databricks

Big Book of Data Engineering

3RD EDITION

Contents

01

Introduction to Data Engineering on Databricks

databricks

A recent MIT Tech Review Report shows that 88% of surveyed organizations are either investing in, adopting or experimenting with generative AI (GenAI) and 71% intend to build their own GenAI models. This increased interest in AI is fueling major investments as AI becomes a differentiating competitive advantage in every industry. As more organizations work to leverage their proprietary data for this purpose, many encounter the same hard truth:

The best GenAI models in the world will not succeed without 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. In other words, good data engineering is an essential component of success in every data and AI initiative and especially for GenAI.

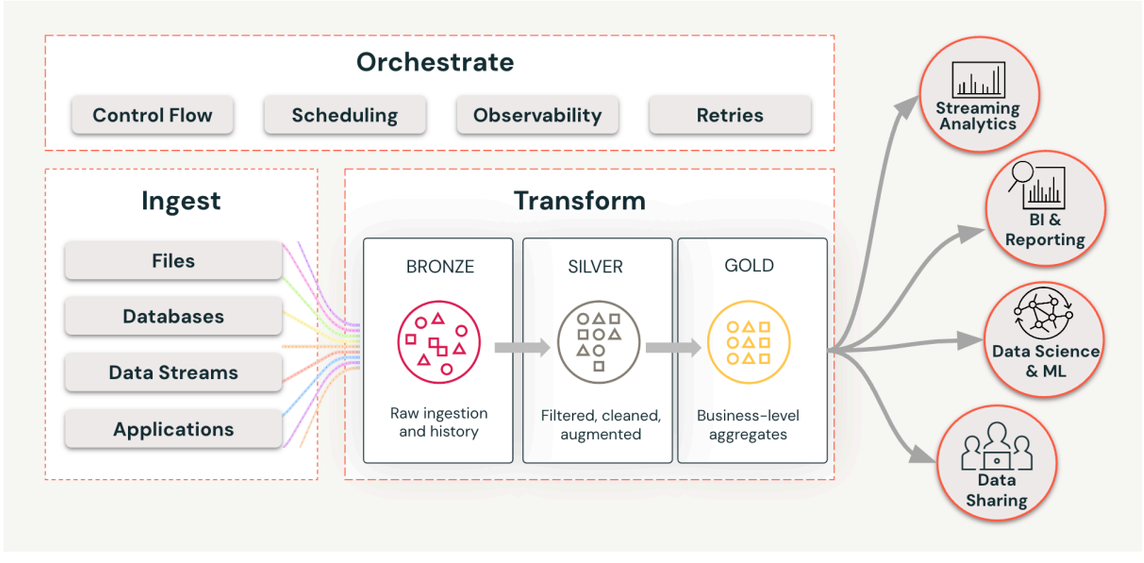
Using practical guidance, useful patterns, best practices and real-world examples, this book will provide you with an understanding of how the Databricks Data Intelligence Platform helps data engineers meet the challenges of this new era.

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, business intelligence (BI) or machine learning (ML) model training. In other words, it's the process of preparing data so 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 on 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 handling 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 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.

- **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 data pipelines 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 who 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 many organizations face, especially in highly regulated industries.

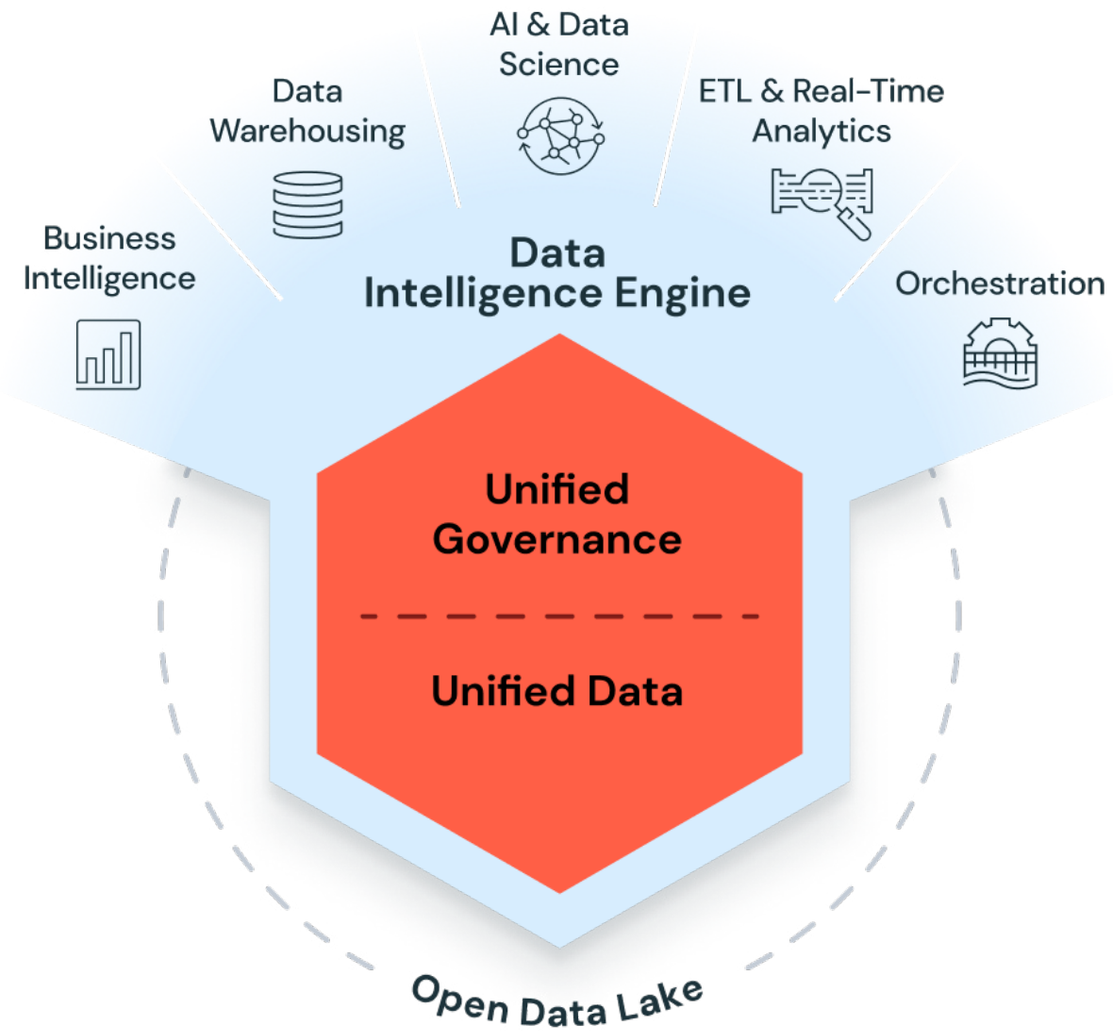
These challenges stress the importance of choosing the right data platform for navigating new waters in the age of AI. But a data platform in this new age can also go beyond addressing just the challenges of building AI solutions. The right platform can improve the experience and productivity of data 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.

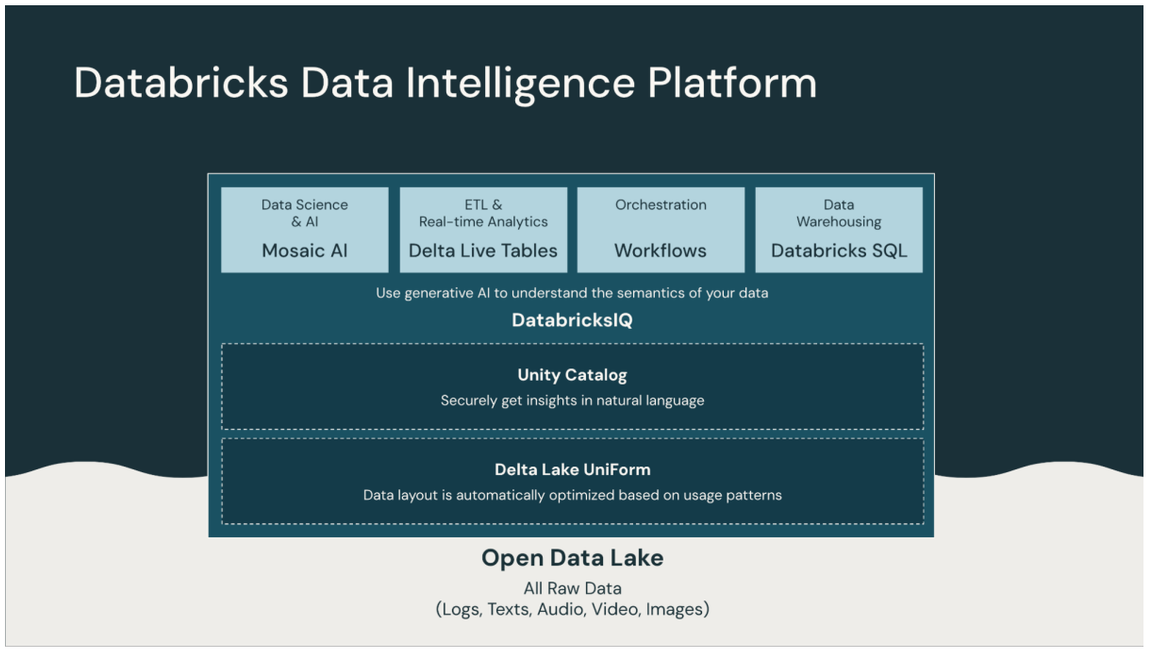
The Databaricks Data Intelligence Platform

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 so they can 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 it's 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, here's an overview of the different parts of the architecture as it relates to data engineering.



Data reliability and performance with Delta Lake

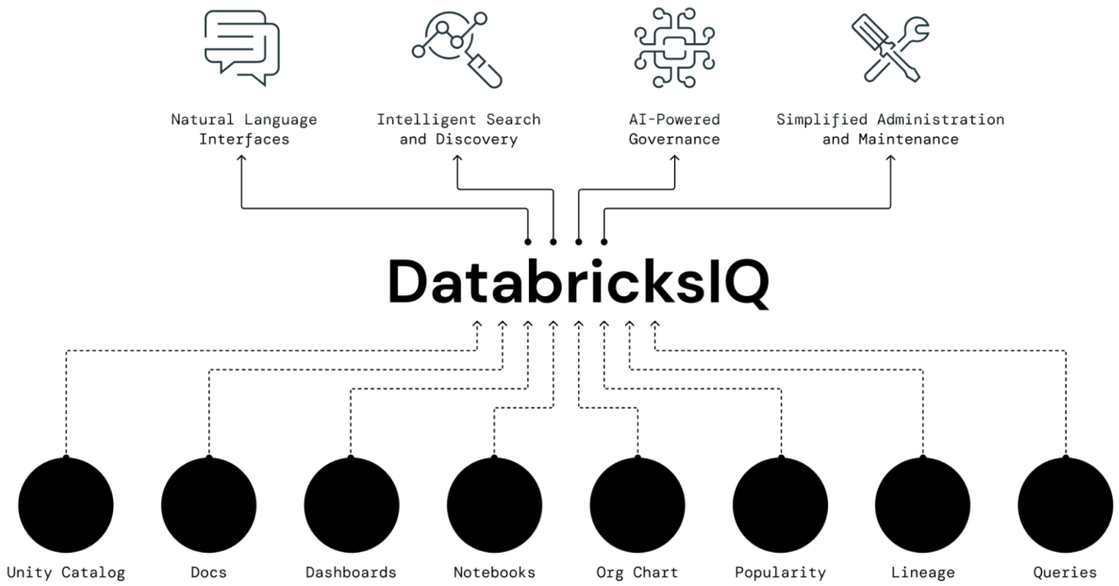
To bring openness, reliability and lifecycle management to data lakes, the Databricks lakehouse architecture is built on the foundation of Delta Lake, an open source, highly performant storage format that solves challenges around unstructured/structured data ingestion, the application of data quality, difficulties with deleting data for compliance or issues with modifying data for data capture. Delta Lake UniForm users can now read Delta tables with Hudi and Iceberg clients, keeping them in control of their data. In addition, Delta Sharing enables easy and secure sharing of datasets inside and outside the organization.

Unified governance with Unity Catalog

With Unity Catalog, data engineering and governance teams benefit from an enterprisewide data catalog with a single interface to manage permissions, centralize auditing, automatically track data lineage down to the column level and share data across platforms, clouds and regions.

DatabricksIQ — the Data Intelligence Engine

At the heart of the Data Intelligence Platform lies DatabricksIQ, the engine that uses AI to infuse intelligence throughout the platform. DatabricksIQ is a first-of-its-kind Data Intelligence Engine that uses AI to power all parts of the Databricks Data Intelligence Platform. It uses signals across your entire Databricks environment, including Unity Catalog, dashboards, notebooks, data pipelines and documentation, to create highly specialized and accurate generative AI models that understand your data, your usage patterns and your business terminology.



Reliable data pipelines and real-time stream processing with Delta Live Tables

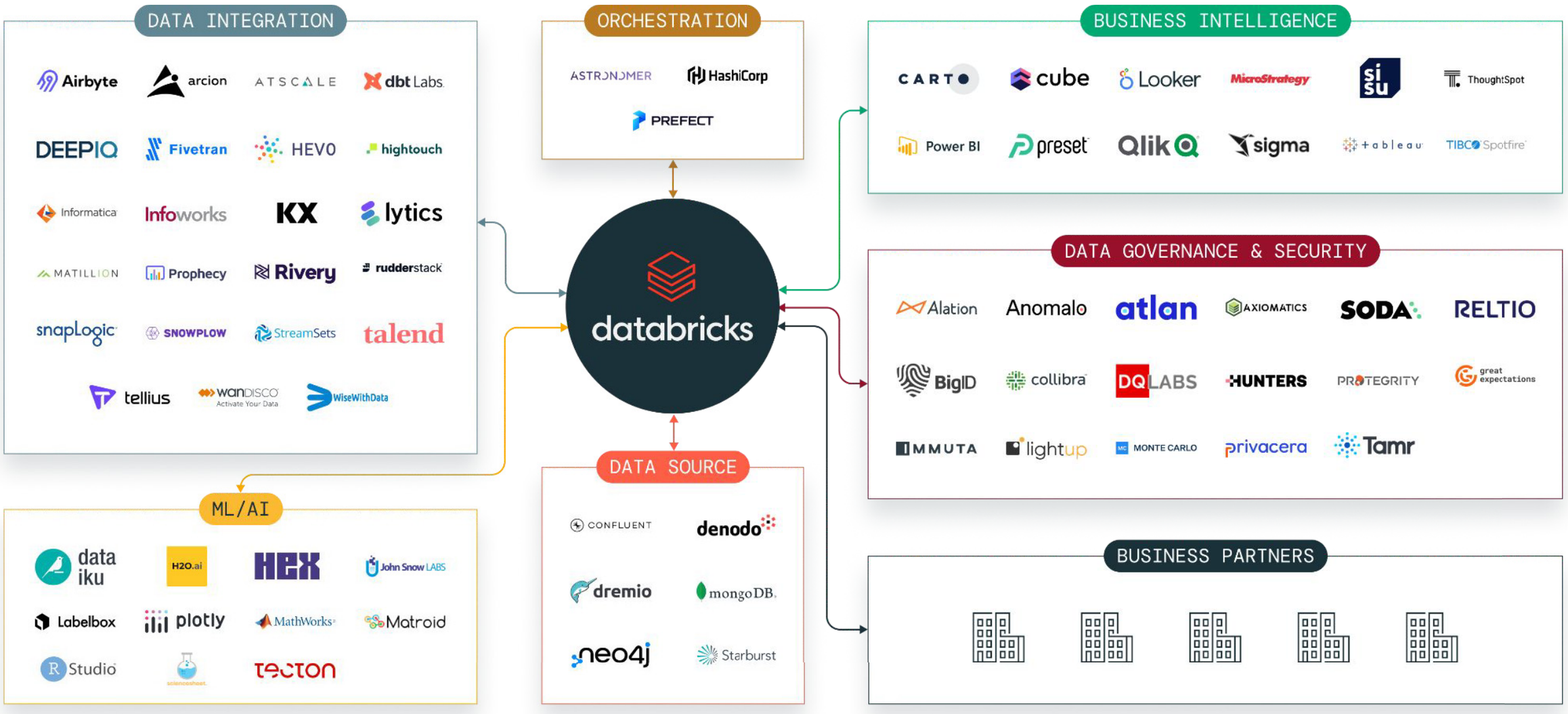
Delta Live Tables (DLT) is a declarative ETL framework that helps data teams simplify and make ETL cost-effective in streaming and batch. Simply define the transformations you want to perform on your data and let DLT pipelines automatically handle task orchestration, cluster management, monitoring, data 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. DLT fully supports both Python and SQL and is tailored to work with both streaming and batch workloads.

Unified data orchestration with Databricks Workflows

Databricks Workflows offers a simple, reliable orchestration solution for data and AI on the Data Intelligence Platform. Databricks Workflows lets you define multi-step 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 native orchestrator, Databricks Workflows also provides advanced observability to monitor and visualize workflow execution along with alerting capabilities for when issues arise. Databricks Worklfows offers serverless compute options so you can leverage smart scaling and efficient task execution.

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.

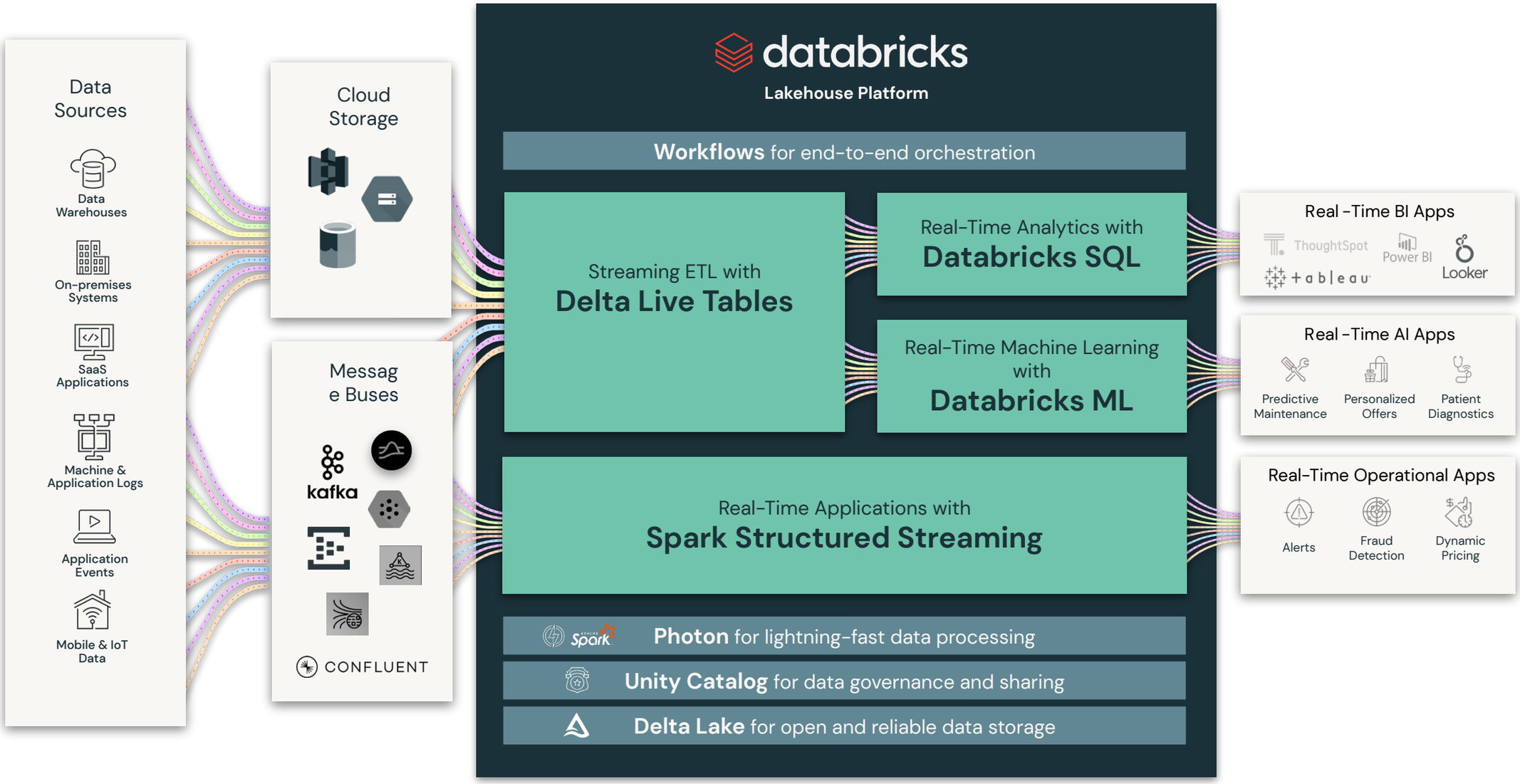


The Data Intelligence Platform integrates with a large collection of technologies

Why data engineers choose the Data Intelligence Platform

So how does the Data Intelligence Platform help with each of the data engineering challenges discussed earlier?

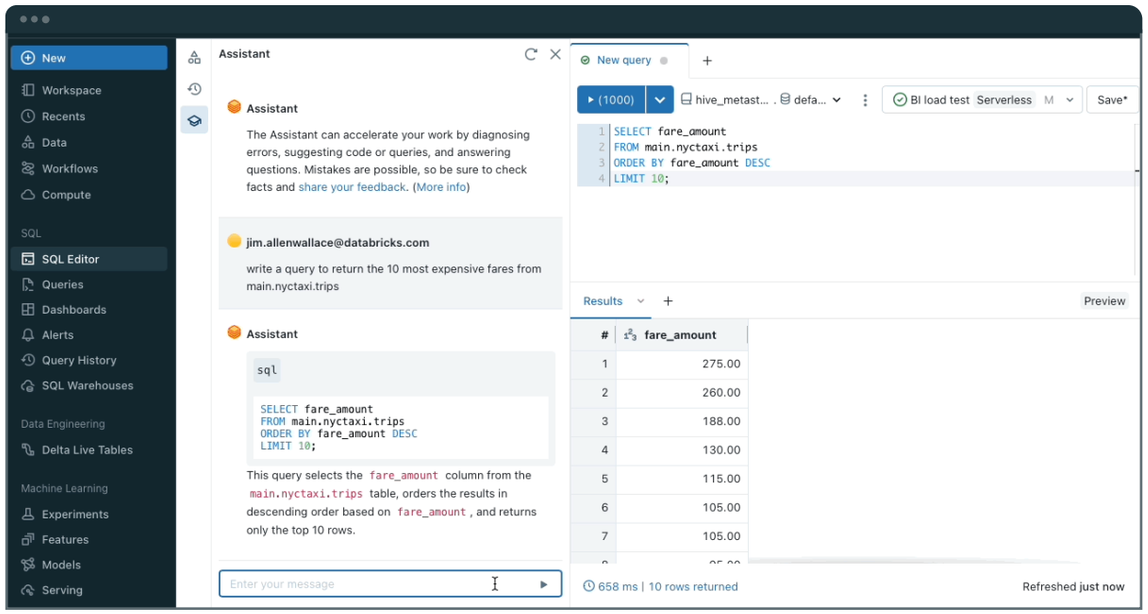
- **Real-time data stream processing:** The Data Intelligence Platform simplifies development and operations by automating the production aspects associated with building and maintaining real-time data workloads. Delta Live Tables provides a declarative way to define streaming ETL pipelines and Spark Structured Streaming helps build real-time applications for real-time decision-making.



- **Reliable data pipelines at scale:** Both Delta Live Tables and Databricks Workflows use smart autoscaling and auto-optimized resource management to handle high-scaled workloads. With lakehouse architecture, the high scalability of data lakes is 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 Delta Lake and coupled with data quality–specific features offered by Delta Live Tables — ensures high data quality. These features include setting data "expectations" to handle corrupt or missing data as well as automatic retries. In addition, both Databricks Workflows and Delta Live Tables provide full observability to data engineers, making issue resolution faster and easier.
- **Unified governance with secured data sharing:** Unity Catalog provides a single governance model for the entire platform so every dataset and pipeline are governed in a consistent way. 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.

In addition, data engineers using the Data Intelligence Platform benefit from cutting-edge innovations in the form of AI-infused intelligence in the form of DatabricksIQ:

- **AI-powered productivity:** Specifically useful for data engineers, DatabricksIQ powers 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.



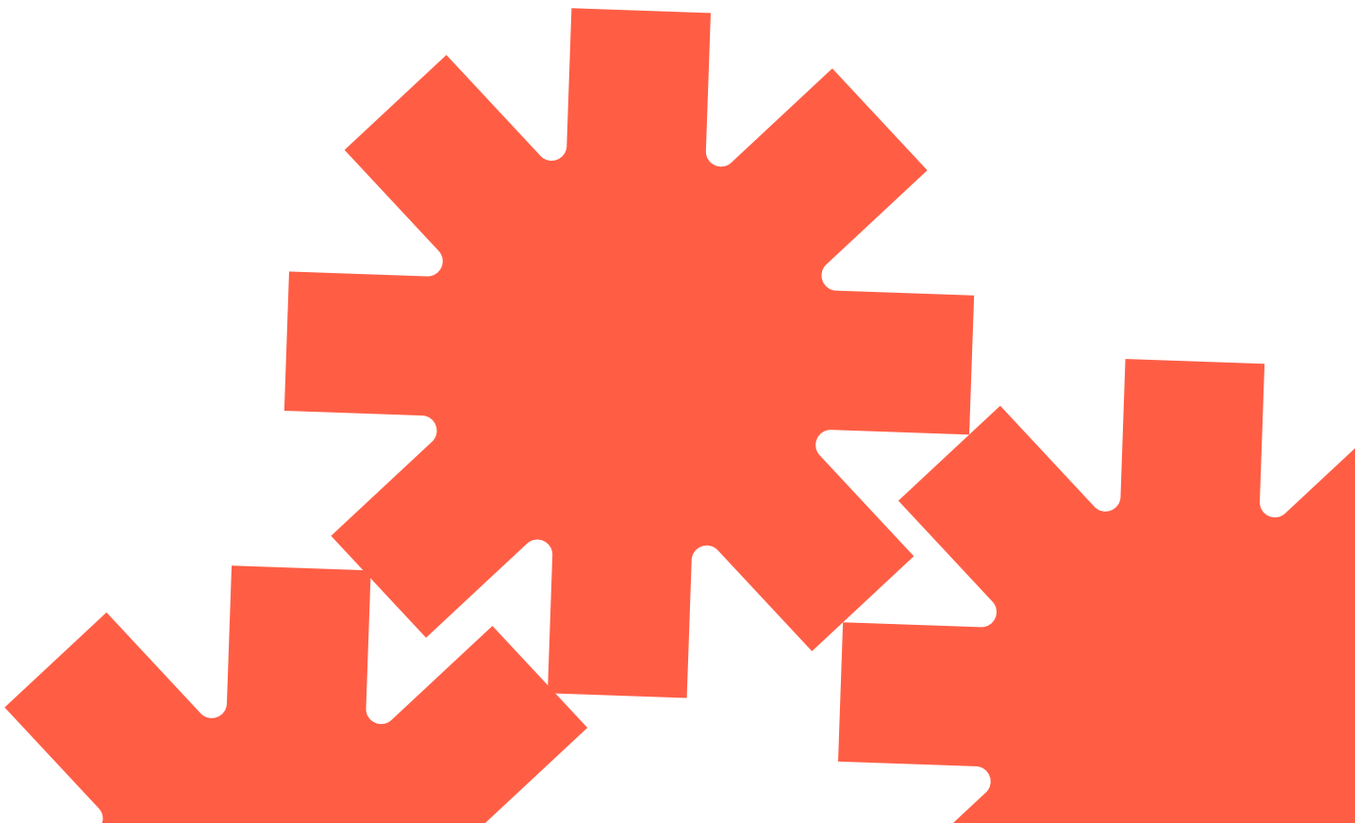
Conclusion

As organizations strive to innovate with AI, data engineering is a focal point for success by delivering reliable, real-time data pipelines that make AI possible. With the Data Intelligence Platform, built on the lakehouse architecture and powered by DatabricksIQ, 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 the 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.

FOLLOW PROVEN BEST PRACTICES

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 the Data Intelligence Platform.

As you explore the rest of this guide, you can find datasets and code samples in the various Databricks Solution Accelerators, so you can get your hands dirty and start building on the Data Intelligence Platform.



02

Guidance and Best Practices

Databricks Assistant Tips and Tricks for Data Engineers

by Jackie Zhang, Rafi Kurlansik and Richard Tomlinson

The generative AI revolution is transforming the way that teams work, and Databricks Assistant leverages the best of these advancements. It allows you to query data through a conversational interface, making you more productive inside your Databricks Workspace. The Assistant is powered by DatabricksIQ, the Data Intelligence Engine for Databricks, helping to ensure your data is secured and responses are accurate and tailored to the specifics of your enterprise. Databricks Assistant lets you describe your task in natural language to generate, optimize, or debug complex code without interrupting your developer experience.

In this chapter we'll discuss how to get the most out of your Databricks Assistant and focus on how the Assistant can improve the life of Data Engineers by eliminating tedium, increasing productivity and immersion, and accelerating time to value. We will follow up with a series of posts focused on different data practitioner personas, so stay tuned for upcoming entries focused on data scientists, SQL analysts, and more.

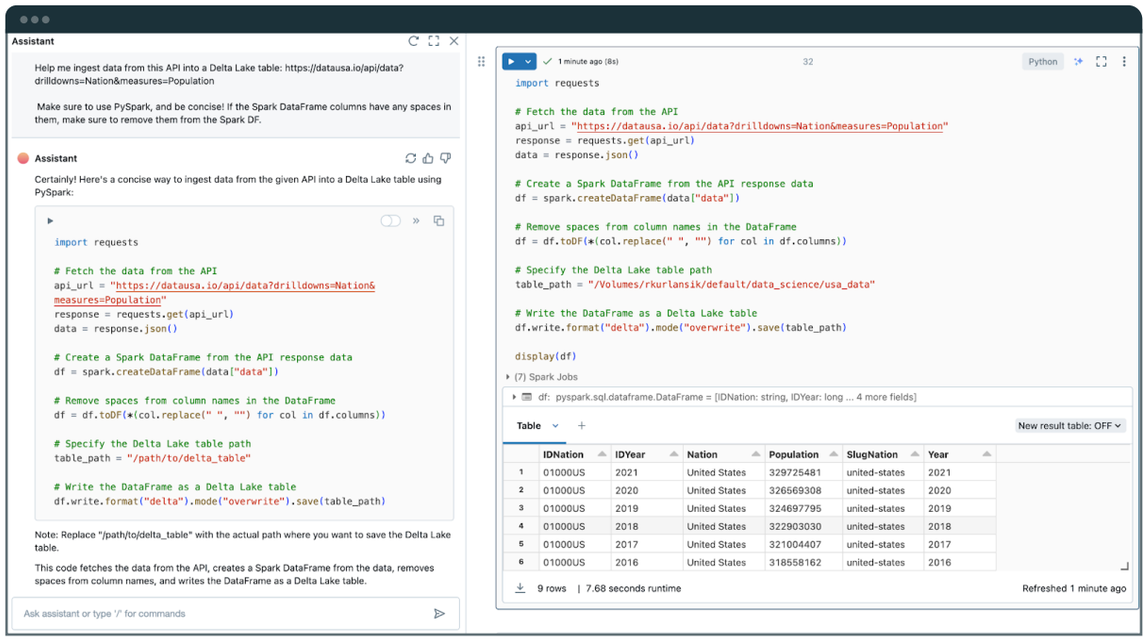
INGESTION

When working with Databricks as a data engineer, ingesting data into Delta Lake tables is often the first step. Let's take a look at two examples of how the Assistant helps load data, one from APIs, and one from files in cloud storage. For each, we will share the prompt and results. As mentioned in the 5 tips blog, being specific in a prompt gives the best results, a technique consistently used in this article.

To get data from the datausa.io API and load it into a Delta Lake table with Python, we used the following prompt:

Help me ingest data from this API into a Delta Lake table: https://datausa.io/api/data?drilldowns=Nation&measures=Population

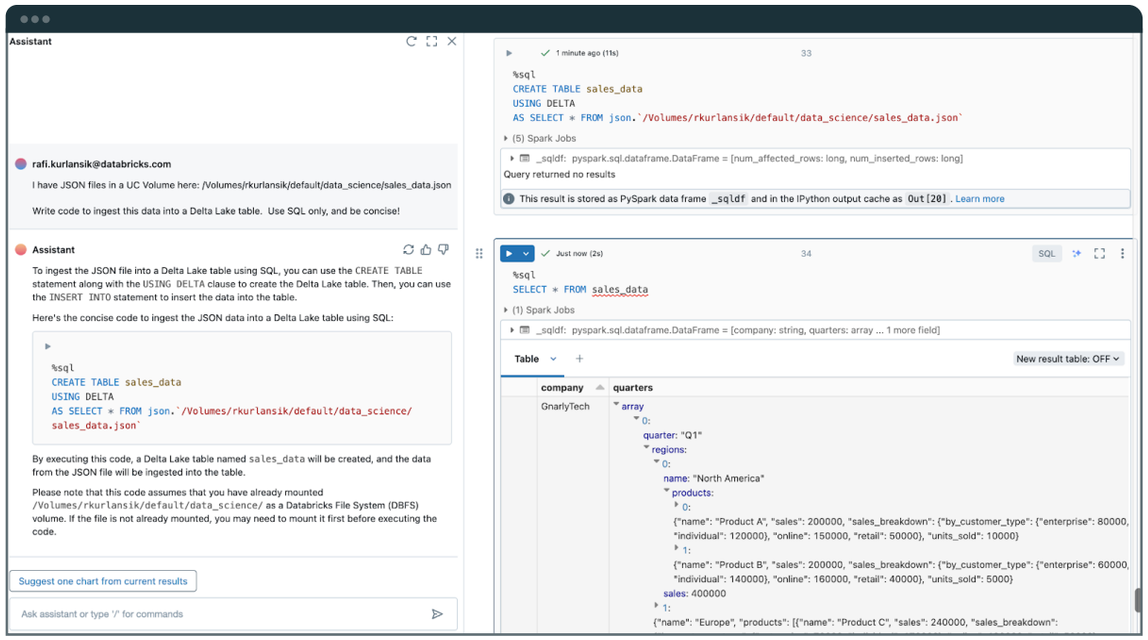
Make sure to use PySpark, and be concise! If the Spark DataFrame columns have any spaces in them, make sure to remove them from the Spark DF.



A similar prompt can be used to ingest JSON files from cloud storage into Delta Lake tables, this time using SQL:

I have JSON files in a UC Volume here: /Volumes/rkurlansik/default/data_science/sales_data.json

Write code to ingest this data into a Delta Lake table. Use SQL only, and be concise!



TRANSFORMING DATA FROM UNSTRUCTURED TO STRUCTURED

Following tidy data principles, any given cell of a table should contain a single observation with a proper data type. Complex strings or nested data structures are often at odds with this principle, and as a result, data engineering work consists of extracting structured data from unstructured data. Let's explore two examples where the Assistant excels at this task — using regular expressions and exploding nested data structures.

Regular expressions

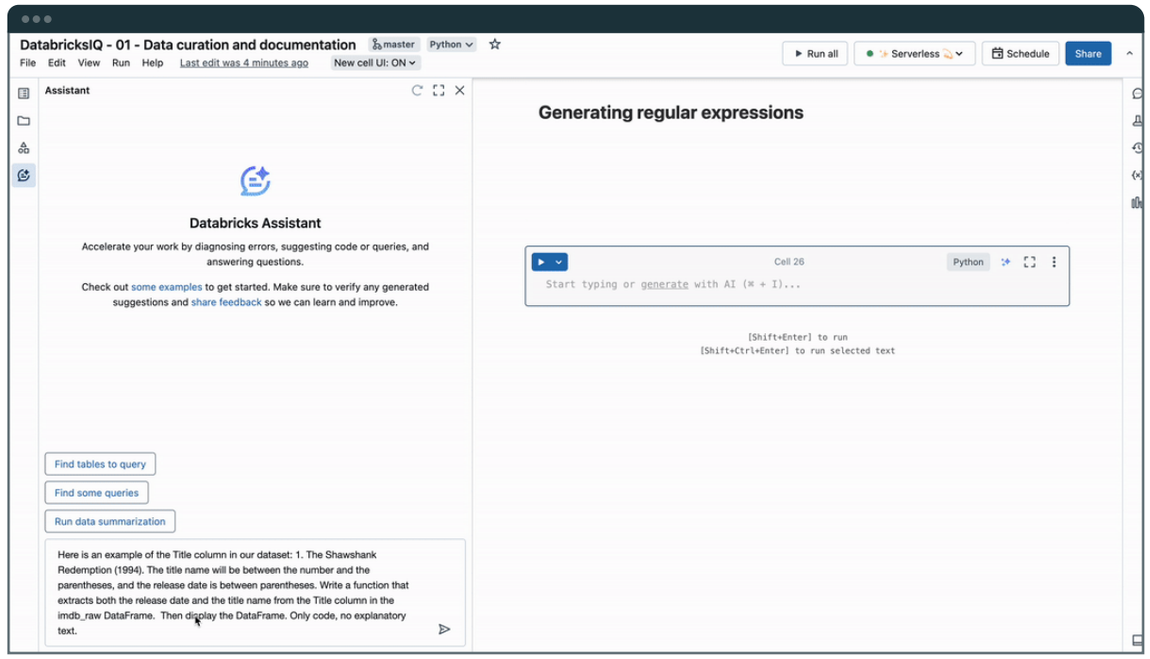
Regular expressions are a means to extract structured data from messy strings, but figuring out the correct regex takes time and is tedious. In this respect, the Assistant is a boon for all data engineers who struggle with regex.

Consider this example using the *Title* column from the IMDb dataset:

Title	Certificate	Duration	Genre
1. The Shawshank Redemption (1994)	R	142 min	Drama

This column contains two distinct observations — film title and release year. With the following prompt, the Assistant identifies an appropriate regular expression to parse the string into multiple columns.

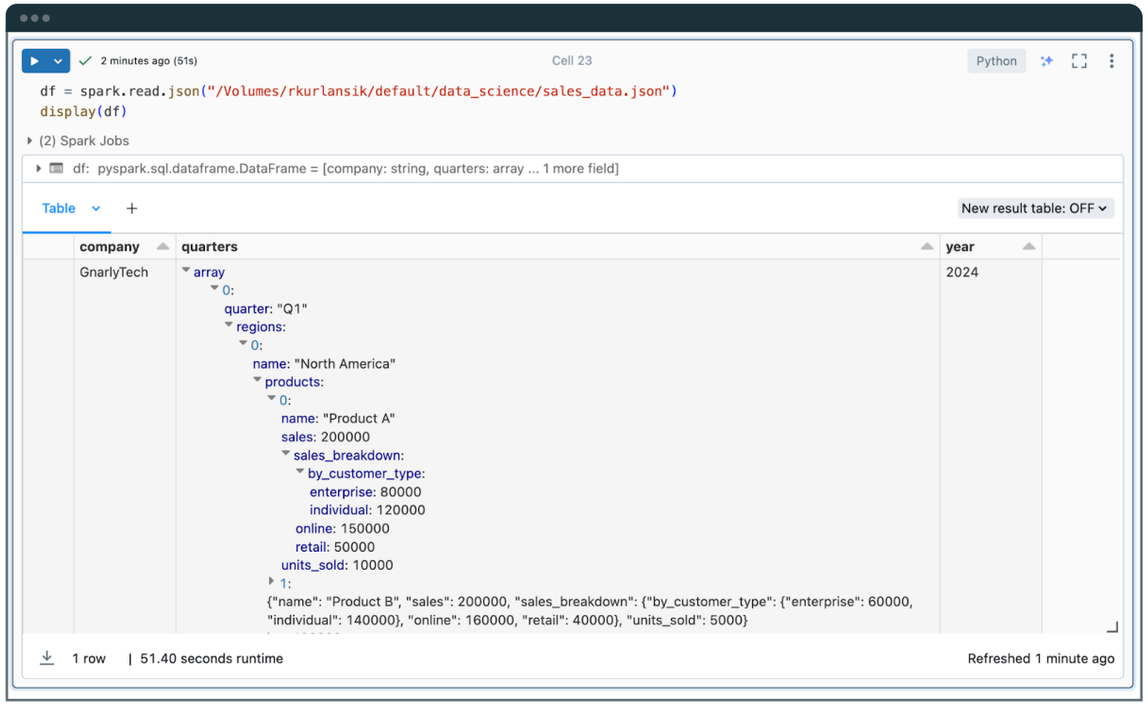
Here is an example of the Title column in our dataset: 1. The Shawshank Redemption (1994). The title name will be between the number and the parentheses, and the release date is between parentheses. Write a function that extracts both the release date and the title name from the Title column in the imdb_raw DataFrame.



Providing an example of the string in our prompt helps the Assistant find the correct result. If you are working with sensitive data, we recommend creating a fake example that follows the same pattern. In any case, now you have one less problem to worry about in your data engineering work.

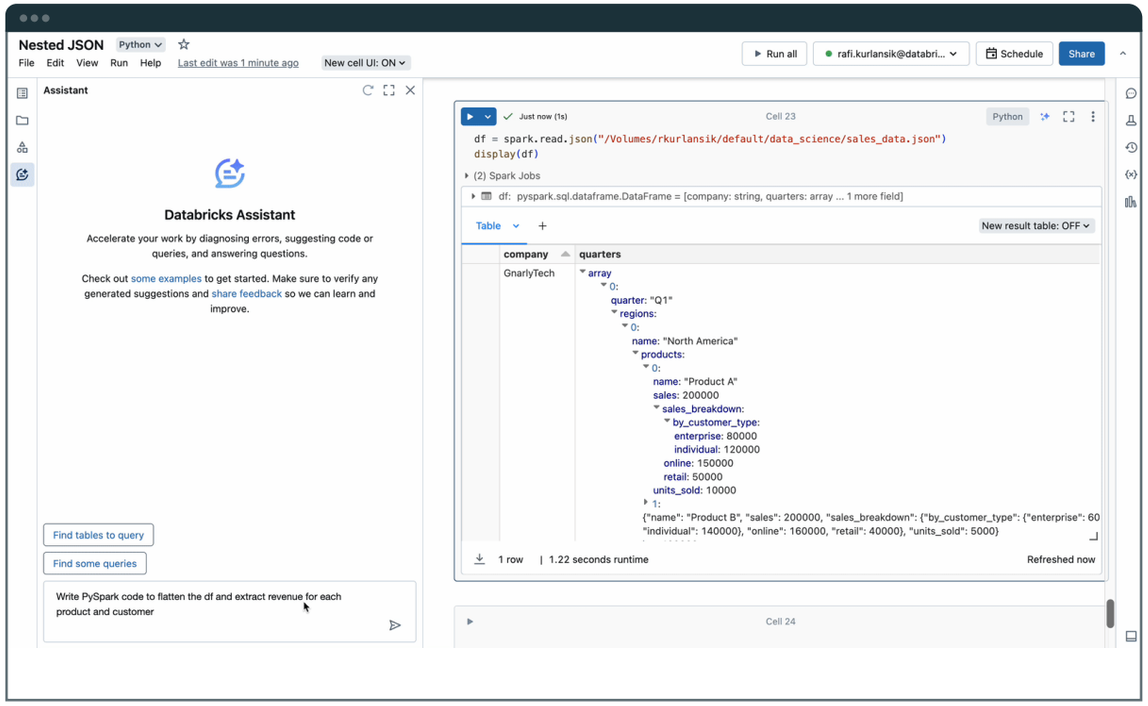
Nested Structs, Arrays (JSON, XML, etc)

When ingesting data via API, JSON files in storage, or noSQL databases, the resulting Spark DataFrames can be deeply nested and tricky to flatten correctly. Take a look at this mock sales data in JSON format:



Data engineers may be asked to flatten the nested array and extract revenue metrics for each product. Normally this task would take significant trial and error — even in a case where the data is relatively straightforward. The Assistant, however, being context-aware of the schemas of DataFrames you have in memory, generates code to get the job done. Using a simple prompt, we get the results we are looking for in seconds.

Write PySpark code to flatten the df and extract revenue for each product and customer



REFACTORING, DEBUGGING AND OPTIMIZATION

Another scenario data engineers face is rewriting code authored by other team members, either ones that may be more junior or have left the company. In these cases, the Assistant can analyze and explain poorly written code by understanding its context and intent. It can suggest more efficient algorithms, refactor code for better readability, and add comments.

Improving documentation and maintainability

This Python code calculates the total cost of items in an online shopping cart.

```
def calculate_total(cart_items):
    total = 0
    for i in range(len(cart_items)):
        if cart_items[i]['type'] == 'book':
            discount = cart_items[i]['price'] * 0.05
            total += cart_items[i]['price'] - discount
        else:
            total += cart_items[i]['price']
    return total

cart_items = [{'name': 'Python Programming', 'type': 'book', 'price': 50},
              {'name': 'Laptop', 'type': 'electronics', 'price': 800}]
total_price = calculate_total(cart_items)
```

The use of conditional blocks in this code makes it hard to read and inefficient at scale. Furthermore, there are no comments to explain what is happening. A good place to begin is to ask the Assistant to explain the code step by step. Once the data engineer understands the code, the Assistant can transform it, making it more performant and readable with the following prompt:

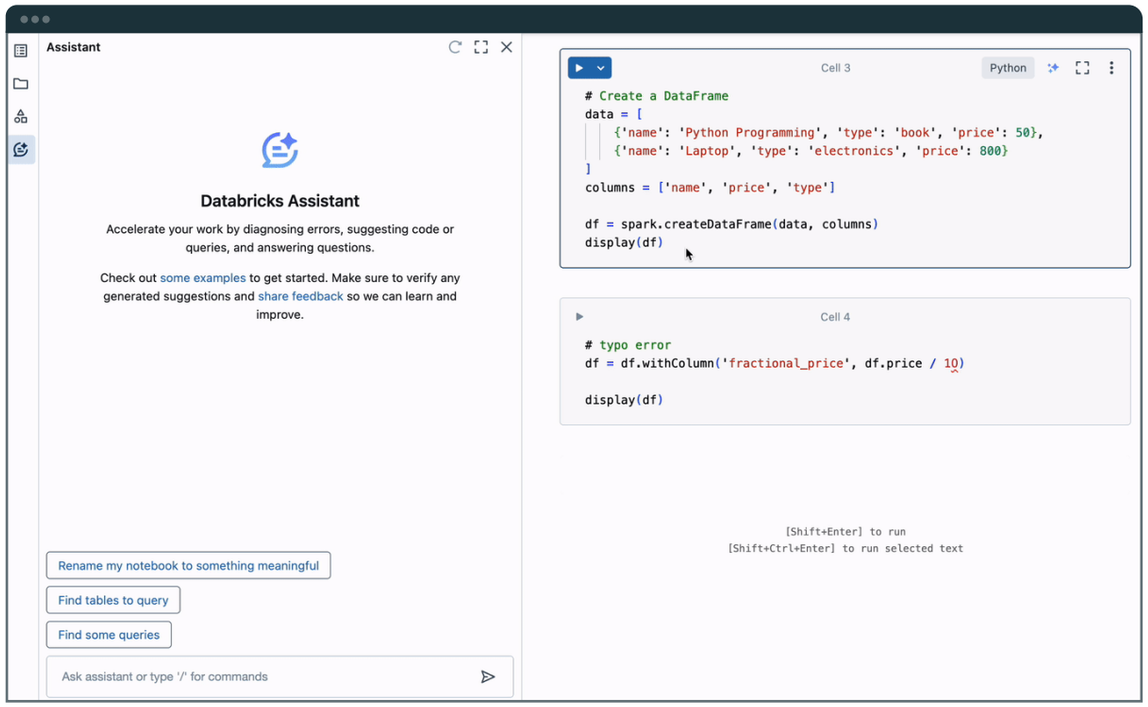
Rewrite this code in a way that is more performant, commented properly, and documented according to Python function documentation standards

The generated example below properly documents the code, and uses generator expressions instead of conditional blocks to improve memory utilization on larger datasets.

```
def calculate_total(cart_items):
    """
    Calculate the total price of items in the shopping cart.

    Apply a 5% discount to book items. The total is calculated by summing
    the prices, taking into account the discount for books.

    Parameters:
    cart_items (list of dict): A list of dictionaries, where each dictionary
                               represents an item in the cart with keys 'name',
                               'type', and 'price'.

    Returns:
    float: The total price of all items in the cart after discounts.
    """
    # Use a generator expression to iterate over cart items, applying a 5% discount
to books.
    return sum(item['price'] * 0.95 if item['type'] == 'book' else item['price'] for
item in cart_items)

# Example usage
cart_items = [{'name': 'Python Programming', 'type': 'book', 'price': 50},
              {'name': 'Laptop', 'type': 'electronics', 'price': 800}]
total_price = calculate_total(cart_items)
```

Diagnosing errors

Inevitably, data engineers will need to debug. The Assistant eliminates the need to open multiple browser tabs or switch contexts in order to identify the cause of errors in code, and staying focused is a tremendous productivity boost. To understand how this works with the Assistant, let's create a simple PySpark DataFrame and trigger an error.



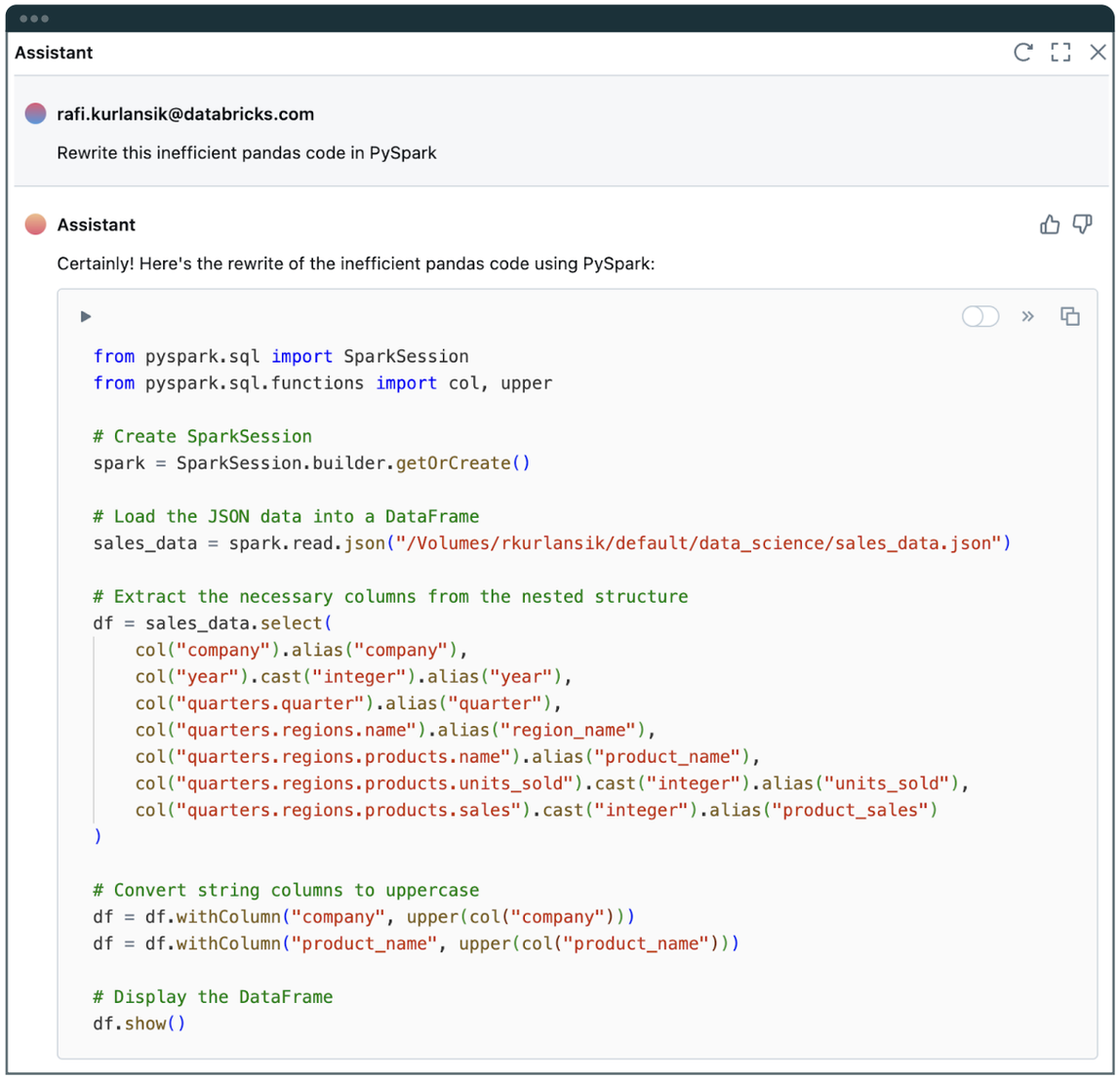
In the above example, a typo is introduced when adding a new column to the DataFrame. The zero in "10" is actually the letter "O", leading to an *invalid decimal literal* syntax error. The Assistant immediately offers to diagnose the error. It correctly identifies the typo, and suggests corrected code that can be inserted into the editor in the current cell. Diagnosing and correcting errors this way can save hours of time spent debugging.

Transpiling pandas to PySpark

Pandas is one of the most successful data-wrangling libraries in Python and is used by data scientists everywhere. Sticking with our JSON sales data, let's imagine a situation where a novice data scientist has done their best to flatten the data using pandas. It isn't pretty, it doesn't follow best practices, but it produces the correct output:

```python
import pandas as pd
import json

with open("/Volumes/rkurlansik/default/data_science/sales_data.json") as file:
    data = json.load(file)

# Bad practice: Manually initializing an empty DataFrame and using a deeply nested for-loop to populate it.
df = pd.DataFrame(columns=['company', 'year', 'quarter', 'region_name', 'product_name', 'units_sold', 'product_sales'])

for quarter in data['quarters']:
    for region in quarter['regions']:
        for product in region['products']:
            df = df.append({
                'company': data['company'],
                'year': data['year'],
                'quarter': quarter['quarter'],
                'region_name': region['name'],
                'product_name': product['name'],
                'units_sold': product['units_sold'],
                'product_sales': product['sales']
            }, ignore_index=True)

# Inefficient conversion of columns after data has been appended
df['year'] = df['year'].astype(int)
df['units_sold'] = df['units_sold'].astype(int)
df['product_sales'] = df['product_sales'].astype(int)

# Mixing access styles and modifying the dataframe in-place in an inconsistent manner
df['company'] = df.company.apply(lambda x: x.upper())
df['product_name'] = df['product_name'].str.upper()
```

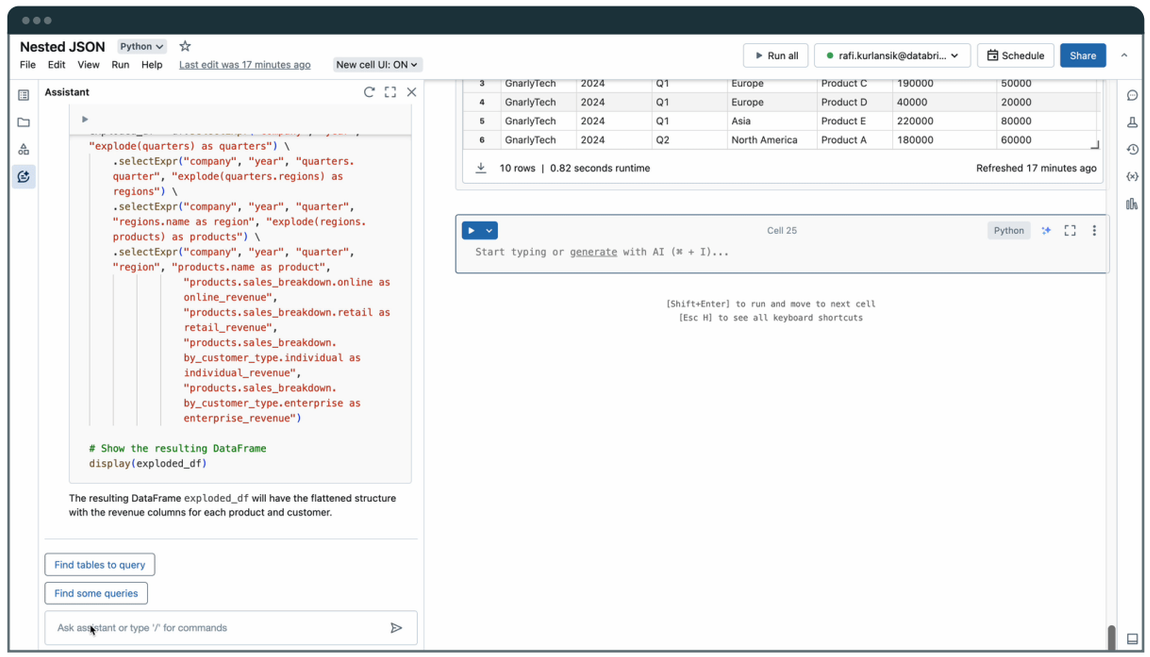
By default, Pandas is limited to running on a single machine. The data engineer shouldn't put this code into production and run it on billions of rows of data until it is converted to PySpark. This conversion process includes ensuring the data engineer understands the code and rewrites it in a way that is maintainable, testable, and performant. The Assistant once again comes up with a better solution in seconds.



Note the generated code includes creating a *SparkSession*, which isn't required in Databricks. Sometimes the Assistant, like any LLM, can be wrong or hallucinate. You, the data engineer, are the ultimate author of your code and it is important to review and understand any code generated before proceeding to the next task. If you notice this type of behavior, adjust your prompt accordingly.

WRITING TESTS

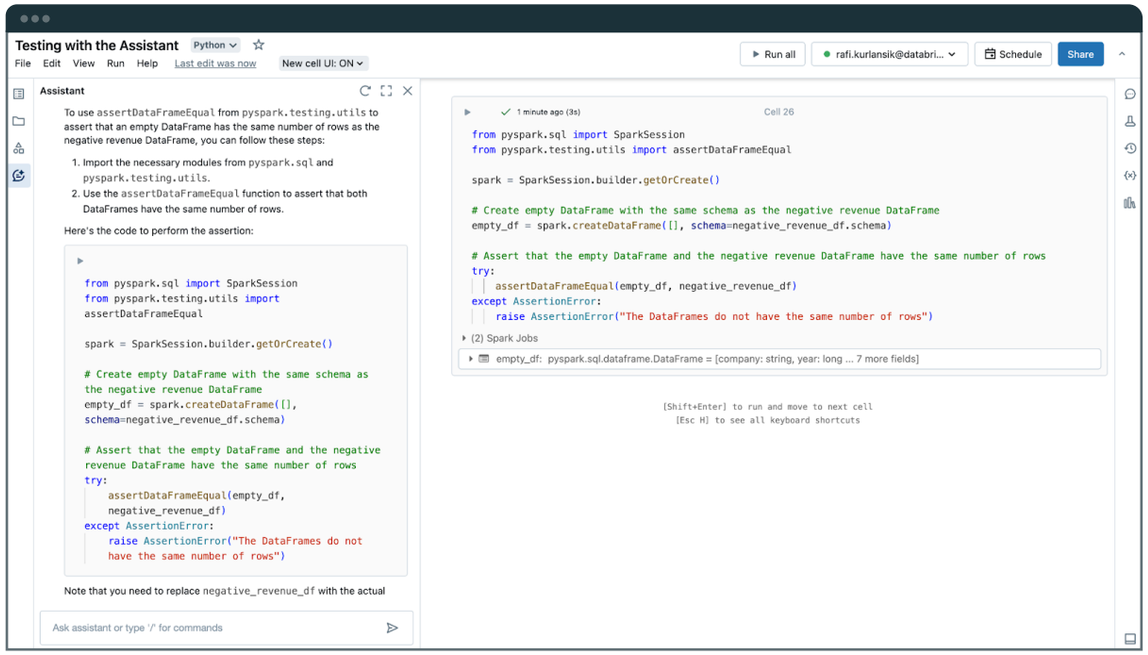
One of the most important steps in data engineering is to write tests to ensure your DataFrame transformation logic is correct, and to potentially catch any corrupted data flowing through your pipeline. Continuing with our example from the JSON sales data, the Assistant makes it a breeze to test if any of the revenue columns are negative - as long as values in the revenue columns are not less than zero, we can be confident that our data and transformations in this case are correct.



We can build off this logic by asking the Assistant to incorporate the test into PySpark's native testing functionality, using the following prompt:

Write a test using assertDataFrameEqual from pyspark.testing.utils to check that an empty DataFrame has the same number of rows as our negative revenue DataFrame.

The Assistant obliges, providing working code to bootstrap our testing efforts.

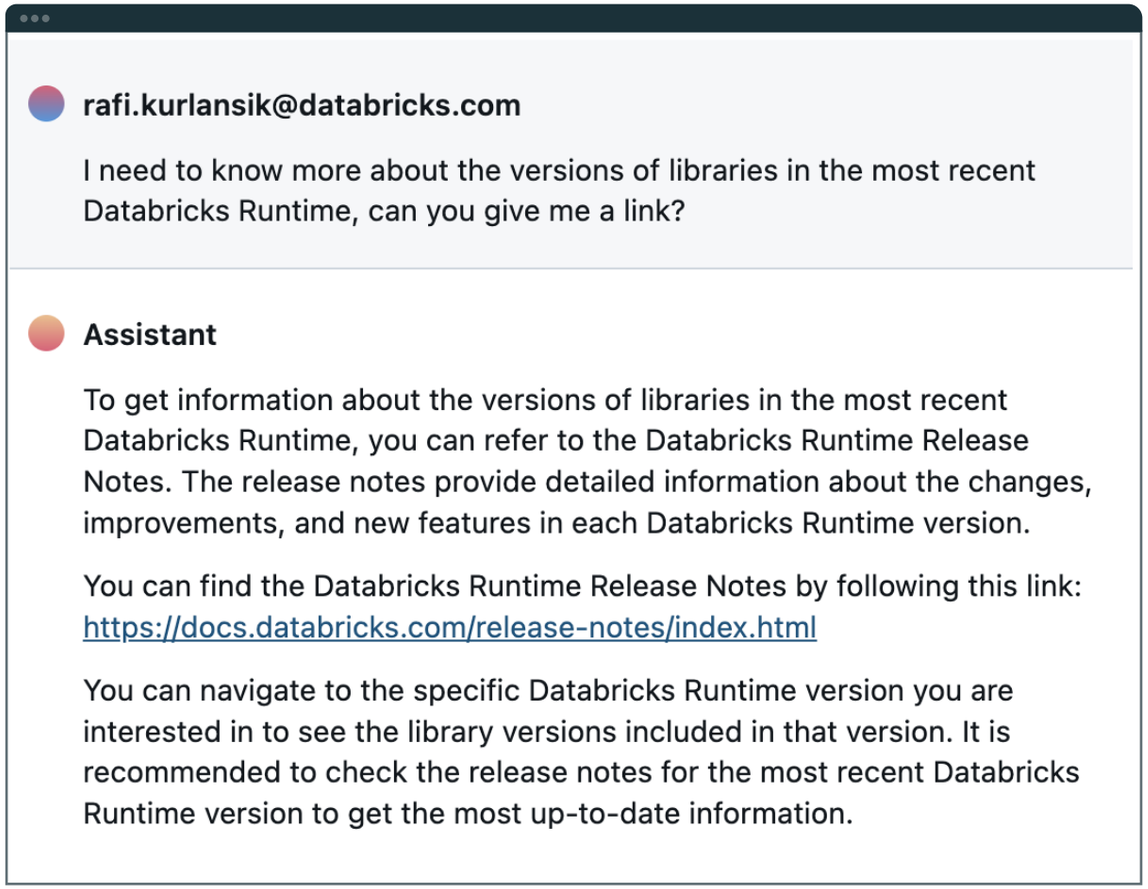


This example highlights the fact that being specific and adding detail to your prompt yields better results. If we simply ask the Assistant to write tests for us without any detail, our results will exhibit more variability in quality. Being specific and clear in what we are looking for — a test using PySpark modules that builds off the logic it wrote — generally will perform better than assuming the Assistant can correctly guess at our intentions.

GETTING HELP

Beyond a general capability to improve and understand code, the Assistant possesses knowledge of the entire Databricks documentation and Knowledge Base. This information is indexed on a regular basis and made available as additional context for the Assistant via a RAG architecture. This allows users to search for product functionality and configurations without leaving the Databricks Platform.

For example, if you want to know details about the system environment for the version of Databricks Runtime you are using, the Assistant can direct you to the appropriate page in the Databricks documentation.



The Assistant can handle simple, descriptive, and conversational questions, enhancing the user experience in navigating Databricks' features and resolving issues. It can even help guide users in filing support tickets! For more details, read the announcement article.

CONCLUSION

The barrier to entry for quality data engineering has been lowered thanks to the power of generative AI with the Databricks Assistant. Whether you are a newcomer looking for help on how to work with complex data structures or a seasoned veteran who wants regular expressions written for them, the Assistant will improve your quality of life. Its core competency of understanding, generating, and documenting code boosts productivity for data engineers of all skill levels. To learn more, see the Databricks documentation on how to get started with the Databricks Assistant today.

Applying Software Development and DevOps Best Practices to Delta Live Table Pipelines

by Alex Ott

Databricks Delta Live Tables (DLT) radically simplifies the development of the robust data processing pipelines by decreasing the amount of code that data engineers need to write and maintain. And also reduces the need for data maintenance and infrastructure operations, while enabling users to seamlessly promote code and pipelines configurations between environments. But people still need to perform testing of the code in the pipelines, and we often get questions on how people can do it efficiently.

In this chapter we'll cover the following items based on our experience working with multiple customers:

- How to apply DevOps best practices to Delta Live Tables
- How to structure the DLT pipeline's code to facilitate unit and integration testing
- How to perform unit testing of individual transformations of your DLT pipeline
- How to perform integration testing by executing the full DLT pipeline
- How to promote the DLT assets between stages
- How to put everything together to form a CI/CD pipeline (with Azure DevOps as an example)

APPLYING DEVOPS PRACTICES TO DLT: THE BIG PICTURE

The DevOps practices are aimed at shortening the software development life cycle (SDLC) providing the high quality at the same time. Typically they include these steps:

- Version control of the source code and infrastructure
- Code reviews
- Separation of environments (development/staging/production)
- Automated testing of individual software components and the whole product with the unit and integration tests
- Continuous integration (testing) and continuous deployment of changes (CI/CD)

All these practices can be applied to Delta Live Tables pipelines as well:

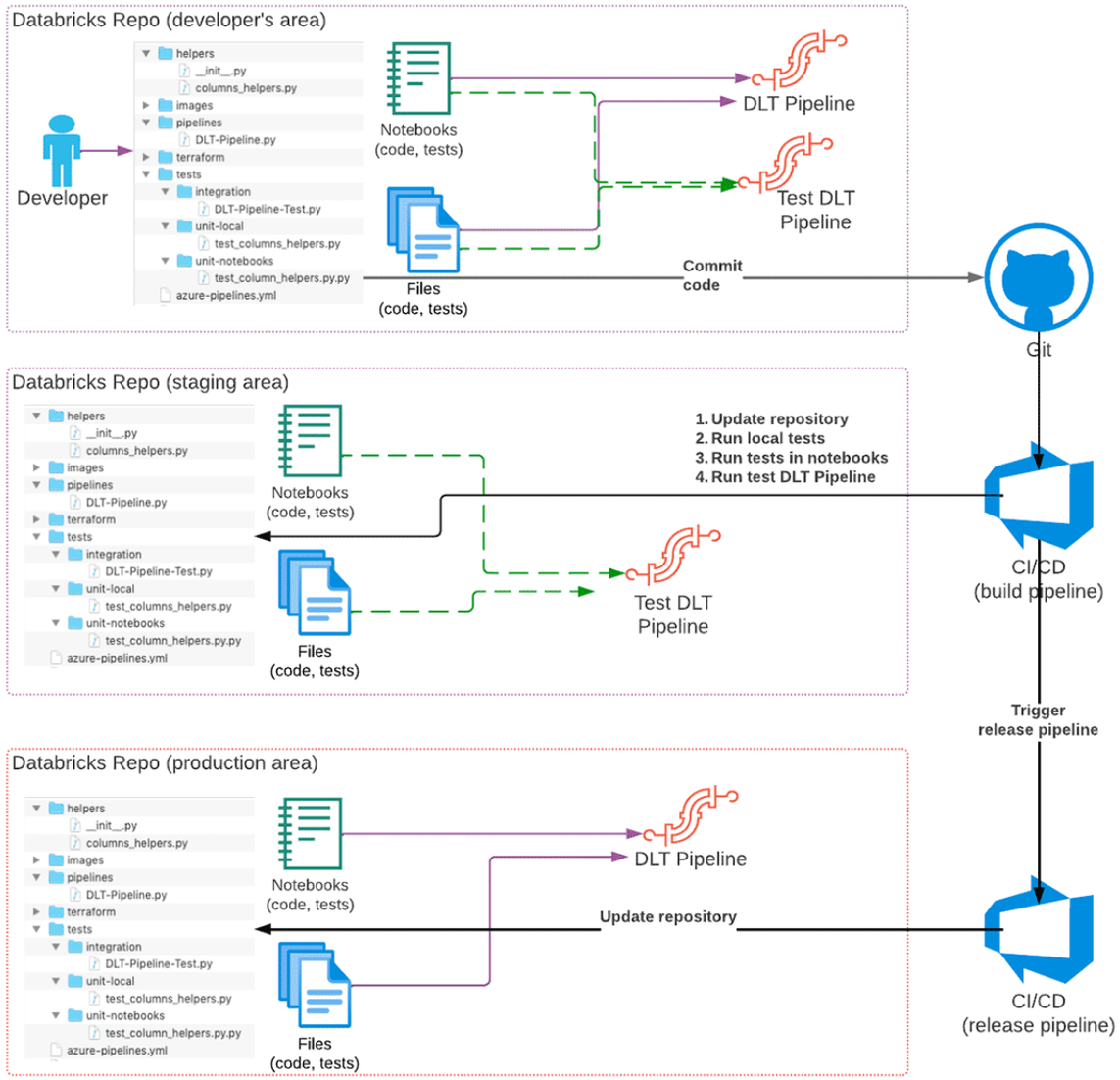
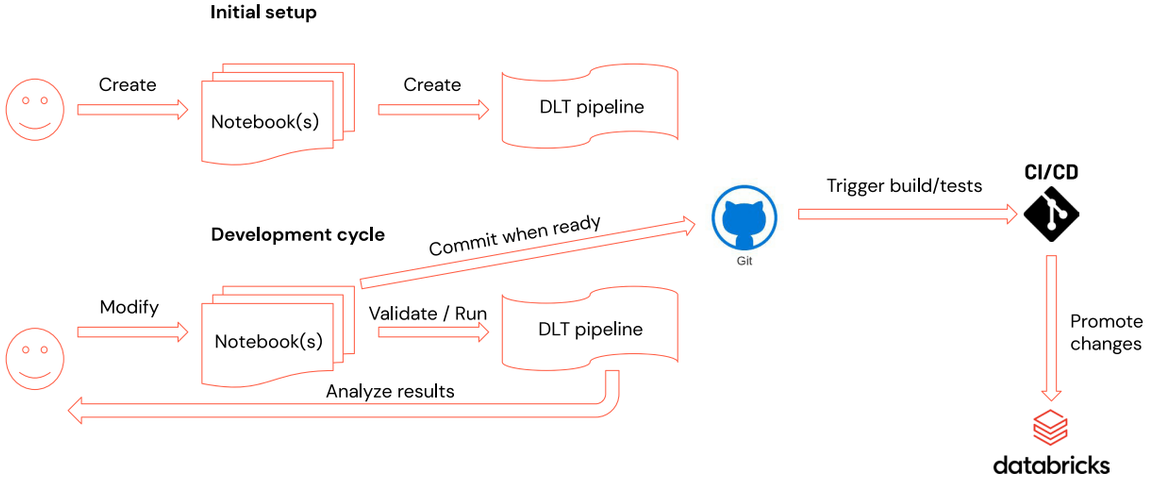


Figure: DLT development workflow

To achieve this we use the following features of Databricks product portfolio:

- Databricks Repos provide an interface to different Git services, so we can use them for code versioning, integration with CI/CD systems, and promotion of the code between environments
- Databricks CLI (or Databricks REST API) to implement CI/CD pipelines
- Databricks Terraform Provider for deployment of all necessary infrastructure and keeping it up to date

The recommended high-level development workflow of a DLT pipeline is as following:



1. A developer is developing the DLT code in their own checkout of a Git repository using a separate Git branch for changes.
2. When code is ready and tested, code is committed to Git and a pull request is created.
3. CI/CD system reacts to the commit and starts the build pipeline (CI part of CI/CD) that will update a staging Databricks Repo with the changes, and trigger execution of unit tests.
    a. Optionally, the integration tests could be executed as well, although in some cases this could be done only for some branches, or as a separate pipeline.
4. If all tests are successful and code is reviewed, the changes are merged into the main (or a dedicated branch) of the Git repository.
5. Merging of changes into a specific branch (for example, releases) may trigger a release pipeline (CD part of CI/CD) that will update the Databricks Repo in the production environment, so code changes will take effect when pipeline runs next time.

As illustration for the rest of the chapter we'll use a very simple DLT pipeline consisting just of two tables, illustrating typical Bronze/Silver layers of a typical lakehouse architecture. Complete source code together with deployment instructions is available on GitHub.

Figure: Example DLT pipeline

Note: DLT provides both SQL and Python APIs, in most of the chapter we focus on Python implementation, although we can apply most of the best practices also for SQL-based pipelines.

DEVELOPMENT CYCLE WITH DELTA LIVE TABLES

When developing with Delta Live Tables, typical development process looks as follows:

1. Code is written in the notebook(s).
2. When another piece of code is ready, a user switches to DLT UI and starts the pipeline. (To make this process faster it's recommended to run the pipeline in the Development mode, so you don't need to wait for resources again and again).
3. When a pipeline is finished or failed because of the errors, the user analyzes results, and adds/modifies the code, repeating the process.
4. When code is ready, it's committed.

For complex pipelines, such dev cycle could have a significant overhead because the pipeline's startup could be relatively long for complex pipelines with dozens of tables/views and when there are many libraries attached. For users it would be easier to get very fast feedback by evaluating the individual transformations and testing them with sample data on interactive clusters.

STRUCTURING THE DLT PIPELINE'S CODE

To be able to evaluate individual functions and make them testable it's very important to have correct code structure. Usual approach is to define all data transformations as individual functions receiving and returning Spark DataFrames, and call these functions from DLT pipeline functions that will form the DLT execution graph. The best way to achieve this is to use files in repos functionality that allows to expose Python files as normal Python modules that could be imported into Databricks notebooks or other Python code. DLT natively supports files in repos that allows importing Python files as Python modules (please note, that when using files in repos, the two entries are added to the Python's sys.path — one for repo root, and one for the current directory of the caller notebook). With this, we can start to write our code as a separate Python file located in the dedicated folder under the repo root that will be imported as a Python module:

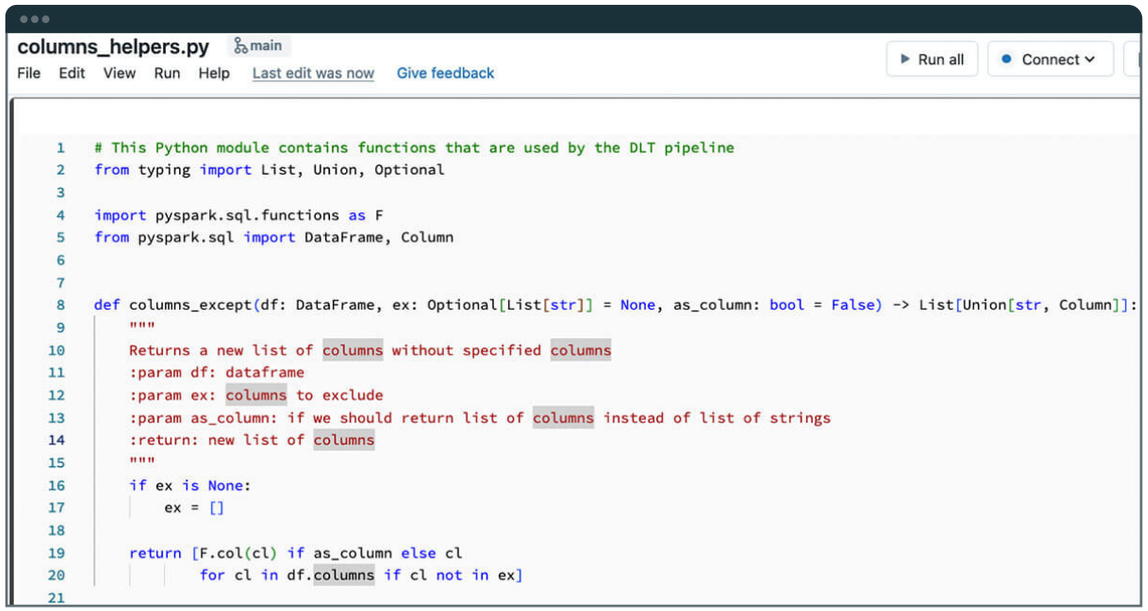


Figure: Source code for a Python package

And the code from this Python package could be used inside the DLT pipeline code:

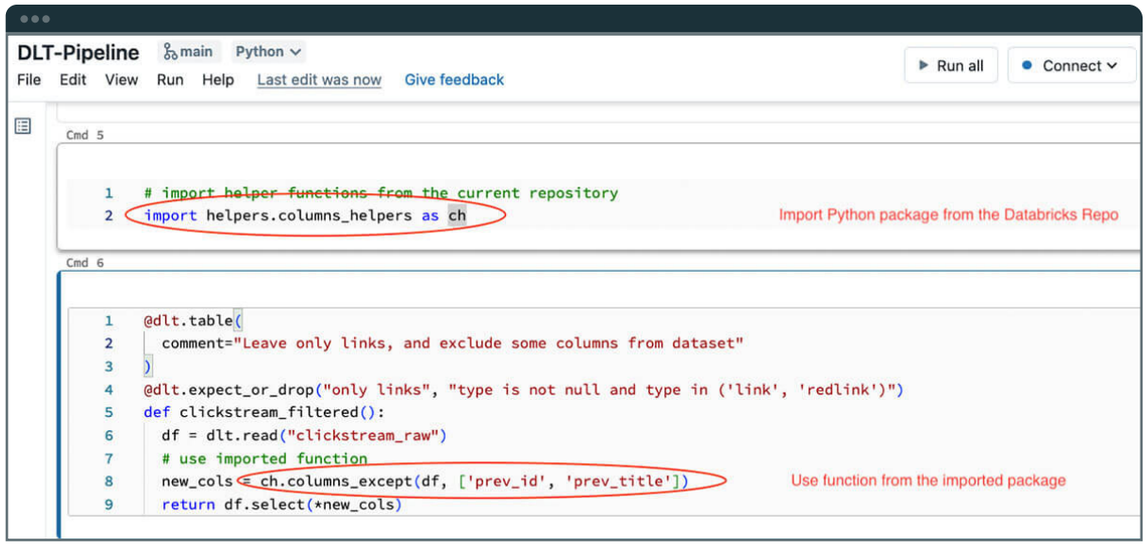


Figure: Using functions from the Python package in the DLT code

Note, that function in this particular DLT code snippet is very small — all it's doing is just reading data from the upstream table, and applying our transformation defined in the Python module. With this approach we can make DLT code simpler to understand and easier to test locally or using a separate notebook attached to an interactive cluster. Splitting the transformation logic into a separate Python module allows us to interactively test transformations from notebooks, write unit tests for these transformations and also test the whole pipeline (we'll talk about testing in the next sections).

The final layout of the Databricks Repo, with unit and integration tests, may look as following:

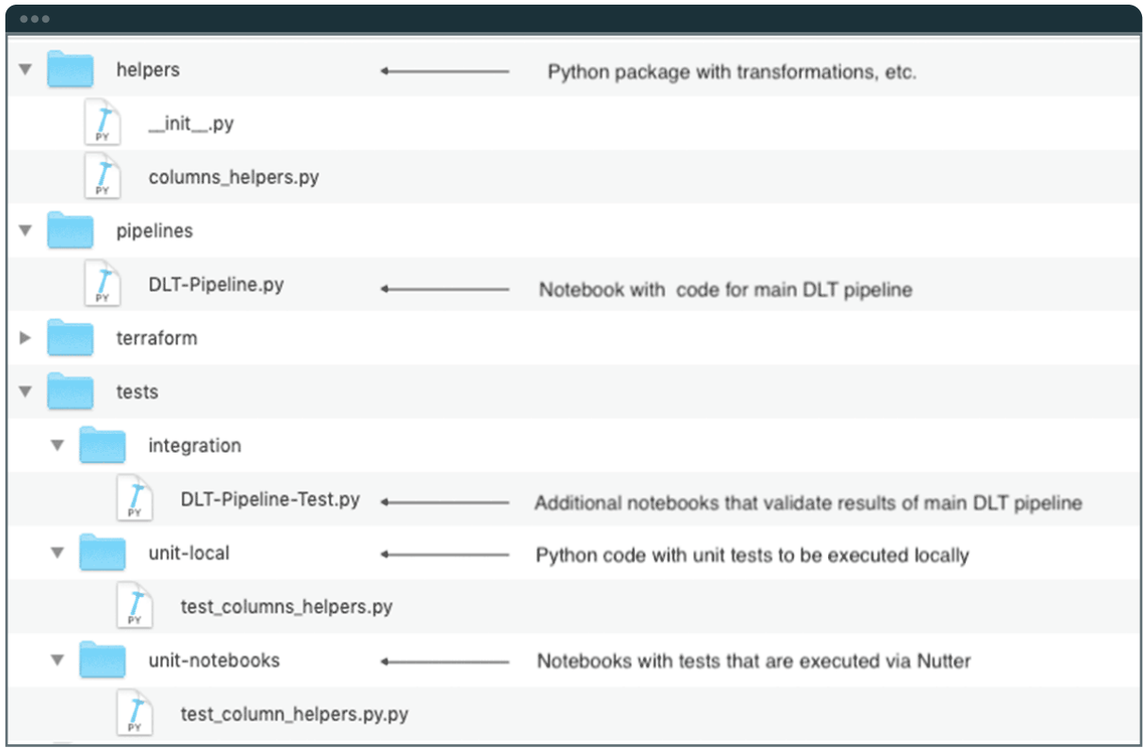


Figure: Recommended code layout in Databricks Repo

This code structure is especially important for bigger projects that may consist of the multiple DLT pipelines sharing the common transformations.

IMPLEMENTING UNIT TESTS

As mentioned above, splitting transformations into a separate Python module allows us easier write unit tests that will check behavior of the individual functions. We have a choice of how we can implement these unit tests:

- We can define them as Python files that could be executed locally, for example, using pytest. This approach has following advantages:
  - We can develop and test these transformations using the IDE, and for example, sync the local code with Databricks repo using the Databricks extension for Visual Studio Code or dbx sync command if you use another IDE
  - Such tests could be executed inside the CI/CD build pipeline without need to use Databricks resources (although it may depend if some Databricks-specific functionality is used or the code could be executed with PySpark)
  - We have access to more development related tools — static code and code coverage analysis, code refactoring tools, interactive debugging, etc.
  - We can even package our Python code as a library, and attach to multiple projects
- We can define them in the notebooks — with this approach:
  - We can get feedback faster as we always can run sample code and tests interactively
  - We can use additional tools like Nutter to trigger execution of notebooks from the CI/CD build pipeline (or from the local machine) and collect results for reporting

The demo repository contains a sample code for both of these approaches — for local execution of the tests, and executing tests as notebooks. The CI pipeline shows both approaches.

Please note that both of these approaches are applicable only to the Python code — if you're implementing your DLT pipelines using SQL, then you need to follow the approach described in the next section.

IMPLEMENTING INTEGRATION TESTS

While unit tests give us assurance that individual transformations are working as they should, we still need to make sure that the whole pipeline also works. Usually this is implemented as an integration test that runs the whole pipeline, but usually it's executed on the smaller amount of data, and we need to validate execution results. With Delta Live Tables, there are multiple ways to implement integration tests:

- Implement it as a Databricks Workflow with multiple tasks — similarly what is typically done for non-DLT code
- Use DLT expectations to check pipeline's results

IMPLEMENTING INTEGRATION TESTS WITH DATABRICKS WORKFLOWS

In this case we can implement integration tests with Databricks Workflows with multiple tasks (we can even pass data, such as, data location, etc. between tasks using task values). Typically such a workflow consists of the following tasks:

- Setup data for DLT pipeline
- Execute pipeline on this data
- Perform validation of produced results.

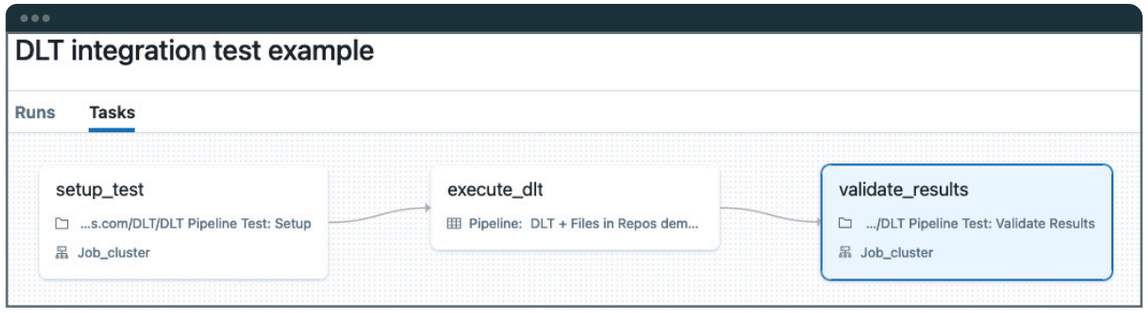


Figure: Implementing integration test with Databricks Workflows

The main drawback of this approach is that it requires writing quite a significant amount of the auxiliary code for setup and validation tasks, plus it requires additional compute resources to execute the setup and validation tasks.

USE DLT EXPECTATIONS TO IMPLEMENT INTEGRATION TESTS

We can implement integration tests for DLT by expanding the DLT pipeline with additional DLT tables that will apply DLT expectations to data using the fail operator to fail the pipeline if results don't match to provided expectations. It's very easy to implement - just create a separate DLT pipeline that will include additional notebook(s) that define DLT tables with expectations attached to them.

For example, to check that silver table includes only allowed data in the type column we can add following DLT table and attach expectations to it:

```
1  @dlt.table(comment="Check type")
2  @dlt.expect_all_or_fail({"valid type": "type in ('link', 'redlink')",
                            "type is not null": "type is not null"})
3  def filtered_type_check():
4    return dlt.read("clickstream_filtered").select("type")
```

Resulting DLT pipeline for integration test may look as following (we have two additional tables in the execution graph that check that data is valid):

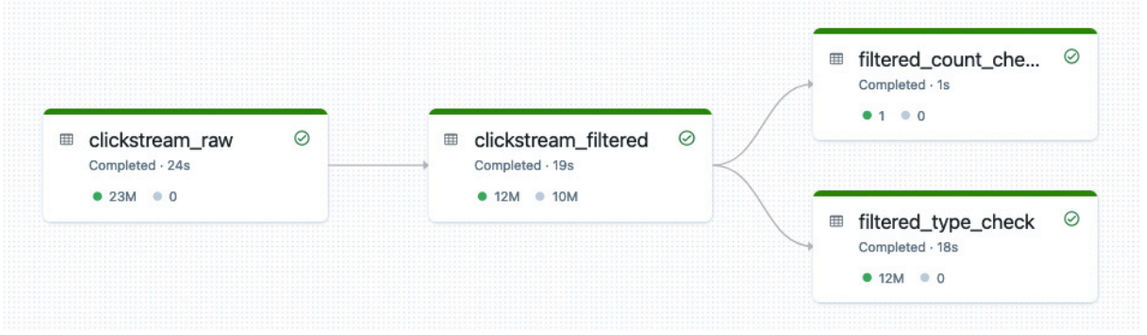


Figure: Implementing integration tests using DLT expectations

This is the recommended approach to performing integration testing of DLT pipelines. With this approach, we don't need any additional compute resources - everything is executed in the same DLT pipeline, so get cluster reuse, all data is logged into the DLT pipeline's event log that we can use for reporting, etc.

Please refer to DLT documentation for more examples of using DLT expectations for advanced validations, such as, checking uniqueness of rows, checking presence of specific rows in the results, etc. We can also build libraries of DLT expectations as shared Python modules for reuse between different DLT pipelines.

PROMOTING THE DLT ASSETS BETWEEN ENVIRONMENTS

When we're talking about promotion of changes in the context of DLT, we're talking about multiple assets:

- Source code that defines transformations in the pipeline
- Settings for a specific Delta Live Tables pipeline

The simplest way to promote the code is to use Databricks Repos to work with the code stored in the Git repository. Besides keeping your code versioned, Databricks Repos allows you to easily propagate the code changes to other environments using the Repos REST API or Databricks CLI.

From the beginning, DLT separates code from the pipeline configuration to make it easier to promote between stages by allowing to specify the schemas, data locations, etc. So we can define a separate DLT configuration for each stage that will use the same code, while allowing you to store data in different locations, use different cluster sizes, etc.

To define pipeline settings we can use Delta Live Tables REST API or Databricks CLI's pipelines command, but it becomes difficult in case you need to use instance pools, cluster policies, or other dependencies. In this case the more flexible alternative is Databricks Terraform Provider's databricks_pipeline resource that allows easier handling of dependencies to other resources, and we can use Terraform modules to modularize the Terraform code to make it reusable. The provided code repository contains examples of the Terraform code for deploying the DLT pipelines into the multiple environments.

PUTTING EVERYTHING TOGETHER TO FORM A CI/CD PIPELINE

After we implemented all the individual parts, it's relatively easy to implement a CI/CD pipeline. GitHub repository includes a build pipeline for Azure DevOps (other systems could be supported as well — the differences are usually in the file structure). This pipeline has two stages to show ability to execute different sets of tests depending on the specific event:

- onPush is executed on push to any Git branch except releases branch and version tags. This stage only runs and reports unit tests results (both local and notebooks).
- onRelease is executed only on commits to the releases branch, and in addition to the unit tests it will execute a DLT pipeline with integration test.

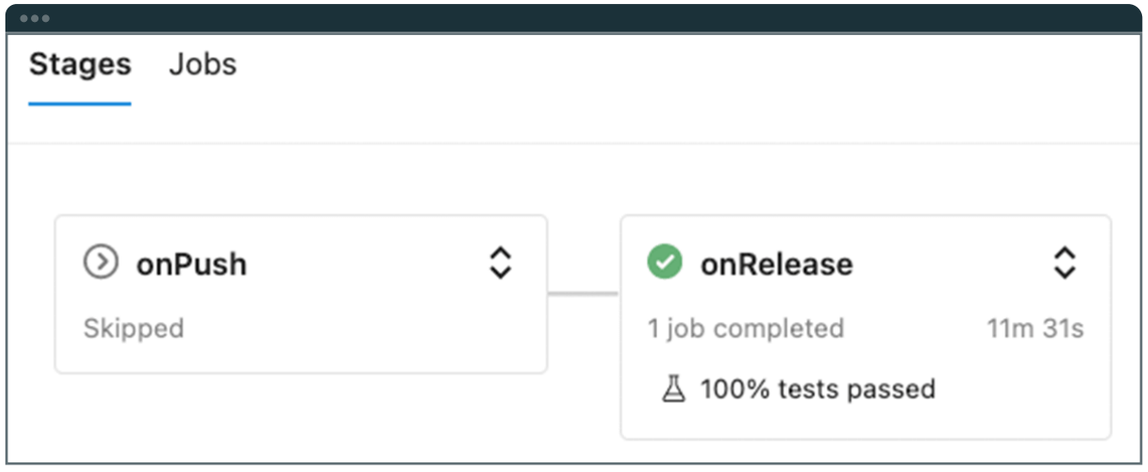
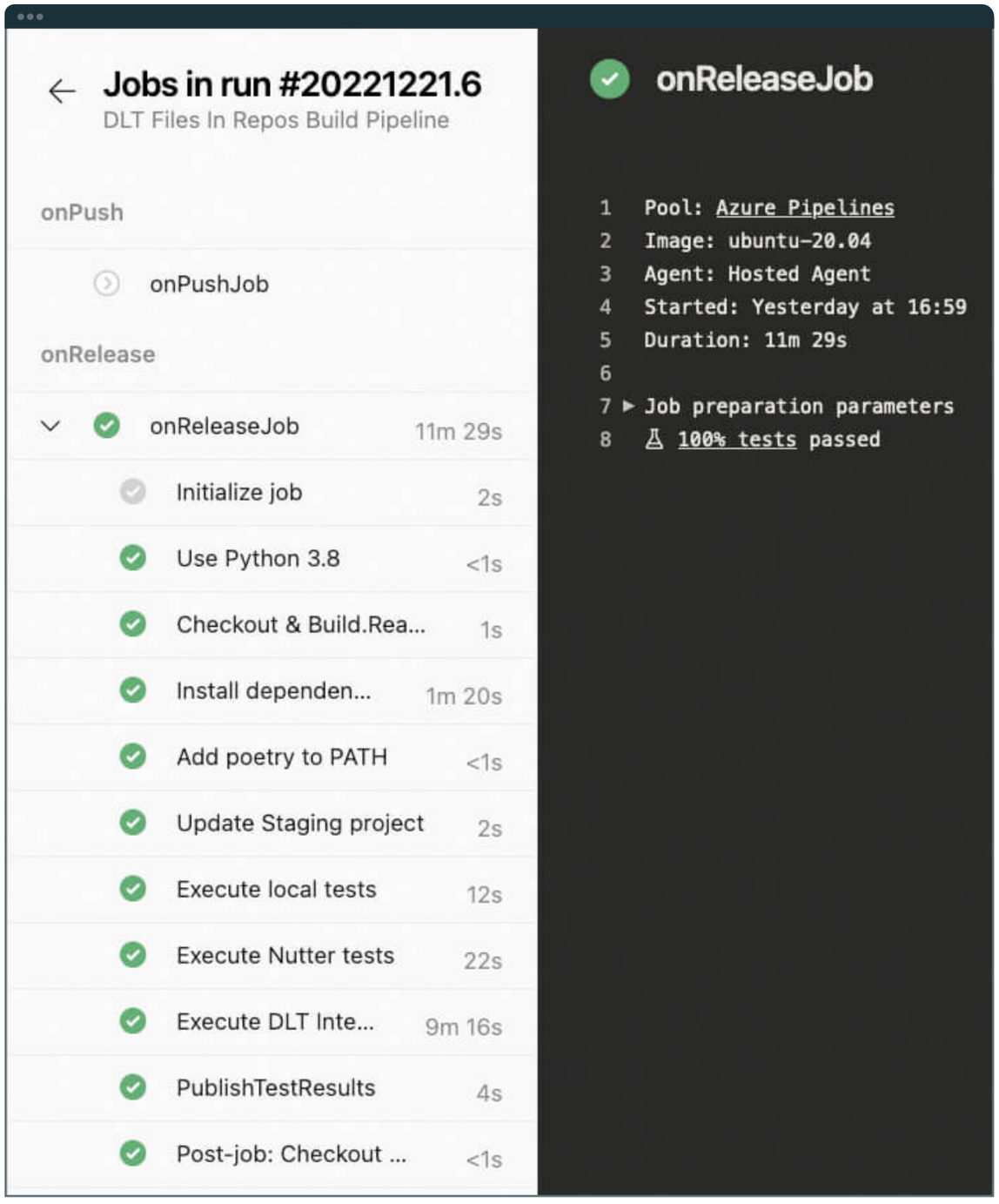


Figure: Structure of Azure DevOps build pipeline

Except for the execution of the integration test in the onRelease stage, the structure of both stages is the same — it consists of following steps:

1. Checkout the branch with changes
2. Set up environment — install Poetry which is used for managing Python environment management, and installation of required dependencies
3. Update Databricks Repos in the staging environment
4. Execute local unit tests using the PySpark
5. Execute the unit tests implemented as Databricks notebooks using Nutter
6. For releases branch, execute integration tests
7. Collect test results and publish them to Azure DevOps



Results of tests execution are reported back to the Azure DevOps, so we can track them:

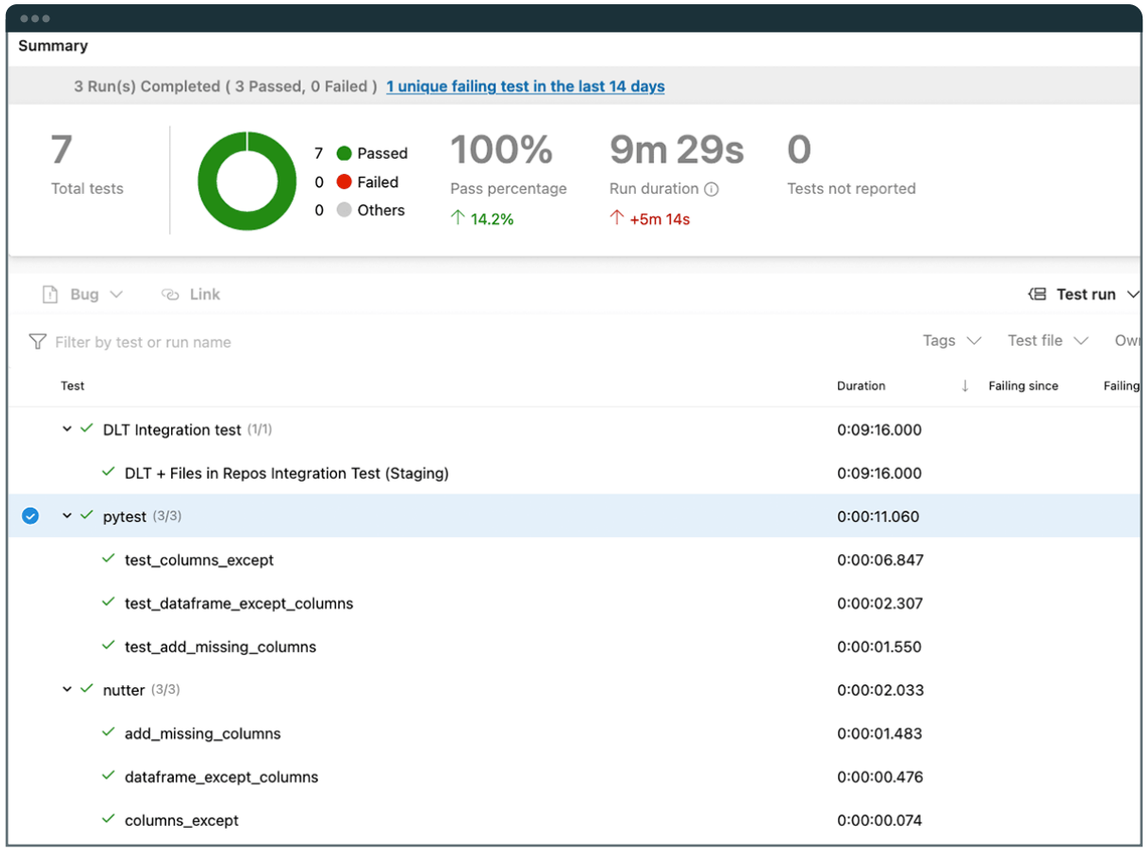


Figure: Reporting the tests execution results

If commits were done to the releases branch and all tests were successful, the release pipeline could be triggered, updating the production Databricks repo, so changes in the code will be taken into account on the next run of DLT pipeline.

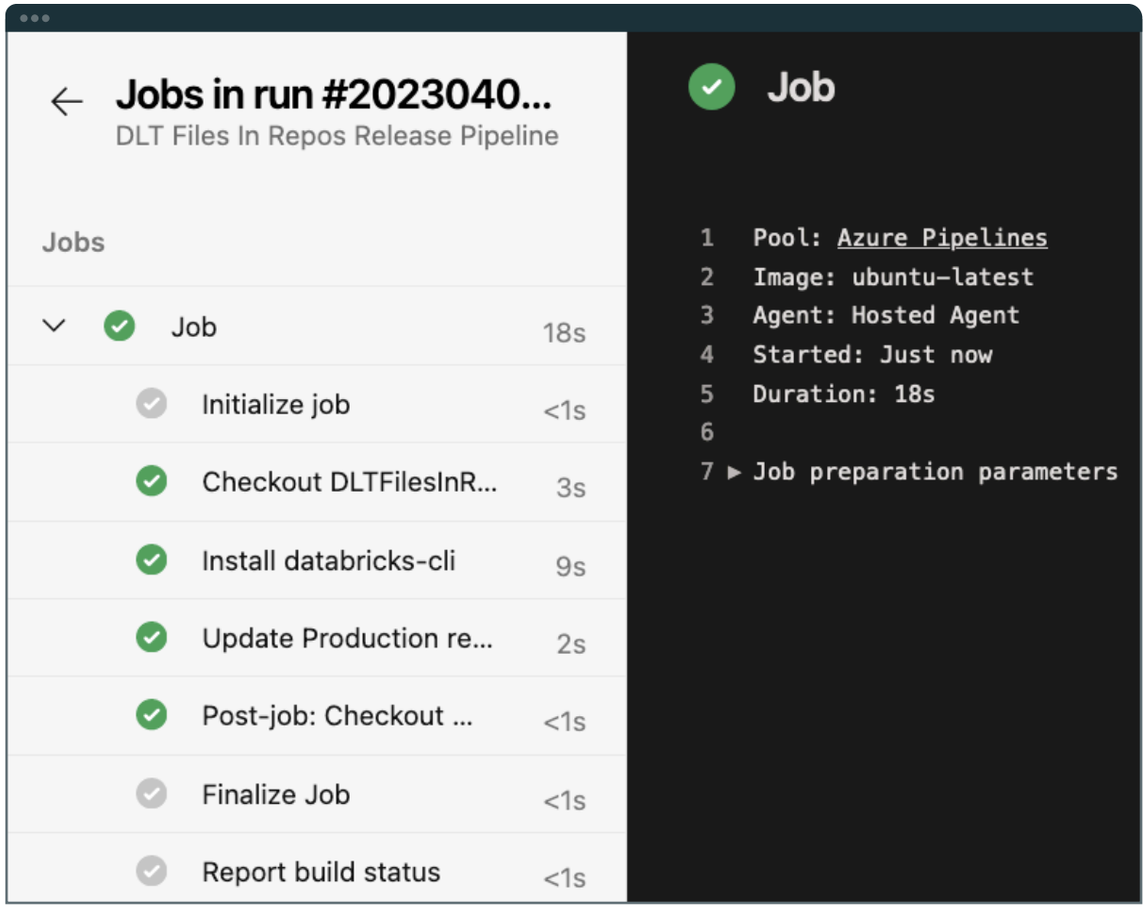


Figure: Release pipeline to deploy code changes to production DLT pipeline

Try to apply approaches described in this chapter to your Delta Live Table pipelines! The provided demo repository contains all necessary code together with setup instructions and Terraform code for deployment of everything to Azure DevOps.

Unity Catalog Governance in Action: Monitoring, Reporting and Lineage

by Ari Kaplan and Pearl Ubaru

Databricks Unity Catalog (UC) provides a single unified governance solution for all of a company's data and AI assets across clouds and data platforms. This chapter digs deeper into the prior Unity Catalog Governance Value Levers blog to show how the technology itself specifically enables positive business outcomes through comprehensive data and AI monitoring, reporting, and lineage.

OVERALL CHALLENGES WITH TRADITIONAL NON-UNIFIED GOVERNANCE

The Unity Catalog Governance Value Levers blog discussed the "why" of the organizational importance of governance for information security, access control, usage monitoring, enacting guardrails, and obtaining "single source of truth" insights from their data assets. These challenges compound as their company grows and without Databricks UC, traditional governance solutions no longer adequately meet their needs.

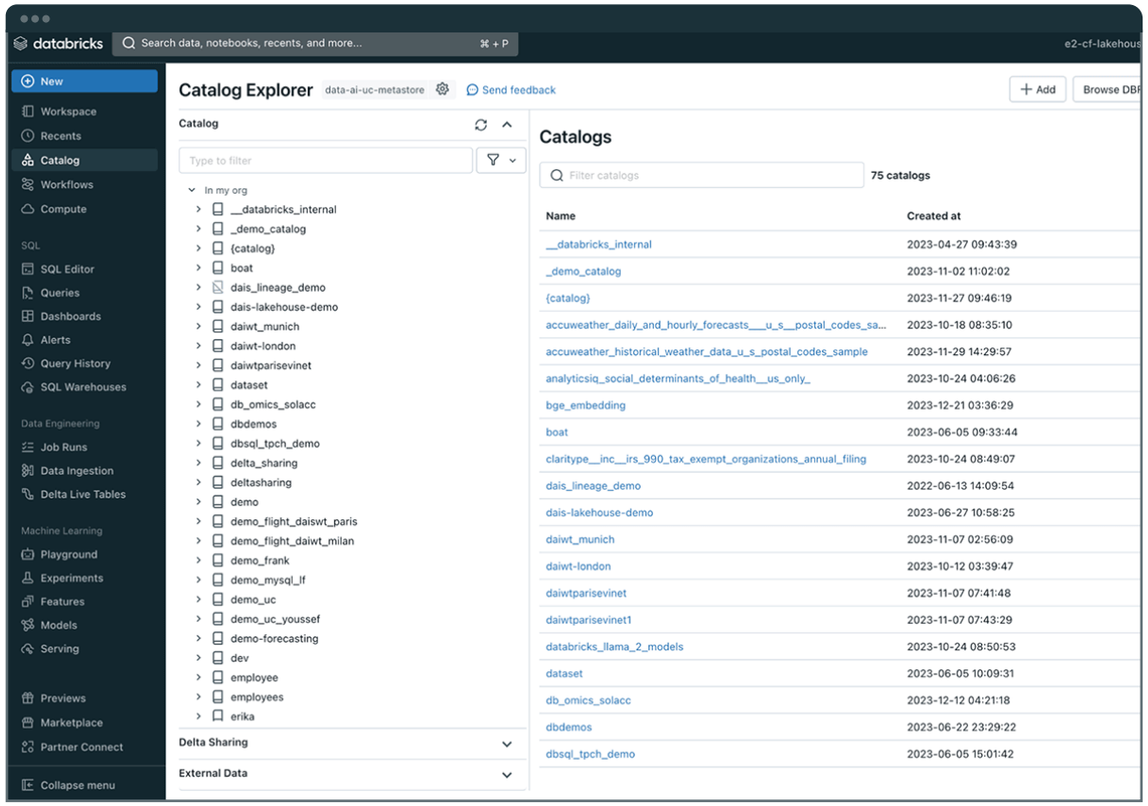
The major challenges discussed included weaker compliance and fractured data privacy controlled across multiple vendors; uncontrolled and siloed data and AI swamps; exponentially rising costs; loss of opportunities, revenue, and collaboration.

HOW DATABRICKS UNITY CATALOG SUPPORTS A UNIFIED VIEW, MONITORING, AND OBSERVABILITY

So, how does this all work from a technical standpoint? UC manages all registered assets across the Databricks Data Intelligence Platform. These assets can be anything within BI, DW, data engineering, data streaming, data science, and ML. This governance model provides access controls, lineage, discovery, monitoring, auditing, and sharing. It also provides metadata management of files, tables, ML models, notebooks, and dashboards. UC gives one single view of your entire end-to-end information, through the Databricks asset catalog, feature store and model registry, lineage capabilities, and metadata tagging for data classifications, as discussed below:

Unified view of the entire data estate

- Asset catalog: through system tables that contain metadata, you can see all that is contained in your catalog such as schemas, tables, columns, files, models, and more. If you are not familiar with volumes within Databricks, they are used for managing non-tabular datasets. Technically, they are logical volumes of storage to access files in any format: structured, semi-structured, and unstructured.

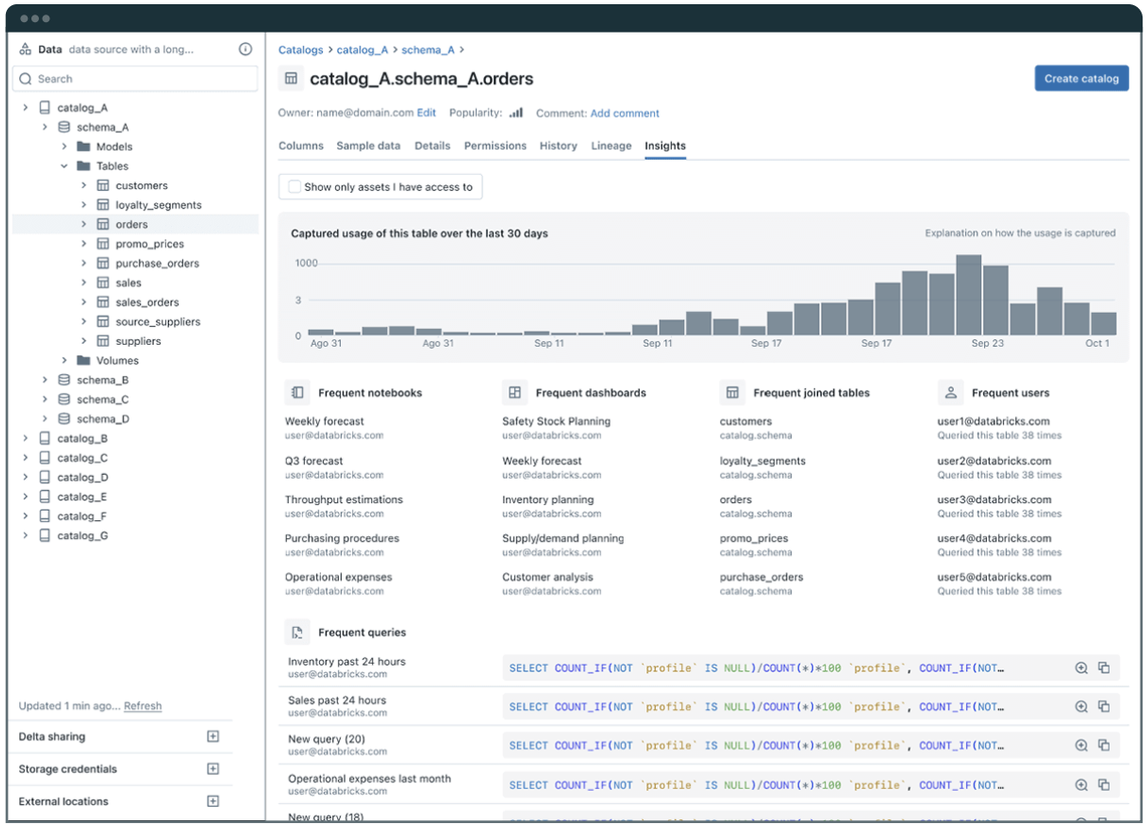


Catalog Explorer lets you discover and govern all your data and ML models

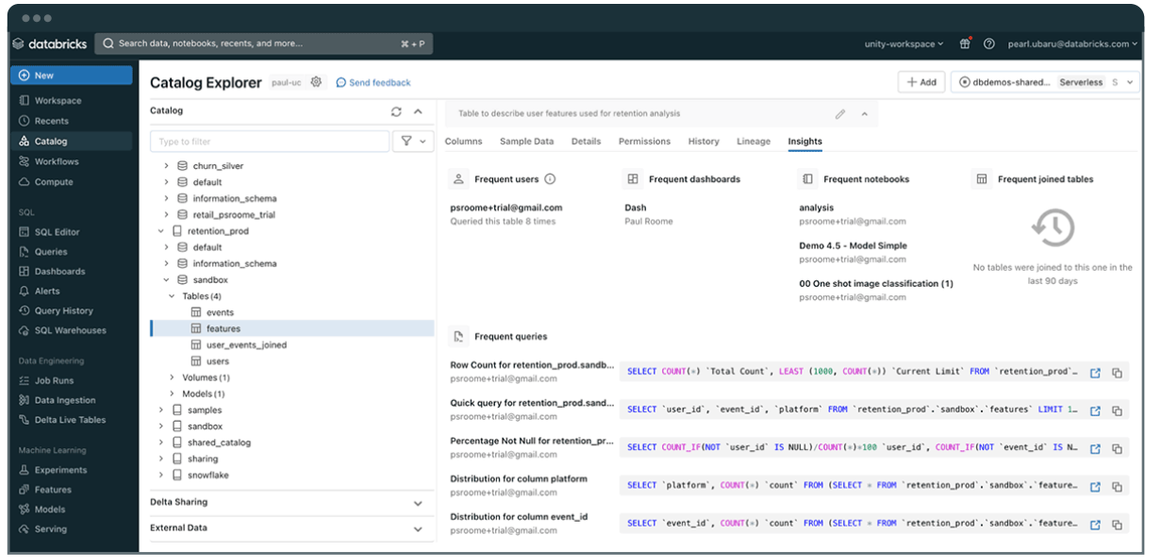
- Feature Store and Model Registry: define features used by data scientists within the centralized repository. This is helpful for consistent model training and inference for your entire AI workflow.
- Lineage capabilities: trust in your data is key for your business to take action in real life. End-to-end transparency into your data is needed for trust in your reports, models, and insights. UC makes this easy through lineage capabilities, providing insights on: What are the raw data sources? Who created it and when? How was data merged and transformed? What is the traceability from the models back to the datasets they are trained on? Lineage shows end-to-end from data to model - both table-level and column-level. You can even query across data sources such as Snowflake and benefit immediately:

Data sources can be across platforms such as Snowflake and Databricks

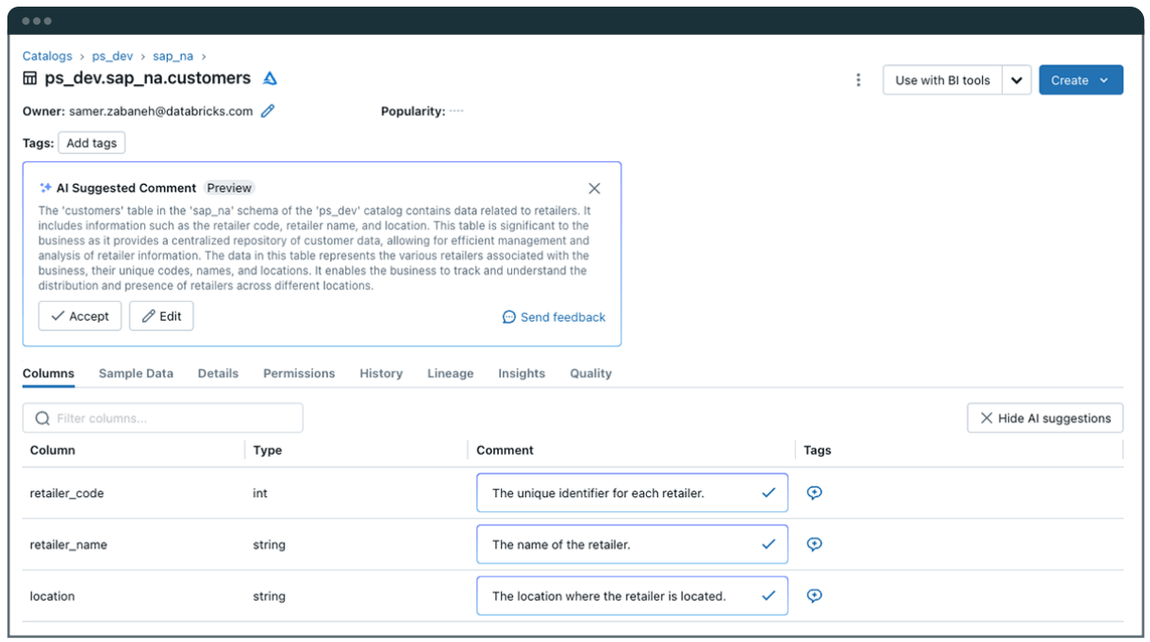
- Metadata tagging for data classifications: enrich your data and queries by providing contextual insights about your data assets. These descriptions at the column and table level can be manually entered, or automatically described with GenAI by Databricks Assistant. Below is an example of descriptions and quantifiable characteristics:



Metadata tagging insights: frequent users, notebooks, queries, joins, billing trends and more



Metadata tagging insights: details on the “features” table



Databricks Assistant uses GenAI to write context-aware descriptions of columns and tables

Having one unified view results in:

- **Accelerated innovation:** your insights are only as good as your data. Your analysis is only as good as the data you access. So, streamlining your data search drives faster and better generation of business insights, driving innovation.
- **Cost reduction through centralized asset cataloging:** lowers license costs (just one vendor solution versus needing many vendors), lowers usage fees, reduces time to market pains, and enables overall operational efficiencies.
- **It's easier to discover and access all data** by reducing data sprawl across several databases, data warehouses, object storage systems, and more.

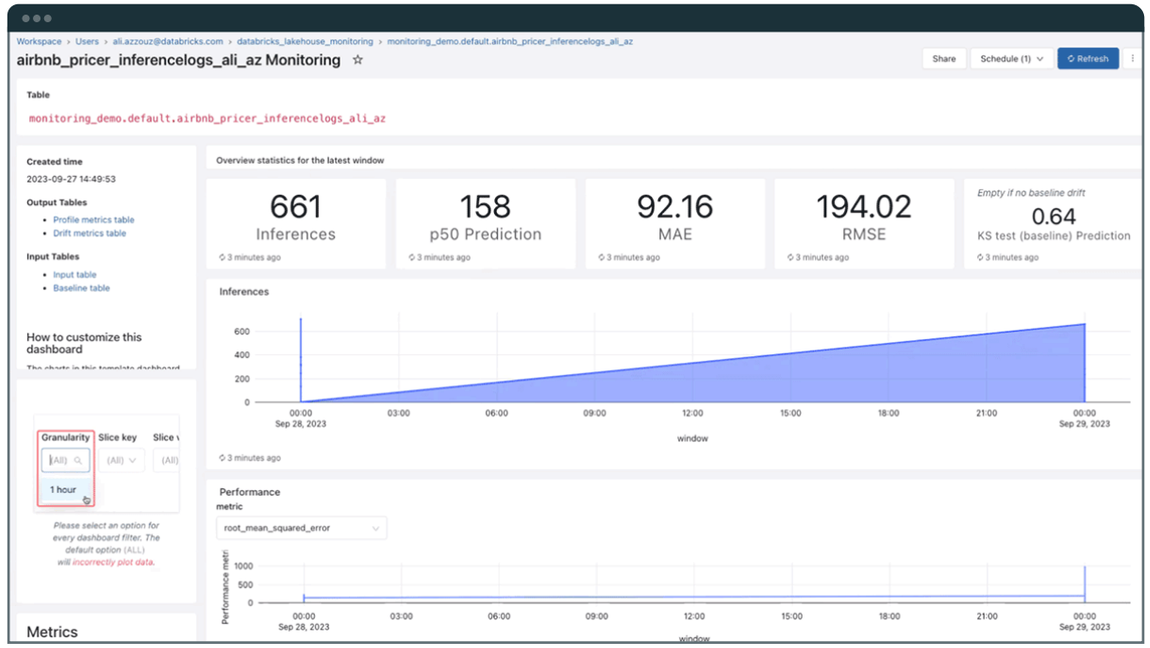
COMPREHENSIVE DATA AND AI MONITORING AND REPORTING

Databricks Lakehouse Monitoring allows teams to monitor their entire data pipelines — from data and features to ML models — without additional tools and complexity. Powered by Unity Catalog, it lets users uniquely ensure that their data and AI assets are high quality, accurate and reliable through deep insight into the lineage of their data and AI assets. The single, unified approach to monitoring enabled by lakehouse architecture makes it simple to diagnose errors, perform root cause analysis, and find solutions.

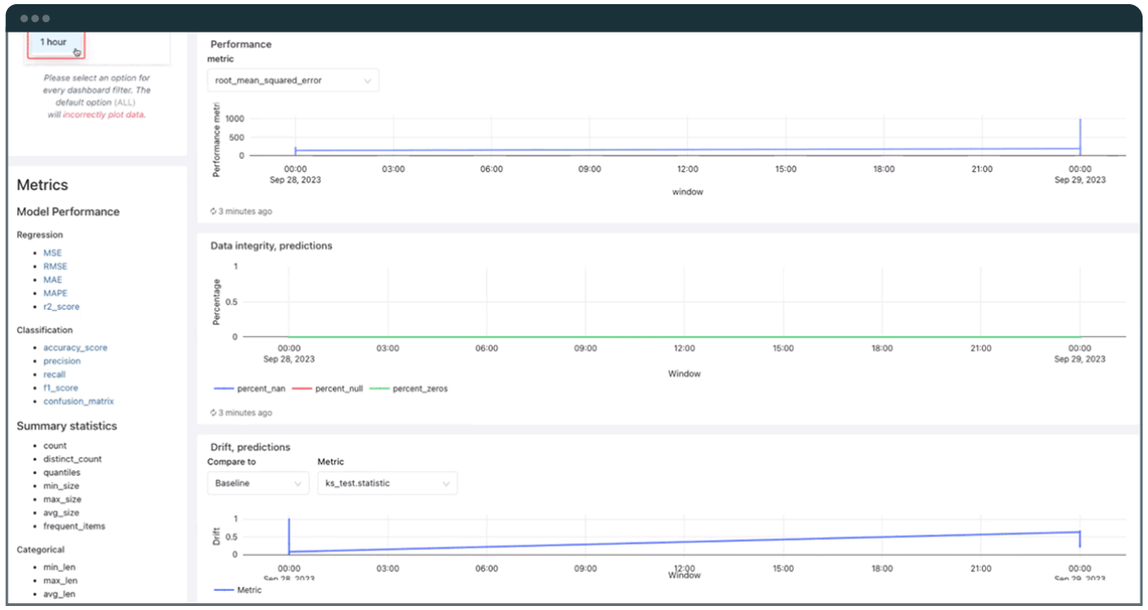
How do you ensure trust in your data, ML models, and AI across your entire data pipeline in a single view regardless of where the data resides? Databricks Lakehouse Monitoring is the industry's only comprehensive solution from data (regardless of where it resides) to insights. It accelerates the discovery of issues, helps determine root causes, and ultimately assists in recommending solutions.

UC provides Lakehouse Monitoring capabilities with both democratized dashboards and granular governance information that can be directly queried through system tables. The democratization of governance extends operational oversight and compliance to non-technical people, allowing a broad variety of teams to monitor all of their pipelines.

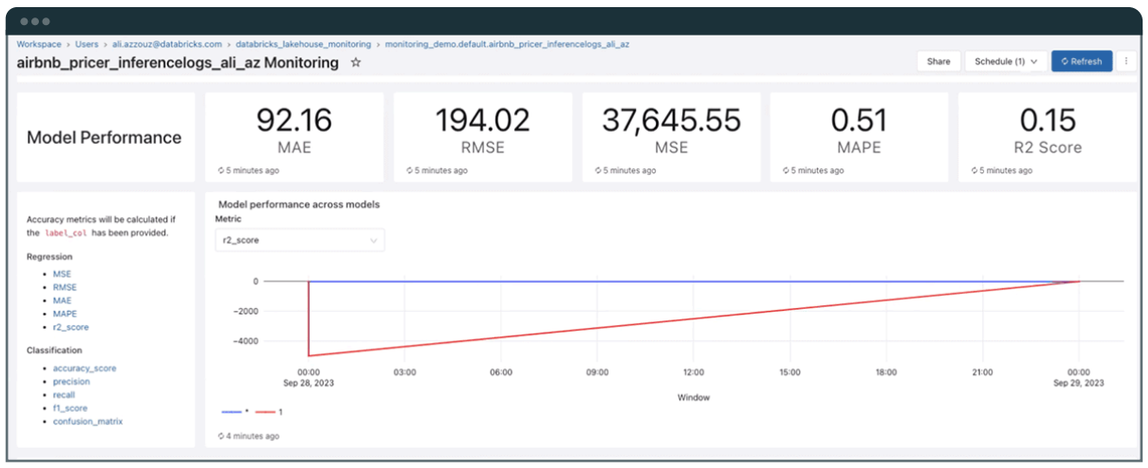
Below is a sample dashboard of the results of an ML model including its accuracy over time:



It further shows data integrity of predictions and data drift over time:

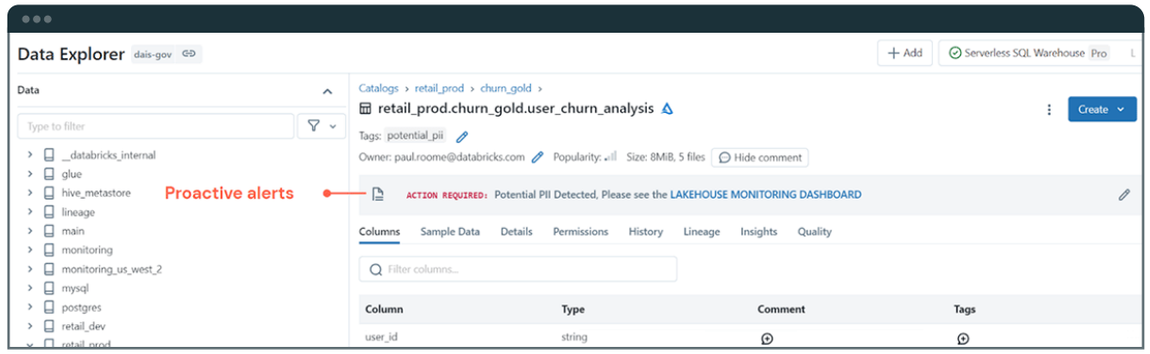


And model performance over time, according to a variety of ML metrics such as R2, RMSE, and MAPE:



Lakehouse Monitoring dashboards show data and AI assets quality

It's one thing to intentionally seek out ML model information when you are looking for answers, but it is a whole other level to get automated proactive alerts on errors, data drift, model failures, or quality issues. Below is an example alert for a potential PII (Personal Identifiable Information) data breach:



Example proactive alert of potential unmasked private data

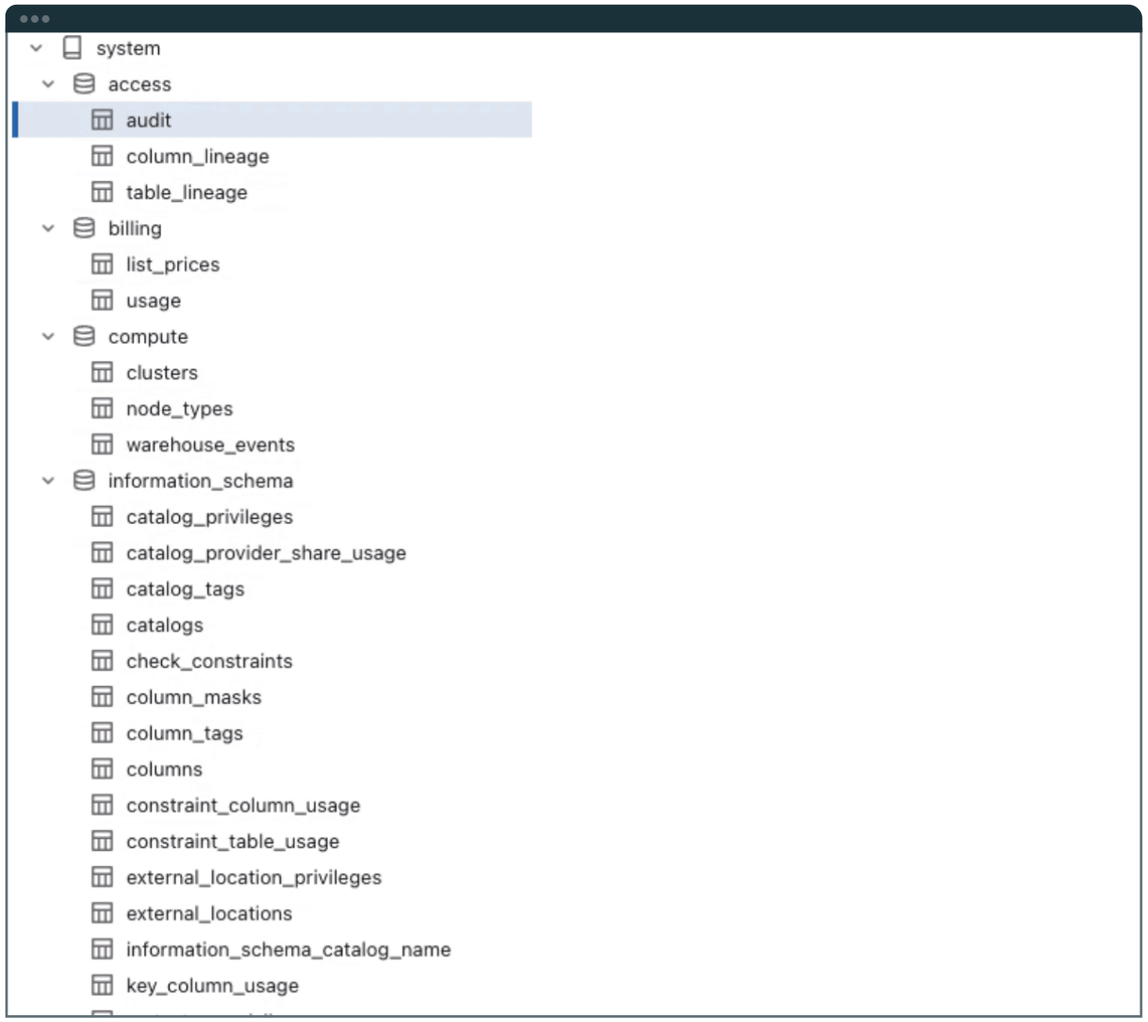
One more thing — you can assess the impact of issues, do a root cause analysis, and assess the downstream impact by Databrick's powerful lineage capabilities — from table-level to column-level.

System tables: metadata information for lakehouse observability and ensuring compliance

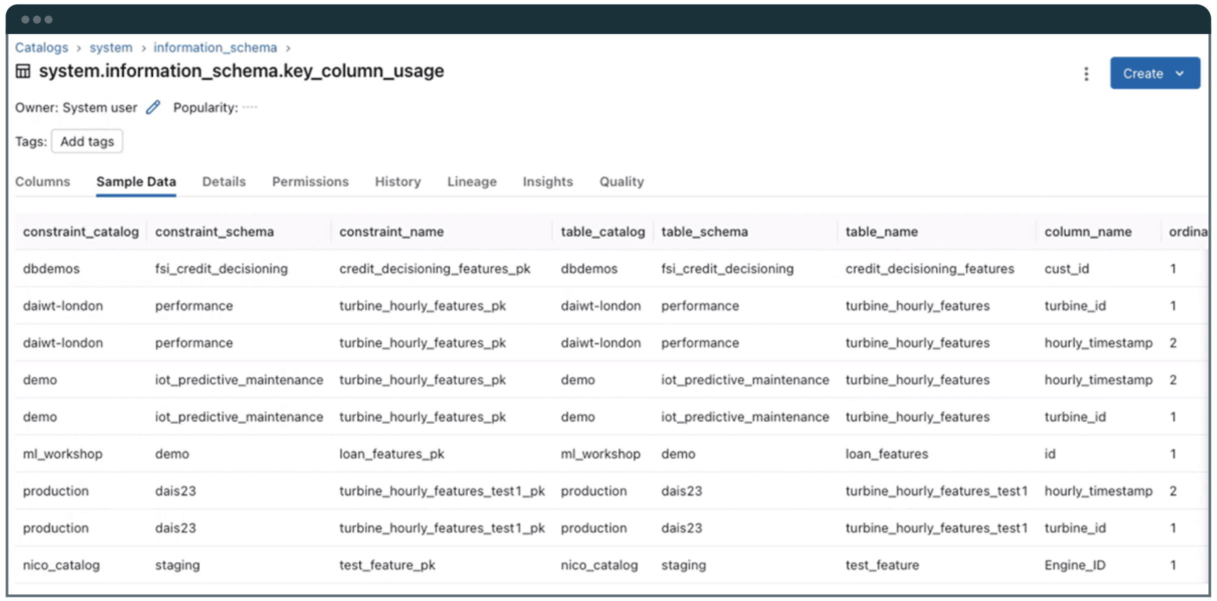
These underlying tables can be queried through SQL or activity dashboards to provide observability about every asset within the Databricks Intelligence Platform. Examples include which users have access to which data objects; billing tables that provide pricing and usage; compute tables that take cluster usage and warehouse events into consideration; and lineage information between columns and tables:

- Audit tables include information on a wide variety of UC events. UC captures an audit log of actions performed against the metastore giving administrators access to details about who accessed a given dataset and the actions that they performed.
- Billing and historical pricing tables will include records for all billable usage across the entire account; therefore you can view your account's global usage from whichever region your workspace is in.
- Table lineage and column lineage tables are great because they allow you to programmatically query lineage data to fuel decision making and reports. Table lineage records each read-and-write event on a UC table or path that might include job runs, notebook runs and dashboards associated with the table. For column lineage, data is captured by reading the column.
- Node types tables capture the currently available node types with their basic hardware information outlining the node type name, the number of vCPUs for the instance, and the number of GPUs and memory for the instance. Also in private preview are node_utilization metrics on how much usage each node is leveraging.
- Query history holds information on all SQL commands, i/o performance, and number of rows returned.
- Clusters table contains the full history of cluster configurations over time for all-purpose and job clusters.
- Predictive optimization tables are great because they optimize your data layout for peak performance and cost efficiency. The tables track the operation history of optimized tables by providing the catalog name, schema name, table name, and operation metrics about compaction and vacuuming.

From the catalog explorer, here are just a few of the system tables any of which can be viewed for more details:

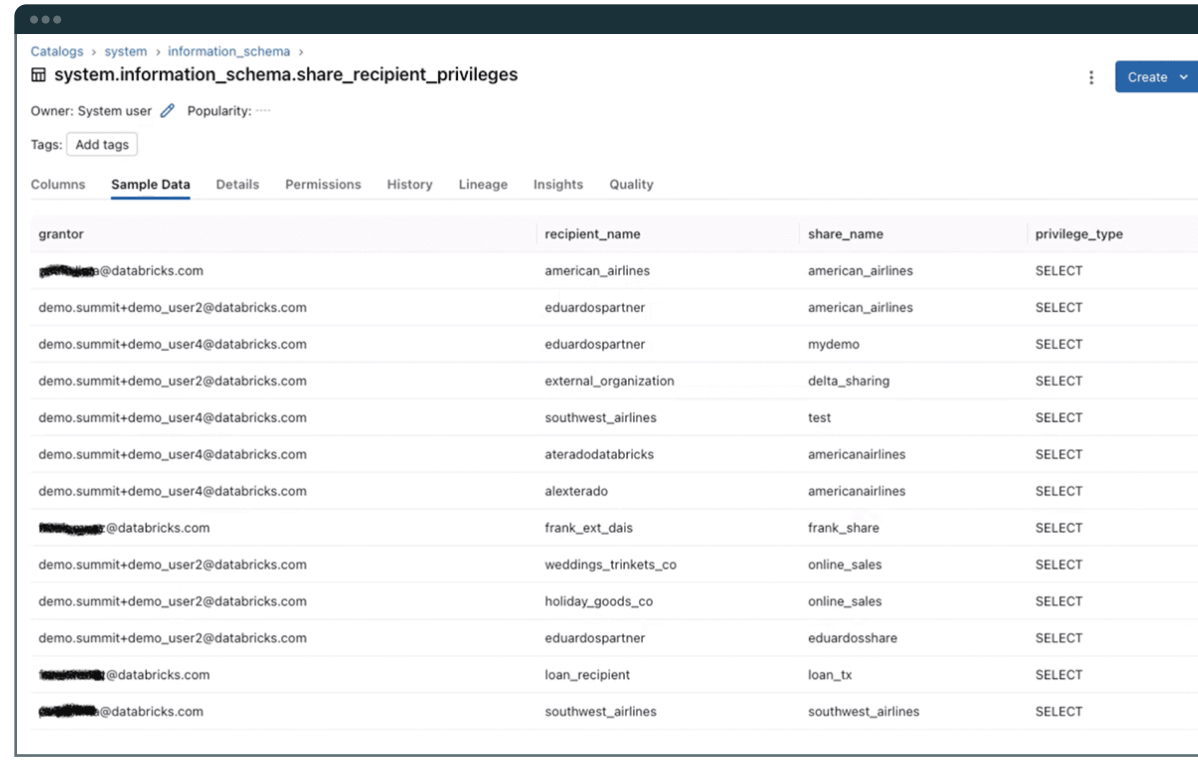


As an example, drilling down on the "key_column_usage" table, you can see precisely how tables relate to each other via their primary key:



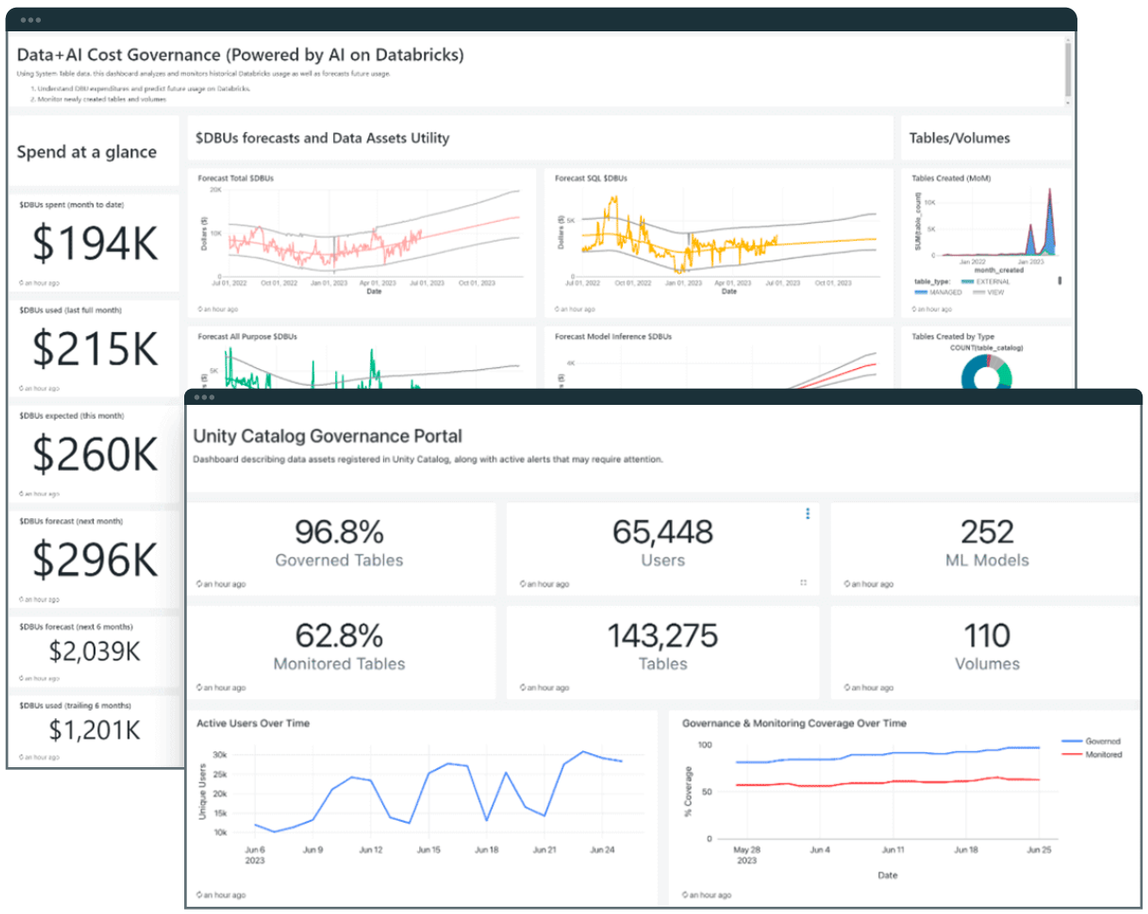
constraint_catalog	constraint_schema	constraint_name	table_catalog	table_schema	table_name	column_name	ordina
dbdemos	fsi_credit_decisioning	credit_decisioning_features_pk	dbdemos	fsi_credit_decisioning	credit_decisioning_features	cust_id	1
daiwt-london	performance	turbine_hourly_features_pk	daiwt-london	performance	turbine_hourly_features	turbine_id	1
daiwt-london	performance	turbine_hourly_features_pk	daiwt-london	performance	turbine_hourly_features	hourly_timestamp	2
demo	iot_predictive_maintenance	turbine_hourly_features_pk	demo	iot_predictive_maintenance	turbine_hourly_features	hourly_timestamp	2
demo	iot_predictive_maintenance	turbine_hourly_features_pk	demo	iot_predictive_maintenance	turbine_hourly_features	turbine_id	1
ml_workshop	demo	loan_features_pk	ml_workshop	demo	loan_features	id	1
production	dais23	turbine_hourly_features_test1_pk	production	dais23	turbine_hourly_features_test1	hourly_timestamp	2
production	dais23	turbine_hourly_features_test1_pk	production	dais23	turbine_hourly_features_test1	turbine_id	1
nico_catalog	staging	test_feature_pk	nico_catalog	staging	test_feature	Engine_ID	1

Another example is the "share_recipient_privileges" table, to see who granted which shares to whom:



grantor	recipient_name	share_name	privilege_type
[redacted]@databricks.com	american_airlines	american_airlines	SELECT
demo.summit+demo_user2@databricks.com	eduardospartner	american_airlines	SELECT
demo.summit+demo_user4@databricks.com	eduardospartner	mydemo	SELECT
demo.summit+demo_user2@databricks.com	external_organization	delta_sharing	SELECT
demo.summit+demo_user4@databricks.com	southwest_airlines	test	SELECT
demo.summit+demo_user4@databricks.com	ateradodatabricks	americanairlines	SELECT
demo.summit+demo_user4@databricks.com	alexterado	americanairlines	SELECT
[redacted]@databricks.com	frank_ext_dais	frank_share	SELECT
demo.summit+demo_user2@databricks.com	weddings_trinkets_co	online_sales	SELECT
demo.summit+demo_user2@databricks.com	holiday_goods_co	online_sales	SELECT
demo.summit+demo_user2@databricks.com	eduardospartner	eduardosshare	SELECT
[redacted]@databricks.com	loan_recipient	loan_tx	SELECT
[redacted]@databricks.com	southwest_airlines	southwest_airlines	SELECT

The example dashboard below shows the number of users, tables, ML models, percent of tables that are monitored or not, dollars spent on Databricks DBUs over time, and so much more:



Governance dashboard showing billing trends, usage, activity and more

What does having a comprehensive data and AI monitoring and reporting tool result in?

- Reduced risk of non-compliance with better monitoring of internal policies and security breach potential results in safeguarded reputation and improved data and AI trust from employees and partners.
- Improved integrity and trustworthiness of data and AI with "one source of truth", anomaly detection, and reliability metrics.

Value Levers with Databricks Unity Catalog

If you are looking to learn more about the values Unity Catalog brings to businesses, the prior Unity Catalog Governance Value Levers blog went into detail: mitigating risk around compliance; reducing platform complexity and costs; accelerating innovation; facilitating better internal and external collaboration; and monetizing the value of data.

CONCLUSION

Governance is key to mitigating risks, ensuring compliance, accelerating innovation, and reducing costs. Databricks Unity Catalog is unique in the market, providing a single unified governance solution for all of a company's data and AI across clouds and data platforms.

UC Databricks architecture makes governance seamless: a unified view and discovery of all data assets, one tool for access management, one tool for auditing for enhanced data and AI security, and ultimately enabling platform-independent collaboration that unlocks new business values.

Getting started is easy - UC comes enabled by default with Databricks if you are a new customer! Also if you are on premium or enterprise workspaces, there are no additional costs.

Scalable Spark Structured Streaming for REST API Destinations

by Art Rask and Jay Palaniappan

Spark Structured Streaming is the widely-used open source engine at the foundation of data streaming on the Data Intelligence Platform. It can elegantly handle diverse logical processing at volumes ranging from small-scale ETL to the largest Internet services. This power has led to adoption in many use cases across industries.

Another strength of Structured Streaming is its ability to handle a variety of both sources and sinks (or destinations). In addition to numerous sink types supported natively (incl. Delta, AWS S3, Google GCS, Azure ADLS, Kafka topics, Kinesis streams, and more), Structured Streaming supports a specialized sink that has the ability to perform arbitrary logic on the output of a streaming query: the foreachBatch extension method. With foreachBatch, any output target addressable through Python or Scala code can be the destination for a stream.

In this chapter we will share best practice guidance we've given customers who have asked how they can scalably turn streaming data into calls against a REST API. Routing an incoming stream of data to calls on a REST API is a requirement seen in many integration and data engineering scenarios.

Some practical examples that we often come across are in Operational and Security Analytics workloads. Customers want to ingest and enrich real-time streaming data from sources like kafka, eventhub, and Kinesis and publish it into operational search engines like Elasticsearch, Opensearch, and Splunk. A key advantage of Spark Streaming is that it allows us to enrich, perform data quality checks, and aggregate (if needed) before data is streamed out into the search engines. This provides customers a high quality real-time data pipeline for operational and security analytics.

The most basic representation of this scenario is shown in Figure 1. Here we have an incoming stream of data - it could be a Kafka topic, AWS Kinesis, Azure Event Hub, or any other streaming query source. As messages flow off the stream we need to make calls to a REST API with some or all of the message data.

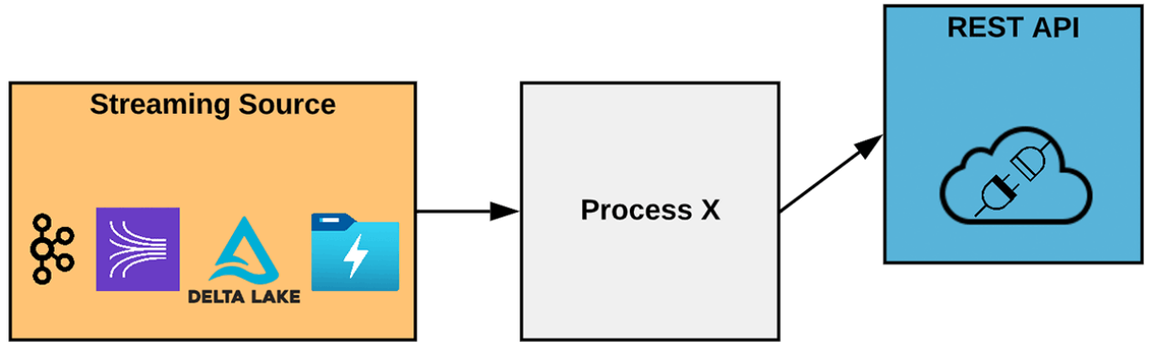


Figure 1

In a greenfield environment, there are many technical options to implement this. Our focus here is on teams that already have streaming pipelines in Spark for preparing data for machine learning, data warehousing, or other analytics-focused uses. In this case, the team will already have skills, tooling and DevOps processes for Spark. Assume the team now has a requirement to route some data to REST API calls. If they wish to leverage existing skills or avoid re-working their tool chains, they can use Structured Streaming to get it done.

KEY IMPLEMENTATION TECHNIQUES, AND SOME CODE

A basic code sample is included as Exhibit 1. Before looking at it in detail, we will call out some key techniques for effective implementation.

For a start, you will read the incoming stream as you would any other streaming job. All the interesting parts here are on the output side of the stream. If your data must be transformed in flight before posting to the REST API, do that as you would in any other case. This code snippet reads from a Delta table; as mentioned, there are many other possible sources.

```
dfSource = (spark.readStream
                .format("delta")
                .table("samples.nyctaxi.trips"))
```

For directing streamed data to the REST API, take the following approach:

1. Use the foreachBatch extension method to pass incoming micro-batches to a handler method (callRestAPIBatch) which will handle calls to the REST API.

```
streamHandle = (dfSource.writeStream
                        .foreachBatch(callRestAPIBatch)
                        .start())
```

2. Whenever possible, group multiple rows from the input on each outgoing REST API call. In relative terms, making the API call over HTTP will be a slow part of the process. Your ability to reach high throughput will be dramatically improved if you include multiple messages/records on the body of each API call. Of course, what you can do will be dictated by the target REST API. Some APIs allow a POST body to include many items up to a maximum body size in bytes. Some APIs have a max count of items on the POST body. Determine the max you can fit on a single call for the target API. In your method invoked by foreachBatch, you will have a prep step to transform the micro-batch dataframe into a pre-batched dataframe where each row has the grouped records for one call to the API. This step is also a chance for any last transform of the records to the format expected by the target API. An example is shown in the code sample in Exhibit 1 with the call to a helper function named preBatchRecordsForRestCall.
3. In most cases, to achieve a desired level of throughput, you will want to make calls to the API from parallel tasks. You can control the degree of parallelism by calling repartition on the dataframe of pre-batched data. Call repartition with the number of parallel tasks you want calling the API. This is actually just one line of code.

```
### Repartition pre-batched df for parallelism of API calls
new_df = pre_batched_df.repartition(8)
```

It is worth mentioning (or admitting) that using repartition here is a bit of an anti-pattern. Explicit repartitioning with large datasets can have performance implications, especially if it causes a shuffle between nodes on the cluster. In most cases of calling a REST API, the data size of any micro-batch is not massive. So, in practical terms, this technique is unlikely to cause a problem. And, it has a big positive effect on throughput to the API.

4. Execute a dataframe transformation that calls a nested function dedicated to making a single call to the REST API. The input to this function will be one row of pre-batched data. In the sample, the payload column has the data to include on a single call. Call a dataframe action method to invoke execution of the transformation.

```
submitted_df = new_df.withColumn("RestAPIResponseCode",\
                           callRestApiOnce(new_df["payload"])).\
                           collect()
```

5. Inside the nested function which will make one API call, use your libraries of choice to issue an HTTP POST against the REST API. This is commonly done with the Requests library but any library suitable for making the call can be considered. See the callRestApiOnce method in Exhibit 1 for an example.
6. Handle potential errors from the REST API call by using a try..except block or checking the HTTP response code. If the call is unsuccessful, the overall job can be failed by throwing an exception (for job retry or troubleshooting) or individual records can be diverted to a dead letter queue for remediation or later retry.

```
if not (response.status_code==200 or response.status_code==201) :
 raise Exception("Response status : {} .Response message : {}".\
                 format(str(response.status_code),response.text))
```

The six elements above should prepare your code for sending streaming data to a REST API, with the ability to scale for throughput and to handle error conditions cleanly. The sample code in Exhibit 1 is an example implementation. Each point stated above is reflected in the full example.

```
from pyspark.sql.functions import *
from pyspark.sql.window import Window
import math
import requests
from requests.adapters import HTTPAdapter

def preBatchRecordsForRestCall(microBatchDf, batchSize):
    batch_count = math.ceil(microBatchDf.count() / batchSize)
    microBatchDf = microBatchDf.withColumn("content", to_json(struct(col("*"))))
    microBatchDf = microBatchDf.withColumn("row_number",\
                                           row_number().over(Window().
orderBy(lit('A'))))
    microBatchDf = microBatchDf.withColumn("batch_id", col("row_number") % batch_
count)
    return microBatchDf.groupBy("batch_id").\
                                        agg(concat_ws(",|", collect_
list("content")).\
                                        alias("payload"))


def callRestAPIBatch(df, batchId):
  restapi_uri = "<REST API URL>"

  @udf("string")
  def callRestApiOnce(x):
    session = requests.Session()
    adapter = HTTPAdapter(max_retries=3)
    session.mount('http://', adapter)
    session.mount('https://', adapter)

    #this code sample calls an unauthenticated REST endpoint; add headers
necessary for auth
    headers = {'Authorization':'abcd'}
    response = session.post(restapi_uri, headers=headers, data=x, verify=False)
    if not (response.status_code==200 or response.status_code==201) :
      raise Exception("Response status : {} .Response message : {}".\
                      format(str(response.status_code),response.text))

    return str(response.status_code)
```

```
  ### Call helper method to transform df to pre-batched df with one row per REST
API call
  ### The POST body size and formatting is dictated by the target API; this is an
example
  pre_batched_df = preBatchRecordsForRestCall(df, 10)

  ### Repartition pre-batched df for target parallelism of API calls
  new_df = pre_batched_df.repartition(8)

  ### Invoke helper method to call REST API once per row in the pre-batched df
  submitted_df = new_df.withColumn("RestAPIResponseCode",\
                                   callRestApiOnce(new_df["payload"])).collect()


dfSource = (spark.readStream
                .format("delta")
                .table("samples.nyctaxi.trips"))

streamHandle = (dfSource.writeStream
                        .foreachBatch(callRestAPIBatch)
                        .trigger(availableNow=True)
                        .start())
```

Exhibit 1

DESIGN AND OPERATIONAL CONSIDERATIONS

Exactly Once vs At Least Once Guarantees

As a general rule in Structured Streaming, using foreachBatch only provides at-least-once delivery guarantees. This is in contrast to the exactly-once delivery guarantee provided when writing to sinks like a Delta table or file sinks. Consider, for example, a case where 1,000 records arrive on a micro-batch and your code in foreachBatch begins calling the REST API with the batch. In a hypothetical failure scenario, let's say that 900 calls succeed before an error occurs and fails the job. When the stream restarts, processing will resume by re-processing the failed batch. Without additional logic in your code, the 900 already-processed calls will be repeated. It is important that you determine in your design whether this is acceptable, or whether you need to take additional steps to protect against duplicate processing.

The general rule when using foreachBatch is that your target sink (REST API in this case) should be idempotent or that you must do additional tracking to account for multiple calls with the same data.

Estimating Cluster Core Count for a Target Throughput

Given these techniques to call a REST API with streaming data, it will quickly become necessary to estimate how many parallel executors/tasks are necessary to achieve your required throughput. And you will need to select a cluster size. The following table shows an example calculation for estimating the number of worker cores to provision in the cluster that will run the stream.

A	Target Throughput (records/sec)	3200	←Required rate of records processed by stream and submitted to REST API
B	# records batched per REST API call	10	←Average number of records included on POST body for each API call
C	Total required calls per second	320	A / B, rounded up
D	API Latency (ms to make one call)	50	← Plug in estimated or observed latency
E	Max calls per task per second	20	← 1000ms / D, rounded up
F	Min required parallelism	16	C / E, rounded up
G	Shared use factor	2	Rule-of-thumb multiplier to account for other work being done on cluster
H	Estimated core count	32	F * G, rounded up.

Line H in the table shows the estimated number of worker cores necessary to sustain the target throughput. In the example shown here, you could provision a cluster with two 16-core workers or 4 8-core workers, for example. For this type of workload, fewer nodes with more cores per node is preferred.

Line H is also the number that would be put in the repartition call in foreachBatch, as described in item 3 above.

Line G is a rule of thumb to account for other activity on the cluster. Even if your stream is the only job on the cluster, it will not be calling the API 100% of the time. Some time will be spent reading data from the source stream, for example. The value shown here is a good starting point for this factor - you may be able to fine tune it based on observations of your workload.

Obviously, this calculation only provides an estimated starting point for tuning the size of your cluster. We recommend you start from here and adjust up or down to balance cost and throughput.

OTHER FACTORS TO CONSIDER

There are other factors you may need to plan for in your deployment. These are outside the scope of this post, but you will need to consider them as part of implementation. Among these are:

1. Authentication requirements of the target API: It is likely that the REST API will require authentication. This is typically done by adding required headers in your code before making the HTTP POST.
2. Potential rate limiting: The target REST API may implement rate limiting which will place a cap on the number of calls you can make to it per second or minute. You will need to ensure you can meet throughout targets within this limit. You'll also want to be ready to handle throttling errors that may occur if the limit is exceeded.
3. Network path required from worker subnet to target API: Obviously, the worker nodes in the host Spark cluster will need to make HTTP calls to the REST API endpoint. You'll need to use the available cloud networking options to configure your environment appropriately.
4. If you control the implementation of the target REST API (e.g., an internal custom service), be sure the design of that service is ready for the load and throughput generated by the streaming workload.

MEASURED THROUGHPUT TO A MOCKED API WITH DIFFERENT NUMBERS OF PARALLEL TASKS

To provide representative data of scaling REST API calls as described here, we ran tests using code very similar to Example 1 against a mocked up REST API that persisted data in a log.

Results from the test are shown in Table 1. These metrics confirm near-linear scaling as the task count was increased (by changing the partition count using repartition). All tests were run on the same cluster with a single 16-core worker node.

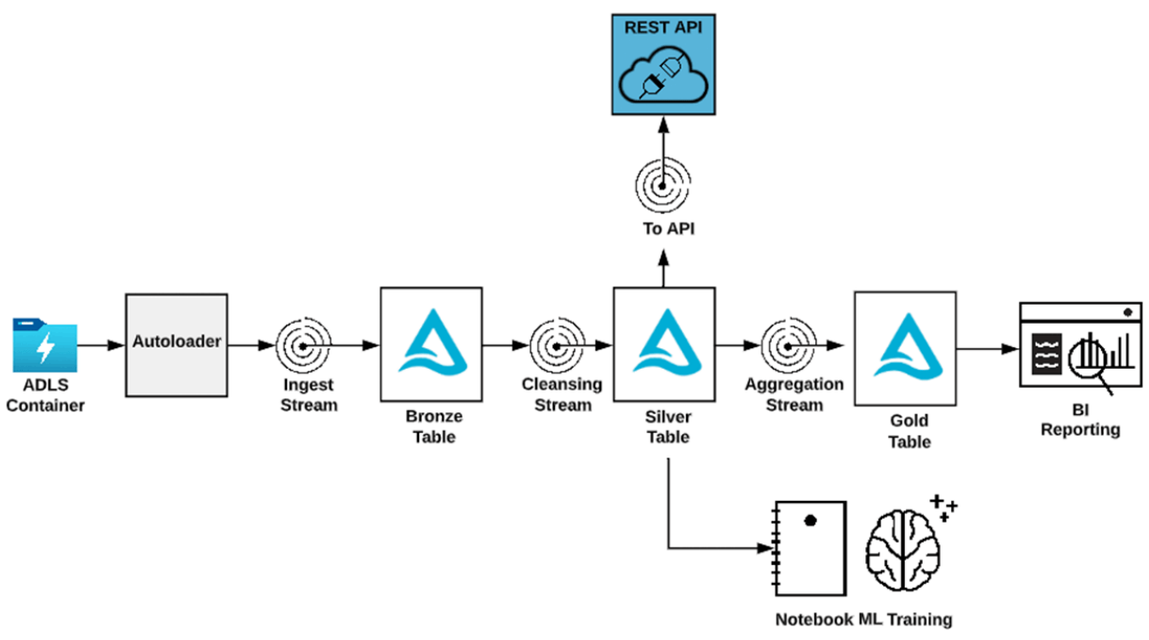
Rows per API Call	Partitions (threads)	Total calls	Seconds to complete processing	Rows per second	Calls per second	Calls per thread per second	Apprx API Latency (ms)
1	4	21,932	179	122	122	30.5	34
	8	21,932	88	249	249	31.1	33
	16	21,932	49	447	447	27.9	38
10	4	21,932	48	456	45	11.3	91
	8	21,932	25	877	87	10.9	100
	16	21,932	13	1687	168	10.5	100

Table 1

REPRESENTATIVE ALL UP PIPELINE DESIGNS

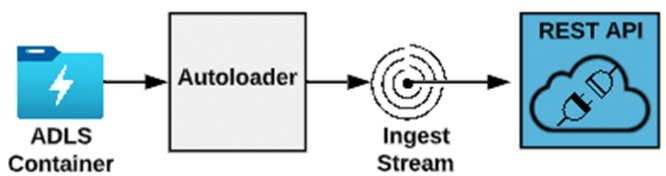
1. Routing some records in a streaming pipeline to REST API (in addition to persistent sinks)

   This pattern applies in scenarios where a Spark-based data pipeline already exists for serving analytics or ML use cases. If a requirement emerges to post cleansed or aggregated data to a REST API with low latency, the technique described here can be used.



2. Simple Autoloader to REST API job

   This pattern is an example of leveraging the diverse range of sources supported by Structured Streaming. Databricks makes it simple to consume incoming near real-time data - for example using Autoloader to ingest files arriving in cloud storage. Where Databricks is already used for other use cases, this is an easy way to route new streaming sources to a REST API.



SUMMARY

We have shown here how structured streaming can be used to send streamed data to an arbitrary endpoint - in this case, via HTTP POST to a REST API. This opens up many possibilities for flexible integration with analytics data pipelines. However, this is really just one illustration of the power of foreachBatch in Spark Structured Streaming.

The foreachBatch sink provides the ability to address many endpoint types that are not among the native sinks. Besides REST APIs, these can include databases via JDBC, almost any supported Spark connector, or other cloud services that are addressable via a helper library or API. One example of the latter is pushing data to certain AWS services using the boto3 library.

This flexibility and scalability enables Structured Streaming to underpin a vast range of real-time solutions across industries.

If you are a Databricks customer, simply follow the getting started tutorial to familiarize yourself with Structured Streaming. If you are not an existing Databricks customer, sign up for a free trial.

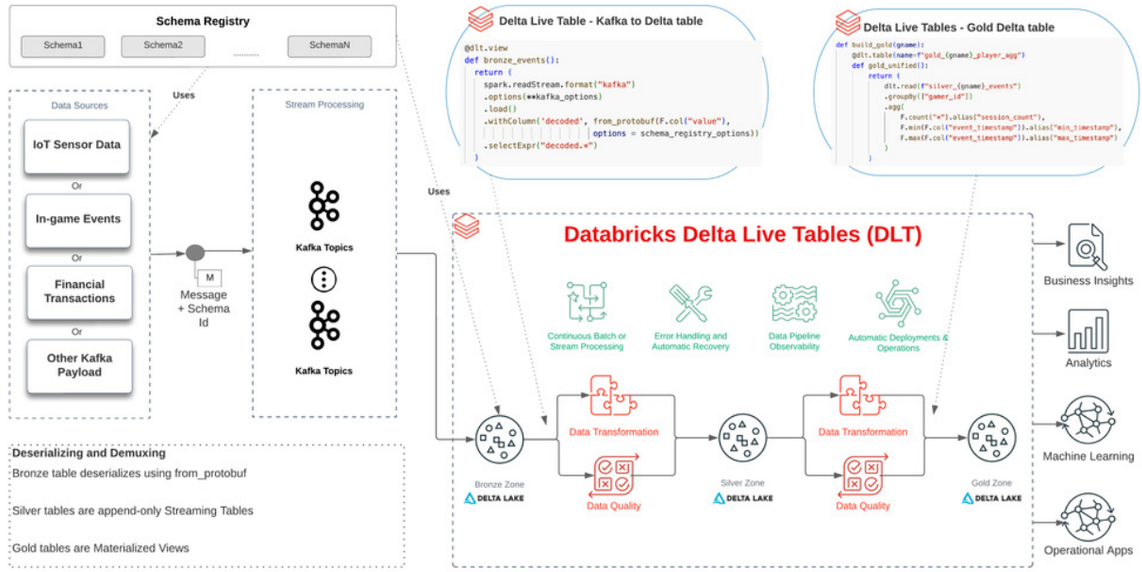
A Data Engineer's Guide to Optimized Streaming With Protobuf and Delta Live Tables

by Craig Lukasik

This article describes an example use case where events from multiple games stream through Kafka and terminate in Delta tables. The example illustrates how to use Delta Live Tables (DLT) to:

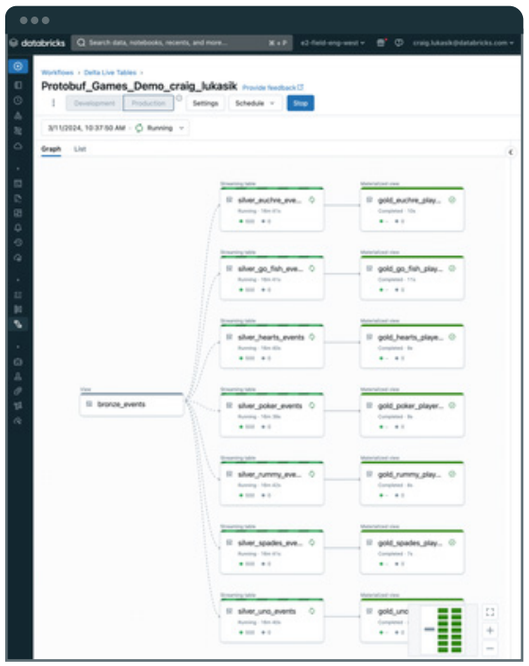
1. Stream from Kafka into a Bronze Delta table.
2. Consume streaming Protobuf messages with schemas managed by the Confluent Schema Registry, handling schema evolution gracefully.
3. Demultiplex (demux) messages into multiple game-specific, append-only Silver Streaming Tables. Demux indicates that a single stream is split or fanned out into separate streams.
4. Create Materialized Views to recalculate aggregate values periodically.

A high-level view of the system architecture is illustrated below.



First, let's look at the Delta Live Tables code for the example and the related pipeline DAG so that we can get a glimpse of the simplicity and power of the DLT framework.

```
The source bronze view, consuming the Kafk...
@dlt.view
def bronze_events():
  return (
    spark.readStream.format("kafka")
    .options(**kafka_options)
    .load()
    .withColumn('decoded', from_protobuf(F.col("value"),
                                         options = schema_registry_options))
    .selectExpr("decoded.*")
  )

Helper functions to define DLT tables
def build_silver(gname):
    @dlt.table(name=f"silver_{gname}_events")
    def gold_unified():
        return dlt.read_stream("bronze_events").where(F.col("game_name") == gname)

def build_gold(gname):
    @dlt.table(name=f"gold_{gname}_player_agg")
    def gold_unified():
        return (
            dlt.read(f"silver_{gname}_events")
            .groupBy(["gamer_id"])
            .agg(
                F.count("*").alias("session_count"),
                F.min(F.col("event_timestamp")).alias("min_timestamp"),
                F.max(F.col("event_timestamp")).alias("max_timestamp")
            )
        )

Silver Delta tables for each game. Notice the...
for game in GAMES_ARRAY:
    build_silver(game)
    build_gold(game)
```



On the left, we see the DLT Python code. On the right, we see the view and the tables created by the code. The bottom cell of the notebook on the left operates on a list of games (GAMES_ARRAY) to dynamically generate the fourteen target tables we see in the DAG.

Before we go deeper into the example code, let's take a step back and review streaming use cases and some streaming payload format options.

STREAMING OVERVIEW

Skip this section if you're familiar with streaming use cases, protobuf, the schema registry, and Delta Live Tables. In this article, we'll dive into a range of exciting topics.

- **Common streaming use cases:** Uncover the diverse streaming data applications in today's tech landscape.
- Protocol buffers (Protobuf): Learn why this fast and compact serialization format is a game-changer for data handling.
- Delta Live Tables (DLT): Discover how DLT pipelines offer a rich, feature-packed platform for your ETL (Extract, Transform, Load) needs.
- **Building a DLT pipeline**: A step-by-step guide on creating a DLT pipeline that seamlessly consumes Protobuf values from an Apache Kafka stream.
- **Utilizing the** Confluent Schema Registry: Understand how this tool is crucial for decoding binary message payloads effectively.
- Schema evolution **in DLT pipelines**: Navigate the complexities of schema evolution within the DLT pipeline framework when streaming protobuf messages with evolving schema.

COMMON STREAMING USE CASES

The Databricks Data Intelligence Platform is a comprehensive data-to-AI enterprise solution that combines data engineers, analysts, and data scientists on a single platform. Streaming workloads can power near real-time prescriptive and predictive analytics and automatically retrain Machine Learning (ML) models using Databricks built-in MLOps support. The models can be exposed as scalable, serverless REST endpoints, all within the Databricks platform.

The data comprising these streaming workloads may originate from various use cases:

STREAMING DATA	USE CASE
IoT sensors on manufacturing floor equipment	Generating predictive maintenance alerts and preemptive part ordering
Set-top box telemetry	Detecting network instability and dispatching service crews
Player metrics in a game	Calculating leader-board metrics and detecting cheat

Data in these scenarios is typically streamed through open source messaging systems, which manage the data transfer from producers to consumers. Apache Kafka stands out as a popular choice for handling such payloads. Confluent Kafka and AWS MSK provide robust Kafka solutions for those seeking managed services.

OPTIMIZING THE STREAMING PAYLOAD FORMAT

Databricks provides capabilities that help optimize the AI journey by unifying Business Analysis, Data Science, and Data Analysis activities in a single, governed platform. In your quest to optimize the end-to-end technology stack, a key focus is the serialization format of the message payload. This element is crucial for efficiency and performance. We'll specifically explore an optimized format developed by Google, known as protocol buffers (or "protobuf"), to understand how it enhances the technology stack.

WHAT MAKES PROTOBUF AN OPTIMIZED SERIALIZATION FORMAT?

Google enumerates the advantages of protocol buffers, including compact data storage, fast parsing, availability in many programming languages, and optimized functionality through automatically generated classes.

A key aspect of optimization usually involves using pre-compiled classes in the consumer and producer programs that a developer typically writes. In a nutshell, consumer and producer programs that leverage protobuf are "aware" of a message schema, and the binary payload of a protobuf message benefits from primitive data types and positioning within the binary message, removing the need for field markers or delimiters.

WHY IS PROTOBUF USUALLY PAINFUL TO WORK WITH?

Programs that leverage protobuf must work with classes or modules compiled using protoc (the protobuf compiler). The protoc compiler compiles those definitions into classes in various languages, including Java and Python. To learn more about how protocol buffers work, go here.

DATABRICKS MAKES WORKING WITH PROTOBUF EASY

Starting in Databricks Runtime 12.1, Databricks provides native support for serialization and deserialization between Apache Spark struct.... Protobuf support is implemented as an Apache Spark DataFrame transformation and can be used with Structured Streaming or for batch operations. It optionally integrates with the Confluent Schema Registry (a Databricks-exclusive feature).

Databricks makes it easy to work with protobuf because it handles the protobuf compilation under the hood for the developer. For instance, the data pipeline developer does not have to worry about installing protoc or using it to compile protocol definitions into Python classes.

EXPLORING PAYLOAD FORMATS FOR STREAMING IOT DATA

Before we proceed, it is worth mentioning that JSON or Avro may be suitable alternatives for streaming payloads. These formats offer benefits that, for some use cases, may outweigh protobuf. Let's quickly review these formats.

JSON

JSON is an excellent format for development because it is primarily human-readable. The other formats we'll explore are binary formats, which require tools to inspect the underlying data values. Unlike Avro and protobuf, however, the JSON document is stored as a large string (potentially compressed), meaning more bytes may be used than a value represents. Consider the short int value of 8. A short int requires two bytes. In JSON, you may have a document that looks like the following, and it will require several bytes (~30) for the associated key, quotes, etc.

```
{
  "my_short": 8
}
```

When we consider protobuf, we expect 2 bytes plus a few more for the overhead related to the positioning metadata.

JSON SUPPORT IN DATABRICKS

On the positive side, JSON documents have rich benefits when used with Databricks. Databricks Autoloader can easily transform JSON to a structured DataFrame while also providing built-in support for:

- Schema inference - when reading JSON into a DataFrame, you can supply a schema so that the target DataFrame or Delta table has the desired schema. Or you can let the engine infer the schema. Alternatively, schema hints can be supplied if you want a balance of those features.
- Schema evolution - Autoloader provides options for how a workload should adapt to changes in the schema of incoming files.

Consuming and processing JSON in Databricks is simple. To create a Spark DataFrame from JSON files can be as simple as this:

```
df = spark.read.format("json").load("example.json")
```

AVRO

Avro is an attractive serialization format because it is compact, encompasses schema information in the files themselves, and has built-in database support in Databricks that includes schema registry integration. This tutorial, co-authored by Databricks' Angela Chu, walks you through an example that leverages Confluent's Kafka and Schema Registry.

To explore an Avro-based dataset, it is as simple as working with JSON:

```
df = spark.read.format("avro").load("example.avro")
```

This datageeks.com article compares Avro and protobuf. It is worth a read if you are on the fence between Avro and protobuf. It describes protobuf as the "fastest amongst all.", so if speed outweighs other considerations, such as JSON and Avro's greater simplicity, protobuf may be the best choice for your use case.

EXAMPLE DEMUX PIPELINE

The source code for the end-to-end example is located on GitHub. The example includes a simulator (Producer), a notebook to install the Delta Live Tables pipeline (Install_DLT_Pipeline), and a Python notebook to process the data that is streaming through Kafka (DLT).

SCENARIO

Imagine a scenario where a video gaming company is streaming events from game consoles and phone-based games for a number of the games in its portfolio. Imagine the game event messages have a single schema that evolves (i.e., new fields are periodically added). Lastly, imagine that analysts want the data for each game to land in its own Delta Lake table. Some analysts and BI tools need pre-aggregated data, too.

Using DLT, our pipeline will create 1+2N tables:

- One table for the raw data (stored in the Bronze view).
- One Silver Streaming Table for each of the N games, with events streaming through the Bronze table.
- Each game will also have a Gold Delta table with aggregates based on the associated Silver table.

CODE WALKTHROUGH

BRONZE TABLE DEFINITION

We'll define the Bronze table (bronze_events) as a DLT view by using the @dlt.view annotation

```
import pyspark.sql.functions as F
from pyspark.sql.protobuf.functions import from_protobuf

@dlt.view
def bronze_events():
  return (
    spark.readStream.format("kafka")
    .options(**kafka_options)
    .load()
    .withColumn('decoded', from_protobuf(F.col("value"), options = schema_registry_
options))
    .selectExpr("decoded.*")
  )
```

The repo includes the source code that constructs values for kafka_options. These details are needed so the streaming Delta Live Table can consume messages from the Kafka topic and retrieve the schema from the Confluent Schema registry (via config values in schema_registry_options). This line of code is what manages the deserialization of the protobuf messages:

```
.withColumn('decoded', from_protobuf(F.col("value"), options = schema_registry_
options))
```

The simplicity of transforming a DataFrame with protobuf payload is thanks to this function: from_protobuf (available in Databricks Runtime 12.1 and later). In this article, we don't cover to_protobuf, but the ease of use is the same. The schema_registry_options are used by the function to look up the schema from the Confluent Schema Registry.

Delta Live Tables is a declarative ETL framework that simplifies the development of data pipelines. So, suppose you are familiar with Apache Spark Structured Streaming. In that case, you may notice the absence of a checkpointLocation (which is required to track the stream's progress so that the stream can be stopped and started without duplicating or dropping data). The absence of the checkpointLocation is because Delta Live Tables manages this need out-of-the-box for you. Delta Live Tables also has other features that help make developers more agile and provide a common framework for ETL across the enterprise. Delta Live Tables Expectations, used for managing data quality, is one such feature.

SILVER TABLES

The following function creates a Silver Streaming Table for the given game name provided as a parameter:

```
def build_silver(gname):
    .table(name=f"silver_{gname}_events")
    def gold_unified():
        return dlt.read_stream("bronze_events").where(F.col("game_name") == gname)
```

Notice the use of the @dlt.table annotation. Thanks to this annotation, when build_silver is invoked for a given gname, a DLT table will be defined that depends on the source bronze_events table. We know that the tables created by this function will be Streaming Tables because of the use of dlt.read_stream.

GOLD TABLES

The following function creates a Gold Materialized View for the given game name provided as a parameter:

```
def build_gold(gname):
    .table(name=f"gold_{gname}_player_agg")
    def gold_unified():
        return (
            dlt.read(f"silver_{gname}_events")
            .groupBy(["gamer_id"])
            .agg(
                F.count("*").alias("session_count"),
                F.min(F.col("event_timestamp")).alias("min_timestamp"),
                F.max(F.col("event_timestamp")).alias("max_timestamp")
            )
        )
```

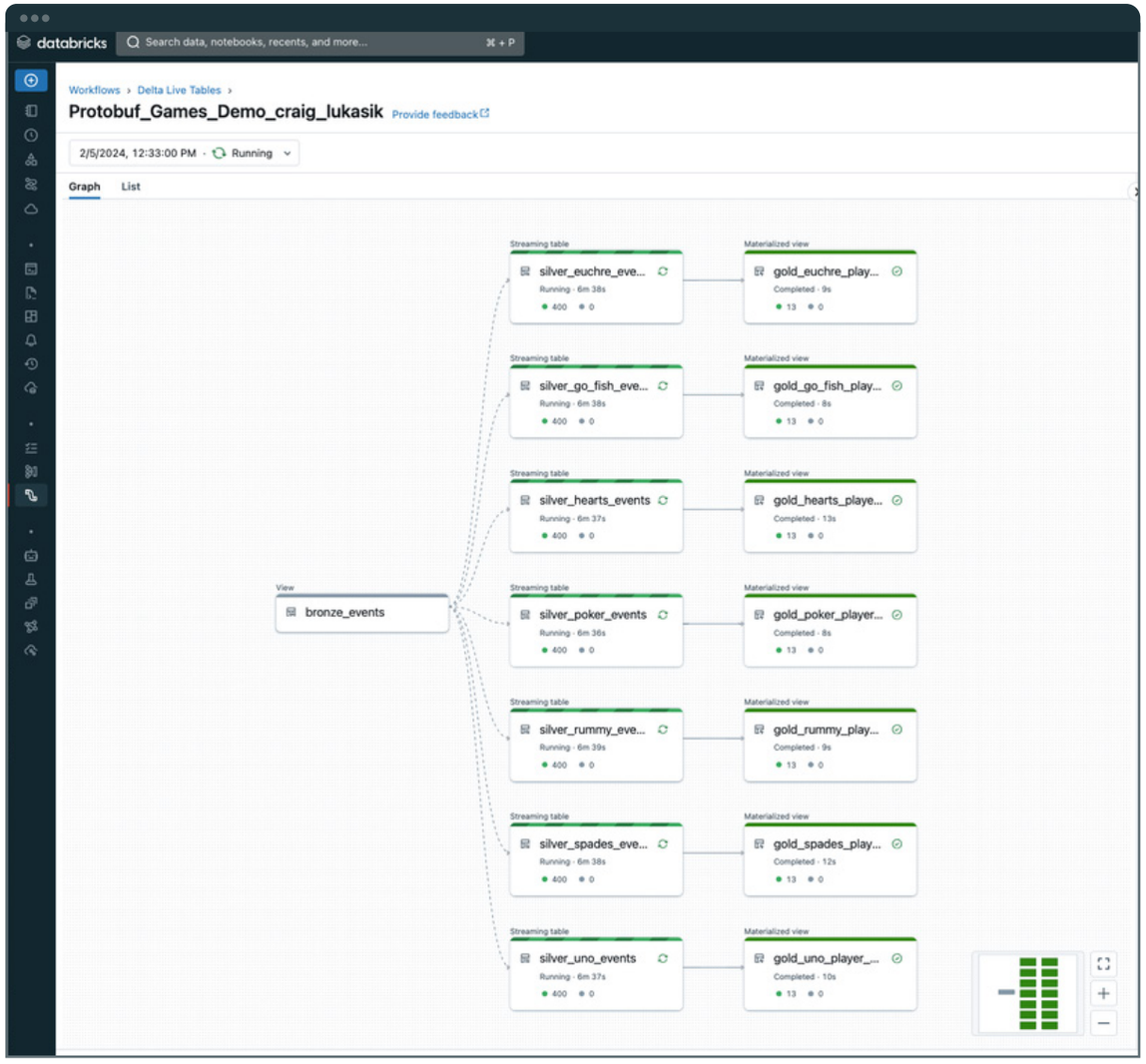
We know the resulting table will be a "Materialized View" because of the use of dlt.read. This is a simple Materialized View definition; it simply performs a count of source events along with min and max event times, grouped by gamer_id.

METADATA-DRIVEN TABLES

The previous two sections of this article defined functions for creating Silver (Streaming) Tables and Gold Materialized Views. The metadata-driven approach in the example code uses a pipeline input parameter to create N*2 target tables (one Silver table for each game and one aggregate Gold table for each game). This code drives the dynamic table creation using the aforementioned build_silver and build_gold functions:

```
1  GAMES_ARRAY = spark.conf.get("games").split(",")
2
3  for game in GAMES_ARRAY:
4      build_silver(game)
       build_gold(game)
```

At this point, you might have noticed that much of the control flow code data engineers often have to write is absent. This is because, as mentioned above, DLT is a declarative programming framework. It automatically detects dependencies and manages the pipeline's execution flow. Here's the DAG that DLT creates for the pipeline:



A note about aggregates in a streaming pipeline

For a continuously running stream, calculating some aggregates can be very resource-intensive. Consider a scenario where you must calculate the "median" for a continuous stream of numbers. Every time a new number arrives in the stream, the median calculation will need to explore the entire set of numbers that have ever arrived. In a stream receiving millions of numbers per second, this fact can present a significant challenge if your goal is to provide a destination table for the median of the entire stream of numbers. It becomes impractical to perform such a feat every time a new number arrives. The limits of computation power and persistent storage and network would mean that the stream would continue to grow a backlog much faster than it could perform the calculations.

In a nutshell, it would not work out well if you had such a stream and tried to recalculate the median for the universe of numbers that have ever arrived in the stream. So, what can you do? If you look at the code snippet above, you may notice that this problem is not addressed in the code! Fortunately, as a Delta Live Tables developer, I do not have to worry about it. The declarative framework handles this dilemma by design. DLT addresses this by materializing results only periodically. Furthermore, DLT provides a table property that allows the developer to set an appropriate trigger interval.

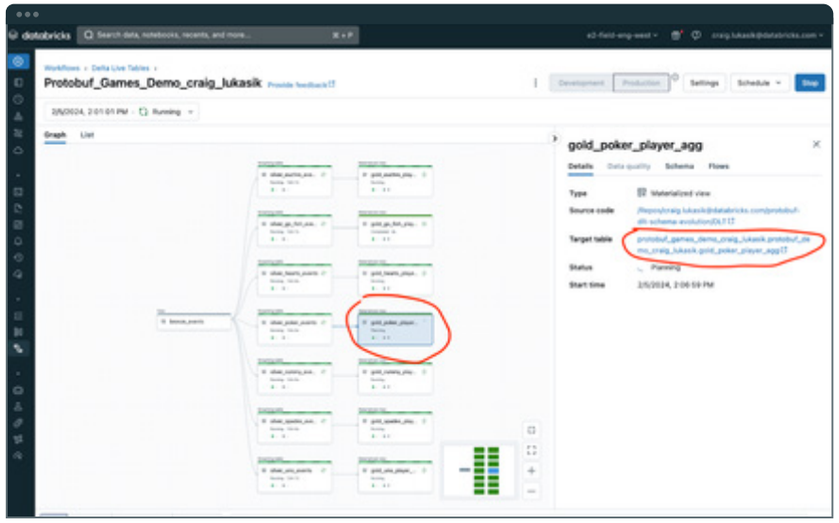
REVIEWING THE BENEFITS OF DLT

Governance

Unity Catalog governs the end-to-end pipeline. Thus, permission to target tables can be granted to end-users and service principals needing access across any Databricks workspaces attached to the same metastore.

Lineage

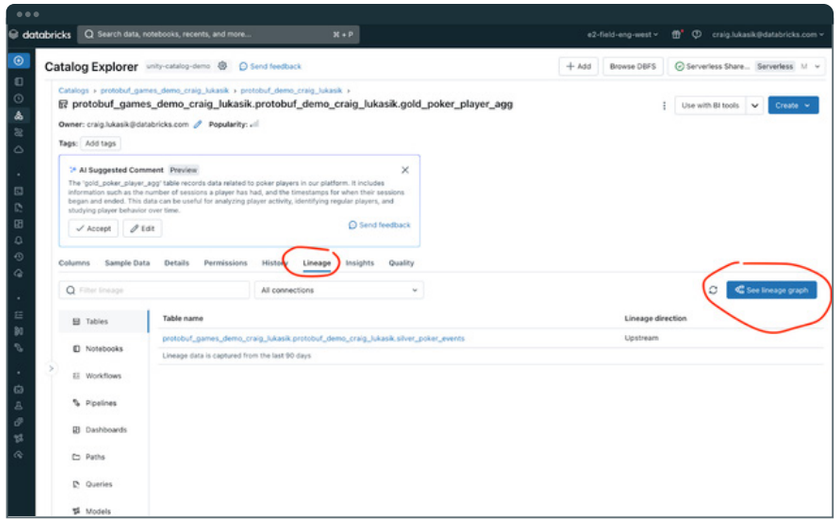
From the Delta Live Tables interface, we can navigate to the Catalog and view lineage.



Click on a table in the DAG. Then click on the "Target table" link.

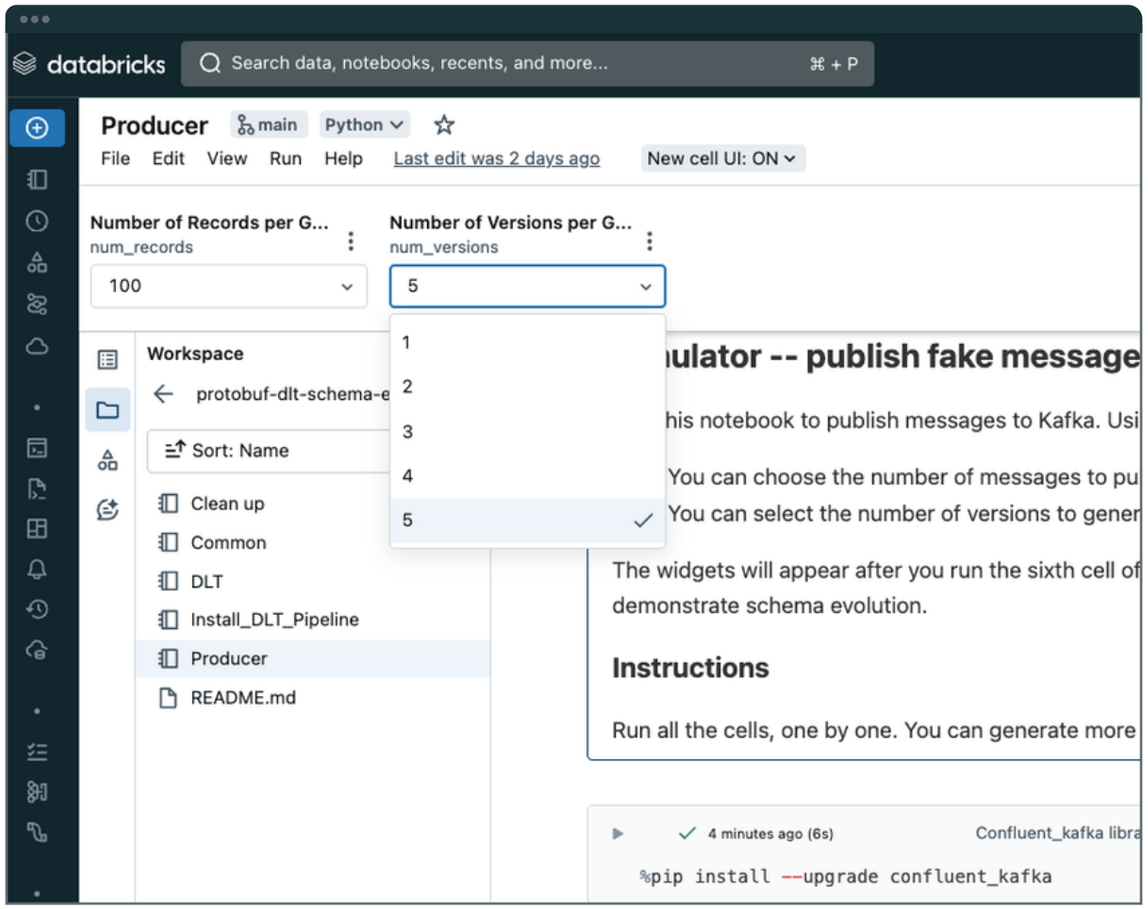
Click on the "Lineage" tab for the table. Then click on the "See lineage graph" link.

Lineage also provides visibility into other related artifacts, such as notebooks, models, etc.



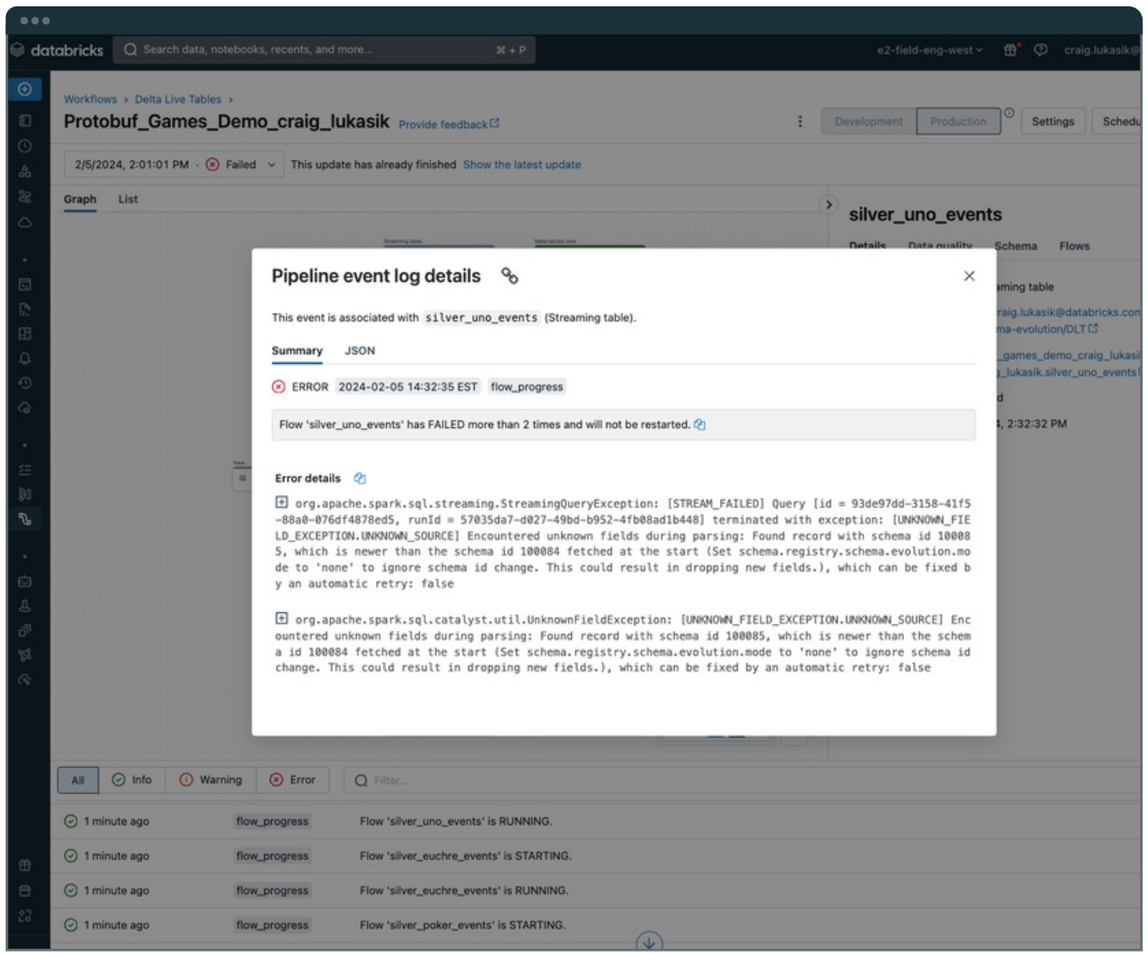
This lineage helps accelerate team velocity by making it easier to understand how assets in the workspace are related.

HANDS-OFF SCHEMA EVOLUTION

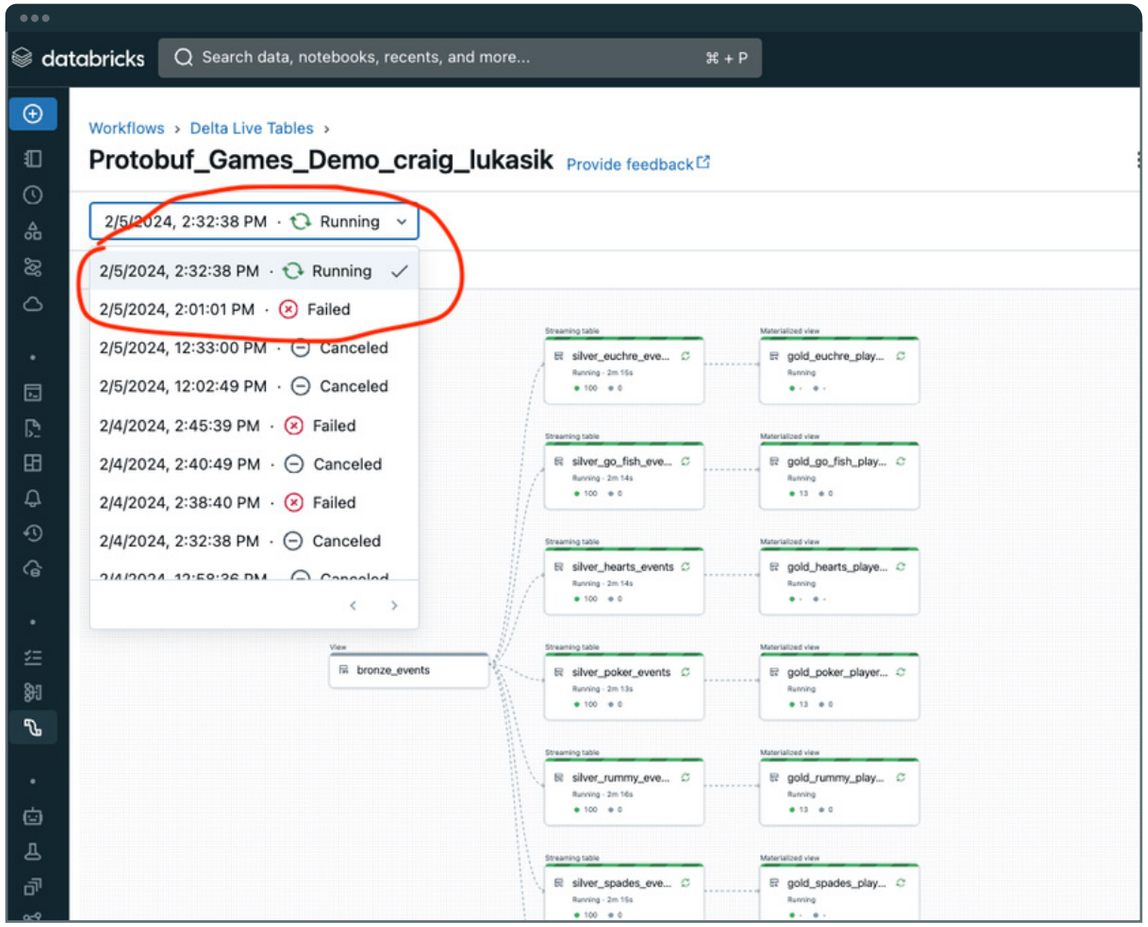


Delta Live Tables will detect this as the source stream's schema evolves, and the pipeline will restart. To simulate a schema evolution for this example, you would run the Producer notebook a subsequent time but with a larger value for num_versions, as shown on the left. This will generate new data where the schema includes some additional columns. The Producer notebook updates the schema details in the Confluent Schema Registry.

When the schema evolves, you will see a pipeline failure like this one:



If the Delta Live Tables pipeline runs in Production mode, a failure will result in an automatic pipeline restart. The Schema Registry will be contacted upon restart to retrieve the latest schema definitions. Once back up, the stream will continue with a new run:



CONCLUSION

In high-performance IoT systems, optimization extends through every layer of the technology stack, focusing on the payload format of messages in transit. Throughout this article, we've delved into the benefits of using an optimized serialization format, protobuf, and demonstrated its integration with Databricks to construct a comprehensive end-to-end demultiplexing pipeline. This approach underlines the importance of selecting the right tools and formats to maximize efficiency and effectiveness in IoT systems.

Instructions for running the example

To run the example code, follow these instructions:

1. In Databricks, clone this repo: https://github.com/craig-db/protobuf-dlt-schema-evolution.
2. Set up the prerequisites (documented below).
3. Follow the instructions in the README notebook included in the repo code.

Prerequisites

1. A Unity Catalog-enabled workspace — this demo uses a Unity Catalog-enabled Delta Live Tables pipeline. Thus, Unity Catalog should be configured for the workspace where you plan to run the demo.
2. As of January 2024, you should use the Preview channel for the Delta Live Tables pipeline. The "Install_DLT_Pipeline" notebook will use the Preview channel when installing the pipeline.
3. Confluent account – this demo uses Confluent Schema Registry and Confluent Kafka.

Secrets to configure

The following Kafka and Schema Registry connection details (and credentials) should be saved as Databricks Secrets and then set within the Secrets notebook that is part of the repo:

- **SR_URL:** Schema Registry URL (e.g. https://myschemaregistry.aws.confluent.cloud)
- **SR_API_KEY:** Schema Registry API Key
- **SR_API_SECRET:** Schema Registry API Secret
- **KAFKA_KEY:** Kafka API Key
- **KAFKA_SECRET:** Kafka Secret
- **KAFKA_SERVER:** Kafka host:port (e.g. mykafka.aws.confluent.cloud:9092)
- **KAFKA_TOPIC:** The Kafka Topic
- **TARGET_SCHEMA:** The target database where the streaming data will be appended into a Delta table (the destination table is named unified_gold)
- **CHECKPOINT_LOCATION:** Some location (e.g., in DBFS) where the checkpoint data for the streams will be stored

Go here to learn how to save secrets to secure sensitive information (e.g., credentials) within the Databricks Workspace: https://docs.databricks.com/security/secrets/index.html.

Design Patterns for Batch Processing in Financial Services

by Eon Retief

Financial services institutions (FSIs) around the world are facing unprecedented challenges ranging from market volatility and political uncertainty to changing legislation and regulations. Businesses are forced to accelerate digital transformation programs; automating critical processes to reduce operating costs and improve response times. However, with data typically scattered across multiple systems, accessing the information required to execute on these initiatives tends to be easier said than done.

Architecting an ecosystem of services able to support the plethora of data-driven use cases in this digitally transformed business can, however, seem to be an impossible task. This chapter will focus on one crucial aspect of the modern data stack: batch processing. A seemingly outdated paradigm, we'll see why batch processing remains a vital and highly viable component of the data architecture. And we'll see how Databricks can help FSIs navigate some of the crucial challenges faced when building infrastructure to support these scheduled or periodic workflows.

WHY BATCH INGESTION MATTERS

Over the last two decades, the global shift towards an instant society has forced organizations to rethink the operating and engagement model. A digital-first strategy is no longer optional but vital for survival. Customer needs and demands are changing and evolving faster than ever. This desire for instant gratification is driving an increased focus on building capabilities that support real-time processing and decisioning. One might ask whether batch processing is still relevant in this new dynamic world.

While real-time systems and streaming services can help FSIs remain agile in addressing the volatile market conditions at the edge, they do not typically meet the requirements of back-office functions. Most business decisions are not reactive but rather, require considered, strategic reasoning. By definition, this approach requires a systematic review of aggregate data collected over a period of time. Batch processing in this context still provides the most efficient and cost-effective method for processing large, aggregate volumes of data. Additionally, batch processing can be done offline, reducing operating costs and providing greater control over the end-to-end process.

The world of finance is changing, but across the board incumbents and startups continue to rely heavily on batch processing to power core business functions. Whether for reporting and risk management or anomaly detection and surveillance, FSIs require batch processing to reduce human error, increase the speed of delivery, and reduce operating costs.

GETTING STARTED

Starting with a 30,000-ft view, most FSIs will have a multitude of data sources scattered across on-premises systems, cloud-based services and even third-party applications. Building a batch ingestion framework that caters for all these connections require complex engineering and can quickly become a burden on maintenance teams. And that's even before considering things like change data capture (CDC), scheduling, and schema evolution. In this section, we will demonstrate how the Databricks Lakehouse for Financial Services (LFS) and its ecosystem of partners can be leveraged to address these key challenges and greatly simplify the overall architecture.

The Databricks lakehouse architecture was designed to provide a unified platform that supports all analytical and scientific data workloads. Figure 1 shows the reference architecture for a decoupled design that allows easy integration with other platforms that support the modern data ecosystem. The lakehouse makes it easy to construct ingestion and serving layers that operate irrespective of the data's source, volume, velocity, and destination.

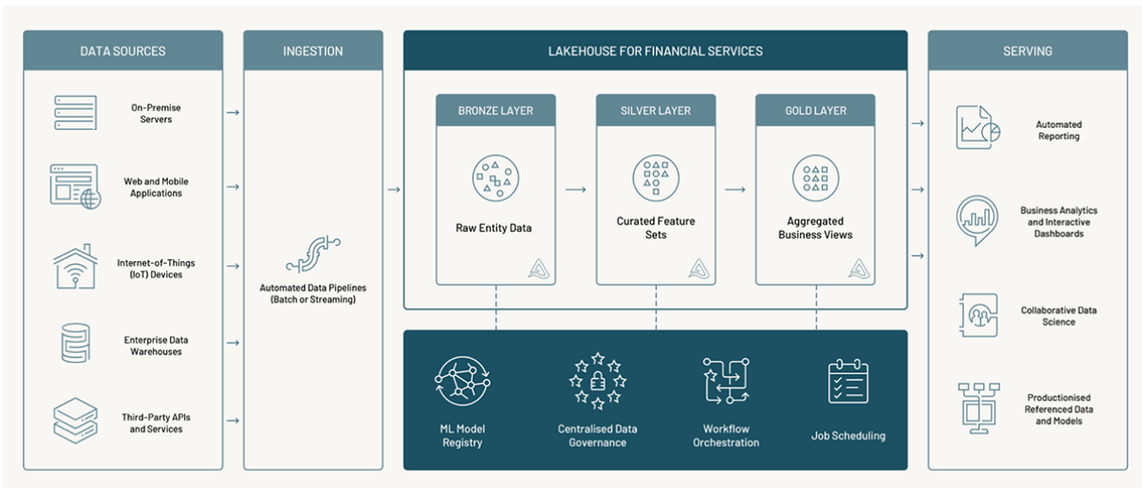


Figure 1: Reference architecture of the Lakehouse for Financial Services

To demonstrate the power and efficiency of the LFS, we turn to the world of insurance. We consider the basic reporting requirements for a typical claims workflow. In this scenario, the organization might be interested in the key metrics driven by claims processes. For example:

- Number of active policies
- Number of claims
- Value of claims
- Total exposure
- Loss ratio

Additionally, the business might want a view of potentially suspicious claims and a breakdown by incident type and severity. All these metrics are easily calculable from two key sources of data: 1) the book of policies and 2) claims filed by customers. The policy and claims records are typically stored in a combination of enterprise data warehouses (EDWs) and operational databases. The main challenge becomes connecting to these sources and ingesting data into our lakehouse, where we can leverage the power of Databricks to calculate the desired outputs.

Luckily, the flexible design of the LFS makes it easy to leverage best-in-class products from a range of SaaS technologies and tools to handle specific tasks. One possible solution for our claims analytics use case would be to use Fivetran for the batch ingestion plane. Fivetran provides a simple and secure platform for connecting to numerous data sources and delivering data directly to the Databricks lakehouse. Additionally, it offers native support for CDC, schema evolution and workload scheduling. In Figure 2, we show the technical architecture of a practical solution for this use case.

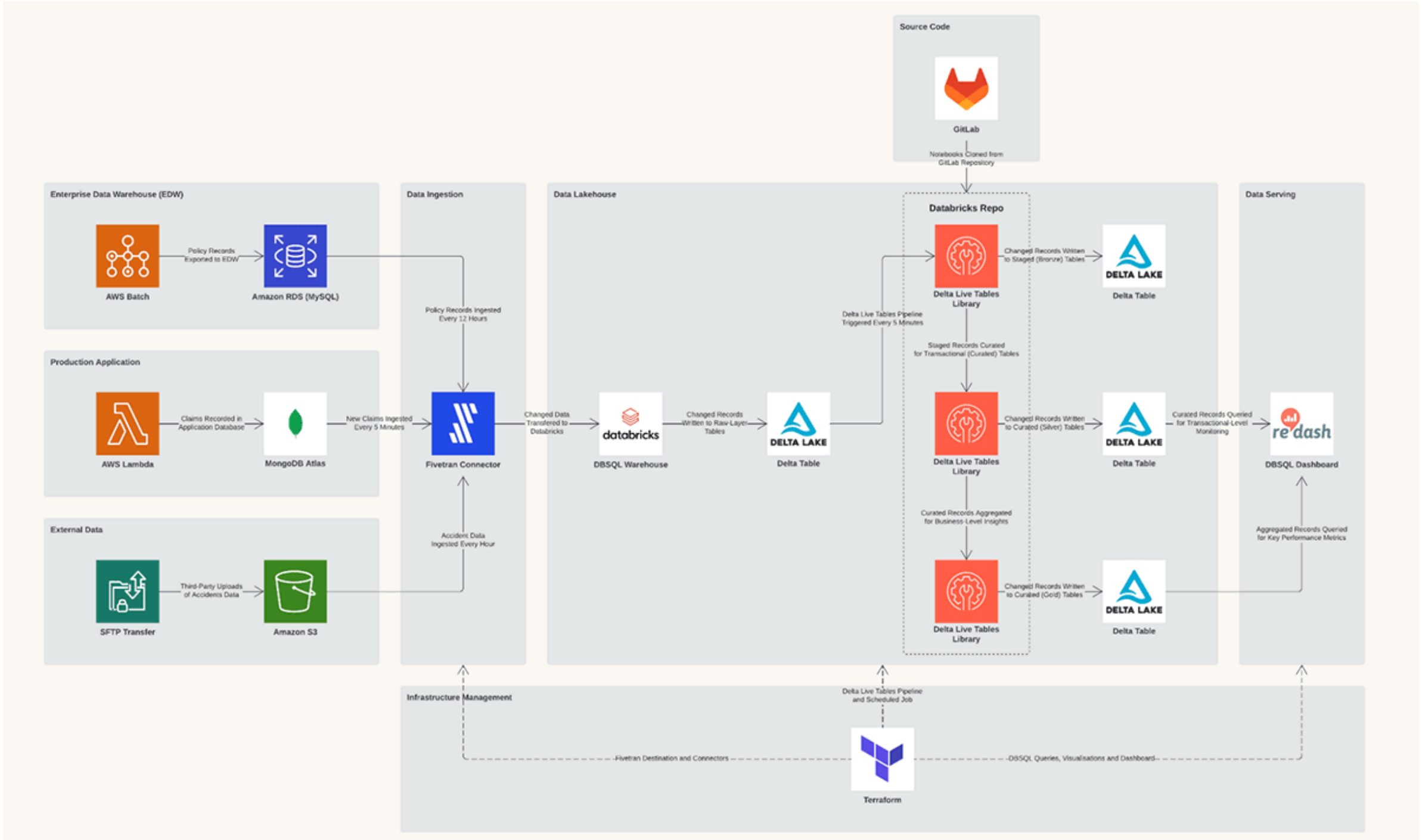


Figure 2: Technical architecture for a simple insurance claims workflow

Once the data is ingested and delivered to the LFS, we can use Delta Live Tables (DLT) for the entire engineering workflow. DLT provides a simple, scalable declarative framework for automating complex workflows and enforcing data quality controls. The outputs from our DLT workflow, our curated and aggregated assets, can be interrogated using Databricks SQL (DB SQL). DB SQL brings data warehousing to the LFS to power business-critical analytical workloads. Results from DB SQL queries can be packaged in easy-to-consume dashboards and served to business users.

STEP 1: CREATING THE INGESTION LAYER

Setting up an ingestion layer with Fivetran requires a two-step process. First, configuring a so-called *destination* where data will be delivered, and second, establishing one or more connections with the source systems. The Partner Connect interface takes care of the first step with a simple, guided interface to connect Fivetran with a Databricks Warehouse. Fivetran will use the warehouse to convert raw source data to Delta Tables and store the results in the Databricks Lakehouse. Figures 3 and 4 show steps from the Partner Connect and Fivetran interfaces to configure a new destination.

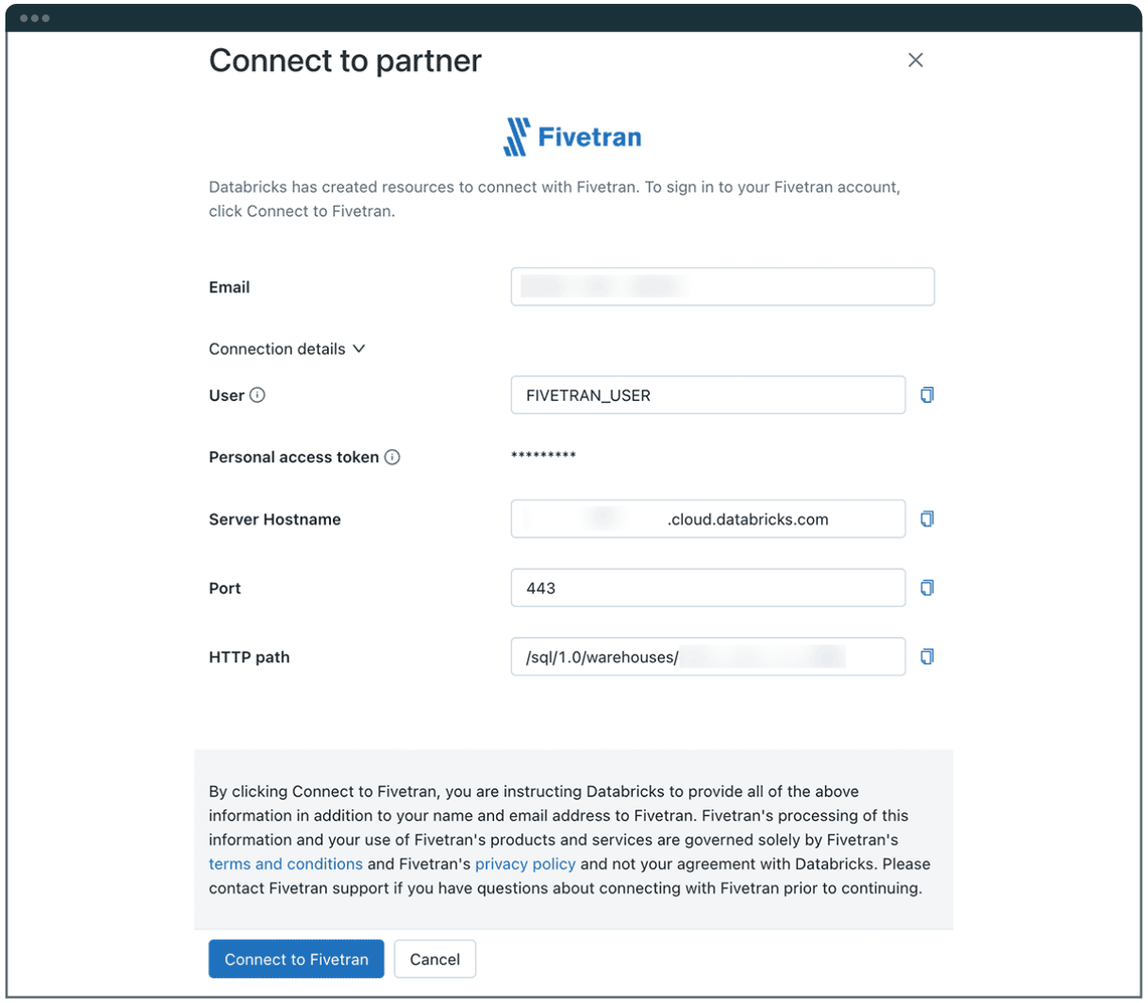


Figure 3: Databricks Partner Connect interface for creating a new connection

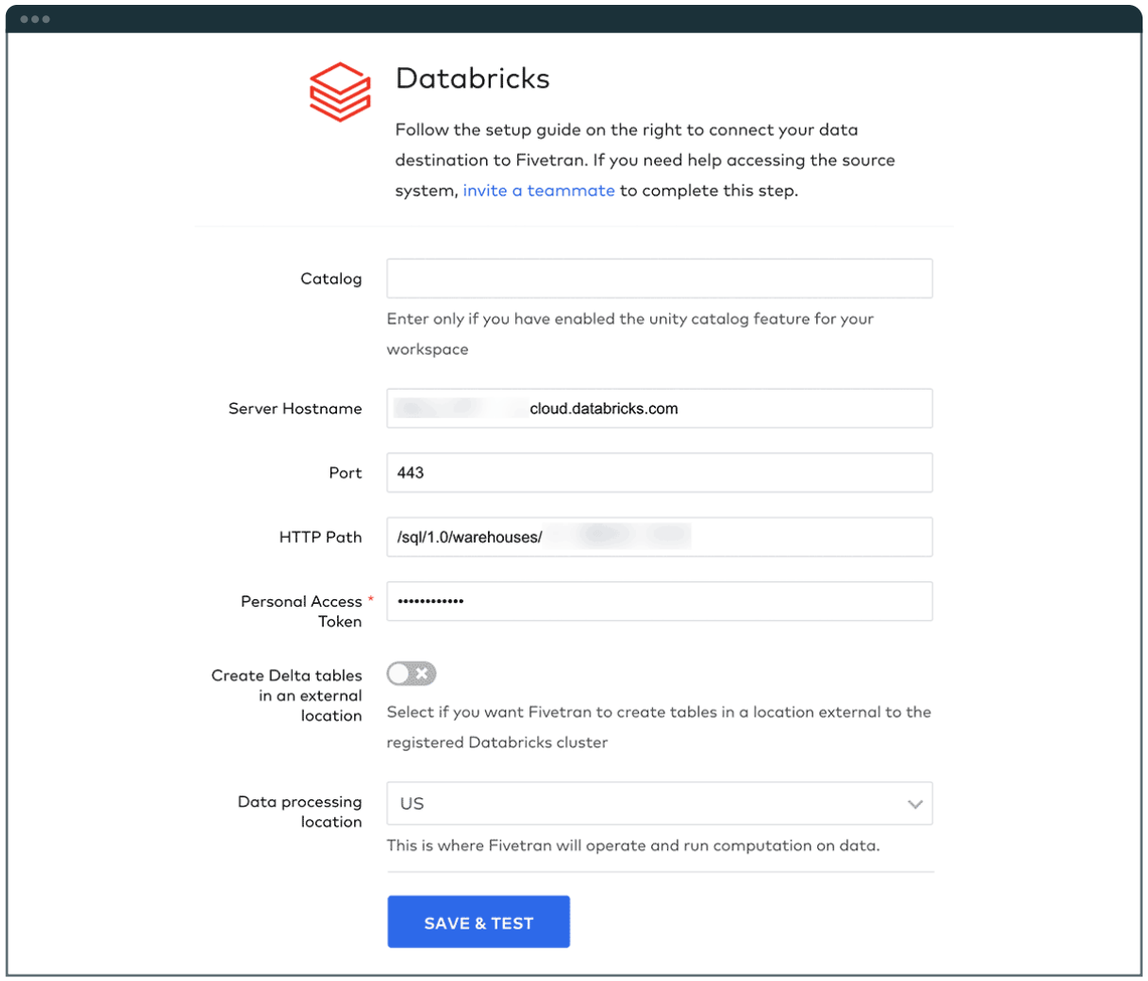


Figure 4: Fivetran interface for confirming a new destination

For the next step, we move to the Fivetran interface. From here, we can easily create and configure connections to several different source systems (please refer to the official documentation for a complete list of all supported connections). In our example, we consider three sources of data: 1) policy records stored in an Operational Data Store (ODS) or Enterprise Data Warehosue (EDW), 2) claims records stored in an operational database, and 3) external data delivered to blob storage. As such, we require three different connections to be configured in Fivetran. For each of these, we can follow Fivetran's simple guided process to set up a connection with the source system. Figures 5 and 6 show how to configure new connections to data sources.

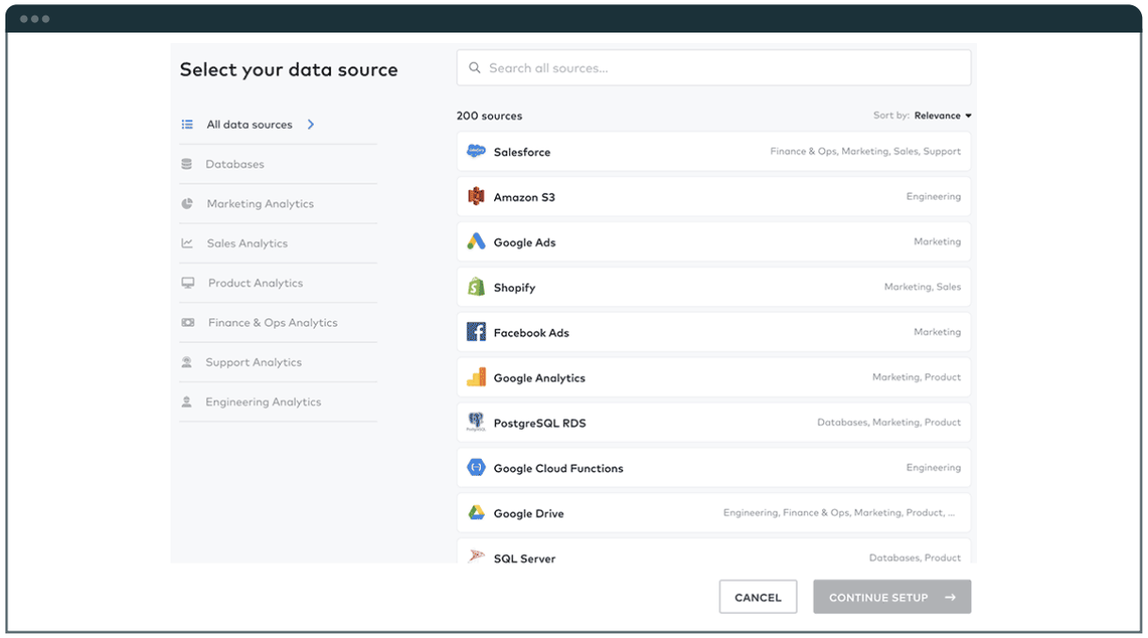


Figure 5: Fivetran interface for selecting a data source type

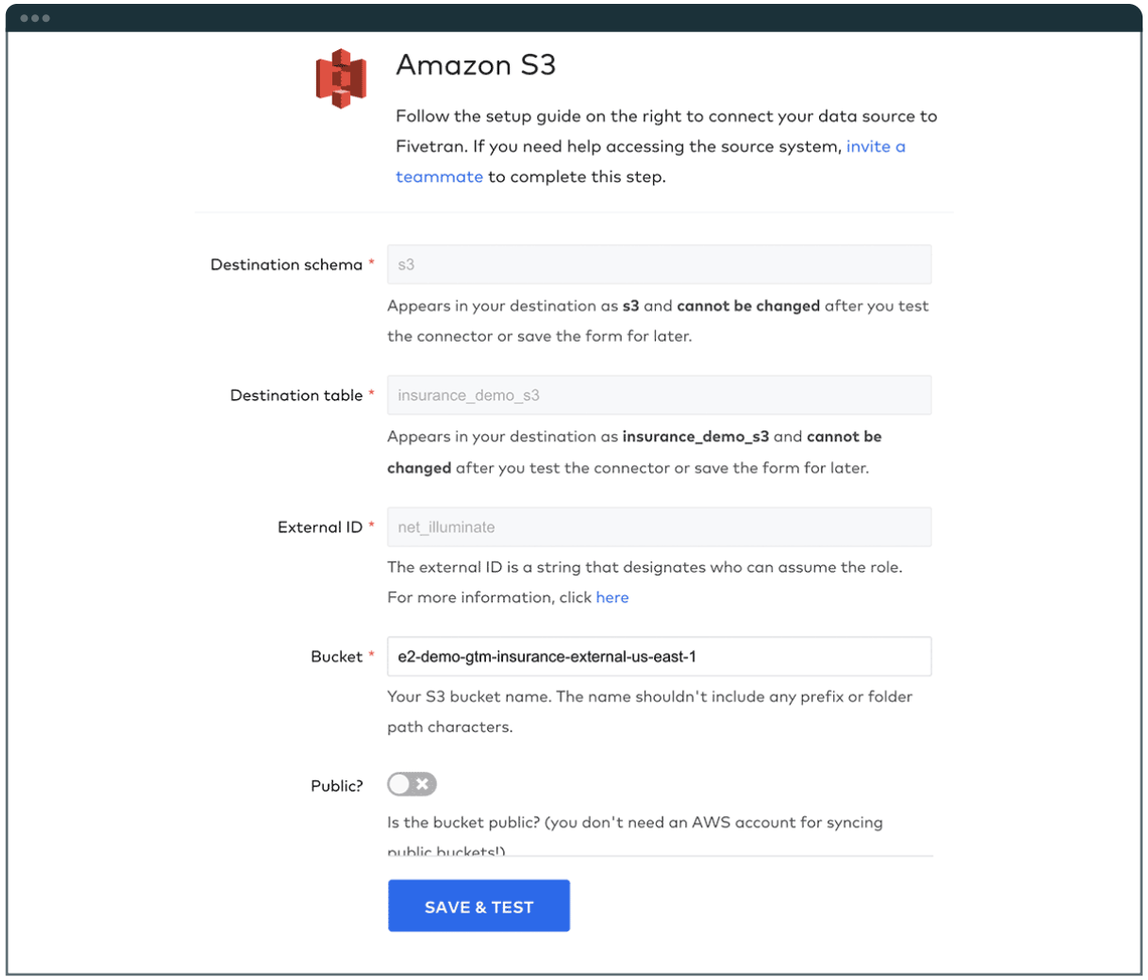


Figure 6: Fivetran interface for configuring a data source connection

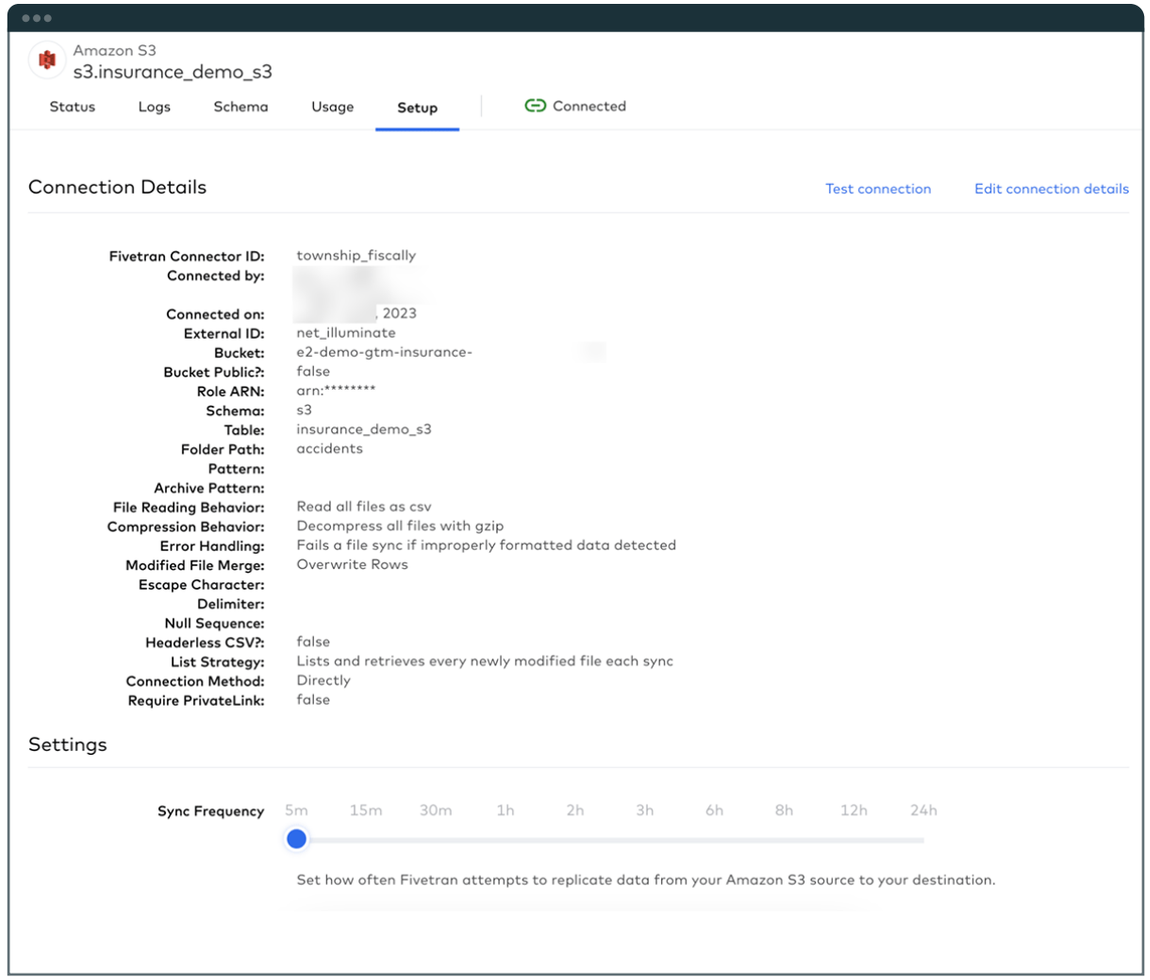


Figure 7: Overview of configuration for a Fivetran connect

Connections can further be configured once they have been validated. One important option to set is the frequency with which Fivetran will interrogate the source system for new data. In Figure 7, we can see how easy Fivetran has made it to set the sync frequency with intervals ranging from 5 minutes to 24 hours.

Fivetran will immediately interrogate and ingest data from source systems once a connection is validated. Data is stored as Delta tables and can be viewed from within Databricks through the DB SQL Data Explorer. By default, Fivetran will store all data under the Hive metastore. A new schema is created for each new connection, and each schema will contain at least two tables: one containing the data and another with logs from each attempted ingestion cycle (see Figure 8).

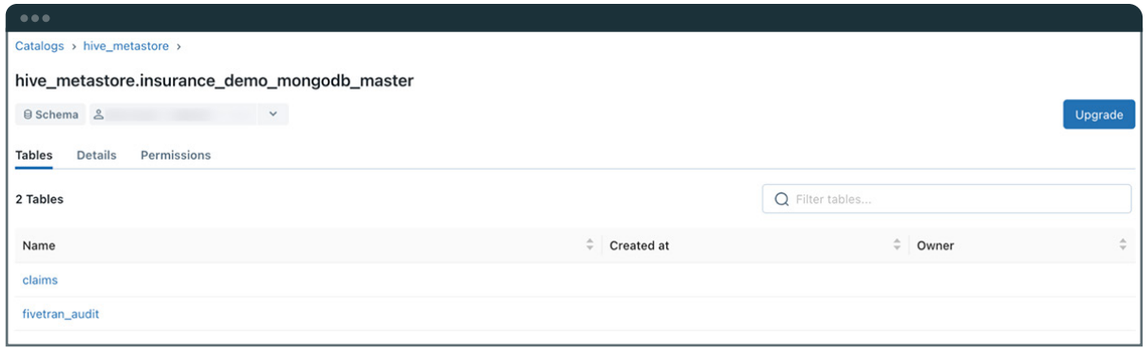


Figure 8: Summary of tables created by Fivetran in the Databricks Warehouse for an example connection

Having the data stored in Delta tables is a significant advantage. Delta Lake natively supports granular data versioning, meaning we can time travel through each ingestion cycle (see Figure 9). We can use DB SQL to interrogate specific versions of the data to analyze how the source records evolved.

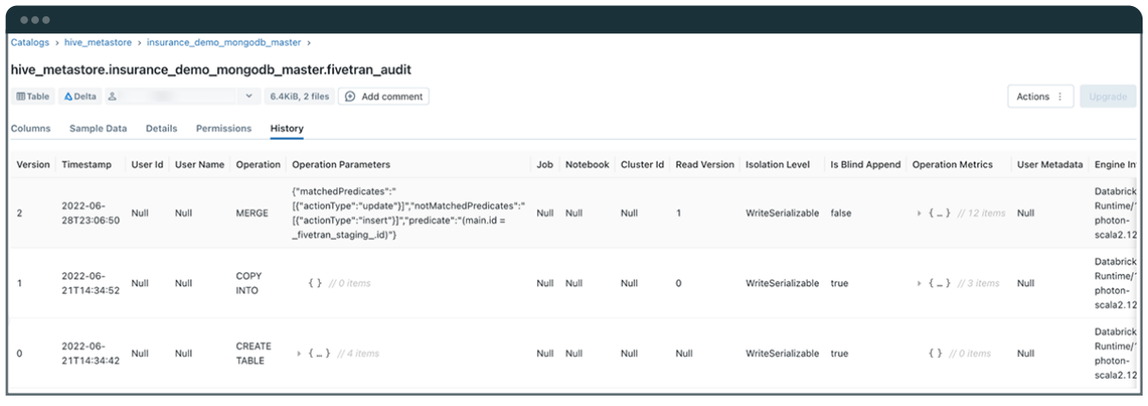


Figure 9: View of the history showing changes made to the Fivetran audit table

It is important to note that if the source data contains semi-structured or unstructured values, those attributes will be flattened during the conversion process. This means that the results will be stored in grouped text-type columns, and these entities will have to be dissected and unpacked with DLT in the curation process to create separate attributes.

STEP 2: AUTOMATING THE WORKFLOW

With the data in the Databricks Data Intelligence Platform, we can use Delta Live Tables (DLT) to build a simple, automated data engineering workflow. DLT provides a declarative framework for specifying detailed feature engineering steps. Currently, DLT supports APIs for both Python and SQL. In this example, we will use Python APIs to build our workflow.

The most fundamental construct in DLT is the definition of a table. DLT interrogates all table definitions to create a comprehensive workflow for how data should be processed. For instance, in Python, tables are created using function definitions and the `dlt.table` decorator (see example of Python code below). The decorator is used to specify the name of the resulting table, a descriptive comment explaining the purpose of the table, and a collection of table properties.

```
@dlt.table(
   name             = "curated_claims",
   comment          = "Curated claim records",
   table_properties = {
       "layer": "silver",
       "pipelines.autoOptimize.managed": "true",
       "delta.autoOptimize.optimizeWrite": "true",
       "delta.autoOptimize.autoCompact": "true"
   }
)
def curate_claims():
   # Read the staged claim records into memory
   staged_claims = dlt.read("staged_claims")
   # Unpack all nested attributes to create a flattened table structure
   curated_claims = unpack_nested(df = staged_claims, schema = schema_claims)

   ...
```

Instructions for feature engineering are defined inside the function body using standard PySpark APIs and native Python commands. The following example shows how PySpark joins claims records with data from the policies table to create a single, curated view of claims.

```
...

   # Read the staged claim records into memory
   curated_policies = dlt.read("curated_policies")
   # Evaluate the validity of the claim
   curated_claims = curated_claims \
       .alias("a") \
       .join(
           curated_policies.alias("b"),
           on  = F.col("a.policy_number") == F.col("b.policy_number"),
           how = "left"
       ) \
       .select([F.col(f"a.{c}") for c in curated_claims.columns] + [F.col(f"b.{c}").
alias(f"policy_{c}")  for c in ("effective_date", "expiry_date")]) \
       .withColumn(
           # Calculate the number of months between coverage starting and the
claim being filed
           "months_since_covered", F.round(F.months_between(F.col("claim_date"),
F.col("policy_effective_date")))
       ) \
       .withColumn(
           # Check if the claim was filed before the policy came into effect
           "claim_before_covered", F.when(F.col("claim_date") < F.col("policy_
effective_date"), F.lit(1)).otherwise(F.lit(0))
       ) \
       .withColumn(
           # Calculate the number of days between the incident occurring and the
claim being filed
           "days_between_incident_and_claim", F.datediff(F.col("claim_date"),
F.col("incident_date"))
       )

   # Return the curated dataset
   return curated_claims
```

One significant advantage of DLT is the ability to specify and enforce data quality standards. We can set expectations for each DLT table with detailed data quality constraints that should be applied to the contents of the table. Currently, DLT supports expectations for three different scenarios:

DECORATOR	DESCRIPTION
expect	Retain records that violate expectations
expect_or_drop	Drop records that violate expectations
expect_or_fail	Halt the execution if any record(s) violate constraints

Expectations can be defined with one or more data quality constraints. Each constraint requires a description and a Python or SQL expression to evaluate. Multiple constraints can be defined using the expect_all, expect_all_or_drop, and expect_all_or_fail decorators. Each decorator expects a Python dictionary where the keys are the constraint descriptions, and the values are the respective expressions. The example below shows multiple data quality constraints for the retain and drop scenarios described above.

```python
@dlt.expect_all({
    "valid_driver_license": "driver_license_issue_date > (current_date() -
cast(cast(driver_age AS INT) AS INTERVAL YEAR))",
    "valid_claim_amount": "total_claim_amount > 0",
    "valid_coverage": "months_since_covered > 0",
    "valid_incident_before_claim": "days_between_incident_and_claim > 0"
})
@dlt.expect_all_or_drop({
    "valid_claim_number": "claim_number IS NOT NULL",
    "valid_policy_number": "policy_number IS NOT NULL",
    "valid_claim_date": "claim_date < current_date()",
    "valid_incident_date": "incident_date < current_date()",
    "valid_incident_hour": "incident_hour between 0 and 24",
    "valid_driver_age": "driver_age > 16",
    "valid_effective_date": "policy_effective_date < current_date()",
    "valid_expiry_date": "policy_expiry_date <= current_date()"
})
def curate_claims():
    ...
```

We can use more than one Databricks Notebook to declare our DLT tables. Assuming we follow the medallion architecture, we can, for example, use different notebooks to define tables comprising the bronze, silver, and gold layers. The DLT framework can digest instructions defined across multiple notebooks to create a single workflow; all inter-table dependencies and relationships are processed and considered automatically. Figure 10 shows the complete workflow for our claims example. Starting with three source tables, DLT builds a comprehensive pipeline that delivers thirteen tables for business consumption.

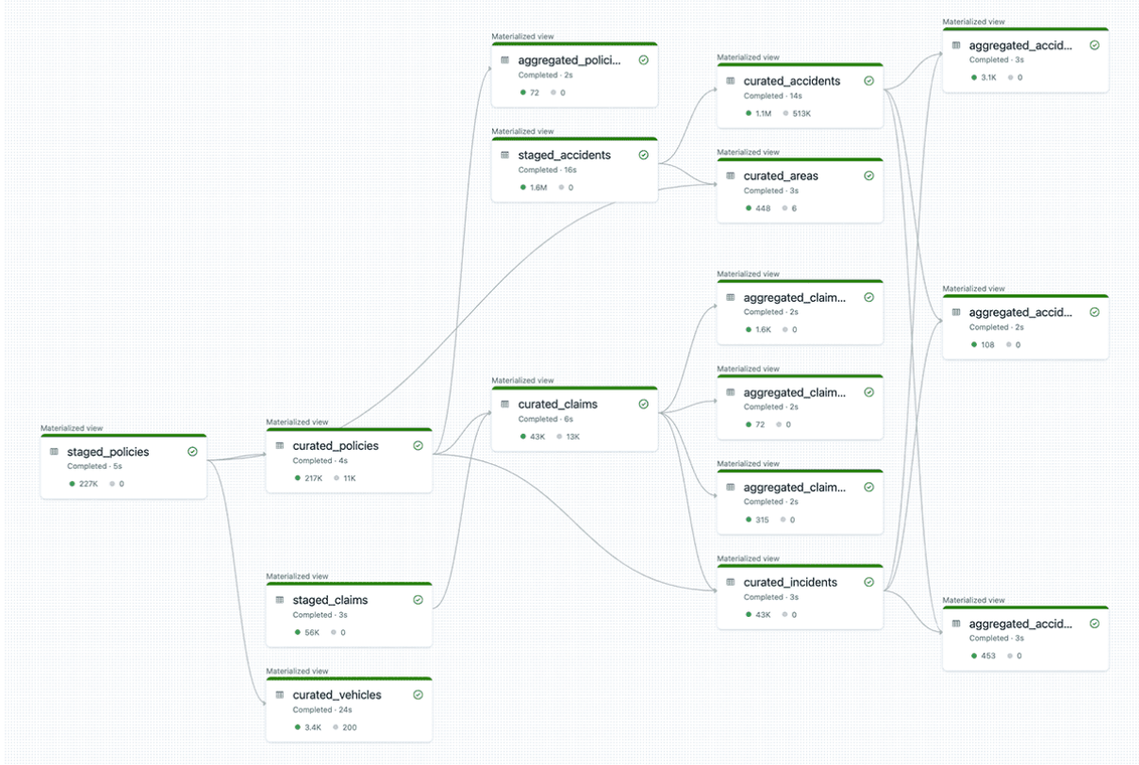


Figure 10: Overview of a complete Delta Live Tables (DLT) workflow

Results for each table can be inspected by selecting the desired entity. Figure 11 provides an example of the results of the curated claims table. DLT provides a high-level overview of the results from the data quality controls:

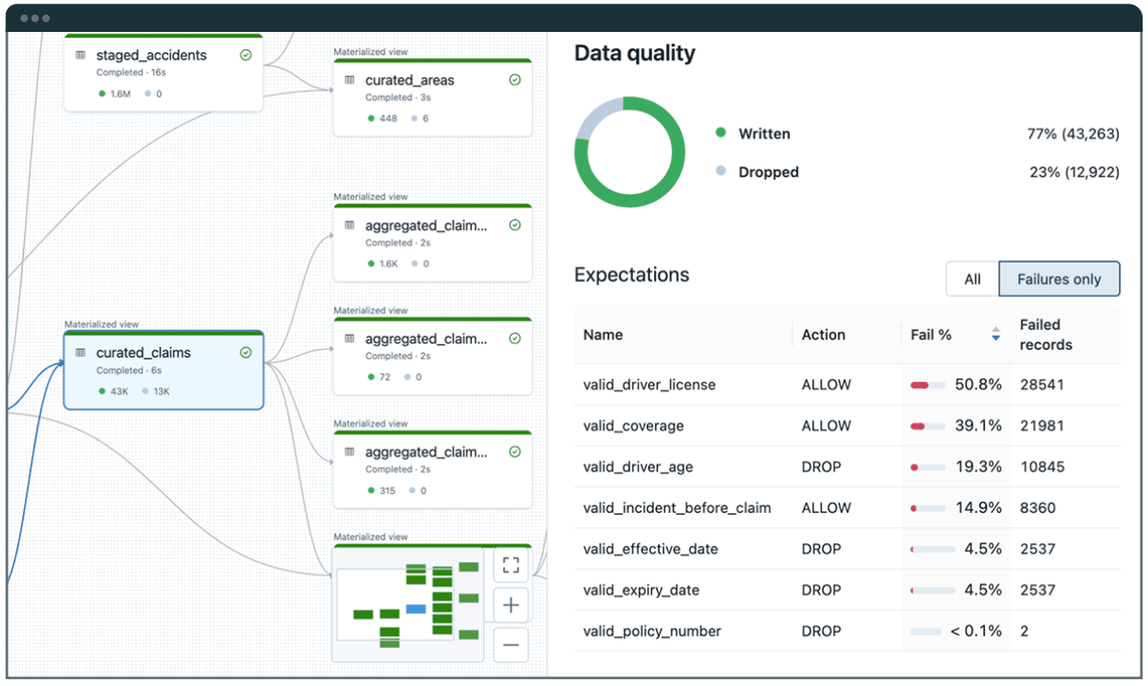


Figure 11: Example of detailed view for a Delta Live Tables (DLT) table entity with the associated data quality report

Results from the data quality expectations can be analyzed further by querying the event log. The event log contains detailed metrics about all expectations defined for the workflow pipeline. The query below provides an example for viewing key metrics from the last pipeline update, including the number of records that passed or failed expectations:

```
SELECT
  row_expectations.dataset AS dataset,
  row_expectations.name AS expectation,
  SUM(row_expectations.passed_records) AS passing_records,
  SUM(row_expectations.failed_records) AS failing_records
FROM
  (
    SELECT
      explode(
        from_json(
          details :flow_progress :data_quality :expectations,
          "array<struct<name: string, dataset: string, passed_records: int, failed_
records: int>>"
        )
      ) row_expectations
    FROM
      event_log_raw
    WHERE
      event_type = 'flow_progress'
      AND origin.update_id = '${latest_update.id}'
  )
GROUP BY
  row_expectations.dataset,
  row_expectations.name;
```

Again, we can view the complete history of changes made to each DLT table by looking at the Delta history logs (see Figure 12). It allows us to understand how tables evolve over time and investigate complete threads of updates if a pipeline fails.

Columns | Sample Data | Details | Permissions | History

Version	Timestamp	User Id	User Name	Operation	Operation Parameters	Job	Notebook	Cluster Id	Read Version	Isolation Level	Is Blind Append	Operation Metrics	User Metadata	Engine Info
12	2023-01-22T19:10:09	Null	Null	WRITE	{ … } // 1 item	Null	Null	Null	11	WriteSerializable	false	{ … } // 3 items	Null	Databricks-Runtime/dlt:11.0-delta-pipelines-1eca0d9-750b289-9ea72db-custom-local
11	2022-12-09T11:48:23	Null	Null	WRITE	{ … } // 1 item	Null	Null	Null	10	WriteSerializable	false	{ … } // 3 items	Null	Databricks-Runtime/dlt:11.0-delta-pipelines-ed5cc83-e81c5c7-17c692e-custom-local
10	2022-11-08T19:48:31	Null	Null	WRITE	{ … } // 1 item	Null	Null	Null	9	WriteSerializable	false	{ … } // 3 items	Null	Databricks-Runtime/dlt:11.0-delta-pipelines-de92f9e-8a33b70-a333f54-custom-local

Figure 12: View the history of changes made to a resulting Delta Live Tables (DLT) table entity

We can further use change data capture (CDC) to update tables based on changes in the source datasets. DLT CDC supports updating tables with slow-changing dimensions (SCD) types 1 and 2.

We have one of two options for our batch process to trigger the DLT pipeline. We can use the Databricks Auto Loader to incrementally process new data as it arrives in the source tables or create scheduled jobs that trigger at set times or intervals. In this example, we opted for the latter with a scheduled job that executes the DLT pipeline every five minutes.

OPERATIONALIZING THE OUTPUTS

The ability to incrementally process data efficiently is only half of the equation. Results from the DLT workflow must be operationalized and delivered to business users. In our example, we can consume outputs from the DLT pipeline through ad hoc analytics or prepacked insights made available through an interactive dashboard.

AD HOC ANALYTICS

Databricks SQL (or DB SQL) provides an efficient, cost-effective data warehouse on top of the Data Intelligence Platform. It allows us to run our SQL workloads directly against the source data with up to 12x better price/performance than its alternatives.

We can leverage DB SQL to perform specific ad hoc queries against our curated and aggregated tables. We might, for example, run a query against the curated policies table that calculates the total exposure. The DB SQL query editor provides a simple, easy-to-use interface to build and execute such queries (see example below).

```
SELECT
  round(curr.total_exposure, 0) AS total_exposure,
  round(prev.total_exposure, 0) AS previous_exposure
FROM
  (
    SELECT
      sum(sum_insured) AS total_exposure
    FROM
      insurance_demo_lakehouse.curated_policies
    WHERE
      expiry_date > '{{ date.end }}'
      AND (effective_date <= '{{ date.start }}'
        OR (effective_date BETWEEN '{{ date.start }}' AND '{{ date.end }}'))
  ) curr
  JOIN
  (
    SELECT
      ...
```

We can also use the DB SQL query editor to run queries against different versions of our Delta tables. For example, we can query a view of the aggregated claims records for a specific date and time (see example below). We can further use DB SQL to compare results from different versions to analyze only the changed records between those states.

```
SELECT
  *
FROM
  insurance_demo_lakehouse.aggregated_claims_weekly TIMESTAMP AS OF '2022-06-05T17:00:00';
```

DB SQL offers the option to use a serverless compute engine, eliminating the need to configure, manage or scale cloud infrastructure while maintaining the lowest possible cost. It also integrates with alternative SQL workbenches (e.g., DataGrip), allowing analysts to use their favorite tools to explore the data and generate insights.

BUSINESS INSIGHTS

Finally, we can use DB SQL queries to create rich visualizations on top of our query results. These visualizations can then be packaged and served to end users through interactive dashboards (see Figure 13).

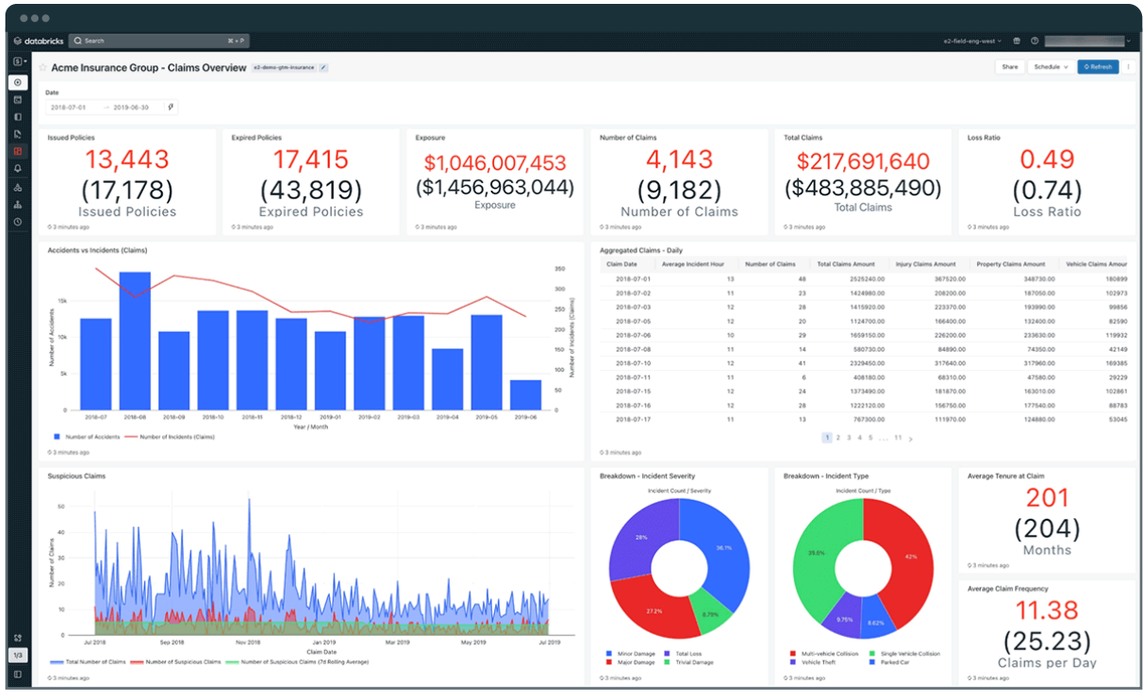


Figure 13: Example operational dashboard built on a set of resulting Delta Live Table (DLT) table entities

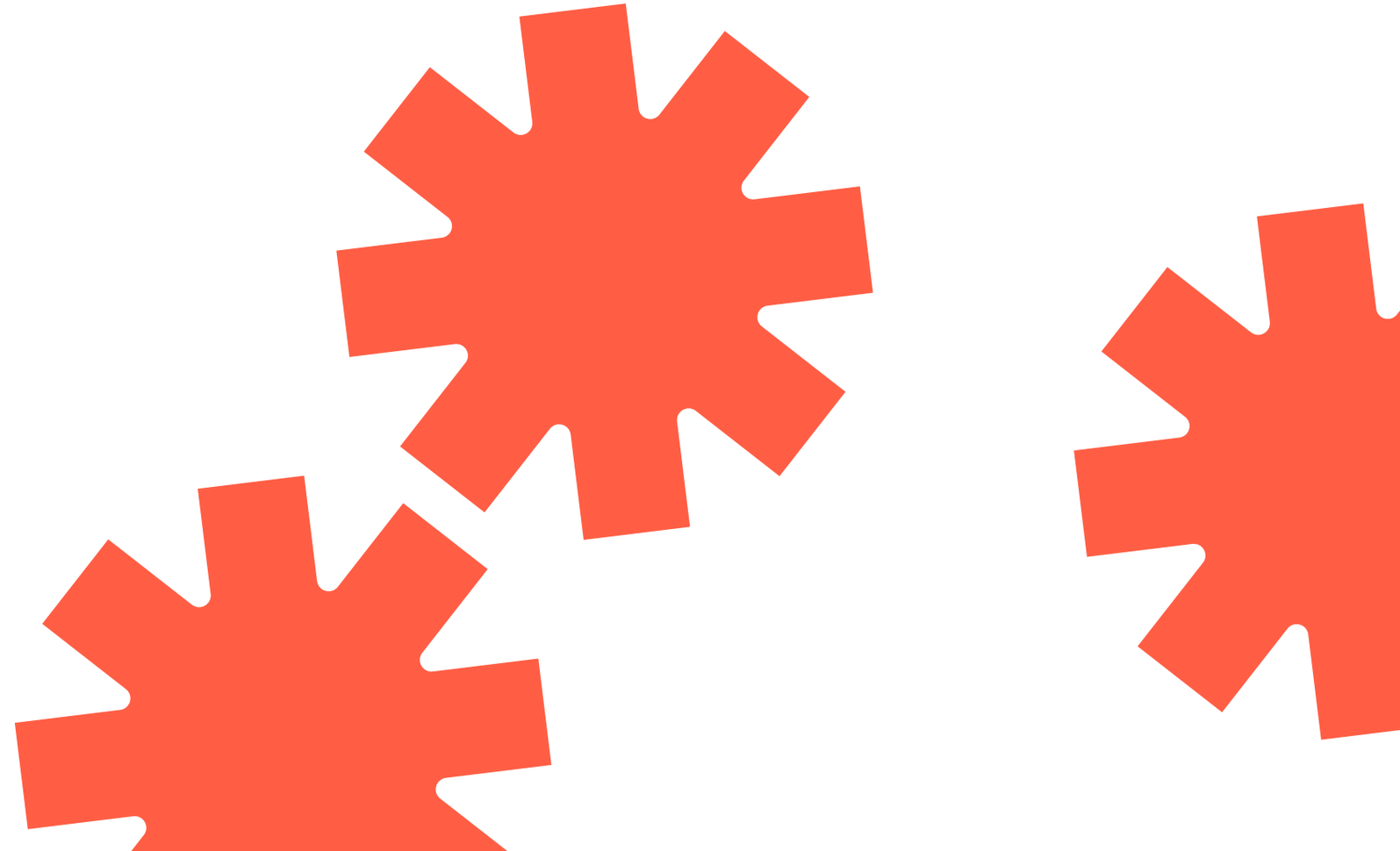
For our use case, we created a dashboard with a collection of key metrics, rolling calculations, high-level breakdowns, and aggregate views. The dashboard provides a complete summary of our claims process at a glance. We also added the option to specify specific date ranges. DB SQL supports a range of query parameters that can substitute values into a query at runtime. These query parameters can be defined at the dashboard level to ensure all related queries are updated accordingly.

DB SQL integrates with numerous third-party analytical and BI tools like Power BI, Tableau and Looker. Like we did for Fivetran, we can use Partner Connect to link our external platform with DB SQL. This allows analysts to build and serve dashboards in the platforms that the business prefers without sacrificing the performance of DB SQL and the Databricks Lakehouse.

CONCLUSION

As we move into this fast-paced, volatile modern world of finance, batch processing remains a vital part of the modern data stack, able to hold its own against the features and benefits of streaming and real-time services. We've seen how we can use the Databricks Data Intelligence Platform for Financial Services and its ecosystem of partners to architect a simple, scalable and extensible framework that supports complex batch-processing workloads with a practical example in insurance claims processing. With Delta Live Tables (DLT) and Databricks SQL (DB SQL), we can build a data platform with an architecture that scales infinitely, is easy to extend to address changing requirements and will withstand the test of time.

To learn more about the sample pipeline described, including the infrastructure setup and configuration used, please refer to this GitHub repository or watch this demo video.



How to Set Up Your First Federated Lakehouse

by Mike Dobing

Lakehouse Federation in Databricks is a groundbreaking new capability that allows you to query data across external data sources — including Snowflake, Synapse, many others and even Databricks itself — without having to move or copy the data. This is done by using Databricks Unity Catalog, which provides a unified metadata layer for all of your data.

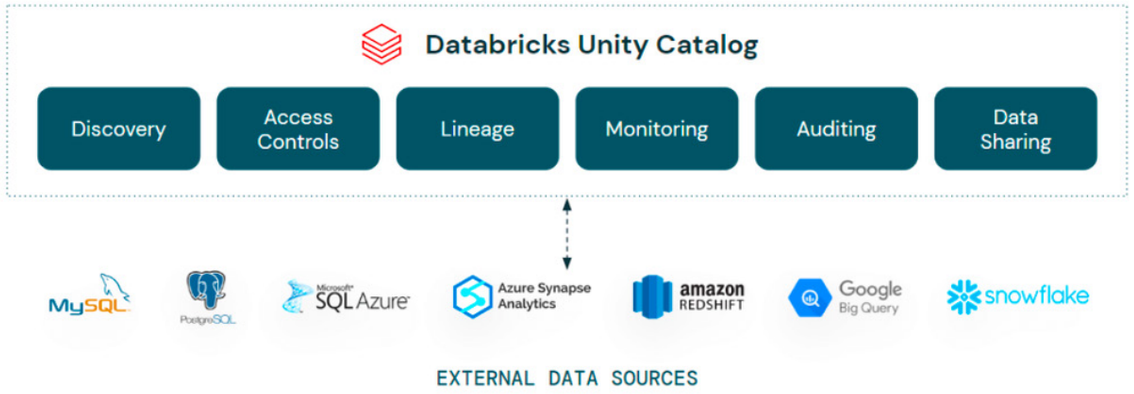
Lakehouse Federation is a game-changer for data teams, as it breaks down the silos that have traditionally kept data locked away in different systems. With Lakehouse Federation, you can finally access all of your data in one place, making it easier to get the insights you need to make better business decisions.

As always, though, not one solution is a silver bullet for your data integration and querying needs. See below for when Federation is a good fit, and for when you'd prefer to bring your data into your solution and process as part of your lakehouse platform pipelines.

A few of the benefits of using Lakehouse Federation in Databricks are:

- **Improved data access and discovery:** Lakehouse Federation makes it easy to find and access the data you need from your database estate. This is especially important for organizations with complex data landscapes.
- **Reduced data silos:** Lakehouse Federation can help to break down data silos by providing a unified view of all data across the organization.
- **Improved data governance:** Lakehouse Federation can help to improve data governance by providing a single place to manage permissions and access to data from within Databricks.
- **Reduced costs:** Lakehouse Federation can help to reduce costs by eliminating the need to move or copy data between different data sources.

If you are looking for a way to improve the way you access and manage your data across your analytics estate, then Lakehouse Federation in Databricks is a top choice.



REALITY CHECK

While Lakehouse Federation is a powerful tool, it is not a good fit for all use cases. There are some specific examples of use cases when Lakehouse Federation is not a good choice:

- **Real-time data processing:** Lakehouse Federation queries can be slower than queries on data that is stored locally in the lake. Therefore, Lakehouse Federation is not a good choice for applications that require real-time data processing.
- **Complex data transformations:** Where you need complex data transformations and processing, or need to ingest and transform vast amounts of data. For probably the large majority of use cases, you will need to apply some kind of ETL/ELT process against your data to make it fit for consumption by end users. In these scenarios, it is still best to apply a medallion style approach and bring the data in, process it, clean it, then model and serve it so it is performant and fit for consumption by end users.

Therefore, while Lakehouse Federation is a great option for certain use cases as highlighted above, it's not a silver bullet for all scenarios. Consider it an augmentation of your analytics capability that allows for additional use cases that need agility and direct source access for creating a holistic view of your data estate, all controlled through one governance layer.

SETTING UP YOUR FIRST FEDERATED LAKEHOUSE

With that in mind, let's get started on setting up your first federated lakehouse in Databricks using Lakehouse Federation.

For this example, we will be using a familiar sample database — AdventureWorks — running on an Azure SQL Database. We will be walking you through how to set up your connection to Azure SQL and how to add it as a foreign catalog inside Databricks.

PREREQUISITES

To set up Lakehouse Federation in Databricks, you will need the following prerequisites:

- A Unity Catalog-enabled Databricks workspace with Databricks Runtime 13.1 or above and shared or sin...
- A Databricks Unity Catalog metastore
- Network connectivity from your Databricks Runtime cluster or SQL warehouse to the target database systems, including any firewall connectivity requirements, such as here
- The necessary permissions to create connections and foreign catalogs in Databricks Unity Catalog
- The SQL Warehouse must be Pro or Serverless
- For this demo — an example database such as AdventureWorks to use as our data source, along with the necessary credentials. Please see this example.

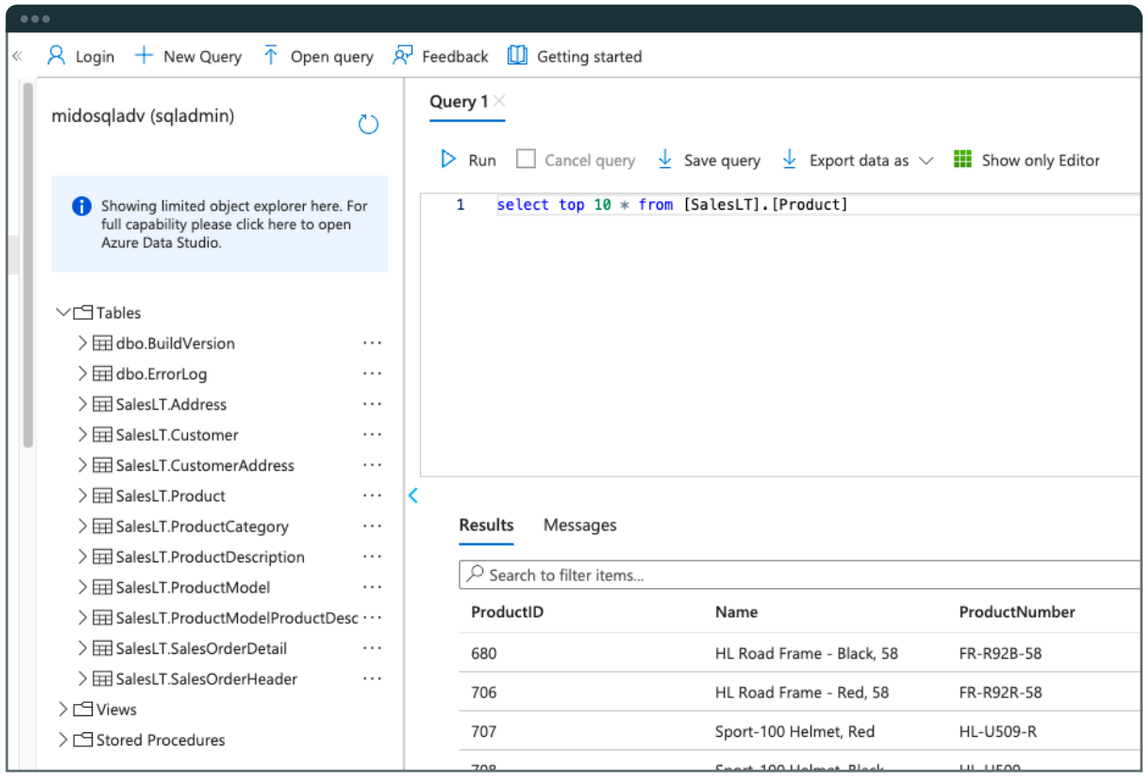
SETUP

Setting up federation is essentially a three-step process, as follows:

- Set up a connection
- Set up a foreign catalog
- Query your data sources

SETTING UP A CONNECTION

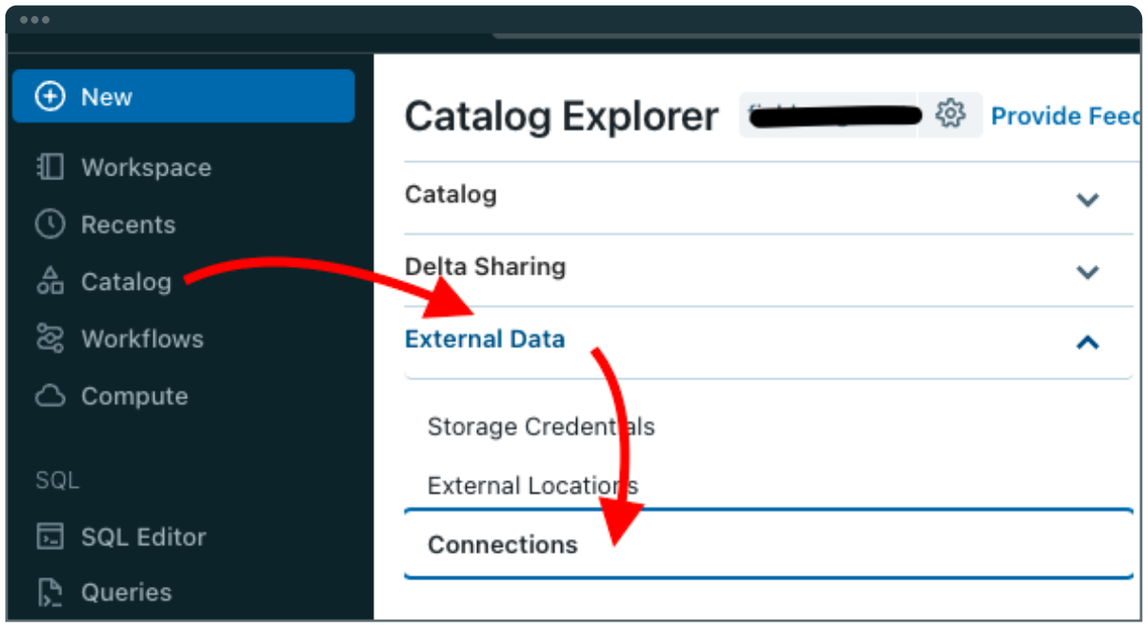
We are going to use Azure SQL Database as the test data source with the sample database AdventureWorksLT database already installed and ready to query:



We want to add this database as a foreign catalog in Databricks to be able to query it alongside other data sources. To connect to the database, we need a username, password and hostname, obtained from my Azure SQL Instance.

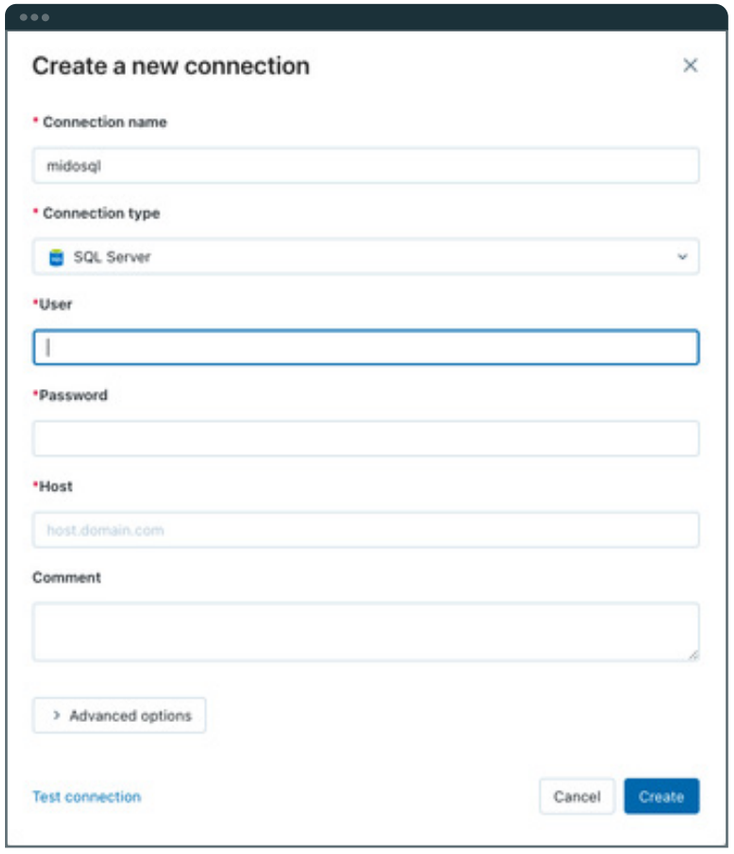
With these details ready, we can now go into Databricks and add the connection there as our first step.

First, expand the Catalog view, go to Connections and click "Create Connection":



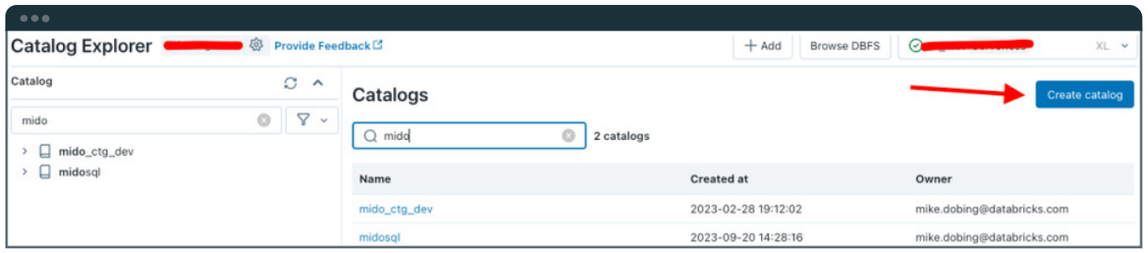
Example query on the source database

To add your new connection, give it a name, choose your connection type and then add the relevant login details for that data source:

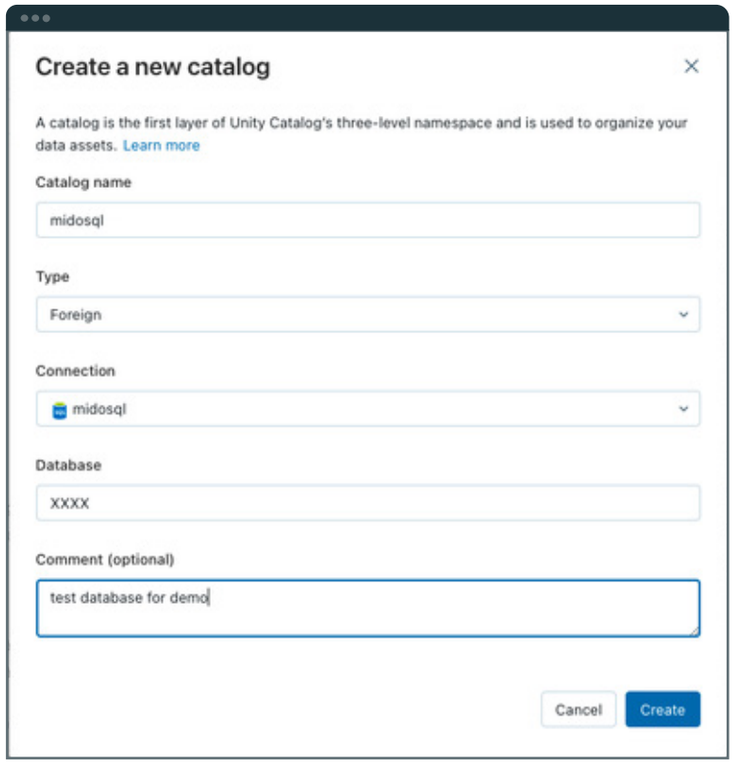


CREATE A FOREIGN CATALOG

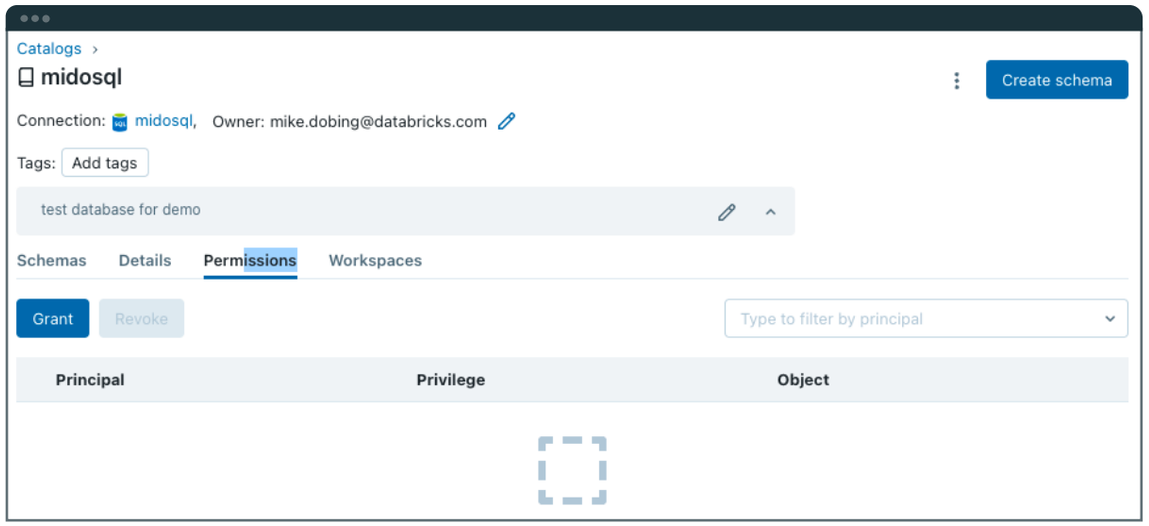
Test your connection and verify all is well. From there, go back to the Catalog view and go to Create Catalog:



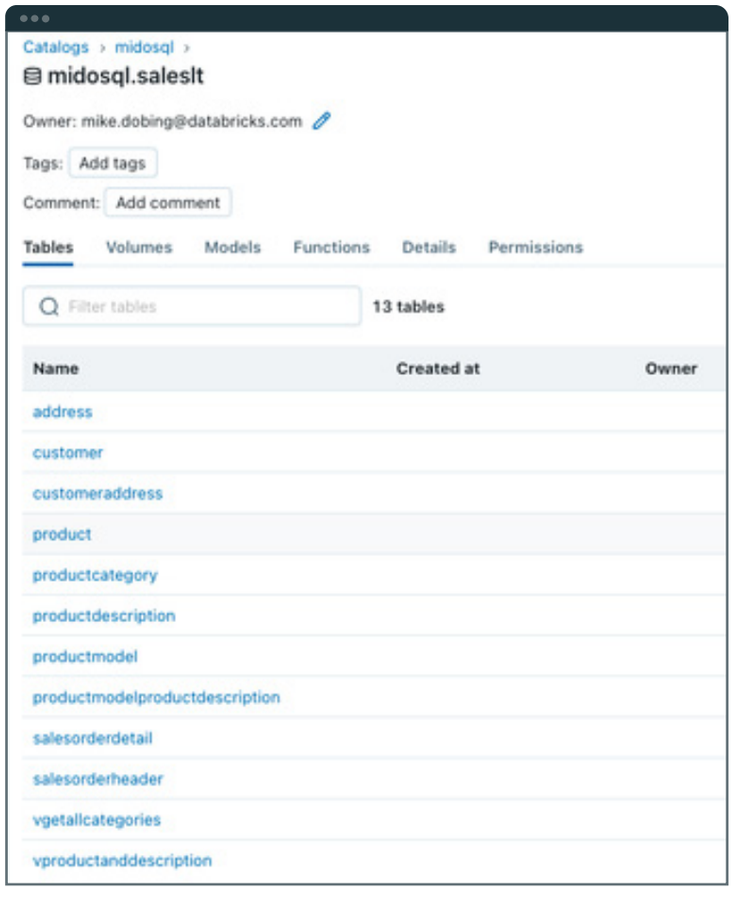
From there, populate the relevant details (choosing Type as "Foreign"), including choosing the connection you created in the first step, and specifying the database you want to add as an external catalog:



Once added, you can have the option of adding the relevant user permissions to the objects here, all governed by Unity Catalog (skipped this in this article as there are no other users using this database):

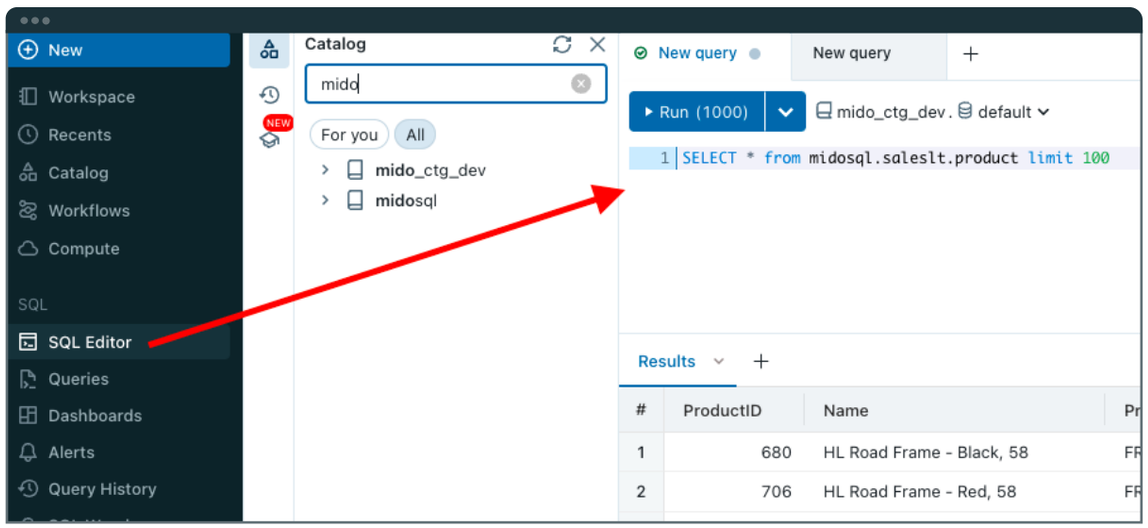


Our external catalog is now available for querying as you would any other catalog inside Databricks, bringing our broader data estate into our lakehouse:

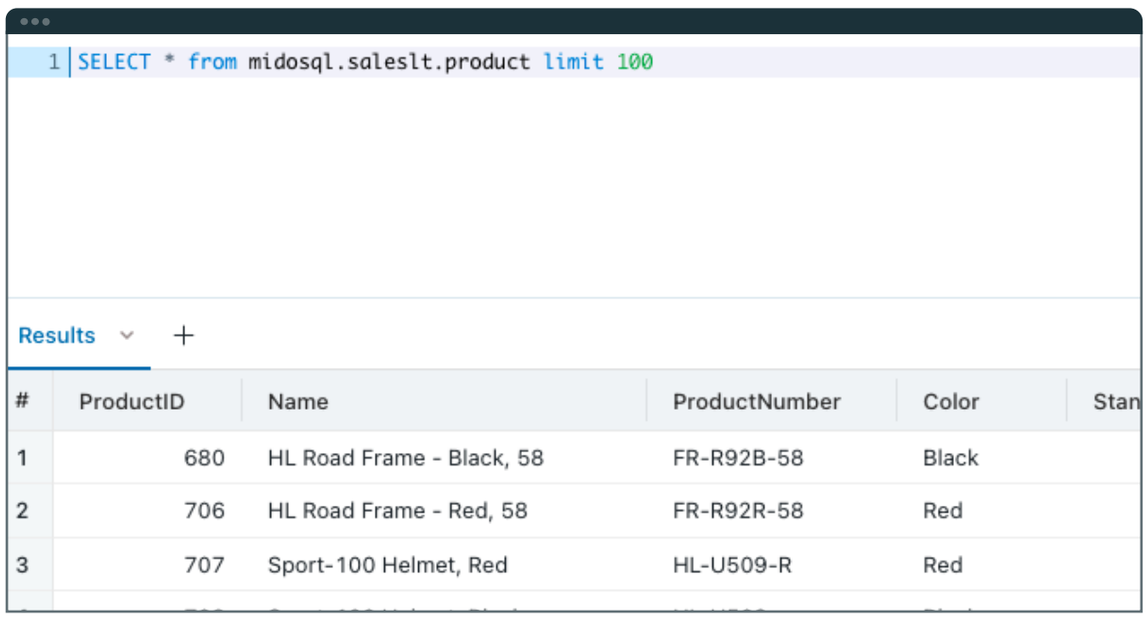


QUERYING THE FEDERATED DATA

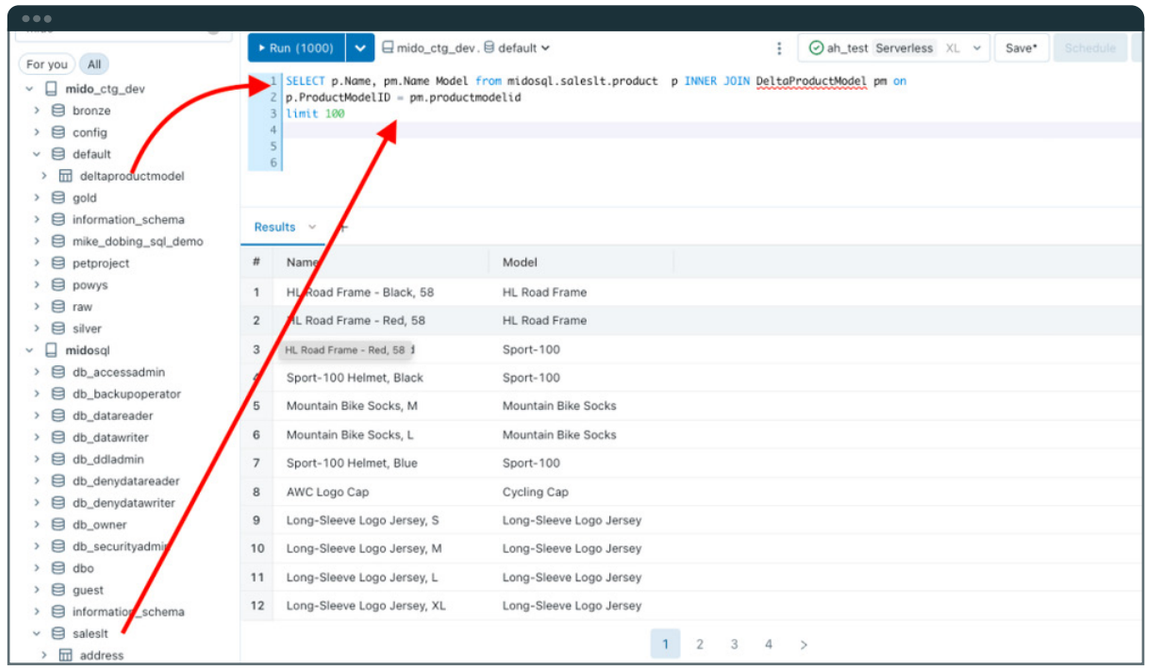
We can now access our federated Azure SQL Database as normal, straight from our Databricks SQL Warehouse:



And query it as we would any other object:



Or even join it to a local Delta table inside our Unity Catalog:



CONCLUSION

What we've shown here is just scratching the surface of what Lakehouse Federation can do with a simple connection and query. By leveraging this offering, combined with the governance and capabilities of Unity Catalog, you can extend the range of your lakehouse estate, ensuring consistent permissions and controls across all of your data sources and thus enabling a plethora of new use cases and opportunities.

Orchestrating Data Analytics With Databricks Workflows

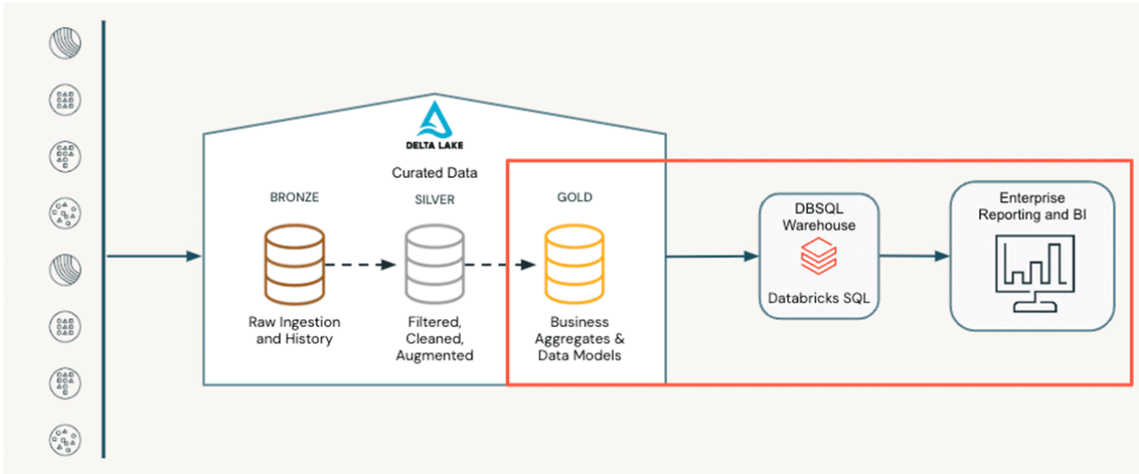
by Matthew Kuehn

For data-driven enterprises, data analysts play a crucial role in extracting insights from data and presenting it in a meaningful way. However, many analysts might not have the familiarity with data orchestration required to automate their workloads for production. While a handful of ad hoc queries can quickly turn around the right data for a last-minute report, data teams must ensure that various processing, transformation and validation tasks are executed reliably and in the right sequence. Without the proper orchestration in place, data teams lose the ability to monitor pipelines, troubleshoot failures and manage dependencies. As a result, sets of ad hoc queries that initially brought quick-hitting value to the business end up becoming long-term headaches for the analysts who built them.

Pipeline automation and orchestration becomes particularly crucial as the scale of data grows and the complexity of pipelines increases. Traditionally, these responsibilities have fallen on data engineers, but as data analysts begin to develop more assets in the lakehouse, orchestration and automation becomes a key piece to the puzzle.

For data analysts, the process of querying and visualizing data should be seamless, and that's where the power of modern tools like Databricks Workflows comes into play. In this chapter, we'll explore how data analysts can leverage Databricks Workflows to automate their data processes, enabling them to focus on what they do best — deriving value from data.

THE DATA ANALYST'S WORLD

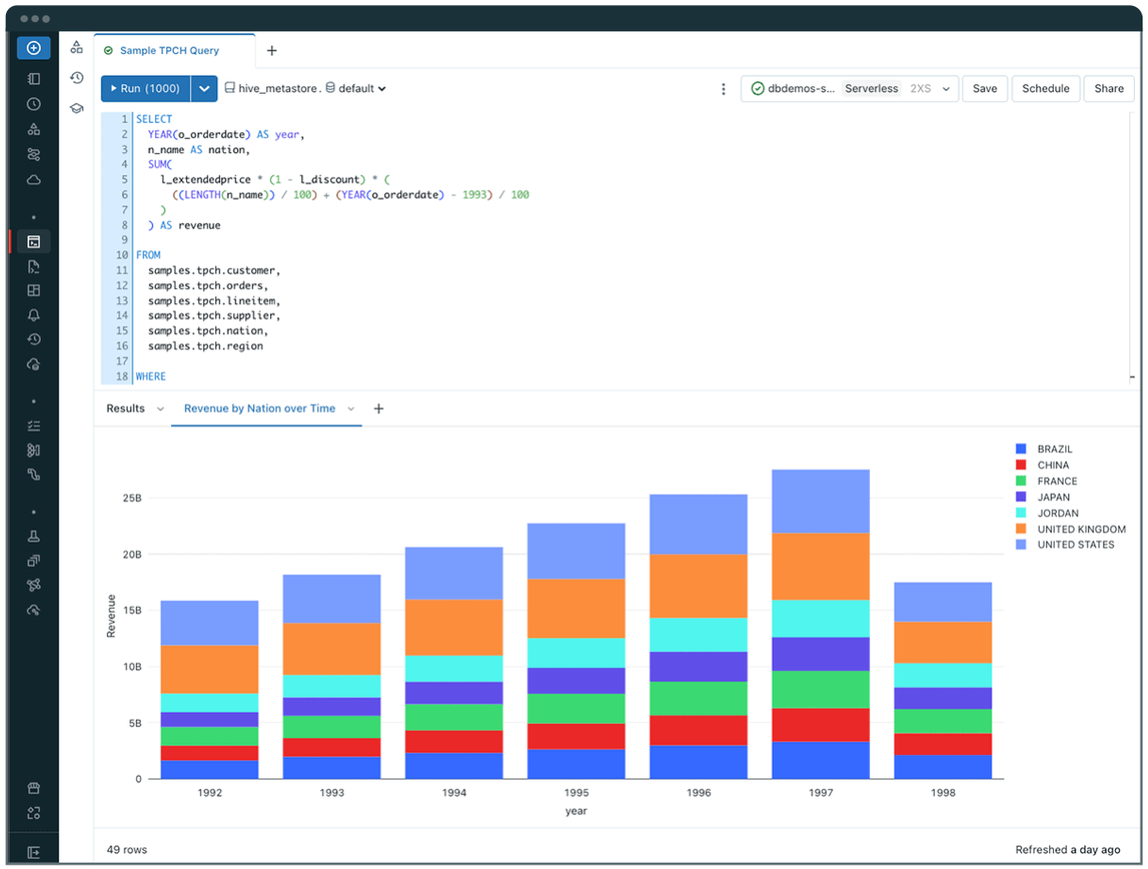


Data analysts play a vital role in the final stages of the data lifecycle. Positioned at the "last mile," they rely on refined data from upstream pipelines. This could be a table prepared by a data engineer or the output predictions of machine learning models built by data scientists. This refined data, often referred to as the Silver layer in a medallion architecture, serves as the foundation for their work. Data analysts are responsible for aggregating, enriching and shaping this data to answer specific questions for their business, such as:

- "How many orders were placed for each SKU last week?"
- "What was monthly revenue for each store last fiscal year?"
- "Who are our 10 most active users?"

These aggregations and enrichments build out the Gold layer of the medallion architecture. This Gold layer enables easy consumption and reporting for downstream users, typically in a visualization layer. This can take the form of dashboards within Databricks or be seamlessly generated using external tools like Tableau or Power BI via Partner Connect. Regardless of the tech stack, data analysts transform raw data into valuable insights, enabling informed decision-making through structured analysis and visualization techniques.

THE DATA ANALYST'S TOOLKIT ON DATABRICKS



In Databricks, data analysts have a robust toolkit at their fingertips to transform data effectively on the lakehouse. Centered around the Databricks SQL Editor, analysts have a familiar environment for composing ANSI SQL queries, accessing data and exploring table schemas. These queries serve as building blocks for various SQL assets, including visualizations that offer in-line data insights. Dashboards consolidate multiple visualizations, creating a user-friendly interface for comprehensive reporting and data exploration for end users.

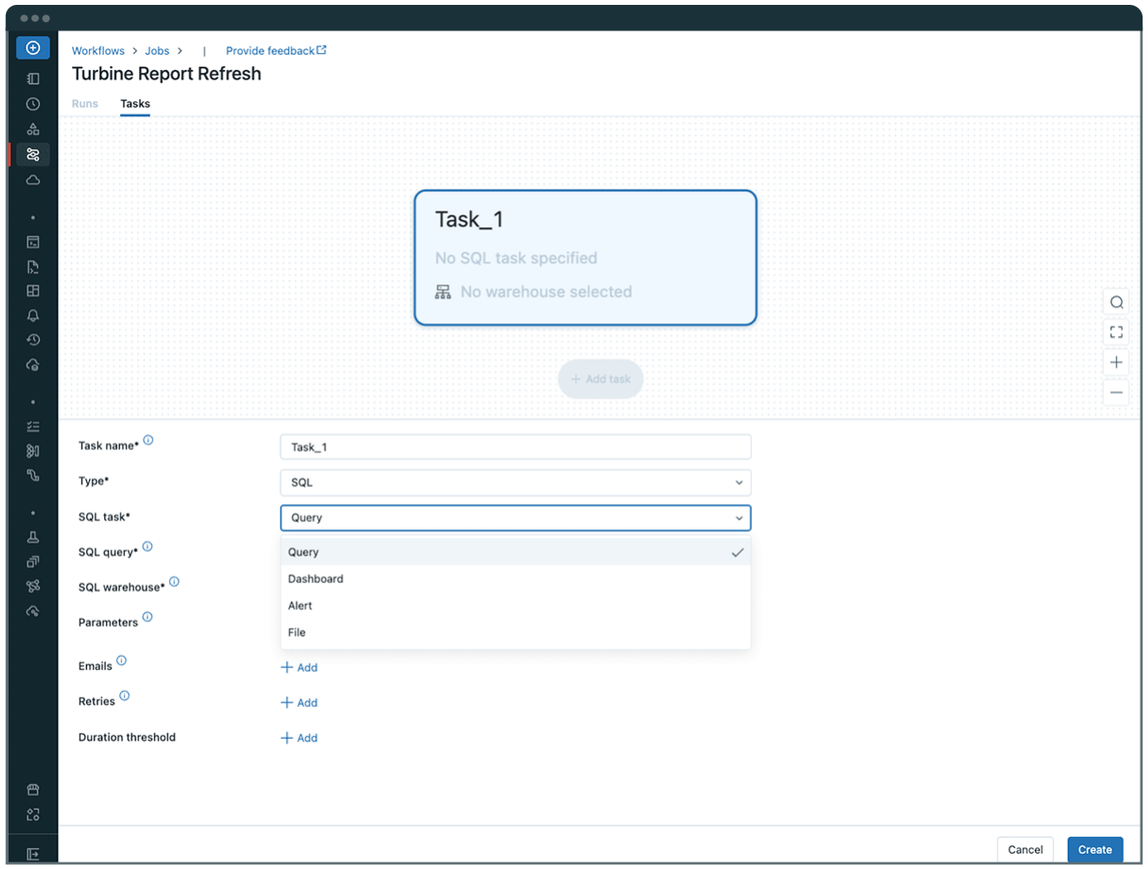
Additionally, alerts keep analysts informed about critical dataset changes in real time. Serverless SQL Warehouses are underpinning all these features, which can scale to handle diverse data volumes and query demands. By default, this compute uses Photon, the high-performance Databricks-native vectorized query engine, and is optimized for high-concurrency SQL workloads. Finally, Unity Catalog allows users to easily govern structured and unstructured data, machine learning models, notebooks, dashboards and files in the lakehouse. This cohesive toolkit empowers data analysts to transform raw data into enriched insights seamlessly within the Databricks environment.

ORCHESTRATING THE DATA ANALYST'S TOOLKIT WITH WORKFLOWS

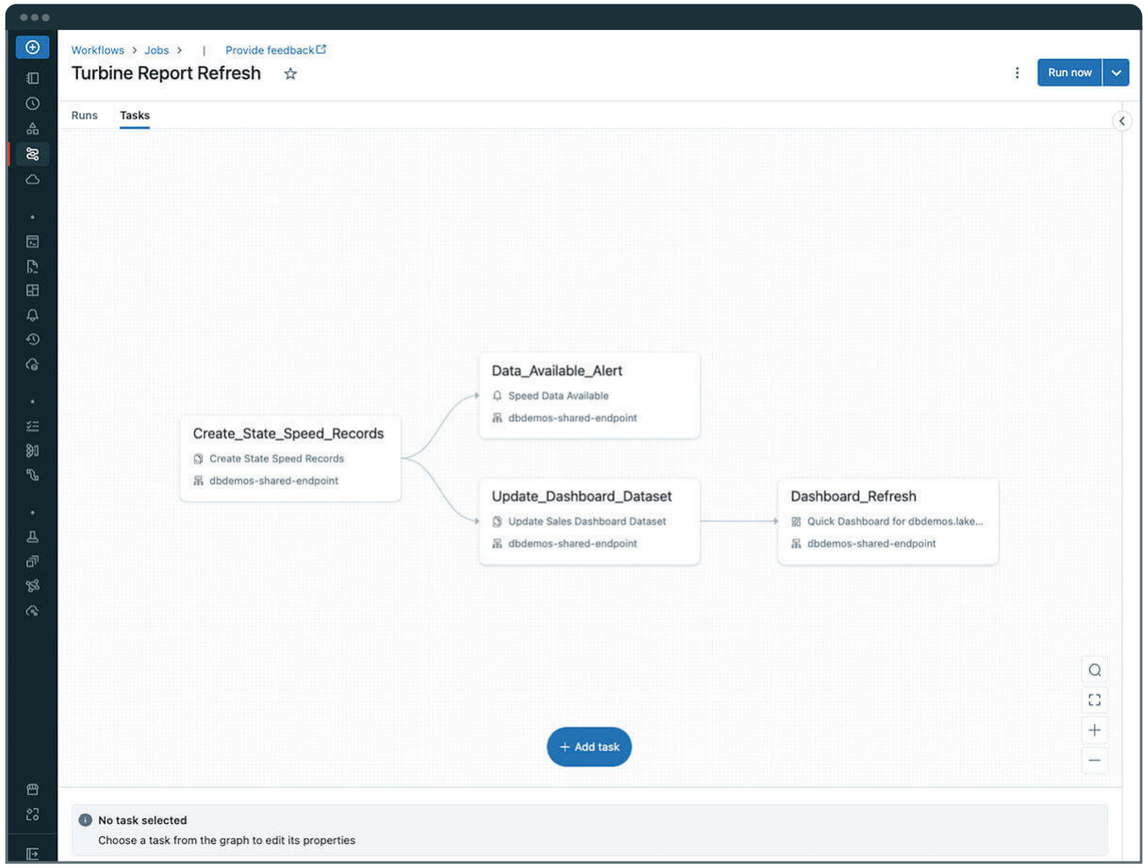
For those new to Databricks, Workflows orchestrates data processing, machine learning and analytics pipelines in the Data Intelligence Platform. Workflows is a fully managed orchestration service integrated with the Databricks Platform, with high reliability and advanced observability capabilities. This allows all users, regardless of persona or background, to easily orchestrate their workloads in production environments.

Authoring Your SQL Tasks

Building your first workflow as a data analyst is extremely simple. Workflows now seamlessly integrates the core tools used by data analysts — queries, alerts and dashboards — within its framework, enhancing its capabilities through the SQL task type. This allows data analysts to build and work with the tools they are already familiar with and then easily bring them into a Workflow as a Task via the UI.



As data analysts begin to chain more SQL tasks together, they will begin to easily define dependencies between and gain the ability to schedule and automate SQL-based tasks within Databricks Workflows. In the below example workflow, we see this in action:



Imagine that we have received upstream data from our data engineering team that allows us to begin our dashboard refresh process. We can define SQL-centric tasks like the ones below to automate our pipeline:

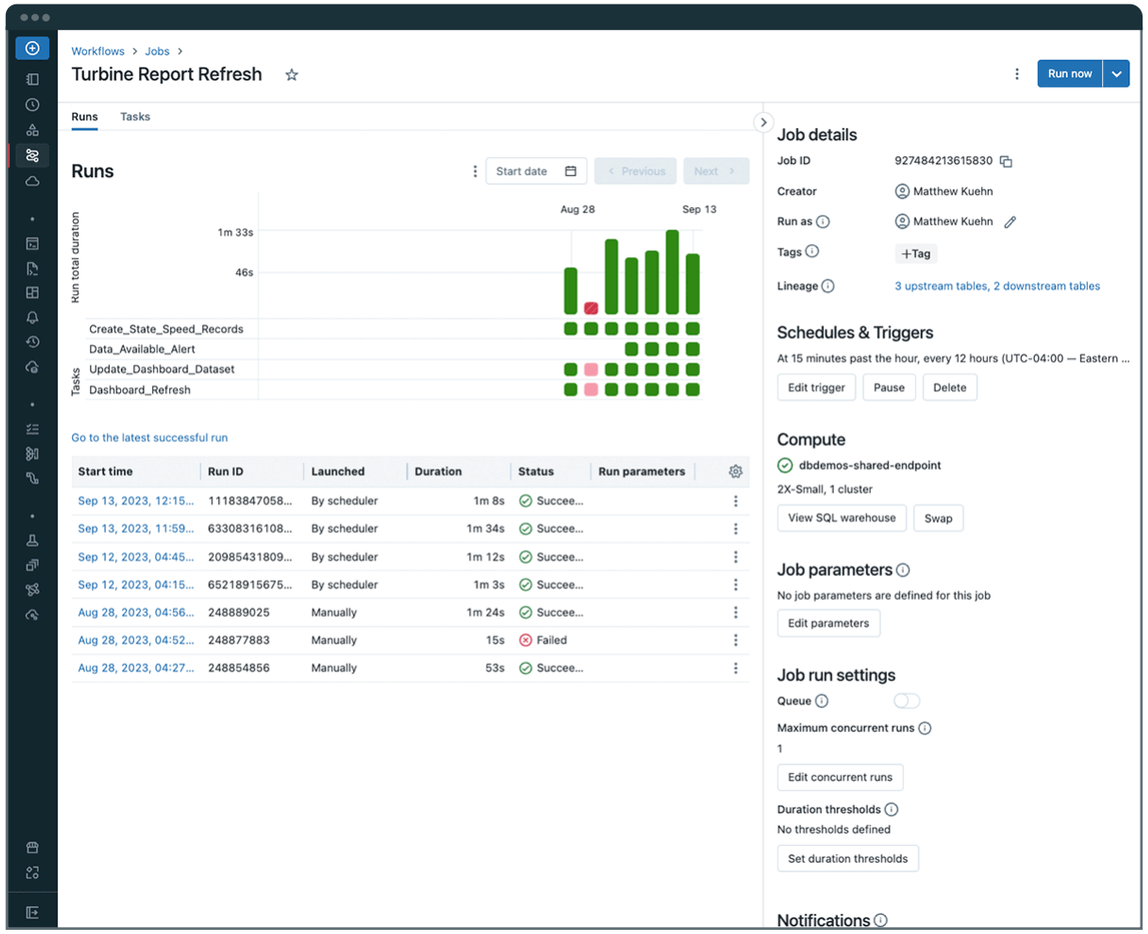
- **Create_State_Speed_Records:** First, we define our refreshed data in our Gold layer with the Query task. This inserts data into a Gold table and then optimizes it for better performance.
- **Data_Available_Alert:** Once this data is inserted, imagine we want to notify other data analysts who consume this table that new records have been added. We can do this by creating an alert which will trigger when we have new records added. This will send an alert to our stakeholder group. You can imagine using an alert in a similar fashion for data quality checks to warn users of stale data, null records or other similar situations. *For more information on creating your first alert, check out this link.*
- **Update_Dashboard_Dataset:** It's worth mentioning that tasks can be defined in parallel if needed. In our example, while our alert is triggering we can also begin refreshing our tailored dataset view that feeds our dashboard in a parallel query.
- **Dashboard_Refresh:** Finally, we create a dashboard task type. Once our dataset is ready to go, this will update all previously defined visualizations with the newest data and notify all subscribers upon successful completion. Users can even pass specific parameters to the dashboard while defining the task, which can help generate a default view of the dashboard depending on the end user's needs.

It is worth noting that this example workflow utilizes queries directly written in the Databricks SQL Editor. A similar pattern can be achieved with SQL code coming from a repository using the File task type. With this task type, users can execute .sql files stored in a Git repository as part of an automated workflow. Each time the pipeline is executed, the latest version from a specific branch will be retrieved and executed.

Although this example is basic, you can begin to see the possibilities of how a data analyst can define dependencies across SQL task types to build a comprehensive analytics pipeline.

MONITORING YOUR PRODUCTION PIPELINES

While authoring is comprehensive within Databricks Workflows, it is only one part of the picture. Equally important is the ability to easily monitor and debug your pipelines once they are built and in production.



Databricks Workflows allows users to monitor individual job runs, offering insights into task outcomes and overall execution times. This visibility helps analysts understand query performance, identify bottlenecks and address issues efficiently. By promptly recognizing tasks that require attention, analysts can ensure seamless data processing and quicker issue resolution.

When it comes to executing a pipeline at the right time, Databricks Workflows allows users to schedule jobs for execution at specific intervals or trigger them when certain files arrive. In the above image, we were first manually triggering this pipeline to test and debug our tasks. Once we got this to a steady state, we began triggering this every 12 hours to accommodate for data refresh needs across time zones. This flexibility accommodates varying data scenarios, ensuring timely pipeline execution. Whether it's routine processing or responding to new data batches, analysts can tailor job execution to match operational requirements.

Late-arriving data can bring a flurry of questions to a data analyst from end users. Workflows enables analysts and consumers alike to stay informed on data freshness by setting up notifications for job outcomes such as successful execution, failure or even a long-running job. These notifications ensure timely awareness of changes in data processing. By proactively evaluating a pipeline's status, analysts can take proactive measures based on real-time information.

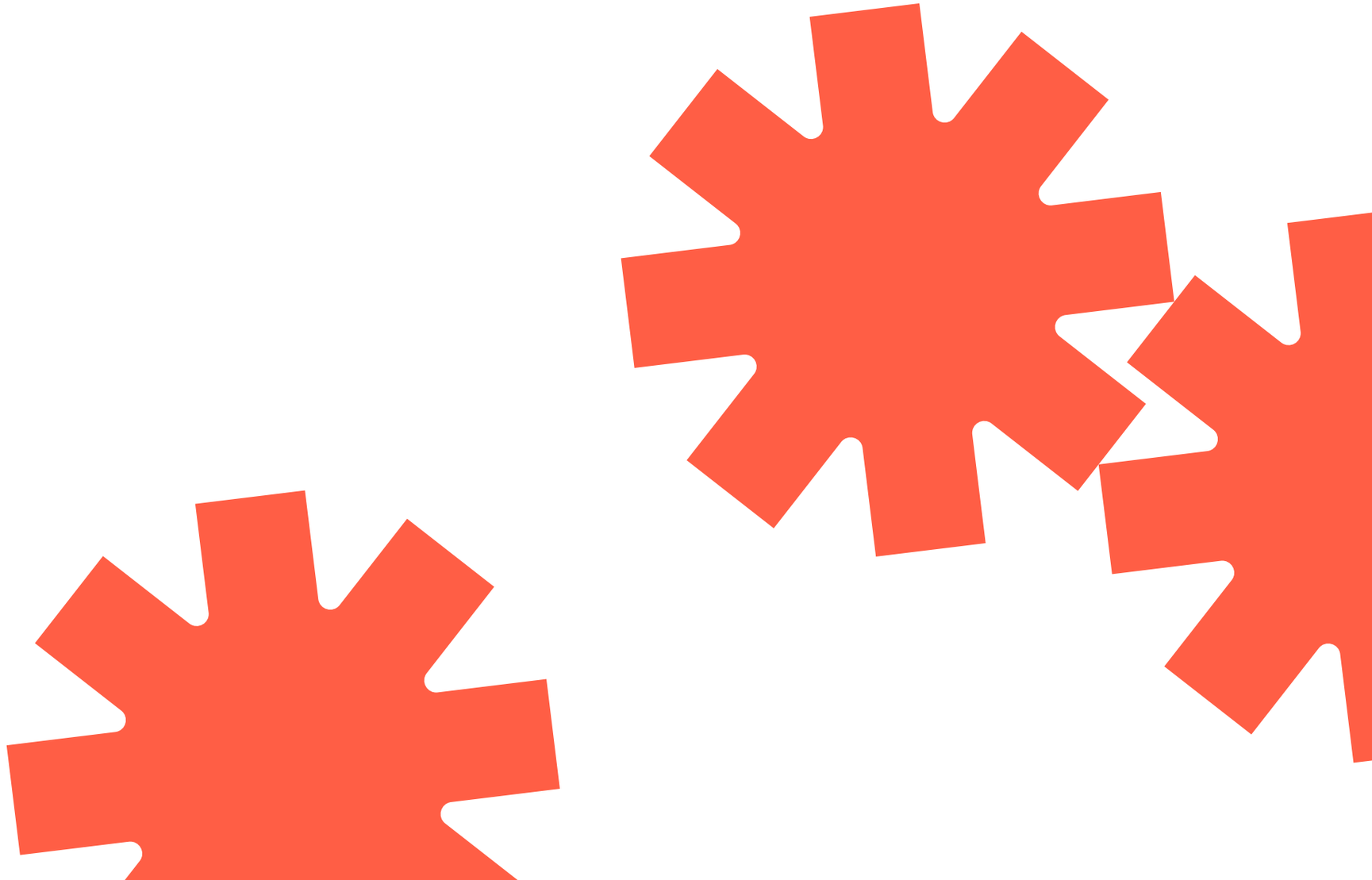
As with all pipelines, failures will inevitably happen. Workflows helps manage this by allowing analysts to configure job tasks for automatic retries. By automating retries, analysts can focus on generating insights rather than troubleshooting intermittent technical issues.

CONCLUSION

In the evolving landscape of data analysis tools, Databricks Workflows bridges the gap between data analysts and the complexities of data orchestration. By automating tasks, ensuring data quality and providing a user-friendly interface, Databricks Workflows empowers analysts to focus on what they excel at — extracting meaningful insights from data. As the concept of the lakehouse continues to unfold, Workflows stands as a pivotal component, promising a unified and efficient data ecosystem for all personas.

GET STARTED

- Learn more about Databricks Workflows
- Take a product tour of Databricks Workflows
- Create your first workflow with this quickstart guide



Schema Management and Drift Scenarios via Databricks Auto Loader

by Garrett Peternel

Data lakes notoriously have had challenges with managing incremental data processing at scale without integrating open table storage format frameworks (e.g., Delta Lake, Apache Iceberg, Apache Hudi). In addition, schema management is difficult with schema-less data and schema-on-read methods. With the power of the Databricks Platform, Delta Lake and Apache Spark provide the essential technologies integrated with Databricks Auto Loader (AL) to consistently and reliably stream and process raw data formats incrementally while maintaining stellar performance and data governance.

AUTO LOADER FEATURES

AL is a boost over Spark Structured Streaming, supporting several additional benefits and solutions including:

- Databricks Runtime only Structured Streaming cloudFiles source
- Schema drift, dynamic inference and evolution support
- Ingests data via JSON, CSV, PARQUET, AVRO, ORC, TEXT and BINARYFILE input file formats
- Integration with cloud file notification services (e.g., Amazon SQS/SNS)
- Optimizes directory list mode scanning performance to discover new files in cloud storage (e.g., AWS, Azure, GCP, DBFS)

For further information please visit the official Databricks Auto Loader documentation.

SCHEMA CHANGE SCENARIOS

In this chapter I will showcase a few examples of how AL handles schema management and drift scenarios using a public IoT sample dataset with schema modifications to showcase solutions. Schema 1 will contain an IoT sample dataset schema with all expected columns and expected data types. Schema 2 will contain unexpected changes to the IoT sample dataset schema with new columns and changed data types. The following variables and paths will be used for this demonstration along with Databricks Widgets to set your username folder.

```scala
%scala
dbutils.widgets.text("dbfs_user_dir", "your_user_name") // widget for account email

val userName = dbutils.widgets.get("dbfs_user_dir")
val rawBasePath = s"dbfs:/user/$userName/raw/"
val repoBasePath = s"dbfs:/user/$userName/repo/"

val jsonSchema1Path = rawBasePath + "iot-schema-1.json"
val jsonSchema2Path = rawBasePath + "iot-schema-2.json"
val repoSchemaPath = repoBasePath + "iot-ddl.json"

dbutils.fs.rm(repoSchemaPath, true) // remove schema repo for demos
```

SCHEMA 1

```scala
%scala
spark.read.json(jsonSchema1Path).printSchema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: long (nullable = true)
|-- c02_level: long (nullable = true)
|-- cca2: string (nullable = true)
|-- cca3: string (nullable = true)
|-- cn: string (nullable = true)
|-- coordinates: struct (nullable = true)
| |-- latitude: double (nullable = true)
| |-- longitude: double (nullable = true)
|-- date: string (nullable = true)
|-- device_id: long (nullable = true)
|-- device_serial_number: string (nullable = true)
|-- device_type: string (nullable = true)
|-- epoch_time_miliseconds: long (nullable = true)
|-- humidity: long (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
```

```
%scala
display(spark.read.json(jsonSchema1Path).limit(10))
```

Table + New result table: OFF

	alarm_status	battery_level	c02_level	cca2	cca3	cn	coordinates	date	device_id	device_serial_num
1	yellow	3	1082	US	USA	United States	{"latitude": 38, "longitude": -97}	2016-03-20	62	62fH8oKr8aiT
2	green	0	920	US	USA	null	{"latitude": 47, "longitude": 8}	2016-03-20	241	241un29KmR
3	yellow	7	1004	US	USA	United States	{"latitude": 42.36, "longitude": -71.05}	2016-03-20	260	260F7DTEbnz
4	yellow	9	1081	DE	DEU	Germany	{"latitude": 51.45, "longitude": 7.02}	2016-03-20	298	298hJceoQv0
5	yellow	8	1311	US	USA	United States	{"latitude": 38.88, "longitude": -92.4}	2016-03-20	301	301wHcyNzw
6	red	1	1484	IN	IND	India	{"latitude": 20, "longitude": 77}	2016-03-20	334	334DUAg4mbXi
7	yellow	9	1266	US	USA	null	{"latitude": 47, "longitude": 8}	2016-03-20	349	349gRZSGtcGR
8	yellow	1	1085	GB	GBR	United Kingdom	{"latitude": 51.51, "longitude": -0.09}	2016-03-20	383	383yNMmfRq0zG
9	red	3	1508	GB	GBR	United Kingdom	{"latitude": 53.5, "longitude": -2.19}	2016-03-20	391	391Qn3lLA
10	yellow	5	1191	KR	KOR	Republic of Korea	{"latitude": 37.29, "longitude": 127.01}	2016-03-20	455	455L4J9FUG

SCHEMA 2

```
%scala
// NEW => device_serial_number_device_type, location
spark.read.json(jsonSchema2Path).printSchema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: long (nullable = true)
|-- c02_level: long (nullable = true)
|-- date: string (nullable = true)
|-- device_id: long (nullable = true)
|-- device_serial_number_device_type: string (nullable = true)
|-- epoch_time_miliseconds: long (nullable = true)
|-- humidity: double (nullable = true)
|-- ip: string (nullable = true)
|-- latitude: double (nullable = true)
|-- location: struct (nullable = true)
| |-- cca2: string (nullable = true)
| |-- cca3: string (nullable = true)
| |-- cn: string (nullable = true)
|-- longitude: double (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
```

```
%scala
display(spark.read.json(jsonSchema1Path).limit(10))
```

Table + New result table: OFF

	device_serial_number_device_type	epoch_time_miliseconds	humidity	ip	latitude	location	longitude	scale
1	{"2n2Pea":"sensor-pad"}	1458444054119	70	213.161.254.1	62.47	{"cca2": "NO", "cca3": "NOR", "cn": "Norway"}	6.15	Fahre
2	{"12Y2klm0o":"sensor-pad"}	1458444054126	92	68.28.91.22	38	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-97	Fahre
3	{"230lupA":"meter-gauge"}	1458444054133	47	59.90.65.1	12.98	{"cca2": "IN", "cca3": "IND", "cn": "India"}	77.58	Fahre
4	{"36VQv8fnEg":"sensor-pad"}	1458444054141	47	213.7.14.1	35	{"cca2": "CY", "cca3": "CYP", "cn": "Cyprus"}	33	Fahre
5	{"448DeWGL":"sensor-pad"}	1458444054149	63	62.128.16.74	49.46	{"cca2": "DE", "cca3": "DEU", "cn": "Germany"}	11.1	Fahre
6	{"49YesGXwt":"meter-gauge"}	1458444054152	70	170.37.224.1	42.28	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-71.44	Fahre
7	{"64djcln":"sensor-pad"}	1458444054162	55	38.99.198.186	38	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-97	Fahre
8	{"77IKW3YAB55":"meter-gauge"}	1458444054169	82	218.248.255.30	12.98	{"cca2": "IN", "cca3": "IND", "cn": "India"}	77.58	Fahre
9	{"80TY4dWSMH":"sensor-pad"}	1458444054171	57	159.128.0.181	50.01	{"cca2": "CA", "cca3": "CAN", "cn": "Canada"}	-97.22	Fahre
10	{"94HL9jChD":"sensor-pad"}	1458444054179	66	24.32.26.1	39.33	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-120.25	Fahre

EXAMPLE 1: SCHEMA TRACKING/MANAGEMENT

AL tracks schema versions, metadata and changes to input data over time via specifying a location directory path. These features are incredibly useful for tracking history of data lineage, and are tightly integrated with the Delta Lake transactional log DESCRIBE HISTORY and Time Travel.

```
%scala
val rawAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.schemaLocation", repoSchemaPath) // schema history tracking
.load(jsonSchema1Path)
)
```

```
%scala
rawAlDf.printSchema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: string (nullable = true)
|-- c02_level: string (nullable = true)
|-- cca2: string (nullable = true)
|-- cca3: string (nullable = true)
|-- cn: string (nullable = true)
|-- coordinates: string (nullable = true)
|-- date: string (nullable = true)
|-- device_id: string (nullable = true)
|-- device_serial_number: string (nullable = true)
|-- device_type: string (nullable = true)
|-- epoch_time_miliseconds: string (nullable = true)
|-- humidity: string (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: string (nullable = true)
|-- timestamp: string (nullable = true)
|-- _rescued_data: string (nullable = true)
```

```
%scala
display(rawAlDf.limit(10))
```

By default (for JSON, CSV and XML file format) AL infers all column data types as strings, including nested fields.

Table + New result table: OFF

	alarm_status	battery_level	c02_level	cca2	cca3	cn	coordinates	date	device_id	device_serial_
1	yellow	3	1082	US	USA	United States	{"latitude":38.0,"longitude":-97.0}	2016-03-20	62	62fH8oKr8aiT
2	green	0	920	US	USA	null	{"latitude":47.0,"longitude":8.0}	2016-03-20	241	241un29KmR
3	yellow	7	1004	US	USA	United States	{"latitude":42.36,"longitude":-71.05}	2016-03-20	260	260F7DTEbnz
4	yellow	9	1081	DE	DEU	Germany	{"latitude":51.45,"longitude":7.02}	2016-03-20	298	298hJceoQv0
5	yellow	8	1311	US	USA	United States	{"latitude":38.88,"longitude":-92.4}	2016-03-20	301	301wHcyNzw
6	red	1	1484	IN	IND	India	{"latitude":20.0,"longitude":77.0}	2016-03-20	334	334DUAg4mb
7	yellow	9	1266	US	USA	null	{"latitude":47.0,"longitude":8.0}	2016-03-20	349	349gRZSGtcG
8	yellow	1	1085	GB	GBR	United Kingdom	{"latitude":51.51,"longitude":-0.09}	2016-03-20	383	383yNMmfRq0
9	red	3	1508	GB	GBR	United Kingdom	{"latitude":53.5,"longitude":-2.19}	2016-03-20	391	391Qn3lLA
10	yellow	5	1191	KR	KOR	Republic of Korea	{"latitude":37.29,"longitude":127.01}	2016-03-20	455	455L4J9FUG

Here is the directory structure where AL stores schema versions. These files can be read via Spark DataFrame API.

Schema Repository

```
%scala
display(dbutils.fs.ls(repoSchemaPath + "/_schemas"))
```

Table + New result table: OFF

	path	name	size	modificationTime
1	dbfs:/user/garrett.peternel@databricks.com/repo/iot-ddl.json/_schemas/0	0	1628	1709306037000

Schema Metadata

```
%scala
display(spark.read.json(repoSchemaPath + "/_schemas"))
```

Table + New result table: OFF

	_corrupt_record	dataSchemaJson	partitionSchemaJson
1	v1	null	null
2	null	{"type":"struct","fields":[{"name":"alarm_status","type":"string","nullable":true,"metadata":{}}, {"name":"battery_level","type":"long","nullable":true,"metadata":{}}, {"name":"c02_level","type":"long","nullable":true,"metadata":{}}, {"name":"cca2","type":"string","nullable":true,"metadata":{}},{"name":"cca3","type":"string","nullable":true,"metadata":{}},{"name":"cn","type":"string","nullable":true,"metadata":{}},{"name":"coordinates","type":{"type":"struct","fields":[{"name":"latitude","type":"do...	{"type":"struct","fields":[]}

EXAMPLE 2: SCHEMA HINTS

AL provides hint logic using SQL DDL syntax to enforce and override dynamic schema inference on known single data types, as well as semi-structured complex data types.

```
%scala
val hintAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.schemaLocation", repoSchemaPath)
.option("cloudFiles.schemaHints", "coordinates STRUCT<latitude:DOUBLE,
longitude:DOUBLE>, humidity LONG, temp DOUBLE") // schema ddl hints
.load(jsonSchema1Path)
)
```

```
%scala
hintAlDf.printSchema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: string (nullable = true)
|-- c02_level: string (nullable = true)
|-- cca2: string (nullable = true)
|-- cca3: string (nullable = true)
|-- cn: string (nullable = true)
|-- coordinates: struct (nullable = true)
| |-- latitude: double (nullable = true)
| |-- longitude: double (nullable = true)
|-- date: string (nullable = true)
|-- device_id: string (nullable = true)
|-- device_serial_number: string (nullable = true)
|-- device_type: string (nullable = true)
|-- epoch_time_miliseconds: string (nullable = true)
|-- humidity: long (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
|-- _rescued_data: string (nullable = true)
```

The schema hints specified in the AL options perform the data type mappings on the respective columns. Hints are useful for applying schema enforcement on portions of the schema where data types are known while in tandem with dynamic schema inference covered in Example 3.

```
%scala
display(hintAlDf.limit(10))
```

Table + New result table: OFF

	coordinates	date	device_id	device_serial_number	device_type	epoch_time_miliseconds	humidity	ip	scale
1	{"latitude": 38, "longitude": -97}	2016-03-20	62	62fH8oKr8aiT	sensor-pad	1458444054161	31	151.198.215.1	Fahrenheit
2	{"latitude": 47, "longitude": 8}	2016-03-20	241	241un29KmR	meter-gauge	1458444054259	64	80.84.21.105	Fahrenheit
3	{"latitude": 42.36, "longitude": -71.05}	2016-03-20	260	260F7DTEbnz	sensor-pad	1458444054267	57	72.21.168.109	Fahrenheit
4	{"latitude": 51.45, "longitude": 7.02}	2016-03-20	298	298hJceoQv0	sensor-pad	1458444054280	84	217.76.103.233	Fahrenheit
5	{"latitude": 38.88, "longitude": -92.4}	2016-03-20	301	301wHcyNzw	meter-gauge	1458444054281	26	150.199.7.154	Fahrenheit
6	{"latitude": 20, "longitude": 77}	2016-03-20	334	334DUAg4mbXi	sensor-pad	1458444054292	72	202.71.135.89	Fahrenheit
7	{"latitude": 47, "longitude": 8}	2016-03-20	349	349gRZSGtcGR	meter-gauge	1458444054296	45	195.219.212.9	Fahrenheit
8	{"latitude": 51.51, "longitude": -0.09}	2016-03-20	383	383yNMmfRq0zG	meter-gauge	1458444054308	46	166.49.222.157	Fahrenheit
9	{"latitude": 53.5, "longitude": -2.19}	2016-03-20	391	391Qn3lLA	meter-gauge	1458444054310	97	89.21.16.18	Fahrenheit
10	{"latitude": 37.29, "longitude": 127.01}	2016-03-20	455	455L4J9FUG	therm-stick	1458444054330	64	211.41.134.73	Fahrenheit

EXAMPLE 3: DYNAMIC SCHEMA INFERENCE

AL dynamically searches a sample of the dataset to determine nested structure. This avoids costly and slow full dataset scans to infer schema. The following configurations are available to adjust the amount of sample data used on read to discover initial schema:

1. spark.databricks.cloudFiles.schemaInference.sampleSize.numBytes (default 50 GB)
2. spark.databricks.cloudFiles.schemaInference.sampleSize.numFiles (default 1000 files)

```
%scala
val inferAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.schemaLocation", repoSchemaPath)
.option("cloudFiles.inferColumnTypes", true) // schema inference
.load(jsonSchema1Path)
)
```

```
%scala
inferAlDf.printSchema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: long (nullable = true)
|-- c02_level: long (nullable = true)
|-- cca2: string (nullable = true)
|-- cca3: string (nullable = true)
|-- cn: string (nullable = true)
|-- coordinates: struct (nullable = true)
| |-- latitude: double (nullable = true)
| |-- longitude: double (nullable = true)
|-- date: string (nullable = true)
|-- device_id: long (nullable = true)
|-- device_serial_number: string (nullable = true)
|-- device_type: string (nullable = true)
|-- epoch_time_miliseconds: long (nullable = true)
|-- humidity: long (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
|-- _rescued_data: string (nullable = true)
```

AL saves the initial schema to the schema location path provided. This schema serves as the base version for the stream during incremental processing. Dynamic schema inference is an automated approach to applying schema changes over time.

```
%scala
display(inferAlDf.limit(10))
```

Table + New result table: OFF

	alarm_status	battery_level	c02_level	cca2	cca3	cn	coordinates	date	device_id	device_serial_num
1	yellow	3	1082	US	USA	United States	{"latitude": 38, "longitude": -97}	2016-03-20	62	62fH8oKr8aiT
2	green	0	920	US	USA	null	{"latitude": 47, "longitude": 8}	2016-03-20	241	241un29KmR
3	yellow	7	1004	US	USA	United States	{"latitude": 42.36, "longitude": -71.05}	2016-03-20	260	260F7DTEbnz
4	yellow	9	1081	DE	DEU	Germany	{"latitude": 51.45, "longitude": 7.02}	2016-03-20	298	298hJceoQv0
5	yellow	8	1311	US	USA	United States	{"latitude": 38.88, "longitude": -92.4}	2016-03-20	301	301wHcyNzw
6	red	1	1484	IN	IND	India	{"latitude": 20, "longitude": 77}	2016-03-20	334	334DUAg4mbXi
7	yellow	9	1266	US	USA	null	{"latitude": 47, "longitude": 8}	2016-03-20	349	349gRZSGtcGR
8	yellow	1	1085	GB	GBR	United Kingdom	{"latitude": 51.51, "longitude": -0.09}	2016-03-20	383	383yNMmfRq0zG
9	red	3	1508	GB	GBR	United Kingdom	{"latitude": 53.5, "longitude": -2.19}	2016-03-20	391	391Qn3ILA
10	yellow	5	1191	KR	KOR	Republic of Korea	{"latitude": 37.29, "longitude": 127.01}	2016-03-20	455	455L4J9FUG

EXAMPLE 4: STATIC USER-DEFINED SCHEMA

AL also supports static custom schemas just like Spark Structured Streaming. This eliminates the need for dynamic schema-on-read inference scans, which trigger additional Spark jobs and schema versions. The schema can be retrieved as a DDL string or a JSON payload.

DDL

```
%scala
inferAlDf.schema.toDDL
```

```
String = alarm_status STRING,battery_level BIGINT,c02_level BIGINT,cca2 STRING,cca3
STRING,cn STRING,coordinates STRUCT<latitude: DOUBLE, longitude: DOUBLE>,date
STRING,device_id BIGINT,device_serial_number STRING,device_type STRING,epoch_time_
miliseconds BIGINT,humidity BIGINT,ip STRING,scale STRING,temp DOUBLE,timestamp
STRING,_rescued_data STRING
```

JSON

```
%scala
spark.read.json(repoSchemaPath + "/_schemas").select("dataSchemaJson").
where("dataSchemaJson is not null").first()
```

```
org.apache.spark.sql.Row = [{"type":"struct","fields":[{"name":"alarm_
status","type":"string","nullable":true,"metadata":{}},{"name":"battery_level","type":
"long","nullable":true,"metadata":{}},{"name":"c02_level","type":"long","nullable":true,
"metadata":{}},{"name":"cca2","type":"string","nullable":true,"metadata":{}},{"name":
"cca3","type":"string","nullable":true,"metadata":{}},{"name":"cn","type":"string",
"nullable":true,"metadata":{}},{"name":"coordinates","type":{"type":"struct","fields":
[{"name":"latitude","type":"double","nullable":true,"metadata":{}},{"name":"longitude",
"type":"double","nullable":true,"metadata":{}}]},"nullable":true,"metadata":{}},
{"name":"date","type":"string","nullable":true,"metadata":{}},{"name":"device_id",
"type":"long","nullable":true,"metadata":{}},{"name":"device_serial_number","type":
"string","nullable":true,"metadata":{}},{"name":"device_type","type":"string",
"nullable":true,"metadata":{}},{"name":"epoch_time_miliseconds","type":"long",
"nullable":true,"metadata":{}},{"name":"humidity","type":"long","nullable":true,
"metadata":{}},{"name":"ip","type":"string","nullable":true,"metadata":{}},{"name":
"scale","type":"string","nullable":true,"metadata":{}},{"name":"temp","type":"double",
"nullable":true,"metadata":{}},{"name":"timestamp","type":"string","nullable":true,
"metadata":{}}]}]
```

Here's an example of how to generate a user-defined StructType (Scala) | StructType (Python) via DDL DataFrame command or JSON queried from AL schema repository.

```
%scala
import org.apache. spark.sql. types. {DataType, StructType}

val ddl = """alarm_status STRING, battery_level BIGINT,c02_level BIGINT,cca2 STRING,
cca3 STRING, cn STRING, coordinates STRUCT<latitude: DOUBLE, longitude: DOUBLE>,
date STRING, device_id BIGINT, device_serial_number STRING, device_type STRING,
epoch_time_miliseconds BIGINT, humidity BIGINT, ip STRING, scale STRING,temp DOUBLE,
timestamp STRING, _rescued_data STRING"""

val ddlSchema = StructType.fromDDL(ddl)

val json = """{"type":"struct","fields":[{"name":"alarm_status","type":"string",
"nullable":true,"metadata":{}},{"name":"battery_level","type":"long","nullable":
true,"metadata":{}},{"name":"c02_level","type":"long","nullable":true, "metadata"
:{}},{"name":"cca2","type":"string","nullable":true,"metadata":{}},{"name":"cca3",
"type":"string","nullable":true,"metadata":{}},{"name":"cn","type":"string","nullable":
true,"metadata":{}},{"name":"coordinates","type":{"type":"struct","fields":
[{"name":"latitude","type":"double","nullable":true,"metadata":{}},{"name":"longitude",
"type":"double","nullable":true,"metadata":{}}]},"nullable":true,"metadata":{}},{"name"
:"date","type":"string","nullable":true,"metadata":{}},{"name":"device_id","type":
"long","nullable":true,"metadata":{}},{"name":"deviceserial_number","type":"string",
"nullable":true,"metadata":{}},{"name":"device_type","type":"string","nullable":true,
"metadata":{}},{"name":"epochtime_miliseconds","type":"long","nullable":true,"metadata":{}}
,{"name":"humidity","type":"long","nullable":true,"metadata":{}},{"name":
"ip","type":"string","nullable":true,"metadata":{}},{"name":"scale","type":"string",
"nullable":true,"metadata":{}},{"name":"temp","type":"double","nullable":true,
"metadata":{}},{"name":"timestamp","type":"string","nullable":true,"metadata":{}}]}"""

val jsonSchema = DataType. fromJson(json).asInstanceOf[StructType]
```

```
%scala
val schemaAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.schema(jsonSchema) // schema structtype definition
.load(jsonSchema1Path)
)
```

```
%scala
schemaAlDf.printSchema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: long (nullable = true)
|-- c02_level: long (nullable = true)
|-- cca2: string (nullable = true)
|-- cca3: string (nullable = true)
|-- cn: string (nullable = true)
|-- coordinates: struct (nullable = true)
| |-- latitude: double (nullable = true)
| |-- longitude: double (nullable = true)
|-- date: string (nullable = true)
|-- device_id: long (nullable = true)
|-- device_serial_number: string (nullable = true)
|-- device_type: string (nullable = true)
|-- epoch_time_miliseconds: long (nullable = true)
|-- humidity: long (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
```

Passing in the schema definition will enforce the stream. AL also provides a schema enforcement option achieving basically the same results as providing a static StructType schema-on-read. This method will be covered in Example 7.

```
%scala
display(schemaAlDf.limit(10))
```

Table + New result table: OFF

	alarm_status	battery_level	c02_level	cca2	cca3	cn	coordinates	date	device_id	device_serial_num
1	yellow	3	1082	US	USA	United States	{"latitude": 38, "longitude": -97}	2016-03-20	62	62fH8oKr8aiT
2	green	0	920	US	USA	null	{"latitude": 47, "longitude": 8}	2016-03-20	241	241un29KmR
3	yellow	7	1004	US	USA	United States	{"latitude": 42.36, "longitude": -71.05}	2016-03-20	260	260F7DTEbnz
4	yellow	9	1081	DE	DEU	Germany	{"latitude": 51.45, "longitude": 7.02}	2016-03-20	298	298hJceoQv0
5	yellow	8	1311	US	USA	United States	{"latitude": 38.88, "longitude": -92.4}	2016-03-20	301	301wHcyNzw
6	red	1	1484	IN	IND	India	{"latitude": 20, "longitude": 77}	2016-03-20	334	334DUAg4mbXi
7	yellow	9	1266	US	USA	null	{"latitude": 47, "longitude": 8}	2016-03-20	349	349gRZSGtcGR
8	yellow	1	1085	GB	GBR	United Kingdom	{"latitude": 51.51, "longitude": -0.09}	2016-03-20	383	383yNMmfRq0zG
9	red	3	1508	GB	GBR	United Kingdom	{"latitude": 53.5, "longitude": -2.19}	2016-03-20	391	391Qn3lLA
10	yellow	5	1191	KR	KOR	Republic of Korea	{"latitude": 37.29, "longitude": 127.01}	2016-03-20	455	455L4J9FUG

EXAMPLE 5: SCHEMA DRIFT

AL stores new columns and data types via the rescue column. This column captures schema changes-on-read. The stream does not fail when schema and data type mismatches are discovered. This is a very impressive feature!

```scala
%scala
val driftAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.schemaLocation", repoSchemaPath)
.option("cloudFiles.inferColumnTypes", true)
.option("cloudFiles.schemaEvolutionMode", "rescue") // schema drift tracking
.load(rawBasePath + "/*.json")
)
```

```scala
%scala
driftAlDf.printSchema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: long (nullable = true)
|-- c02_level: long (nullable = true)
|-- cca2: string (nullable = true)
|-- cca3: string (nullable = true)
|-- cn: string (nullable = true)
|-- coordinates: struct (nullable = true)
| |-- latitude: double (nullable = true)
| |-- longitude: double (nullable = true)
|-- date: string (nullable = true)
|-- device_id: long (nullable = true)
|-- device_serial_number: string (nullable = true)
|-- device_type: string (nullable = true)
|-- epoch_time_miliseconds: long (nullable = true)
|-- humidity: long (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
|-- _rescued_data: string (nullable = true)
```

The rescue column preserves schema drift such as newly appended columns and/or different data types via a JSON string payload. This payload can be parsed via Spark DataFrame or Dataset APIs to analyze schema drift scenarios. The source file path for each individual row is also available in the rescue column to investigate the root cause.

```scala
%scala
display(driftAlDf.where("_rescued_data is not null").limit(10))
```

Table + New result table: OFF

	seconds	humidity	ip	scale	temp	timestamp	_rescued_data
1		null	213.161.254.1	Fahrenheit	51.8	2016-03-20 03:20:54.119	{"location":{"cca3":"NOR","cn":"Norway","cca2":"NO"},"latitude":62.47,"device_serial_number_device_type":"{\"2n2Pea\":\"sensor-pad\"}","longitude":6.15,"humidity":70.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
2		null	68.28.91.22	Fahrenheit	53.6	2016-03-20 03:20:54.126	{"location":{"cca3":"USA","cn":"United States","cca2":"US"},"latitude":38.0,"device_serial_number_device_type":"{\"12Y2klm0o\":\"sensor-pad\"}","longitude":-97.0,"humidity":92.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
3		null	59.90.65.1	Fahrenheit	73.4	2016-03-20 03:20:54.133	{"location":{"cca3":"IND","cn":"India","cca2":"IN"},"latitude":12.98,"device_serial_number_device_type":"{\"230IupA\":\"meter-gauge\"}","longitude":77.58,"humidity":47.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
4		null	213.7.14.1	Fahrenheit	75.2	2016-03-20 03:20:54.141	{"location":{"cca3":"CYP","cn":"Cyprus","cca2":"CY"},"latitude":35.0,"device_serial_number_device_type":"{\"36VQv8fnEg\":\"sensor-pad\"}","longitude":33.0,"humidity":47.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
5		null	62.128.16.74	Fahrenheit	80.6	2016-03-20 03:20:54.149	{"location":{"cca3":"DEU","cn":"Germany","cca2":"DE"},"latitude":49.46,"device_serial_number_device_type":"{\"448DeWGLL\":\"sensor-pad\"}","longitude":11.1,"humidity":63.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
6		null	170.37.224.1	Fahrenheit	77	2016-03-20 03:20:54.152	{"location":{"cca3":"USA","cn":"United States","cca2":"US"},"latitude":42.28,"device_serial_number_device_type":"{\"49YesGXwt\":\"meter-gauge\"}","longitude":-71.44,"humidity":70.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
7		null	38.99.198.186	Fahrenheit	53.6	2016-03-20 03:20:54.162	{"location":{"cca3":"USA","cn":"United States","cca2":"US"},"latitude":38.0,"device_serial_number_device_type":"{\"64djcln\":\"sensor-pad\"}","longitude":-97.0,"humidity":55.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
8		null	218.248.255.30	Fahrenheit	62.6	2016-03-20 03:20:54.169	{"location":{"cca3":"IND","cn":"India","cca2":"IN"},"latitude":12.98,"device_serial_number_device_type":"{\"77IKW3YAB55\":\"meter-gauge\"}","longitude":77.58,"humidity":82.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
9		null	159.128.0.181	Fahrenheit	89.6	2016-03-20 03:20:54.171	{"location":{"cca3":"CAN","cn":"Canada","cca2":"CA"},"latitude":50.01,"device_serial_number_device_type":"{\"80TY4dWSMH\":\"sensor-pad\"}","longitude":-97.22,"humidity":57.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
10		null	24.32.26.1	Fahrenheit	82.4	2016-03-20 03:20:54.179	{"location":{"cca3":"USA","cn":"United States","cca2":"US"},"latitude":39.33,"device_serial_number_device_type":"{\"94HL9jChDl\":\"sensor-pad\"}","longitude":-120.25,"humidity":66.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}

EXAMPLE 6: SCHEMA EVOLUTION

AL merges schemas as new columns arrive via schema evolution mode. New schema JSON will be updated and stored as a new version in the specified schema repository location.

```scala
%scala
val evolveAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.schemaLocation", repoSchemaPath)
.option("cloudFiles.inferColumnTypes", true)
.option("cloudFiles.schemaEvolutionMode", "addNewColumns") // schema evolution
.load(rawBasePath + "/*.json")
)
```

```scala
%scala
evolveAlDf.printSchema // original schema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: long (nullable = true)
|-- c02_level: long (nullable = true)
|-- cca2: string (nullable = true)
|-- cca3: string (nullable = true)
|-- cn: string (nullable = true)
|-- coordinates: struct (nullable = true)
| |-- latitude: double (nullable = true)
| |-- longitude: double (nullable = true)
|-- date: string (nullable = true)
|-- device_id: long (nullable = true)
|-- device_serial_number: string (nullable = true)
|-- device_type: string (nullable = true)
|-- epoch_time_miliseconds: long (nullable = true)
|-- humidity: long (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
|-- _rescued_data: string (nullable = true)
```

```scala
%scala
display(evolveAlDf.limit(10)) // # stream will fail
```

AL purposely fails the stream with an UnknownFieldException error when it detects a schema change via dynamic schema inference. The updated schema instance is created as a new version and metadata file in the schema repository location, and will be used against the input data after restarting the stream.

```scala
%scala
val evolveAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.schemaLocation", repoSchemaPath)
.option("cloudFiles.inferColumnTypes", true)
.option("cloudFiles.schemaHints", "humidity DOUBLE")
.option("cloudFiles.schemaEvolutionMode", "addNewColumns") // schema evolution
.load(rawBasePath + "/*.json")
)
```

```scala
%scala
evolveAlDf.printSchema // evolved schema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: long (nullable = true)
|-- c02_level: long (nullable = true)
|-- cca2: string (nullable = true)
|-- cca3: string (nullable = true)
|-- cn: string (nullable = true)
|-- coordinates: struct (nullable = true)
| |-- latitude: double (nullable = true)
| |-- longitude: double (nullable = true)
|-- date: string (nullable = true)
|-- device_id: long (nullable = true)
|-- device_serial_number: string (nullable = true)
|-- device_type: string (nullable = true)
|-- epoch_time_miliseconds: long (nullable = true)
|-- humidity: double (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
|-- device_serial_number_device_type: string (nullable = true)
|-- latitude: double (nullable = true)
|-- location: struct (nullable = true)
| |-- cca2: string (nullable = true)
| |-- cca3: string (nullable = true)
| |-- cn: string (nullable = true)
|-- longitude: double (nullable = true)
|-- _rescued_data: string (nullable = true)
```

AL has evolved the schema to merge the newly acquired data fields.

```scala
%scala
display(evolveAlDf.where("device_serial_number_device_type is not null").limit(10))
```

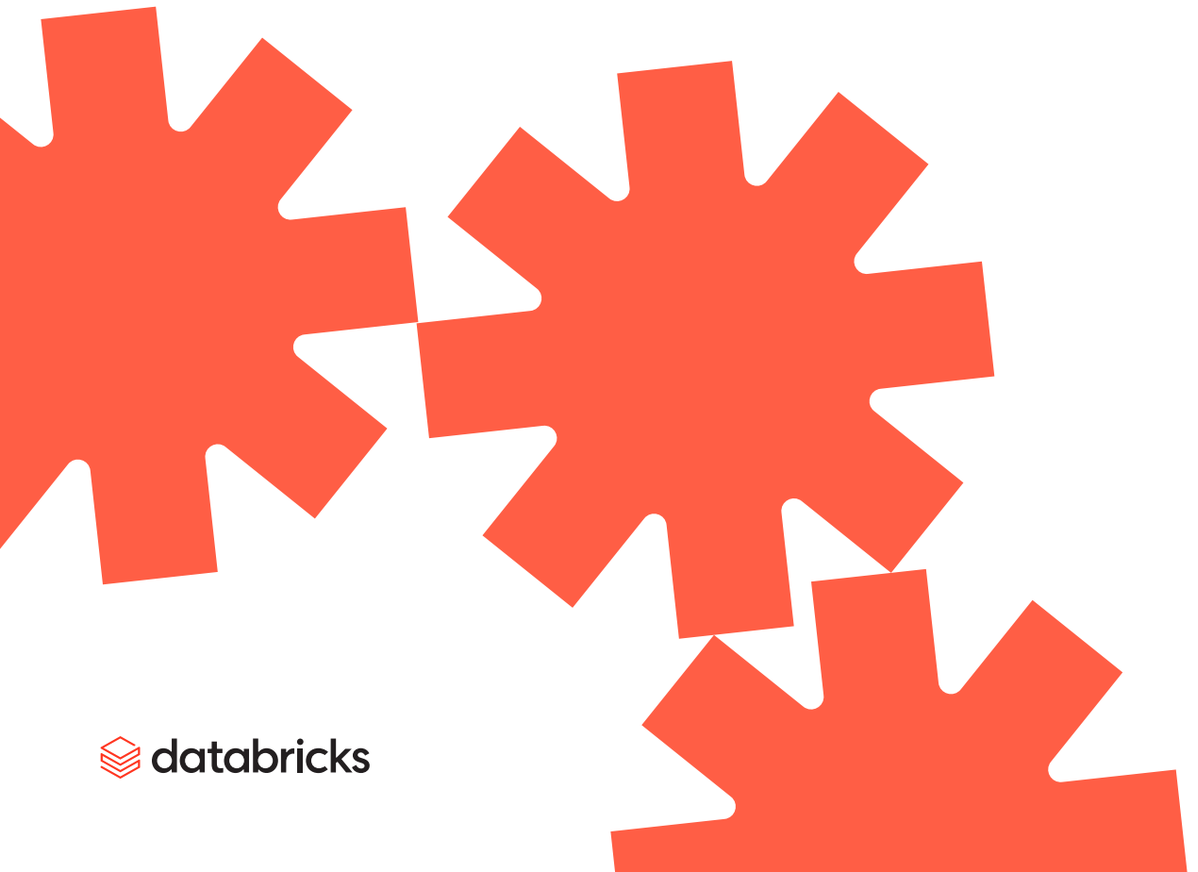


Table + New result table: OFF

	temp	timestamp	device_serial_number_device_type	latitude	location	longitude	_rescued_data
1	51.8	2016-03-20 03:20:54.119	{"2n2Pea":"sensor-pad"}	62.47	{"cca2": "NO", "cca3": "NOR", "cn": "Norway"}	6.15	null
2	53.6	2016-03-20 03:20:54.126	{"12Y2klm0o":"sensor-pad"}	38	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-97	null
3	73.4	2016-03-20 03:20:54.133	{"230IupA":"meter-gauge"}	12.98	{"cca2": "IN", "cca3": "IND", "cn": "India"}	77.58	null
4	75.2	2016-03-20 03:20:54.141	{"36VQv8fnEg":"sensor-pad"}	35	{"cca2": "CY", "cca3": "CYP", "cn": "Cyprus"}	33	null
5	80.6	2016-03-20 03:20:54.149	{"448DeWGL":"sensor-pad"}	49.46	{"cca2": "DE", "cca3": "DEU", "cn": "Germany"}	11.1	null
6	77	2016-03-20 03:20:54.152	{"49YesGXwt":"meter-gauge"}	42.28	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-71.44	null
7	53.6	2016-03-20 03:20:54.162	{"64djcln":"sensor-pad"}	38	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-97	null
8	62.6	2016-03-20 03:20:54.169	{"77IKW3YAB55":"meter-gauge"}	12.98	{"cca2": "IN", "cca3": "IND", "cn": "India"}	77.58	null
9	89.6	2016-03-20 03:20:54.171	{"80TY4dWSMH":"sensor-pad"}	50.01	{"cca2": "CA", "cca3": "CAN", "cn": "Canada"}	-97.22	null
10	82.4	2016-03-20 03:20:54.179	{"94HL9jChD":"sensor-pad"}	39.33	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-120.25	null

The newly merged schema transformed by AL is stored in the original schema repository path as version 1 along with the base version 0 schema. This history is valuable for tracking changes to schema over time, as well as quickly retrieving DDL on the fly for schema enforcement.

Schema Repository

```
%scala
display(dbutils.fs.ls(repoSchemaPath + "/_schemas"))
```

Table + New result table: OFF

	path	name	size	modificationTime
1	dbfs:/user/garrett.peternel@databricks.com/repo/iot-ddl.json/_schemas/0	0	1628	1709325391000
2	dbfs:/user/garrett.peternel@databricks.com/repo/iot-ddl.json/_schemas/1	1	2209	1709325578000

Schema Metadata

```
%scala
display(spark.read.json(repoSchemaPath + "/_schemas"))
```

Table + New result table: OFF

	_corrupt_record	dataSchemaJson	partitionSchemaJson
1	v1	null	null
2	null	{"type":"struct","fields":[{"name":"alarm_status","type":"string","nullable":true,"metadata":{}},{"name":"battery_level","type":"long","nullable":true,"metadata":{}},{"name":"c02_level","type":"long","nullable":true,"metadata":{}},{"name":"cca2","type":"string","nullable":true,"metadata":{}},{"name":"cca3","type":"string","nullable":true,"metadata":{}},{"name":"cn","type":"string","nullable":true,"metadata":{}},{"name":"coordinates","type":{"type":"struct","fields":[{"name":"latitude","type":"do...	{"type":"struct","fields":[]}
3	v1	null	null
4	null	{"type":"struct","fields":[{"name":"alarm_status","type":"string","nullable":true,"metadata":{}},{"name":"battery_level","type":"long","nullable":true,"metadata":{}},{"name":"c02_level","type":"long","nullable":true,"metadata":{}},{"name":"cca2","type":"string","nullable":true,"metadata":{}},{"name":"cca3","type":"string","nullable":true,"metadata":{}},{"name":"cn","type":"string","nullable":true,"metadata":{}},{"name":"coordinates","type":{"type":"struct","fields":[{"name":"latitude","type":"do...	{"type":"struct","fields":[]}

Schema evolution can be a messy problem if frequent. With AL and Delta Lake it becomes easier and simpler to manage. Adding new columns is relatively straightforward as AL combined with Delta Lake uses schema evolution to append them to the existing schema. Note: the values for these columns will be NULL for data already processed. The greater challenge occurs when the data types change because there will be a type mismatch against the data already processed. Currently, the "safest" approach is to perform a complete overwrite of the target Delta table to refresh all data with the changed data type(s). Depending on the data volume this operation is also relatively straightforward if infrequent. However, if data types are changing daily/weekly then this operation is going to be very costly to reprocess large data volumes. This can be an indication that the business needs to improve their data strategy.

Constantly changing schemas can be a sign of a weak data governance strategy and lack of communication with the data business owners. Ideally, organizations should have some kind of SLA for data acquisition and know the expected schema. Raw data stored in the landing zone should also follow some kind of pre-ETL strategy (e.g., ontology, taxonomy, partitioning) for better incremental loading performance into the lakehouse. Skipping these steps can cause a plethora of data management issues that will negatively impact downstream consumers building data analytics, BI, and AI/ML pipelines and applications. If upstream schema and formatting issues are never addressed, downstream pipelines will consistently break and result in increased cloud storage and compute costs. Garbage in, garbage out.

EXAMPLE 7: SCHEMA ENFORCEMENT

AL validates data against the linked schema version stored in repository location via schema enforcement mode. Schema enforcement is a schema-on-write operation, and only ingested data matching the target Delta Lake schema will be written to output. Any future input schema changes will be ignored, and AL streams will continue working without failure. Schema enforcement is a very powerful feature of AL and Delta Lake. It ensures only clean and trusted data will be inserted into downstream Silver/Gold datasets used for data analytics, BI, and AI/ML pipelines and applications.

```
%scala
val enforceAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.schemaLocation", repoSchemaPath)
.option("cloudFiles.schemaEvolutionMode", "none") // schema enforcement
.schema(jsonSchema)
.load(rawBasePath + "/*.json")
)
```

```
%scala
enforceAlDf.printSchema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: long (nullable = true)
|-- c02_level: long (nullable = true)
|-- cca2: string (nullable = true)
|-- cca3: string (nullable = true)
|-- cn: string (nullable = true)
|-- coordinates: struct (nullable = true)
| |-- latitude: double (nullable = true)
| |-- longitude: double (nullable = true)
|-- date: string (nullable = true)
|-- device_id: long (nullable = true)
|-- device_serial_number: string (nullable = true)
|-- device_type: string (nullable = true)
|-- epoch_time_miliseconds: long (nullable = true)
|-- humidity: long (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
```

Please note the rescue column is no longer available in this example because schema enforcement has been enabled. However, a rescue column can still be configured separately as an AL option if desired. In addition, schema enforcement mode uses the latest schema version in the repository to enforce incoming data. For older versions, set a user-defined schema as explained in Example 4.

```
%scala
display(enforceAlDf.limit(10))
```

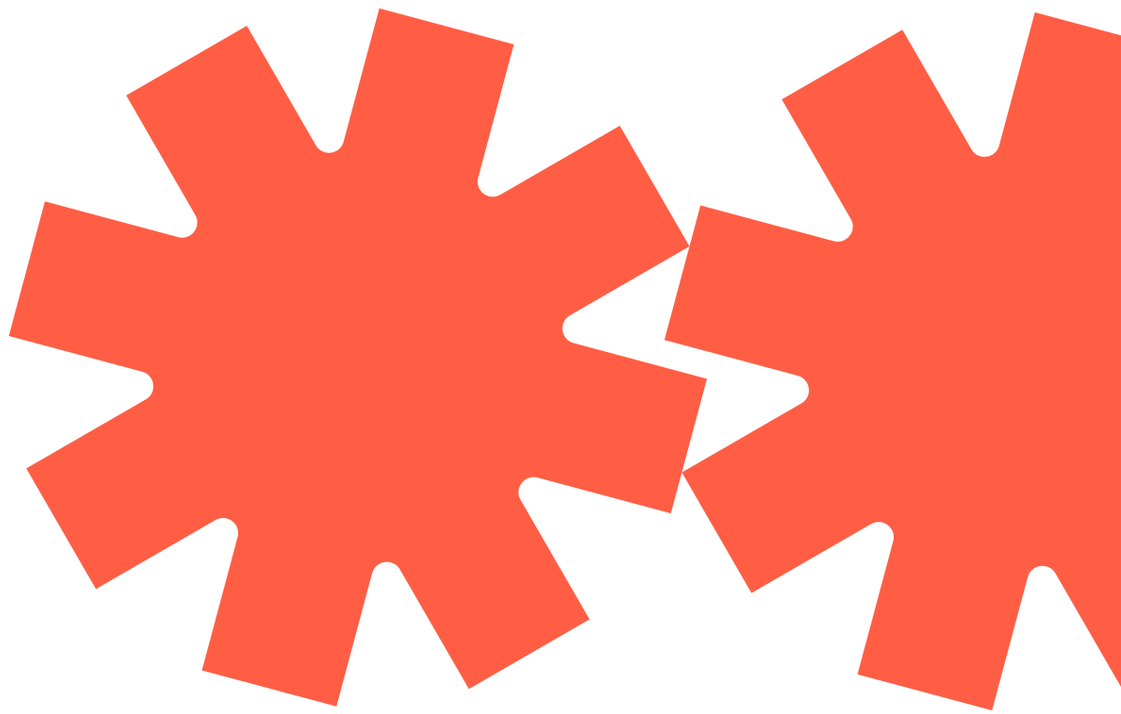
Table + New result table: OFF

	alarm_status	battery_level	c02_level	cca2	cca3	cn	coordinates	date	device_id	device_serial_num
1	yellow	3	1082	US	USA	United States	{"latitude": 38, "longitude": -97}	2016-03-20	62	62fH8oKr8aiT
2	green	0	920	US	USA	null	{"latitude": 47, "longitude": 8}	2016-03-20	241	241un29KmR
3	yellow	7	1004	US	USA	United States	{"latitude": 42.36, "longitude": -71.05}	2016-03-20	260	260F7DTEbnz
4	yellow	9	1081	DE	DEU	Germany	{"latitude": 51.45, "longitude": 7.02}	2016-03-20	298	298hJceoQv0
5	yellow	8	1311	US	USA	United States	{"latitude": 38.88, "longitude": -92.4}	2016-03-20	301	301wHcyNzw
6	red	1	1484	IN	IND	India	{"latitude": 20, "longitude": 77}	2016-03-20	334	334DUAg4mbXi
7	yellow	9	1266	US	USA	null	{"latitude": 47, "longitude": 8}	2016-03-20	349	349gRZSGtcGR
8	yellow	1	1085	GB	GBR	United Kingdom	{"latitude": 51.51, "longitude": -0.09}	2016-03-20	383	383yNMmfRq0zG
9	red	3	1508	GB	GBR	United Kingdom	{"latitude": 53.5, "longitude": -2.19}	2016-03-20	391	391Qn3lLA
10	yellow	5	1191	KR	KOR	Republic of Korea	{"latitude": 37.29, "longitude": 127.01}	2016-03-20	455	455L4J9FUG

CONCLUSION

At the end of the day, data issues are inevitable. However, the key is to limit data pollution as much as possible and have methods to detect discrepancies, changes and history via schema management. Databricks Auto Loader provides many solutions for schema management, as illustrated by the examples in this chapter. Having a solidified data governance and landing zone strategy will make ingestion and streaming easier and more efficient for loading data into the lakehouse. Whether it is simply converting raw JSON data incrementally to the Bronze layer as Delta Lake format, or having a repository to store schema metadata, AL makes your job easier. It acts as an anchor to building a resilient lakehouse architecture that provides reusable, consistent, reliable and performant data throughout the data and AI lifecycle.

HTML notebooks (Spark Scala and Spark Python) with code and both sample datasets can be found at the GitHub repo here.



From Idea to Code: Building With the Databricks SDK for Python

by Kimberly Mahoney

The focus of this chapter is to demystify the Databricks SDK for Python — the authentication process and the components of the SDK — by walking through the start-to-end development process. I'll also be showing how to utilize IntelliSense and the debugger for real-time suggestions in order to reduce the amount of context-switching from the IDE to documentation and code examples.

What is the Databricks SDK for Python . . . and why you should use it

The Databricks Python SDK lets you interact with the Databricks Platform programmatically using Python. It covers the entire Databricks API surface and Databricks REST operations. While you can interact directly with the API via curl or a library like 'requests' there are benefits to utilizing the SDKs such as:

- Secure and simplified authentication via Databricks client-unified authentication
- Built-in debug logging with sensitive information automatically redacted
- Support to wait for long-running operations to finish (kicking off a job, starting a cluster)
- Standard iterators for paginated APIs (we have multiple pagination types in our APIs!)
- Retrying on transient errors

There are numerous practical applications, such as building multi-tenant web applications that interact with your ML models or a robust UC migration toolkit like Databricks Labs project UCX. Don't forget the silent workhorses — those simple utility scripts that are more limited in scope but automate an annoying task such as bulk updating cluster policies, dynamically adding users to groups or simply writing data files to UC Volumes. Implementing these types of scripts is a great way to familiarize yourself with the Python SDK and Databricks APIs.

SCENARIO

Imagine my business is establishing best practices for development and CI/CD on Databricks. We're adopting DABs to help us define and deploy workflows in our development and production workspaces, but in the meantime, we need to audit and clean up our current environments. We have a lot of jobs people created in our dev workspace via the UI. One of the platform admins observed many of these jobs are inadvertently configured to run on a recurring schedule, racking up unintended costs. As part of the cleanup process, we want to identify any scheduled jobs in our development workspace with an option to pause them. We'll need to figure out:

- How to install the SDK
- How to connect to the Databricks workspace
- How to list all the jobs and examine their attributes
- How to log the problematic jobs — or a step further, how to call the API to pause their schedule

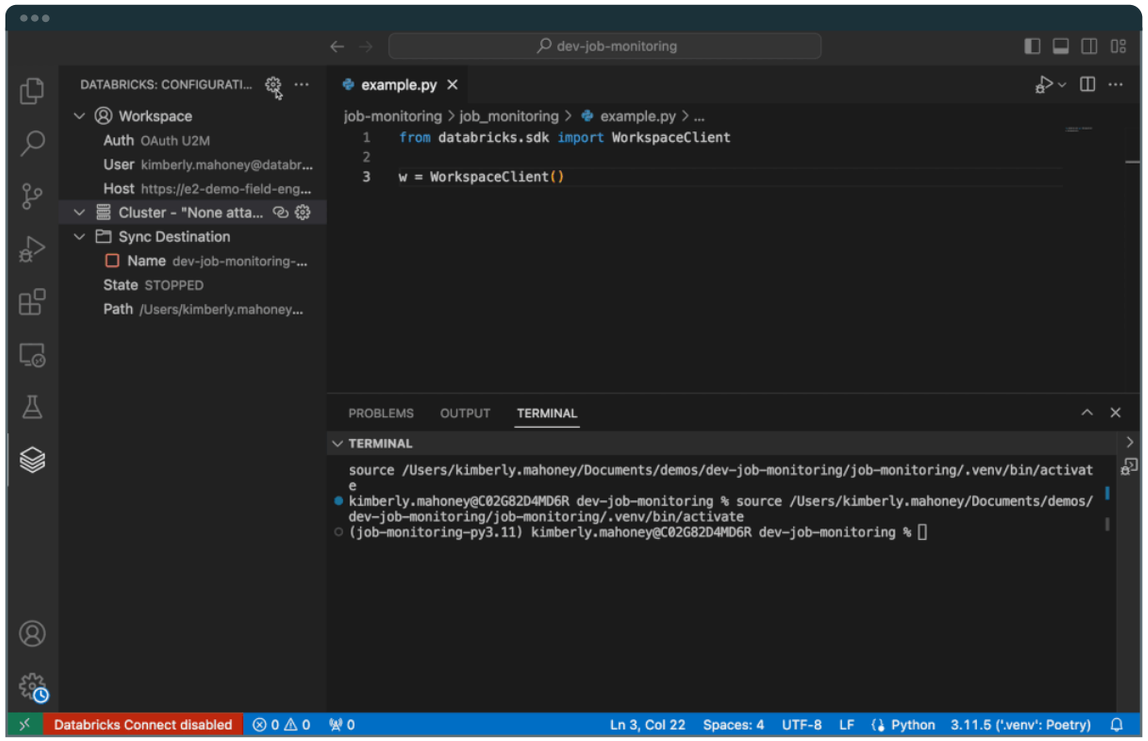
DEVELOPMENT ENVIRONMENT

Before diving into the code, you need to set up your development environment. I highly recommend using an IDE that has a comprehensive code completion feature as well as a debugger. Code completion features, such as IntelliSense in VS Code, are really helpful when learning new libraries or APIs — they provide useful contextual information, autocompletion, and aid in code navigation. For this chapter, I'll be using Visual Studio Code so I can also make use of the Databricks Extension as well as Pylance. You'll also need to install the databricks-sdk (docs). In this chapter, I'm using Poetry + Pyenv. The setup is similar for other tools — just 'poetry add databricks-sdk' or alternatively 'pip install databricks-sdk' in your environment.

AUTHENTICATION

The next step is to authorize access to Databricks so we can work with our workspace. There are several ways to do this, but because I'm using the VS Code Extension, I'll take advantage of its authentication integration. It's one of the tools that uses unified client authentication — that just means all these development tools follow the same process and standards for authentication and if you set up auth for one, you can reuse it among the other tools. I set up both the CLI and VS Code Extension previously, but here is a primer on setting up the CLI and installing the extension. Once you've connected successfully, you'll see a notification banner in the lower right-hand corner and see two hidden files generated in the *.databricks* folder — project.json and databricks.env (don't worry, the extension also handles adding these to .gitignore).

For this example, while we're interactively developing in our IDE, we'll be using what's called U2M (user-to-machine) OAuth. We won't get into the technical details, but OAuth is a secure protocol that handles authorization to resources without passing sensitive user credentials such as PAT or username/password that persist much longer than the one-hour short-lived OAuth token.



OAuth flow for the Databricks Python SDK

WORKSPACECLIENT VS. ACCOUNTCLIENT

The Databricks API is split into two primary categories — account and workspace. They let you manage different parts of Databricks, like user access at the account level or cluster policies in a workspace. The SDK reflects this with two clients that act as our entry points to the SDK — the WorkspaceClient and AccountClient. For our example we'll be working at the workspace level, so I'll be initializing the WorkspaceClient. If you're unsure which client to use, check out the SDK documentation.

Tip: Because we ran the previous steps to authorize access via unified client auth, the SDK will automatically use the necessary Databricks environment variables, so there's no need for extra configurations when setting up your client. All we need are these two lines of code:

- *Initializing our WorkspaceClient*

```
from databricks.sdk import WorkspaceClient

w = WorkspaceClient()
```

MAKING API CALLS AND INTERACTING WITH DATA

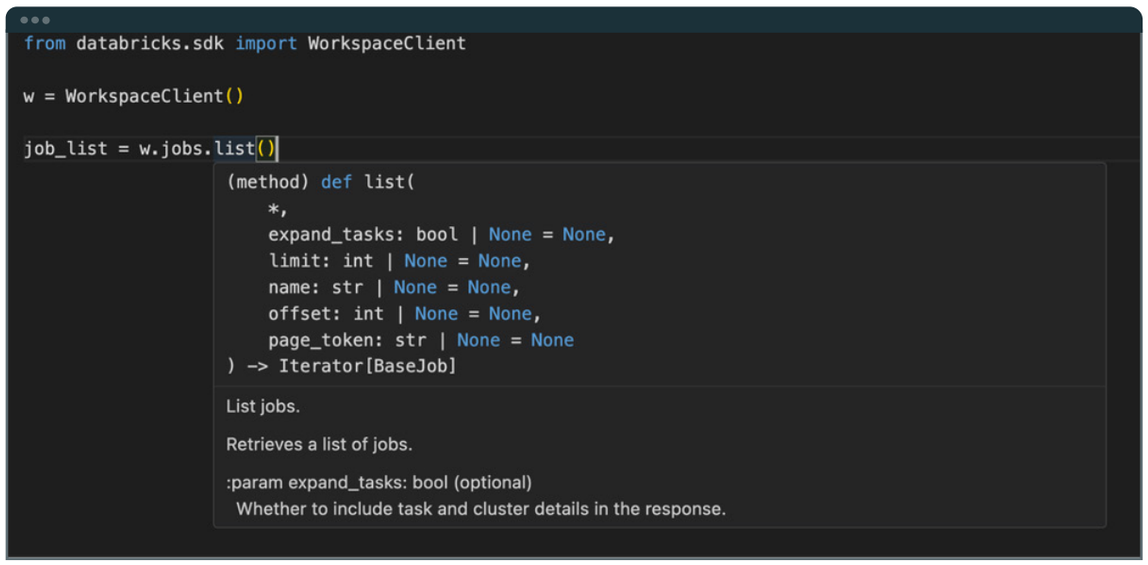
The WorkspaceClient we instantiated will allow us to interact with different APIs across the Databricks workspace services. A **service** is a smaller component of the Databricks Platform — e.g., Jobs, Compute, Model Registry.
In our example, we'll need to call the Jobs API in order to retrieve a list of all the jobs in the workspace.

```
import logging
import sys

w = WorkspaceClient()

w.
```

```
git_credentials
global_init_scripts
grants
groups
instance_pools
instance_profiles
ip_access_lists
jobs
libraries
metastores
model_registry
model_versions
```

Services accessible via the Python SDK

This is where IntelliSense really comes in handy. Instead of context switching between the IDE and the Documentation page, I can use autocomplete to provide a list of methods as well as examine the method description, the parameters and return types from within the IDE. I know the first step is getting a list of all the jobs in the workspace:



As you can see, it returns an iterator over an object called BaseJob. Before we talk about what a BaseJob actually is, it'll be helpful to understand how data is used in the SDK. To interact with data you are sending to and receiving from the API, the Python SDK takes advantage of Python data classes and enums. The main advantage of this approach over passing around dictionaries is improved readability while also minimizing errors through enforced type checks and validations.

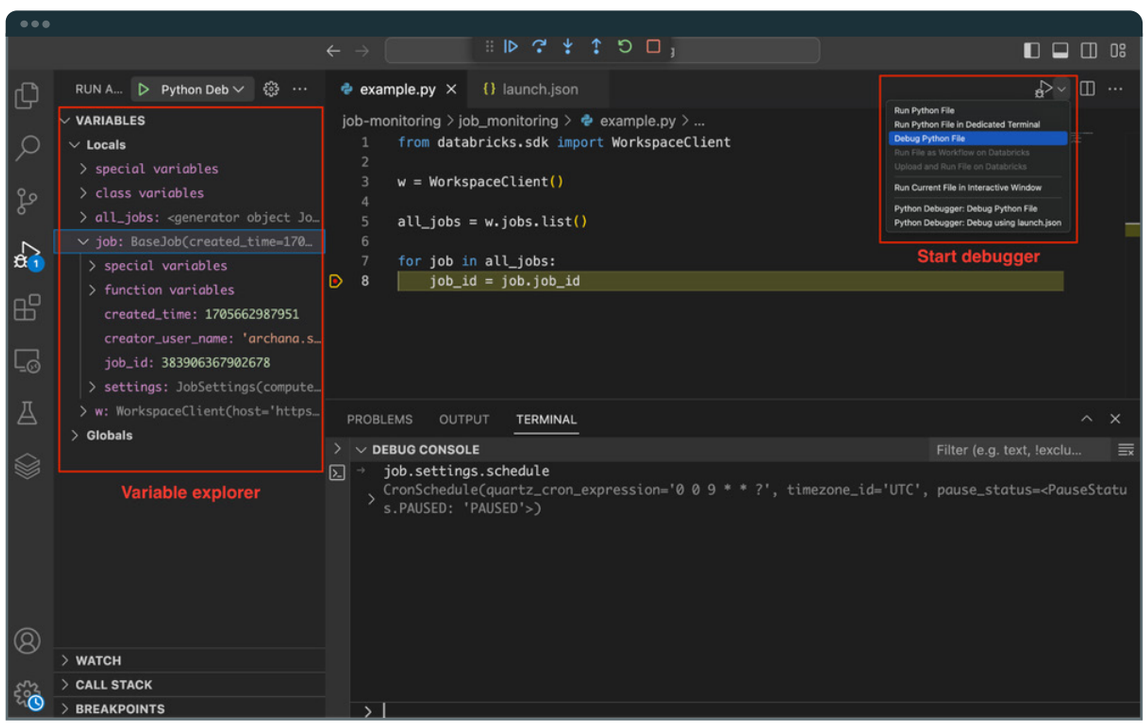
You can construct objects with data classes and interact with enums. For example:

- *Creating an Employee via Employee DataClass and company departments, using enums for possible department values*

```
1  from dataclasses import dataclass
2  from enum import Enum
3  from typing import Optional

4  class CompanyDepartment(Enum):
5      MARKETING = 'MARKETING'
6      SALES = 'SALES'
7      ENGINEERING = 'ENGINEERING'

8  @dataclass
9  class Employee:
10     name: str
11     email: str
12     department: Optional[CompanyDepartment] = None


13 emp = Employee('Bob', 'bob@company.com', CompanyDepartment.ENGINEERING)
```

In the Python SDK all of the data classes, enums and APIs belong to the same module for a service located under **databricks.sdk.service** — e.g., databricks.sdk.service.**jobs**, databricks.sdk.service.**billing**, databricks.sdk.service.**sql**.

For our example, we'll need to loop through all of the jobs and make a decision on whether or not they should be paused. I'll be using a debugger in order to look at a few example jobs and get a better understanding of what a 'BaseJob' looks like. The Databricks VS Code extension comes with a debugger that can be used to troubleshoot code issues interactively on Databricks via Databricks Connect. But, because I do not need to run my code on a cluster, I'll just be using the standard Python debugger. I'll set a breakpoint inside for my loop and use the VS Code Debugger to examine a few examples. A breakpoint allows us to stop code execution and interact with variables during our debugging session. This is preferable over print statements, as you can use the debugging console to interact with the data as well as progress the loop. In this example I'm looking at the **settings** field and drilling down further in the debugging console to take a look at what an example job schedule looks like:



Inspecting BaseJob in the VS Code debugger

We can see a BaseJob has a few top-level attributes and has a more complex Settings type that contains most of the information we care about. At this point, we have our WorkspaceClient and are iterating over the jobs in our workspace. To flag problematic jobs and potentially take some action, we'll need to better understand **job.settings.schedule**. We need to figure out how to programmatically identify if a job has a schedule and flag if it's not paused. For this we'll be using another handy utility for code navigation — Go to Definition. I've opted to Open Definition to the Side (⌘K F12) in order to reduce switching to a new window. This will allow us to quickly navigate through the data class definitions without having to switch to a new window or exit our IDE:

As we can see, a BaseJob contains some top-level fields that are common among jobs such as 'job_id' or 'created_time'. A job can also have various settings (JobSettings). These configurations often differ between jobs and encompass aspects like notification settings, tasks, tags and the schedule. We'll be focusing on the schedule field, which is represented by the CronSchedule data class. CronSchedule contains information about the pause status (PauseStatus) of a job. PauseStatus in the SDK is represented as an enum with two possible values — **PAUSED** and **UNPAUSED**.

Tip: VSCode + Pylance provides code suggestions, and you can enable auto imports in your User Settings or on a per-project basis in Workspace Settings. By default, only top-level symbols are suggested for auto import and suggested code (see original GitHub issue). However, the SDK has nested elements we want to generate suggestions for. We actually need to go down 5 levels — databricks.sdk.service.jobs.<Enum|Dataclass>. In order to take full advantage of these features for the SDK, I added a couple of workspace settings:

- *Selection of theVSCode Workspace settings.json*

```
...
  "python.analysis.autoImportCompletions": true,
  "python.analysis.indexing": true,
 "python.analysis.packageIndexDepths": [
{
            "name": "databricks",
            "depth": 5,
            "includeAllSymbols": true
        }
]
...
```

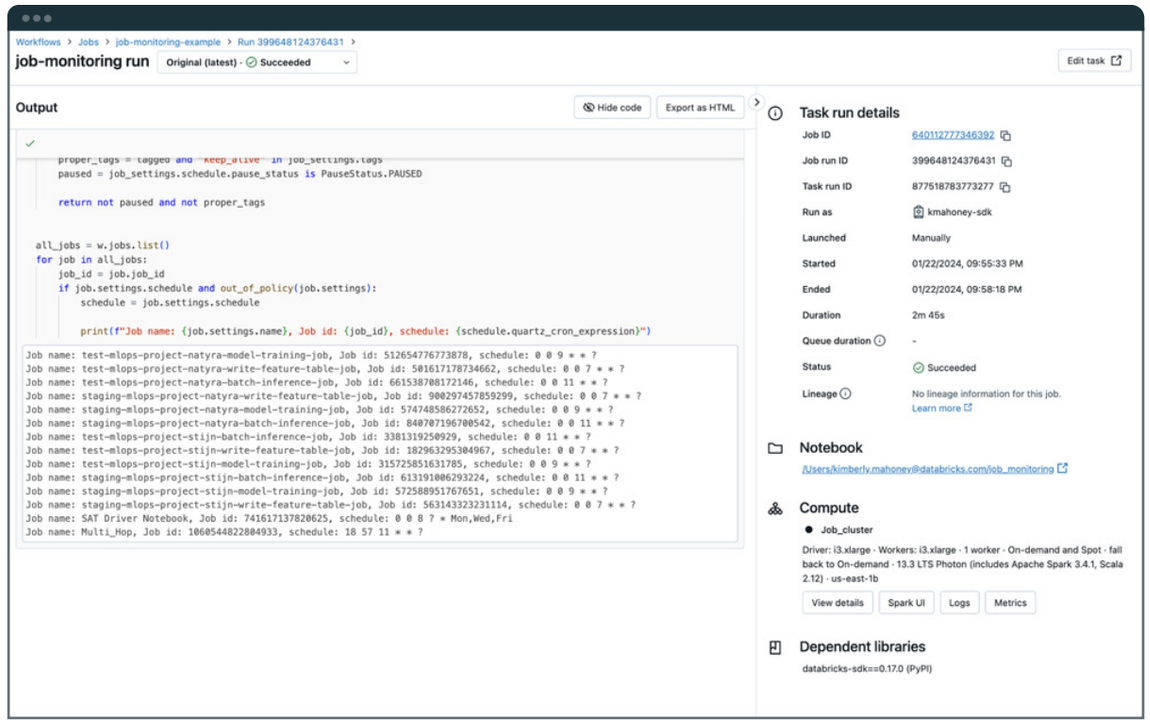
Putting it all together:

I broke out the policy logic into its own function for unit testing, added some logging and expanded the example to check for any jobs tagged as an exception to our policy. Now we have:

- *Logging out of policy jobs*

```
import logging

from databricks.sdk import WorkspaceClient
from databricks.sdk.service.jobs import CronSchedule, JobSettings, PauseStatus

# Initialize WorkspaceClient
w = WorkspaceClient()


def update_new_settings(job_id, quarts_cron_expression, timezone_id):
    """Update out of policy job schedules to be paused"""
    new_schedule = CronSchedule(
        quartz_cron_expression=quarts_cron_expression,
        timezone_id=timezone_id,
        pause_status=PauseStatus.PAUSED,
    )
    new_settings = JobSettings(schedule=new_schedule)

    logging.info(f"Job id: {job_id}, new_settings: {new_settings}")
    w.jobs.update(job_id, new_settings=new_settings)



def out_of_policy(job_settings: JobSettings):
    """Check if a job is out of policy.
    If it unpaused and has a schedule and is not tagged as keep_alive
    Return true if out of policy, false if in policy
    """

    tagged = bool(job_settings.tags)
    proper_tags = tagged and "keep_alive" in job_settings.tags
    paused = job_settings.schedule.pause_status is PauseStatus.PAUSED

    return not paused and not proper_tags

all_jobs = w.jobs.list()
for job in all_jobs:
    job_id = job.job_id
    if job.settings.schedule and out_of_policy(job.settings):
        schedule = job.settings.schedule

        logging.info(
            f"Job name: {job.settings.name}, Job id: {job_id}, creator: {job.creator_
user_name}, schedule: {schedule}"
        )
        ....
```

Now we have not only a working example but also a great foundation for building out a generalized job monitoring tool. We're successfully connecting to our workspace, listing all the jobs and analyzing their settings, and, when we're ready, we can simply call our `update_new_settings function` to apply the new paused schedule. It's fairly straightforward to expand this to meet other requirements you may want to set for a workspace — for example, swap job owners to service principles, add tags, edit notifications or audit job permissions. See the example in the GitHub repository.



SCHEDULING A JOB ON DATABRICKS

You can run your script anywhere, but you may want to schedule scripts that use the SDK to run as a Databricks Workflow or job on a small single-node cluster. When running a Python notebook interactively or via automated workflow, you can take advantage of default Databricks Notebook authentication. If you're working with the Databricks WorkspaceClient and your cluster meets the requirements listed in the docs, you can initialize your WorkspaceClient without needing to specify any other configuration options or environment variables — it works automatically out of the box.

CONCLUSION

In conclusion, the Databricks SDKs offer limitless potential for a variety of applications. We saw how the Databricks SDK for Python can be used to automate a simple yet crucial maintenance task and also saw an example of an OSS project that uses the Python SDK to integrate with the Databricks Platform. Regardless of the application you want to build, the SDKs streamline development for the Databricks Platform and allow you to focus on your particular use case. The key to quickly mastering a new SDK such as the Databricks Python SDK is setting up a proper development environment. Developing in an IDE allows you to take advantage of features such as a debugger, parameter info and code completion, so you can quickly navigate and familiarize yourself with the codebase. Visual Studio Code is a great choice for this as it provides the above capabilities and you can utilize the VSCode extension for Databricks to benefit from unified authentication.

Any feedback is greatly appreciated and welcome. Please raise any issues in the Python SDK GitHub repository. Happy developing!

ADDITIONAL RESOURCES

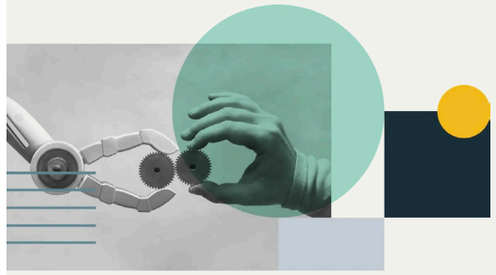
- Databricks SDK for Python Documentation
- DAIS Presentation: Unlocking the Power of Databricks SDKs
- How to install Python libraries in your local development environment: How to Create and Use Virtual Environments in Python With Poetry
- Installing the Databricks Extension for Visual Studio Code

03

Ready-to-Use Notebooks and Datasets

databricks

This section includes several Solution Accelerators — free, ready-to-use examples of data solutions from different industries ranging from retail to manufacturing and healthcare. Each of the following scenarios includes notebooks with code and step-by-step instructions to help you get started. Get hands-on experience with the Databricks Data Intelligence Platform by trying the following for yourself:



Digital Twins

Leverage digital twins — virtual representations of devices and objects — to optimize operations and gain insights

Explore the Solution

Overall Equipment Effectiveness

Ingest equipment sensor data for metric generation and data-driven decision-making

Explore the Solution

Recommendation Engines for Personalization

Improve customers' user experience and conversion with personalized recommendations

Explore the Solution

Real-Time Point-of-Sale Analytics

Calculate current inventories for various products across multiple store locations with Delta Live Tables

Explore the Solution

Understanding Price Transparency Data

Efficiently ingest large healthcare datasets to create price transparency for better understanding of healthcare costs

Explore the Solution

Additional Solution Accelerators with ready-to-use notebooks can be found here:

Databricks Solution Accelerators

04 | Case Studies

databricks

Cox Automotive — changing the way the world buys, sells and uses vehicles

"We use Databricks Workflows as our default orchestration tool to perform ETL and enable automation for about 300 jobs, of which approximately 120 are scheduled to run regularly."

— Robert Hamlet, Lead Data Engineer, Enterprise Data Services, Cox Automotive

Cox Automotive Europe is part of Cox Automotive, the world's largest automotive service organization, and is on a mission to transform the way the world buys, sells, owns and uses vehicles. They work in partnership with automotive manufacturers, fleets and retailers to improve performance and profitability throughout the vehicle lifecycle. Their businesses are organized around their customers' core needs across vehicle solutions, remarketing, funding, retail and mobility. Their brands in Europe include Manheim, Dealer Auction, NextGear Capital, Modix and Codeweavers.

Cox's enterprise data services team recently built a platform to consolidate the company's data and enable their data scientists to create new data-driven products and services more quickly and easily. To enable their small engineering team to unify data and analytics on one platform while enabling orchestration and governance, the enterprise data services team turned to the Databricks Data Intelligence Platform, Workflows, Unity Catalog and Delta Sharing.

INDUSTRY
Automotive

SOLUTION
Data-Driven ESG, Customer Entity Resolution, Demand Forecasting, Product Matching

PLATFORM
Workflows, Unity Catalog, Delta Sharing, ETL

CLOUD
Azure

databricks

EASY ORCHESTRATION AND OBSERVABILITY IMPROVE ABILITY TO DELIVER VALUE

Cox Automotive's enterprise data services team maintains a data platform that primarily serves internal customers spanning across business units, though they also maintain a few data feeds to third parties. The enterprise data services team collects data from multiple internal sources and business units. "We use Databricks Workflows as our default orchestration tool to perform ETL and enable automation for about 300 jobs, of which approximately 120 are scheduled to run regularly," says Robert Hamlet, Lead Data Engineer, Enterprise Data Services, at Cox Automotive.

Jobs may be conducted weekly, daily or hourly. The amount of data processed in production pipelines today is approximately 720GB per day. Scheduled jobs pull from different areas both within and outside of the company. Hamlet uses Databricks Workflows to deliver data to the data science team, to the in-house data reporting team through Tableau, or directly into Power BI. "Databricks Workflows has a great user interface that allows you to quickly schedule any type of workflow, be it a notebook or JAR," says Hamlet. "Parametrization has been especially useful. It gives us clues as to how we can move jobs across environments. Workflows has all the features you would want from an orchestrator."

Hamlet also likes that Workflows provides observability into every workflow run and failure notifications so they can get ahead of issues quickly and troubleshoot before the data science team is impacted. "We use the job notifications feature to send failure notifications to a webhook, which is linked to our Microsoft Teams account," he says. "If we receive an alert, we go into Databricks to see what's going on. It's very useful to be able to peel into the run logs and see what errors occurred. And the Repair Run feature is nice to remove blemishes from your perfect history."

UNITY CATALOG AND DELTA SHARING IMPROVE DATA ACCESS ACROSS TEAMS

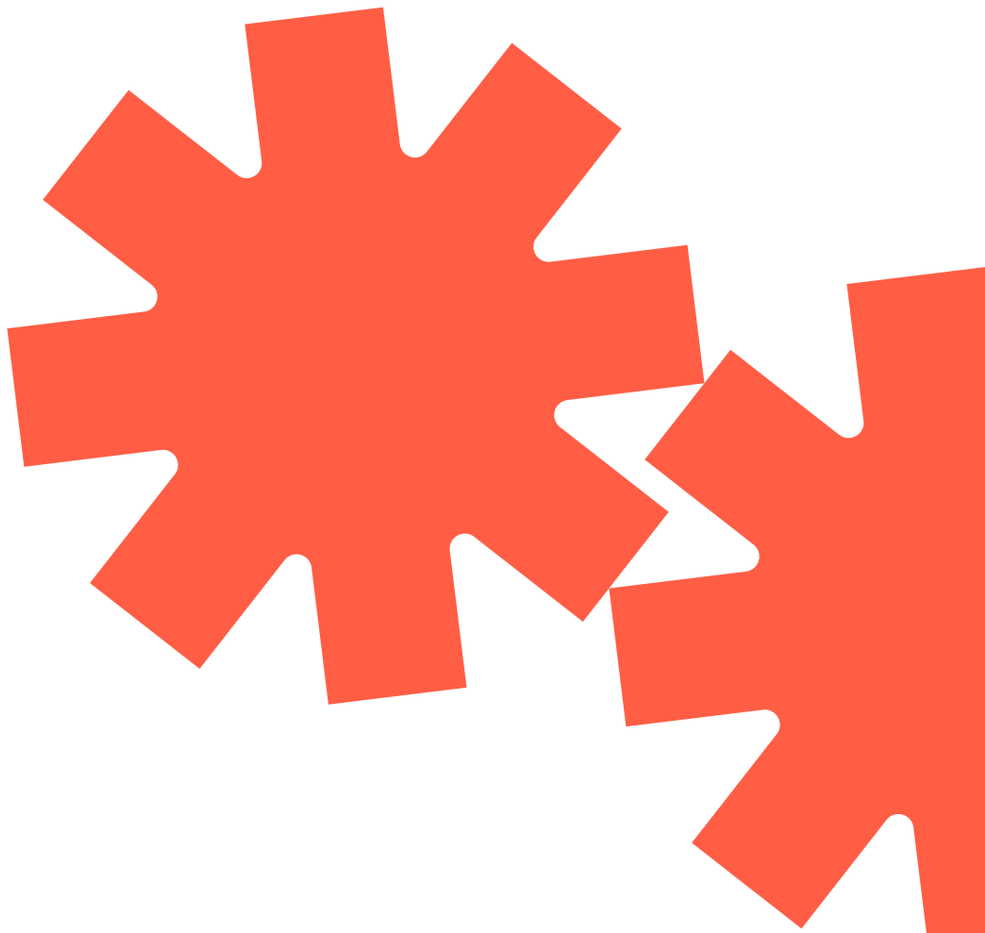
Hamlet's team recently began using Unity Catalog to manage data access, improving their existing method, which lacked granularity and was difficult to manage. "With our new workspace, we're trying to use more DevOps principles, infrastructure-as-code and groups wherever possible," he says. "I want to easily manage access to a wide range of data to multiple different groups and entities, and I want it to be as simple as possible for my team to do so. Unity Catalog is the answer to that."

The enterprise data services team also uses Delta Sharing, which natively integrates with Unity Catalog and allows Cox to centrally manage and audit shared data outside the enterprise data services team while ensuring robust security and governance. "Delta Sharing makes it easy to securely share data with business units and subsidiaries without copying or replicating it," says Hamlet. "It enables us to share data without the recipient having an identity in our workspace."

LOOKING AHEAD: INCORPORATING ADDITIONAL DATA INTELLIGENCE PLATFORM CAPABILITIES

Going forward, Hamlet plans to use Delta Live Tables (DLT) to make it easy to build and manage batch and streaming data pipelines that deliver data on the Databricks Data Intelligence Platform. DLT will help data engineering teams simplify ETL development and management. Eventually, Hamlet may also use Delta Sharing to easily share data securely with external suppliers and partners while meeting security and compliance needs. “DLT provides us an opportunity to make it simpler for our team. Scheduling Delta Live Tables will be another place we’ll use Workflows,” he says.

Hamlet is also looking forward to using the data lineage capabilities within Unity Catalog to provide his team with an end-to-end view of how data flows in the lakehouse for data compliance requirements and impact analysis of data changes. “That’s a feature I'm excited about,” Hamlet says. “Eventually, I hope we get to a point where we have all our data in the lakehouse, and we get to make better use of the tight integrations with things like data lineage and advanced permissions management.”



Block — building a world-class data platform

Block standardizes on Delta Live Tables to expand secure economic access for millions

90%
Improvement in development velocity

150
Pipelines being onboarded in addition to the 10 running daily

INDUSTRY
Financial Services

PLATFORM
Delta Live Tables, Data Streaming, Machine Learning, ETL

CLOUD
AWS

Block is a global technology company that champions accessible financial services and prioritizes economic empowerment. Its subsidiaries, including Square, Cash App and TIDAL, are committed to expanding economic access. By utilizing artificial intelligence (AI) and machine learning (ML), Block proactively identifies and prevents fraud, ensuring secure customer transactions in real time. In addition, Block enhances user experiences by delivering personalized recommendations and using identity resolution to gain a comprehensive understanding of customer activities across its diverse services. Internally, Block optimizes operations through automation and predictive analytics, driving efficiency in financial service delivery. Block uses the Data Intelligence Platform to bolster its capabilities, consolidating and streamlining its data, AI and analytics workloads. This strategic move positions Block for the forthcoming automation-driven innovation shift and solidifies its position as a pioneer in AI-driven financial services.

ENABLING CHANGE DATA CAPTURE FOR STREAMING DATA EVENTS ON DELTA LAKE

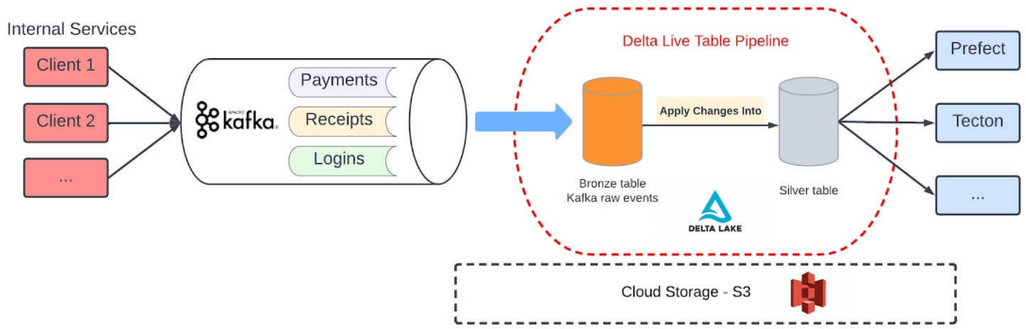
Block's Data Foundations team is dedicated to helping its internal customers aggregate, orchestrate and publish data at scale across the company's distributed system to support business use cases such as fraud detection, payment risk evaluation and real-time loan decisions. The team needs access to high-quality, low-latency data to enable fast, data-driven decisions and reactions.

Block had been consolidating on Kafka for data ingestion and Delta Lake for data storage. More recently, the company sought to make real-time streaming data available in Delta Lake as Silver (cleansed and conformed) data for analytics and machine learning. It also wanted to support event updates and simple data transformations and enable data quality checks to ensure higher-quality data. To accomplish this, Block considered a few alternatives, including the Confluent-managed Databricks Delta Lake Sink connector, a fully managed solution with low latency. However, that solution did not offer change data capture support and had limited transformation and data quality check support. The team also considered building their own solution with Spark Structured Streaming, which also provided low latency and strong data transformation capabilities. But that solution required the team to maintain significant code to define task workflows, change data and capture logic. They'd also have to implement their own data quality checks and maintenance jobs.

LEVERAGING THE LAKEHOUSE TO SYNC KAFKA STREAMS TO DELTA TABLES IN REAL TIME

Rather than redeveloping its data pipelines and applications on new, complex, proprietary and disjointed technology stacks, Block turned to the Data Intelligence Platform and Delta Live Tables (DLT) for change data capture and to enable the development of end-to-end, scalable streaming pipelines and applications. DLT pipelines simply orchestrate the way data flows between Delta tables for ETL jobs, requiring only a few lines of declarative code. It automates much of the operational complexity associated with running ETL pipelines and, as such, comes with preselected smart defaults yet is also tunable, enabling the team to optimize and debug easily. "DLT offers declarative syntax to define a pipeline, and we believed it could greatly improve our development velocity," says Yue Zhang, Staff Software Engineer for the Data Foundations team at Block. "It's also a managed solution, so it manages the maintenance tasks for us, it has data quality support, and it has advanced, efficient autoscaling and Unity Catalog integration."

Today, Block's Data Foundations team ingests events from internal services in Kafka topics. A DLT pipeline consumes those events into a Bronze (raw data) table in real time, and they use the DLT API to apply changes and merge data into a higher-quality Silver table. The Silver table can then be used by other DLT pipelines for model training, to schedule model orchestration, or to define features for a features store. "It's very straightforward to implement and build DLT pipelines," says Zhang.



The Block Data Foundations team's streaming data architecture with Delta Live Tables pipelines

Using the Python API to define a pipeline requires three steps: a row table, a Silver table and a merge process. The first step is to define the row table. Block consumes events from Kafka, performs some simple transformations, and establishes the data quality check and its rule. The goal is to ensure all events have a valid event ID.

The next step is to define the Silver table or target table, its storage location and how it is partitioned. With those tables defined, the team then determines the merge logic. Using the DLT API, they simply select APPLY CHANGES INTO. If two units have the same event ID, DLT will choose the one with the latest ingest timestamp. "That's all the code you need to write," says Zhang.

Finally, the team defines basic configuration settings from the DLT UI, such as characterizing clusters and whether the pipeline will run in continuous or triggered modes.

Following their initial DLT proof of concept, Zhang and his team implemented CI/CD to make DLT pipelines more accessible to internal Block teams. Different teams can now manage pipeline implementations and settings in their own repos, and, once they merge, simply use the Databricks pipelines API to create, update and delete those pipelines in the CI/CD process.

BOOSTING DEVELOPMENT VELOCITY WITH DLT

Implementing DLT has been a game-changer for Block, enabling it to boost development velocity. "With the adoption of Delta Live Tables, the time required to define and develop a streaming pipeline has gone from days to hours," says Zhang.

Meanwhile, managed maintenance tasks have resulted in better query performance, improved data quality has boosted customer trust, and more efficient autoscaling has improved cost efficiency. Access to fresh data means Block data analysts get more timely signals for analytics and decision-making, while Unity Catalog integration means they can better streamline and automate data governance processes. "Before we had support for Unity Catalog, we had to use a separate process and pipeline to stream data into S3 storage and a different process to create a data table out of it," says Zhang. "With Unity Catalog integration, we can streamline, create and manage tables from the DLT pipeline directly."

Block is currently running approximately 10 DLT pipelines daily, with about two terabytes of data flowing through them, and has another 150 pipelines to onboard. "Going forward, we're excited to see the bigger impacts DLT can offer us," adds Zhang.

Trek — global bicycle leader accelerates retail analytics

80%-90%
Acceleration in runtime of retail analytics solution globally

3X
Increase in daily data refreshes on Databricks

1 Week
Reduction in ERP data replication, which now happens in near real time

"How do you scale up analytics without blowing a hole in your technology budget? For us, the clear answer was to run all our workloads on Databricks Data Intelligence Platform and replicate our data in near real-time with Qlik."

— Garrett Baltzer, Software Architect, Data Engineering, Trek Bicycle

INDUSTRY
Retail and Consumer Goods

SOLUTION
Real-Time Point-of-Sale Analytics

PLATFORM
Delta Lake, Databricks SQL, Delta Live Tables, Data Streaming

PARTNER
Qlik

CLOUD
Azure

Trek Bicycle started in a small Wisconsin barn in 1976, but their founders always saw something bigger. Decades later, the company is on a mission to make the world a better place to live and ride. Trek only builds products they love and provides incredible hospitality to customers as they aim to change the world for the better by getting more people on bikes. Frustrated by the rising costs and slow performance of their data warehouse, Trek migrated to Databricks Data Intelligence Platform. The company now uses Qlik to replicate their ERP data to Databricks in near real-time and stores data in Delta Lake tables. With Databricks and Qlik, Trek has dramatically accelerated their retail analytics to provide a better experience for their customers with a unified view of the global business to their data consumers, including business and IT users.

databricks

SLOW DATA PROCESSING HINDERS RETAIL ANALYTICS

As Trek Bicycle works to make the world better by encouraging more people to ride bikes, the company keeps a close eye on what's happening in their hundreds of retail stores. But until recently, running analytics on their retail data proved challenging because Trek relied on a data warehouse that couldn't scale cost-effectively.

"The more stores we added, the more information we added to our processes and solutions," explained Garrett Baltzer, Software Architect, Data Engineering, at Trek Bicycle. "Although our data warehouse did scale to support greater data volumes, our processing costs were skyrocketing, and processes were taking far too long. Some of our solutions were taking over 30 hours to produce analytics, which is unacceptable from a business perspective."

Adding to the challenge, Trek's data infrastructure hindered the company's efforts to achieve a global view of their business performance. Slow processing speeds meant Trek could only process data once per day for one region at a time.

"We were processing retail data separately for our North American, European and Asia-Pacific stores, which meant everyone downstream had to wait for actionable insights for different use cases," recalled Advait Raje, Team Lead, Data Engineering, at Trek Bicycle. "We soon made it a priority to migrate to a unified data platform that would produce analytics more quickly and at a lower cost."

DELTA LAKE UNIFIES RETAIL DATA FROM AROUND THE GLOBE

Seeking to modernize their data infrastructure to speed up data processing and unify all their data from global sources, Trek started migrating to the Databricks Data Intelligence Platform in 2019. The company's processing speeds immediately increased. Qlik's integration with the Databricks Data Intelligence Platform helps feed Trek's lakehouse. This replication allows Trek to build a wide range of valuable data products for their sales and customer service teams.

"Qlik enabled us to move relevant ERP data into Databricks where we don't have to worry about scaling vertically because it automatically scales parallel. Since 70 to 80% of our operational data comes from our ERP system, Qlik has made it possible to get far more out of our ERP data without increasing our costs," Baltzer explained.

Trek is now running all their retail analytics workloads in the Databricks Data Intelligence Platform. Today, Trek uses the Databricks Data Intelligence Platform to collect point-of-sale data from nearly 450 stores around the globe. All computation happens on top of Trek's lakehouse. The company runs a semantic layer on top of this lakehouse to power everything from strategic high-level reporting for C-level executives to daily sales and operations reports for individual store employees.

"Databricks Data Intelligence Platform has been a game-changer for Trek," said Raje. "With Qlik Cloud Data Integration on Databricks, it became possible to replicate relevant ERP data to our Databricks in real time, which made it far more accessible for downstream retail analytics. Suddenly, all our data from multiple repositories was available in one place, enabling us to reduce costs and deliver on business needs much more quickly."

Trek's BI and data analysts leverage Databricks SQL, their serverless data warehouse, for ad hoc analysis to answer business questions much more quickly. Internal customers can leverage Power BI connecting directly to Databricks to consume retail analytics data from Gold tables. This ease of analysis helps the company monitor and enhance their Net Promoter Scores. Trek uses Structured Streaming and Auto Loader functionality within Delta Live Tables to transform the data from Bronze to Silver or Gold, according to the medallion architecture.

"Delta Live Tables have greatly accelerated our development velocity," Raje reported. "In the past, we had to use complicated ETL processes to take data from raw to parsed. Today, we just have one simple notebook that does it, and then we use Delta Live Tables to transform the data to Silver or Gold as needed."

DATA INTELLIGENCE PLATFORM ACCELERATES ANALYTICS BY 80% TO 90%

By moving their data processing to the Databricks Data Intelligence Platform and integrating data with Qlik, Trek has dramatically increased the speed of their processing and overall availability of data. Prior to implementing Qlik, they had a custom program that, once a week, on a Sunday, replicated Trek's ERP data from on-premises servers to a data lake using bulk copies. Using Qlik, Trek now replicates relevant data from their ERP system as Delta tables directly in their lakehouse.

"We used to work with stale ERP data all week because replication only happened on Sundays," Raje remarked. "Now we have a nearly up-to-the-minute view of what's going on in our business. That's because Qlik lets us keep replicating through the day, streaming data from ERP into our lakehouse."

Trek's retail analytics solution used to take 48+ hours to produce meaningful results. Today, Trek runs the solution on the Databricks Data Intelligence Platform to get results in six to eight hours — an 80 to 90% improvement, thus allowing daily runs. A complementary retail analytics solution went from 12–14 hours down to under 4–5 hours, thereby enabling the lakehouse to refresh three times per day, compared to only once a day previously.

"Before Databricks, we had to run our retail analytics once a day on North American time, which meant our other regions got their data late," said Raje. "Now, we refresh the lakehouse three times per day, one for each region, and stakeholders receive fresh data in time to drive their decisions. Based on the results we've achieved in the lakehouse, we're taking a Databricks-first approach to all our new projects. We're even migrating many of our on-premises BI solutions to Databricks because we're all-in on the lakehouse."

"Databricks Data Intelligence Platform, along with data replication to Databricks using Qlik, aligns perfectly with our broader cloud-first strategy," said Steve Novoselac, Vice President, IT and Digital, at Trek Bicycle. "This demonstrates confidence in the adoption of this platform at Trek."

Coastal Community Bank — mastering the modern data platform for exponential growth

< 10
Minutes to securely share large datasets across organizations

99%
Decrease in processing duration (2+ days to 30 min.)

12X
Faster partner onboarding by eliminating sharing complexity

> "We've done two years' worth of work here in nine months. Databricks enables access to a single source of truth and our ability to process high volumes of transactions that gives us confidence we can drive our growth as a community bank and a leading banking-as-a-service provider."

— Curt Queyrouze, President, Coastal Community Bank

Many banks continue to rely on decades-old, mainframe-based platforms to support their back-end operations. But banks that are modernizing their IT infrastructures and integrating the cloud to share data securely and seamlessly are finding they can form an increasingly interconnected financial services landscape. This has created opportunities for community banks, fintechs and brands to collaborate and offer customers more comprehensive and personalized services. Coastal Community Bank is headquartered in Everett, Washington, far from some of the world's largest financial centers. The bank's CCBX division offers banking as a service (BaaS) to financial technology companies and broker-dealers. To provide personalized financial products, better risk oversight, reporting and compliance, Coastal turned to the Databricks Data Intelligence Platform and Delta Sharing, an open protocol for secure data sharing, to enable them to share data with their partners while ensuring compliance in a highly regulated industry.

INDUSTRY
Financial Services

SOLUTION
Financial Crimes Compliance, Customer Profile Scoring, Financial Reconciliation, Credit Risk Reporting, Synthesizing Multiple Data Sources, Data Sharing and Collaboration

PLATFORM
Delta Lake, ETL, Delta Sharing, Data Streaming, Databricks SQL

CLOUD
Azure

databricks

LEVERAGING TECH AND INNOVATION TO FUTURE-PROOF A COMMUNITY BANK

Coastal Community Bank was founded in 1997 as a traditional brick-and-mortar bank. Over the years, they grew to 14 full-service branches in Washington state offering lending and deposit products to approximately 40,000 customers. In 2018, the bank's leadership broadened their vision and long-term growth objectives, including how to scale and serve customers outside their traditional physical footprint. Coastal leaders took an innovative step and launched a plan to offer BaaS through CCBX, enabling a broad network of virtual partners and allowing the bank to scale much faster and further than they could via their physical branches alone.

Coastal hired Barb MacLean, Senior Vice President and Head of Technology Operations and Implementation, to build the technical foundation required to help support the continued growth of the bank. "Most small community banks have little technology capability of their own and often outsource tech capabilities to a core banking vendor," says MacLean. "We knew that story had to be completely different for us to continue to be an attractive banking-as-a-service partner to outside organizations."

To accomplish their objectives, Coastal would be required to receive and send vast amounts of data in near real-time with their partners, third parties and the variety of systems used across that ecosystem. This proved to be a challenge as most banks and providers still relied on legacy technologies and antiquated processes like once-a-day batch processing. To scale their BaaS offering, Coastal needed a better way to manage and share data. They also required a solution that could scale while ensuring that the highest levels of security, privacy and strict compliance requirements were met. "The list of things we have to do to prove that we can safely and soundly operate as a regulated financial institution is ever-increasing," says MacLean. "As we added more customers and therefore more customer information, we needed to scale safely through automation."

Coastal also needed to accomplish all this with their existing small team. "As a community bank, we can't compete on a people basis, so we have to have technology tools in place that teams can learn easily and deploy quickly," adds MacLean.

TACKLING A COMPLEX DATA ENVIRONMENT WITH DELTA SHARING

With the goal of having a more collaborative approach to community banking and banking as a service, Coastal began their BaaS journey in January 2023 when they chose Cavallo Technologies to help them develop a modern, future-proof data platform to support their stringent customer data sharing and compliance requirements. This included tackling infrastructure challenges, such as data ingestion complexity, speed, data quality and scalability. "We wanted to use our small, nimble team to our advantage and find the right technology to help us move fast and do this right," says MacLean.

"We initially tested several vendors, however learned through those tests we needed a system that could scale for our needs going forward," says MacLean. Though very few members of the team had used Databricks before, Coastal decided to move from a previously known implementation pattern and a data lake–based platform to a lakehouse approach with Databricks. The lakehouse architecture addressed the pain points they experienced using a data lake–based platform, such as trying to sync batch and streaming data. The dynamic nature and changing environments of Coastal's partners required handling changes to data structure and content. The Databricks Data Intelligence Platform provided resiliency and tooling to deal with both data and schema drift cost-effectively at scale.

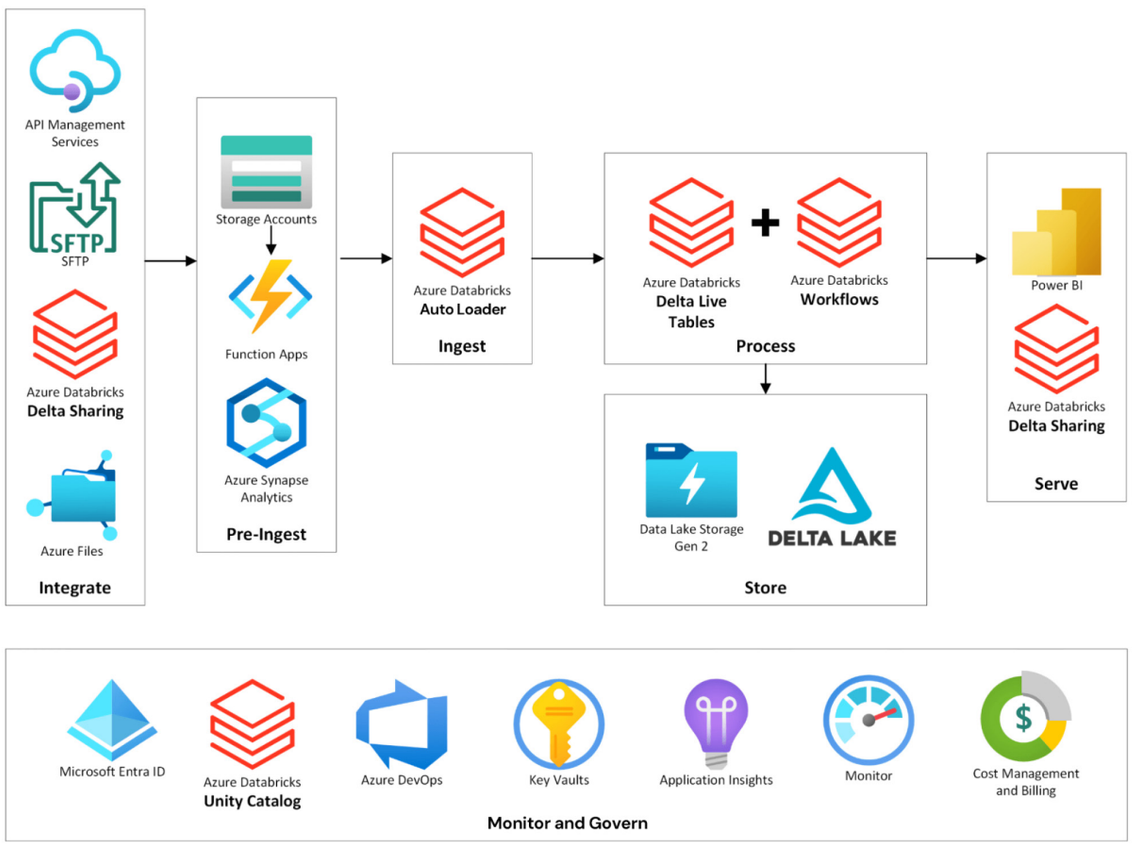
Coastal continued to evolve and extend their use of Databricks tools, including Auto Loader, Structured Streaming, Delta Live Tables, Unity Catalog and Databricks repos for CI/CD, as they created a robust software engineering practice for data at the bank. Applying software engineering principles to data can often be neglected or ignored by engineering teams, but Coastal knew that it was critical to managing the scale and complexity of the internal and external environment in which they were working. This included having segregated environments for development, testing and production, having technical leaders approve the promotion of code between environments, and include data privacy and security governance.

Coastal also liked that Databricks worked well with Azure out of the box. And because it offered a consolidated toolkit for data transformation and engineering, Databricks helped address any risk concerns. "When you have a highly complex technical environment with a myriad of tools, not only inside your own environment but in our partners' environments that we don't control, a consolidated toolkit reduces complexity and thereby reduces risk," says MacLean.

Initially, MacLean's team evaluated several cloud-native solutions, with the goal of moving away from a 24-hour batch world and into real-time data processing since any incident could have wider reverberations in a highly interconnected financial system. "We have all these places where data is moving in real time. What happens when someone else's system has an outage or goes down in the middle of the day? How do you understand customer and bank exposure as soon as it happens? How do we connect the batch world with the real-time world? We were trapped in a no-man's-land of legacy, batch-driven systems, and partners are too," explains MacLean.

"We wanted to be a part of a community of users, knowing that was the future, and wanted a vendor that was continually innovating," says MacLean. Similarly, MacLean's team evaluated the different platforms for ETL, BI, analytics and data science, including some already in use by the bank. "Engineers want to work with modern tools because it makes their lives easier … working within the century in which you live. We didn't want to Frankenstein things because of a wide toolset," says MacLean. "Reducing complexity in our environment is a key consideration, so using a single platform has a massive positive impact. Databricks is the hands-down winner in apples-to-apples comparisons to other tools like Snowflake and SAS in terms of performance, scalability, flexibility and cost."

MacLean explained that Databricks included everything, such as Auto Loader, repositories, monitoring and telemetry, and cost management. This enabled the bank to benefit from robust software engineering practices so they could scale to serving millions of customers, whether directly or via their partner network. MacLean explained, "We punch above our weight, and our team is extremely small relative to what we're doing, so we wanted to pick the tools that are applicable to any and all scenarios."



The Databricks Data Intelligence Platform has greatly simplified how Coastal and their vast ecosystem of financial service partners securely share data across data platforms, clouds or regions.

IMPROVING TIME TO VALUE AND GROWING THEIR PARTNER NETWORK

In the short time since Coastal launched CCBX, it has become the bank's primary customer acquisition and growth division, enabling them to grow BaaS program fee income by 32.3% year over year. Their use of Databricks has also helped them achieve unprecedented time to value. "We've done two years' worth of work here in nine months," says Curt Queyrouze, President at Coastal.

Almost immediately, Coastal saw exponential improvements in core business functions. "Activities within our risk and compliance team that we need to conduct every few months would take 48 hours to execute with legacy inputs," says MacLean. "Now we can run those in 30 minutes using near real-time data."

Despite managing myriad technology systems, Databricks helps Coastal remove barriers between teams, enabling them to share live data with each other safely and securely in a matter of minutes so the bank can continue to grow quickly through partner acquisition. "The financial services industry is still heavily reliant on legacy, batch-driven systems, and other data is moving in real time and needs to be understood in real time. How do we marry those up?" asks MacLean. "That was one of the fundamental reasons for choosing Databricks. We have not worked with any other tool or technology that allows us to do that well."

CCBX leverages the power and scale of a network of partners. Delta Sharing uses an open source approach to data sharing and enables users to share live data across platforms, clouds and regions with strong security and governance. Using Delta Sharing meant Coastal could manage data effectively even when working with partners and third parties using inflexible legacy technology systems. "The data we were ingesting is difficult to deal with," says MacLean. "How do we harness incoming data from about 20 partners with technology environments that we don't control? The data's never going to be clean. We decided to make dealing with that complexity our strength and take on that burden. That's where we saw the true power of Databricks' capabilities. We couldn't have done this without the tools their platform gives us."

Databricks also enabled Coastal to scale from 40,000 customers (consumers and small-medium businesses in the north Puget Sound region) to approximately 6 million customers served through their partner ecosystem and dramatically increase the speed at which they integrate data from those partners. In one notable case, Coastal was working with a new partner and faced the potential of having to load data on 80,000 customers manually. "We pointed Databricks at it and had 80,000 customers and the various data sources ingested, cleaned and prepared for our business teams to use in two days," says MacLean. "Previously, that would have taken one to two months at least. We could not have done that with any prior existing tool we tried."

With Delta Sharing on Databricks, Coastal now has a vastly simplified, faster and more secure platform for onboarding new partners and their data. "When we want to launch and grow a product with a partner, such as a point-of-sale consumer loan, the owner of the data would need to send massive datasets on tens of thousands of customers. Before, in the traditional data warehouse approach, this would typically take one to two months to ingest new data sources, as the schema of the sent data would need to be changed in order for our systems to read it. But now we point Databricks at it and it's just two days to value," shares MacLean.

While Coastal's data engineers and business users love this improvement in internal productivity, the larger transformation has been how Databricks has enabled Coastal's strategy to focus on building a rich partner network. They now have about 20 partners leveraging different aspects of Coastal's BaaS.

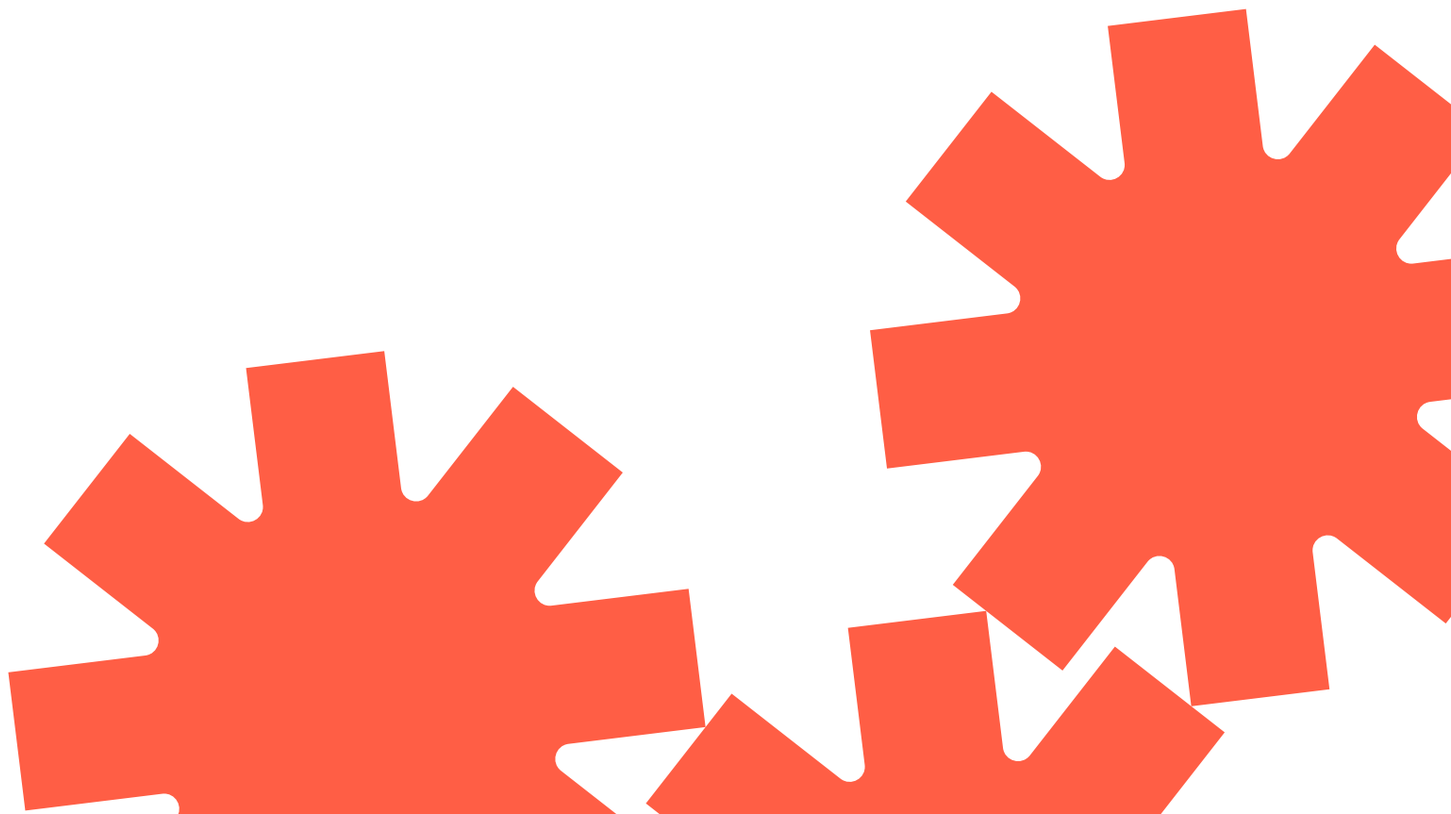
Recently, Coastal's CEO had an ask about a specific dataset. Based on experience from their previous data tools, they brought in a team of 10 data engineers to comb through the data, expecting this to be a multiday or even multi-week effort. But when they actually got into their Databricks Data Intelligence Platform, using data lineage on Unity Catalog, they were able to give a definitive answer that same afternoon. MacLean explains that this is not an anomaly. "Time and time again, we find that even for the most seemingly challenging questions, we can grab a data engineer with no context on the data, point them to a data pipeline and quickly get the answers we need."

The bank's use of Delta Sharing has also allowed Coastal to achieve success with One, an emerging fintech startup. One wanted to sunset its use of Google BigQuery, which Coastal was using to ingest One's data. The two organizations needed to work together to find a solution. Fortunately, One was also using Databricks. "We used Delta Sharing, and after we gave them a workspace ID, we had tables of data showing up in our Databricks workspace in under 10 minutes," says MacLean. (To read more about how Coastal is working with One, read the blog.) MacLean says Coastal is a leader in skills, technology and modern tools for fintech partners.

DATA AND AI FOR GOOD

With a strong data foundation set, MacLean has a larger vision for her team. "Technologies like generative AI open up self-serve capabilities to so many business groups. For example, as we explore how to reduce financial crimes, if you are taking a day to do an investigation, that doesn't scale to thousands of transactions that might need to be investigated," says MacLean. "How do we move beyond the minimum regulatory requirements on paper around something like anti-money laundering and truly reduce the impact of bad actors in the financial system?"

For MacLean this is about aligning her organization with Coastal's larger mission to use finance to do better for all people. Said MacLean, "Where are we doing good in terms of the application of technology and financial services? It's not just about optimizing the speed of transactions. We care about doing better on behalf of our fellow humans with the work that we do."



Powys Teaching Health Board — improving decision-making to save lives faster

< 1 year
To modernize data infrastructure

40%
Decrease in time to insight

65%
More productive with Databricks Assistant

"The adoption of Databricks has ensured that we can future-proof our data capabilities. It has transformed and modernized the way we work, and that has a direct impact on the quality of care delivered to our community."

— Jake Hammer, Chief Data Officer, Powys Teaching Health Board (PTHB)

INDUSTRY
Healthcare and Life Sciences

SOLUTION
Forward-Looking Intelligence

PLATFORM
Data Intelligence Platform, Unity Catalog

CLOUD
Azure

The inability to access complete and high-quality data can have a direct impact on a healthcare system's ability to deliver optimal patient outcomes. Powys Teaching Health Board (PTHB), serving the largest county in Wales, is responsible for planning and providing national health services for approximately a quarter of the country. However, roughly 50% of the data they need to help inform patient-centric decisions doesn't occur within Powys and is provided by neighboring organizations in varying formats, slowing their ability to connect data with the quality of patient care. Converting all this data — from patient activity (e.g., appointments) and workforce data (e.g., schedules) — to actionable insights is difficult when it comes in from so many disparate sources. PTHB needed to break down these silos and make it easier for nontechnical teams to access the data. With the Databricks Data Intelligence Platform, PTHB now has a unified view of all their various data streams, empowering healthcare systems to make better decisions that enhance patient care.

SILOS AND SYSTEM STRAIN HINDER DATA-DRIVEN INSIGHTS

The demand for PTHB's services has increased significantly over the years as they've dealt with evolving healthcare needs and population growth. As new patients enter the national healthcare system, so does the rise in data captured about the patient, hospital operations and more. With this rapid influx of data coming from various hospitals and healthcare systems around the country, PTHB's legacy system began to reach its performance and scalability limits, quickly developing data access and ingestion bottlenecks that not only wasted time, but directly impacted patient care. And as the diversity of data rose, their legacy system buckled under the load.

"Our data sat in so many places that it caused major frustrations. Our on-premises SQL warehouse couldn't cope with the scale of our growing data estate," explained Jake Hammer, Chief Data Officer at PTHB. "We needed to move away from manually copying data between places. Finding a platform that would allow us to take advantage of the cloud and was flexible enough to safeguard our data within a single view for all to easily access was critical."

How could PTHB employees make data-driven decisions if the data was hard to find and difficult to understand? Hammer realized that they needed to first modernize their data infrastructure in the cloud and migrate to a platform capable of unifying their data and making it readily available for downstream analytics use cases: from optimizing staff schedules to providing actionable insights for clinicians so they can provide timely and targeted care. Hammer's team estimated that it would take five to 10 years to modernize their tech stack in this way if they were to follow their own processes and tech stack. But they needed a solution now. Enter Databricks.

IMPROVING DATA DEMOCRATIZATION WITH A UNIFIED PLATFORM

PTHB chose the Databricks Data Intelligence Platform to house all new incoming data, from any source. This includes the data for a large number of low-code apps (e.g., Power Apps) so that Hammer's team can now work with data that was historically kept on paper — making it significantly easier for people to access and analyze the data at scale.

Data governance is also critical, but creating standard processes was difficult before transitioning to Databricks as their core platform. With Unity Catalog, PTHB has a model where all of their security and governance is done only once at the Databricks layer. "The level of auditing in Databricks gives us a high level of assurance. We need to provide different levels of access to many different individuals and systems," added Hammer. "Having a tool that enables us to confidently manage this complex security gives both ourselves and our stakeholders assurance. We can more easily and securely share data with partners."

Deriving actionable insights on data through numerous Power BI dashboards with ease is something PTHB could not do before. "Now HR has the data they need to improve operational efficiency while protecting the bottom line," said Hammer. "They can self-serve any necessary data, and they can see where there are gaps in rosters or inefficiencies in on-the-ground processes. Being able to access the right data at the right time means they can be smarter with rostering, resource management and scheduling."

FEDERATED LAKEHOUSE IMPROVES TEAM EFFICIENCY AND REAL-TIME DATA ENHANCES PATIENT CARE

With the Databricks Data Intelligence Platform, PTHB has taken their first step toward modernization by moving to the cloud in less than a year — a much quicker timeline than their 10-year estimate — and providing a federated lakehouse to unify all their data. Through the lakehouse, they are able to seamlessly connect to their on-premises SQL warehouse and remote BigQuery environment at NHS Wales to create a single view of their data estate. "With the Databricks Platform, and by leveraging features such as Lakehouse Federation to integrate remote data, PTHB data practitioners now work from a single source of truth to improve decision-making and patient outcomes," explained Hammer.

From an operational standpoint, the impact of a modern platform has been significant, with efficiencies skyrocketing to an estimated 40% time savings in building data pipelines for analytics. They also estimate spending 65% less time answering questions from business data users with the help of Databricks Assistant. This AI-powered tool accelerated training for PTHB, helping traditional SQL staff embrace new programming languages and empowering them to be more productive without overreliance on the data engineering team.

A modern stack and newfound efficiencies throughout the data workflow has fueled Hammer's ability to expand into more advanced use cases. "Data science was an area we simply hadn't thought about," explained Hammer. "Now we have the means to explore predictive use cases like forecasting feature utilization."

Most importantly, PTHB has a solution that's future-proof and can tackle any data challenge, which is critical given how they are seeing a rapidly growing environment of APIs, adoption of open standards and new sources of real-time data. "I can trust our platform is future-proof, and I'm probably the only health board to be able to say that in Wales at the moment," said Hammer. "Just like with the data science world, the prediction world was something we never thought was possible. But now we have the technology to do anything we put our minds and data to."

Tens of millions of production workloads run daily on Databricks

Easily ingest and transform batch and streaming data on the Databricks Data Intelligence Platform. Orchestrate reliable production workflows while Databricks automatically manages your infrastructure at scale. Increase the productivity of your teams with built-in data quality testing and support for software development best practices.

Try Databricks free **Get started with a free demo**

About Databricks

Databricks is the data and AI company. More than 10,000 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 and MLflow. To learn more, follow Databricks on LinkedIn, X and Facebook.

databricks