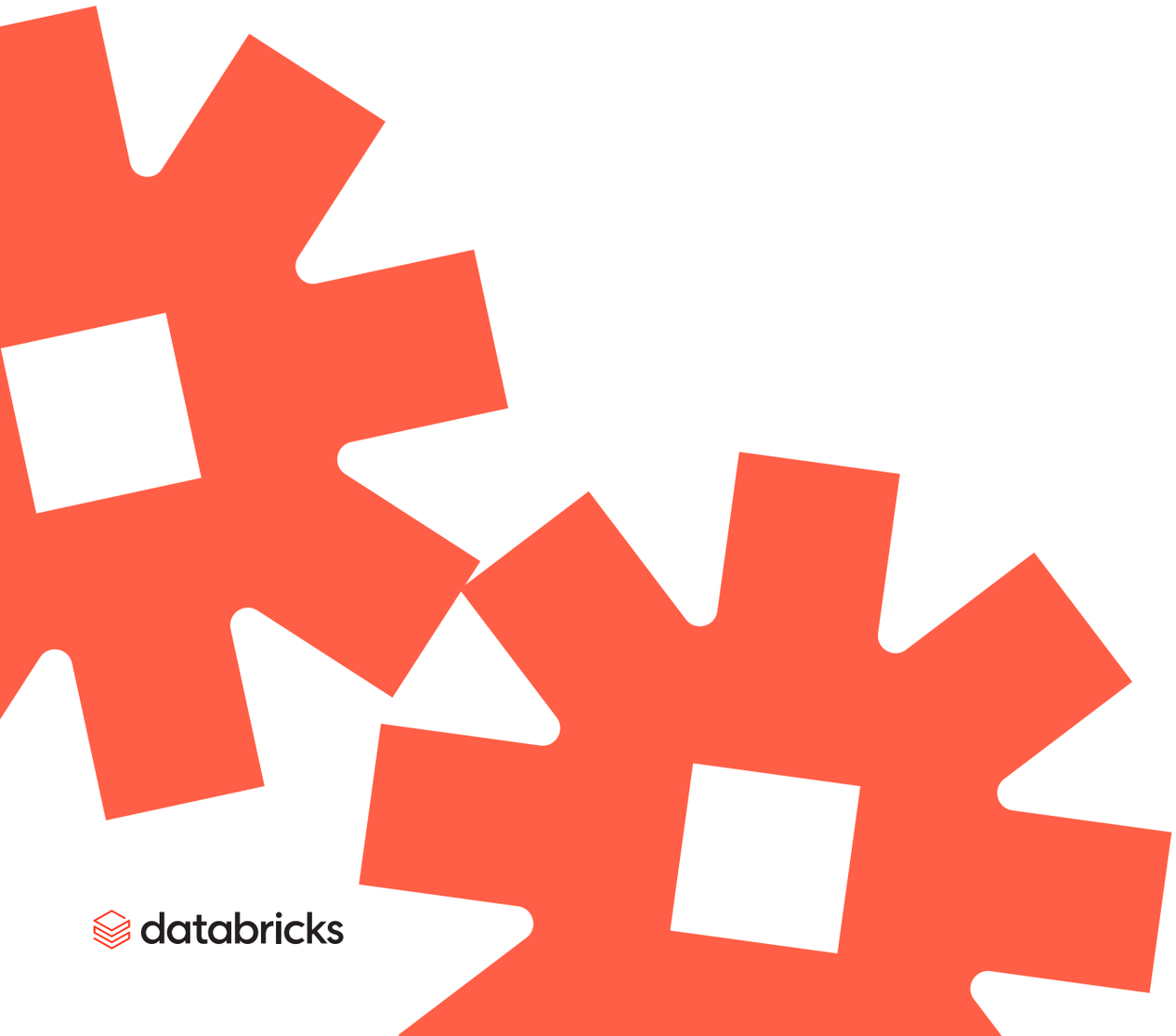
Lakeflow simplifies and unifies the data engineering experience with an integrated and modern ETL platform built on top of the Data Intelligence Platform. Its intuitive and centralized interface for data ingestion, transformation and orchestration reduces tool sprawl, accelerates time to production and value, and unlocks data engineering for more users across the organization. Lakeflow enables data engineering teams to move faster and more reliably, without compromising governance, scalability or performance.



Lakeflow is made of three deeply integrated components:

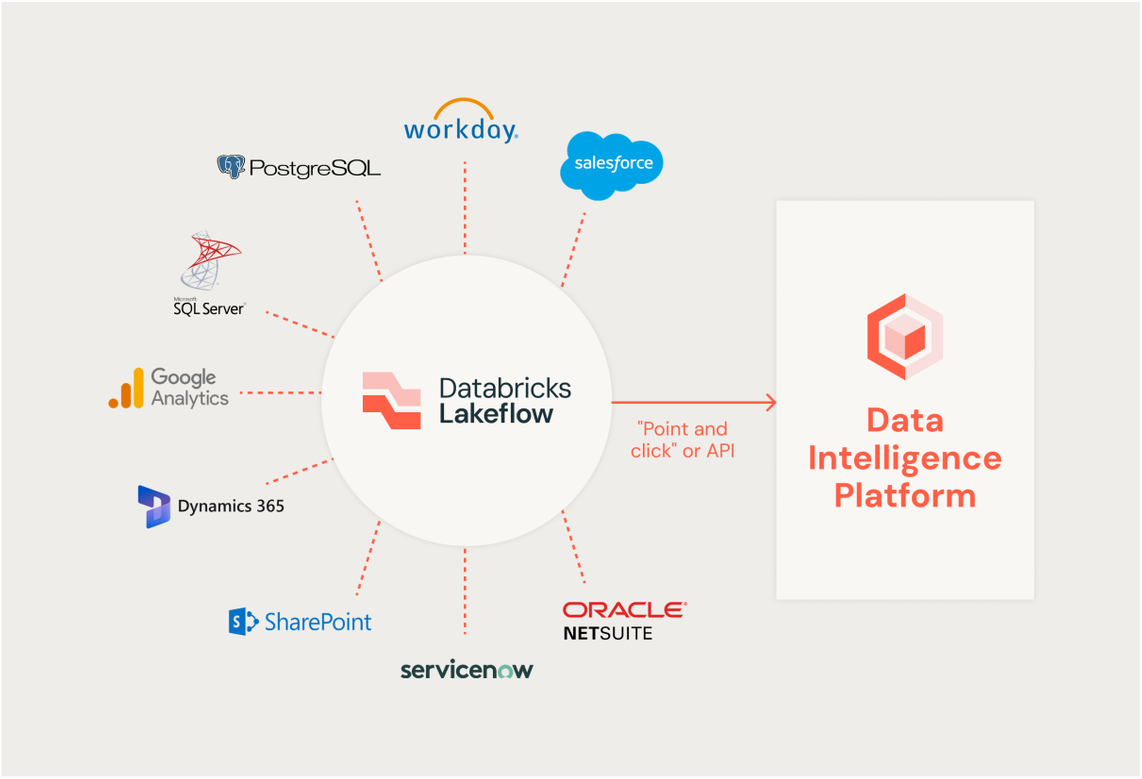
- **Lakeflow Connect** for ingestion
- **Lakeflow Spark Declarative Pipelines** for transformation
- **Lakeflow Jobs** for orchestration

In addition, the **Lakeflow Designer** is a no-code experience meant to help any practitioner build data pipelines on top of Lakeflow.



DATA INGESTION WITH LAKEFLOW CONNECT

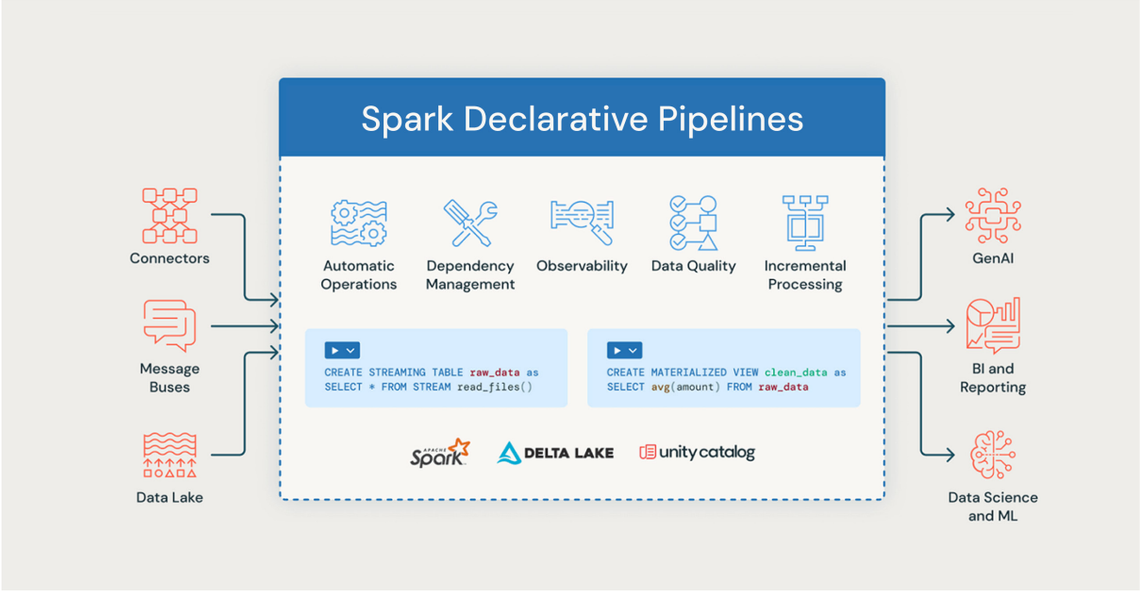
Databricks enables organizations to efficiently ingest data from various systems into a single, open and unified lakehouse architecture. Lakeflow Connect offers simple data ingestion connectors for applications, databases, cloud storage, message buses and more, into your lakehouse. These built-in connectors provide efficient end-to-end incremental ingestion at scale, easy setup with a point-and-click UI or API, and unified governance via Unity Catalog. With Lakeflow Connect, Databricks has continued to innovate, expanding the breadth of supported data sources with built-in, managed connectors, as well as connectors with more customization options. Zerobus Ingest enables you to push event data directly to your lakehouse without requiring a message bus, thereby simplifying ingestion for IoT, clickstream, telemetry and other similar use cases. Auto Loader is a fully customizable connector that provides a Structured Streaming source for incrementally and efficiently processing data as it arrives in cloud object storage, recommended for usage with Declarative Pipelines.



Built-in data connectors for enterprise applications, file sources and databases

DATA TRANSFORMATION FOR SPARK DECLARATIVE PIPELINES

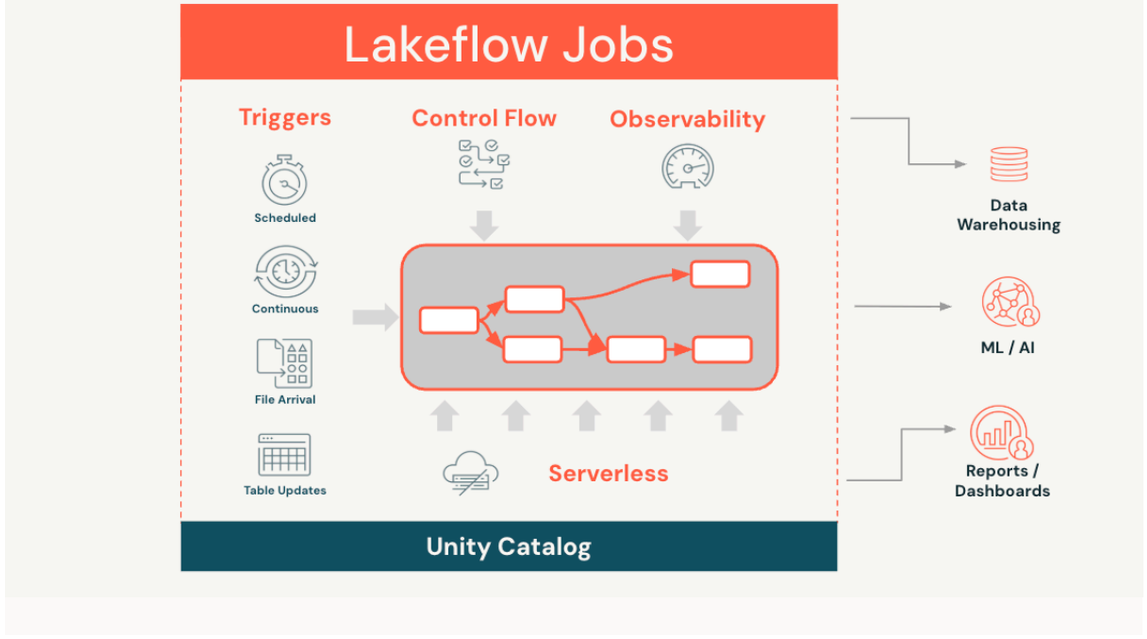
Spark Declarative Pipelines is a declarative ETL framework (built on the open source Apache Spark™ Declarative Pipelines) that helps data teams simplify and make ETL cost-effective in streaming and batch. Simply define the transformations you want to perform on your data and let Spark Declarative Pipelines automatically handle task orchestration, compute management, monitoring, data quality and error management. Engineers can treat their data as code and apply modern software engineering best practices like testing, error handling, monitoring and documentation to deploy reliable pipelines at scale. Declarative Pipelines fully supports both Python and SQL and is tailored to work with both streaming and batch workloads.



Reliable data pipelines made easy with Spark Declarative Pipelines

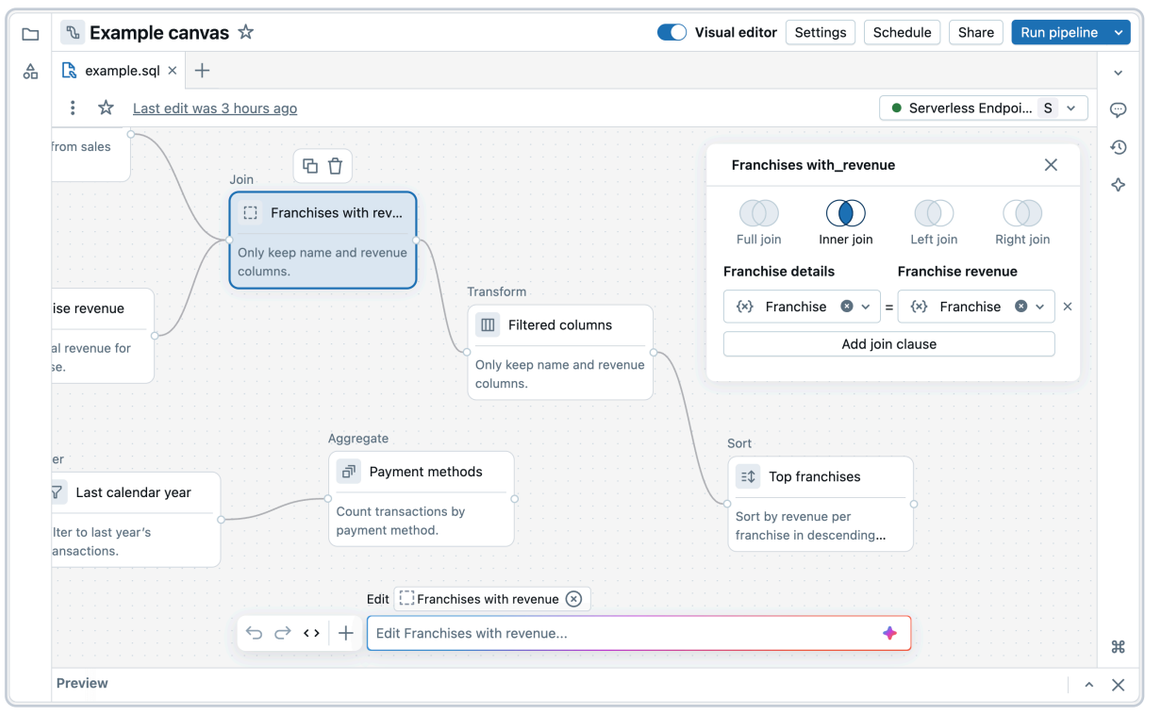
UNIFIED DATA ORCHESTRATION WITH LAKEFLOW JOBS

Lakeflow Jobs offers a simple, reliable orchestration solution for data, analytics and AI on the Data Intelligence Platform. Lakeflow Jobs lets you define multistep workflows to implement ETL pipelines, ML training workflows and more. It offers enhanced control flow capabilities and supports different task types and workflow triggering options. As the platform's native orchestrator, Lakeflow Jobs also provides advanced observability to monitor and visualize workflow execution, along with alerting and troubleshooting capabilities for when issues arise. Lakeflow Jobs comes with serverless compute options, so you can leverage smart scaling and efficient task execution.



PRODUCTION-GRADE DATA PIPELINES, NO-CODE REQUIRED WITH LAKEFLOW DESIGNER

Analytics projects require collaboration between data engineers and business analysts, who often work on separate platforms. While no-code tools can help bridge this gap, challenges remain with siloed workflows, production headaches and limited AI productivity. Lakeflow Designer solves this with a no-code, AI-assisted tool — native to the Databricks Data Intelligence Platform — that empowers business analysts and others to build production-ready data pipelines with natural language. Powered by data intelligence, Lakeflow Designer empowers shared, collaborative workflows with a built-in path to production, built on AI that understands your business.



Why data engineers choose Databricks Lakeflow and the Data Intelligence Platform

So, how do the Data Intelligence Platform and Lakeflow help with each of the data engineering challenges discussed earlier?

- **Unified data engineering experience:** Lakeflow brings together all key aspects of data engineering, from ingestion (Lakeflow Connect) to transformation (Spark Declarative Pipelines) and orchestration (Lakeflow Jobs), in a single solution built on top of the Data Intelligence Platform. Through a centralized and simplified data platform, you can eliminate silos across teams, minimize tool sprawl and improve operational efficiencies.
- **Efficient ingestion, wide range of data connectors:** Lakeflow allows you to efficiently ingest data, only bringing in new data or table updates. With a growing set of native connectors for popular data sources, as well as a broad network of data ingestion partners, you can easily move data from siloed systems into your data platform. Ingesting and storing your data in Delta Lake while leveraging the reliability and scalability of the Data Intelligence Platform is the first step in extracting value from your data and accelerating innovation.
- **Real-time data stream processing:** Lakeflow simplifies development and operations by automating the production aspects associated with building and maintaining real-time data workloads. Spark Declarative Pipelines provides a declarative way to define streaming ETL pipelines, and Spark Structured Streaming helps build real-time applications for real-time decision-making.
- **Reliable data pipelines at scale:** Lakeflow uses smart autoscaling and auto-optimized resource management to handle high-scale workloads. With lakehouse architecture, the high scalability of data lakes are combined with the high reliability of data warehouses, thanks to Delta Lake — the storage format that sits at the foundation of the lakehouse.
- **Data quality:** High reliability — starting at the storage level with Lakehouse Storage and coupled with data quality-specific features offered by Spark Declarative Pipelines — ensures high data quality. These features include setting data "expectations" to handle corrupt or missing data, as well as automatic retries. In addition, both Lakeflow Jobs and Spark Declarative Pipelines provide full observability to data engineers, making issue resolution faster and easier.
- **Unified governance with secure data sharing**: Unity Catalog provides a single governance model for the entire platform, so every dataset and pipeline is governed consistently. Datasets are discoverable and can be securely shared with internal or external teams using Delta Sharing. In addition, because Unity Catalog is a cross-platform governance solution, it provides valuable lineage information, so it's easy to have a full understanding of how each dataset and table is used downstream and where it originates upstream.

Conclusion

As organizations strive to innovate by leveraging their data, data engineering is a focal point for success by delivering reliable, real-time data pipelines that make AI possible. With Databricks Lakeflow, built on lakehouse architecture and powered by data intelligence, data engineers are set up for success in dealing with the critical challenges posed in the modern data landscape. By leaning on the advanced capabilities of Lakeflow and the entire Data Intelligence Platform, data engineers don't need to spend as much time managing complex pipelines or dealing with reliability, scalability and data quality issues. Instead, they can focus on innovation and bringing more value to the organization.

In the next section, we describe best practices for data engineering and end-to-end use cases drawn from real-world examples. From data ingestion and real-time processing to orchestration and data federation, you'll learn how to apply proven patterns and make the best use of the different capabilities of Lakeflow and the Data Intelligence Platform.

02

Guidance and Best Practices

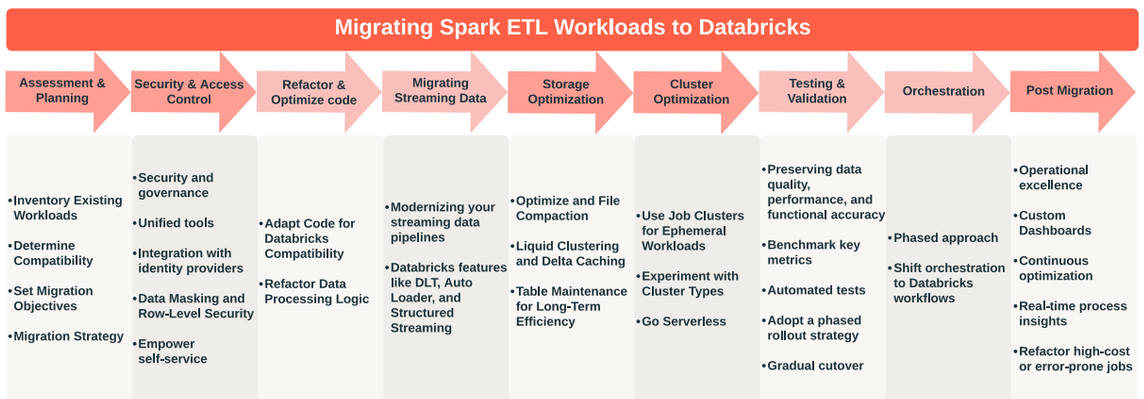
Migrating Apache Spark™ ETL Workloads to Databricks

By Dinesh Kumar

Introduction

Migrating Spark ETL workloads to Databricks unlocks faster performance, lowers costs and enhances scalability. With built-in support for **Delta Lake**, automated compute management and an optimized **Spark** engine, Databricks simplifies and modernizes your data pipelines.

In this chapter, I will cover key best practices for a smooth and efficient migration, ranging from workload assessment to performance tuning. Whether you're lifting and shifting, refactoring or re-architecting your pipelines, these steps will ensure you unlock the full potential of Databricks.



Assessment and planning for Databricks migration

A successful migration to Databricks begins with a structured approach, including a comprehensive inventory of existing workloads, assessing compatibility and setting clear migration objectives.

1. **Inventory existing workloads:** Classify your workloads as batch or streaming and document key assets for migration, including jobs, workflows, parameters, resource utilization, execution frequency and data consumers. A comprehensive inventory will give you a clear overview of the migration-ready components.
2. **Determine compatibility:** Identify any proprietary or environment-specific features that might not directly translate to Databricks. You must also identify and assess cloud providers and external service dependencies to ensure smooth integration during migration.
3. **Set migration objectives:** Define clear goals and success criteria for the migration, including cost of ownership, governance, performance improvements, scalability and architecture simplicity
4. **Choose the right migration strategy:** The key to a successful Databricks migration is choosing the right strategy: Lift-and-Shift, Refactor or Re-architect. Each option has its trade-offs regarding effort, flexibility and long-term scalability. Lift-and-Shift is the quickest strategy, involving minimal changes to workloads. It lets you immediately leverage Databricks performance and cost benefits, delivering a fast ROI. While it may not be a fully optimized architecture, it accelerates value realization and sets the stage for future enhancements.

For more details, please check out our documentation here.

Security and access control

As your data scales on Databricks, robust security and governance become critical. Databricks unified tools help centralize data permissions, enabling decentralized, secure innovation across teams.

Unity Catalog is the foundation for unifying governance, providing fine-grained permissions across catalogs, schemas, tables, views and columns. It also captures runtime data lineage, enables easy data discovery via the catalog explorer and allows for monitoring through audit logs. Unity Catalog also helps protect sensitive data with built-in data masking and row-level security.

Identity federation simplifies centralized user and group management by integrating with identity providers like Azure AD or Okta. SCIM provisioning automates syncing users and groups, supporting seamless access management and Single Sign-On (SSO) for an enhanced user experience.

Adopting the Data Mesh architecture for federated governance and assigning data ownership to domain-specific teams accelerates decision-making, empowers teams to manage their data and enhances agility and scalability.

Terraform templates will also help automate resource deployment and maintain scalability across the platform, ensuring infrastructure consistency and security.

Refactor and optimize code

Optimizing and refactoring your code is crucial for improving compatibility and performance on Databricks while setting up a solid foundation for long-term success.

ADAPT CODE FOR DATABRICKS COMPATIBILITY

While Lift-and-Shift can speed up deployment, legacy code might impact performance and stability. I'd like you to take this opportunity to audit and streamline your pipelines, identifying components that might not integrate seamlessly with Databricks. Common challenges include custom JARs, Hive UDFs and infrastructure-specific configurations that may need refactoring to align with Databricks' cloud-native environment. Proactively addressing these issues helps reduce technical debt and ensures smoother, more stable migrations.

REFACTOR DATA PROCESSING LOGIC

Refactoring legacy data processing logic modernizes your pipelines and simplifies the migration. Remorph Transpile and BladeBridge tools can automate SQL code conversion, reducing manual effort. Leveraging Delta Lake enhances reliability and performance with features like ACID transactions and schema enforcement, ensuring compatibility and providing a scalable foundation for future growth.

Migrate streaming data to Databricks: Simplify and optimize your pipelines

Migrating streaming data pipelines to Databricks can unlock faster insights, improve performance and reduce operational costs. Whether you're moving away from legacy stream processors or consolidating multiple tools, Databricks offers a seamless migration path, providing key features designed for streamlined performance, scalability and ease of management.

SPARK DECLARATIVE PIPELINES (SDP): SIMPLIFYING STREAMING DATA PIPELINES

SDP simplifies building and managing streaming pipelines by allowing you to use SQL or Python in a declarative format. It automatically handles orchestration, retries and dependencies so that you can focus on pipeline logic instead of operational challenges.

SDP supports batch and streaming data, streamlining the migration of streaming workloads and improving pipeline maintainability by reducing the need for multiple tools or custom solutions.

KEY FEATURES OF SPARK DECLARATIVE PIPELINES

- **Declarative streaming pipelines:** Simplifies pipeline creation by eliminating low-level details like scheduling, retries and state management, allowing developers to focus on business logic
- **Batch and continuous modes:** Supports batch and real-time streaming, offering flexibility for various data processing needs
- **Data quality checks:** Integrated data quality checks to ensure accurate data processing
- **Schema evolution:** Adapting to data model changes reduces manual effort and ensures pipeline robustness
- **Multi-catalog publishing:** Allows data publishing to multiple catalogs for flexible access and management

AUTO LOADER: SCALABLE INGESTION WITH MINIMAL OVERHEAD

Auto Loader simplifies incremental file ingestion from cloud storage (e.g., S3, ADLS). It automatically infers schemas, adapts to schema changes in real time and handles high-volume, low-latency workloads. Auto Loader is ideal for streaming migrations, ensuring efficient, seamless data pipeline management.

STRUCTURED STREAMING: REAL-TIME ANALYTICS AT SCALE

Structured Streaming, built on Apache Spark, provides flexibility and resilience for your streaming architecture. Combined with **Delta Lake**, it offers low-latency processing, exactly-once semantics and ACID compliance, enabling scalable and reliable real-time data pipelines.

LAKEFLOW CONNECT: STREAMLINED INTEGRATION WITH EXTERNAL SOURCES

Lakeflow Connect provides fully managed connectors to easily ingest data from SaaS applications and databases into your lakehouse. Powered by Unity Catalog and serverless compute, it ensures fast, scalable and cost-effective incremental data ingestion, keeping your data fresh and ready for downstream use.

Optimizing data storage and access in Databricks

As data volumes grow, optimizing how data is stored and accessed becomes critical for maintaining performance, reducing costs and ensuring reliability. Databricks offers several advanced features to optimize Delta Lake and enhance performance at scale.

KEY OPTIMIZATIONS:

- **Optimize and file compaction:** Address small file issues with auto compaction and optimized writes, but still rely on OPTIMIZE for manual fine-tuning. In addition, **Liquid Clustering** and **Z-Ordering** can be used to adjust data organization based on predicted fields to improve performance further.
- **Disk Caching and Photon Execution Engine:** Delta Caching stores frequently accessed data in memory or SSDs to reduce query latency, and Photon Execution Engine runs workloads and queries more efficiently
- **Predictive optimization:** Removes the need to manage maintenance operations for Unity Catalog manually managed tables on Databricks

If further reference is needed, please check the best practices for performance efficiency, tune-file size and delta vacuum documentation.

Cluster management and optimization in Databricks

Optimizing cluster configurations is essential for balancing performance and cost in Databricks. A workload-aware approach is key and you should experiment with different cluster types and settings that best fit your pipeline requirements.

BEST PRACTICES:

- **Job clusters:** Use job clusters for ephemeral workloads to minimize idle costs
- **Cluster types:** Select memory-optimized VMs for cache-heavy tasks and compute-optimized VMs for high-throughput transformations
- **Auto-scaling:** Enable autoscaling to adjust resources based on workload demand automatically
- **Serverless:** Go serverless to eliminate cluster management overhead for ad hoc queries and BI dashboards

If further reference is needed, please check cluster configuration, cost optimization for the data lakehouse, Comprehensive Guide to Optimize Databricks, Spark, and Delta Lake Workloads and connect to serverless compute.

Testing and validation for ETL workload migration

Migrating ETL workloads isn't just about moving code — it's about preserving data quality, performance and functional accuracy. Automate tests to validate data accuracy and pipeline dependencies as well as benchmark key metrics to track performance improvements.

The recommended approach is automating testing with tools like Remorph Reconcile, DataCompy or SQLglot. These tools help reduce manual effort, boosting confidence in your migration's success.

It is also essential to adopt a phased rollout strategy, running the new and legacy systems side by side for validation and monitoring before entirely switching to Databricks. This approach ensures a low-risk migration with minimal disruption.

Workflow management and orchestration in Databricks

Shifting orchestration into Lakeflow Jobs can streamline ETL management, improve dependency handling and enable centralized monitoring. Databricks natively supports **Apache Airflow** via the DatabricksRunNowOperator, allowing you to migrate orchestration without breaking your existing DAGs. Over time, consider moving workflows to the Databricks native visual interface and YAML-based configuration for more flexibility and ease of use.

Monitoring and optimization post-migration

After migration, maintaining an efficient Databricks environment is crucial. Built-in tools like **Spark UI**, **Databricks Metrics UI**, **System Tables** and **Cluster Event Logs** can help you gain insights into execution, resource usage and cluster behavior.

As workloads stabilize, fine-tune performance by analyzing real-time data. Adjust resources by right-sizing clusters, refining autoscaling and optimizing partitioning or caching strategies to improve query performance and reduce I/O.

Additional references are available on create a dashboard, usage dashboard and system tables overview.

Conclusion

Migrating to Databricks requires careful planning, execution, and ongoing optimization. By following a structured approach, assessing workloads, optimizing code, and leveraging Databricks' powerful features, you can ensure a seamless migration that improves performance, scalability, and cost-efficiency without interrupting ongoing business. With Databricks, your organization can scale data processing pipelines while reducing operational costs, setting you up for short-term success and long-term growth.

How Observability in Lakeflow Helps You Build Reliable Data Pipelines

Lakeflow's suite of observability features provides data engineers with the tools to confidently maintain efficient, reliable and healthy data pipelines at scale.

By Joanna Zouhour and Theresa Hammer

As data volume grows, so do the risks for your data platform: from stale pipelines to hidden errors and runaway costs. Without observability integrated into your data engineering solution, you are flying blind and risking impacting not just the health and freshness of your data pipelines but also missing serious issues in your downstream data, analytics, and AI workloads. With Lakeflow, a Databricks unified and intelligent data engineering solution, you can easily tackle this challenge with built-in observability solutions in an intuitive interface directly within your ETL platform, on top of your data intelligence.

In this chapter, we will introduce Lakeflow's observability capabilities and show how to build reliable, fresh, and healthy data pipelines.

Observability is essential for data engineering

Observability for data engineering is the ability to discover, monitor, and troubleshoot systems to ensure the ETL operates correctly and effectively. It is the key to maintaining healthy and reliable data pipelines, surfacing insights and delivering trustworthy downstream analytics.

As organizations manage an increasingly growing number of business-critical pipelines, monitoring and ensuring the reliability of a data platform has become vital for a business. To tackle this challenge, more data engineers are recognizing and seeking the benefits of observability. According to Gartner, 65% of data and analytics leaders expect data observability to become a core part of their data strategy within two years. Data engineers who want to stay current and find ways to improve productivity while driving stable data at scale should implement observability practices in their data engineering platform.

Establishing the right observability for your organization involves bringing the following key capabilities:

- **End-to-end visibility at scale:** Eliminate blind spots and uncover system insights by easily viewing and analyzing your jobs and data pipelines in one single location
- **Proactive monitoring and early failure detection:** Identify potential issues as soon as they arise, before they impact anything downstream
- **Troubleshooting and optimization:** Fix problems to ensure the quality of your outputs and optimize your system's performance to improve operational costs

Read on to see how Lakeflow supports all of these in a single experience. Watch the video.

End-to-end visibility at scale into jobs and pipelines

Effective observability begins with complete visibility. Lakeflow comes with a variety of out-of-the-box visualizations and unified views to help you stay on top of your data pipelines and make sure your entire ETL process is running smoothly.

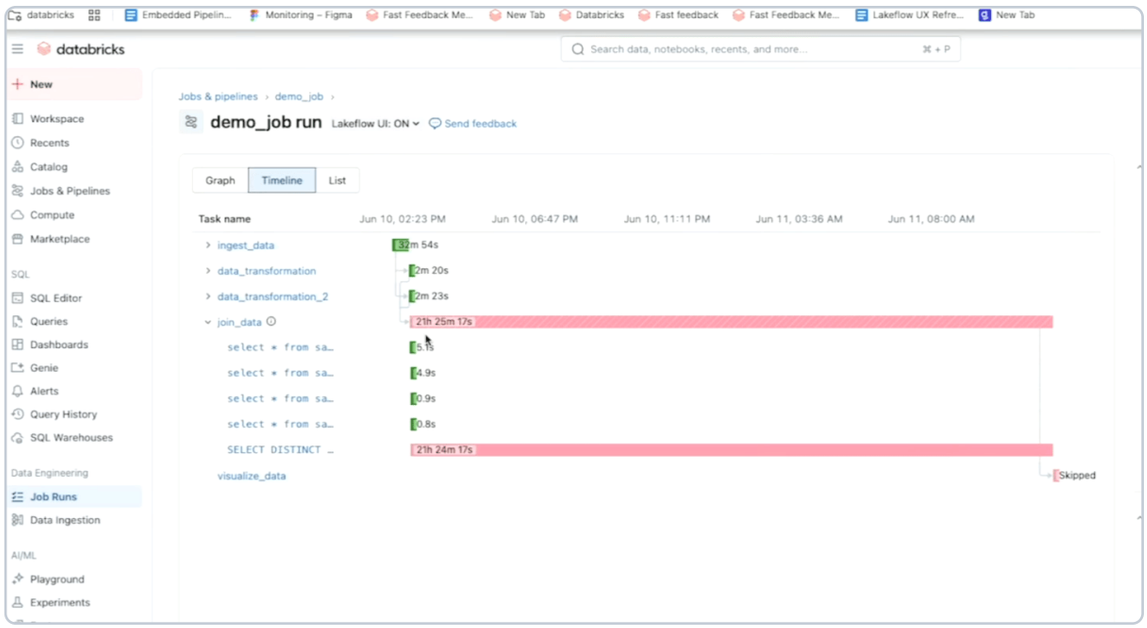
FEWER BLIND SPOTS WITH A CENTRALIZED AND GRANULAR VIEW OF YOUR JOBS AND PIPELINES

The Jobs and Pipelines page centralizes access to all your jobs, pipelines, and their run history across the workspace. This unified overview of your runs simplifies the discovery and management of your data pipelines and makes it easier to visualize executions and track trends for more proactive monitoring.

Looking for more information about your jobs? Just click on any job to go to a dedicated page that features a Matrix View and highlights key details like status, duration, trends, warnings, and more. You can:

- Easily drill down into a specific job run for additional insights, such as the graph view to visualize dependencies or point of failure
- Zoom in to see the task level (like pipeline, notebook output, etc.) for more details, such as streaming metrics (available in Public Preview)

Lakeflow also offers a dedicated Pipeline Run page where you can easily monitor the status and metrics and track the progress of your pipeline execution across tables.

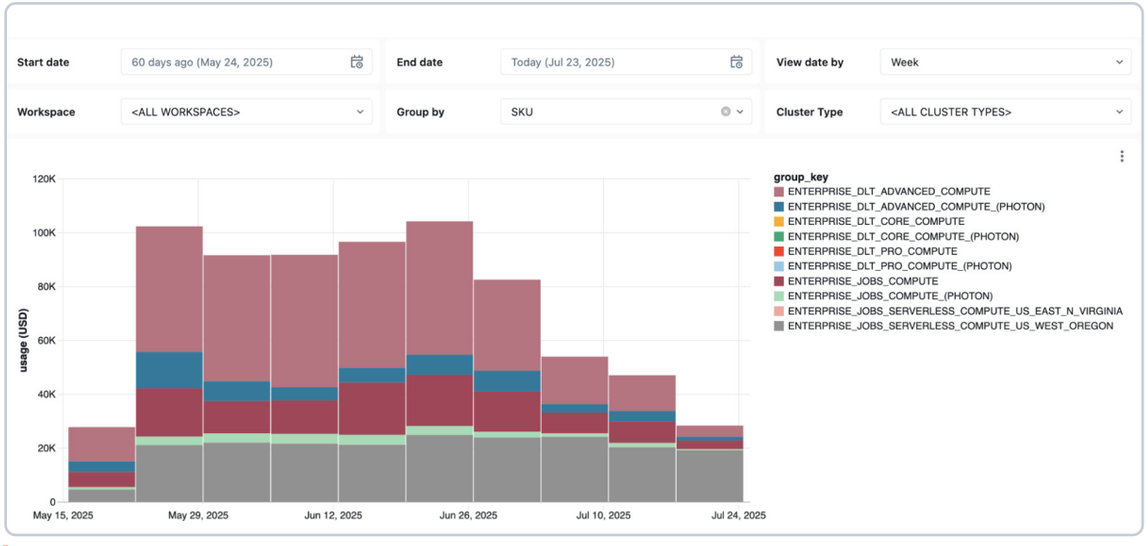


Easily go from an overview of your jobs and pipeline runs to more detailed information on jobs and tasks

MORE INSIGHTS WITH VISUALIZATION OF YOUR DATA AT SCALE

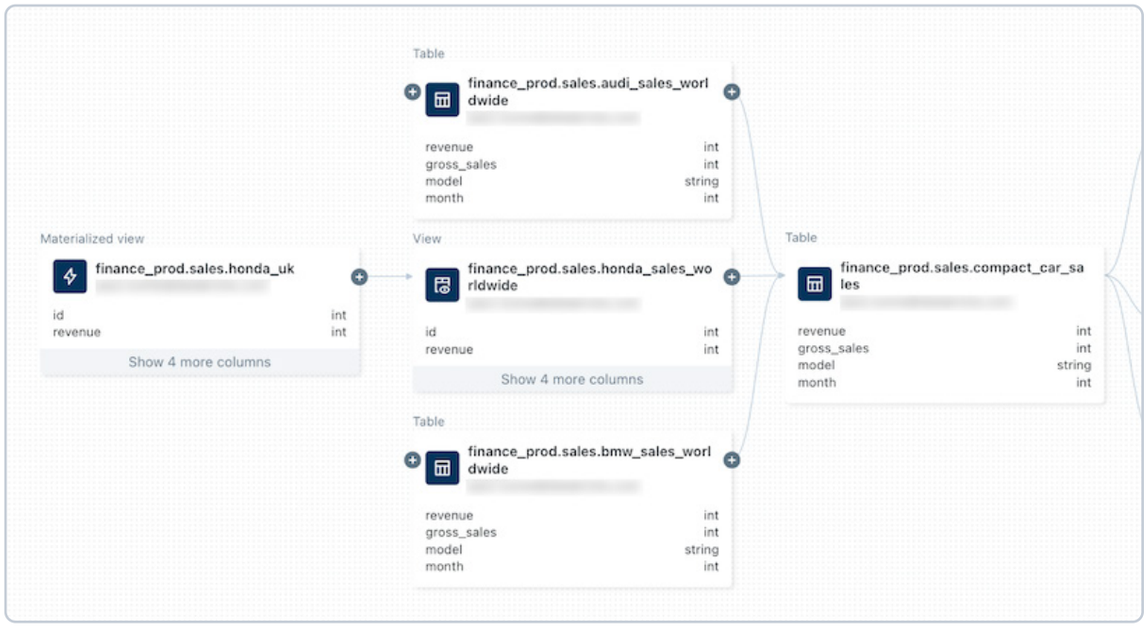
In addition to these unified views, Lakeflow provides historical observability for your workloads to get insights into your usage and trends. Using System Tables, Databricks-managed tables that track and consolidate every job and pipeline created across all workspaces in a region. You can build detailed dashboards and reports to visualize your jobs and pipelines' data at scale. With the recently updated interactive dashboard template for Lakeflow System Tables, it's much easier and faster to:

- Track execution trends: easily surface insights around job behavior over time for better data-driven decisions
- Identify bottlenecks: detect potential performance issues (covered in more detail in the following section)
- Cross-reference with billing: improve cost monitoring and avoid billing surprises



Build dashboards using system tables in Lakeflow and get a high-level overview of your jobs and pipelines' health

Visibility extends beyond just the task or job level. Lakeflow's integration with Unity Catalog, the Databricks unified governance solution, helps complete the picture with a visual of your entire data lineage. This makes it easier to trace data flow and dependencies and get the full context and impact of your pipelines and jobs in one place.



Track data lineage using Databricks Unity Catalog

Proactive monitoring, early detection of job failures, troubleshooting and optimization

As data engineers, you're not just responsible for monitoring your systems. You also need to be proactive about any issues or performance gaps that might come up in your ETL development and address them before they impact your outputs and costs.

PROACTIVE ALERTING TO CATCH THINGS EARLY

With Lakeflow's native notifications, you can choose if and how to be alerted about critical job errors, durations or backlogs via Slack, email or even PagerDuty. Event hooks in Spark Declarative Pipelines (currently in Public Preview) give you even more flexibility by defining custom Python callback functions so you decide what to monitor or when to be alerted on specific events.

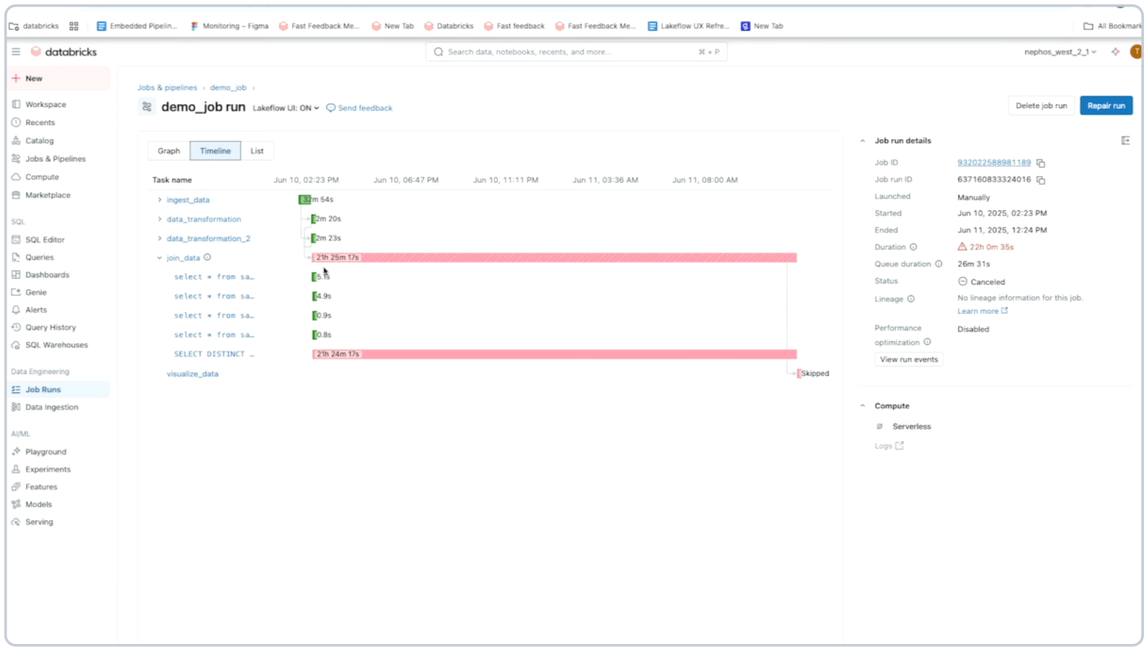
FASTER ROOT CAUSE ANALYSIS FOR QUICK REMEDIATION

Once you receive the alert, the next step is to understand why something went wrong.

Lakeflow allows you to jump from the notification directly into the detailed view of a specific job or task failure for in-context root cause analysis. The level of detail and flexibility with which you can see your workflow data allows you to easily identify what exactly is responsible for the error.

For instance, using the matrix view of a job, you can track failure and performance patterns across tasks for one specific workflow. Meanwhile, the timeline (Gantt) view breaks down the duration of each task and query (for serverless jobs) so you can spot slow performance issues in one job and dig deeper for root causes using Query Profiles. As a reminder, Databricks Query Profiles show a quick overview of your SQL, Python and Declarative Pipeline executions, making it easy to identify bottlenecks and optimize workloads in your ETL platform.

You can also rely on System Tables to make root cause analysis easier by building dashboards that highlight irregularities across your jobs and their dependencies. These dashboards help you quickly identify not just failures but also performance gaps or latency improvement opportunities, such as latency P50/P90/P99 and cluster metrics. To complement your analysis, you can leverage lineage data and query history system tables so you can easily track upstream errors and downstream impacts through data lineage.

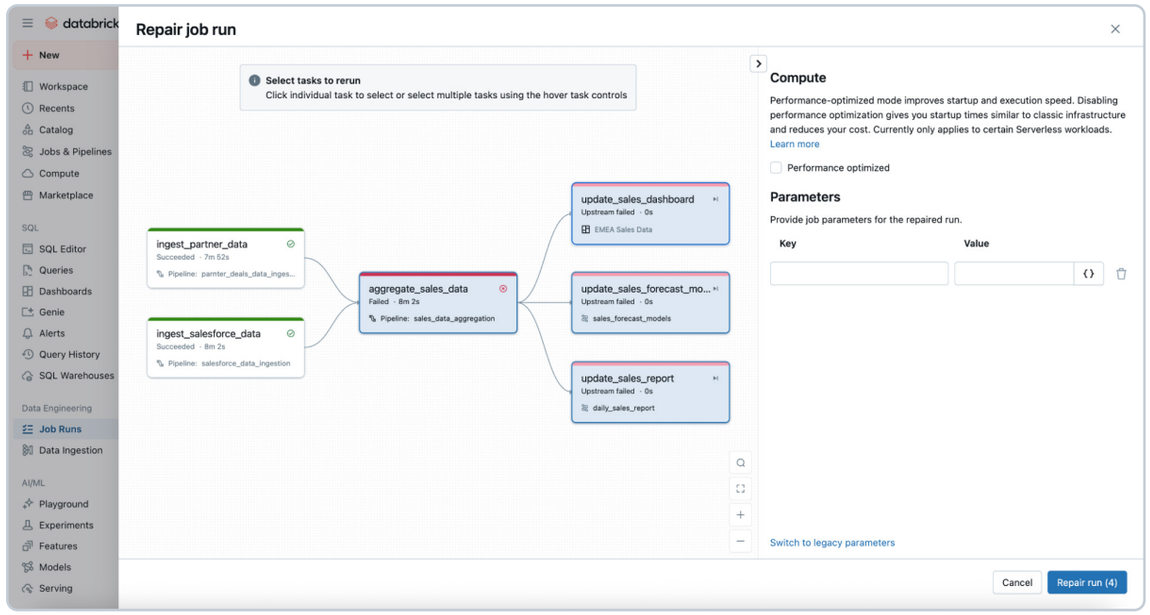


Easily find the root cause of your issues with Query History System Tables

DEBUGGING AND OPTIMIZATION FOR RELIABLE PIPELINES

In addition to root cause analysis, Lakeflow gives you tools for quick troubleshooting, whether it's a compute resource issue or a configuration error. Once you've addressed the issue, you can run the failed tasks and their dependencies without re-running the entire job, saving you computational resources.

Facing more complex troubleshooting use cases? Databricks Assistant, our AI-powered assistant (currently in Public Preview), provides clear insights and helps you diagnose errors in your jobs and pipelines.



Easily troubleshoot issues in your data pipelines with the "Repair job run" functionality

KEY OBSERVABILITY PILLARS	LAKEFLOW CAPABILITIES
End-to-end visibility at scale	▪ Unified Views ▪ System Tables for Jobs and Pipelines ▪ Data Lineage with Unity Catalog
Proactive Monitoring and Early Failure Detection	▪ Alerts and Notifications ▪ Event Hooks ▪ Health metrics (coming soon)
Troubleshooting and Performance Optimization	▪ Matrix views ▪ Gantt views ▪ Query Profiles (System Tables) ▪ Re-run tasks ▪ Databricks Assistant

Summary of Lakeflow's observability capabilities

Start building reliable data engineering with Lakeflow

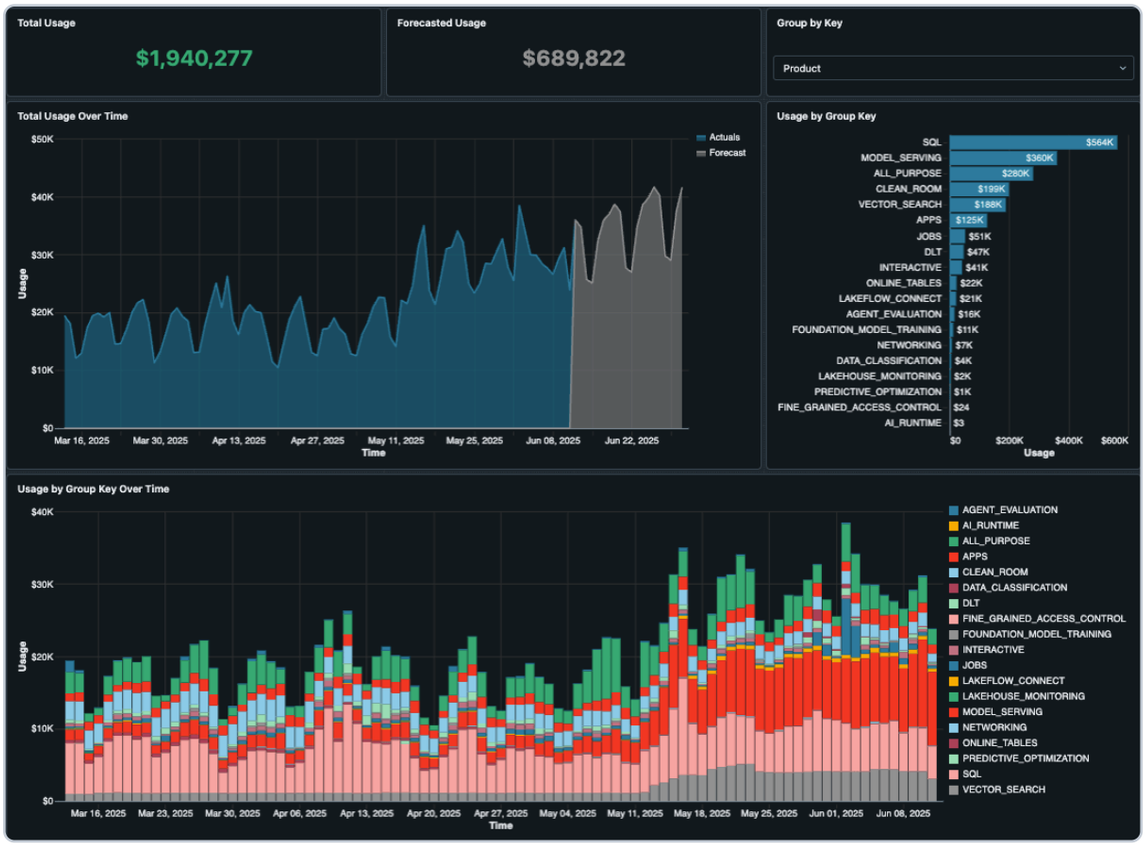
Lakeflow offers the support you need to ensure your jobs and pipelines run smoothly, are healthy, and operate reliably at scale. Try our built-in observability solutions and see how you can build a data engineering platform ready for your data intelligence efforts and business needs.

- Learn more about Lakeflow Observability
- Watch the Lakeflow Observability session from our Data + AI summit on demand

A Cost Maturity Journey With Databricks

Use a structured process to assess Databricks cost control maturity, identify usage patterns, enforce budgets, optimize workloads and reduce unnecessary spend

By Zach King and Rajneesh Arora



Introduction: The importance of FinOps in data and AI environments

Companies across every industry have continued to prioritize optimization and the value of *doing more with less*. This is especially true of digital native companies in today's data landscape, which yields higher and higher demand for AI and data-intensive workloads. These organizations manage thousands of resources in various cloud and platform environments. In order to innovate and iterate quickly, many of these resources are democratized across teams or business units. However, higher velocity for data practitioners can lead to chaos unless balanced with careful cost management.

Digital native organizations frequently employ central platform, DevOps or FinOps teams to oversee the costs and controls for cloud and platform resources. Formal practice of cost control and oversight, popularized by The FinOps Foundation™, is also supported by Databricks with features such as tagging, budgets, compute policies and more. Nonetheless, the decision to prioritize cost management and establish structured ownership does not create cost maturity overnight. The methodologies and features covered in this chapter enable teams to incrementally mature cost management within the Data Intelligence Platform.

What we'll cover:

- Cost Attribution: Reviewing the key considerations for allocating costs with tagging and budget policies
- Cost Reporting: Monitoring costs with Databricks AI/BI dashboards
- Cost Control: Automatically enforcing cost controls with Terraform, Compute Policies, and Databricks Asset Bundles (DABs)
- Cost Optimization: Common Databricks optimization checklist items

Whether you're an engineer, architect or FinOps professional, this chapter will help you maximize efficiency while minimizing costs, ensuring that your Databricks environment remains both high-performing and cost-effective.

Technical solution breakdown

We will now take an incremental approach to implementing mature cost management practices on the Databricks Platform. Think of this as the "crawl, walk, run" journey to go from chaos to control. We will explain how to implement this journey step by step.

Step 1: Cost attribution

The first step is to correctly assign expenses to the right teams, projects or workloads. This involves efficiently tagging all the resources (including serverless compute) to gain a clear view of where costs are being incurred. Proper attribution enables accurate budgeting and accountability across teams.

Cost attribution can be done for all compute SKUs with a tagging strategy, whether for a classic or serverless compute model. Classic compute (workflows, Declarative Pipelines, SQL Warehouse, etc.) inherits tags on the cluster definition, while serverless adheres to Serverless Budget Policies (AWS | Azure | GCP).

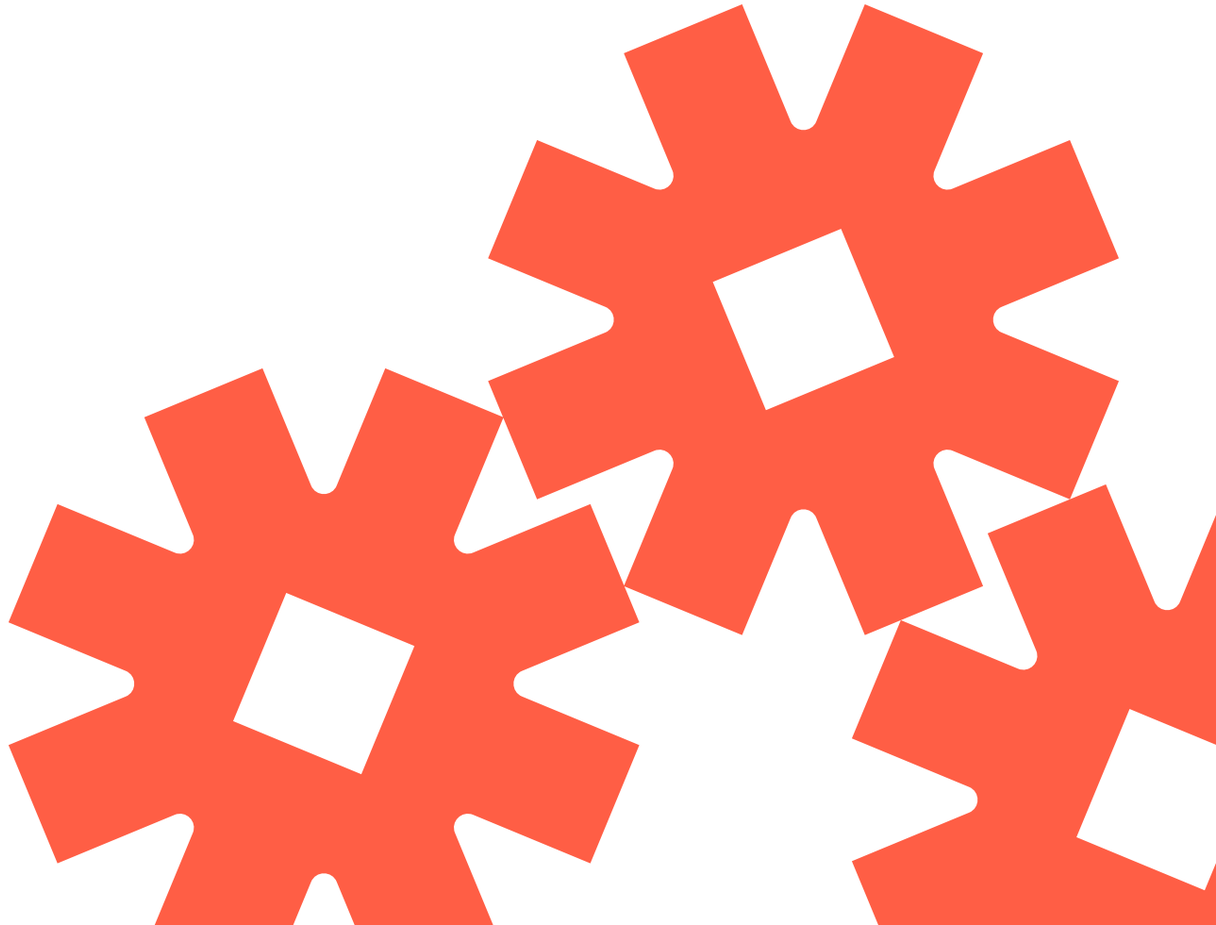
In general, you can add tags to two kinds of resources:

1. Compute Resources: Includes SQL Warehouse, jobs, instance pools, etc.
2. Unity Catalog Securables: Includes catalog, schema, table, view, etc.

Tagging for both types of resources would contribute to effective governance and management:

1. Tagging the compute resources has a direct impact on cost management
2. Tagging Unity Catalog securables helps with organizing and searching those objects, but this is outside the scope of this chapter

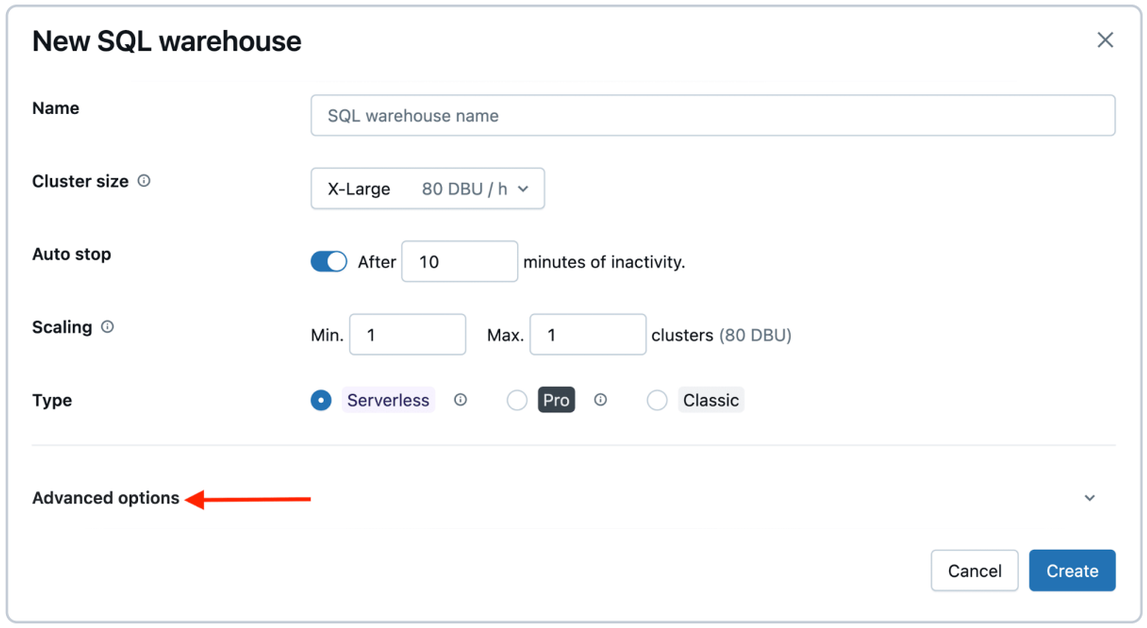
Refer to this article (AWS | AZURE | GCP) for details about tagging different compute resources and this article (AWS | Azure | GCP) for details about tagging Unity Catalog securables.



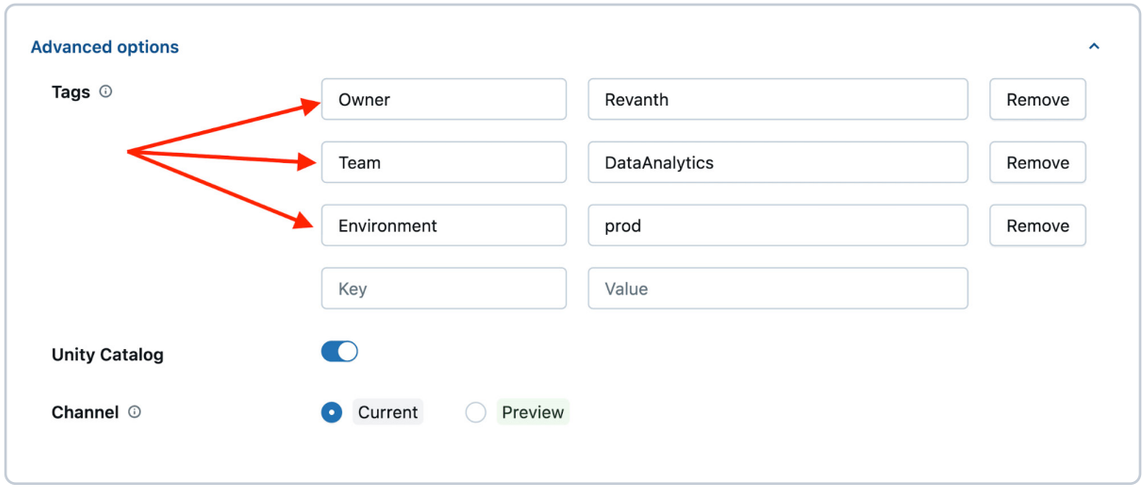
TAGGING CLASSIC COMPUTE

For classic compute, tags can be specified in the settings when creating the compute. Below are some examples of different types of compute to show how tags can be defined for each, using both the UI and the Databricks SDK.

SQL Warehouse Compute:



You can set the tags for a SQL Warehouse in the Advanced Options section.

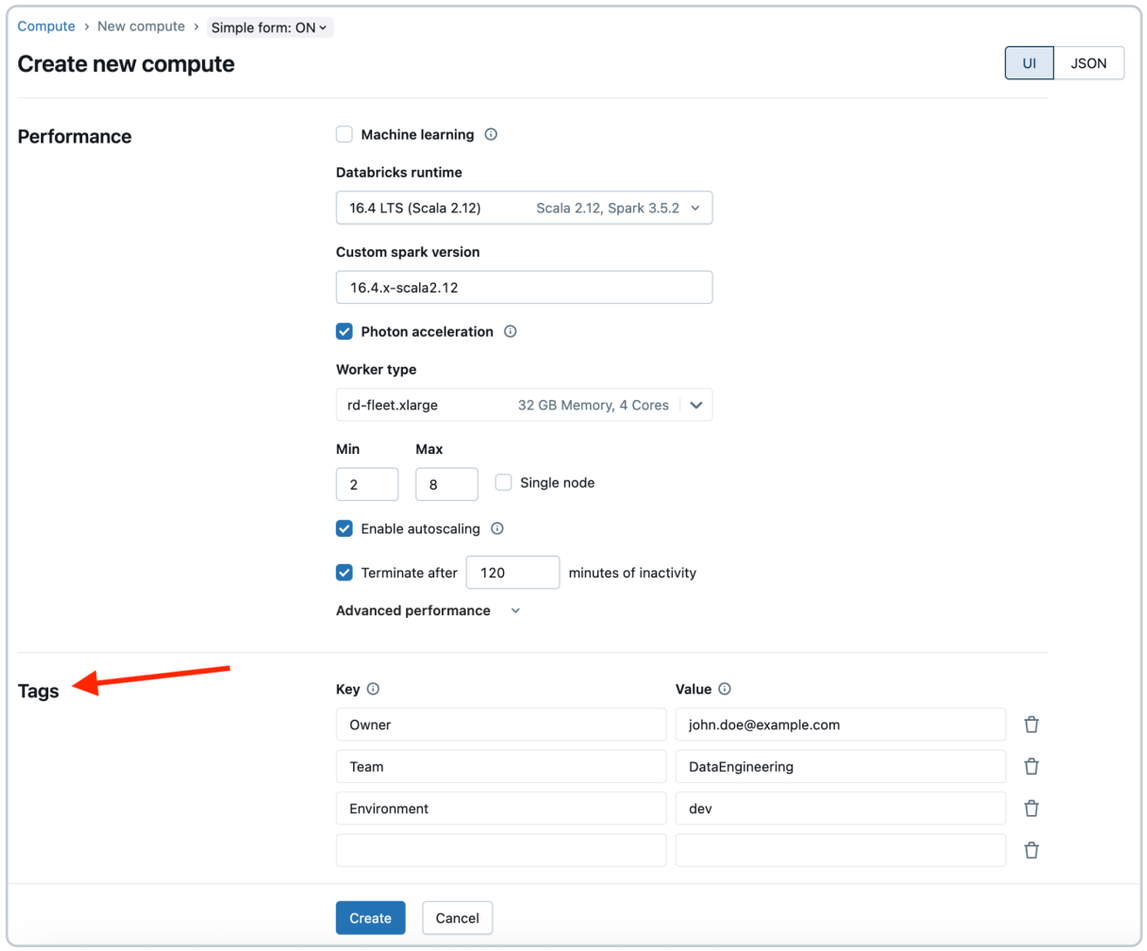


With Databricks SDK:

PYTHON

```
from databricks.sdk import WorkspaceClient
from databricks.sdk.service.sql import EndpointTags, EndpointTagPair

w = WorkspaceClient()

def update_warehouse_tags(warehouse_id: str, tags: dict):
    """Update the tags for a warehouse.

    Args:
        warehouse_id: The ID of the warehouse to update.
        tags: A dictionary of tags to update.
    """
    warehouse = w.warehouses.get(warehouse_id)
    current_tags = {}
    if warehouse.tags and warehouse.tags.custom_tags:
        for tag in warehouse.tags.custom_tags:
            current_tags[tag.key] = tag.value

    current_tags.update(tags)
    pairs = [EndpointTagPair(key=key, value=value) for key, value in
current_tags.items()]
    w.warehouses.edit_and_wait(warehouse_id,
tags=EndpointTags(custom_tags=pairs))
```

All-Purpose Compute:

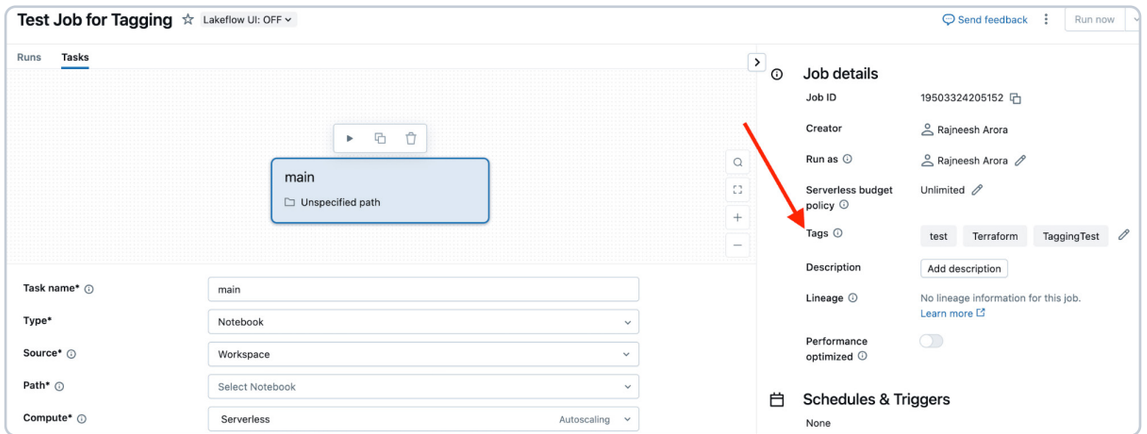


With Databricks SDK:

PYTHON

```python
def update_ap_cluster_tags(cluster_id: str, tags: dict):
    """Update the tags for an AP cluster.

    Args:
        cluster_id: The ID of the cluster to update.
        tags: A dictionary of tags to update.
    """
    cluster = w.clusters.get(cluster_id)
    current_tags = {}
    if cluster.custom_tags:
        current_tags = cluster.custom_tags

    current_tags.update(tags)
    cluster.custom_tags = current_tags
    w.clusters.update(cluster_id, "custom_tags",
cluster=UpdateClusterResource(custom_tags=current_tags))
```

Job Compute:

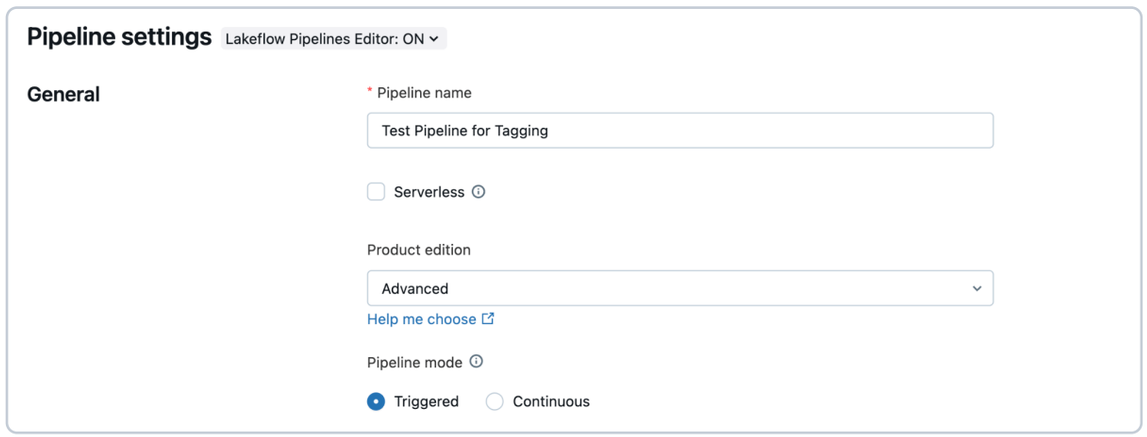


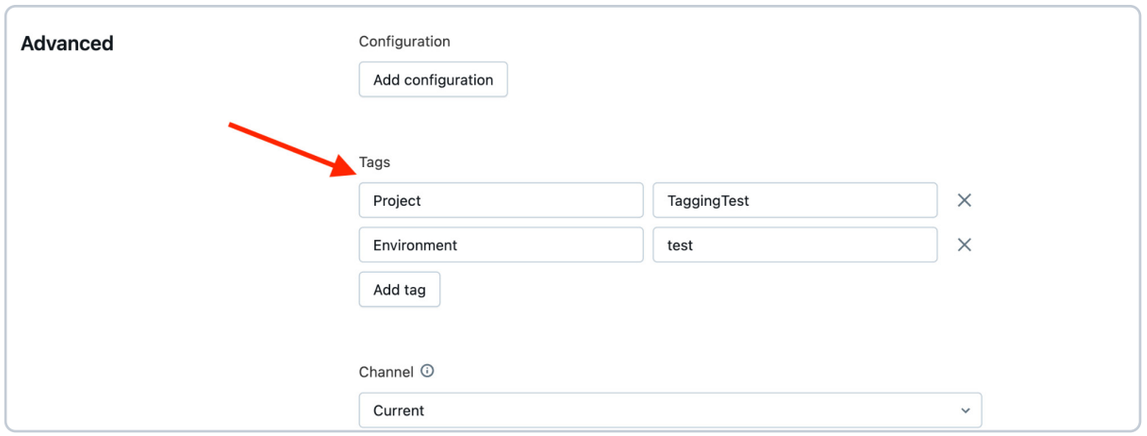
With Databricks SDK:

PYTHON

```python
def update_job_tags(job_id: str, tags: dict):
    """Update the tags for a job.

    Args:
        job_id: The ID of the job to update.
        tags: A dictionary of tags to update.
    """
    job = w.jobs.get(job_id)
    settings = job.settings
    if not job.settings.tags:
        settings.tags = tags
    else:
        settings.tags.update(tags)
    w.jobs.update(job_id, new_settings=settings)
```

Declarative Pipelines:





TAGGING SERVERLESS COMPUTE

For serverless compute, you should assign tags with a budget policy. Creating a policy allows you to specify a policy name and tags of string keys and values.

It's a three-step process:

- **Step 1:** Create a budget policy (workspace admins can create one and users with Manage access can manage them)
- **Step 2:** Assign Budget Policy to users, groups and service principals
- **Step 3:** Once the policy is assigned, the user is required to select a policy when using serverless compute. If the user has only one policy assigned, that policy is automatically selected. If the user has multiple policies assigned, they have the option to choose one of them.

You can refer to details about serverless Budget Policies (BP) in these articles (AWS | Azure | GCP).

Certain aspects to keep in mind about Budget Policies:

- A Budget Policy is very different from Budgets (AWS | Azure | GCP). We will cover Budgets in Step 2: Cost Reporting.
- Budget Policies exist at the account level, but they can be created and managed from a workspace. Admins can restrict which workspaces a policy applies to by binding it to specific workspaces.
- A Budget Policy only applies to serverless workloads. Currently, at the time of writing this chapter, it applies to notebooks, jobs, pipelines, serving endpoints, apps and Vector Search endpoints.
- Let's take an example of jobs having a couple of tasks. Each task can have its own compute, while BP tags are assigned at the job level (and not at the task level). So, there is a possibility that one task runs on serverless while the other runs on general non-serverless compute. Let's see how Budget Policy tags would behave in the following scenarios:
  - Case 1: Both tasks run on serverless
    - In this case, BP tags would propagate to system tables
  - Case 2: Only one task runs on serverless
    - In this case, BP tags would also propagate to system tables for the serverless compute usage, while the classic compute billing record inherits tags from the cluster definition
  - Case 3: Both tasks run on non-serverless compute
    - In this case, BP tags would not propagate to the system tables

With Terraform:

BASH

```
resource "databricks_budget_policy" "this" {
  policy_name = "default-policy"
  custom_tags = [
    {
      key   = "ServerlessPolicy"
      value = "default-policy"
    },
    {
      key   = "Environment"
      value = "prod"
    },
    {
      key   = "CostCenter"
      value = "CC1234"
    },
  ]

  # Optional list of workspaces this policy is bound to.
  # Empty list means this policy is open to any workspace in the
account.
  binding_workspace_ids = [
    databricks_mws_workspaces.prod.workspace_id
  ]
}
```

Best practices related to tags

General:
(Everyone should start w/ these)
- Business Unit
- Project
- *(optional)* Environment (dev/test/prod)

High Specificity:

Builder (SI partner name, Built on, etc.), Version, Role, Product, Service, Customer, Other / Custom

- It's recommended that everyone apply General Keys. For organizations that want more granular insights, they should apply high-specificity keys that are right for their organization.
- A business policy should be developed and shared among all users regarding the fixed keys and values that you want to enforce across your organization. In Step 4, we will see how Compute Policies are used to systematically control allowed values for tags and require tags in the right spots.
- Tags are case-sensitive. Use consistent and readable casing styles such as Title Case, PascalCase or kebab-case.
- For initial tagging compliance, consider building a scheduled job that queries tags and reports any misalignments with your organization's policy.
- It is recommended that every user have permission to at least one budget policy. That way, whenever the user creates a notebook/job/pipeline/etc., using serverless compute, the assigned BP is automatically applied.

SAMPLE TAG - KEY: VALUE PAIRINGS

KEY	BUSINESS UNIT	KEY	PROJECT
Value	101 (finance)	Value	Armadillo
	102 (legal)		BlueBird
	103 (product)		Rhino
	104 (sales)		Dolphin
	105 (field engineering)		Lion
	106 (marketing)		Eagle

Step 2: Cost reporting

SYSTEM TABLES

Next is cost reporting, or the ability to monitor costs with the context provided by Step 1. Databricks provides built-in system tables, like `system.billing.usage`, which is the foundation for cost reporting. System tables are also useful when customers want to customize their reporting solution.

For example, the account usage dashboard you'll see next is the Databricks AI/BI dashboard, so you can view all the queries and customize the dashboard to fit your needs very easily. If you need to write ad hoc queries against your Databricks usage, with very specific filters, this is at your disposal.

THE ACCOUNT USAGE DASHBOARD

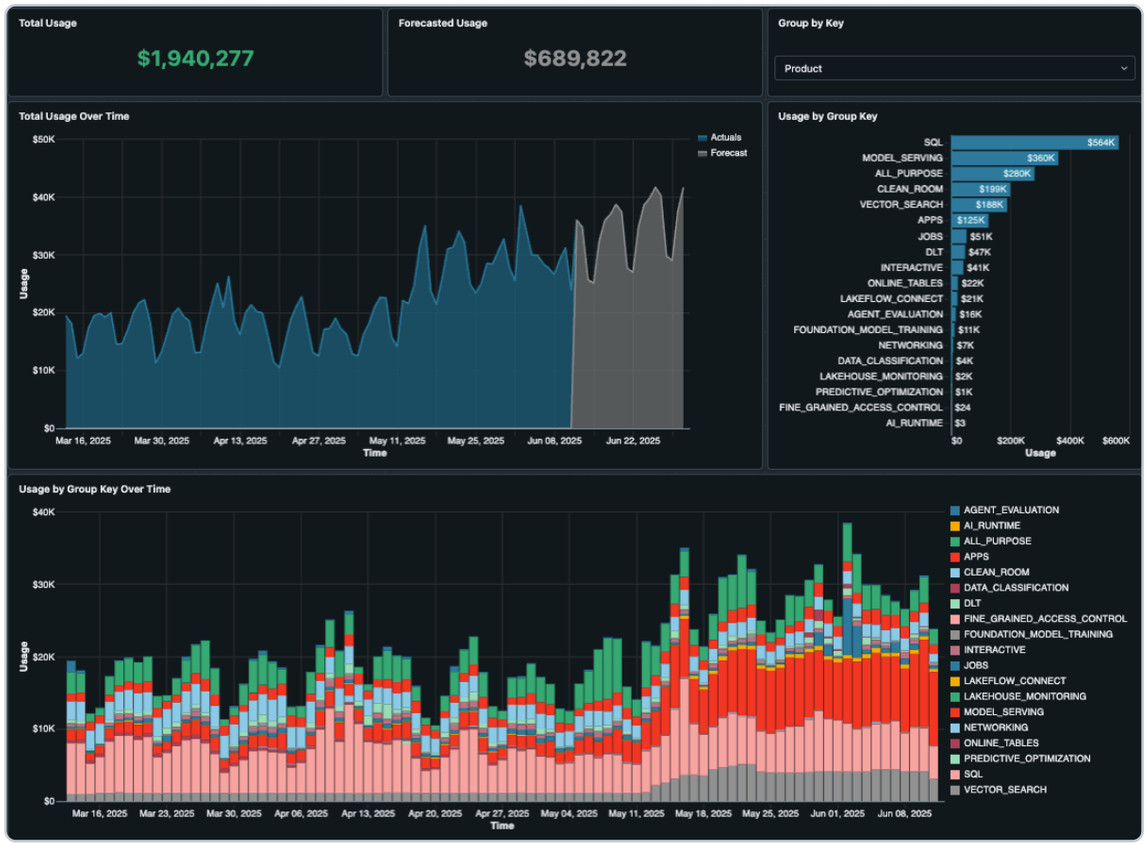
Once you have started tagging your resources and attributing costs to their cost centers, teams, projects or environments, you can begin to discover the areas where costs are the highest. Databricks provides a usage dashboard you can simply import to your workspace as an AI/BI dashboard, providing immediate out-of-the-box cost reporting.

A new version version 2.0 of this dashboard is available for preview, with several improvements shown below. Even if you have previously imported the Account Usage dashboard, please import the new version from GitHub today!

This dashboard provides a ton of useful information and visualizations, including data like the following:

- Usage overview, highlighting total usage trends over time, and by groups like SKUs and workspaces
- Top N usage that ranks top usage by selected billable objects such as job_id, warehouse_id, cluster_id, endpoint_id, etc.
- Usage analysis based on tags (the more tagging you do per Step 1, the more useful this will be)
- AI forecasts that indicate what your spending will be in the coming weeks and months

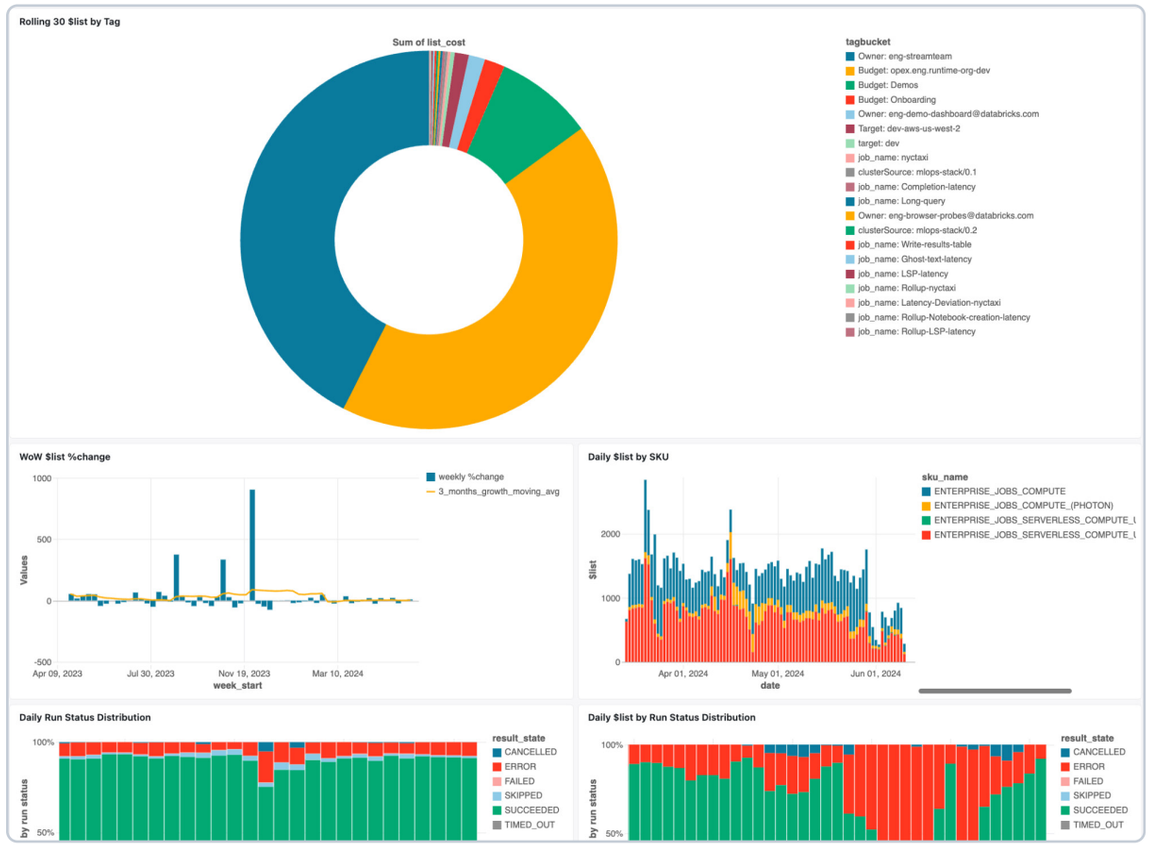
The dashboard also allows you to filter by date ranges, workspaces, products and even enter custom discounts for private rates. With so much packed into this dashboard, it really is your primary one-stop shop for most of your cost reporting needs.



JOBS MONITORING DASHBOARD

For Lakeflow Jobs, we recommend the Jobs System Tables AI/BI Dashboard to quickly see potential resource-based costs, as well as opportunities for optimization, such as:

- Top 25 jobs by potential savings per month
- Top 10 jobs with lowest avg CPU utilization
- Top 10 jobs with highest avg memory utilization
- Jobs with fixed number of workers last 30 days
- Jobs running on outdated DBR version Last 30 days



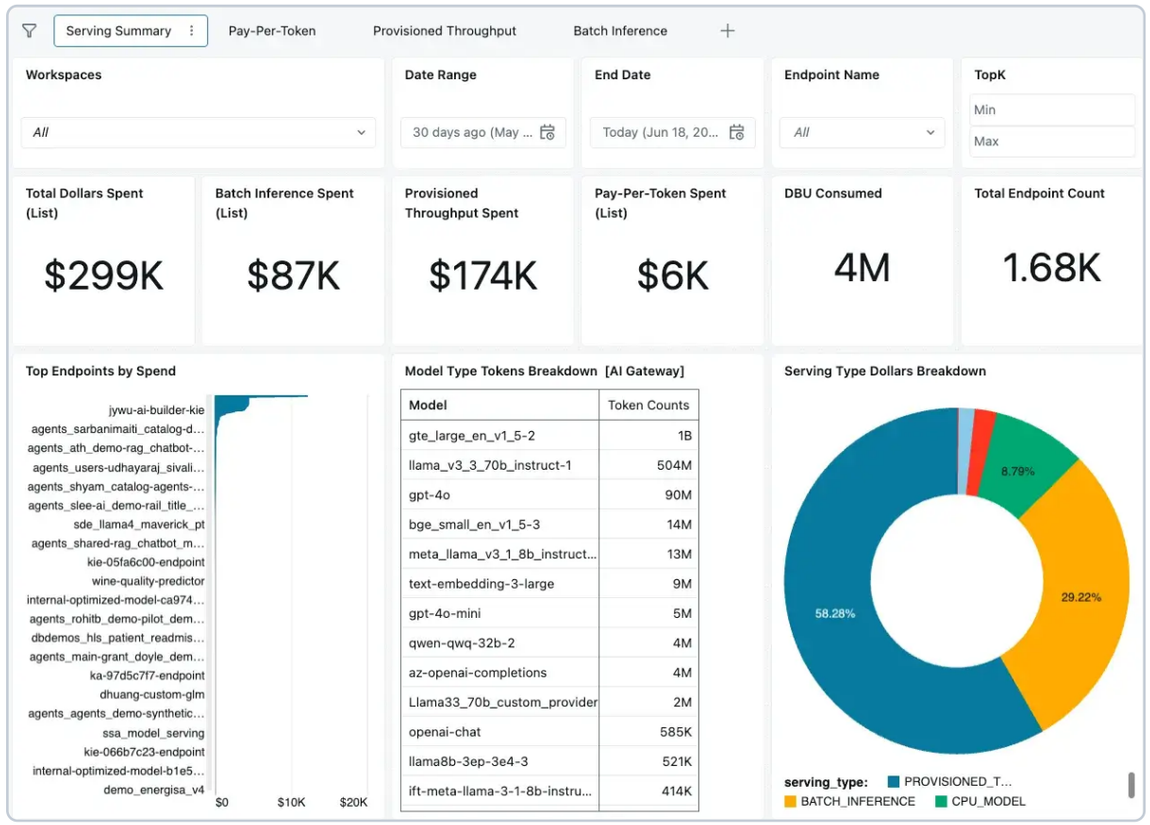
DBSQL MONITORING

For enhanced monitoring of Databricks SQL, refer to our SQL SME blog here. In this guide, our SQL experts will walk you through the Granular Cost Monitoring dashboard you can set up today to see SQL costs by user, source and even query-level costs.

databricks

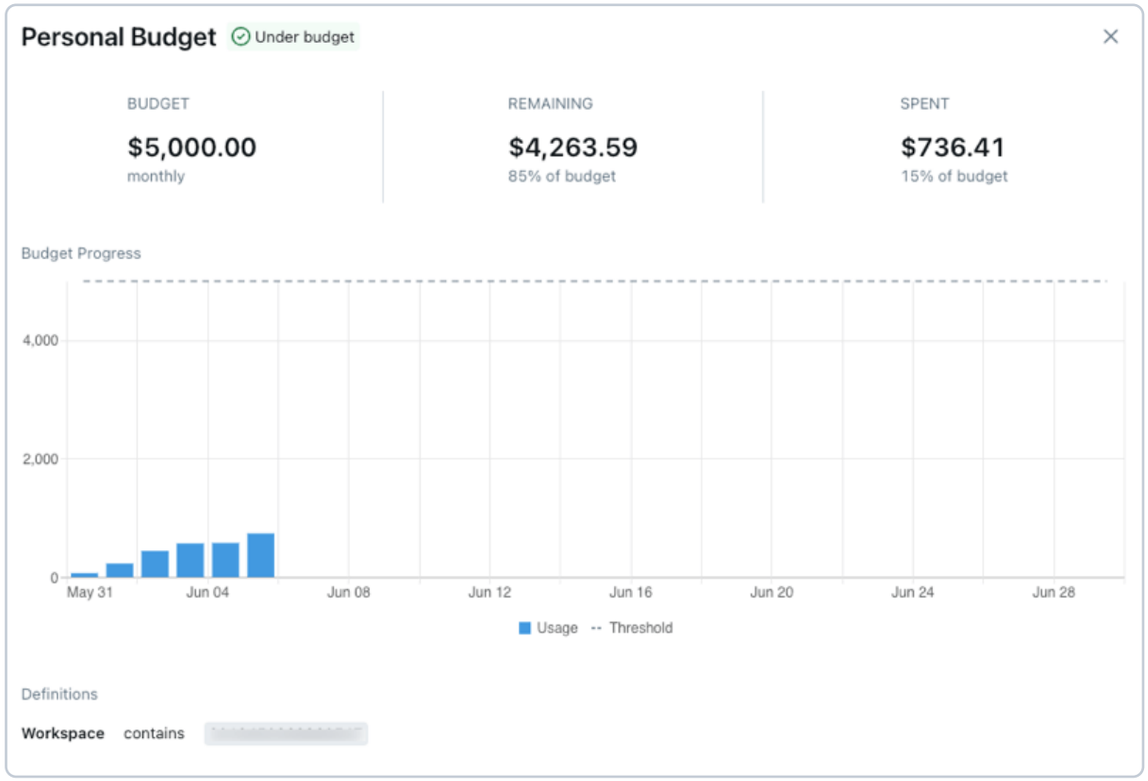
MODEL SERVING

Likewise, we have a specialized dashboard for monitoring costs for Model Serving. This is helpful for more granular reporting on batch inference, pay-per-token usage, provisioned throughput endpoints and more. For more information, see this related blog.



BUDGET ALERTS

Beyond Serverless Budget Policies for tagging serverless compute usage, Databricks offers Budgets (AWS | Azure | GCP) as a distinct feature for managing account-level spend. Budgets can be used to track account-wide spending or apply filters to track the spending of specific teams, projects or workspaces.



With budgets, you specify the workspaces or tags to match, set an amount in USD and define recipients to be notified when the budget is exceeded. This can be useful to reactively alert users when their spending has exceeded a given amount. Please note that budgets use the list price of the SKU.

Step 3: Cost controls

Next, teams must have the ability to set guardrails for data teams to be both self-sufficient and cost-conscious at the same time. Databricks simplifies this for both administrators and practitioners with Compute Policies (AWS | Azure | GCP).

Several attributes can be controlled with compute policies, including all cluster attributes as well as important virtual attributes such as `dbu_per_user`. We'll review a few of the key attributes to govern for cost control, specifically:

LIMITING DBU PER USER AND MAX CLUSTERS PER USER

Often, when creating compute policies to enable self-service cluster creation for teams, we want to control the maximum spending of those users. This is where one of the most important policy attributes for cost control applies: `dbus_per_hour`.

`dbus_per_hour` can be used with a `range` policy type to set lower and upper bounds on DBU cost of clusters that users can create. However, this only enforces max DBU per cluster that uses the policy. A single user with permission to this policy could still create many clusters and each is capped at the specified DBU limit.

JAVASCRIPT

```
"dbus_per_hour": {
   "type": "range",
   "maxValue": 100
}
```

To take this further, and prevent an unlimited number of clusters being created by each user, we can use another setting, `max_clusters_by_user`. This is actually a setting on the top-level compute policy rather than an attribute you would find in the policy definition.

CONTROL ALL-PURPOSE VS. JOB CLUSTERS

Policies should enforce which cluster type it can be used for, using the `cluster_type` virtual attribute, which can be one of: "all-purpose", "job" or "dlt". We recommend using `fixed` type to enforce exactly the cluster type that the policy is designed for when writing it:

JAVASCRIPT

```
"cluster_type": {"type": "fixed", "value": "job"}
```

A common pattern is to create separate policies for jobs and pipelines versus all-purpose clusters, setting `max_clusters_by_user` to 1 for all-purpose clusters (e.g., how the Databricks default Personal Compute policy is defined) and allowing more clusters per user for jobs.

ENFORCE INSTANCE TYPES

VM instance types can be conveniently controlled with the `allowlist` or `regex` type. This allows users to create clusters with some flexibility in the instance type without being able to choose sizes that may be too expensive or outside their budget.

JAVASCRIPT

```
"node_type_id": {
   "type": "allowlist",
   "values": [
       "i3.xlarge",
       "i3.2xlarge",
       "i3.4xlarge"
   ],
   "defaultValue": "i3.xlarge"
},
"driver_node_type_id": {
   "type": "regex",
   "pattern": "[rmci][3-7][dg]?.[2-4]?xlarge"
}
```

ENFORCE LATEST DATABRICKS RUNTIMES

It's important to stay up-to-date with newer Databricks Runtimes (DBRs) and, for extended support periods, consider Long-Term Support (LTS) releases. Compute policies have several special values to easily enforce this in the `spark_version` attribute, and here are a few to be aware of:

- `auto:latest-lts:` Maps to the latest long-term support (LTS) Databricks Runtime version
- `auto:latest-lts-ml:` Maps to the latest LTS Databricks Runtime ML version
- Or `auto:latest` and `auto:latest-ml` for the latest Generally Available (GA) Databricks Runtime version (or ML, respectively), which may not be LTS
  - Note: These options may be useful if you need access to the latest features before they reach LTS

We recommend controlling the `spark_version` in your policy using an `allowlist` type:

JAVASCRIPT

```
"spark_version": {
   "type": "allowlist",
   "values": [
       "auto:latest-lts",
       "auto:latest-lts-ml",
       "auto:latest",
       "auto:latest-ml"
   ],
   "defaultValue": "auto:latest-lts"
}
```

SPOT INSTANCES

Cloud attributes can also be controlled in the policy, such as enforcing instance availability of spot instances with fallback to on-demand. Note that whenever using spot instances, you should always configure the "first_on_demand" to at least 1 so the driver node of the cluster is always on-demand.

On AWS:

JAVASCRIPT

```
"aws_attributes.first_on_demand": { "type": "fixed", "value": 1 },
"aws_attributes.availability": { "type": "fixed", "value": "SPOT_
WITH_FALLBACK" }
```

On Azure:

JAVASCRIPT

```
"azure_attributes.first_on_demand": { "type": "fixed", "value": 1},
"azure_attributes.availability": { "type": "fixed", "value":
"SPOT_WITH_FALLBACK_AZURE" }
```

On GCP (note: GCP cannot currently support the `first_on_demand` attribute):

JAVASCRIPT

```
"gcp_attributes.availability": { "type" :"fixed", "value":
"PREEMPTIBLE_WITH_FALLBACK_GCP" }
```

ENFORCE TAGGING

As seen earlier, tagging is crucial to an organization's ability to allocate cost and report it at granular levels. There are two things to consider when enforcing consistent tags in Databricks:

1. Compute policy controlling the `custom_tags`. attribute
2. For serverless, use Serverless Budget Policies as we discussed in Step 1

In the compute policy, we can control multiple custom tags by suffixing them with the tag name. It is recommended to use as many fixed tags as possible to reduce manual input from users, but allowlist is excellent for allowing multiple choices yet keeping values consistent.

JAVASCRIPT

```
{"custom_tags.cost_center": {"type":"allowlist", "values":["9999",
"9921", "9531" ]}},
{"custom_tags.environment": {"type":"fixed", "value":"dev"}}
```

QUERY TIMEOUT FOR WAREHOUSES

Long-running SQL queries can be very expensive and even disrupt other queries if too many begin to queue up. Long-running SQL queries are usually due to unoptimized queries (poor filters or even no filters) or unoptimized tables.

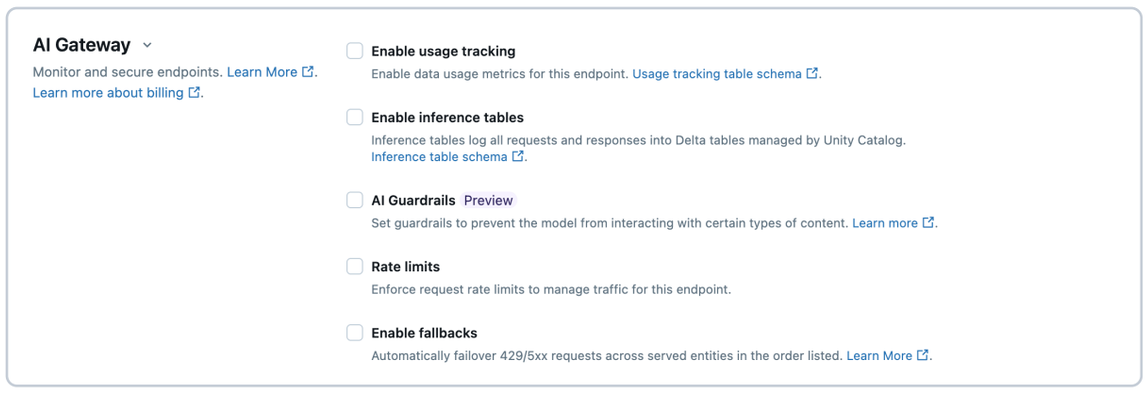
Admins can control for this by configuring the Statement Timeout at the workspace level. To set a workspace-level timeout, go to the workspace admin settings, click Compute, then click Manage next to SQL warehouses. In the SQL Configuration Parameters setting, add a configuration parameter where the timeout value is in seconds.

SQL

```
-- Set a workspace-level timeout of 3600 seconds (1 hour)
STATEMENT_TIMEOUT 3600
```

MODEL RATE LIMITS

ML models and LLMs can also be abused with too many requests, incurring unexpected costs. Databricks provides usage tracking and rate limits with an easy-to-use AI Gateway on model serving endpoints.



You can set rate limits on the endpoint as a whole or per user. This can be configured with the Databricks UI, SDK, API or Terraform. For example, we can deploy a Foundation Model endpoint with a rate limit using Terraform:

BASH

```
resource "databricks_model_serving" "llama_3_2_3b" {
 name = "llama_3_2_3b_instruct"
 ai_gateway {
   usage_tracking_config {
     enabled = true
   }
   # Rate limit the endpoint to 10 QPM
   rate_limits {
     calls = 10
     key   = "endpoint"
     renewal_period = "minute"
   }
 }


 config {
   served_entities {
     name                          = "meta_llama_v3_2_3b_instruct-3"
     entity_name                   = "system.ai.llama_v3_2_3b_instruct"
     entity_version                = "2"
     scale_to_zero_enabled         = true
     max_provisioned_throughput = 44000
   }
 }
}
```

PRACTICAL COMPUTE POLICY EXAMPLES

For more examples of real-world compute policies, see our Solution Accelerator here: https://github.com/databricks-industry-solutions/cluster-policy

Step 4: Cost optimization

Lastly, we will look at some optimizations you can check for in your workspace, clusters and storage layers. Most of these can be checked and implemented automatically, which we'll explore. Several optimizations take place at the compute level. These include actions such as right-sizing the VM instance type, knowing when to use Photon or not, the appropriate selection of compute type and more.

CHOOSING OPTIMAL RESOURCES

- Use job compute instead of all-purpose (we'll cover this more in depth next)
- Use SQL warehouses for SQL-only workloads for the best cost-efficiency
- Use up-to-date runtimes to receive the latest patches and performance improvements. For example, DBR 17.0 takes the leap to Spark 4.0 (blog) which includes many performance optimizations.
- Use Serverless for quicker startup, termination, and better total cost of ownership (TCO)
- Use autoscaling workers, unless using continuous streaming or the AvailableNow trigger
  - There are also advances in Spark Declarative Pipelines where autoscaling works well for streaming workloads thanks to a feature called Enhanced Autoscaling (AWS | Azure | GCP)
- Choose the correct VM instance type:
  - Newer generation instance types and modern processor architectures usually perform better and often at a lower cost. For example, on AWS, Databricks prefers Graviton-enabled VMs (e.g. c7g.xlarge instead of c7i.xlarge). These may yield up to 3x better price-to-performance (blog).
  - Memory-optimized for most ML workloads, e.g., r7g.2xlarge
  - Compute-optimized for streaming workloads, e.g., c6i.4xlarge
  - Storage-optimized for workloads that benefit from disk caching (ad hoc and interactive data analysis), e.g., i4g.xlarge and c7gd.2xlarge
  - Only use GPU instances for workloads that use GPU-accelerated libraries. Furthermore, unless performing distributed training, clusters should be single node.
  - General purpose otherwise, e.g., m7g.xlarge
  - Use Spot or Spot Fleet instances in lower environments like Dev and Stage

AVOID RUNNING JOBS ON ALL-PURPOSE COMPUTE

As mentioned in Cost Controls, cluster costs can be optimized by running automated jobs with Job Compute, not All-Purpose Compute. Exact pricing may depend on promotions and active discounts, but Job Compute is typically 2-3x cheaper than All-Purpose.

Job Compute also provides new compute instances each time, isolating workloads from one another, while still permitting multitask workflows to reuse the compute resources for all tasks if desired. See how to configure compute for jobs (AWS | Azure | GCP).

Using Databricks system tables, the following query can be used to find jobs running on interactive All-Purpose clusters. This is also included as part of the Jobs System Tables AI/BI Dashboard you can easily import to your workspace.

SQL

```
with clusters AS (
 SELECT
   *,
   ROW_NUMBER() OVER(PARTITION BY workspace_id, cluster_id ORDER BY
change_time DESC) as rn
 FROM system.compute.clusters
 WHERE cluster_source="UI"
 QUALIFY rn=1
),
job_tasks_exploded AS (
 SELECT
   workspace_id,
   job_id,
   EXPLODE(compute_ids) as cluster_id
 FROM system.lakeflow.job_task_run_timeline
 WHERE period_start_time >= CURRENT_DATE() - INTERVAL 30 DAY
),
all_purpose_cluster_jobs AS (
   SELECT
   t1.*,
   t2.cluster_name,
   t2.owned_by,
   t2.dbr_version
 FROM job_tasks_exploded t1
   INNER JOIN clusters t2 USING (workspace_id, cluster_id)
),
most_recent_jobs as (
 SELECT
   *,
   ROW_NUMBER() OVER(PARTITION BY workspace_id, job_id ORDER BY
change_time DESC) as rn
 FROM
   system.lakeflow.jobs QUALIFY rn=1
)
SELECT
 t1.workspace_id,
 t1.job_id,
 first(t2.name, TRUE) as name,
 array_size(collect_list(t1.cluster_id)) as task_runs,
 collect_set(
   named_struct(
     'cluster_id', t1.cluster_id,
     'cluster_name', t1.cluster_name
   )
 ) as clusters_list
FROM all_purpose_cluster_jobs t1
 LEFT JOIN most_recent_jobs t2 USING (workspace_id, job_id)
GROUP BY t1.workspace_id, t1.job_id
ORDER BY task_runs DESC
LIMIT 100;
```

MONITOR PHOTON FOR ALL-PURPOSE CLUSTERS AND CONTINUOUS JOBS

Photon is an optimized vectorized engine for Spark on the Databricks Data Intelligence Platform that provides extremely fast query performance. Photon increases the amount of DBUs the cluster costs by a multiple of 2.9x for job clusters, and approximately 2x for All-Purpose clusters. Despite the DBU multiplier, Photon can yield a lower overall TCO for jobs by reducing the runtime duration.

Interactive clusters, on the other hand, may have significant amounts of idle time when users are not running commands. Please ensure all-purpose clusters have the auto-termination setting applied to minimize this idle compute cost. While not always the case, this may result in higher costs with Photon. This also makes Serverless notebooks a great fit, as they minimize idle spend, run with Photon for the best performance and can spin up the session in just a few seconds.

Similarly, Photon isn't always beneficial for continuous streaming jobs that are up 24/7. Monitor whether you can reduce the number of worker nodes required when using Photon, as this lowers TCO; otherwise, Photon may not be a good fit for continuous jobs.

Note: The following query can be used to find interactive clusters that are configured with Photon:

SQL

```sql
-- Find interactive clusters using Photon
SELECT
  "Photon Usage" cost_saving_opportunity,
  u.usage_metadata.cluster_id,
  c.cluster_name,
  u.sku_name,
  c.dbr_version,
  u.billing_origin_product,
  c.owned_by,
  usage_date,
  ROUND(SUM(usage_quantity), 2) as sum_usage_quantity,
  usage_unit
FROM
  system.billing.usage u
    LEFT JOIN system.compute.clusters c
      ON c.cluster_id = u.usage_metadata.cluster_id
WHERE
  c.delete_time IS NULL
  AND c.cluster_id IS NOT NULL
  AND (
    u.product_features.is_photon = true
    AND u.product_features.sql_tier IS NULL
    AND u.billing_origin_product = 'ALL_PURPOSE'
  )
GROUP BY
  usage_metadata.cluster_id,
  u.product_features,
  u.sku_name,
  c.cluster_name,
  c.dbr_version,
  u.billing_origin_product,
  c.owned_by,
  usage_date,
  usage_unit
ORDER BY
  sum_usage_quantity DESC,
  usage_date
```

OPTIMIZING DATA STORAGE AND PIPELINES

There are too many strategies for optimizing data, storage and Spark to cover here. Fortunately, Databricks has compiled these into the Comprehensive Guide to Optimize Databricks, Spark and Delta Lake Workloads, covering everything from data layout and skew to optimizing delta merges and more.

Real-world application

ORGANIZATION BEST PRACTICES

Organizational structure and ownership best practices are just as important as the technical solutions we will go through next.

Digital natives running highly effective FinOps practices that include the Databricks Platform usually prioritize the following within the organization:

- Clear ownership for platform administration and monitoring
- Consideration of solution costs before, during and after projects
- Culture of continuous improvement — always optimizing

These are some of the most successful organizational structures for FinOps:

- Centralized (e.g., center of excellence, hub-and-spoke)
  - This may take the form of a central platform or data team responsible for FinOps and distributing policies, controls, and tools to other teams from there
- Hybrid/distributed budget centers
  - Disperses the centralized model out to different domain-specific teams. May have one or more admins delegated to that domain/team to align larger platform and FinOps practices with localized processes and priorities.

CENTER OF EXCELLENCE EXAMPLE

A center of excellence has many benefits, such as centralizing core platform administration and empowering business units with safe, reusable assets such as policies and bundle templates.

The center of excellence often puts teams such as Data Platform, Platform Engineer or Data Ops at the center, or “hub,” in a hub-and-spoke model. This team is responsible for allocating and reporting costs with the Usage Dashboard. To deliver an optimal and cost-aware self-service environment for teams, the platform team should create compute policies and budget policies that can be tailored to use cases and business units (the ”spokes”). While not required, we recommend managing these artifacts with Terraform and VCS for strong consistency, versioning and the ability to modularize.

Key takeaways

This has been a fairly exhaustive guide to help you take control of your costs with Databricks and we have covered several things along the way. To recap, the crawl, walk, run journey is this:

- Cost attribution
- Cost reporting
- Cost controls
- Cost optimization

Finally, to recap some of the most important takeaways:

- Solid tagging is the foundation of all good cost attribution and reporting. Use Compute Policies to enforce high-quality tags.
- Import the Usage Dashboard for your main stop when it comes to reporting and forecasting Databricks spending
- Import the Jobs System Tables AI/BI Dashboard to monitor and find jobs with cost-saving opportunities
- Use Compute Policies to enforce cost controls and resource limits on cluster creations

Next steps

Get started today and create your first Compute Policy, or use one of our policy examples. Then, import the Usage Dashboard as your main stop for reporting and forecasting Databricks spending. Check off optimizations from Step 3 we shared earlier for your clusters, workspaces, and data.

Databricks Delivery Solutions Architects (DSAs) accelerate data and AI initiatives across organizations. They provide architectural leadership, optimize platforms for cost and performance, enhance developer experience and drive successful project execution. DSAs bridge the gap between initial deployment and production-grade solutions — working closely with various teams, including data engineering, technical leads, executives and other stakeholders to ensure tailored solutions and faster time to value. To benefit from a custom execution plan, strategic guidance and support throughout your data and AI journey from a DSA, please contact your Databricks Account Team.

Lakeflow Connect: Efficient and Easy Data Ingestion Using the Microsoft SQL Server Connector

Explore Databricks Lakeflow Connect's fully managed SQL Server Connector to simplify ingesting and integrating data seamlessly with Databricks tools for data processing and analytics.

By Andrea Tardif, Prasanna Selvaraj, Hector Bustamante and Phanitha Kommareddi

Complexities of extracting SQL server data

While digital native companies recognize AI's critical role in driving innovation, many still face challenges in making their data readily available for downstream uses, such as machine learning development and advanced analytics. For these organizations, supporting business teams that rely on the SQL Server means having data engineering resources and maintaining custom connectors, preparing data for analytics and ensuring it is available to data teams for model development. Often, this data needs to be enriched with additional sources and transformed before it can inform data-driven decisions.

Maintaining these processes quickly becomes complex and brittle, slowing down innovation. That's why Databricks developed Lakeflow Connect, which includes built-in data connectors for popular databases, enterprise applications and file sources. These connectors provide efficient end-to-end, incremental ingestion, are flexible and easy to set up, and are fully integrated with the Databricks Data Intelligence Platform for unified governance, observability and orchestration. The new SQL Server connector from Lakeflow Connect is the first database connector with robust integration for both on-premises and cloud databases to help derive data insights within Databricks.

In this chapter, we'll review the key considerations for when to use Lakeflow Connect for SQL Server and explain how to configure the connector to replicate data from an Azure SQL Server instance. Then, we'll review a specific use case, best practices and how to get started.

Key architectural considerations

Below are the key considerations to help decide when to use the SQL Server connector.

Region compatibility	AWS | Azure | GCP
Serverless compute	✓
Change data capture and change tracking integration	✓
Unity Catalog compatibility	✓
Private networking security requirements	✓

REGION AND FEATURE COMPATIBILITY

Lakeflow Connect supports a wide range of SQL Server database variations, including:

- Microsoft Azure SQL Database
- Microsoft Azure SQL Managed Instance
- Amazon RDS for SQL Server
- SQL Server on GCP
- Microsoft SQL Server running on Azure VMs and Amazon EC2
- On-premises SQL Server accessed through Azure ExpressRoute or AWS Direct Connect

Since Lakeflow Connect runs on serverless pipelines under the hood, built-in features such as pipeline observability, event log alerting and lakehouse monitoring can be leveraged. If Serverless is not supported in your region, work with your Databricks Account Team to request help prioritizing development or deployment in that region.

Lakeflow Connect is built on the Data Intelligence Platform, which provides seamless integration with Unity Catalog to reuse established permissions and access controls across new SQL Server sources for unified governance. If your Databricks tables and views are on Hive, we recommend upgrading them to UC to benefit from these features (AWS | Azure | GCP).

CHANGE DATA REQUIREMENTS

Lakeflow Connect can be integrated with SQL Server with change tracking (CT) or change data capture (CDC) enabled to support efficient, incremental ingestion.

CDC provides historical change information about insert, update and delete operations, and when the actual data has changed. Change tracking identifies which rows were modified in a table without capturing the actual data changes themselves. Learn more about CDC and the benefits of using CDC with SQL Server.

Databricks recommends using change tracking for any table with a primary key to minimize the load on the source database. For source tables without a primary key, use CDC. Learn more about when to use it here.

The SQL Server connector captures an initial load of historical data on the first run of your ingestion pipeline. Then, the connector tracks and ingests only the changes made to the data since the last run, leveraging SQL Server's CT/CDC features to streamline operations and efficiency.

GOVERNANCE AND PRIVATE NETWORKING SECURITY

When a connection is established with the SQL Server using Lakeflow Connect:

- Traffic between the client interface and the control plane is encrypted in transit using TLS 1.2 or later
- The staging volume, where raw files are stored during ingestion, is encrypted by the underlying cloud storage provider
- Data at rest is protected following best practices and compliance standards
- When configured with private endpoints, all data traffic stays within the cloud provider's private network, avoiding the public internet

Once the data is ingested into Databricks, it is encrypted like other datasets within UC. The ingestion gateway that extracts snapshots, change logs and metadata from the source database lands in a UC Volume, a storage abstraction best for registering non-tabular datasets such as JSON files. This UC Volume resides within the customer's cloud storage account within their Virtual Networks or Virtual Private Clouds.

Additionally, UC enforces fine-grained access controls and maintains audit trails to govern access to this newly ingested data. UC Service credentials and Storage Credentials are stored as securable objects within UC, ensuring secure and centralized authentication management. These credentials are never exposed in logs or hardcoded into SQL ingestion pipelines, providing robust protection and access control.

If your organization meets the above criteria, consider Lakeflow Connect for SQL Server to help simplify data ingestion into Databricks.

Breakdown of technical solution

Next, review the steps for configuring Lakeflow Connect for the SQL Server and replicating data from an Azure SQL Server instance.

CONFIGURE UNITY CATALOG PERMISSIONS

Within Databricks, ensure serverless compute is enabled for notebooks, workflows and pipelines (AWS | Azure | GCP). Then, validate that the user or service principal creating the ingestion pipeline has the following UC permissions:

Permission type	Reason	Documentation
CREATE CONNECTION on the metastore	Lakeflow Connect needs to establish a secure connection to the SQL Server.	CREATE CONNECTION
USE CATALOG on the target catalog	Required as it provides access to the catalog where Lakeflow Connect will land the SQL Server data tables in UC.	USE CATALOG
USE SCHEMA, CREATE TABLE and CREATE VOLUME on an existing schema or CREATE SCHEMA on the target catalog	Provides the necessary rights to access schemas and create storage locations for ingested data tables.	GRANT PRIVILEGES
Unrestricted permissions to create clusters, or a custom compute policy	Required to spin up the compute resources required for the gateway ingestion process.	MANAGE COMPUTE POLICIES

SET UP AZURE SQL SERVER

To use the SQL Server connector, confirm that the following requirements are met:

- Confirm SQL version
  - SQL Server 2012 or a later version must be enabled to use change tracking. However, 2016+ is recommended. Review SQL version requirements here.
- Configure the database service account dedicated to the Databricks ingestion.
  - Validate privilege requirements based on cloud (AWS | Azure | GCP)
- Enable change tracking or built-in CDC
  - You must have SQL Server 2012 or a later version to use CDC. Versions earlier than SQL Server 2016 additionally require the Enterprise edition.

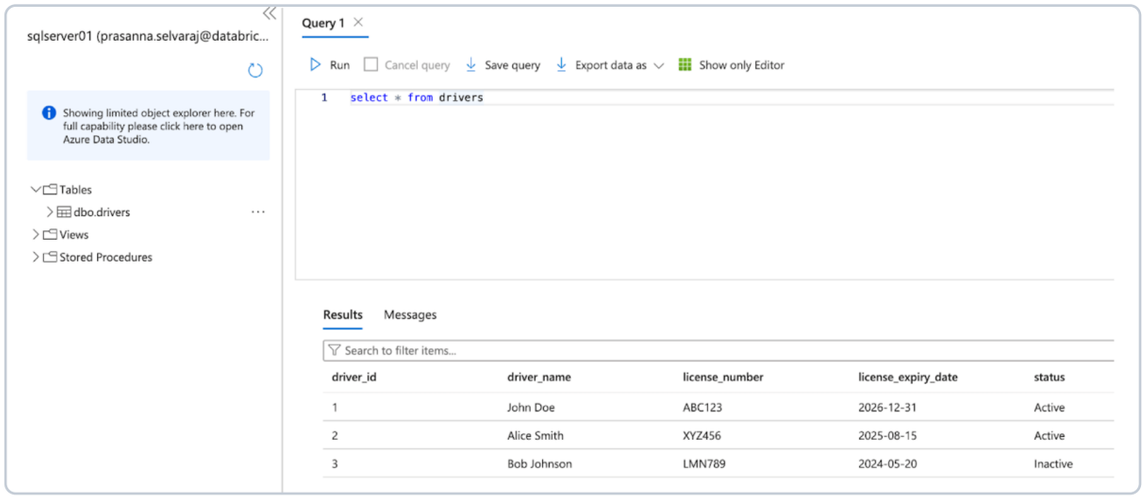
EXAMPLE: INGESTING FROM AZURE SQL SERVER TO DATABRICKS

Next, we will ingest a table from an Azure SQL Server database to Databricks using Lakeflow Connect. In this example, CDC and CT provide an overview of all available options. Since the table in this example has a primary key, CT could have been the primary choice. However, since there is only one small table in this example, there is no concern about load overhead, so CDC was also included. It is recommended to review when to use CDC, CT, or both to determine which is best for your data and refresh requirements.

1. [Azure SQL Server] Verify and configure Azure SQL Server for CDC and CT

Start by accessing the Azure portal and signing in with your Azure account credentials. On the left-hand side, click All services and search for SQL Servers. Find and click your server, then click the Query Editor. In this example, sqlserver01 was selected.

The screenshot below shows that the SQL Server database has one table called "'drivers."



Azure SQL Server UI — No CDC or CT enabled

Before replicating the data to Databricks, either CDC, CT or both must be enabled.

For this example, the following script is run on the database to enable CT:

SQL

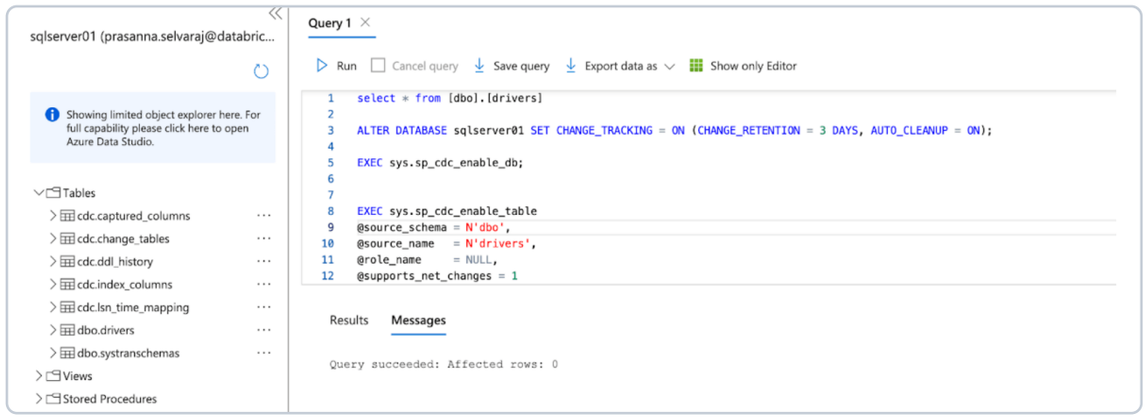
```
ALTER DATABASE sqlserver01 SET CHANGE_TRACKING = ON (CHANGE_RENTION
= 3 DAYS, AUTO_CLEANUP = ON);
```

This command enables change tracking for the database with the following parameters:

- CHANGE_RETENTION = 3 DAYS: This value tracks changes for three days (72 hours). A full refresh will be required if your gateway is offline longer than the set time. It is recommended that this value be increased if more extended outages are expected.
- AUTO_CLEANUP = ON: This is the default setting. To maintain performance, it automatically removes change tracking data older than the retention period.

Then, the following script is run on the database to enable CDC:

```
YAML
EXEC sys.sp_cdc_enable_table
@source_schema = N'MySchema',
@source_name   = N'MyTable',
@role_name     = NULL,
@supports_net_changes = 1
```



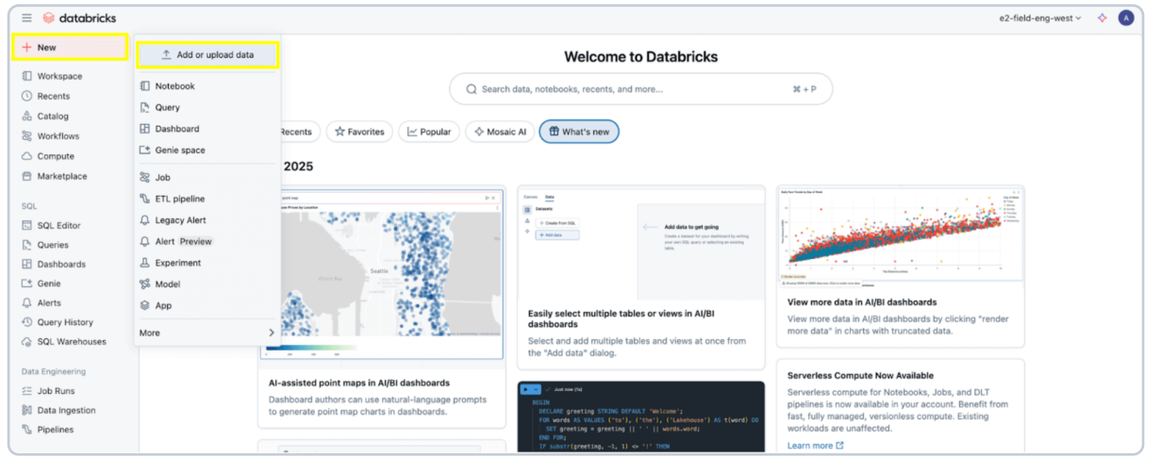
Azure SQL Server UI — CDC enabled

When both scripts finish running, review the tables section under the SQL Server instance in Azure and ensure that all CDC and CT tables are created.

2. [Databricks] Configure the SQL Server connector in Lakeflow Connect

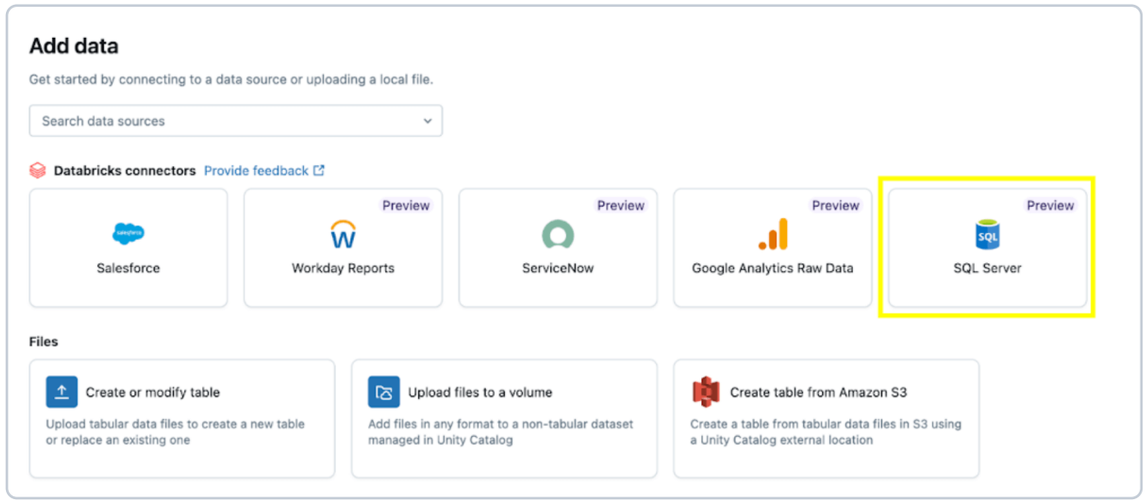
In this next step, the Databricks UI will be shown to configure the SQL Server connector. Alternatively, Databricks Asset Bundles, a programmatic way to manage the Lakeflow Connect pipelines as code, can also be leveraged. An example of the full DABs script is in the appendix below.

Once all the permissions are set, as laid out in the Permission Prerequisites section, you are ready to ingest data. Click the + New button at the top left, then select Add or Upload data.



Databricks UI — Add Data

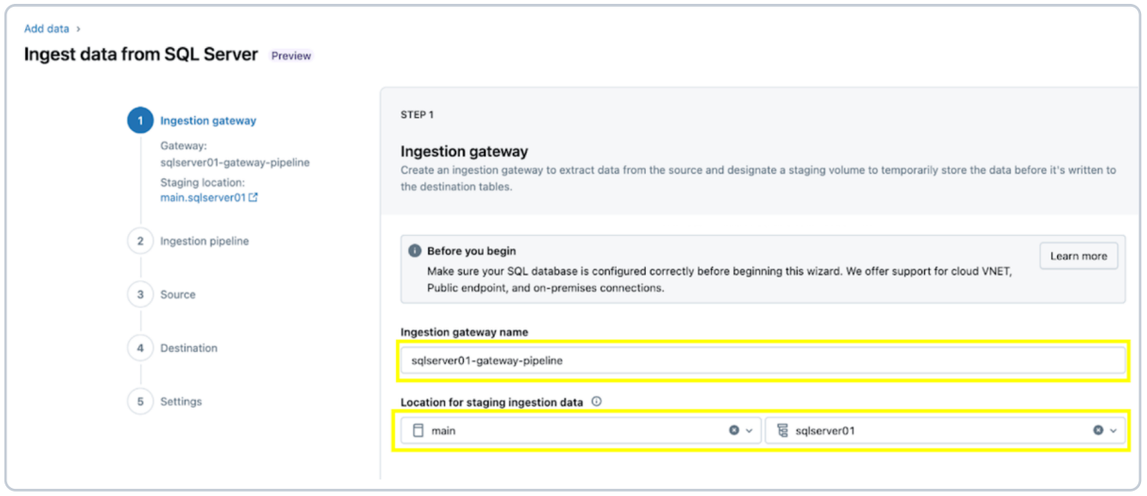
Then select the SQL Server option.



Databricks UI — SQL Server Connector

The SQL Server connector is configured in several steps.

1. Set up the ingestion gateway (AWS | Azure | GCP). In this step, provide a name for the ingestion gateway pipeline and a catalog and schema for the UC Volume location to extract snapshots and continually change data from the source database.



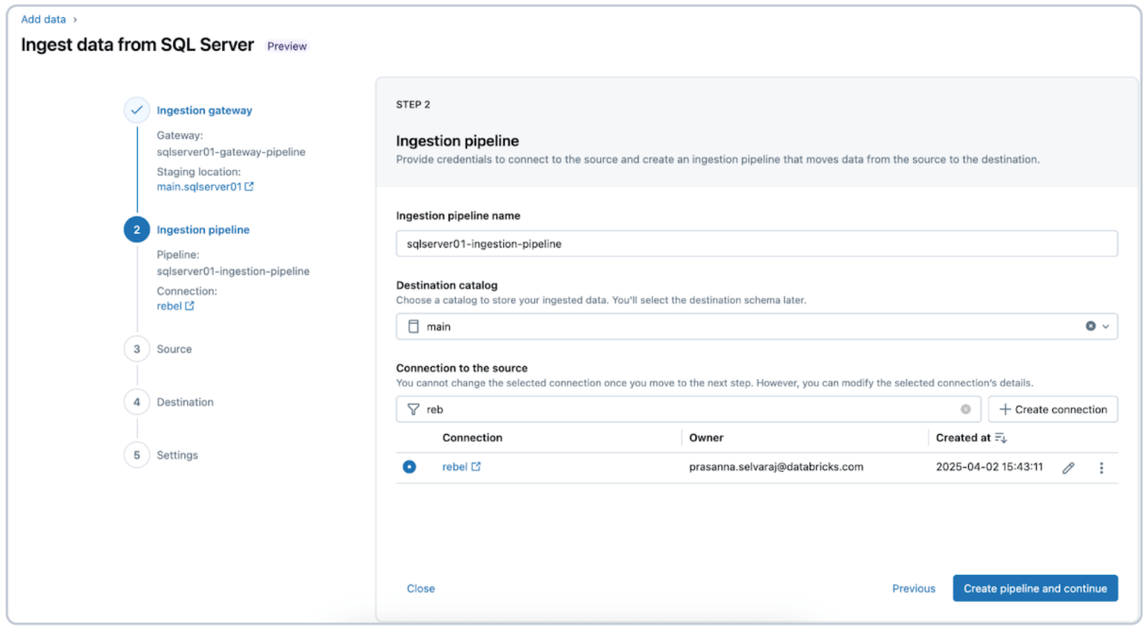
Databricks UI — SQL Server Connector: Ingestion Gateway

2. Configure the ingestion pipeline. This replicates the CDC/CT data source and the schema evolution events. A SQL Server connection is required, which is created through the UI following these steps or with the following SQL code below:

YAML

```
CREATE CONNECTION rebel
    TYPE SQLSERVER
    OPTIONS (
      host 'rebel.database.windows.net',
      port '1433',
      user '<username>',
      password '<password>');
```

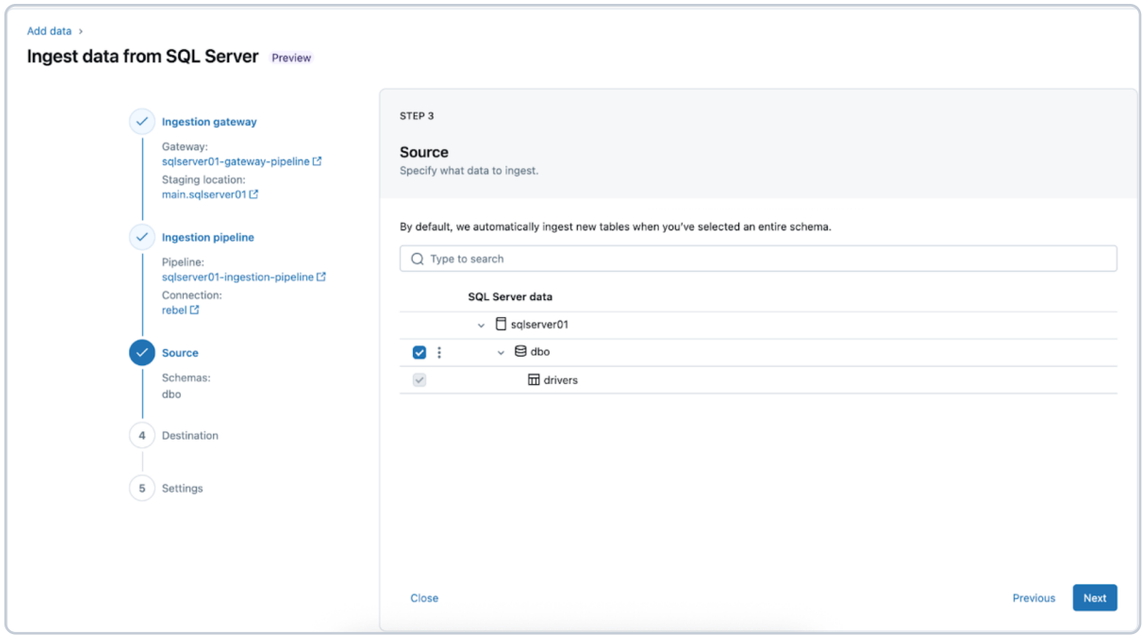
For this example, name the SQL server connection `rebel` as shown.



Databricks UI — SQL Server Connector: Ingestion Pipeline

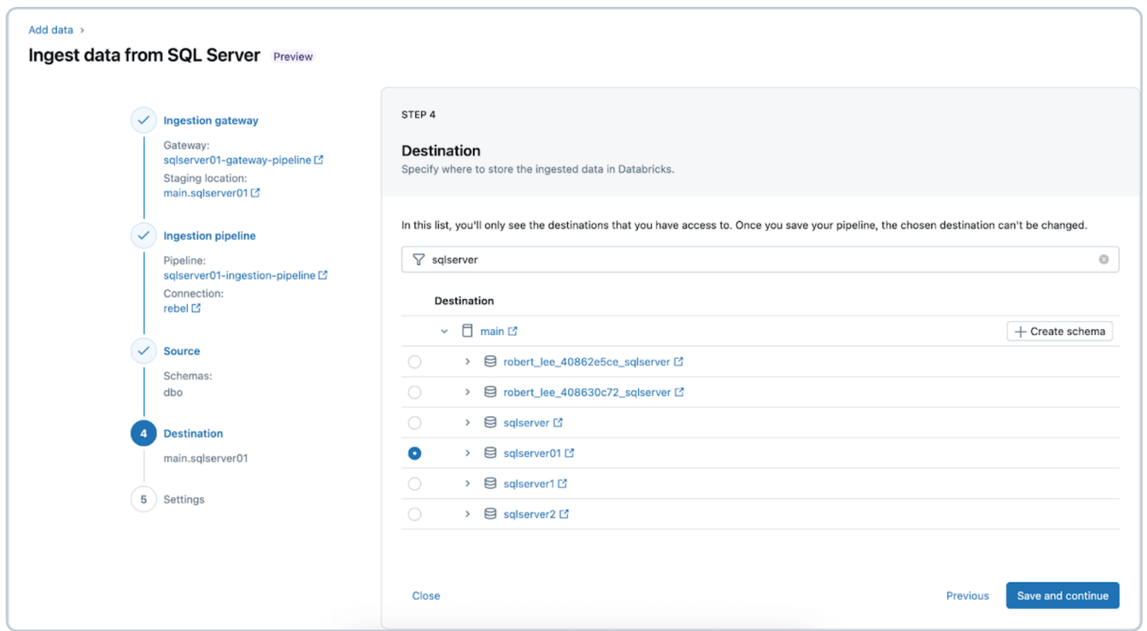
3. Selecting the SQL Server tables for replication. Select the whole schema to be ingested into Databricks instead of choosing individual tables to ingest.

The whole schema can be ingested into Databricks during initial exploration or migrations. If the schema is large or exceeds the allowed number of tables per pipeline (see connector limits), Databricks recommends splitting the ingestion across multiple pipelines to maintain optimal performance. For use case-specific workflows such as a single ML model, dashboard, or report, it's generally more efficient to ingest individual tables tailored to that specific need, rather than the whole schema.



Databricks UI — SQL Server Connector: Source

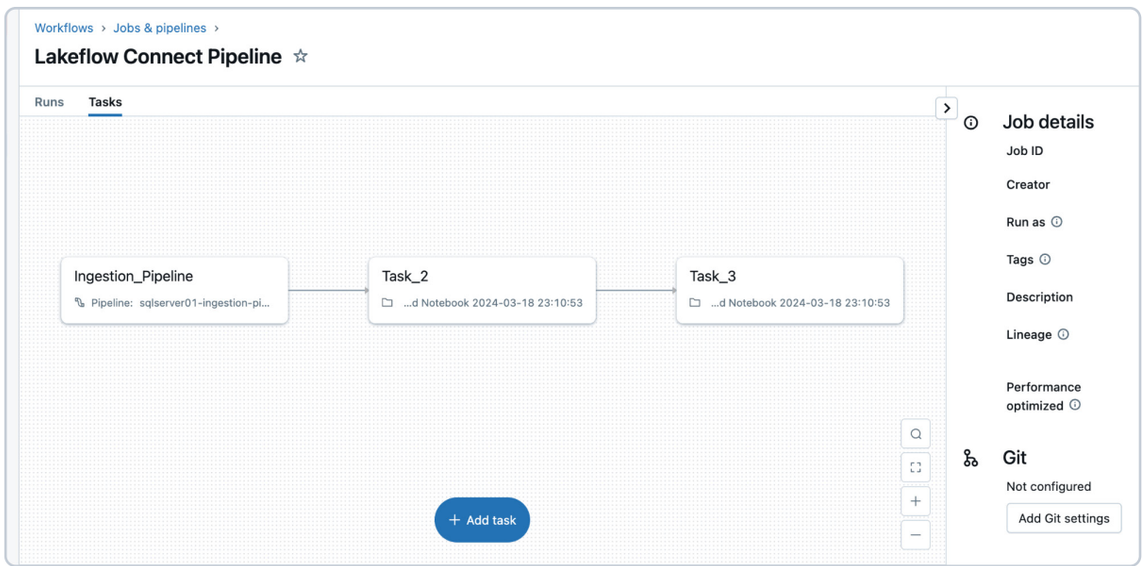
4. Configure the destination where the SQL Server tables will be replicated within UC. Select the `main` catalog and `sqlserver01` schema to land the data in UC.



Databricks UI — SQL Server Connector: Destination

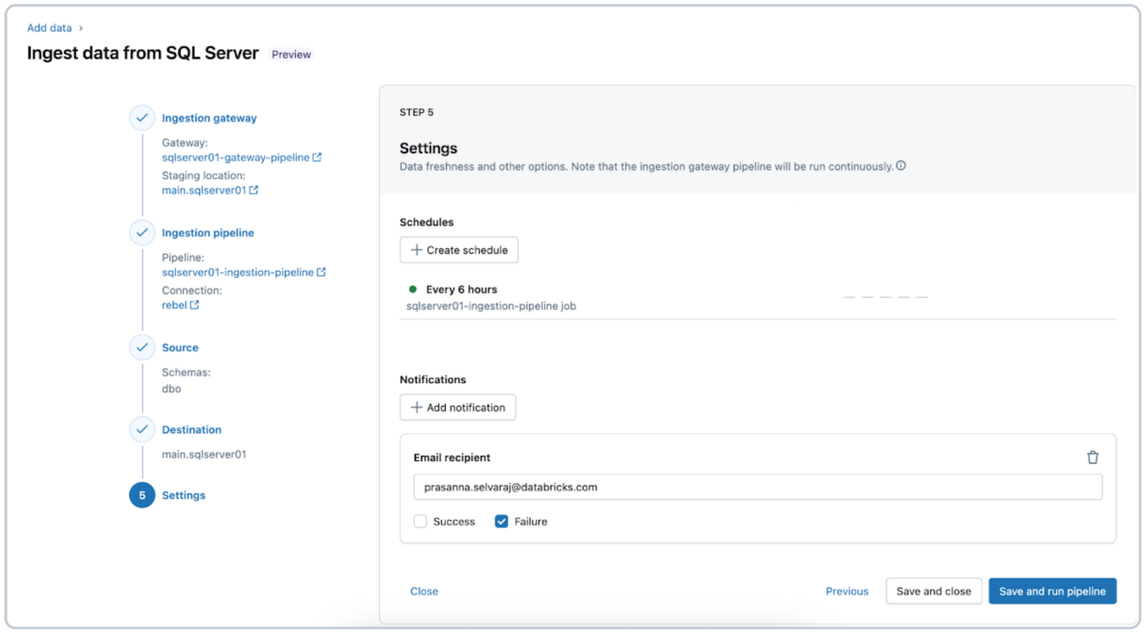
5. Configure schedules and notifications (AWS | Azure | GCP). This final step will help determine how often to run the pipeline and where success or failure messages should be sent. Set the pipeline to run every six hours and notify the user only of pipeline failures. This interval can be configured to meet the needs of your workload.

The ingestion pipeline can be triggered on a custom schedule. Lakeflow Connect will automatically create a dedicated job for each scheduled pipeline trigger. The ingestion pipeline is a task within the job. Optionally, more tasks can be added before or after the ingestion task for any downstream processing.



Databricks UI — Lakeflow Connect Pipeline

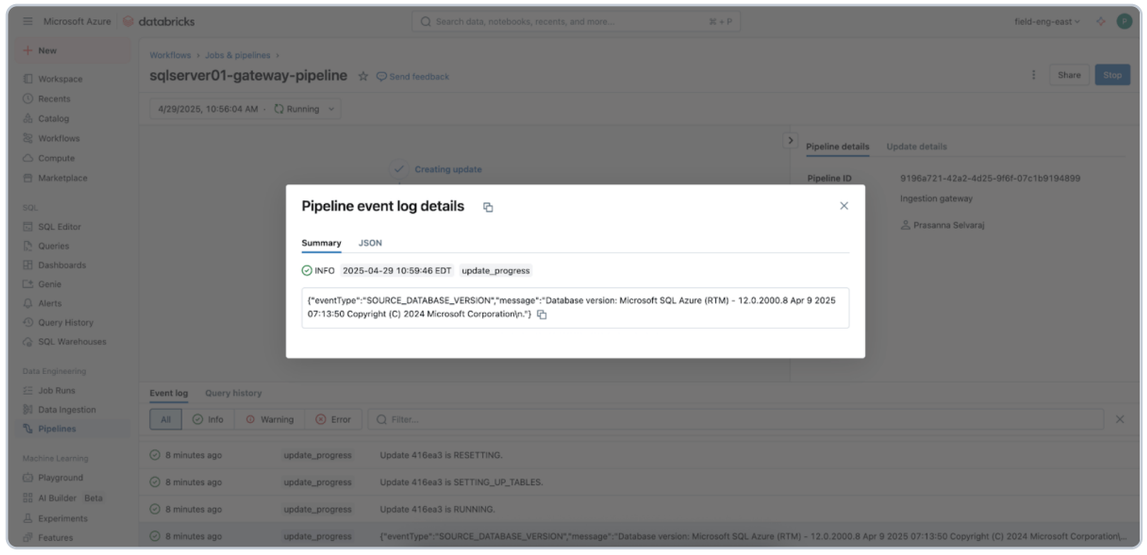
After this step, the ingestion pipeline is saved and triggered, starting a full data load from the SQL Server into Databricks.



Databricks UI — SQL Server Connector: Settings

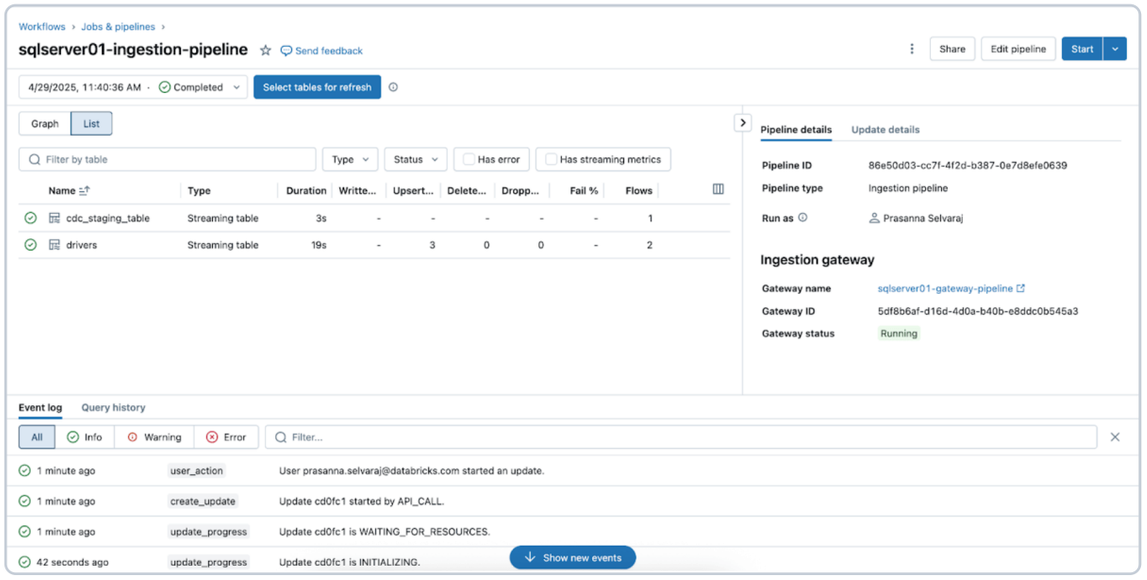
3. [Databricks] Validate successful runs of the gateway and ingestion pipelines

Navigate to the Pipeline menu to check if the gateway ingestion pipeline is running. Once complete, search for 'update_progress' within the pipeline event log interface at the bottom pane to ensure the gateway successfully ingests the source data.



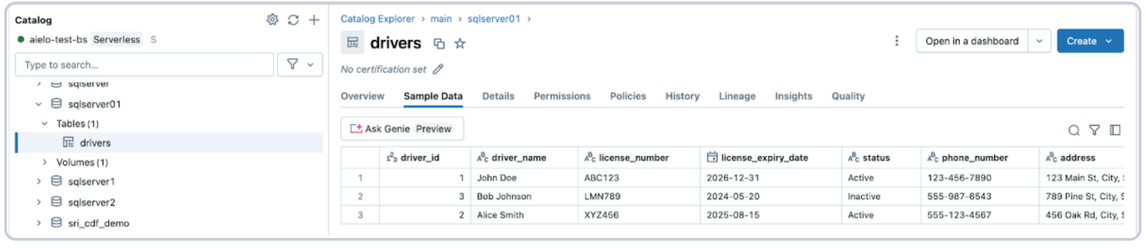
Databricks Pipeline UI — Pipeline Event Log: 'update_progress'

To check the sync status, navigate to the pipeline menu. The screenshot below shows that the ingestion pipeline has performed three insert and update (UPSERT) operations.



Databricks Pipeline UI — Validate Insert and Update Operations

Navigate to the target catalog, **main**, and schema, **sqlserver01**, to view the replicated table, as shown below.



Databricks UC — Replicated Target Table

4. [Databricks] Test CDC and Schema Evolution

Next, verify a CDC event by performing insert, update and delete operations in the source table. The screenshot of the Azure SQL Server below depicts the three events.

Azure SQL Server UI — Insert Rows

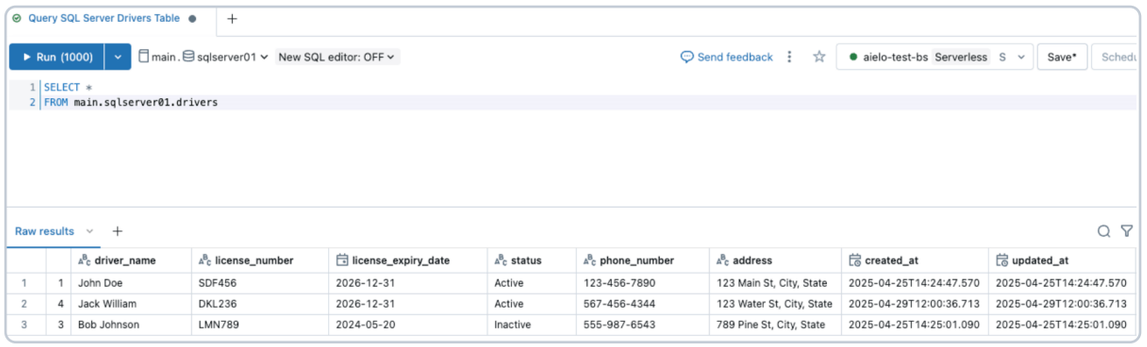
SQL

```
INSERT INTO dbo.drivers (driver_name, license_number, license_
expiry_date, status, phone_number, address)
VALUES ('Jack William', 'DKL236', '2026-12-31', 'Active', '567-456-
4344', '123 Water St, City, State');

update [dbo].[drivers]
set license_number = 'SDF456'
where driver_id = 1;

delete from [dbo].[drivers]
where driver_id = 2;

SELECT * FROM [dbo].[drivers];
```

Once the pipeline is triggered and is completed, query the delta table under the target schema and verify the changes.

SQL

```
SELECT *
FROM main.sqlserver01.drivers
```

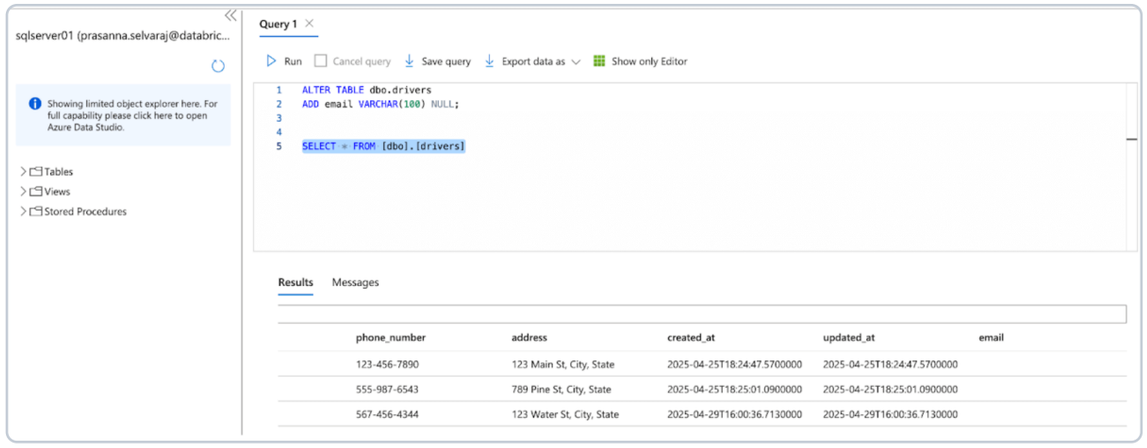


Databricks SQL UI — View Inserted Rows

Similarly, let's perform a schema evolution event and add a column to the SQL Server source table, as shown below:

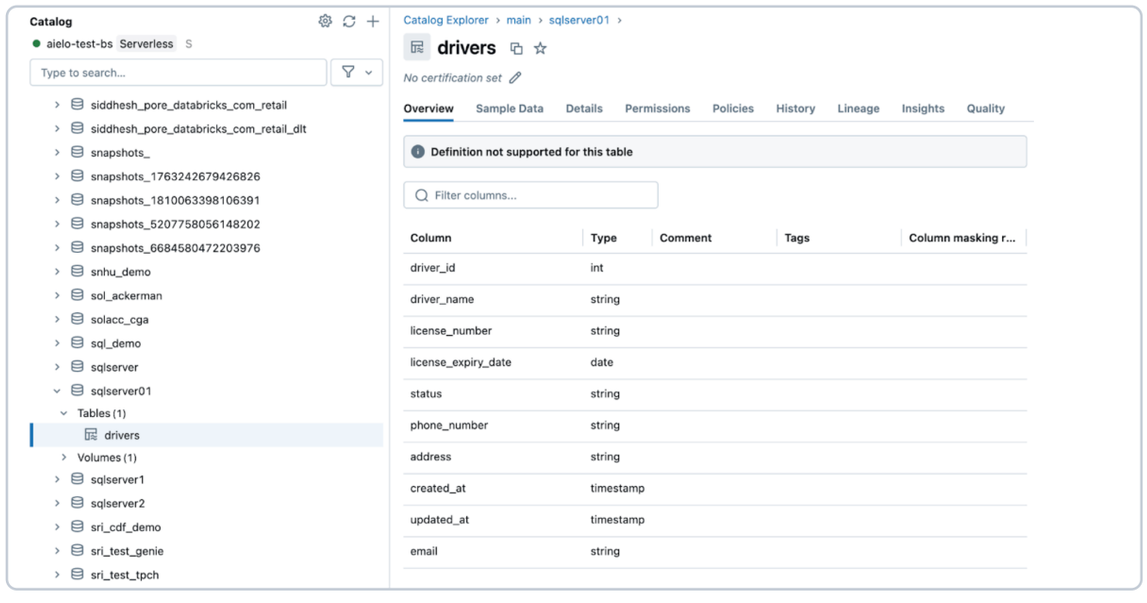
SQL

```
ALTER TABLE dbu.drivers
ADD email VARCHAR(100) NULL;

SELECT * FROM [dbo].[drivers]
```



Azure SQL Server UI — Schema Evolution

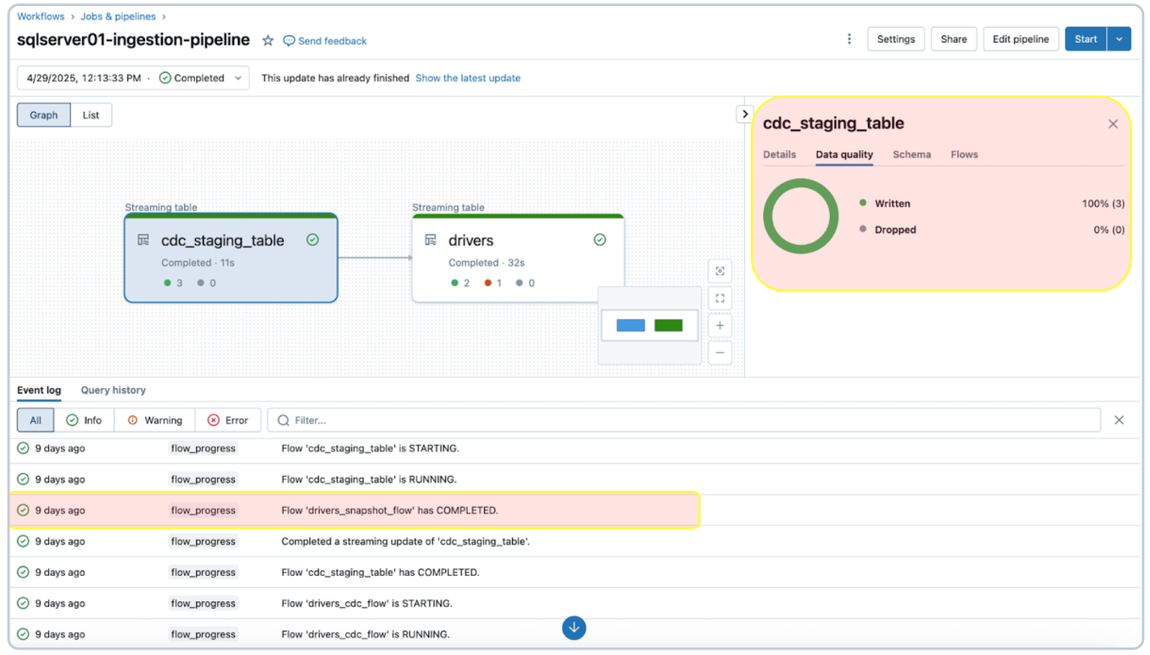
After changing the sources, trigger the ingestion pipeline by clicking the start button within the Spark Declarative Pipelines UI. Once the pipeline has been completed, verify the changes by browsing the target table, as shown below. The new column `email` will be appended to the end of the drivers table.



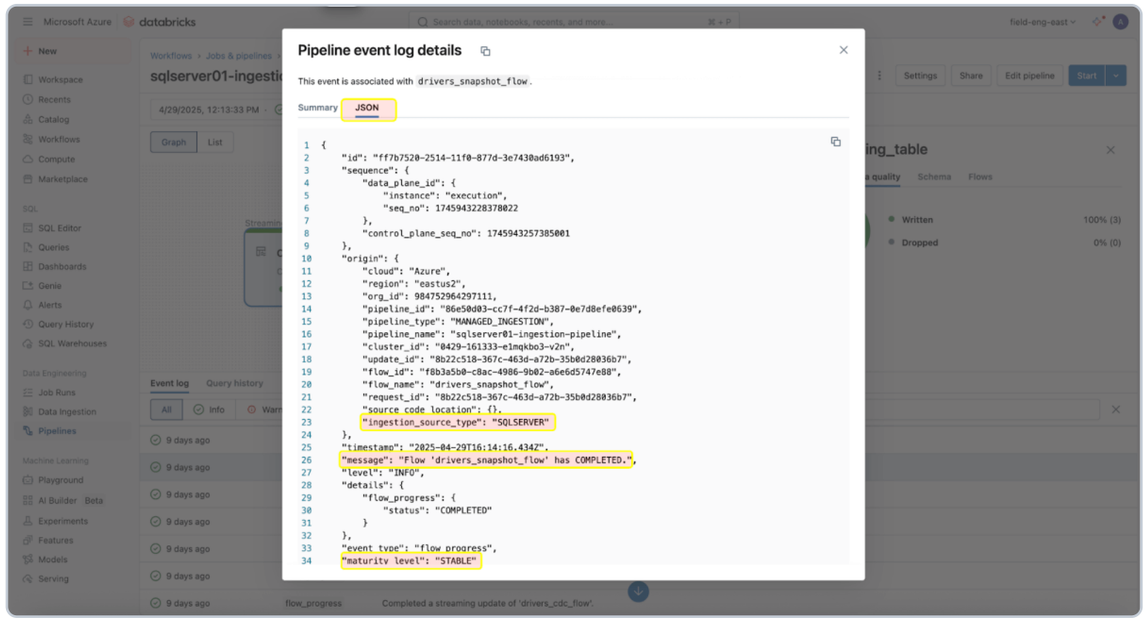
Databricks UC — View Schema Change

5. [Databricks] Continuous pipeline monitoring

Monitoring their health and behavior is crucial once the ingestion and gateway pipelines are successfully running. The pipeline UI provides data quality checks, pipeline progress and data lineage information. To view the event log entries in the pipeline UI, locate the bottom pane under the pipeline DAG, as shown below.



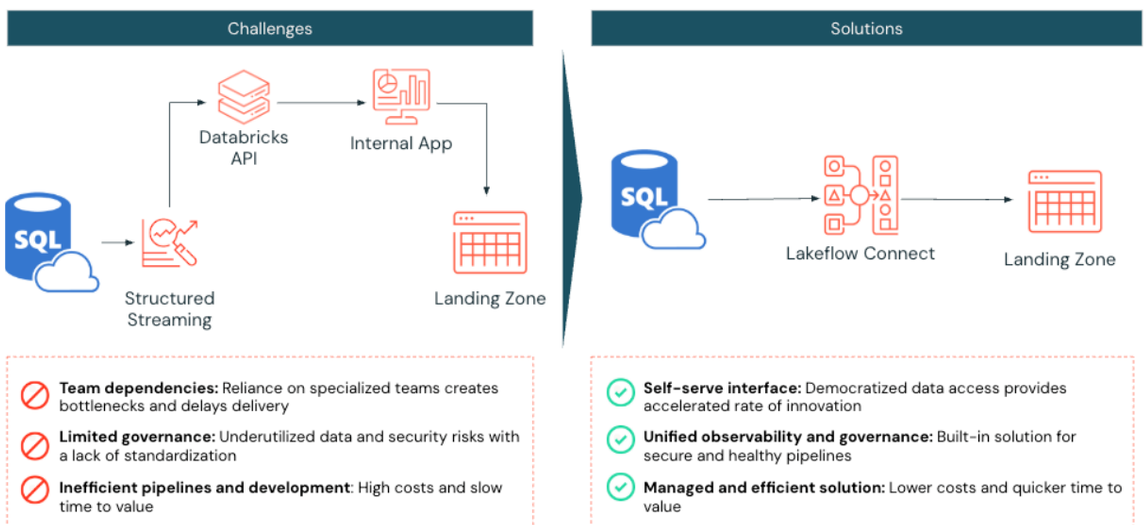
Databricks Pipeline Event Log UI



Databricks Pipeline Event Log Details — JSON

The event log entry above shows that the ‘drives_snapshot_flow’ was ingested from the SQL Server and completed. The maturity level of STABLE indicates that the schema is stable and has not changed. More information on the event log schema can be found here.

Real-world example



Challenges → Solutions

A large-scale medical diagnostic lab using Databricks faced challenges efficiently ingesting SQL Server data into its lakehouse. Before implementing Lakeflow Connect, the lab used Databricks Spark notebooks to pull two tables from Azure SQL Server into Databricks. Their application would then interact with the Databricks API to manage compute and job execution.

Recognizing that this process could be simplified, the medical diagnostic lab implemented Lakeflow Connect for SQL Server. Once enabled, the implementation was completed in just one day, allowing the medical diagnostic lab to leverage Databricks' built-in tools for observability with daily incremental ingestion refreshes.

Operational considerations

Once the SQL Server connector has successfully established a connection to your Azure SQL Database, the next step is to efficiently schedule your data pipelines to optimize performance and resource utilization. In addition, it's essential to follow best practices for programmatic pipeline configuration to ensure scalability and consistency across environments.

PIPELINE ORCHESTRATION

There is no limit on how often the ingestion pipeline can be scheduled to run. However, to minimize costs and ensure consistency in pipeline executions without overlap, Databricks recommends at least a five-minute interval between ingestion executions. This allows new data to be introduced at the source while accounting for computational resources and startup time.

The ingestion pipeline can be configured as a task within a job. If downstream workloads rely on fresh data, you can set task dependencies to ensure the ingestion pipeline run completes before executing those tasks.

Additionally, suppose the pipeline is still running when the next refresh is scheduled. In that case, the ingestion pipeline will behave similarly to a job and skip the update until the next scheduled one, assuming the currently running update completes on time.

OBSERVABILITY AND COST TRACKING

Lakeflow Connect operates on a compute-based pricing model, ensuring efficiency and scalability for various data integration needs. The ingestion pipeline operates on serverless compute, which allows for flexibility in scaling based on demand and simplifies management by eliminating the need for users to configure and manage the underlying infrastructure.

However, it's important to note that while the ingestion pipeline can run on serverless compute, the ingestion gateway for database connectors currently operates on classic compute to simplify connections to the database source. As a result, users might see a combination of classic and serverless Spark Declarative Pipelines DBU charges reflected in their billing.

The easiest way to track and monitor Lakeflow Connect usage is through system tables. Below is an example query to view a particular Lakeflow Connect pipeline's usage:

SQL

```
SELECT
  u.billing_origin_product,
  u.usage_metadata.dlt_pipeline_id as pipeline_id,
  u.ingestion_date,
  u.usage_date,
  SUM(u.usage_quantity) as DBU_COUNT,
  SUM(u.usage_quantity * l.pricing.effective_list.default) as DBU_
AMOUNT
FROM system.billing.usage u
LEFT JOIN system.billing.list_prices l
  ON u.sku_name = l.sku_name
WHERE
  u.billing_origin_product = 'LAKEFLOW_CONNECT'
  AND u.usage_metadata.dlt_pipeline_id = '<dlt-pipeline-id>'
  AND u.usage_date >= '<date>' --optional
GROUP BY
  u.billing_origin_product,
  u.usage_metadata.dlt_pipeline_id,
  u.ingestion_date,
  u.usage_date
ORDER BY u.usage_date DESC;
```

	billing_origin_product	pipeline_id	ingestion_date	usage_date	DBU_COUNT	DBU_AMOUNT
1	LAKEFLOW_CONNECT	d9d0df2f-83e5-455f-a654-264ccf90cced	2025-05-07	2025-05-07	2.54	$0.80
2	LAKEFLOW_CONNECT	d9d0df2f-83e5-455f-a654-264ccf90cced	2025-05-06	2025-05-06	2.63	$0.83
3	LAKEFLOW_CONNECT	d9d0df2f-83e5-455f-a654-264ccf90cced	2025-05-05	2025-05-05	2.69	$0.85
4	LAKEFLOW_CONNECT	d9d0df2f-83e5-455f-a654-264ccf90cced	2025-05-04	2025-05-04	2.84	$0.90
5	LAKEFLOW_CONNECT	d9d0df2f-83e5-455f-a654-264ccf90cced	2025-05-03	2025-05-03	3.81	$1.20
6	LAKEFLOW_CONNECT	d9d0df2f-83e5-455f-a654-264ccf90cced	2025-05-02	2025-05-02	3.88	$1.22
7	LAKEFLOW_CONNECT	d9d0df2f-83e5-455f-a654-264ccf90cced	2025-05-01	2025-05-01	3.82	$1.20

Databricks SQL — System Table Query Output

The official pricing for Lakeflow Connect documentation (AWS | Azure | GCP) provides detailed rate information. Additional costs, such as serverless egress fees (pricing), may apply. Egress costs from the Cloud provider for classic compute can be found here (AWS | Azure | GCP).

Best practices and key takeaways

Below are some of the best practices and considerations to follow when implementing this SQL Server connector:

1. Configure each Ingestion Gateway to authenticate with a user or entity with access only to the replicated source database
2. Ensure the user is given the necessary permissions to create connections in UC and ingest the data
3. Utilize DABs to reliably configure Lakeflow Connect ingestion pipelines, ensuring repeatability and consistency in infrastructure management
4. For source tables with primary keys, enable change tracking to achieve lower overhead and improved performance
5. For source tables without a primary key, enable CDC due to its ability to capture changes at the column level, even without unique row identifiers

Lakeflow Connect for SQL Server provides a fully managed, built-in integration for both on-premises and cloud databases for efficient, incremental ingestion into Databricks.

Next steps and additional resources

Try the SQL Server connector today to help solve your data ingestion challenges. Follow the steps outlined in this chapter or review the documentation. Learn more about Lakeflow Connect on the product page, view a product tour or view a demo of the Salesforce connector to help predict customer churn.

Appendix

In this optional step, to manage the Lakeflow Connect pipelines as code using DABs, you simply need to add two files to your existing bundle:

- A workflow file that controls the frequency of data ingestion (resources/sqlserver.yml)
- A pipeline definition file (resources/sqlserver_pipeline.yml)

resources/sqlserver.yml:

YAML

```
variables:
  # Common variables used multiple places in the DAB definition.
  gateway_name:
    default: sqlserver01-gateway-pipeline
  dest_catalog:
    default: main
  dest_schema:
    default: sqlserver01

resources:
  pipelines:
    gateway:
      name: ${var.gateway_name}
      gateway_definition:
        connection_name: rebel
        gateway_storage_catalog: main
        gateway_storage_schema: sqlserver01
        gateway_storage_name: sqlserver01-gateway-pipeline
      catalog: main
      target: sqlserver01
      channel: PREVIEW

    pipeline_sqlserver:
      name: sqlserver-ingestion-pipeline
      ingestion_definition:
        ingestion_gateway_id: ${resources.pipelines.gateway.id}
        objects:
          - schema:
              # Ingest all tables in the sqlserver01.dbo schema to
main.dest_schema. The destination table name will be drivers, the
same as it is on the source.
              source_catalog: sqlserver01
              source_schema: dbo
              destination_catalog: main
              destination_schema: sqlserver01
      target: sqlserver01
      catalog: main
      channel: PREVIEW
```

resources/sqlserver_job.yml:

YAML

```
resources:
  jobs:
    sqlserver_dab_job:
      name: sqlserver-ingestion-pipeline job

      trigger:
        periodic:
          interval: 6
          unit: HOURS

      email_notifications:
        on_failure:
          - <user-email>

      tasks:
        - task_key: refresh_pipeline
          pipeline_task:
            pipeline_id: ${resources.pipelines.pipeline_sqlserver.
id}
```

Implement CDC Data Load and SCD Type 2 With Lakeflow

By Cary Moore

Introduction

Every data warehousing project, once defined and deemed necessary, contains three primary tasks:

1. Ingest data from the source
2. Populate and transform data for reporting
3. Leverage Slowly Changing Dimensions Type II methodology (maintain history)

Before Databricks, developers would need a separate tool for each function and have to maintain them, adding a lot of complexity.

This chapter demonstrates the new Databricks Lakeflow Connect capabilities combined with Spark Declarative Pipelines to perform these three requirements for data warehousing — all in one combined package using Microsoft's SQL Server as the source database.

Setting up the source database for CDC

When working with large data sets, constantly processing sources from scratch tends to be an expensive and time consuming affair.

Many "poor man's" change data capture (CDC) implementations, such as "Last Update Date," other date filters or difference operations, are often performed due to a lack of control on the source database. Beyond the time and expense, there are also deficiencies in capturing deletes in the source data, requiring additional operations to determine those transactions.

When the stars align and developers are provided the capability to leverage change data capture in the source database, you would need a tool that can call the database procedure to retrieve the changed records then process them forward in the ETL.

Databricks now has Lakeflow Connect. With it, you can ingest the CDC records. You still, however, have to enable the CDC components on the source database. For that, you need to follow the provided steps:

- Create a user in the source database that has privileges to read and execute the necessary procedures and change logs. Those requirements are specified in this documentation for SQL Server.
- You will need to decide whether to use change data capture or change tracking. The documentation can assist in making the decision: *Databricks requires either Microsoft change tracking or Microsoft change data capture (CDC) to extract data from SQL Server.*
- Change tracking captures the fact that rows in a table have changed, but doesn't capture the actual operations. Enable change tracking for the tables.
- CDC captures every operation on a table. Enable the built-in CDC.

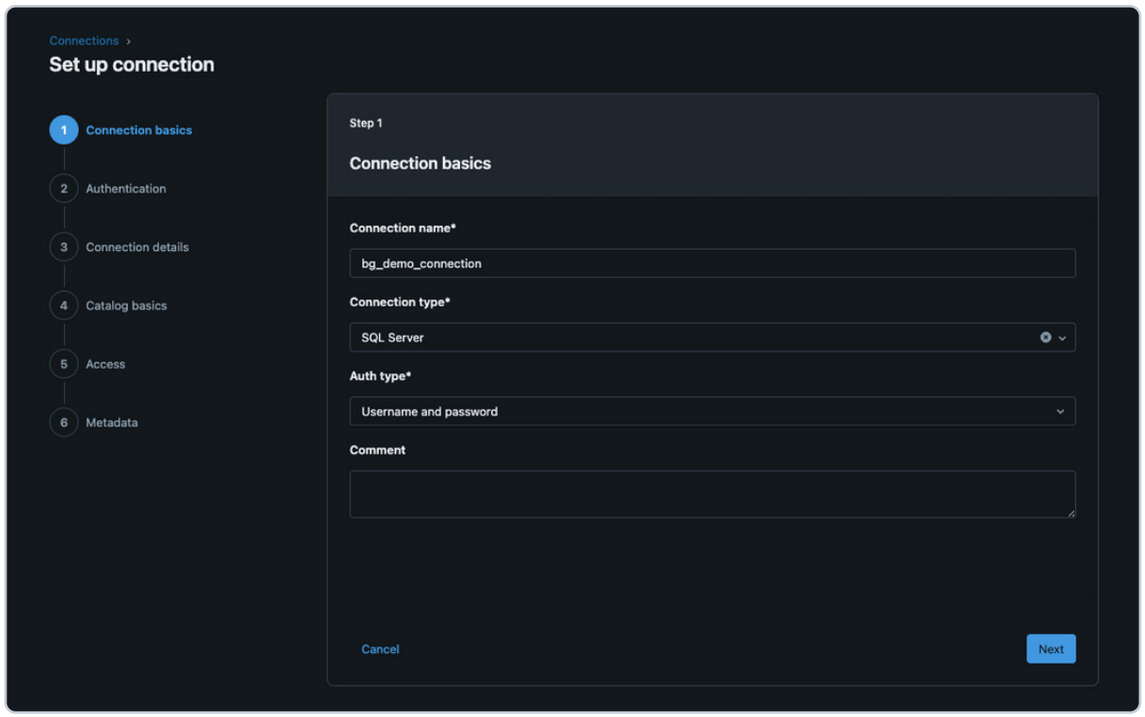
Creating a connection for the source database

With the database user you created for the CDC replication, you will need to create a **connection** in Databricks so the replication pipeline can connect to the database and perform the ingestion/replication. In this chapter, we focus on connecting to an Azure SQL Server instance, though we also support an on-premises SQL Server instance with the correct network configurations (Express Route, etc.).

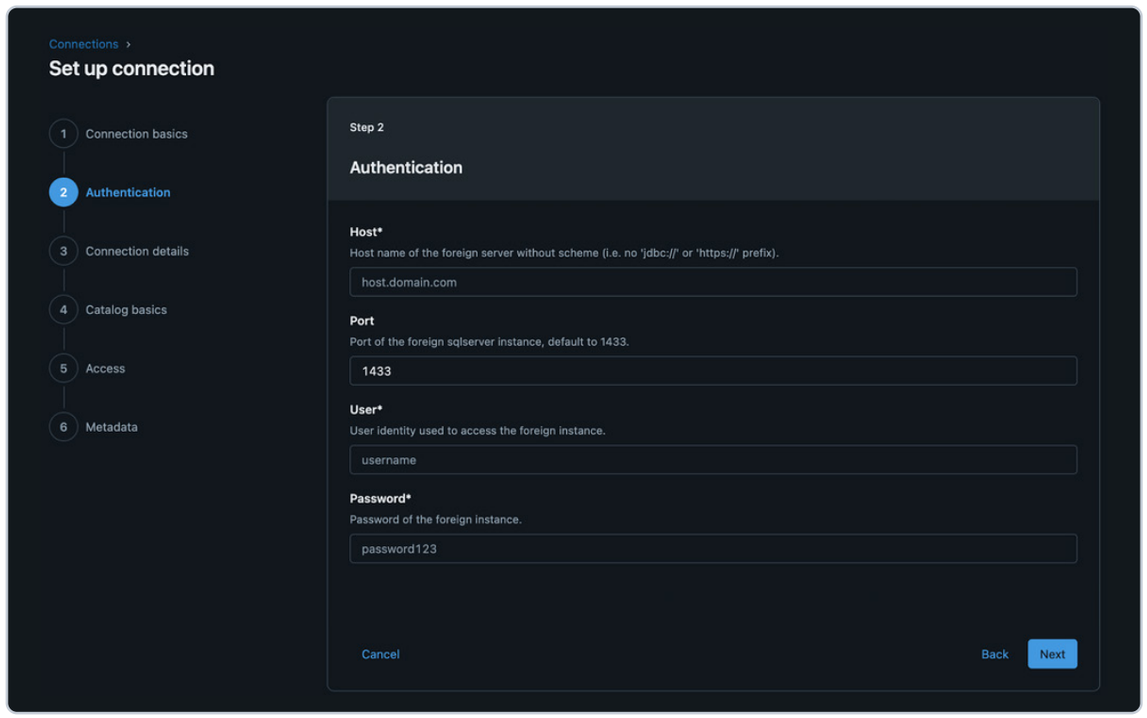
The steps are getting easier now that we are purely in the UI on Databricks. The following steps are in this guide, but are presented here in reprise.

Note that the UI combines the connection and the Unity Catalog's catalog creation steps.

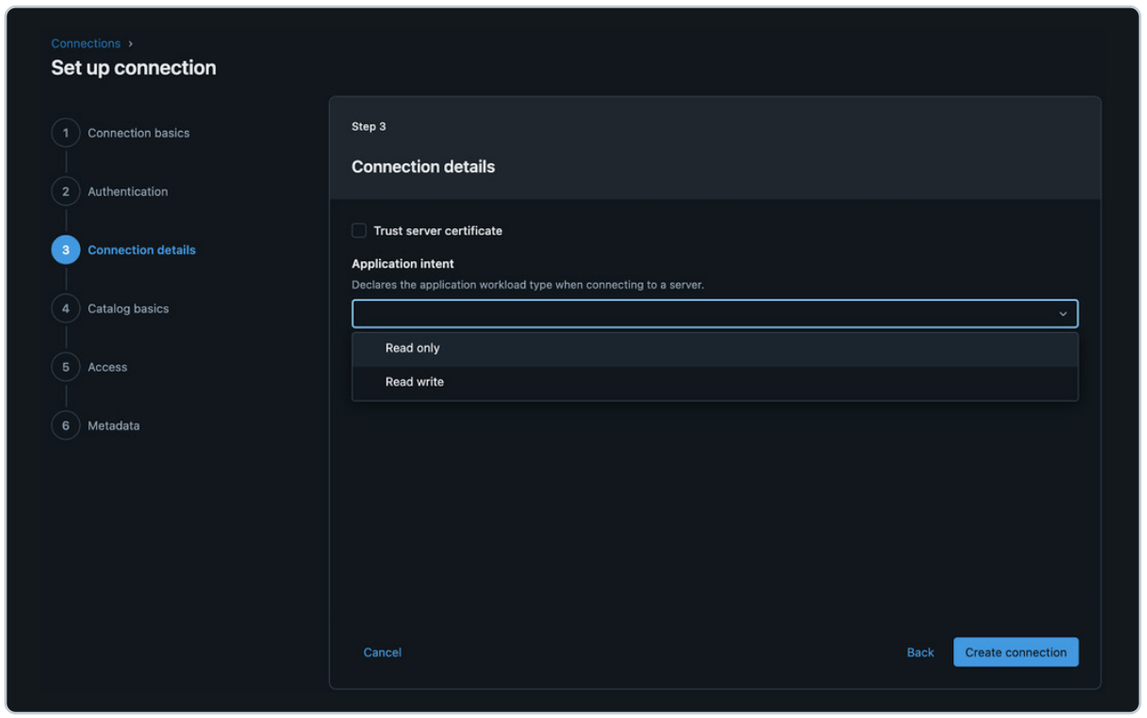
1. Using the Databricks workspace UI, create an external connection for the SQL Server database via the steps from Catalog > External Data > Connections location.
2. Provide a connection name and select the SQL Server connection type. This will open up additional steps. Select the "Username and password" Auth type (Databricks also supports OAuth). Optionally, add a comment, then click "Next."



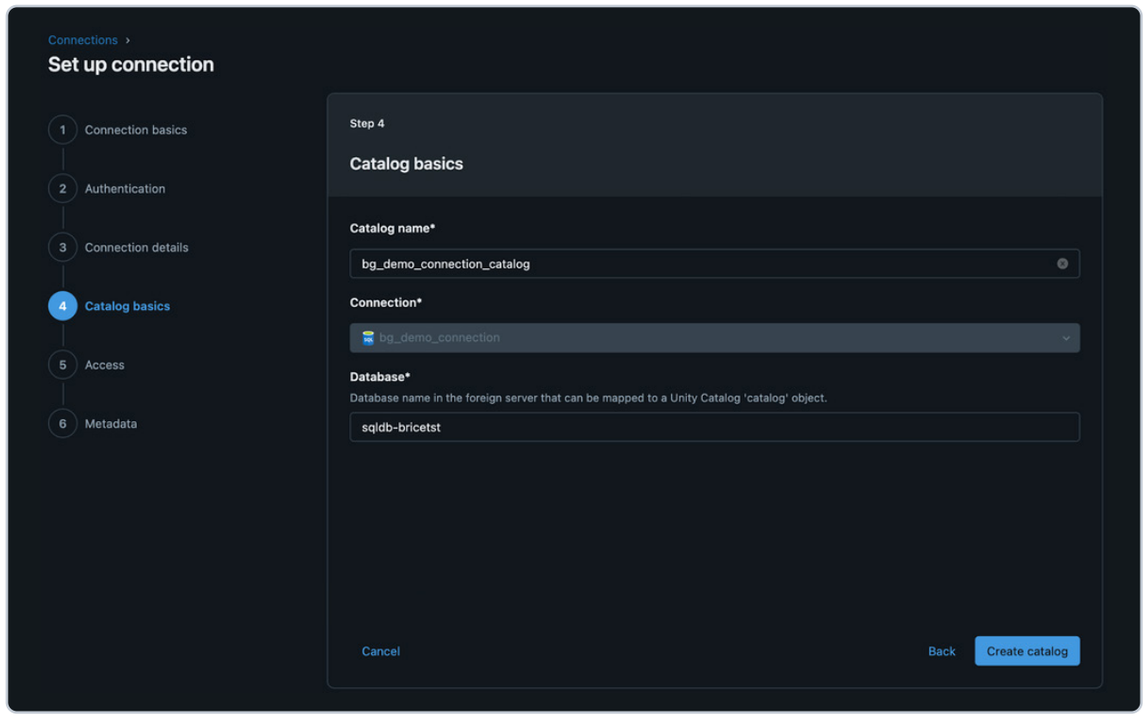
Provide your connection and authentication information and continue.



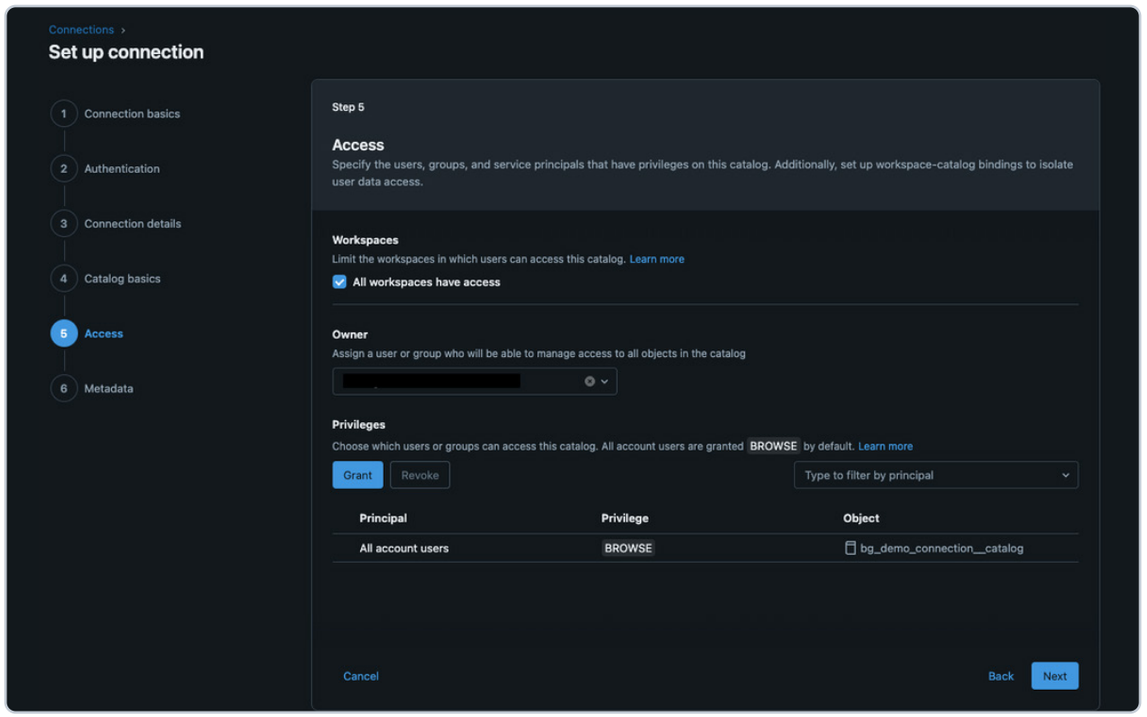
In the “Connection details” step, pick the appropriate certificate option for your scenario, select “Read only” for Application intent and create the connection.



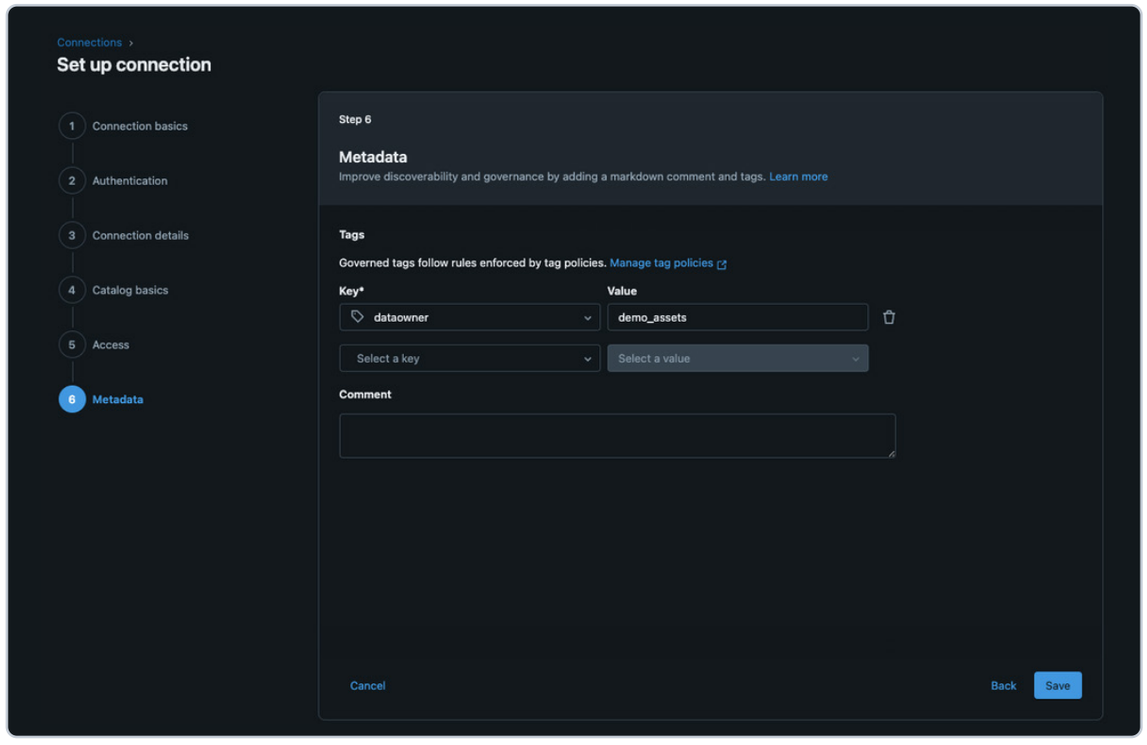
Provide a catalog name to be used in Unity Catalog, and provide the SQL Server database name that will be mapped to our new catalog. Finally, click on "Create catalog." Note that this catalog is a foreign catalog and will not be the same catalog that you use for pipeline operations later.



Specify the appropriate access and sharing parameters for this catalog by choosing workspaces that can use the catalog, selecting an owner and granting the appropriate access privileges.



Finally, add the necessary tags required by your organization or for your benefit in the metadata for the connection.



At this point, the catalog will be created. You can then use Catalog Explorer to view and query the SQL Server database using federated queries through Lakehouse Federation.

Creating the replication pipeline

The Lakeflow Connect replication workflow is called an ingestion pipeline, which consists of two separate components: the gateway and the ingest pipeline.

1. **The gateway pipeline:** Creates a staging volume, connects to your SQL Server database, extracts a snapshot and change data and stores it in the staging volume. The gateway stores an initial seed snapshot followed by change data. It's recommended to run the gateway as a continuous pipeline to avoid gaps in change data due to change log retention policies on the source database.
2. **The ingest pipeline:** Applies the initial snapshot and subsequent change data from the gateway pipeline (stored in the staging volume) into destination streaming tables. It's important to note that only one ingestion pipeline per gateway is supported. If you need to write to multiple destination catalogs or schemas, you'll need to create multiple gateway-ingestion pipeline pairs.

You can create the gateway and ingest pipelines using:

1. Databricks User Interface (UI)
2. Databricks Command Line Interface (CLI)
3. Databricks Asset Bundle (DAB)
4. A notebook

These are covered in the documentation, but we'll show you the Databricks UI approach here for convenience. As always, check the documentation for the latest directions and updates.

PREREQUISITES

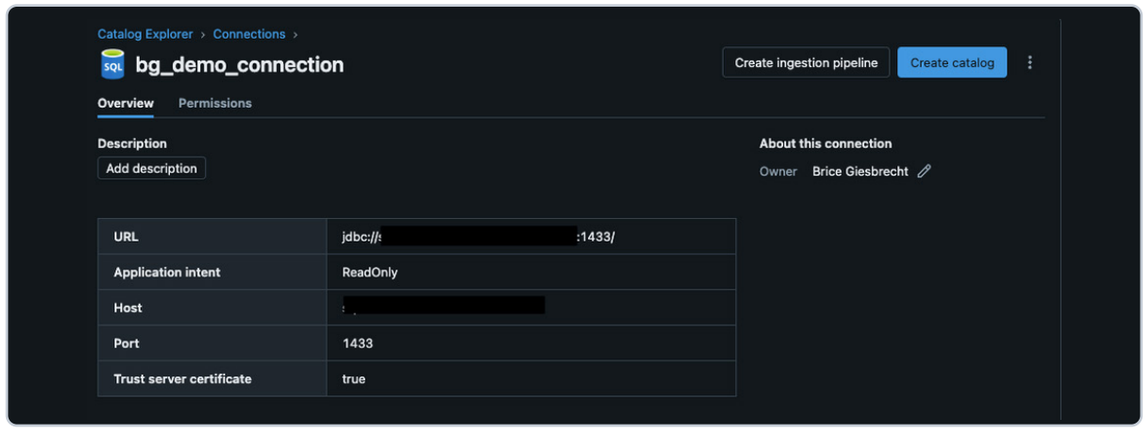
Before creating the ingestion pipeline, you will need to select or create catalogs and schemas to store the staging data from the gateway pipeline and the destination tables created by the ingest pipeline. These are referred to as the "staging catalog" and the "destination catalog" in subsequent steps.

The staging and destination catalogs can be the same catalog and share the same schema, which is the approach shown here, but you can also create and use separate catalogs and schemas. The gateway stores the source snapshot and change data in a volume, so there is no naming collision when using the same catalog and schema for both the staging catalog and the destination catalog.

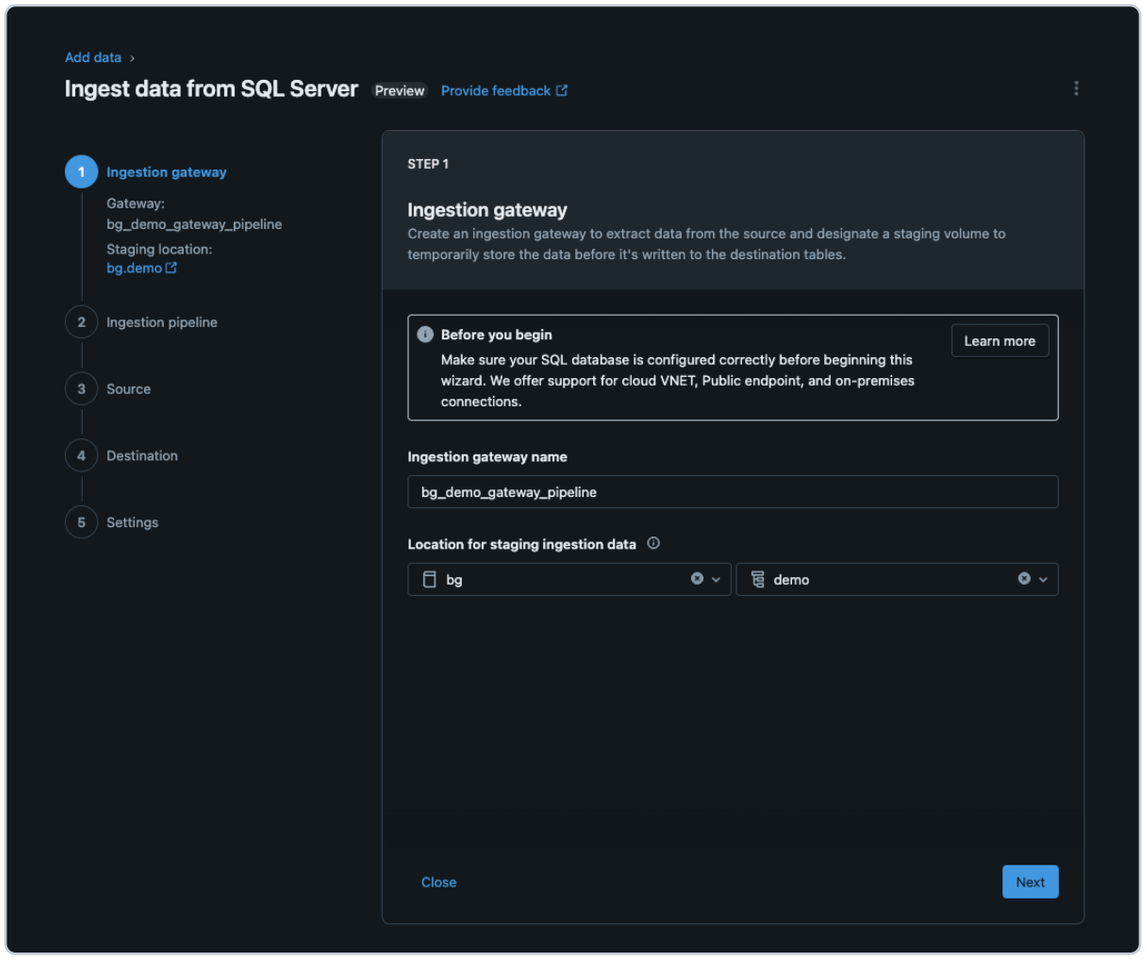
Note that the catalog created during the connection setup cannot be used, as that is a foreign catalog for query federation, and the ingestion requires a managed catalog.

CREATE THE INGESTION PIPELINE

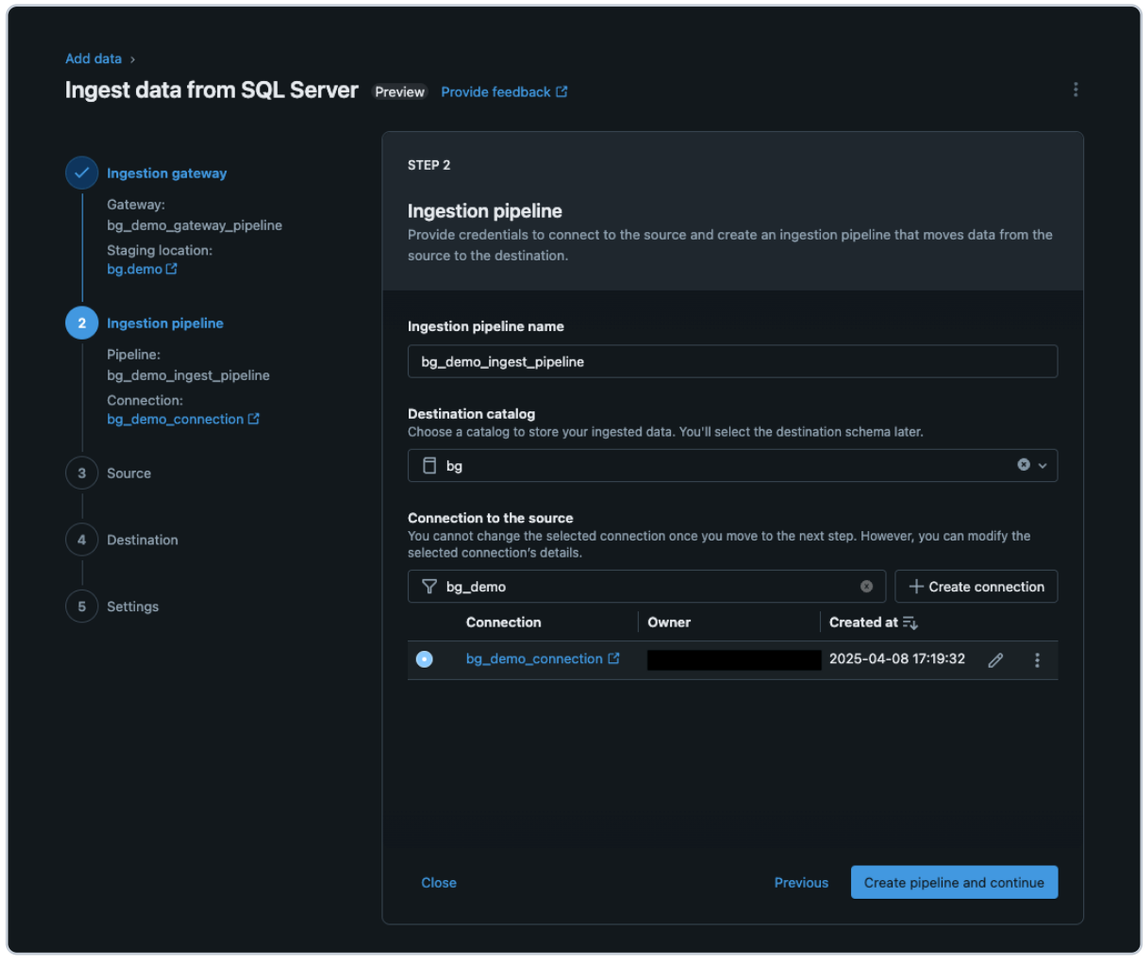
From the catalog explorer > external data > connection page, use the "Create ingestion pipeline" button.



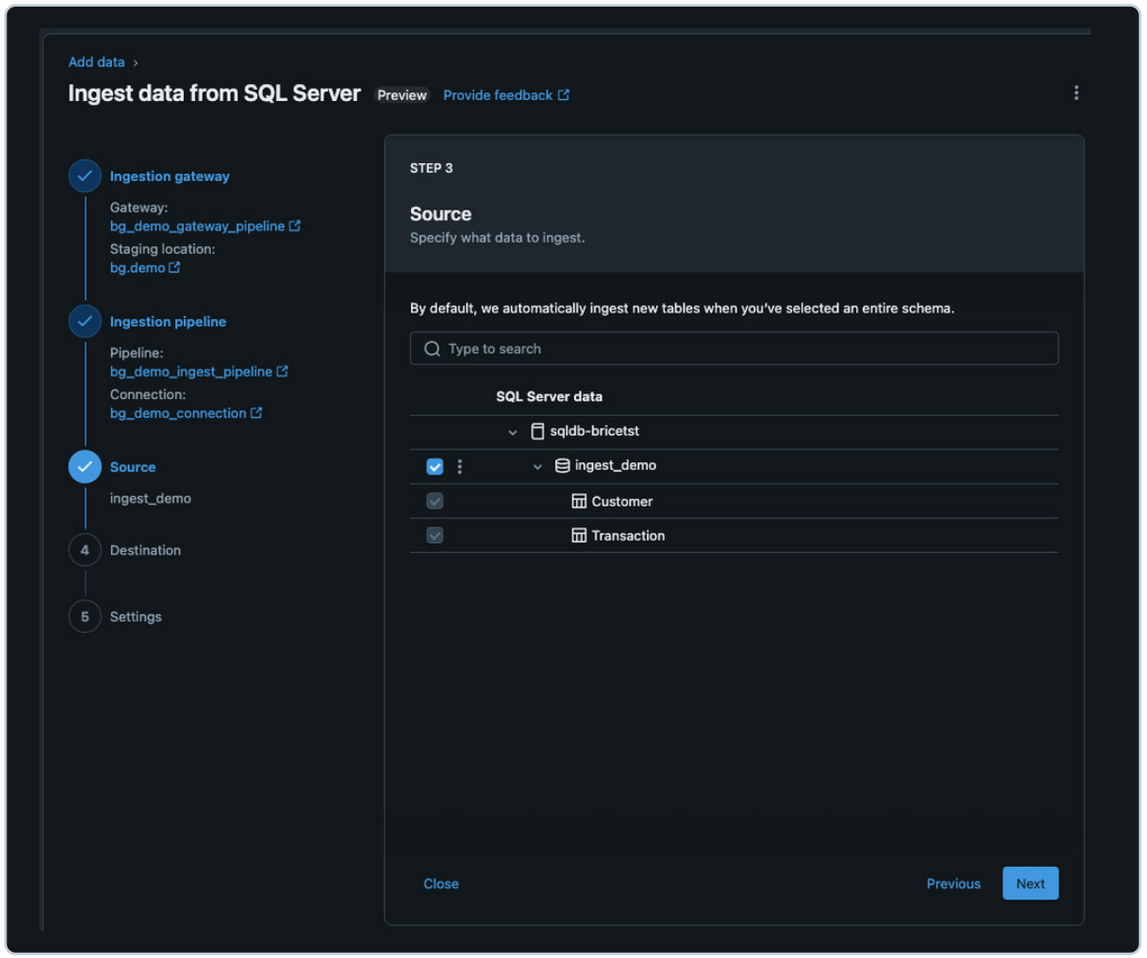
Enter a name for your ingestion gateway, as well as the staging catalog and schema, then continue to the next step.



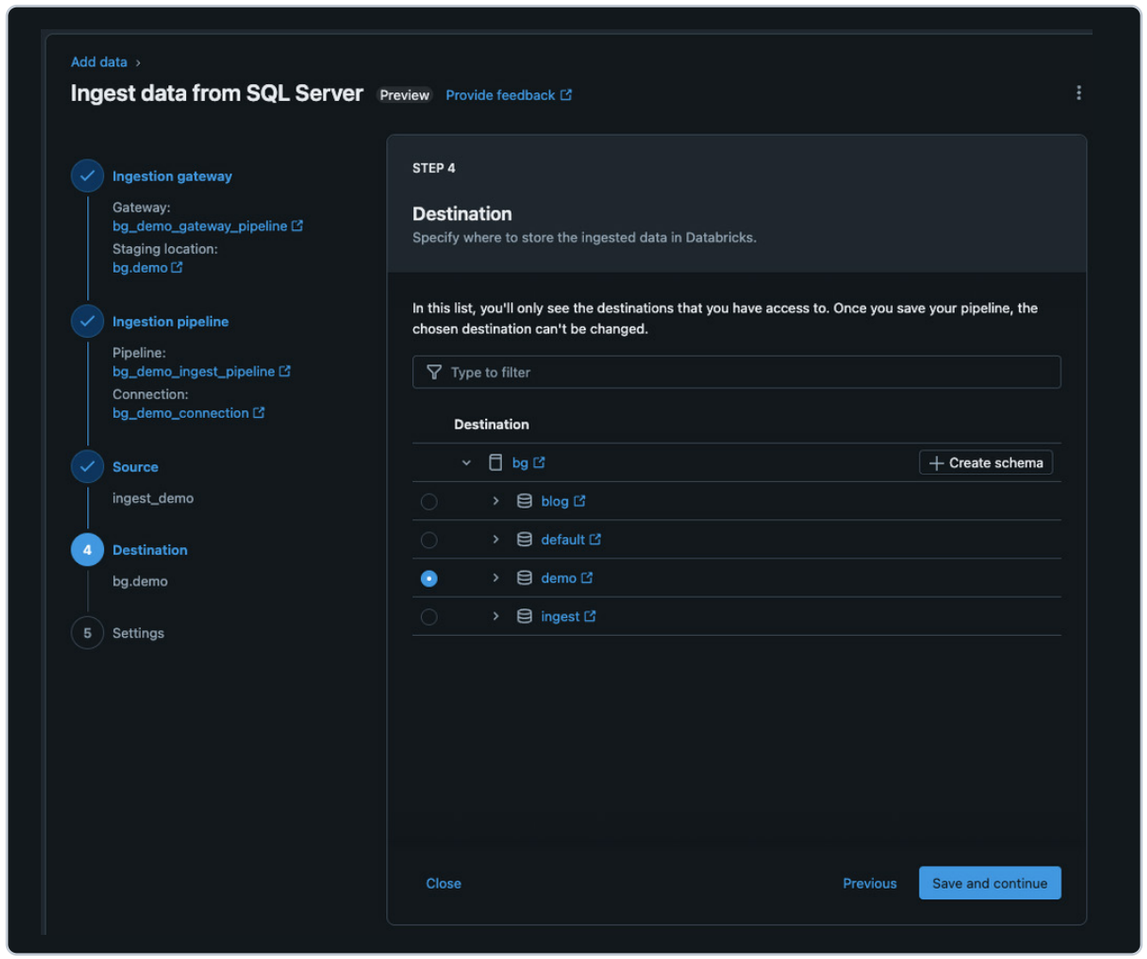
The second pipeline, the ingestion pipeline (part of the overall ingestion/ replication process), also requires a unique name, a connection and a destination catalog. You can use the same catalog as the staging data (shown here) or select a different catalog. Continue on to select the source tables.



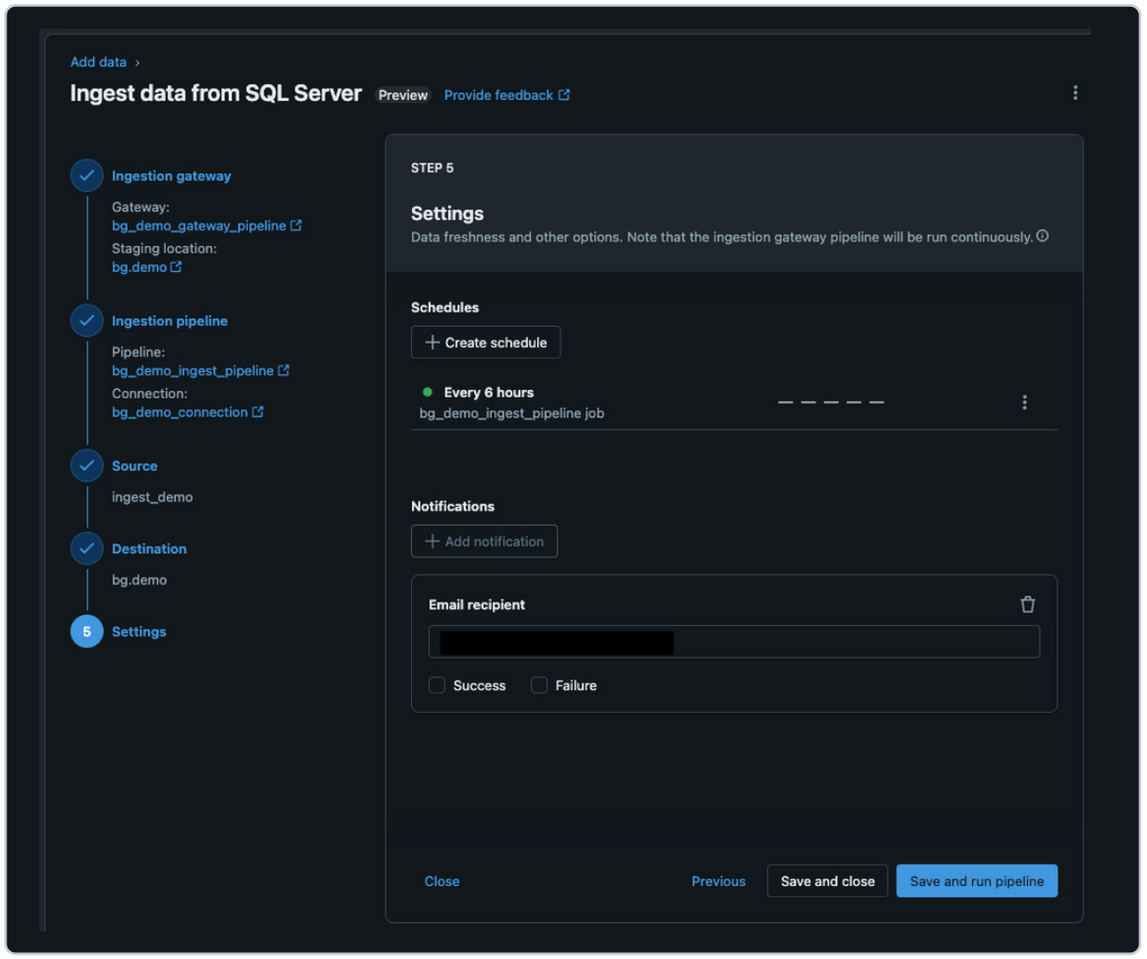
After a few moments, the source database, database schema and tables (per your connection user's permissions) will be displayed. Select the schema (shown here) or specific tables for ingestion. Selecting the schema will bring in all tables in that schema.



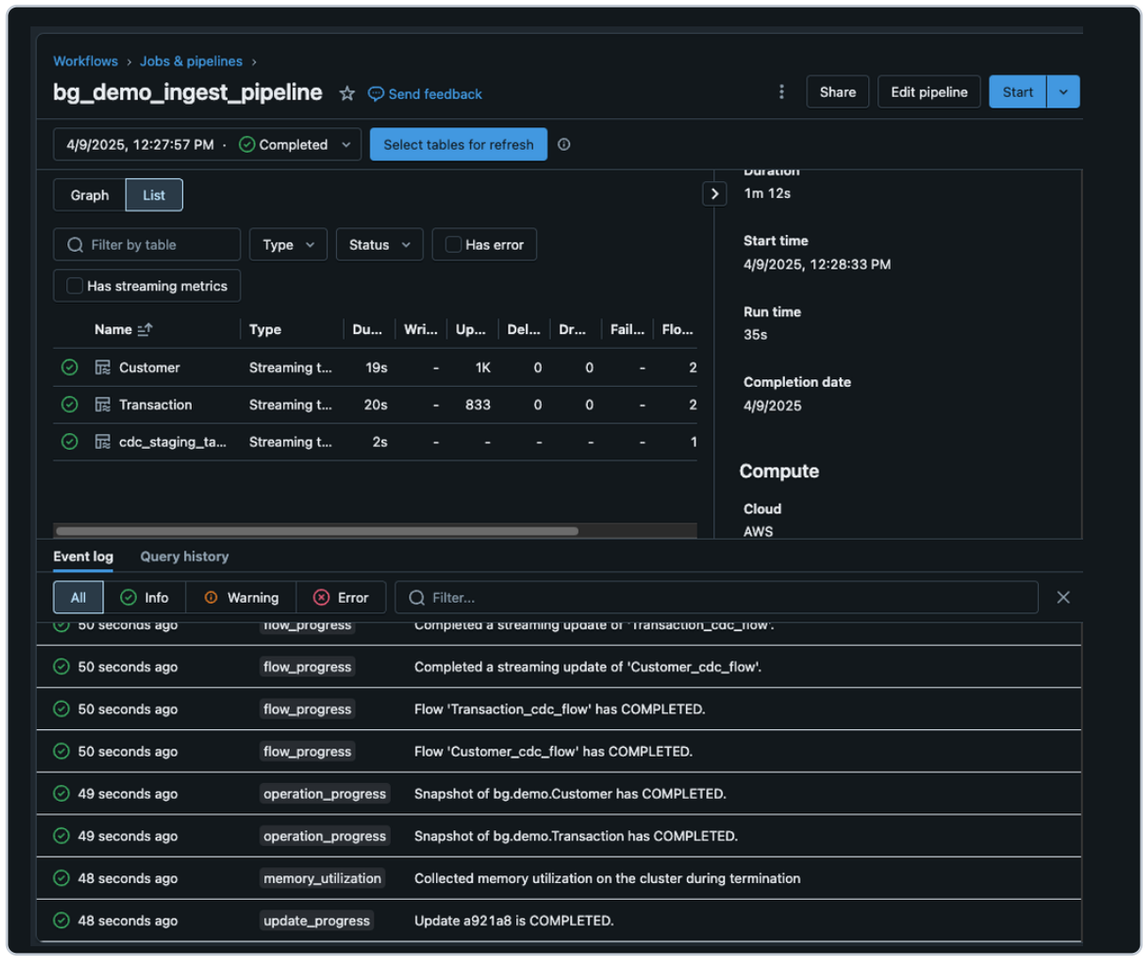
The destination schema is chosen next. Select your schedule and notification settings for this pipeline.



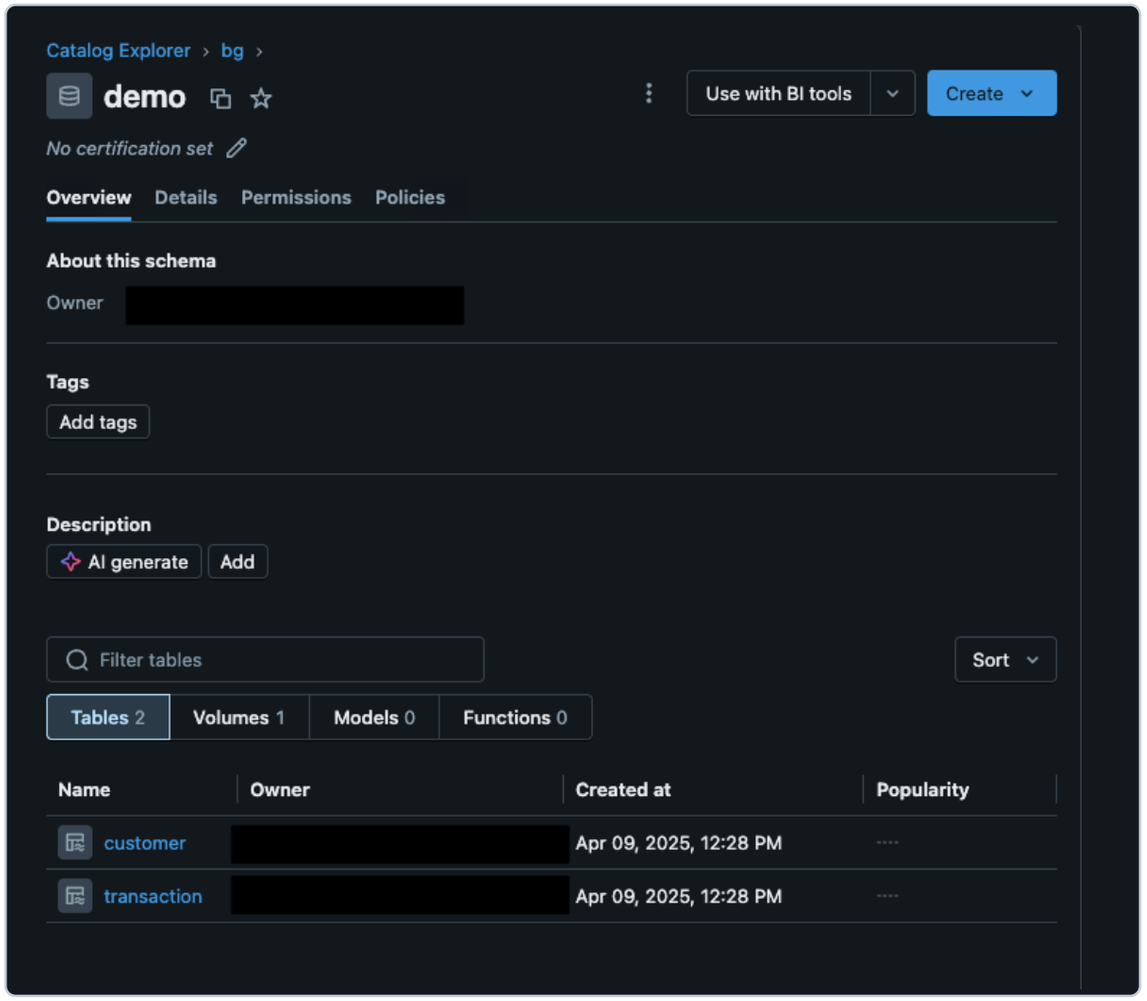
Set up your schedule and notification preferences.



Save and run the pipeline, or save and just close. Saving and running will launch both the gateway and ingest pipelines.



After a short time, this pipeline completed successfully and displayed the results and metrics from this run. Checking the destination catalog and schema shows the two newly created tables.

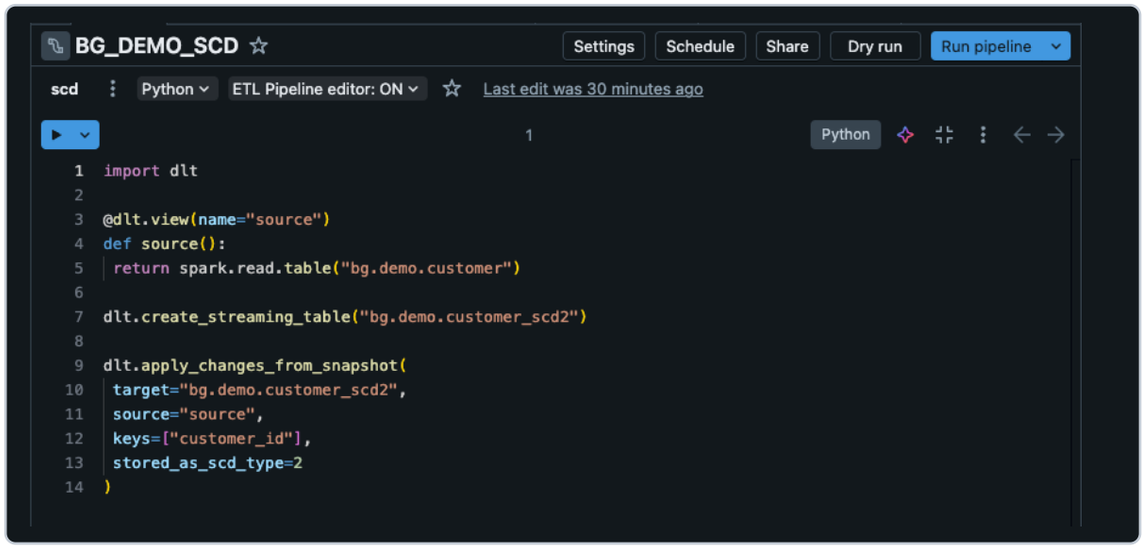


Creating the pipeline

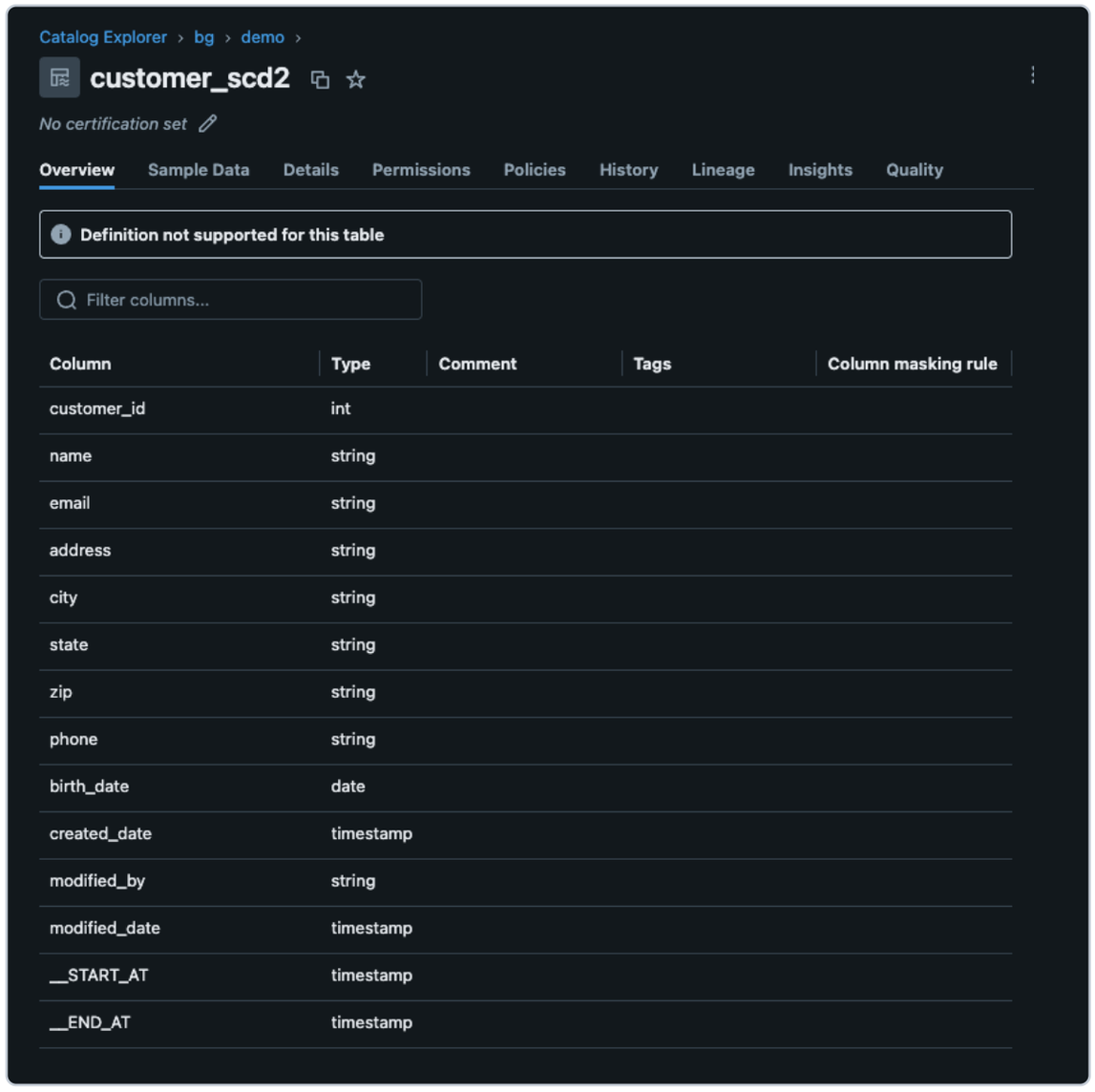
We are following the Databricks Medallion architecture, with a Bronze layer holding the raw data from our source system, and a Silver layer containing the Slowly Changing Dimension Type II (SCD type 2) change tracking of the history of our changing business entities. We are not using a Gold layer here, as our intent is to show CDC and SCD type 2.

1. **Bronze** — The Bronze layer traditionally holds the data as it exists in the source system. In this case, this is the data from SQL Server created in step 4. These are the replicated tables, customers and transactions.
2. **Silver** — This layer resembles the source system, containing the *cleansed and conformed* data, including aggregations and SCD type 2 changes. The data here is only lightly changed, containing our Bronze data plus our SCD type 2 changes.

Native SCD type 2 is on the roadmap for Databricks Lakeflow CDC. In the meantime, we will accomplish this using a Spark Declarative Pipeline that employs the "apply changes from snapshot" capability. There are different options available for apply_changes_from_snapshot, but we are only using a few in this example.

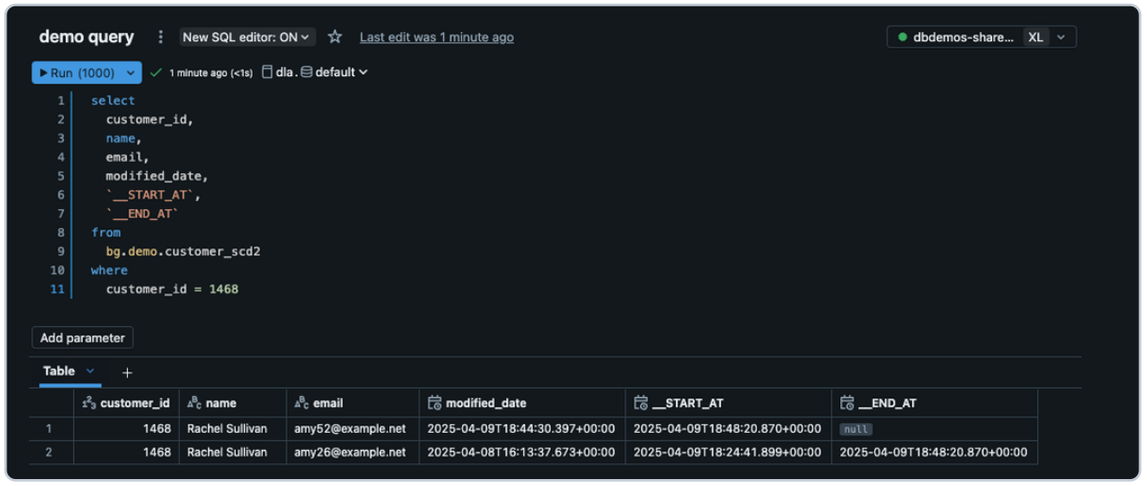


This pipeline reads from the replicated customer table and applies the changes from the last version into a new SCD type 2 table. Notice that in addition to the original columns, our SCD table now has '__START_AT' and '__END_AT' columns used to determine the 'current' value.



Querying the tables

1. Now, to demonstrate the SCD type 2 in action, we update our customer record to change the customer's email address
2. Displaying the email address for our fictitious customer shows amy26@example.net
3. After updating this email in the source database to amy52@example.net and running the ingest and SCD type 2 pipeline gives:



This is displaying the appropriate value along with the __START_AT and __END_AT values.

Conclusion

In summary, I expect that you will find that this chapter demonstrates Databricks' ability to handle data warehousing tasks, including data ingestion, transformation and maintaining version history, SCD type 2, all within one platform. It covers setting up Change Data Capture on a source database, creating connections and replication pipelines and implementing SCD type 2 using Spark Declarative Pipelines. The example shows how changes in source data are captured and reflected in the destination tables, including historical versions of records. Previously, developers were forced to use many tools to accomplish the same type of end-to-end tasks, but with Databricks, it is all accomplished in one open, unified and secure platform.

Optimizing Materialized Views Recomputes

By Andrea Tardif and Justin Edrington

Optimize incremental computation of Materialized Views

While digital-native companies recognize the critical role AI plays in driving innovation, many still face challenges in making their ETL pipelines operationally efficient.

Materialized Views (MVs) exist to store precomputed query results as managed tables, allowing users to access complex or frequently used data much faster by avoiding repeated computation of the same queries.

MVs improve query performance, reduce computational costs and simplify transformation processes.

Spark Declarative Pipelines provide a straightforward, declarative approach to building data pipelines, supporting both full and incremental refreshes for MVs. Databricks pipelines are powered by the Enzyme engine, which efficiently keeps MVs up to date by tracking how new data affects the query results and only updating what is necessary. It utilizes an internal cost model to select among various techniques, including those employed in materialized views and manual ETL patterns commonly used.

This chapter will discuss detecting unexpected full recomputes and optimizing pipelines for proper incremental MV refreshes.

Key architectural considerations

SCENARIOS FOR INCREMENTAL AND FULL REFRESHES

A full recompute overwrites the results in the MV by reprocessing all available data from the source. This can become costly and time-consuming since it requires reprocessing the entire underlying dataset, even if only a small portion has changed.

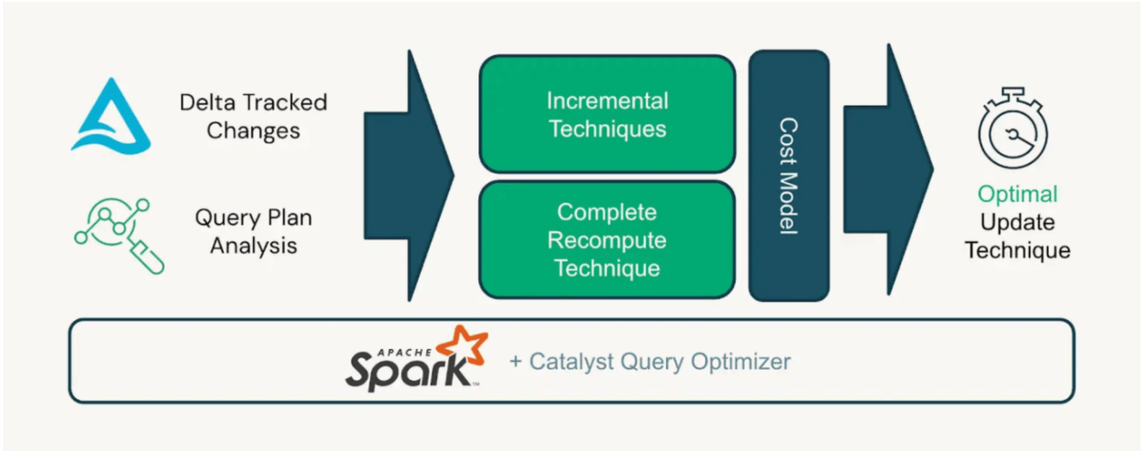
While incremental refresh is generally preferred for efficiency, there are situations where a full refresh is more appropriate. Our cost model follows these high-level guidelines:

- Use a full refresh when there are major changes in the underlying data, especially if records have been deleted or modified in ways that the cost model can efficiently compute and apply the incremental change
- Use an incremental refresh when changes are relatively minor and the source tables are frequently updated — this approach helps reduce compute costs

ENZYME COMPUTE ENGINE

Instead of recomputing entire tables or views from scratch every time new data arrives or changes occur, Enzyme intelligently determines and processes only the new or changed data. This approach dramatically reduces resource consumption and latency compared to traditional batch ETL methods.

The diagram below outlines how the Enzyme engine intelligently determines the optimal way to update a materialized view.



The Enzyme engine selects the update strategy and determines whether to perform an incremental or full refresh based on its internal cost model, optimizing for performance and compute efficiency.

ENABLE DELTA TABLE FEATURES

Enabling row tracking on source tables is required to incrementalize the MV recompute.

Row tracking helps detect which rows have changed since the last MV refresh. It enables Databricks to track row-level lineage in a Delta table and is required for specific incremental updates to materialized views.

Enabling deletion vectors is an optional feature. Deletion vectors allow Databricks to track which rows have been deleted from the source table. This prevents the need to rewrite full files and avoids rewriting entire files when only a few rows are deleted.

To enable these table features on the source table, leverage the following SQL code:

SQL

```
ALTER TABLE my_table SET TBLPROPERTIES (
  'delta.enableRowTracking' = 'true',
  'delta.enableDeletionVectors' = 'true'
);
```

Technical solution breakdown

This next section will walk through an example of how to detect when a pipeline triggers a full recompute vs. an incremental refresh on an MV and how to encourage an incremental refresh.

This technical walkthrough follows these high-level steps:

1. Generate a Delta table with randomly generated data
2. Create and use an SDP to create a materialized view
3. Add a non-deterministic function to the materialized view
4. Rerun the pipeline and observe the impact on the refresh behavior
5. Update the pipeline to restore incremental refresh
6. Query the pipeline event log to inspect the refresh technique

To follow along with this example, please clone this script: MV_Incremental_Technical_Breakdown.ipynb

Within the `run_mv_refresh_demo()` function, the first step generates a Delta table with randomly generated data:

SQL

```
CREATE CATALOG IF NOT EXISTS {catalog_name};
CREATE SCHEMA IF NOT EXISTS {catalog_name}.demo;

CREATE OR REPLACE TABLE random_data (
id STRING,
name STRING)

USING DELTA
TBLPROPERTIES (
'delta.autoOptimize.optimizeWrite' = 'true',
'delta.autoOptimize.autoCompact' = 'true',
'delta.enableRowTracking' = 'true',
'delta.enableDeletionVectors' = 'true',
'delta.feature.allowColumnDefaults' = 'supported');
```

Next, the following function is run to insert randomly generated data. This is run before each new pipeline run to ensure that new records are available for aggregation.

SQL

```
-- Insert new records into tables
INSERT INTO {catalog_name}.demo.random_data
SELECT
  CAST(rand() * 100000 AS BIGINT) AS id,
  CASE CAST(rand() * 5 AS INT)
    WHEN 0 THEN 'Alice'
    WHEN 1 THEN 'Bob'
    WHEN 2 THEN 'Charlie'
    WHEN 3 THEN 'Diana'
    ELSE 'Eve'
  END AS name
FROM RANGE(5);
```

Then, the Databricks SDK is used to create and deploy the SDP.

PYTHON

```
pipeline = w.pipelines.create(
    catalog={catalog_name},
    continuous=False,
    channel="PREVIEW",
    name=f"{catalog_name}_mv_demo_pipeline",
    schema="demo",
    libraries=[
        PipelineLibrary(glob=PathPattern(include=mv_path))
    ],
    serverless=True,
    event_log=EventLogSpec(
catalog=catalog_name,
schema=schema,
name="event_log_incremental_refresh_demo"))
```

MVs can be created through either a serverless SDP or Databricks SQL (DBSQL) and behave the same. DBSQL MVs launch a managed serverless SDP that is coupled to the MV under the hood. This example leverages a serverless SDP to utilize various features, such as publishing the event log, but it would behave the same if a DBSQL MV were used.

Once the pipeline is successfully created, the function will then run an update on the pipeline:

PYTHON

```
w.pipelines.start_update(pipeline.pipeline_id)
```

After the pipeline has successfully run and created the initial materialized view, the next step is to add more data and refresh the view. After running the pipeline, check the event log to review the refresh behavior.

SQL

```
SELECT timestamp, message
FROM {catalog_name}.demo.event_log_incremental_refresh_demo
WHERE event_type = 'planning_information'
ORDER BY timestamp DESC;
```

The results show that the materialized view was incrementally refreshed, indicated by the GROUP_AGGREGATE message.

Run #	Message	Flow type
2	Flow '<catalog_name>.demo.random_data_mv' has been planned in DLT to be executed as GROUP_AGGREGATE.	No non-deterministic function. Incrementally refreshed.
1	Flow '<catalog_name>.demo.random_data_mv' has been planned in DLT to be executed as COMPLETE_RECOMPUTE.	Initial Run. Full recompute.

Next, to demonstrate how adding a non-deterministic function like RANDOM() can prevent the materialized view from incrementally refreshing, the MV is updated to the following:

SQL

```
CREATE MATERIALIZED VIEW random_data_MV
SELECT
    name,
    RANDOM() AS random_number, -- added!
    COUNT(id) as name_count
FROM {catalog_name}.demo.random_data
GROUP BY name;
```

To account for changes in the MV and to demonstrate the non-deterministic function, the pipeline is executed twice and data is added. The event log is then queried again, and the results show a full recompute.

Run #	Message	Explanation
4	Flow 'andrea_tardif.demo.random_data_mv' has been planned in DLT to be executed as COMPLETE_RECOMPUTE.	MV includes non-deterministic — full recompute triggered.
3	Flow '<catalog_name>.demo.random_data_mv' has been planned in DLT to be executed as COMPLETE_RECOMPUTE.	MV definition changed — full recompute triggered.
2	Flow '<catalog_name>.demo.random_data_mv' has been planned in DLT to be executed as GROUP_AGGREGATE.	Incremental refresh — no non-deterministic functions present.
1	Flow '<catalog_name>.demo.random_data_mv' has been planned in DLT to be executed as COMPLETE_RECOMPUTE.	Initial run — full recompute required.

By adding non-deterministic functions, such as RANDOM() or CURRENT_DATE(), the MV cannot incrementally refresh because the output cannot be predicted based solely on changes in the source data.

Within the pipeline event log details, under *planning_information*, the JSON event details provide the following reason for preventing incrementalization:

JSON

```
{
"issue_type": "PLAN_NOT_DETERMINISTIC",
"prevent_incrementalization": true,
"expression_name": "Rand",
"plan_not_deterministic_sub_type": "NON_DETERMINISTIC_EXPRESSION"
}
```

If having a non-deterministic function is necessary for your analysis, a better approach is to push that value into the source table itself rather than calculating it dynamically in the materialized view. We will accomplish this by moving the random_number column to pull from the source table instead of adding it in at the MV level.

Below is the updated materialized view query to reference the static random_number column within the MV:

SQL

```
CREATE MATERIALIZED VIEW random_data_MV
SELECT
    name,
    random_number, -- updated to pull from source table!
    COUNT(id) as name_count
FROM {catalog_name}.demo.random_data
GROUP BY name, random_number;
```

Once new data is added and the pipeline is run again, query the event log. The output shows that the MV performed a GROUP_AGGREGATE rather than a COMPLETE_RECOMPUTE.

Run #	Message	Explanation
5	Flow '<catalog_name>.demo.random_data_mv' has been planned in DLT to be executed as GROUP_AGGREGATE.	MV uses deterministic logic — incremental refresh.
4	Flow '<catalog_name>.demo.random_data_mv' has been planned in DLT to be executed as COMPLETE_RECOMPUTE.	MV includes non-deterministic — full recompute triggered.
3	Flow '<catalog_name>.demo.random_data_mv' has been planned in DLT to be executed as COMPLETE_RECOMPUTE.	MV definition changed — full recompute triggered.
2	Flow '<catalog_name>.demo.random_data_mv' has been planned in DLT to be executed as GROUP_AGGREGATE.	Incremental refresh — no non-deterministic functions present.
1	Flow '<catalog_name>.demo.random_data_mv' has been planned in DLT to be executed as COMPLETE_RECOMPUTE.	Initial run — full recompute required.

A full refresh can be automatically triggered by the pipeline under the following conditions:

- Use of **non-deterministic functions** like UUID() and RANDOM()
- Creating materialized views that involve **complex joins**, such as cross, full outer, semi, anti and large numbers of joins
- **Enzyme** determines that it is **less computationally expensive** to perform a full recompute

Learn more about incremental refresh compatible functions here.

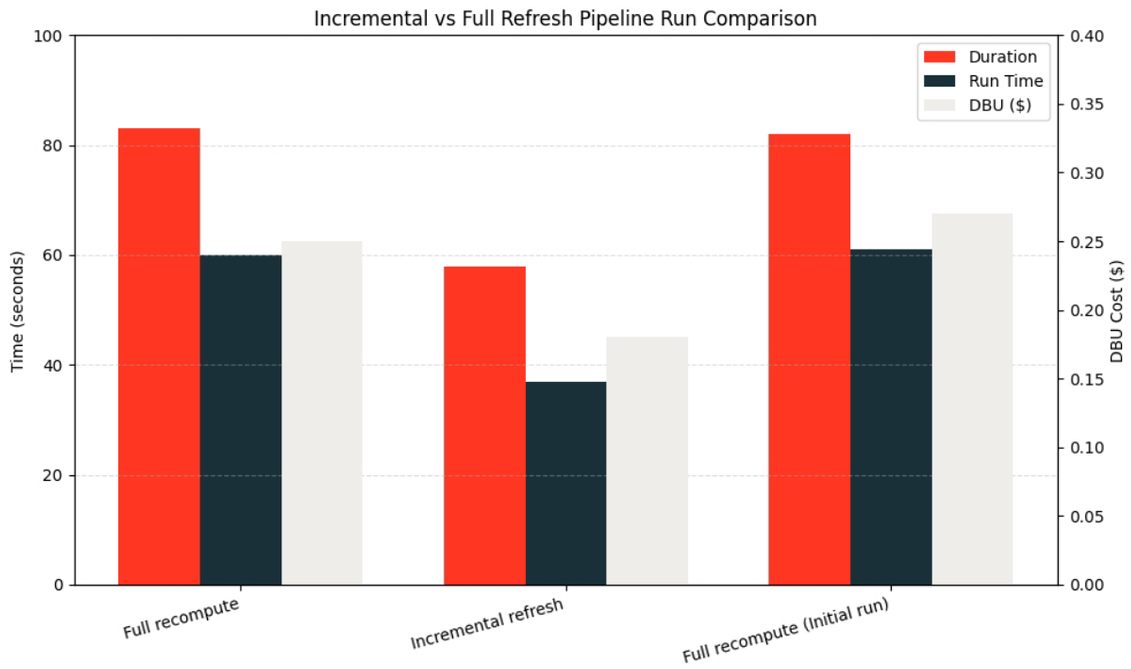
Real-world data volume

In most cases, the data ingestion is much larger than inserting five rows. To illustrate this, let's insert 1 billion rows into the initial load and then 10 million into each pipeline run.

Using dbldatagen to randomly generate data and the Databricks SDK to create and run an SDP, 1 billion rows were inserted into the source table, and the pipeline was run to generate the MV. Then, 10 million rows were added to the source data, and the MV was incrementally refreshed. Afterward, the pipeline was force refreshed to perform a full recompute.

Once the pipeline completes, use the list_pipeline_events and the billing system table, merged on dlt_update_id, to determine the cost per update.

As shown in the graph below, the incremental refresh was twice as fast and cheaper than the full refresh.



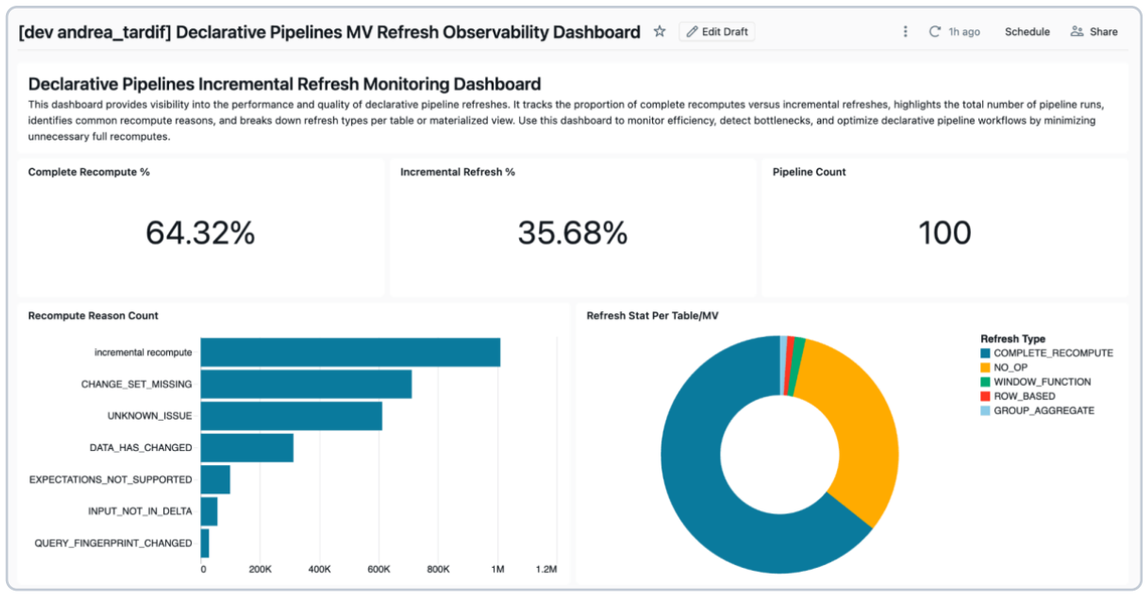
Operational considerations

Strong monitoring, observability and automation practices are crucial for fully realizing the benefits of incremental refreshes in declarative pipelines. The following section outlines how to leverage Databricks monitoring capabilities to track pipeline refreshes and cost.

MONITORING PIPELINE REFRESHES

Tools like the event log and the SDP UI provide visibility into pipeline execution patterns, helping detect when various refreshes occur.

We've included an accelerator tool to help teams track and analyze materialized view refresh behavior. This solution leverages AI/BI dashboards to provide visibility into refresh patterns. It uses the Databricks SDK to retrieve all pipelines in your configured workspace, gather event details for the pipelines and then produce a dashboard similar to the one below.



Github link: monitoring-declarative-pipeline-refresh-behavior

Key takeaways

Incrementalizing the material view refreshes allows Databricks to process only new or changed data in the source tables, improving performance and reducing costs.

With MVs, avoid using non-deterministic functions (i.e., CURRENT_DATE() and RANDOM()) and limit query complexity (i.e., excessive joins) to enable efficient incremental refreshes. Ignoring unexpected full recomputes on MVs that could be refactored to be incremental recomputes could lead to:

- Increased compute costs
- Slower data freshness for downstream applications
- Pipeline bottlenecks as data volumes scale

With serverless compute, SDPs leverage the built-in execution model, allowing Enzyme to perform an incremental or full recompute based on the overall pipeline computation cost.

Leverage the accelerator tool to monitor the behavior of all your pipelines in an AI/BI dashboard to detect unexpected full recomputes.

In conclusion, to create efficient materialized view refreshes, follow these best practices:

- Use deterministic logic where applicable
- Refactor queries to avoid non-deterministic functions
- Simplify join logic
- Enable row tracking on the source tables

Next steps and additional resources

Review your MV refresh types today

Databricks DSAs accelerate data and AI initiatives across organizations. They provide architectural leadership, optimize platforms for cost and performance, enhance developer experience and drive successful project execution. DSAs bridge the gap between initial deployment and production-grade solutions — working closely with various teams, including data engineering, technical leads, executives and other stakeholders to ensure tailored solutions and faster time to value. To benefit from a custom execution plan, strategic guidance and support throughout your data and AI journey from a DSA, please contact your Databricks Account Team.

Additional resources

- Load and process data incrementally with Spark Declarative Pipelines flows | Databricks Documentation
- Incremental refresh for materialized views | Databricks Documentation

Create an SDP and review MV incremental refresh types today.

Reverse ETL With Lakebase: Activate Your Lakehouse Data for Operational Analytics

Serve data from the lakehouse to applications reliably and at scale, without custom pipelines

By Firas Farah and Yatish Anand

Introduction: Analytics and operations are converging

Applications today can't rely on raw events alone. They need curated, contextual and actionable data from the lakehouse to power personalization, automation and intelligent user experiences.

Delivering that data reliably with low latency has been a challenge, often requiring complex pipelines and custom infrastructure.

Lakebase addresses this problem. It pairs a high-performance Postgres database with native lakehouse integration, making reverse ETL simple and reliable.

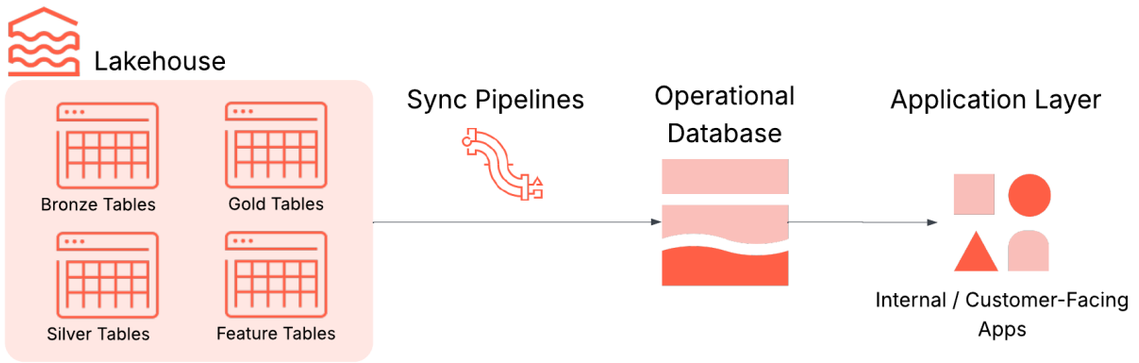
What is reverse ETL?

Reverse ETL syncs high-quality data from a lakehouse into the operational systems that power applications. This ensures that trusted datasets and AI-driven insights flow directly into applications that power personalization, recommendations, fraud detection and real-time decisioning.

Without reverse ETL, insights remain in the lakehouse and don't reach the applications that require them. The lakehouse is where data gets cleaned, enriched and turned into analytics, but it isn't built for low-latency app interactions or transactional workloads. That's where Lakebase comes in, delivering trusted lakehouse data directly into the tools where it drives action, without custom pipelines.

In practice, reverse ETL typically involves four key components, all integrated into Lakebase:

- **Lakehouse:** Stores curated, high-quality data used to drive decisions, such as business-level aggregate tables (aka "Gold tables"), engineered features and ML inference outputs
- **Syncing pipelines:** Move relevant data into operational stores with scheduling, freshness guarantees and monitoring
- **Operational database:** Optimized for high concurrency, low latency and ACID transactions
- **Applications:** The final destination where insights become action, whether in customer-facing applications, internal tools, APIs or dashboards



CHALLENGES OF REVERSE ETL TODAY

Reverse ETL looks simple, but in practice, most teams face the same challenges:

- **Brittle, custom-built ETL pipelines:** These pipelines often require streaming infrastructure, schema management, error handling and orchestration. They're brittle and resource-intensive to maintain.
- **Multiple, disconnected systems:** Separate stacks for analytics and operations mean more infrastructure to manage, more authentication layers and more chances for format mismatches
- **Inconsistent governance models**: Analytical and operational systems usually live in different policy domains, making it difficult to apply consistent quality controls and audit policies

These challenges create friction for both developers and the business, slowing down efforts to reliably activate data and deliver intelligent, real-time applications.

Lakebase: Integrated by default for easy reverse ETL

Lakebase removes these barriers and transforms reverse ETL into a fully managed, integrated workflow. It combines a high-performance Postgres engine, deep lakehouse integration and built-in data synchronization so that fresh insights flow into applications without extra infrastructure.

These capabilities of Lakebase are especially valuable for reverse ETL:

- **Deep lakehouse integration:** Sync data from lakehouse tables to Lakebase on a snapshot, scheduled, or continuous basis, without building or managing external ETL jobs. This replaces the complexity of custom pipelines, retries and monitoring with a native, managed experience.
- **Fully managed Postgres:** Built on open source Postgres, Lakebase supports ACID transactions, indexes, joins and extensions such as PostGIS and pgvector. You can connect with existing drivers and tools like pgAdmin or JDBC, avoiding the need to learn new database technologies or maintain separate OLTP infrastructure.
- **Scalable, resilient architecture:** Lakebase separates compute and storage for independent scaling, delivering sub-10 ms query latency and thousands of QPS. Enterprise-grade features include multi-AZ high availability, point-in-time recovery and encrypted storage. This removes the scaling and resiliency challenges of self-managed databases.
- **Integrated security and governance:** Register Lakebase with Unity Catalog to bring operational data into your centralized governance framework, covering audit trails and permissions at the catalog level. Access via Postgres protocol still uses native Postgres roles and permissions, ensuring authentic transactional security while fitting into your broader Databricks governance model.
- **Cloud-agnostic architecture:** Deploy Lakebase alongside your lakehouse in your preferred cloud environment without re-architecting your workflows

With these capabilities in the Databricks Data Intelligence Platform, Lakebase replaces the fragmented reverse ETL setup that relies on custom pipelines, standalone OLTP systems and separate governance. It delivers an integrated, high-performance and secure service, ensuring that analytical insights flow into applications more quickly — with less operational effort and with governance preserved.

Sample use case: Building an intelligent support portal with Lakebase

As a practical example, let's walk through how to build an intelligent support portal powered by Lakebase. This interactive portal helps support teams triage incoming incidents using ML-driven insights from the lakehouse, such as predicted escalation risk and recommended actions, while allowing users to assign ownership, track status and leave comments on each ticket.

Lakebase makes this possible by syncing predictions into Postgres while also storing updates from the app. The result is a support portal that combines analytics with live operations. The same approach applies to many other use cases, including personalization engines and ML-driven dashboards.

Step 1: Sync predictions from the Lakehouse to Lakebase

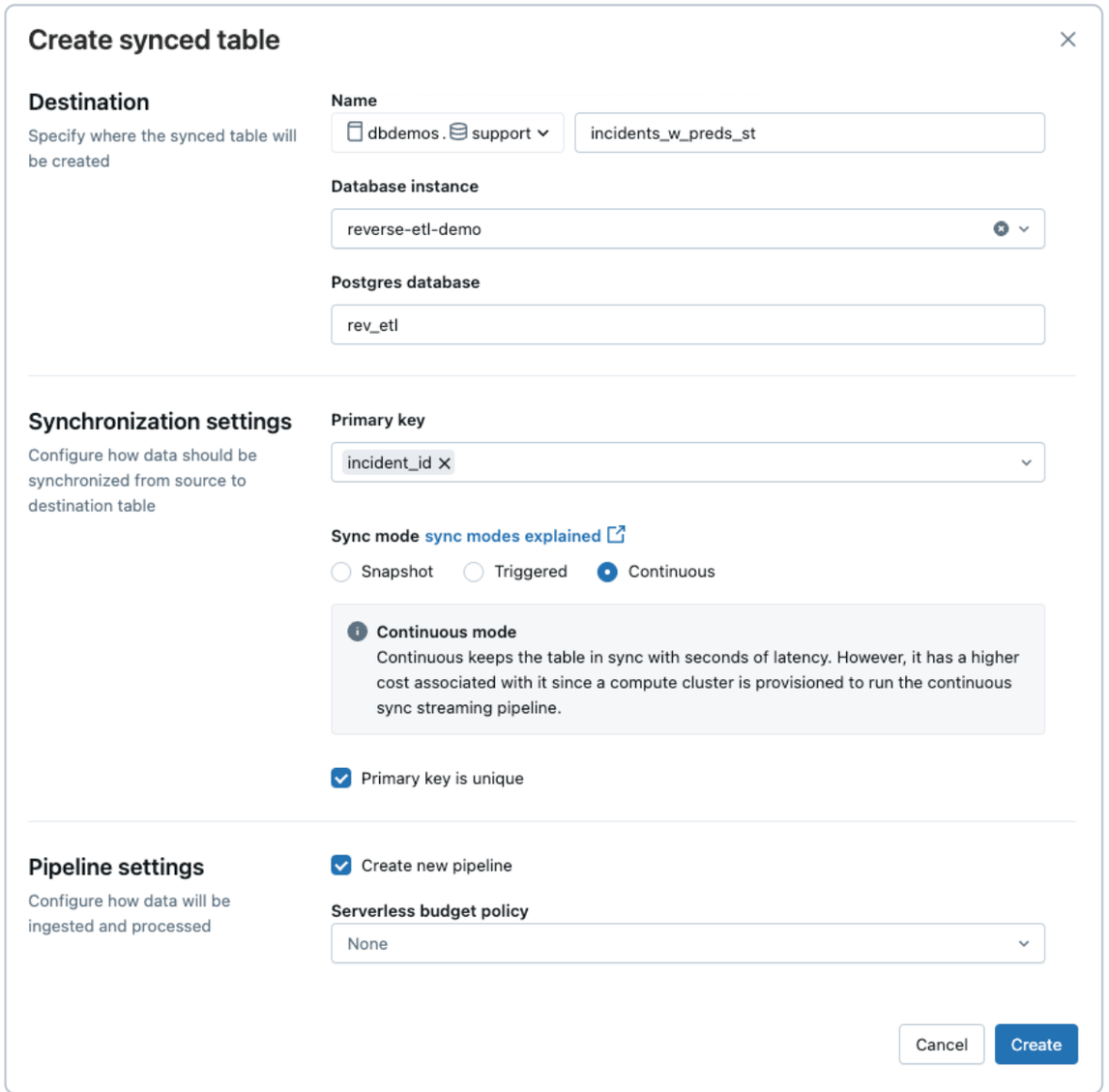
The incident data, enriched with ML predictions, lives in a Delta table and is updated in near real time via a streaming pipeline. To power the support app, we use Lakebase reverse ETL to continuously sync this Delta table to a Postgres table.

In the UI, we select:

- Sync mode: Continuous for low-latency updates
- Primary key: incident_id

This ensures the app reflects the latest data with minimal delay.

Note: You can also create the sync pipeline programmatically using the Databricks SDK.



Step 2: Create a state table for user inputs

The support app also needs a table to store user-entered data like ownership, status and comments. Since this data is written from the app, it should go into a separate table in Lakebase (rather than the synced table).

Here's the schema:

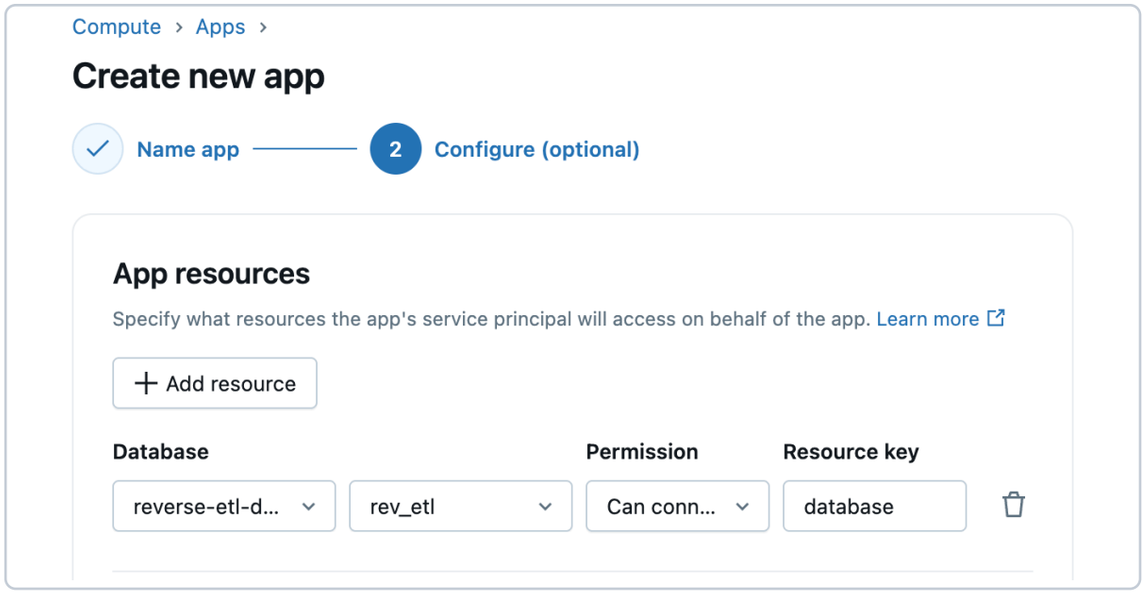
SQL

```
CREATE TABLE support.user_updates (
  incident_id TEXT PRIMARY KEY, owner TEXT, comment TEXT, status
TEXT
)
```

This design ensures that reverse ETL stays unidirectional (Lakehouse → Lakebase), while still allowing interactive updates via the app.

Step 3: Configure Lakebase access in Databricks Apps

Databricks Apps support first-class integration with Lakebase. When creating your app, simply add Lakebase as an app resource and select the Lakebase instance and database. Databricks automatically provisions a corresponding Postgres role for the app's service principal, streamlining app-to-database connectivity. You can then grant this role the required database, schema and table permissions.



Step 4: Deploy your app code

With your data synced and permissions in place, you can now deploy the Flask app that powers the support portal. The app connects to Lakebase via Postgres and serves a rich dashboard with charts, filters and interactivity.

Conclusion

Bringing analytical insights into operational applications no longer needs to be a complex, brittle process. With Lakebase, reverse ETL becomes a fully managed and integrated capability. It combines the performance of a Postgres engine, the reliability of a scalable architecture and the governance of the Databricks Platform.

Whether you are powering an intelligent support portal or building other real-time, data-driven experiences, Lakebase reduces engineering overhead and speeds up the path from insight to action.

To learn more about how to create synced tables in Lakebase, check out our documentation and get started today.

Production-Grade Data Quality Using Spark Declarative Pipelines

By Matt Holzapfel

Good data quality isn't just a nice-to-have — it's a necessity, especially in regulated industries like financial services. Companies in the industry are investing in AI to automate processes and decision-making, creating increasingly sophisticated data ecosystems that demand rigorous quality control.

The success of these AI initiatives relies on high-quality data. End users know poor data quality when they see it, but to be successful with data and AI, teams must go beyond anecdotes and have quantitative measurements. As the adage goes, "You can't manage what you don't measure."

The Databricks Data Intelligence Platform provides a unified solution for data and AI with robust data engineering capabilities for building and monitoring data pipelines. Spark Declarative Pipelines lets users create batch and streaming data pipelines in SQL and Python. This includes giving users the ability to apply quality constraints, known as 'expectations', to capture data quality metrics and automatically take action when issues are found.

Measuring what matters requires having an effective data quality framework in mind before starting to implement expectations. We'll share some of the best practices we've seen so that you can feel confident in the quality of the data feeding your downstream applications and consumers.

Upstream data drift equals downstream chaos

Anyone who has worked with data pipelines knows the challenge. Upstream data changes unexpectedly, and downstream teams discover the issue only after it causes problems. The upstream changes might be subtle — new attributes being collected, new values being allowed in existing fields, or schema changes — but the impact on business processes or data consumers' trust can be significant.

Possibly the worst part is that data pipelines often continue to execute normally despite their polluted nature. ETL jobs complete successfully, dashboards update on schedule and everything looks green.

Best case: A data consumer with a strong intuition of the data catches the error, and it's addressed before it can impact the customer experience.

Worst case: The issue lingers for weeks and requires a multi-team investigation to identify and resolve the root cause. Adding insult to injury, when issues like this happen in highly regulated industries like financial services, they often require detailed root cause analyses to be completed and can trigger regulatory reviews.

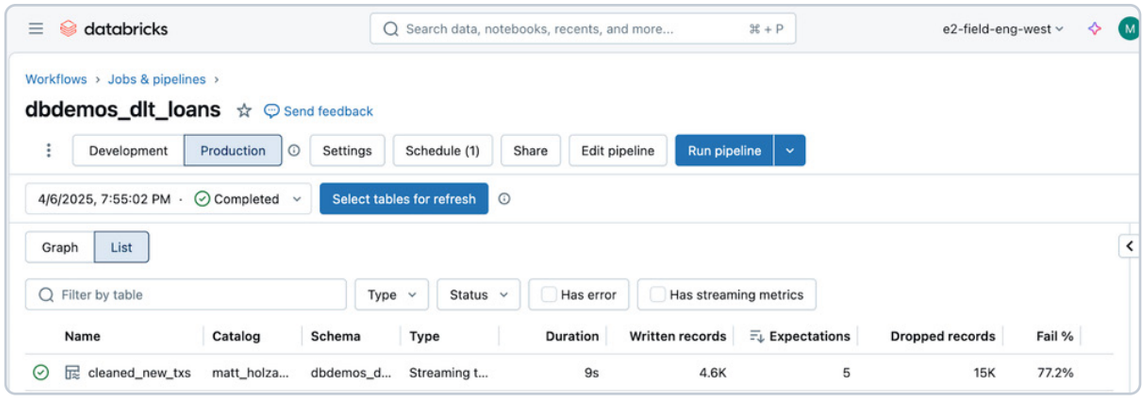
Applying a data quality framework to protect your pipelines from the unexpected

It's not always painfully obvious when poor-quality data has infiltrated a pipeline. A good data quality framework recognizes this and should be designed such that:

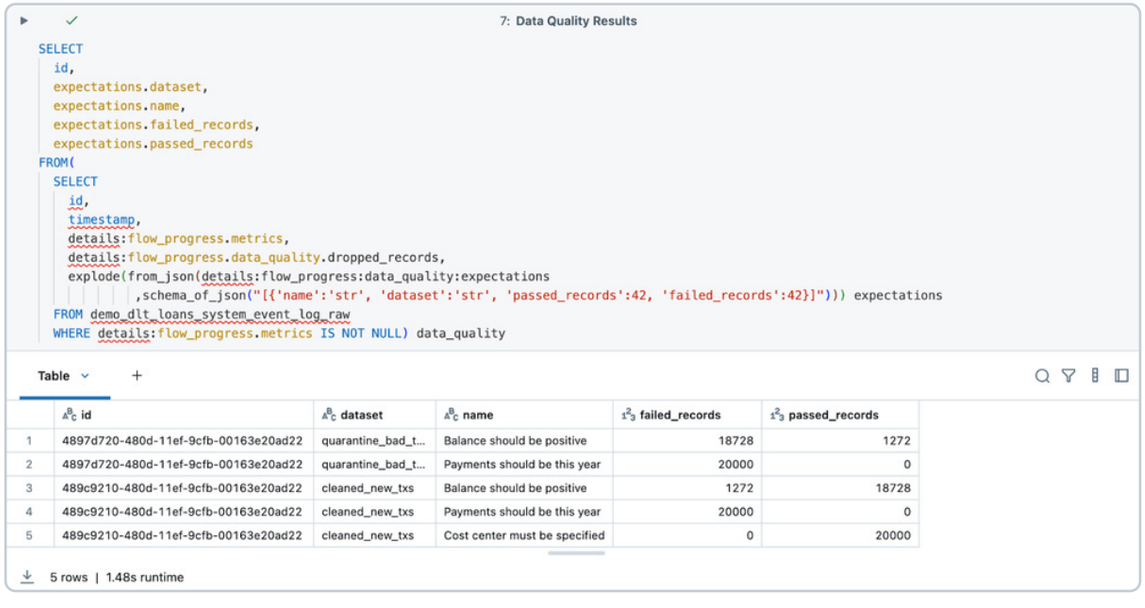
- **Alarm bells go off when there is clearly an issue:** When something is clearly wrong, such as a new client being onboarded without any identifiers, action should be taken automatically to isolate the bad data
- **Potential issues are flagged so workflows can be built around them:** Data that is an outlier when compared to historic values, such as a client being onboarded with a net worth of $10bn, could mean there is an issue, but it's not always cut and dry without more context. In these cases, it can be smarter to flag the value for human review rather than disrupting a business process.

Expectations provide the tooling necessary to apply this nuance (and more) to pipeline monitoring so that issues are handled appropriately. Using standard SQL Boolean statements, expectations apply data quality checks on each record passing through a query.

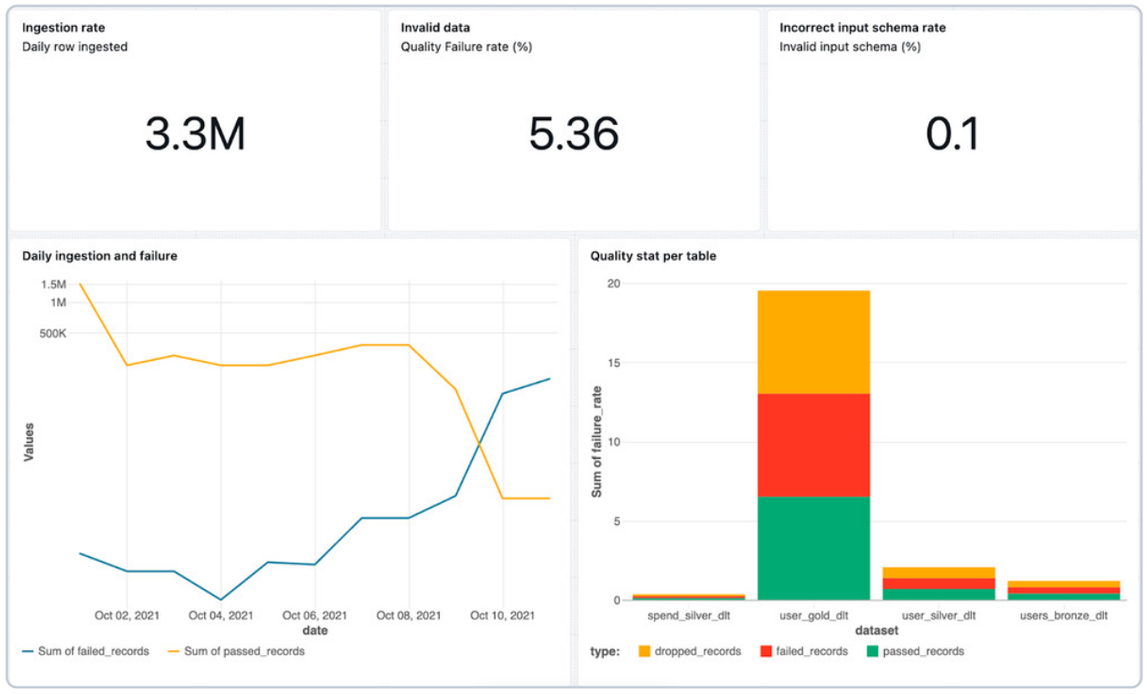
Databricks couples this with tooling to monitor and understand data quality issues across your entire pipeline. The first piece of this tooling is a set of pipeline metrics, available within the Lakeflow UI, that show the number of expectations applied to each table or view and passed/failed records.



Each SDP pipeline saves events and expectations metrics in the storage location defined on the pipeline. The event log table shows the name of each expectation and the number of issues it identified, enabling us to get granular information about changes to data quality over time. Refer to Databricks documentation for additional details about how to access this table and the full list of attributes available.



We can make this information even more broadly available and further simplify and centralize data quality monitoring by creating an AI/BI dashboard using these event logs.



But in order to get these insights about our pipelines, we first need to set up expectations within SDP. Let's look at some specific examples across a variety of data quality issues:

Invalid data

When new data is seen in a pipeline that doesn't follow a common format or structure (e.g., custom country codes), it's often an early indicator of upstream issues that need to be addressed. Diagnosing the root cause can be time-consuming, so as a first order of business, this data shouldn't be allowed to flow downstream. As a simple example, if we acquire lists of prospective clients for an email marketing campaign, we should drop any record without a valid email address.

SQL

```
CREATE STREAMING TABLE email_prospects (

 CONSTRAINT `Emails must contain @` EXPECT (email_address LIKE
'%@%') ON VIOLATION DROP ROW

)

 COMMENT "Livestream of new prospects for email campaign"

AS SELECT * from STREAM(live.new_prospects)
```

Incomplete data

Invalid data by another name, incomplete data is particularly common when upstream systems or transformations have built-in truncation logic. Continuing with our email marketing example, we might find that a value has an "@" but stops there. One of the great things about the expectations feature is that we can simply add this constraint to our previous query.

SQL

```
CREATE STREAMING TABLE email_prospects (

 CONSTRAINT `Emails must contain @` EXPECT (email_address LIKE
'%@%') ON VIOLATION DROP ROW,

 CONSTRAINT `Emails cannot end with @`   EXPECT (SUBSTR(email_
address, -1) != '@') ON VIOLATION DROP ROW
)

 COMMENT "Livestream of new prospects for email campaign"

AS SELECT * REPLACE(rtrim(email_address) AS email_address) from
STREAM(live.new_prospects)
```

Missing data

The difference between missing and nonexistent data can be difficult to decipher without business context. Consulting with downstream consumers is a good idea when implementing most data quality rules, especially when monitoring for missing data. A marketer putting together a personalized campaign will likely say that they can't do anything with prospect data that doesn't contain any contact information. We can use expectations to remove this data and only provide actionable contacts.

SQL

```
CREATE OR REFRESH MATERIALIZED VIEW marketing_prospects (

 CONSTRAINT `Records must contain at least 1 piece of contact
information` EXPECT (email_address IS NOT NULL or full_address IS
NOT NULL or phone_number IS NOT NULL) ON VIOLATION DROP ROW
)

 COMMENT "Marketing prospects for personalized marketing campaign"

AS SELECT * from STREAM(live.marketing_prospects)
```

Keep in mind, when you define a Databricks SQL materialized view directly over a streaming live table that has an EXPECT constraint, the system can't use change data feed to compute "what's new" vs. "what's already been processed." As a result, every REFRESH will re-scan and re-process the entire source, not just the newest data. Alternatively, you can avoid this by applying your EXPECT checks upstream and writing the validated data into a static Delta table.

Inconsistent data

Data consumers are often great at spotting data quality issues because they have an intuition for the range of acceptable values through a mix of business and historical context. Luckily, we can codify some of this context by comparing values to historical statistical ranges and flagging those that fall outside a range. For example, if we capture the net worth of new clients during onboarding, we can set an expectation to flag when we see an outlier value, which could signal that net worth was provided in the wrong currency.

SQL

```
CREATE OR REFRESH MATERIALIZED VIEW net_worth_validation AS

 WITH bounds AS (

  SELECT

  0 as lower_bound,

  avg(net_worth) + 4 * stddev(net_worth) as upper_bound

  FROM historical_stats

  WHERE date >= CURRENT_DATE() - INTERVAL 180 DAYS

)

SELECT

  new_clients.*,

  bounds.*

FROM new_clients

CROSS JOIN bounds;
```

SQL

```
CREATE OR REFRESH MATERIALIZED VIEW validated_net_worth (

 CONSTRAINT 'Net worth must fall within historic range' EXPECT
(amount BETWEEN lower_bound AND upper_bound)

)

AS SELECT * FROM net_worth_validation;
```

Stale data

If we've designed our pipeline to only ingest new, incremental values, then we typically don't expect to see values created that are months or years old. However, that's not always the case. Transactions can be restated, data from legacy systems can be added and one-off files can be added to enrich existing data. Data that seems stale on the surface can actually be legitimate and valuable. That shouldn't stop us from monitoring when this happens so that we can verify that there isn't actually an issue, such as old data being inadvertently replicated.

SQL

```
CREATE STREAMING TABLE client_transactions (

 CONSTRAINT `Warn if transaction is more than 90 days old` EXPECT
(timestamp >= CURRENT_DATE() - INTERVAL 90 DAYS

)

 COMMENT "Cleared and validated client transactions"

AS SELECT * from STREAM(live.client_transactions)
```

Inaccurate data

Perhaps the most challenging data quality issue of all is that inaccurate data can be created for reasons unrelated to any broader issues. A customer service agent may mishear a client on the phone and enter the wrong phone number, a client can enter a typo on a web portal or a client can change their address and forget to notify the bank.

Unfortunately, there's no 'silver bullet' for these types of issues. Instead, we must listen for issues happening downstream and continuously evolve our expectations. We can do this by creating a Delta Table that contains a set of rules used to tag suspect values. The benefit is that these rules can be applied retroactively and across many datasets, allowing us to review the entire corpus as we discover new ways to identify inaccurate data. For example, if we learn that customer support agents have been trained to use a value like "9999999999" when a client isn't willing to provide their income, we can add this to a set of validation rules.

The example below shows what it might look like for a new client questionnaire. First, we define the rules.

SQL

```
CREATE OR REPLACE TABLE

  data_accuracy_rules

AS SELECT

  col1 AS name,

  col2 AS constspark.read.table("data_accuracy_rules").
filter(col("tag") == tag).collect()
```

SQL

```
  return {

      row['name']: row['constraint']

      for row in df

  }


@dlt.table

@dlt.expect_all_or_drop(get_rules('validity'))

def raw_client_questionnaire():

  return (

    spark.read.format('csv').option("header", "true")

      .load('/client_questionnaire_responses/')

  )raint,

  col3 AS tag

FROM (

  VALUES

  ("age_range_valid","age BETWEEN 18 AND 120","validity"),

  ("income_range_valid","annual_income >= 0 AND annual_income <
10000000","validity"),

  ("email_format_valid","RLIKE(email, '^[a-zA-Z0-9._%+-]+@
[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$')","validity")

)
```

Now, we can apply those rules to our questionnaire responses and drop any records that violate them.

PYTHON

```
import dlt

from pyspark.sql.functions import expr, col


def get_rules(tag):

  df = spark.read.table("data_accuracy_rules").filter(col("tag") ==
tag).collect()

  return {

      row['name']: row['constraint']

      for row in df

  }


@Dlt.table

@Dlt.expect_all_or_drop(get_rules('validity'))

def raw_client_questionnaire():

  return (

    spark.read.format('csv').option("header", "true")

      .load('/client_questionnaire_responses/')

  )
```

Since inaccurate data can be so hard to identify through rules alone, there are cases where it may be more appropriate to quarantine records for a data steward to review instead of dropping them entirely. Sticking with our client onboarding example, let's say we want to automatically quarantine records where a welcome email to the client can't be delivered. To do this, we can create separate processing paths for valid and invalid records in downstream operations.

SQL

```
CREATE STREAMING VIEW raw_new_clients AS
  SELECT * FROM STREAM(prod.clients.profile);

CREATE OR REFRESH STREAMING TABLE clients_profile_quarantine(
  CONSTRAINT quarantined_row EXPECT (welcome_email_delivery !=
'false')
)
PARTITIONED BY (is_quarantined)
AS
  SELECT
    *,
    NOT (welcome_email_delivery != 'false') as is_quarantined
  FROM STREAM(raw_new_clients);

CREATE OR REFRESH STREAMING TABLE valid_client_profiles AS
SELECT * FROM clients_profile_quarantine WHERE is_quarantined=FALSE;

CREATE OR REFRESH STREAMING TABLE invalid_client_profiles AS
SELECT * FROM clients_profile_quarantine WHERE is_quarantined=TRUE;
```

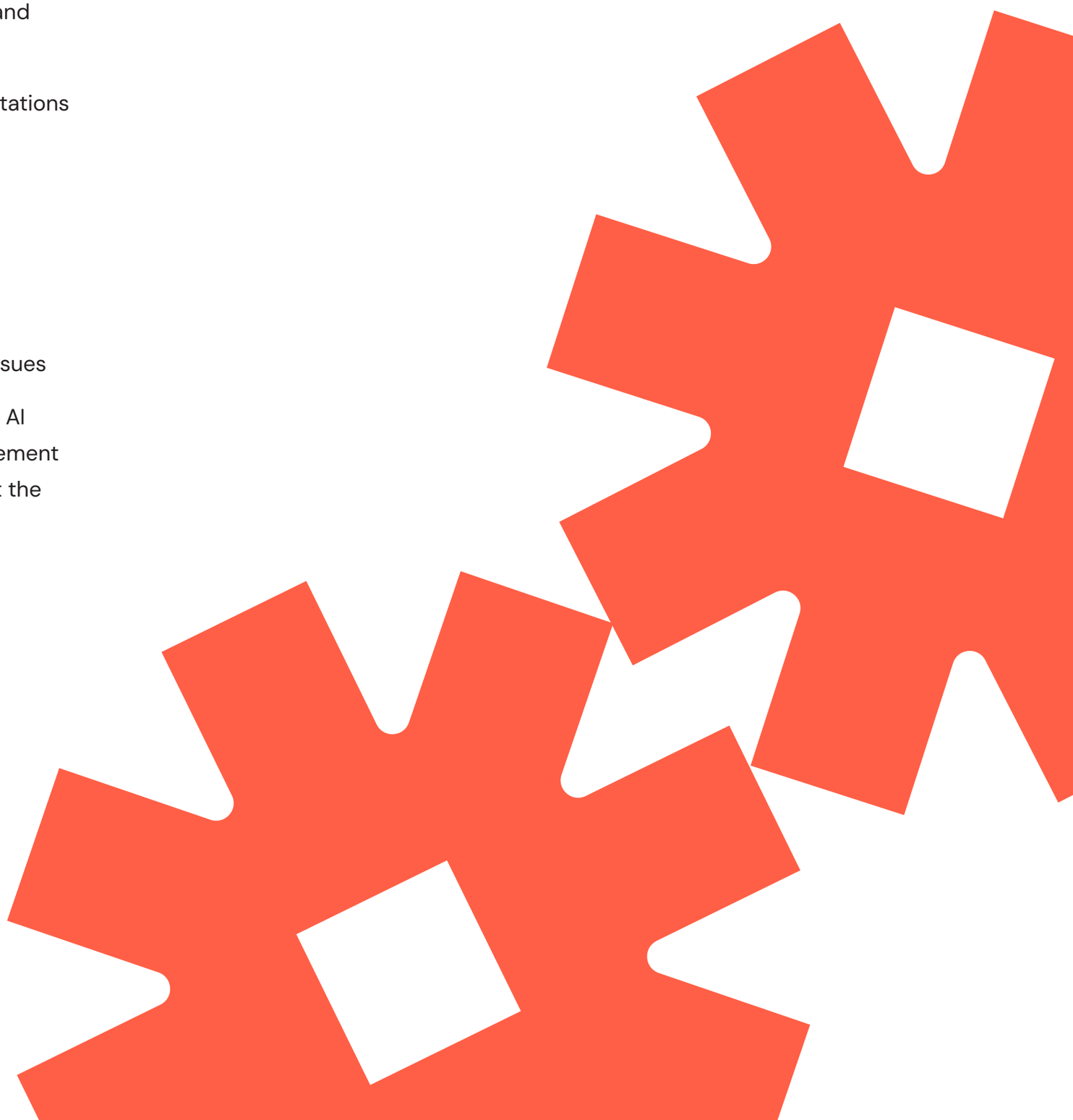
Great data quality is a process, not a destination

It's impossible to solve all data quality issues in one shot, but realizing and maintaining high-quality data doesn't need to be overwhelming.

By implementing a comprehensive data quality framework using expectations in Databricks, you can:

- Establish a scorecard to measure progress over time
- Avoid getting caught off guard by data quality issues
- Identify and address upstream problems that affect downstream processes
- Gradually decrease the frequency and severity of data quality issues

With trust in your data, you'll be able to confidently deliver on data and AI initiatives that drive real business value. By making data quality management a core discipline of your data engineering practice, you can ensure that the benefits are long-lasting.

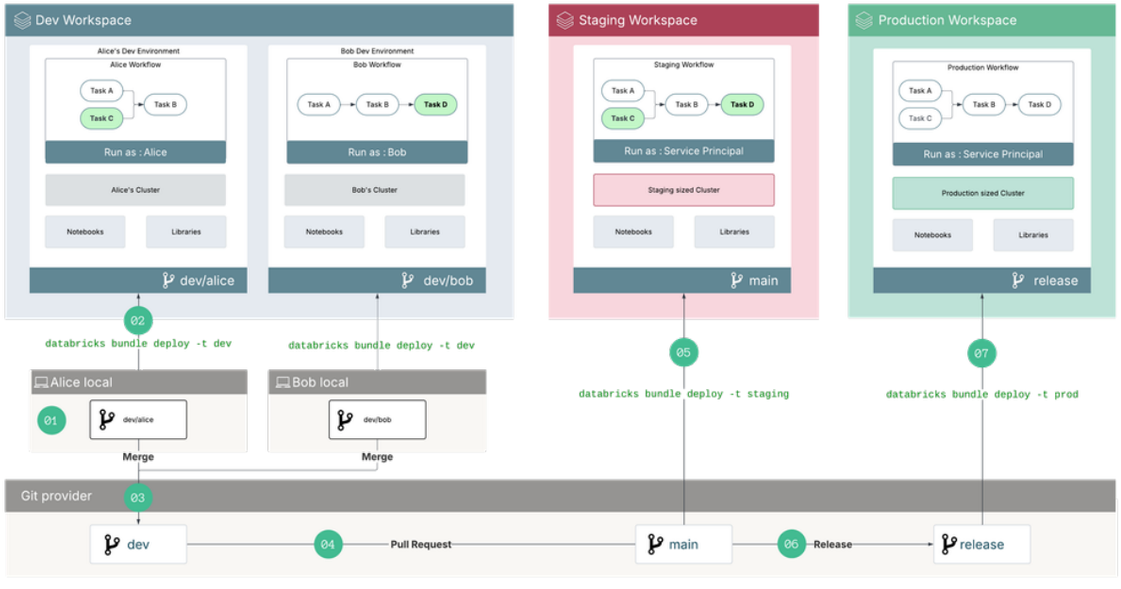


Collaborative Data Engineering Made Simple With Databricks Asset Bundles

By Daniel Taylor and Axel Richier

Data engineering has historically suffered from the inheritance of software engineering best practices. As it emerged in the early 2010s out of the shadow of new data-intensive systems, the industry has always sought to apply the same practices uniformly to a vastly different type of engineering.

Whether that be unit and integration testing, source control, build and release processes and orchestrating through CI/CD (continuous integration and continuous delivery). In the data world, building a lot of these frameworks from scratch can be cumbersome, time-consuming and fragile. Regardless, those best practices exist for a reason and should not be ignored. Instead, they should be improved upon and adapted to suit the workload at hand. This empowers engineers to view these frameworks as necessities in production rather than blockers or nice-to-haves.



Databricks Intelligence Platform: DevX tooling at your fingertips

One of the key strengths of the Databricks Intelligence Platform is the vast amount of DevX tooling at your disposal, helping engineers to get into production and to adhere to these industry-standard best practices. Today, you can use a combination of Terraform for static, core infrastructure assets (more on this later) and Databricks Asset Bundles for application code, data projects and ephemeral resources such as job compute for your application code.

What are Databricks Asset Bundles?

According to the docs, *"Databricks Asset Bundles are a tool to facilitate the adoption of software engineering best practices, including source control, code review, testing and continuous integration and delivery, for your data and AI projects."*

But what does that actually mean in practice? And what tangible, real-world examples can we demonstrate so that we can realize these benefits?

DABs is a tool that comes pre-packaged as part of the Databricks CLI, providing a framework where engineering teams can structure source control repositories so that their application code lives alongside orchestration pipelines. This includes Lakeflow Jobs, Spark Declarative Pipelines, Databricks Apps, training and inference of ML models, etc. These pipelines are defined declaratively using YAML (or JSON, and more recently, Python) syntax and can adhere to any type of configuration your data project requires.

Behind the scenes, Asset Bundles acts as a Databricks-supported wrapper around the Databricks Terraform provider for all resources relating to application code. This lowers the barrier to entry for engineers unfamiliar with Terraform's concepts and syntax, while offering additional flexibility and tooling that caters specifically to that data development lifecycle we've mentioned above. Engineers who are familiar with Terraform will acknowledge its lack of fluidity and ability to quickly iterate when it comes to things like unit and integration testing as well as state management — especially across different, simultaneous development tasks on the same subset of resources.

If you want to understand the core concepts behind Databricks Asset Bundles in much more detail, including the root mapping configuration, resource-specific mappings and CLI commands, we recommend starting with the documentation. We won't be exploring these concepts too deeply in this chapter.

Breaking the Terraform collaboration bottleneck

The true power of DABs becomes clear when we examine how it enables multiple developers working on the same (or different, but within the same repository) data projects simultaneously.

In lower, noncontrolled environments, and if we were to use Terraform as the DevX tool of choice, each individual's deployment would overwrite existing configurations (assuming a single, remote state file was used). This is extremely problematic, not only if individuals are working on the same resource, but also on different resources.

When using Terraform as a DevX tool, you can start to see how collaboration is bottlenecked. As great a tool as it is, it wasn't *really* designed for this.

DABs solves this issue by applying isolation to an individual's state file in lower environments, all while abstracting state file management and maintenance away from the user. What does this actually mean in practice?

- If engineer Alice deployed her modified code repository on her own version-controlled feature branch using Asset Bundles to a development environment, **Asset Bundles would isolate all of her data projects to her state file stored in the workspace file system under her user.**
- If engineer Bob then also happened to simultaneously deploy his own modified code repository from his feature branch using Asset Bundles to the same development environment, **Alice's changes and the data projects in her branch would remain completely unaffected in the workspace** because of the same isolation mechanism.

How does Asset Bundles achieve this behavior?

Asset Bundles have a concept of a top-level configuration; in essence, this is just a YAML file in a project's root (named `databricks.yaml`) that is used to define all of your deployment targets as well as reusable complex and static variables.

The key mapping that we need to consider in our bundles' root configuration in this scenario is `mode`. `mode` can take one of two possible default values, `development` or `production`. For lower environments, we want to specify our targets' deployment mode to `development`. In turn, this will deploy all resources under our bundle and isolate all those resources to the user issuing the deployment from their local machine. Databricks Asset Bundles will also prefix each resource that has been deployed with a unique prefix, so each job, Spark Declarative Pipeline, ML model, etc., will not be overwritten by simultaneous deployments.

Note — *the `production` deployment mode should be used with a service principal for all other environments, such as UAT and production. If `development` or `production` doesn't fit your needs, it is now possible to set up custom modes.*

See it in action: A real-world example

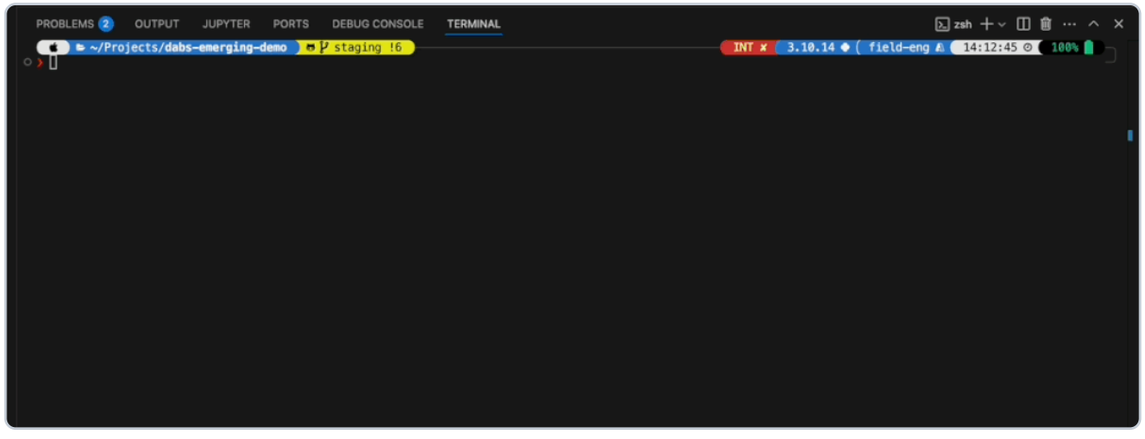
Let's explore seamless collaboration and independent development work with Asset Bundles. As a prerequisite, ensure that you have the Databricks CLI installed and configured for your development environment. For this walkthrough, we're using CLI version 0.277.0.

If you want to follow along, the source code that we'll be utilizing below can be found here.

We're going to imagine the scenario described above: Engineer Alice has been assigned a task that involves a change to a specific Spark Declarative Pipeline based on some upstream changes. In the same stand-up from earlier in the week, engineer Bob has been assigned a similar but distinct task that involves changing a different declarative pipeline based on some downstream reporting changes. Both pipelines are defined in a version-controlled mono-repository.

Both engineers pull the remote changes from the remote repository into their local version and branch off onto their unique feature branches.

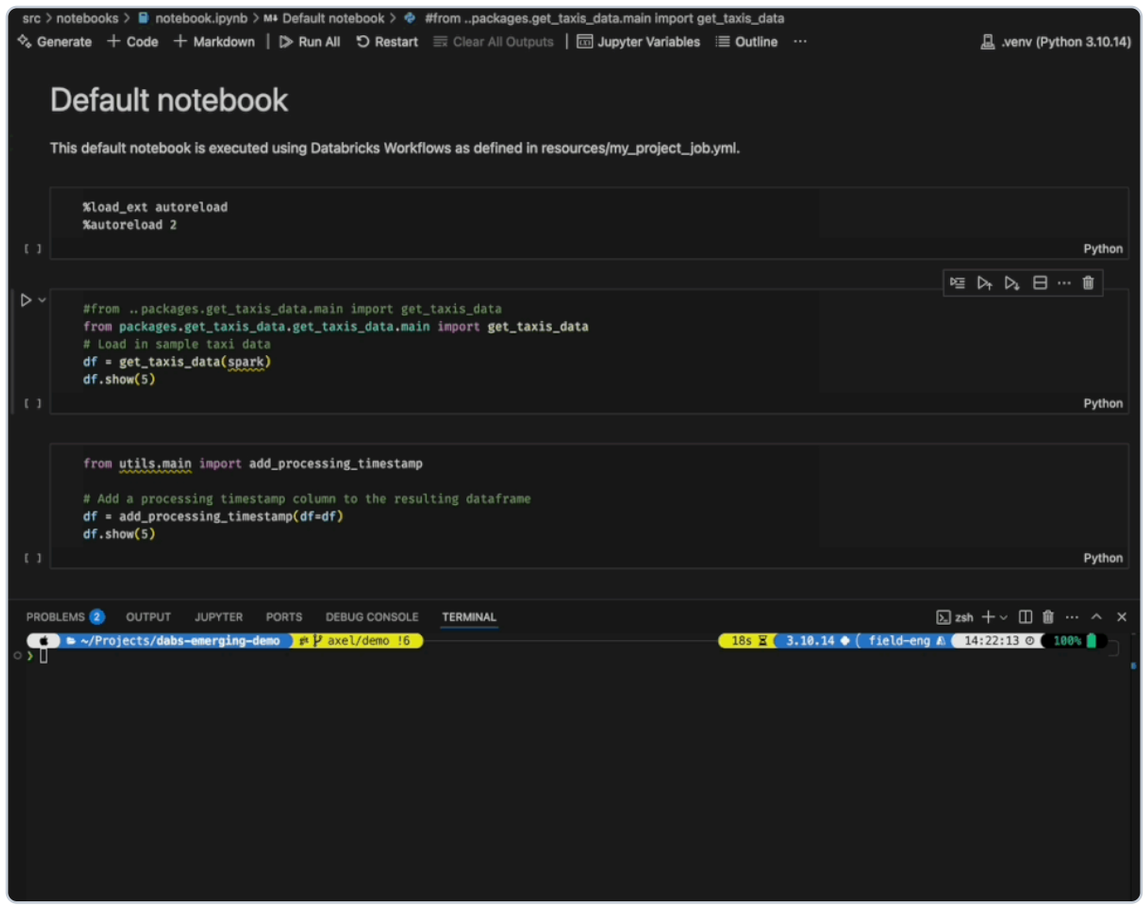
```
git pull origin staging
git checkout -b feature/alice # or feature/bob
```



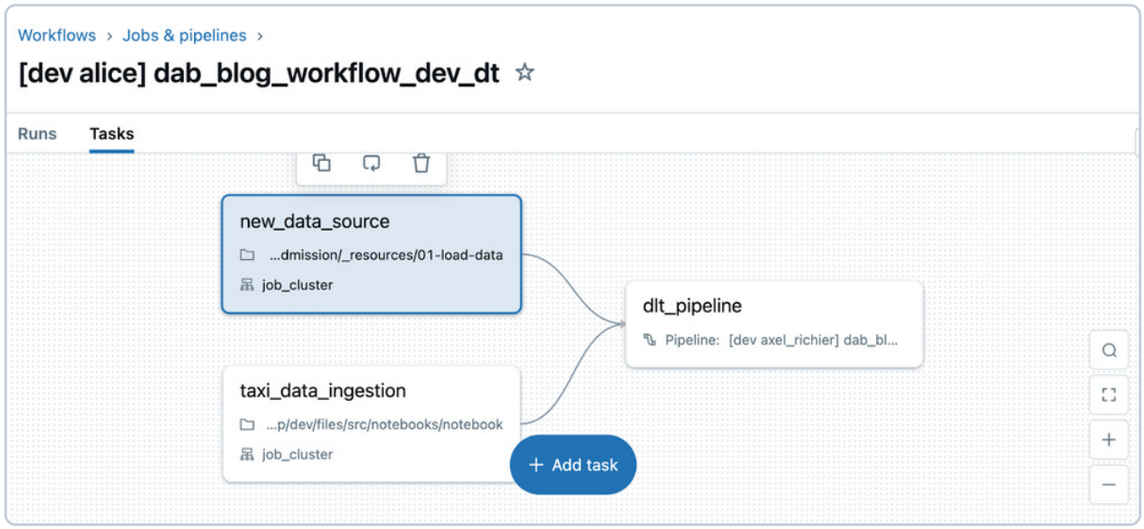
Each engineer then makes local changes to their respective Spark Declarative Pipelines and is now ready to deploy to a lower environment so that they can perform integration tests.

Engineers Alice and Bob both deploy all defined jobs from their respective feature branches to the same lower environment and at the same time.

```
databricks bundle validate -t dev --profile <your_profile>
databricks bundle deploy -t dev --profile <your_profile>
```



We can now see distinct versions of each Spark Declarative Pipeline in our development workspace, specific to the copy in each of our engineers' feature branches. Without Asset Bundles, working on the same assets would lead to risks of overwrite and the impossibility of running developments separately.

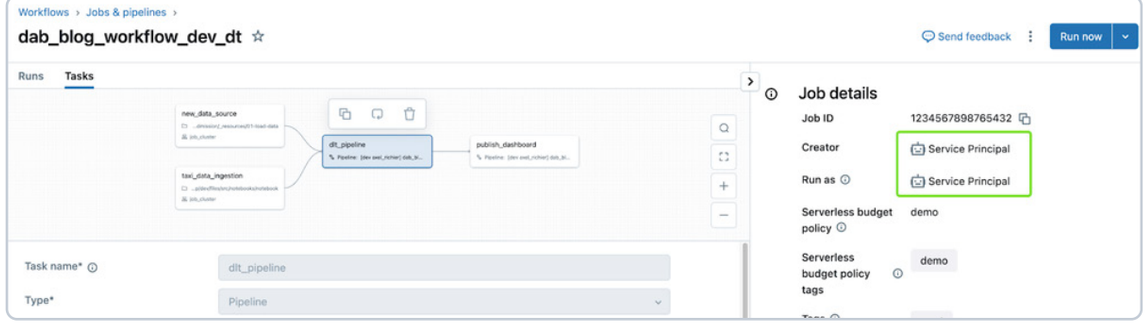


The road to production

Once each developer has tested their isolated changes in their respective branches (e.g., `dev/alice` and `dev/bob`), they can prepare to merge their updates into the main branch. Each developer would create a pull request from their feature branch to the main branch. The PR serves as a proposal to merge their changes, allowing for code reviews and discussions.

Once the PR is approved, the changes from each developer's branch are merged into the main branch, integrating their distinct updates into the shared workflow.

```
databricks bundle deploy -t prod --profile <your_profile>
```



Accelerating time-to-production

This is how Databricks Asset Bundles removes development friction for data engineering teams, enabling iterative deployment while maintaining software engineering best practices. The result is a dramatically faster time-to-production cycle for data products.

By solving the collaboration challenges that typically plague data engineering teams, Databricks Asset Bundles allows you to focus on what matters — delivering valuable data pipelines and insights. Instead of fighting with infrastructure and deployment conflicts, you can prioritize delivering business value.

A Practical Guide to Serverless Migrations

By Diego Gomez

Many customers have been eager to adopt serverless compute on Databricks, attracted by the promise of eliminating infrastructure management, gaining elastic scalability and simplifying the user experience for their teams. Unlocking the full value of serverless involves rethinking environment configuration, cost governance and dependency management to ensure a smooth transition.

In this chapter, we'll walk through a practical playbook for serverless migrations so you can approach them with the right tools and considerations.

Before getting started...

It's worth taking a moment to understand how serverless compute differs from classic compute — because many of the operational and financial changes stem from that.

With classic compute, workloads run on clusters directly provisioned within the customer's own cloud account. This means that the customer is responsible for choosing cluster size, runtime version and the specific instance types to spin up. As the job runs, Databricks charges DBUs for managing and orchestrating the workload, while the cloud provider bills separately for the underlying infrastructure being used.

Serverless flips this paradigm. Compute resources run fully within Databricks-managed cloud accounts. Databricks takes care of provisioning, scaling and managing the underlying infrastructure behind the scenes — from the user's perspective, it just works.

This shift introduces a few important differences worth highlighting, especially around how compute is billed and governed.

- **You only pay for the time your workload is actually running**

There's no separate charge for idle clusters or VMs sitting idle between jobs. Billing starts when the workload starts processing and stops when it completes. ***Note. Pricing details may vary across features. For a full breakdown of pricing rates, refer to the official pricing page.***

- **No virtual machine costs to manage**

With serverless, the $DBU rate includes both the compute resources and the operational costs. While it may appear higher than classic compute, it's fully inclusive.

To put it into perspective:

If a job runs on classic compute for five minutes of processing but requires several minutes of startup time and stays up for idle time after completion, you're billed for all of that: startup, processing and idle — both in $DBU and VM charges.

With serverless, you'd only be billed for the five minutes of actual processing time. Startup happens faster (since Databricks-managed warm capacity pools), and you're not charged for any idle or startup time.

This difference is what enabled many customers to see significant cost efficiencies — *if* the proper controls are in place. This brings us to why cost governance is such an important part of a successful serverless adoption.

Laying the groundwork: Cost controls

Before even thinking about moving into serverless, it's important to understand how to govern its costs — so you don't end up surprised by unexpected charges later on.

One key difference with serverless compute is that once it's enabled in a workspace, it's immediately available to all users by default. Since serverless operates within Databricks-managed cloud accounts, any existing compute policies that govern classic compute — such as cluster policies — don't automatically apply to serverless workloads.

While this flexibility allows teams to get up and running quickly, it also means that organizations need to establish clear cost controls upfront to ensure that serverless consumption aligns with their financial and operational expectations.

Let's make this more concrete by walking through an example.

In this case, serverless compute is being enabled in a development workspace used by three groups: **data engineers, data scientists and analysts**. Each persona has different workload patterns and resource needs:

- Data engineers primarily run batch pipelines and ETL workloads
- Data scientists have started experimenting with deep learning models using newly released GPU serverless instances
- Analysts mostly work with ad hoc queries and dashboards using SQL serverless

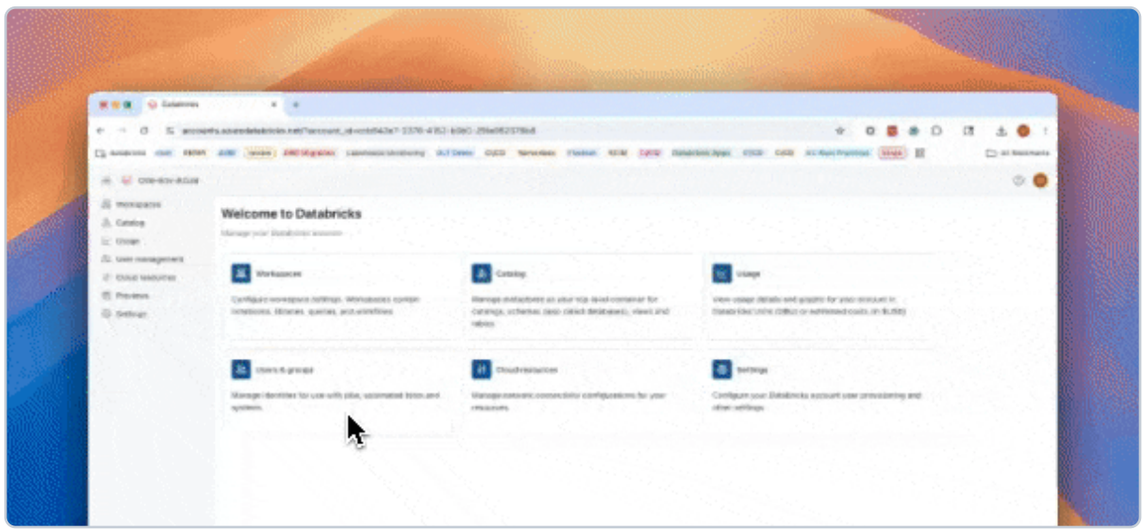
Without proper guardrails, it would be easy for serverless consumption to scale unpredictably — particularly as GPU serverless workloads can generate larger costs if left unchecked.

Budgets

The Budgets feature allows administrators to define spending thresholds and monitor usage across workspaces. In this case, we can configure two separate budgets within the development workspace:

- **One budget for data scientists**, who will be running GPU serverless workloads, with a higher threshold to give them room to experiment. It will apply to any policies tagged with identifiers related to the data science team — for example, tags containing **data-science** or **ds-team**.
- **A second, smaller budget for data engineers and analysts**, whose serverless workloads tend to be more predictable and stable. This budget will apply to jobs or compute resources tagged with identifiers like **data-engineer** or **analytics**.

Budgets give us visibility into spend as it grows and allow us to configure alerts when usage approaches defined thresholds. These budgets serve as proactive guardrails, helping avoid surprises and giving administrators early signals before costs escalate unexpectedly. Budgets can be configured at both workspace and account levels, allowing flexibility in how spend is tracked and controlled across environments.



Serverless budget policies

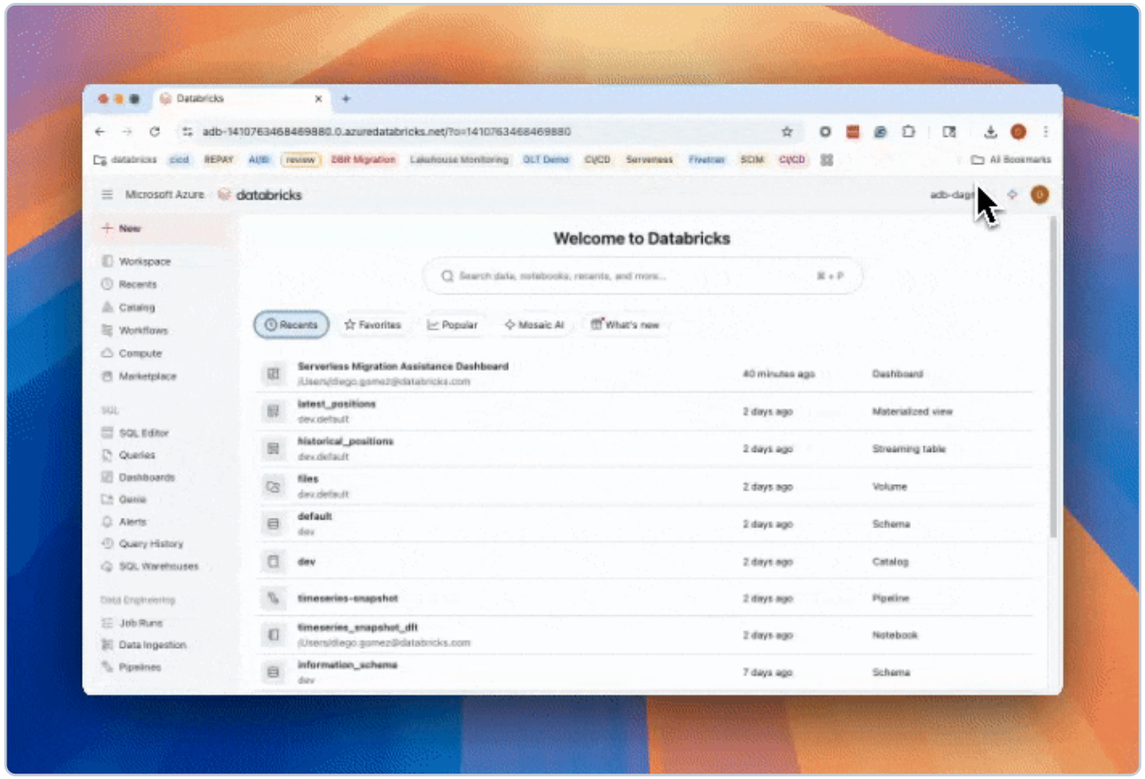
With budgets in place, the next step is to enforce budget policies for serverless compute to automatically apply cost attribution tags and guard spending across different teams.

Serverless budget policies allow administrators to define one or more tag key-value pairs that get automatically applied to any serverless compute activity initiated by users or jobs assigned to the policy. As users launch serverless compute, these tags are automatically attached behind the scenes, ensuring accurate cost attribution and reliable billing records for all serverless workloads.

The tags then surface directly in the system billing tables (for example, system.billing.usage → custom_tags), making it easy to track usage and attribute costs to specific teams, personas or business units as needed.

In our development workspace scenario, we'll create budget policies aligned to each persona by applying tags:

- **team=data-science** for data scientists
- **team=data-engineer** for data engineers
- **team=analyst** for analysts



Monitoring serverless spend

Once the budgets and serverless budget policies are in place, the next step is to establish ongoing monitoring. This ensures that administrators have full visibility into serverless consumption as usage scales — and allows organizations to react quickly if spending starts to drift outside expected patterns.

The **system.billing.usage** system table provides detailed billing records across all compute types, which allows for more advanced usage queries, especially when paired with **system.billing.list_prices**. With this information, teams can build custom dashboards, configure alerts and create periodic reporting pipelines to track serverless usage in near real-time — providing FinOps teams with the level of transparency required to confidently scale serverless adoption.

For example, the following query calculates the total list-price cost for serverless compute by workspace over the last 30 days:

SQL

```
SELECT
   t1.workspace_id,
   SUM(t1.usage_quantity * list_prices.pricing.default) AS list_cost
FROM system.billing.usage t1
INNER JOIN system.billing.list_prices
   ON t1.cloud = list_prices.cloud
   AND t1.sku_name = list_prices.sku_name
   AND t1.usage_start_time >= list_prices.price_start_time
   AND (t1.usage_end_time <= list_prices.price_end_time OR list_
prices.price_end_time IS NULL)
WHERE
   t1.sku_name LIKE '%SERVERLESS%'
   AND billing_origin_product IN ('JOBS', 'INTERACTIVE')
   AND t1.usage_date >= CURRENT_DATE() - INTERVAL 30 DAYS
GROUP BY
   t1.workspace_id
HAVING
   list_cost > {budget}
```

By scheduling this query to run as an alert, administrators can receive proactive notifications as soon as a workspace approaches or exceeds its assigned budget.

Actually migrating a job into serverless

With cost controls and monitoring in place, we can now start migrating workloads into serverless compute. The actual migration process is fairly straightforward, but there are a few important considerations to be aware of when switching compute, managing dependencies and validating performance.

Switching compute to serverless

The first step is updating the job to use serverless compute instead of classic clusters. For jobs that are already running in production, this can typically be done directly from the job configuration UI or API:

Tip: When testing the migration, it's often a good idea to create a duplicate of the job to avoid any unintended impact on production workloads.

At this point, you've officially moved the workload onto serverless — just like that. The only catch is that serverless keeps things light by default, so you'll need to make sure your job has all the libraries it needs before hitting run.

Understanding serverless environment versions

One key difference when running on serverless is that you no longer control the exact runtime image underneath — Databricks takes care of that for you. Serverless environments are versioned, with Databricks periodically rolling out updates to maintain compatibility, security and stability.

You can find the full list of available serverless environment versions in the serverless release notes. Each version includes a set of pre-installed system libraries, drivers and configurations.

When getting started, it's generally good practice to select the latest available serverless environment version to take advantage of the most recent fixes, performance improvements and package updates.

Configuring library dependencies

Since serverless keeps the environment minimal by default, it's important to define any additional libraries your workload requires. There are a few different ways to manage dependencies depending on your use case:

- **Workspace-level default dependencies**

You can configure default libraries that apply to all serverless compute in the workspace. This works well for common packages used across multiple teams. Details on how to set this up can be found in the documentation.

- **Job task-level dependencies (recommended for production)**

For jobs, you can define dependencies directly at the job task level. This provides finer control over which packages are installed for each workload, reduces the risk of package conflicts and keeps things more predictable as multiple teams share the same workspace.

- **Notebook-scoped dependencies (useful for development)**

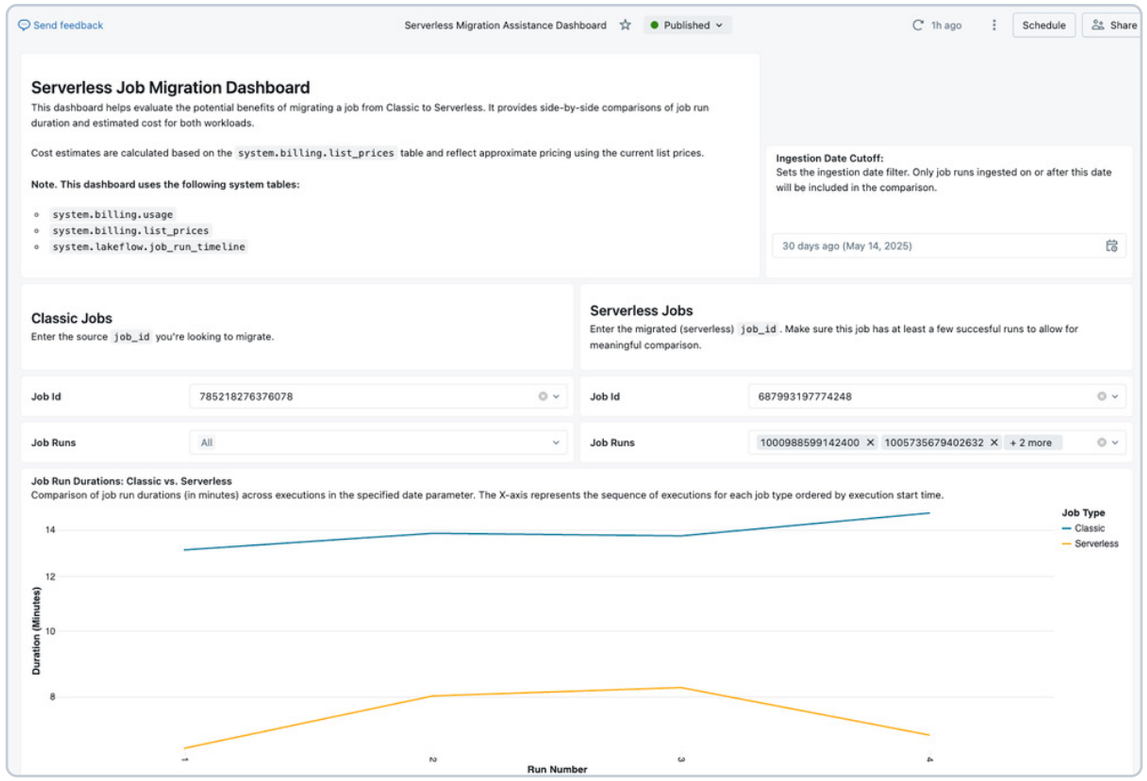
For notebooks running on serverless compute, Databricks provides an **Environment panel** directly inside the notebook UI. This allows you to configure dependencies (including uploading Python wheel files), memory settings, budget policies and the serverless environment version — all from within the notebook itself. Find more here.

These settings will only apply when the notebook is attached to serverless compute.

Run, monitor and compare

With your job now fully serverless-ready, you're ready to hit run. This is where all the cost controls, budget policies and monitoring we set up earlier start to pay off.

We've created a dashboard that uses system tables to compare classic jobs against serverless. Just grab the **job_ids**, and you'll be able to view duration and costs per run to better understand how the migration is performing.



Resource: You can find the code and dashboard in this Git repository.

With these tools in place, you'll be able to experiment safely, validate serverless performance for your workloads and decide which jobs are worth fully moving over.

Conclusion

And with that, we hope this guide helps you feel better equipped to tackle serverless migrations on your own. You've got the controls, the monitoring and the migration process in place — now you're ready to put serverless to work.

03 Case Studies

CUSTOMER STORY

iFood: Transforming Data to Elevate Food Deliveries and Experiences

INDUSTRY Retail and Consumer Goods

CLOUD AWS

67%
Reduction in processing and storage costs

70%
Reduction in maintenance efforts

30%
Less coding time

> The shift from frequent errors to near-zero issues moving to Spark Declarative Pipelines has not only improved operational efficiency but also freed up our team to focus on strategic initiatives instead of firefighting. We've reduced data pipeline maintenance efforts by about 70% by consolidating all pipelines to Spark Declarative Pipelines.
>
> **— Thiago Julião, Data Architecture Specialist, iFood**

---

iFood, a leading food delivery platform in Brazil, operates at a staggering scale: 300,000 drivers, 55 million users and 350,000 restaurant partners. This ecosystem generates an immense volume of streaming data daily, spanning customer orders, delivery logistics and app interactions. With a wide range of products and platforms collecting data across the company, iFood processes billions of data records daily for analysis and modeling to support strategic business decisions. With ambitions to expand and enhance their offerings, iFood needed a more reliable, scalable and efficient data platform to meet the growing demand for real-time insights and innovation.

Data silos and complex architecture stifled agility and accuracy

Prior to Databricks, iFood grappled with a fragmented data architecture that hindered their ability to scale effectively. The company relied on a complex infrastructure with multiple systems to manage vast amounts of users' journey data — comprising billions of records from various sources, including the company's order management system, consumer app and driver app. As the ecosystem grew and new business models emerged, the volume and variety of data grew, too, further complicating data processing. With these datasets spread across disparate systems, consolidating and accessing critical information quickly and reliably became a considerable challenge.

This fragmentation led to significant operational inefficiencies, as iFood's teams struggled to keep track of scattered event data. With the data spread across numerous systems and lacking proper governance, troubleshooting and ensuring data accuracy became a time-consuming and error-prone process. iFood's data engineering team faced immense challenges, including the significant engineering effort to manually code, optimize and maintain complex workflows for data processing. The company's data engineering team previously spent countless hours troubleshooting errors and coordinating with multiple teams to implement even minor changes.

This constant firefighting drained resources, hindered innovation and left engineers with little time for strategic work. The situation was further complicated by iFood's explosive data growth. What was once a legacy architecture capable of handling 100 million events per day was now overwhelmed by a surge of 8 billion–10 billion events daily. As the company began training real-time models to analyze app user journeys and surface actionable insights, low latency at scale became a critical requirement.

Streamlining pipelines, reducing maintenance and enabling real-time insights

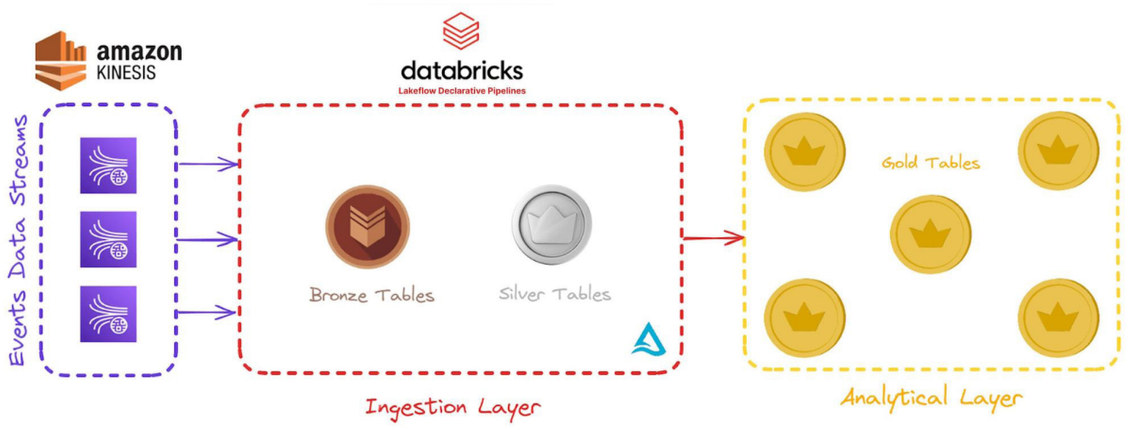
Spark Declarative Pipelines proved to be a game changer for iFood. Spark Declarative Pipelines enabled iFood to shift to a declarative approach for pipeline development. Engineers could now describe their desired transformations in simple code, allowing Spark Declarative Pipelines to automatically handle the operational complexity behind these pipelines, including execution, scaling and monitoring. "So far we've reduced coding time by approximately 30% using the declarative approach, allowing us to build pipelines significantly faster than before," said Thiago Julião, Data Architecture Specialist at iFood. This transition also simplified and consolidated iFood's data architecture, reducing the number of tables from nearly 4,000 to just 100. This reduction also made governance more manageable and laid the foundation for improved data quality.

Before Spark Declarative Pipelines, iFood's pipelines were hindered by out-of-memory errors during high-volume event ingestions, leading to frequent driver shutdowns and operational disruptions. These recurring failures required constant attention from the data engineering team. However, since implementing Spark Declarative Pipelines in production, the transformation has been remarkable.

"With Spark Declarative Pipelines, we gained greater ease in tracking the user's journey in the application while ensuring high performance in data usage by consumer teams — this was a game changer in our process," said Maristela Albuquerque, Data Manager at iFood.

iFood's unified data architecture

iFood's technical architecture is now designed to process streaming data at an immense scale, ensuring efficiency, governance and scalability. Here's a detailed breakdown of the architecture and how its components work together to handle 10 billion daily events while delivering real-time insights.



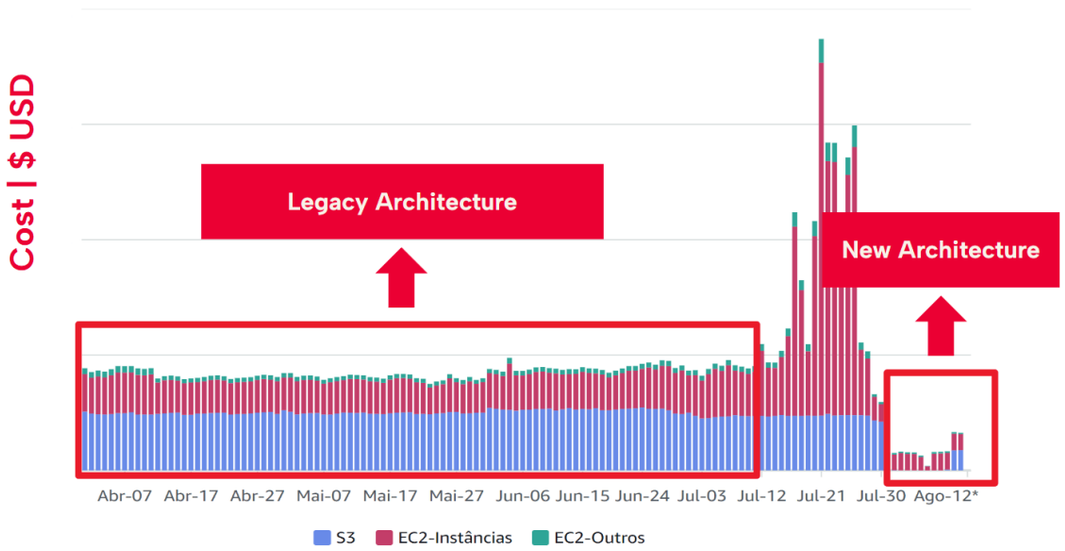
The data pipeline starts with real-time ingestion of events from iFood's ecosystem, including the consumer app, delivery driver app and partner portal. These events flow through Amazon Kinesis queues, where approximately 10 billion records are ingested daily.

The ingestion pipeline, powered by Spark Declarative Pipelines, enables real-time data ingestion with scalability, resilience and quality. By adopting Spark Declarative Pipelines, iFood reduced ingestion latency from hours to seconds — ensuring low-latency data availability critical for real-time analytics, such as training models and extracting insights during the user journey without delays. "Pipelines now run error-free, delivering reliable performance even under the heaviest workloads," said Julião. "The shift from frequent errors to near-zero issues moving to Spark Declarative Pipelines has not only improved operational efficiency but also freed up our team to focus on strategic initiatives instead of firefighting. We've reduced data pipeline maintenance efforts by about 70% by consolidating all pipelines to Spark Declarative Pipelines."

iFood's architecture leverages a structured medallion approach to manage massive data volumes effectively. The Bronze layer consolidates data from various platforms into a single table per product, using a predefined schema partitioned by processing date. Acting as a staging zone, it ensures extended data retention compared to the message queues.

In the Silver layer, iFood applies Spark Declarative Pipelines expectations and quality rules to validate the data. By replacing traditional partitioning with Liquid Clustering, all events for a product are consolidated into a single table, significantly improving performance and usability. This optimization allows iFood to manage massive datasets — such as their largest table, which spans 210TB and 800 billion records — while maintaining high data quality and governance. "Previously, managing two separate environments required constant communication across teams, making even small changes challenging. Now, with everything under our control, the process is streamlined and more efficient," said Gabriel Campos, Head of Data and AI at iFood.

The new architecture has enabled iFood to centralize data governance and enhance data quality without compromising performance or usability. Additionally, it has significantly reduced processing and storage costs, achieving a 67% cost reduction — cutting expenses from tens of thousands to just thousands of dollars per month.



With Spark Declarative Pipelines automating and streamlining data pipeline management, iFood's business analysts from the growth and product teams can now effortlessly access data from the Silver layer to create analytical Gold tables, empowering them to generate business-critical insights with ease. This supports user journey analysis and A/B testing on consumer behavior at various stages of their journey, enabling the creation of data-driven strategies to enhance the customer experience across iFood's ecosystem. For example, the driver app provides critical insights to the logistics team, helping them understand how drivers interact with the app and optimize its usability. These insights allow iFood to fine-tune both consumer-facing and operational processes, ensuring a seamless experience for customers and efficiency for drivers.

iFood plans to further enhance their Databricks implementation by leveraging DABs for streamlined development and serverless computing for greater flexibility. Upcoming initiatives include implementing column masking for sensitive data in the consumption layer and optimizing table performance with variant type handling for complex data structures like structs and maps.

iFood's transformation to a modern, unified data architecture has redefined how the company processes and leverages their vast data ecosystem. And by adopting Spark Declarative Pipelines, iFood streamlined their operations, eliminated inefficiencies and established a foundation for real-time insights and enhanced governance. This shift has not only improved the reliability and agility of the company's data pipelines but also freed up their teams to focus on innovation and delivering value to the business. With a scalable, efficient and future-ready architecture, iFood is now equipped to respond to the demands of a dynamic market while continuing to elevate the customer experience.

NAVY FEDERAL Credit Union

CUSTOMER STORY

NFCU: Using Real-Time Data to Trailblaze Member Personalization

INDUSTRY Financial Services

CLOUD Azure

Billions
of application events per month processed by Spark Declarative Pipelines

9 billion
events streamed continuously for 9 months, 24/7, with ~100% zero maintenance

6-week
time to market for a new real-time omnichannel application

databricks

> The simplicity of the Spark Declarative Pipelines programming model combined with its service capabilities resulted in an incredibly fast turnaround time. It allowed us to get a whole new type of workload to production in record time with good quality.
>
> **— Jian (Miracle) Zhou, Senior Engineering Manager, Navy Federal Credit Union**

Navy Federal Credit Union was founded during the Great Depression by seven U.S. Navy employees who wanted to help themselves and their coworkers reach their financial goals. Today, Navy Federal Credit Union is the largest credit union in the world, serving 13 million member-owners. Navy Federal's priority is to provide a personalized, omnichannel experience to their members. But to understand their members better, they needed to ingest and analyze online telemetry data in real time. To accomplish that, Navy Federal turned to Spark Declarative Pipelines, Databricks SQL and Microsoft Power BI.

Near real-time data introduces new insights

In June 2023, Navy Federal was about to start the process of migrating millions of users to the latest version of their online banking platform. The company planned to complete this migration in a controlled manner with small user groups, gradually expanding to all their members over several months.

To make informed decisions, Navy Federal needed to closely track how members interacted with their product. To accomplish that, Navy Federal engineers created data pipelines to their data lake, and data analysts built dashboards and reports to visualize KPIs. The pipelines ran daily, so the information from the previous day would be available to leadership at the beginning of a new day. "The pipeline ran perfectly, but in today's world, 24 hours is a long turnaround time," Jian (Miracle) Zhou, senior engineering manager at Navy Federal Credit Union, said. "As we were moving closer to the first wave of user migration, we decided to enhance our data solution to introduce new insights in near real time."

To accomplish that, the organization needed to create a near real-time dashboard with two KPIs: the number of unique members who were able to log into the new channel, and the number of unique sessions the channel served. Additionally, they needed to filter KPIs both by time and by member attribute, which was the indicator of which migration wave they belonged to. The expectation was that the dashboard would be updated continuously throughout the day with a latency of no more than 10 minutes.

Zhou faced two primary challenges: time and scale. The organization only had about six weeks to build a solution from scratch and get it into production. "This was actually the first time my team tried to build a real-time solution in Azure Cloud, and we had to figure out everything in six weeks," he said. "And scale-wise, we're talking about billions of application events generated from millions of monthly active users, so the solution had to scale as the migration proceeded."

Databricks Spark Declarative Pipelines helps create reliable, performant data pipeline

Navy Federal's data delivery solutions follow a straightforward path. One side is for engineers and is all about data ingestion and curation. For every data source, they build a data pipeline to ingest data into their data lake, transform it and make it ready to be consumed by customers. This part of the equation is very dynamic because of the diversity of its data sources. The other side is for data analysts and is all about data serving and visualization. Navy Federal uses Azure Data Lake Storage as their storage layer, Databricks SQL as their analytical query engine and Power BI for data visualization. Together, they form a simple but effective foundation for data analytics.

The source of the data Navy Federal needed to work with was the event telemetry generated in their online banking application.

Navy Federal Credit Union faced challenges using Azure Application Insights for dashboards due to the lack of access to member attribute data. To address this, they streamed telemetry from Application Insights to their data lake using Azure Event Hubs, which supported the Kafka protocol for low-latency, high-reliability data consumption without requiring coding.

To process the streamed data, Navy Federal used Databricks Spark Declarative Pipelines. Zhou and his team used Spark Declarative Pipelines to create a simple data pipeline by writing three query functions and two change data capture functions. The first function they wrote connects to the event hubs, applies some transformation and returns a data frame. The second query function takes complex JSON documents and flattens them into individual application event records. They then used Spark Declarative Pipelines's APPLY CHANGES function to remove duplicates. In the third query function, they filtered the data down to just the login events and then selected only the columns relevant to login.

Last, they used the APPLY CHANGES function again to ensure there aren't duplicates in the login event table. Finally, they helped Spark Declarative Pipelines understand which data should be persisted and which can stay in memory (but not be accessible to the outside world) when a pipeline runs. The team then used the Expectations feature to set data-quality expectations and monitor the health of that data. "That's all — three query functions and two CDC calls to complete all four steps. Simple as that," Zhou said. "For me, that means low-code complexity and short development time."

Critically, the pipeline Zhou and his team created scales aggressively, which was important as the omnichannel user migration progressed. "We started with a very small wave and kept increasing the migration size wave after wave. We were anticipating that would increase the demand for compute resource, but we were not sure exactly how much was needed for every wave. So, when we deployed this data pipeline to production, we turned on a feature called Enhanced Autoscaling," Zhou explained. "The processing volume went up quickly, from just a few hundred users and a few thousand sessions every day to millions of users and billions of application events per month. This pipeline automatically scaled the resources depending on the data volume. Our production support engineers never had to intervene. They practically forgot about it."The pipeline was also reliable and fault-tolerant. It ran 24/7 continuous streaming for nine months and successfully processed about nine billion application events with almost zero maintenance. "On some days there were hiccups that caused temporary interruptions," Zhou said. "Those hiccups were the natural result of being in the cloud environment. The great thing is that the pipeline showed great resilience. Without exception it automatically recovered from those transient errors."

The pipeline was also performant. Even as event volume data skyrocketed, the pipeline kept up. "The streaming processing speed and the query response time were to our satisfaction. Part of the reason was that Spark Declarative Pipelines automatically optimizes, compacts and vacuums the tables it manages," Zhou said. "This effectively eliminates or reduces the small file problem and removes files no longer needed from our storage account, which leads to better query performance."

The last mile of the journey was the delivery of the dashboard. Databricks SQL allowed them to curate data in Power BI with speed and scale. "Our analysts connect Power BI to the data in our data lake through the Databricks SQL endpoint, build Power BI semantic models and create insight for visualizations," Zhou explained. "Our dashboard directly queries Databricks. When the visuals are rendered, it brings back the latest updates to users in real time."

Spark Declarative Pipelines get a new type of workload to production in record time

Using Spark Declarative Pipelines, Navy Federal was able to complete a proof of concept in a week; develop, test and figure out a CI/CD process in three weeks; deploy the pipeline to production just before the start date of the first wave migration; and release the dashboard just a few days later.

"The simplicity of the Spark Declarative Pipelines programming model combined with its service capabilities resulted in an incredibly fast turnaround time," Zhou said. "It truly allowed us to get a whole new type of workload to production in a record time with good quality."

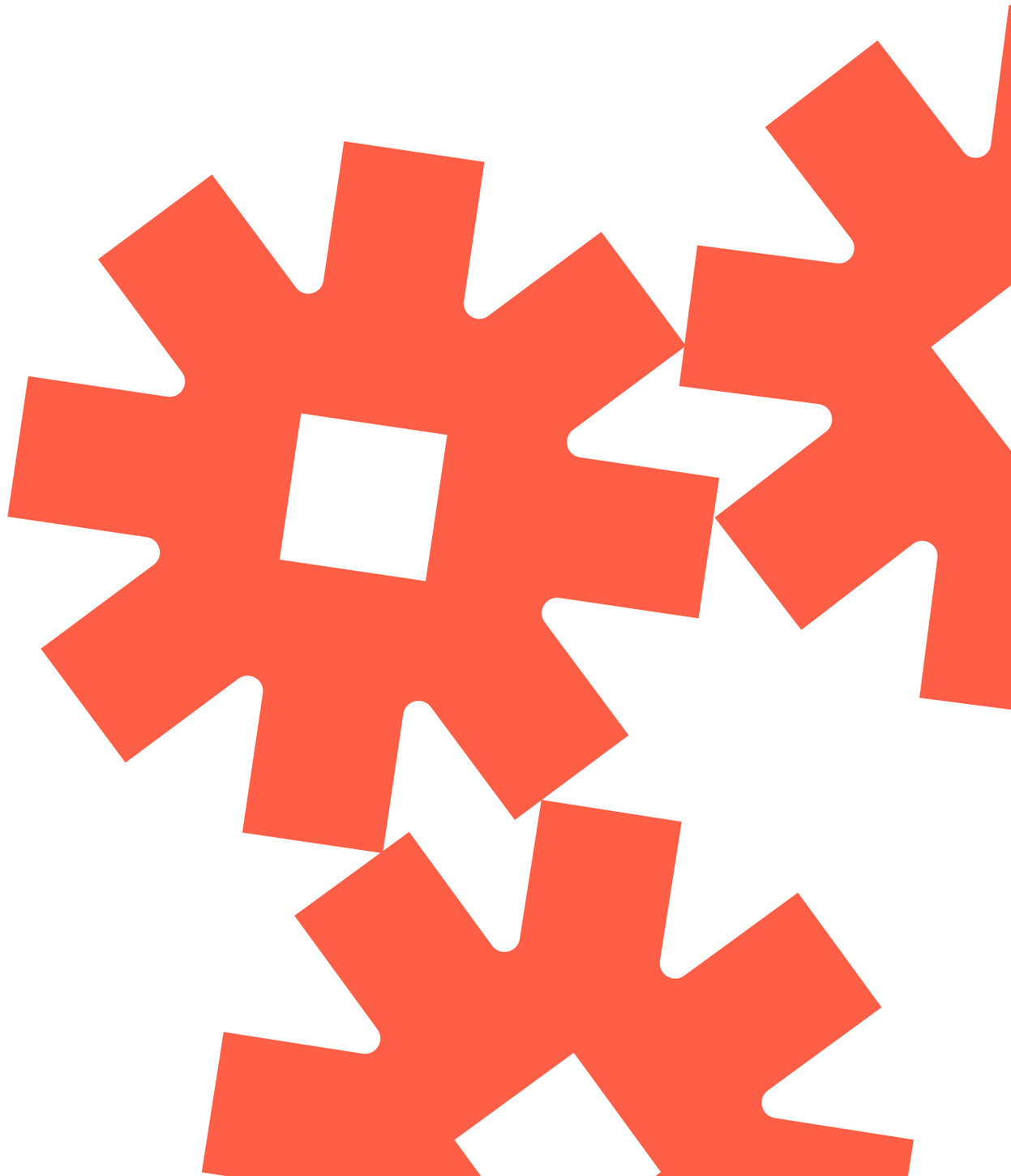
The release of the real-time user activity monitoring dashboard combined with historical views revealed long-term trends, which received overwhelmingly positive feedback from Navy Federal's senior leadership and stakeholders.

"Spark Declarative Pipelines hides the complexity of modern data engineering under its simple, intuitive, declarative programming model," Zhou said. "As an engineering manager, I love the fact that my engineers can focus on what matters the most to the business. Spark Declarative Pipelines can take care of what is underneath the surface and ensure not only that our pipeline was built and deployed in time, but also held up beautifully in the long run."

After the first version of the new pipeline went to production, Zhou and his team continued to innovate, using Spark Declarative Pipelines to develop and deploy a homegrown metadata-driven framework.

Zhou said the three things he likes best about Spark Declarative Pipelines are:

- The simple declarative programming model that accelerated speed to market
- Built-in scalability, optimization and reliability — which simplified operations
- Accessibility through the API, which enabled space for engineering creativity and scalability



CUSTOMER STORY

Cincinnati Reds: Winning On and Off the Field with Cutting-Edge Baseball Analytics

INDUSTRY Media & Entertainment

CLOUD Azure

3–5x
Improvement in latency with serverless Databricks Lakeflow Jobs

1,000s
Of hours of compute time saved daily

83%
Decrease in pipeline runtime

databricks

> With serverless Databricks Lakeflow Jobs, we've achieved a 3–5x improvement in latency. What used to take 10 minutes now takes just 2–3 minutes, significantly reducing processing times. This has enabled us to deliver faster feedback loops for players and coaches, ensuring they get the insights they need in near real time to make actionable decisions.
>
> **— Bryce Dugar, Data Engineering Manager, Cincinnati Reds**

In the high-stakes world of major league baseball, every decision and action can be the difference between winning and losing. Understanding how real-time data can provide a competitive advantage, the Cincinnati Reds looked to modernize their legacy infrastructure that was complex to maintain at scale. To speed up their workloads and achieve greater operational efficiency, they prioritized serverless capabilities as part of their transformation strategy. By adopting the Databricks Data Intelligence Platform, they have completely transformed how they manage and analyze large datasets. With Databricks Lakeflow Jobs, processes are automated and data pipelines are delivering real-time insights that drive smarter, faster decisions — giving players and coaches the tools they need to win at the highest level.

Legacy systems strike out on delivering timely insights

The Cincinnati Reds, one of Major League Baseball's oldest franchises, faced legacy infrastructure and data challenges hindering their ability to make smarter, faster decisions on baseball feedback loops to win more baseball games. Over the past decade, data stream volume and complexity grew by up to 100x, driven by the rise of Internet of Things (IoT) devices and sensors that track player performance, in-stadium activity and fan engagement. This growth created significant challenges in handling and processing data in their on-premises systems. Processing billions of rows and queries often took hours or even days, making real-time applications impractical. Scalability was limited, forcing manual troubleshooting that consumed valuable time.

Latency had also become a critical factor, with users expecting immediate access to insights and reports, which were previously only available after long delays. The data also needed to be accessible and usable for different roles. From data scientists who process large amounts of data and train models to help the team make predictions to less technical users within the business who just need small pieces of information to help drive strategic decision-making. Additionally, application developers require programmatic access to the data stores.

"A lot of the stuff we were doing before was just troubleshooting issues like missing rows or unprocessed data. We'd spend hours just fixing those problems instead of actually driving insights or improving performance. Our team couldn't keep up with the increasing volume and variety of data — and this manual approach left us at a disadvantage, especially with the demand for real-time insights in a sport that moves as fast as baseball," Bryce Dugar, Data Engineering Manager at the Cincinnati Reds, explained.

The Cincinnati Reds realized they needed a more scalable and efficient solution as their data needs rapidly expanded. Recognizing the limitations of their traditional systems, the Reds adopted the Databricks Data Intelligence Platform to automate workflows and enable real-time data access. Specifically, with Databricks Lakeflow Jobs — the unified orchestration tool for data, analytics and AI — they sought to more easily define, manage and monitor jobs with multiple tasks for ETL, reduce manual intervention and deliver more timely insights for improved decision-making.

Unifying data orchestration with Databricks Lakeflow Jobs

With the Databricks Platform serving as a core piece of their data infrastructure, Databricks Lakeflow Jobs became central to solving the Reds' challenges by tailoring orchestration to meet their specific needs and integrating seamlessly via APIs. This enabled triggering jobs with parameters, allowing greater workflow control. Analysts gained independence to develop and execute notebooks with improved traceability and iterative processing capabilities.

The team also moved to a serverless architecture, which significantly reduced latency. Previously, using job compute, even with pools, often led to delays while waiting for jobs to spin up.

"With serverless Databricks Lakeflow Jobs, we've achieved a 3–5x improvement in latency. What used to take 10 minutes now takes just 2–3 minutes, significantly reducing processing times. This has enabled us to deliver faster feedback loops for players and coaches, ensuring they get the insights they need in near real time to make actionable decisions," Bryce said.

In addition to these time savings, the team saw a 65–80% reduction in VM costs by transitioning to serverless Databricks Lakeflow Jobs, making the solution not only faster but also significantly more cost-efficient. With the Reds running 15,000–20,000 workflow steps daily, this efficiency saved hundreds of thousands of compute minutes each day, enabling new workloads and faster report delivery.

The team transitioned from Azure Data Factory to Databricks. "With Databricks, we've completely transformed how we handle data. Moving to a serverless architecture and leveraging serverless Databricks Lakeflow Jobs has not only streamlined our pipelines but also dramatically reduced latency by up to 83%," Bryce said. "What used to take an hour can now be done in 10 minutes, allowing us to run more processes, deliver reports faster and provide near-instant feedback for coaching and player development. It's a game changer for how we make real-time decisions in a fast-paced sport like baseball."

The platform's serverless environment removed constraints related to memory and compute cores. This allowed the team to handle large, complex data queries more efficiently, ultimately supporting faster and more accurate insights for game-day decisions. The traceability of workflows from data ingestion to the final output also improved accountability, making it easier for the team to monitor, troubleshoot and optimize their data pipelines. Data scientists and engineers no longer face system instability, session crashes or lengthy processing times, enabling smoother workflows and higher job satisfaction. Furthermore, the modular, automated nature of Databricks Lakeflow Jobs significantly lowered the overhead for developers, freeing them up to focus on high-value work.

Faster data serves up game-winning decisions

Transitioning to serverless compute significantly reduced job latency, cutting pipeline steps from 10–15 minutes to as little as 2–3 minutes. With Databricks Lakeflow Jobs streamlining processes, ETL tasks and data science workflows now operate with enhanced speed and agility.

This efficiency ensures rapid post-game data availability, empowering coaches with near real-time insights to make critical adjustments and provide immediate feedback to players.

"With Databricks, the time savings we've realized is almost immeasurable — enabling things we couldn't even consider before. That shift has fundamentally changed how we operate and how quickly we can deliver insights to coaches, players and analysts," Bryce concluded. This transformational shift has solidified Databricks Lakeflow Jobs as a critical enabler of the Reds' data-driven strategy, empowering the team to make smarter, faster decisions in the high-stakes world of professional baseball.

PORSCHE HOLDING | CUSTOMER STORY

Porsche Holding: Unifying Customer Data to Drive a New Automotive Experience

INDUSTRY Manufacturing

CLOUD Azure

85%

Faster development using production-ready Lakeflow Connect instead of a homegrown solution

databricks

> Using the Salesforce connector from Lakeflow Connect helps us close a critical gap for Porsche from the business side on ease of use and price. On the customer side, we're able to create a completely new customer experience that strengthens the bond between Porsche and the customer with a unified and unfragmented customer journey.
>
> **— Lucas Sulzberger, Project Manager, Porsche Holding**

Porsche Holding Salzburg is Europe's largest automotive retail company, operating in 23 countries throughout Europe as well as in Colombia, Chile, China, Malaysia, Singapore and Japan. They represent the Volkswagen Group brands in wholesale (as an importer), retail (through their dealers) and the after-sales business (service). The company also covers the entire spectrum of the automotive trade, including spare parts distribution, a full range of automobile financing services and in-house IT system development.

Since March 2011, Porsche has been a 100% subsidiary of Volkswagen AG, contributing decades of know-how in the automotive business to Volkswagen Group global sales. By the end of 2023, the company employed a workforce of 35,900 and had sold more than 747,700 new vehicles, generating a turnover of €29.4 billion.

However, fragmented data systems meant customer engagements remained siloed, limiting the company's ability to connect them and deliver modern, personalized experiences. To transform how they engaged with customers while also enabling data-driven decision-making, Porsche turned to Lakeflow Connect and the Databricks Data Intelligence Platform.

Siloed systems limit a holistic view of customer insights

When customers interact with Porsche — from dealership visits and banking services to after-sales support — they generate millions of data points that hold the key to delivering personalized customer experiences and driving business growth.

In an automotive retail environment where exceptional service directly correlates with repeat purchases and brand loyalty, Porsche data analysts struggled to connect customer interaction data from Salesforce — including email engagement metrics, campaign responses and customer behavior patterns — to the wealth of information stored in their data lake. Marketing teams couldn't easily access the data they needed to optimize campaigns and calculate ROI. Customer experience teams lacked the unified view necessary to deliver personalized recommendations for specific customer offers such as leasing renewals or maintenance services. "We have a lot of touchpoints and systems interacting with our customers," Lucas Sulzberger, Project Manager at Porsche Holding, said. "It was pretty difficult to have all the important data that helps us create the perfect customer story in one place."

The manual maintenance required for a custom solution would likely consume significant resources and make it challenging to scale efficiently. This would be especially problematic given the automotive industry's rapid shift toward omnichannel retail, with most buyers now expecting seamless transitions between online research, virtual showrooms and in-person dealership experiences. The stakes were particularly high because research shows that vehicle buyers looking for premium or luxury brands spend significantly less time at dealerships than mass-market customers, making every interaction more crucial.

Porsche's main priority was to establish a unified data infrastructure that would support the ability to scale while maintaining operational efficiency and implementing robust, sophisticated analytics. Specifically, the company wanted to eliminate data silos between their CRM and analytics environments, reduce the maintenance burden of custom integrations and enable more advanced use cases like predictive analytics and personalized marketing. Instead of having a custom or homegrown solution, they wanted a more scalable and sustainable platform that could handle complex data requirements.

Incremental data ingestion with Databricks Lakeflow Connect

To address their data goals, Porsche turned to Databricks Lakeflow Connect, which offers native data ingestion connectors for popular SaaS applications, databases and file sources for any practitioner wanting to build incremental data pipelines at scale. These built-in connectors provide efficient, end-to-end incremental ingestion and easy setup with a simple UI or API access. Fully integrated with the Databricks Data Intelligence Platform, Lakeflow Connect provides customers with the benefits of unified governance, powered by serverless compute. More broadly, Lakeflow Connect is part of Lakeflow, a new unified data engineering solution that spans ingestion, transformation and orchestration.

Porsche was interested in the Lakeflow Connect Salesforce connector to create unified access to their customer data and access Salesforce Sales Cloud directly from within their Databricks Platform. Lakeflow Connect offered a streamlined solution to the company's data integration challenges — they were able to ingest data from Salesforce, transform it in batch and streaming modes, and deploy and operate in production.

In addition, Lakeflow Connect provided the needed flexibility to adjust data synchronization schedules based on business demands, which is a mission-critical feature for Porsche. Whether the data and AI team want daily updates, monthly updates or hourly synchronization, Lakeflow Connect can be fine-tuned to match specific use cases. "With Databricks, we get great flexibility with very low effort," Markus Gruber, Principal Data Scientist at Porsche Holding, noted. "Basically, you just set the schedule and it's done. That saved us a lot of money during implementation and also long term for maintenance."

"Lakeflow Connect allows us to access all of our CRM data in near real-time — everything with sales and marketing touchpoints — and it's easy to set up, efficient and flexible," Lucas added.

Furthermore, since Lakeflow Connect is a managed product with full observability, monitoring and governance, there's no need to worry about ongoing maintenance. This is particularly helpful for Porsche and the broader automotive sector, where IT resources are increasingly focusing on innovation in emerging areas such as connected car technologies and digital retail experiences rather than allocating time to managing infrastructure.

The Databricks Data Intelligence Platform now forms the core of Porsche Holding's data lake. Unity Catalog provides data governance and security, the ingestion pipeline works with Salesforce and Lakeflow Connect to ensure reliable synchronization of CRM data, and Lakeflow Jobs enables sophisticated job scheduling and notifications.

Transforming customer experience through unified data

The Databricks Data Intelligence Platform and Lakeflow Connect have transformed how Porsche manages and uses their customer data.

By opting to use Lakeflow Connect instead of building a custom solution, the company has reaped the benefits of both operational efficiency and cost management.

Internally, teams at Porsche now spend less time managing data integration processes. "Lakeflow Connect has enabled our dedicated CRM and data science teams to be more productive as they can now focus on their core work to help innovate instead of spending valuable time on the data ingestion integration with Salesforce," Markus explained.

This shift in focus aligns with broader industry trends as automotive companies redirect significant portions of their IT budgets toward customer experience innovations and digital transformation initiatives.

From a business perspective, this managed solution has reduced development time and effort, accelerated time to production and minimized risks and costs through a sustainable approach to data connectivity. "Lakeflow Connect helps us close a critical gap for Porsche from the business side on ease of use and price," Lucas said.

Perhaps most notably, the unified data infrastructure has enabled Porsche to reimagine their customer journey. Instead of fragmented interactions across different touchpoints, customers now experience a more coherent, personalized relationship with the brand. For example, customers can receive a personalized offer based on their previous interactions with Porsche. Lucas explained, "On the customer side, we're able to create a completely new customer experience that strengthens the bond between Porsche and the customer with a unified customer journey."

This transformation also aligns with Porsche's broader strategic vision. "One of our goals is to become a more data-driven company," Markus stated. "The Databricks Platform is one of the important components to help us get there."

Hinge Health | CUSTOMER STORY

Hinge Health: Developing Personalized Care Plans to Improve Patient Outcomes

INDUSTRY Healthcare and Life Sciences

PLATFORM Spark Declarative Pipelines

CLOUD AWS

50%
Reduced costs

80%
Reduced latency

300 TB
Of data transformed by Spark Declarative Pipelines

databricks

> "We've reduced costs by at least 50% by migrating to Spark Declarative Pipelines compared to when we started this journey, even though we now have 10 times more data."
>
> **— Veera Mukkanagoudar, Senior Engineering Manager, Hinge Health**

Hinge Health is transforming the way pain is treated and prevented by connecting people digitally with expert clinical care. The company's solutions are designed to help people decrease musculoskeletal pain, surgeries and opioid use, resulting in proven reductions in spending on medical claims. Hinge Health is available to over 18 million people and has 1,800 employer health plan clients. Personalized care plans require data to be continuously updated and analyzed. Hinge Health needed to reduce costs and scale as new data sources were onboarded. The company turned to Databricks Spark Declarative Pipelines to simplify change data capture, improve data reliability, meet service level agreements (SLAs) and reduce total cost of ownership.

Managing a 10x growth in data

Hinge Health uses custom ingestion and CDC to consume data from 70+ first- and third-party data sources via Fivetran. The company currently has 2,000 Postgres tables spread across 35+ databases. Over the last few years, the amount of data Hinge consumes has grown 10-fold, making it difficult to propagate schema and data at the same time. "When we started building our data platform, we had 20–30 terabytes of data. That grew to nearly 300 terabytes within a year," Veera Mukkanagoudar, Senior Engineering Manager at Hinge Health, said. "As a data engineering leader, cost optimization is an ongoing journey. We have to build things where the costs grow linearly as the data volume and velocity grows. We could not grow our data cost by 10 times."

The company needed to build an efficient, low-cost, low-latency data pipeline to ingest and transform data from heterogeneous sources at scale.

The journey to an optimized CDC architecture

Hinge Health data engineering leaders evaluated a number of solutions to enable their data engineering mission, including Snowflake, EMR, Redshift and Glue. Ultimately, the company chose Databricks Spark Declarative Pipelines. "We liked Databricks the best," Veera said. "One feature stood out the most — the ability to support extract, load, transform (ELT), data warehousing, ML streaming, batch and generative AI (GenAI) workloads without moving data to meet the needs of the use case. We don't have to copy the data between different systems to serve different applications. Name a problem in the data space and there is a Databricks tool available to solve it. We can serve everything on a single platform. That adds up to faster application development time."

Once they selected Databricks, Hinge Health spun up a new data pipeline using Spark Declarative Pipelines to unify batch and streaming workloads and transform 300 terabytes of data. The primary objective was to isolate any large loads coming from large databases so they didn't impact smaller databases. "We don't have control over upstream databases," Veera said. "There is a possibility that an upstream database or source could do a backward-compatible change, which could cause a failure. We did not want it to impact other data pipelines."

Veera and his team built an initial pipeline using Debezium Postgres reader to stream data to Kafka. They then used Amazon S3 to sync the data to an S3 bucket. They used two Spark Declarative Pipelines pipelines: one to ingest the data from the S3 bucket to a staging table, which broadcast that data into a series of history tables, and the other to generate data for the latest version of the table.

That architecture had a couple of challenges. First, using S3 meant they needed storage, query and additional compute to store and retrieve data from intermediary storage. This had implications for compute utilization and cost. There were also reliability and latency challenges that made it difficult for the team to meet their SLAs.

Veera and his team realized they needed to move to Kafka as the data source rather than S3. As they did so, they also improved the staging table design which enabled better compute utilization and allowed them to easily onboard data to pipelines that could accommodate the load rather than to pipelines that already had a taxed driver. The team now mirrors source databases in a lakehouse target by collecting and writing change logs from Aurora to serve data in Delta for GenAI and non-GenAI use cases using Spark Declarative Pipelines. With topics as their new source, the team was able to simplify and consolidate.

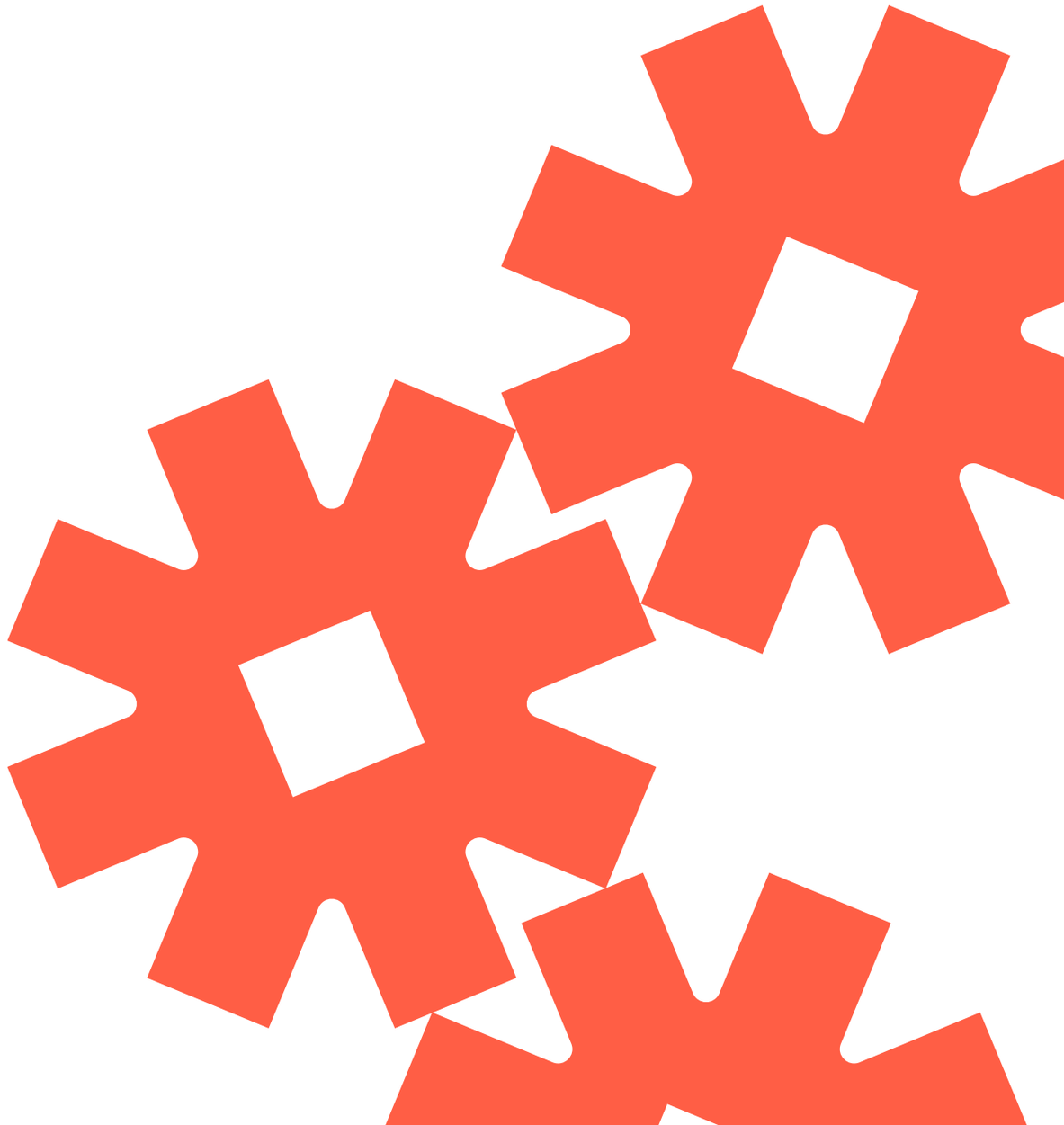
Veera said he especially likes Spark Declarative Pipelines's Python- and SQL-based parameterization features. "We have 35 data pipelines running, but they all run off a single code base," he said. "We have parameterized all the table definitions, columns, data types and the database names — everything."

Lowering costs while scaling

With a new CDC pipeline reading from Kafka instead of S3 and the ability to configure ingestion from multiple databases with a single data pipeline, Hinge Health has dramatically reduced their costs. Veera estimated, "We've reduced costs by at least 50% by migrating to Spark Declarative Pipelines compared to when they started this journey, even though they now have 10 times more data." Veera added, "Taking out S3 as a part of our data pipeline was very powerful." Additionally, they've improved compute utilization in smaller databases, increased reliability and reduced latency by 80%. "With the first iteration of the pipeline, it was challenging to meet SLAs and we were always running into issues with large data changes in upstream databases," he explained. "Latency was nearly four times when we were using S3."

Spark Declarative Pipelines significantly enhances Hinge Health's data architecture by providing the benefits of materialized views, streamlining their CDC processes. Business insights from change data are served to the downstream teams as precomputed materialized views. Spark Declarative Pipelines combines streaming tables with incrementally updated materialized views, avoiding the need to completely rebuild the view when new data arrives. This allows Hinge Health to reduce data processing costs and update reports and dashboards in real time.

Ultimately, Hinge Health's optimized CDC architecture adds up to happier users. "Data engineers are happy because they have less troubleshooting to do," Veera said. "Our data science team is happy because we can meet our SLAs. Our engineering leadership is happy because they are always pushing us to optimize our costs. And our ML applications team is happy because they want to see data as fast as possible."



Ingest, transform and orchestrate with a unified data engineering solution

Databricks Lakeflow is built to address the key challenges data engineering teams face today. As an end-to-end data engineering solution with built-in data intelligence, Lakeflow unifies and simplifies data ingestion, transformation and orchestration by offering a singular approach to data pipeline creation.

Learn more **Watch a demo** **Try Databricks free**

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