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.

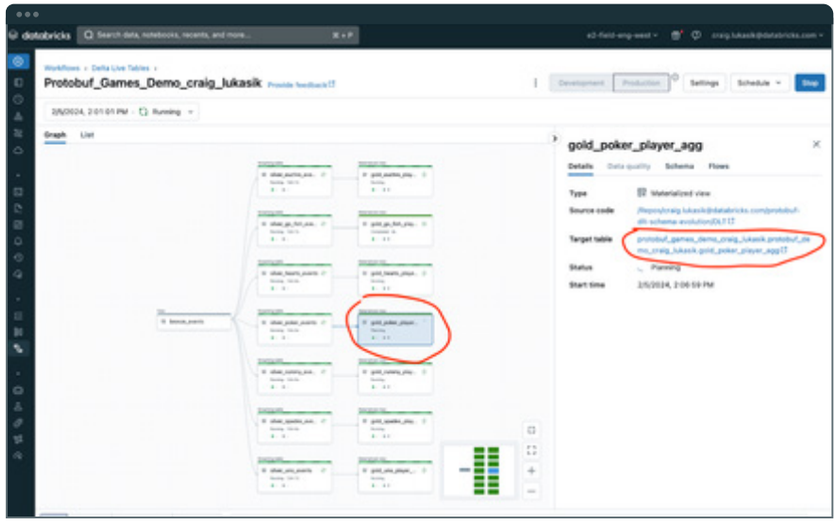
REVIEWING THE BENEFITS OF DLT

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

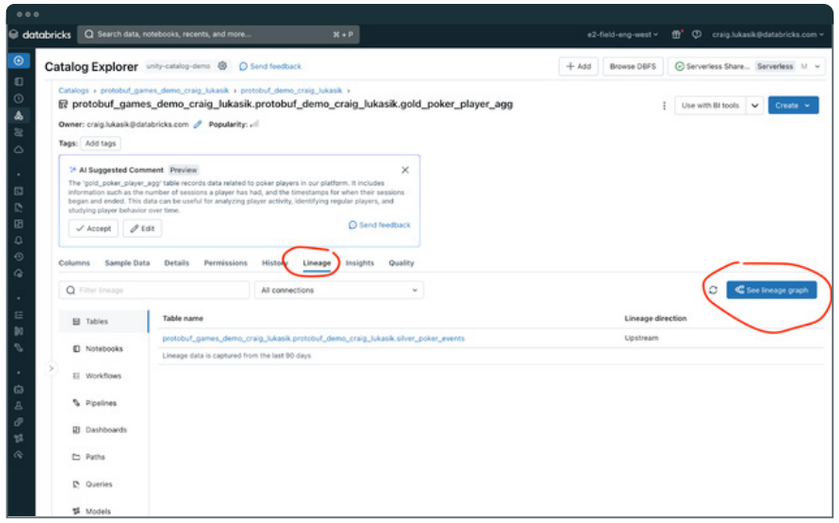
From the Delta Live Tables interface, we can navigate to the Catalog and view lineage.



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

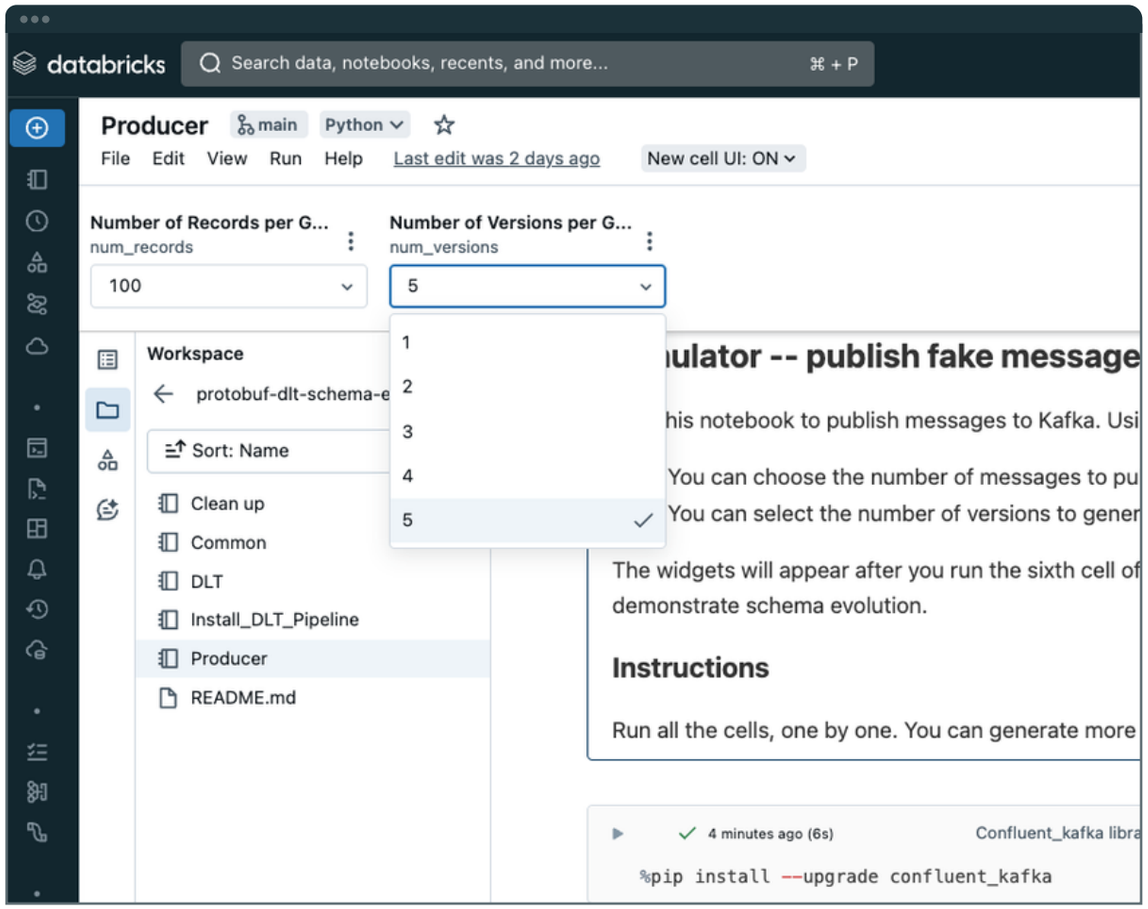
Click on the "Lineage" tab for the table. Then click on the "See lineage graph" link.

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



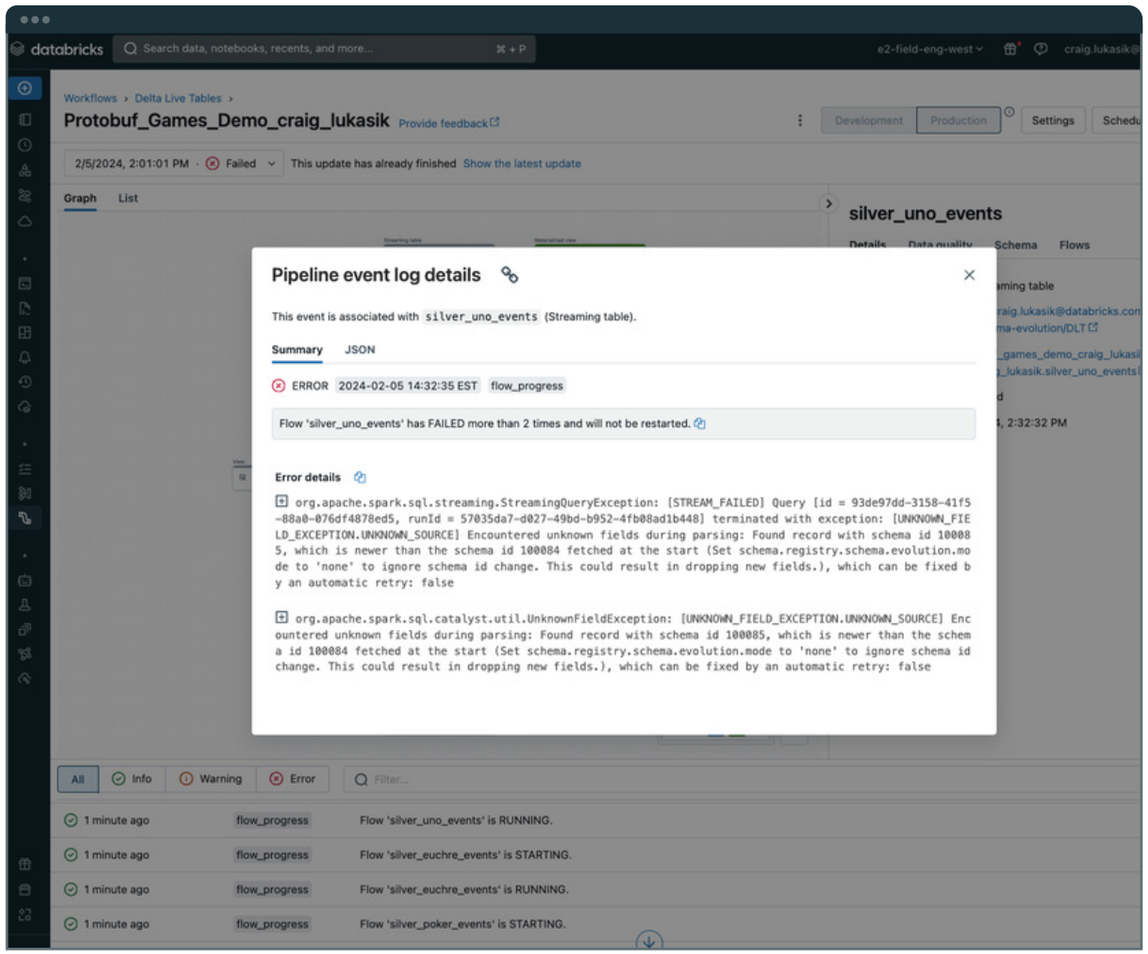
This lineage helps accelerate team velocity by making it easier to understand how assets in the workspace are related.

HANDS-OFF SCHEMA EVOLUTION

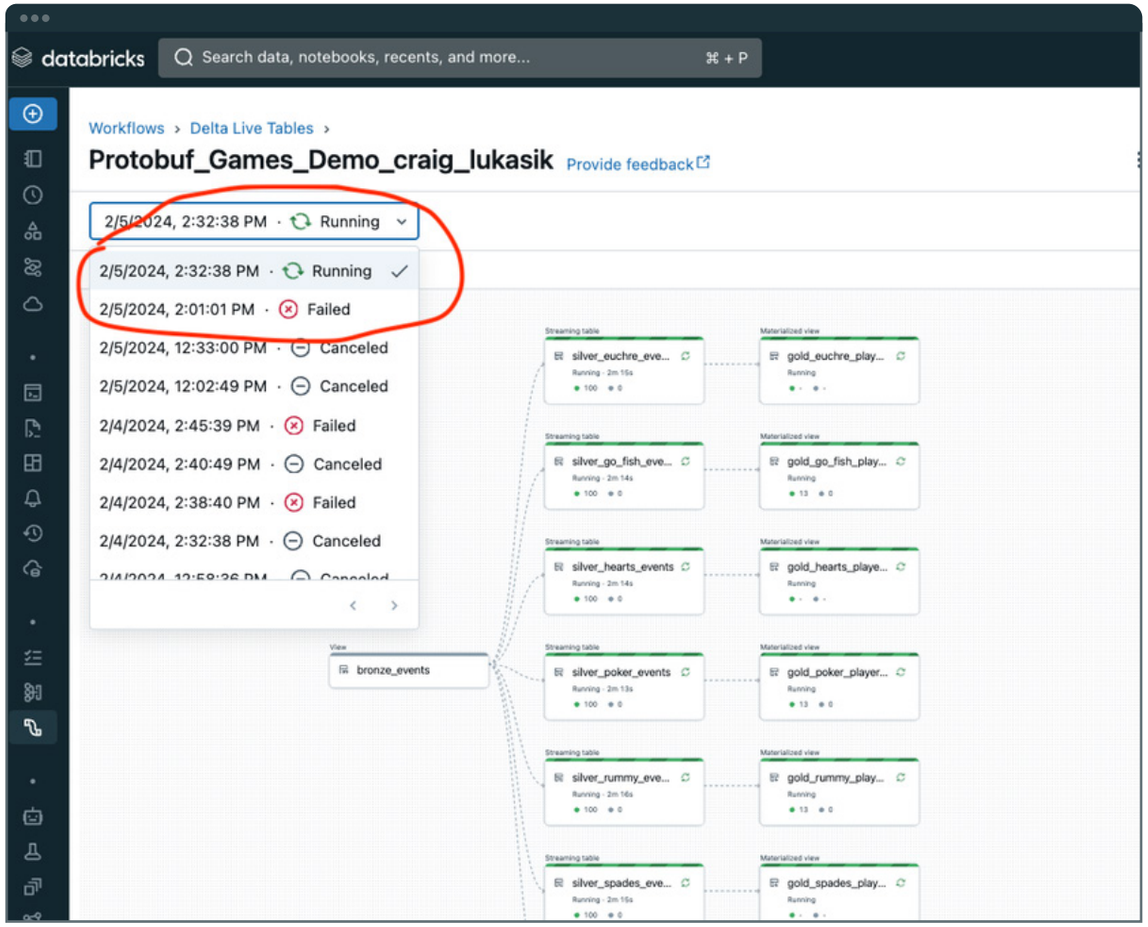


Delta Live Tables will detect this as the source stream's schema evolves, and the pipeline will restart. To simulate a schema evolution for this example, you would run the Producer notebook a subsequent time but with a larger value for num_versions, as shown on the left. This will generate new data where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

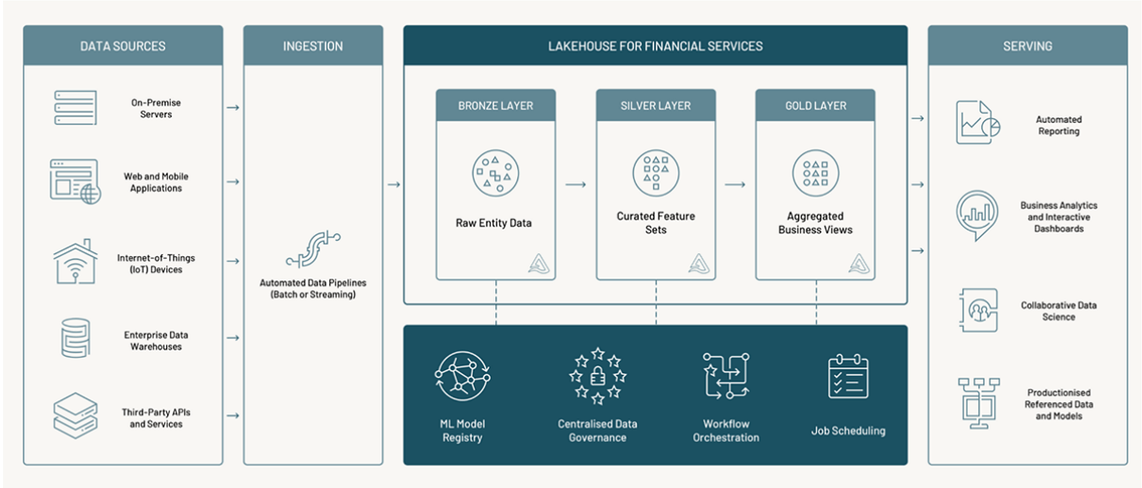


Figure 1: Reference architecture of the Lakehouse for Financial Services

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.

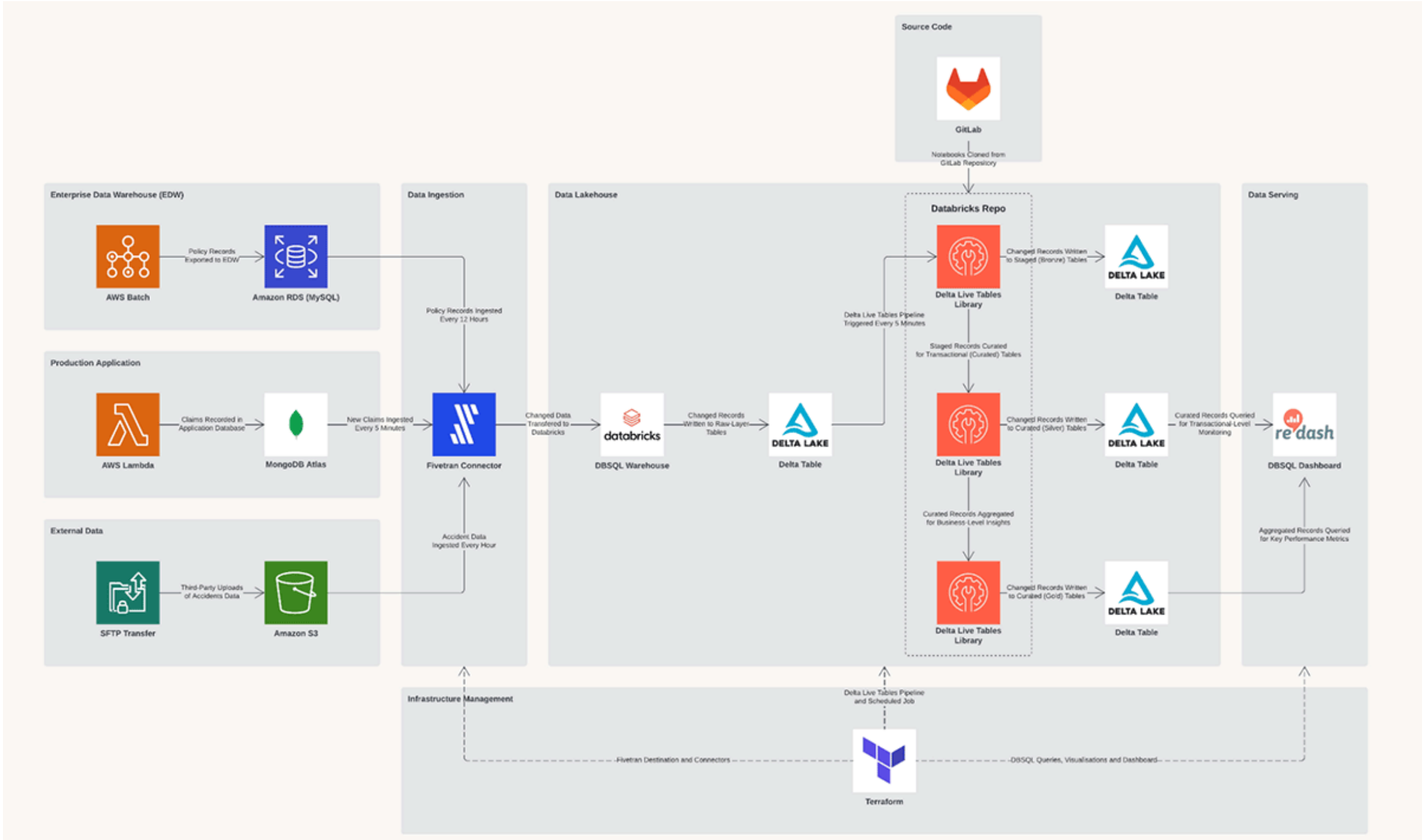


Figure 2: Technical architecture for a simple insurance claims workflow

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.

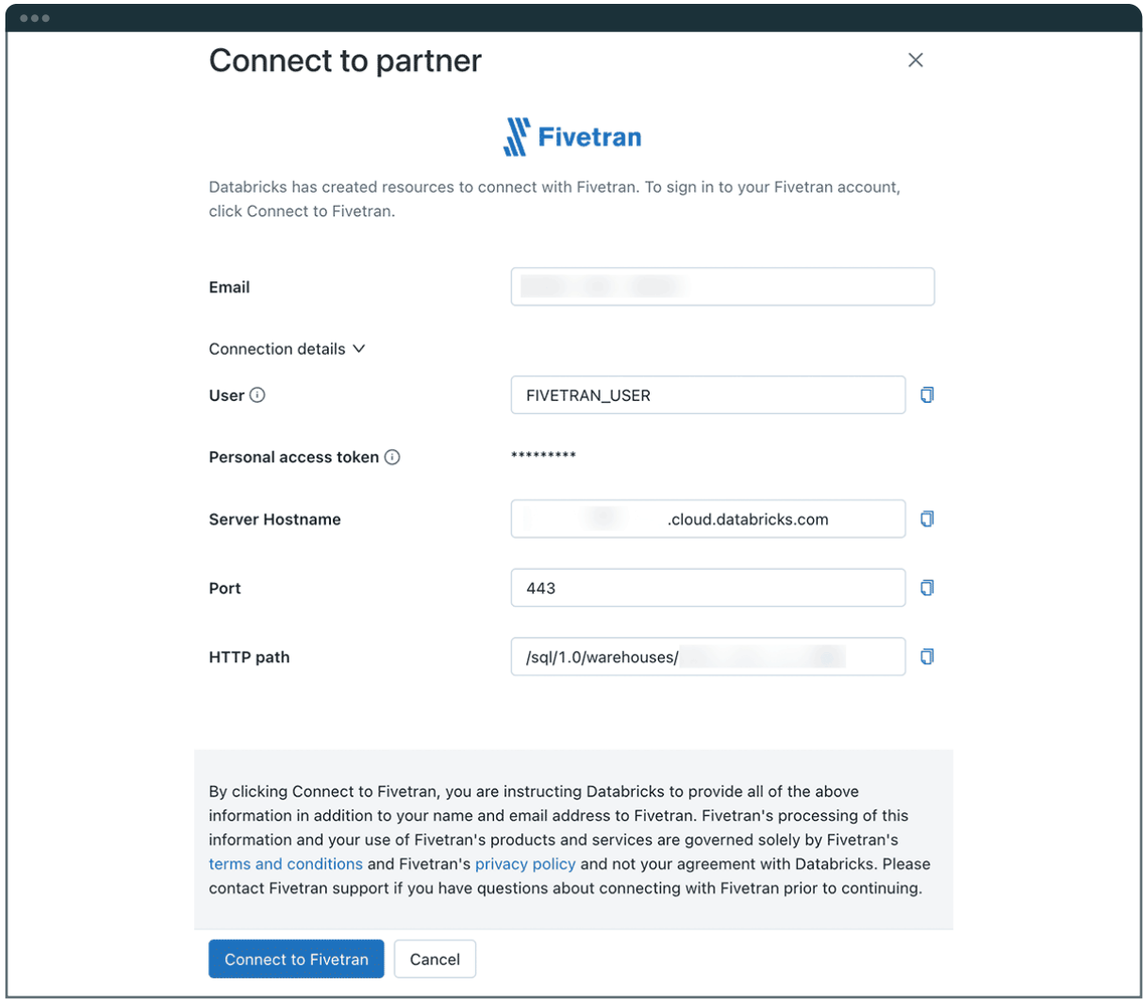


Figure 3: Databricks Partner Connect interface for creating a new connection

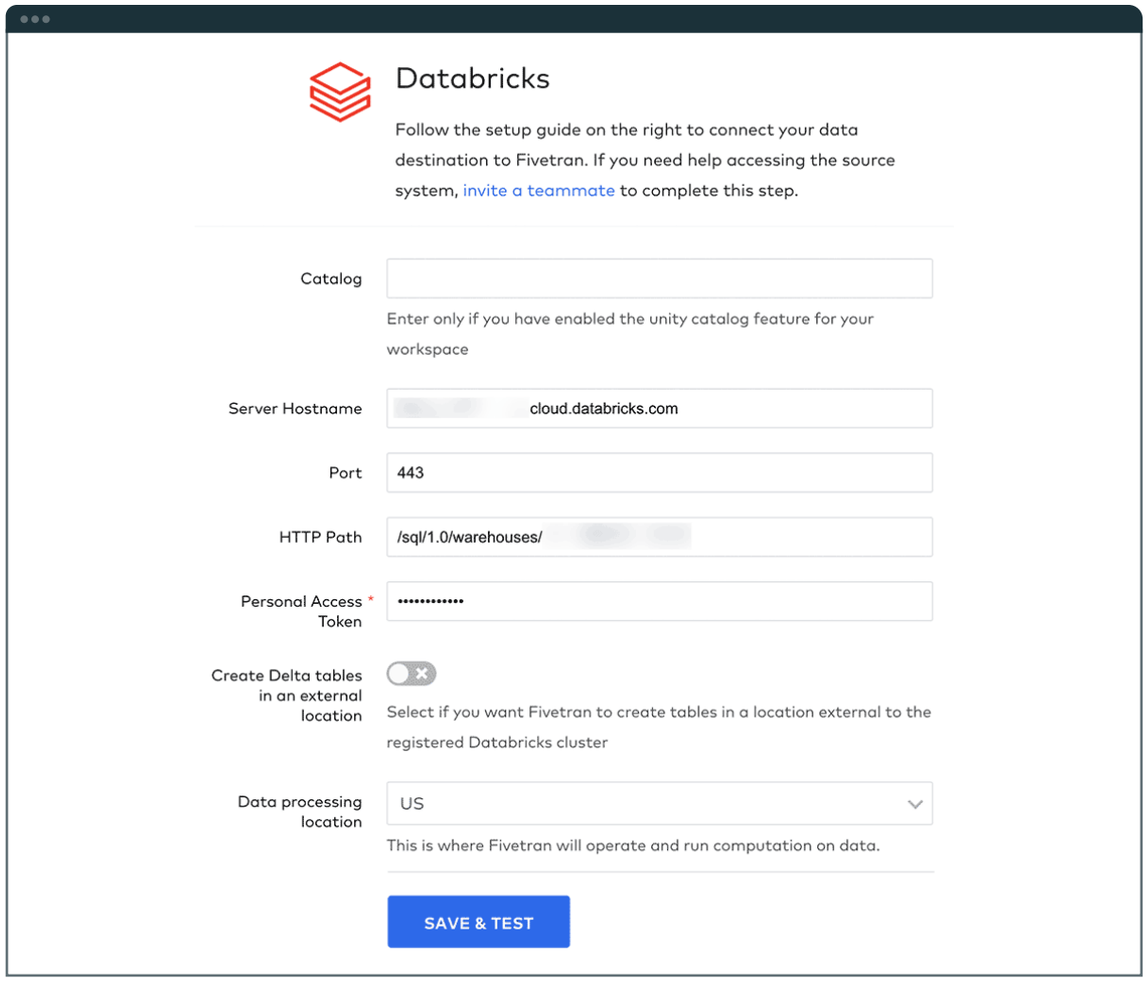


Figure 4: Fivetran interface for confirming a new destination

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.

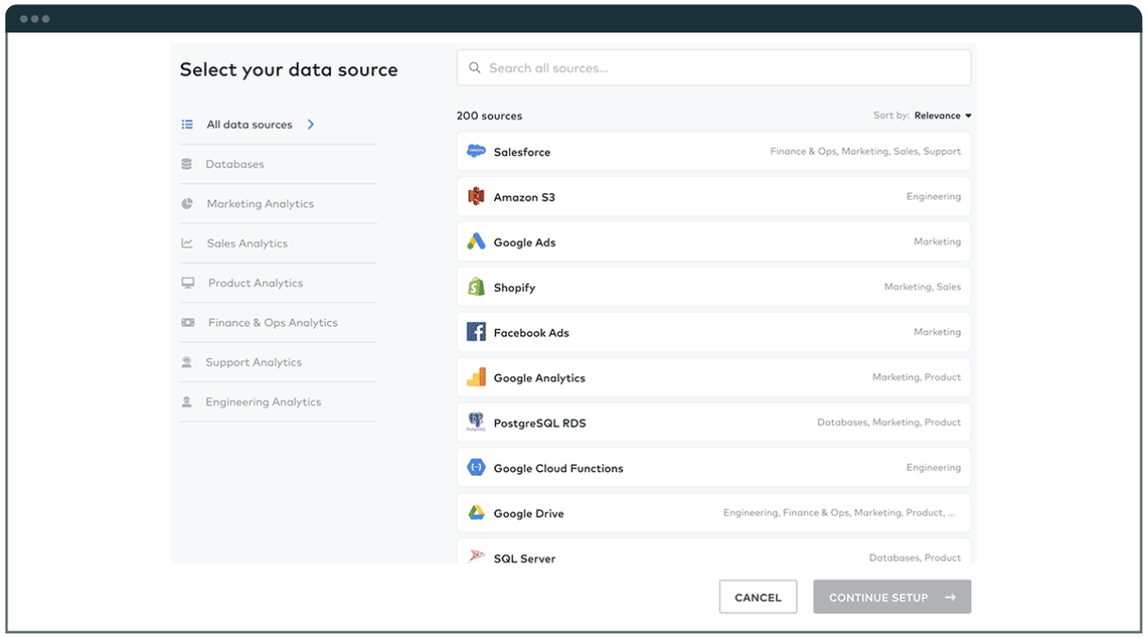


Figure 5: Fivetran interface for selecting a data source type