**Data Observability with Databand**

Elena Lowery, WW Team, Data and AI

A close up of a logo

Description automatically generated

Contents

[Overview 1](#_Toc125974937)

[Required software, access, and files 1](#_Toc125974938)

[Part 1: Introduction to Databand SDK for data pipelines 2](#_Toc125974939)

[Part 2: Use Databand SDK 26](#_Toc125974940)

[Part 3: Review a PySpark example 26](#_Toc125974941)

[Part 4: Create pipelines with lineage 28](#_Toc125974942)

[Part 5: Monitoring DataStage pipelines 32](#_Toc125974943)

[Part 6: Monitoring other pipelines 37](#_Toc125974944)

[Part 7: Monitoring pipelines in Databricks 38](#_Toc125974945)

[Part 8: Integration with Airflow 45](#_Toc125974946)

[Appendix A: Airflow Installation 56](#_Toc125974947)

# Overview

In this lab you will learn how to use Databand to implement data observability for your data pipelines. You will complete the following tasks:

* Use Databand SDK in Python pipelines
* Use Databand SDK in PySpark pipelines
* Review lineage implementation in Databand
* Configure an Airflow cluster to be monitored by Databand.

# Required software, access, and files

1. To complete exercises in this lab, you will need:
2. A *Python IDE*
   * You can use any Python IDE. We provide sample Python code/detailed instructions for
     + *Jupyter Notebooks* in [Cloud Pak for Data as a Service](https://dataplatform.cloud.ibm.com) (*CPDaaS*)
     + *JupyterLab* in [Anaconda Community Edition](https://www.anaconda.com/products/distribution)
     + [PyCharm Community Edition](https://www.jetbrains.com/pycharm/download/#section=windows) (can be installed with Anaconda Community Edition)

We recommend that you use at least 1 notebook environment and at least 1 IDE, such as *PyCharm* or *Visual Studio*.

1. A URL and a userid for the Databand demo environment.   
   The instructor will provide the userid and URL for Databand access.
2. Files from **github** - to be provided by instructor.
3. Optional - if you are planning to complete the *DataStage integration* section:
   1. Cloud Pak for Data as a Service account with the following services provisioned (Lite/free plans): DataStage, DB2

*Note: screenshots and instructions in the lab may be slightly different from the UI because of frequent updates to Databand.*

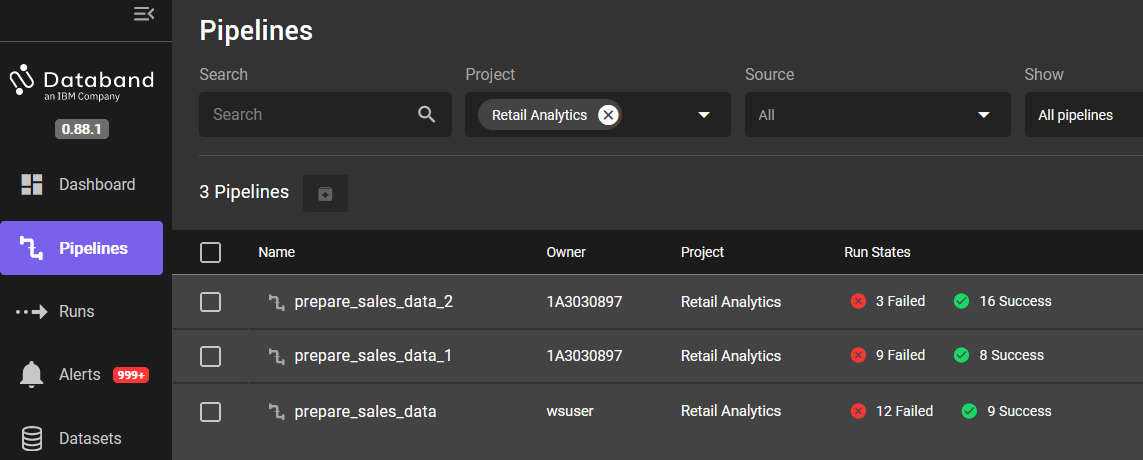
# Part 1: Introduction to Databand SDK for data pipelines

Databand provides the capability to monitor data pipeline status and create alerts for pipeline and data quality issues. In this section you will learn how to use the *Databand* *Tracking SDK* to monitor Python pipelines. Databand also supports data pipelines written in Java, Scala, and dbt. You can find more information about support for these pipelines in [documentation](https://docs.databand.ai/docs/tracking-pipeline-metadata).

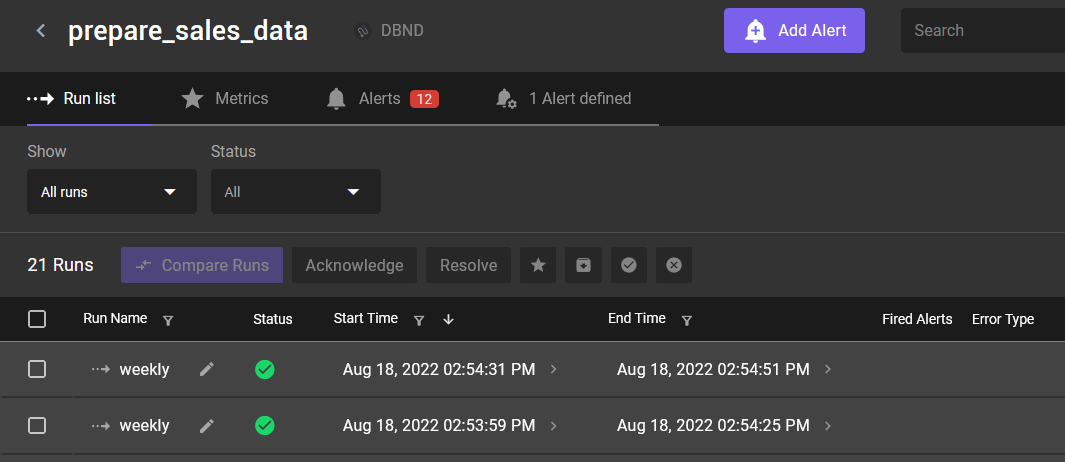
1. Log in to the Databand environment provided by your instructor. If you’re working on this lab on your own time, you can use this [environment](https://ibm-bp-demo.databand.ai/app/dashboard). (<https://ibm-bp-demo.databand.ai/app/dashboard>)

1. Navigate to the **Pipelines** page and find the *prepare\_sales\_data* pipeline. You can filter by project – *Retail Analytics*.

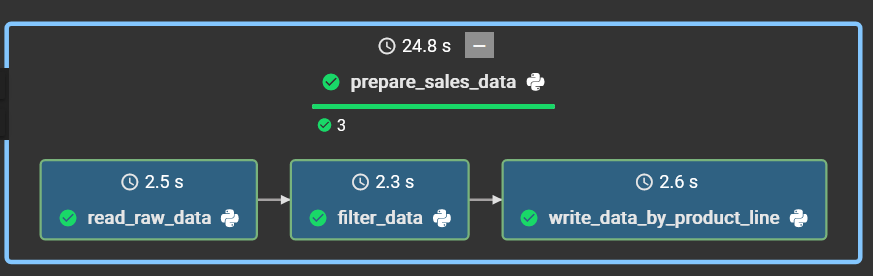
This simple data pipeline reads data from a csv file, filters out several columns, and writes 2 datasets for different product categories.



1. Click on the pipeline, then on one of the successful runs.



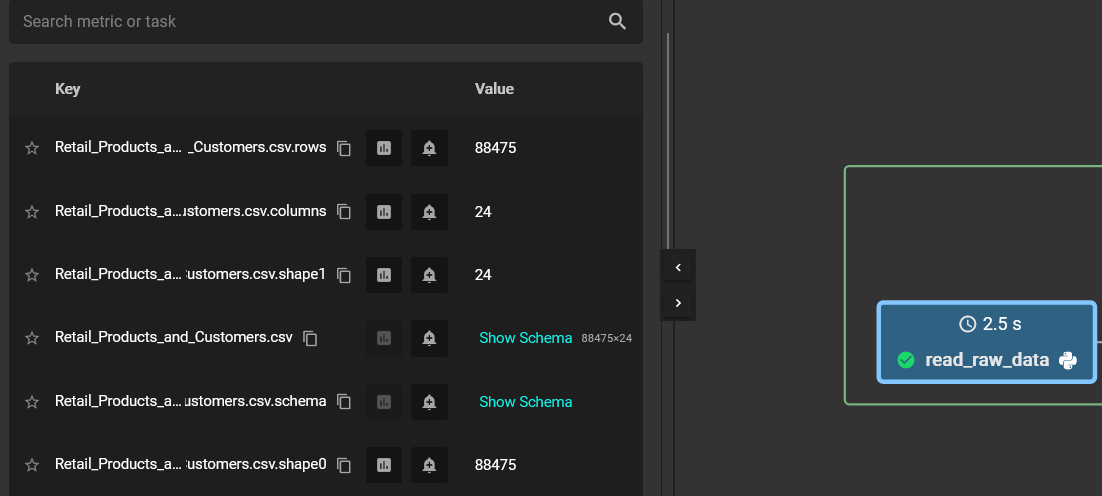
This view shows that the pipeline has three steps: *read\_raw\_data*, *filter\_data*, and *write\_data\_by\_product\_line*.



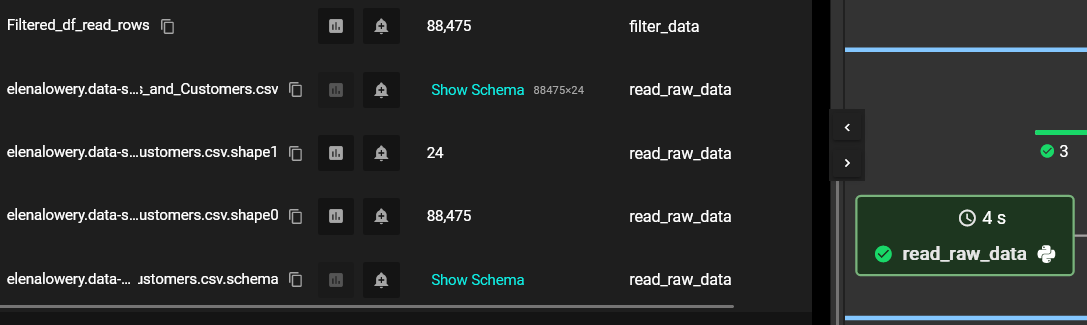
|  |  |
| --- | --- |
|  |  |

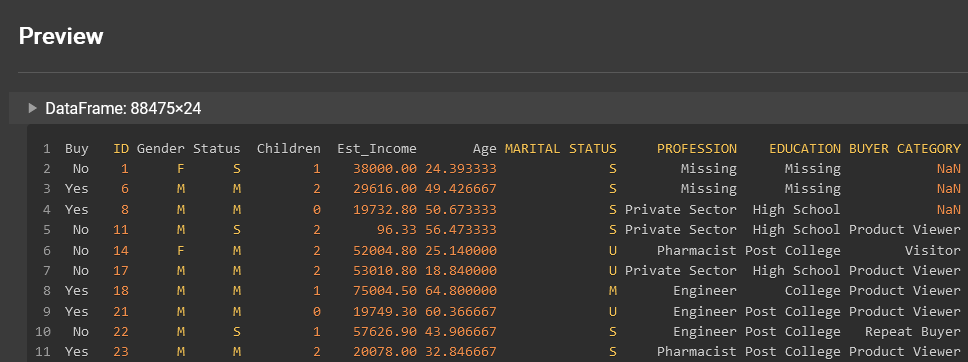
1. Click on the *read\_raw\_data* step in the pipeline, then switch to the **Metrics** tab.

The **Metrics** tab shows default and custom metrics that can be monitored for each step of the pipeline. The metrics on this page will be easier to understand after we review code for the pipeline.



Click **Show Schema** next to *<your\_name>Retail\_Products\_And\_Customers.csv*. Here we see a preview of data that was read in the *read\_raw\_data* step of the pipeline.

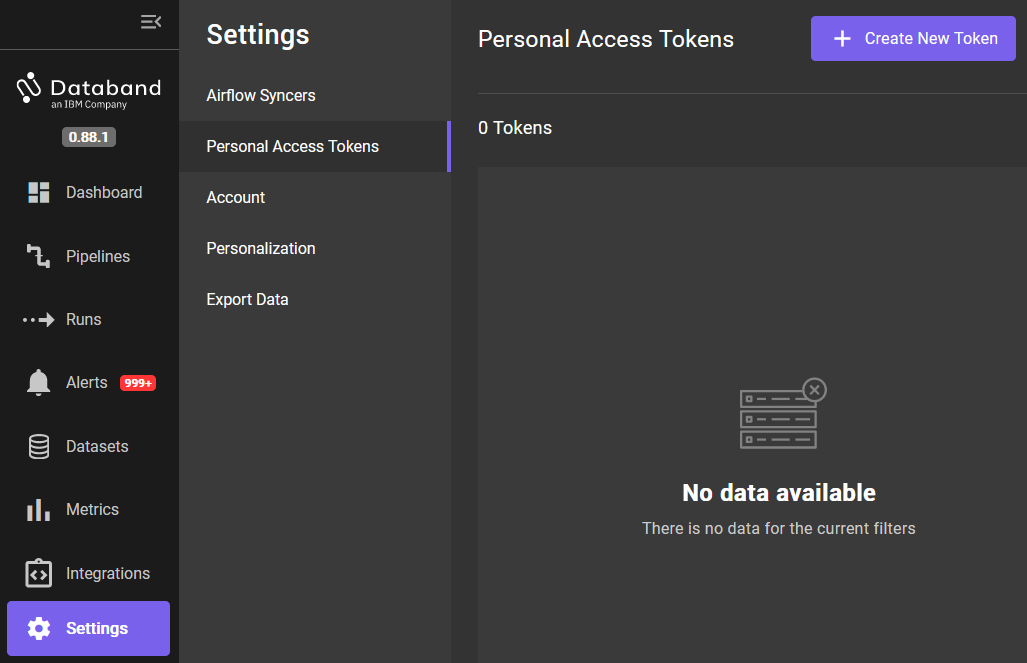




Next, you will create a copy of this pipeline and explore the details of monitoring in Databand.

1. Get an authorization token from Databand. The token is used in data pipelines when establishing a connection to Databand.

In Databand click on **Settings**, then select the **Personal Access Tokens** tab. Click the **Create New Token** button. Provide a unique name (for example, add your initials), and save the token in a notepad.



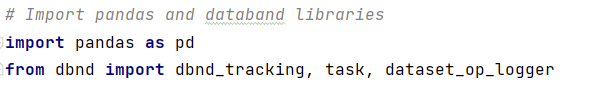
1. Load the sample data file and the Python script into your IDE. The samples are located in the *Workshop*/*Pipelines* folder (downloaded from github).

* Script: *SimpleRetailDataPipeline\_with\_Databand.py*
* Csv file*: Retail\_Products\_and\_Customers.csv*
* *Notebook: SimpleRetailDataPipeline\_with\_Databand.ipynb*

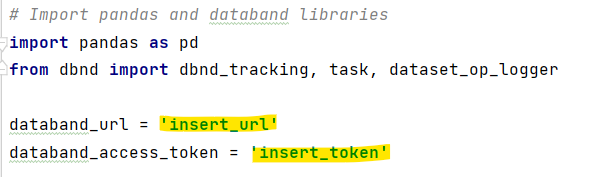
In this lab we will review the *Python script*. If you’re using the notebook example, see the instructions and comments in the notebook.

1. In the beginning of the script, we import pandas and Databand libraries. If your Python IDE does not automatically install libraries, then you can install them with pip:

* pip install pandas
* pip install databand

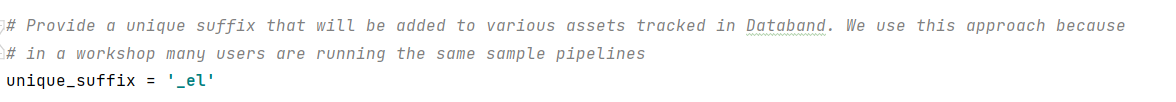


1. Replace the *url* and *token* variables with the values from the Databand cluster.



1. Replace the value of the *unique\_suffix* variable to your initials.

Since we may have many workshop participants using the same cluster, adding a unique suffix to assets that are tracked in Databand will make it easier for your to find your pipelines and datasets.



1. Scroll down to the bottom of the script – to the *prepare\_retail\_data()* function.

Review the code:

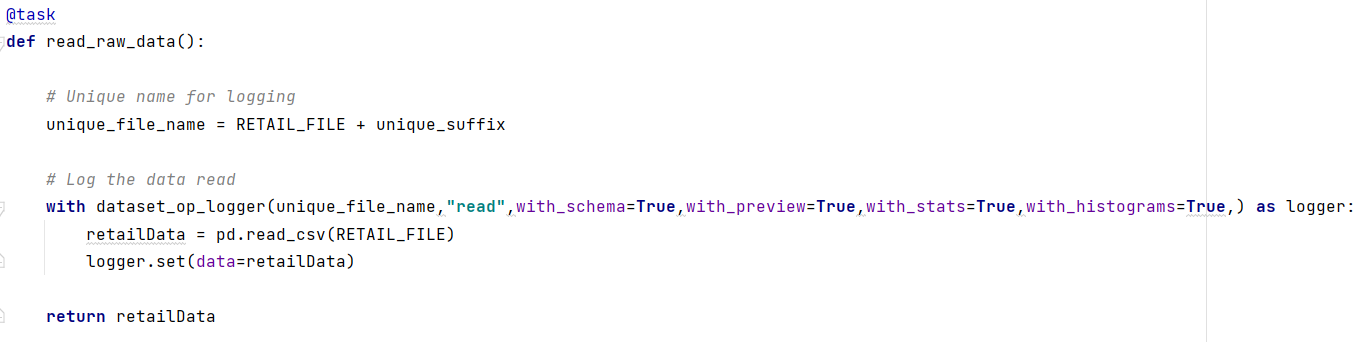
* This function starts tracking the pipeline in Databand and invokes 3 functions that represent pipeline steps: *read\_raw\_data(), filter\_data(), write\_data\_by\_product\_line()*. Notice that in Databand the names of the steps match the names of the Python functions in our pipeline.



* The main function, *prepare\_retail\_data,* starts tracking execution of the pipeline in Databand. Notice that the *job\_name* corresponds to the *pipeline name* in Databand and *project\_name* corresponds to the *project name*.

|  |  |
| --- | --- |
|  |  |

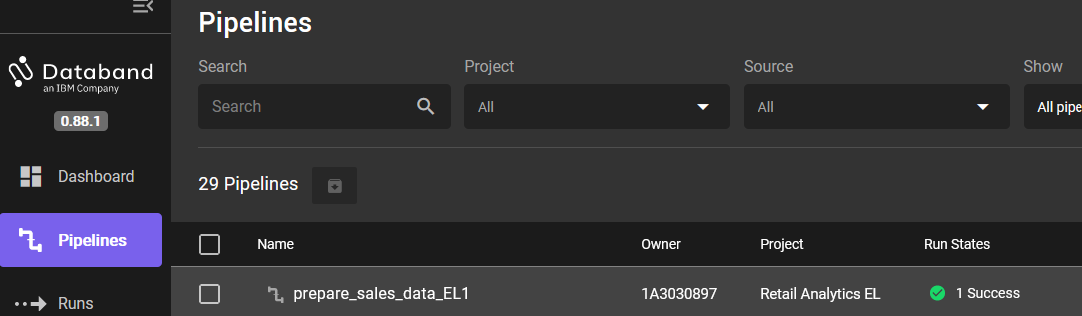
* Review the implementation of each function.
  + First, we add the *@task* decorator before the declaration of each function in Python. This lets Databand know that we are starting execution of a pipeline step. The *@task* decorator uses the name of the function directly below it as the name of the pipeline step in Databand.
  + Next, we log datasets that are used in the pipeline with the *logger.set()* call. This call will log *metadata* and a small set of sample data. If the customer has concerns about logging data, they can turn off logging the data sample.
  + In our example we read a csv file into a pandas dataframe, which is what’s logged in Databand. Logging dataset metadata in Databand is optional, and it should only be used if data engineers need dataset metadata for troubleshooting issues. When you log datasets, you will see data in the **Metrics** tab of a pipeline step. If you switch back to Databand **Metrics** view of the *read\_raw\_data* step, you will notice that the schema and sample records match the few rows of the csv file.



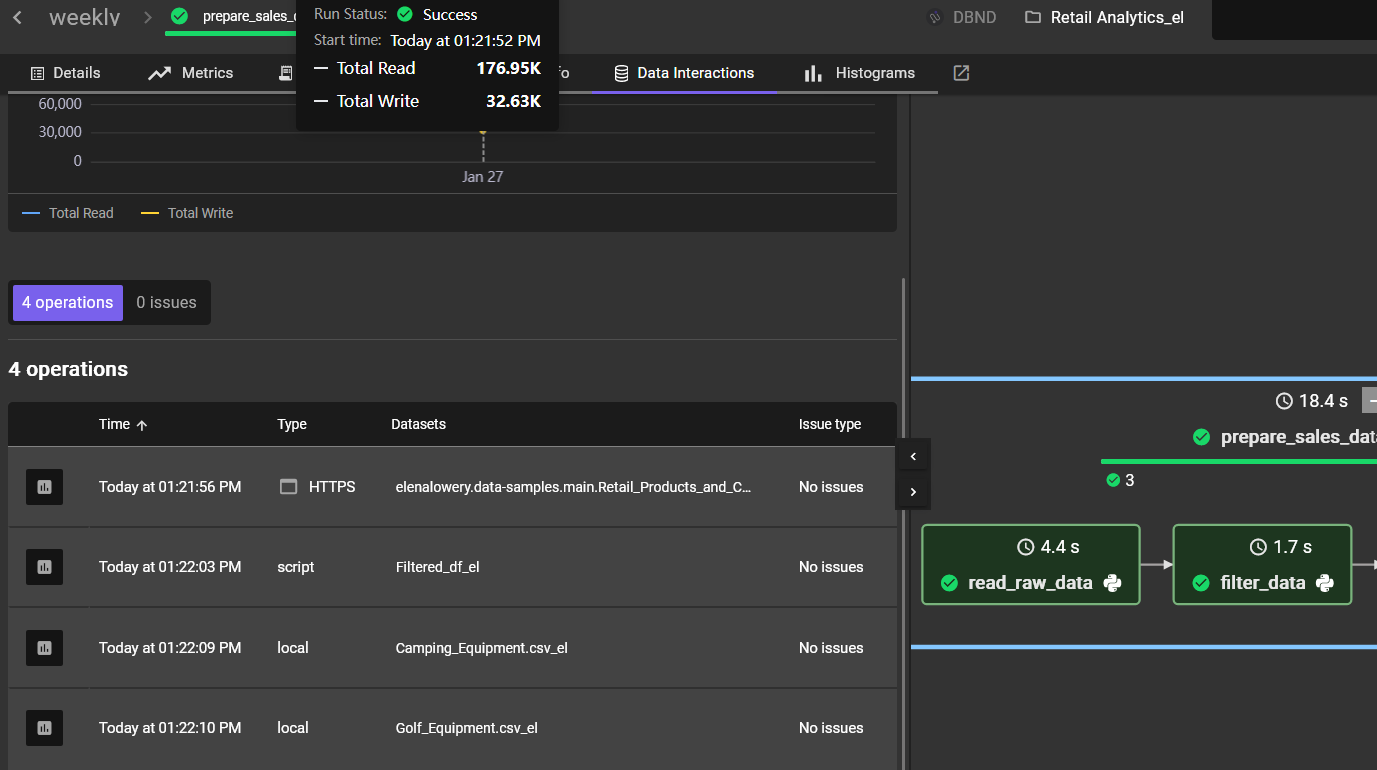
Next, we will run the code.

1. Save the changes you’ve made in the script and run it at least 5 times to generate some metrics data. You can run the script in debug mode if you would like to review the SDK in more detail.

* As you are running the script, switch to the Databand environment, and monitor pipeline execution. You can find the pipeline in the **Pipelines** view.



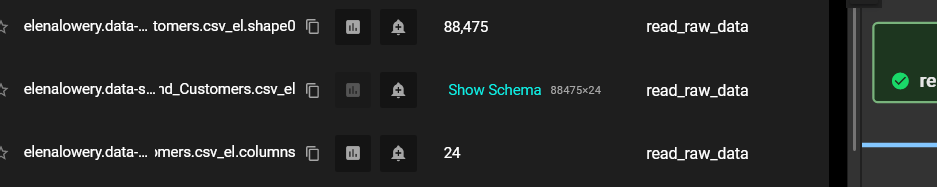
* Drill down to see the details of the run.



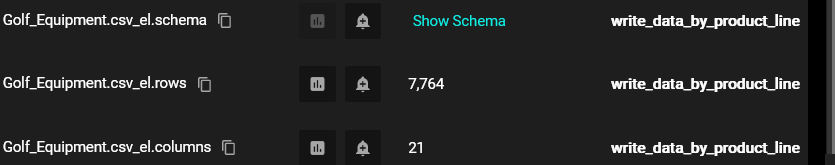
1. Click on the **Metrics** tab, then select each step of the pipeline.

Explore the **Metrics**, notice that the schema corresponds to the dataset read or written in each step.

For example, the *Retail\_Products\_And\_Customers.csv* dataset (in Python the *retailData* pandas dataframe) has 88,475 rows and 24 colums.

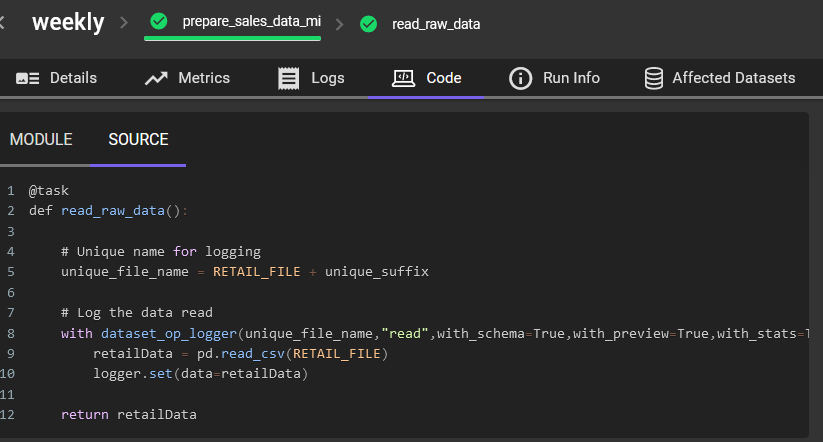


If you click on the *write\_data\_by\_product\_line* step, you will notice that one of the output datasets, *Golf Equipment* (in Python the *GoflEquipment* pandas dataframe), has 7764 rows and 21 columns.

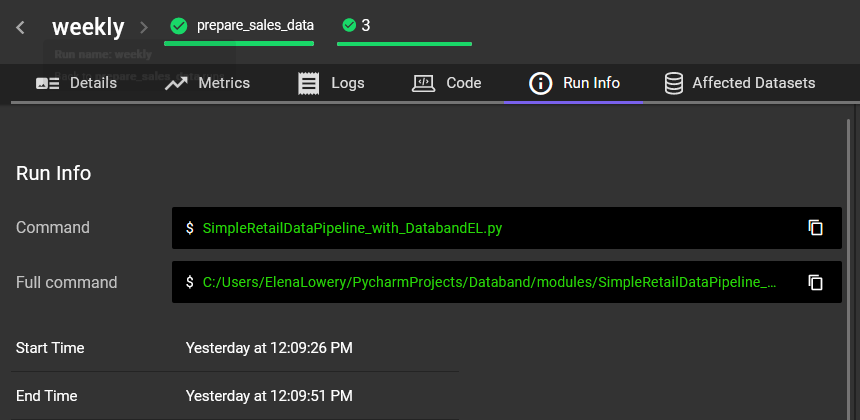


1. Click on the **Code** tab, then select each step of the pipeline.

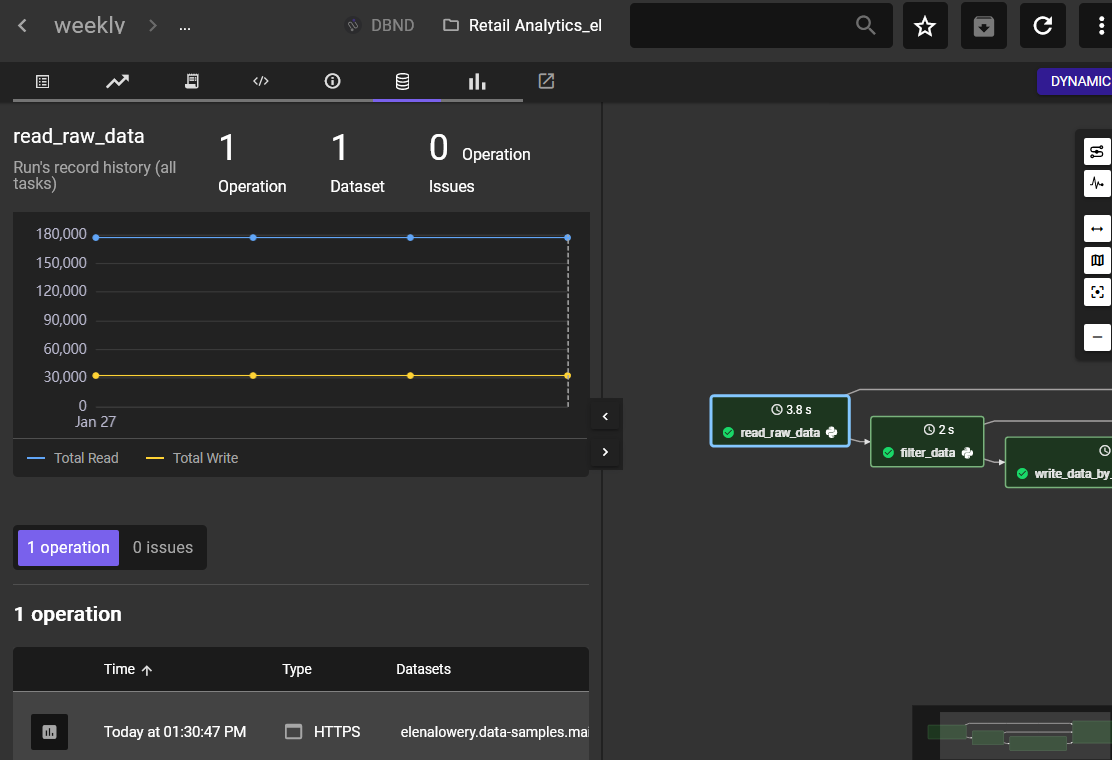
Here we can review (but not edit) code that corresponds to the pipeline step. Databand is not used for editing code, code changes should be done in the IDE that’s used to develop pipelines.



1. Click on the **Run info** tab and notice that it captures the name of the Python script that implements the pipeline.



1. Click on the **Data Interactions** tab, then on one of the steps in the pipeline. Here we have a single view of all datasets used by the pipeline step. Since we don’t have any issues yet, we don’t need to review additional details at this time.



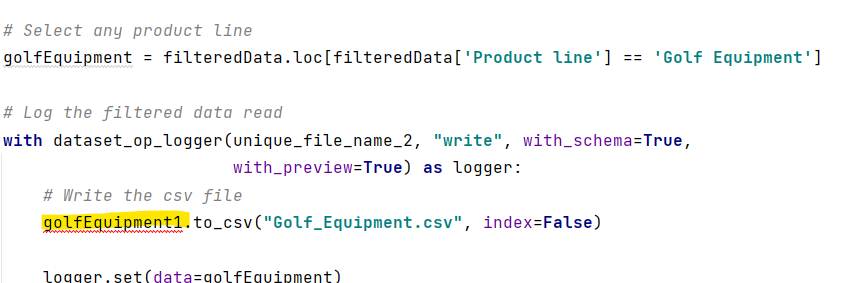
Next, we will introduce a few errors in the pipeline to understand how Databand can help us monitor and troubleshoot the errors.

We will start with introducing an error in the last step of the pipeline.

*Note: In this lab we are adding errors that are typically resolved at* ***build time*** *because it’s the easiest way to create an error in code. In a production environment* ***runtime errors*** *will be handled the same way by Databand.*

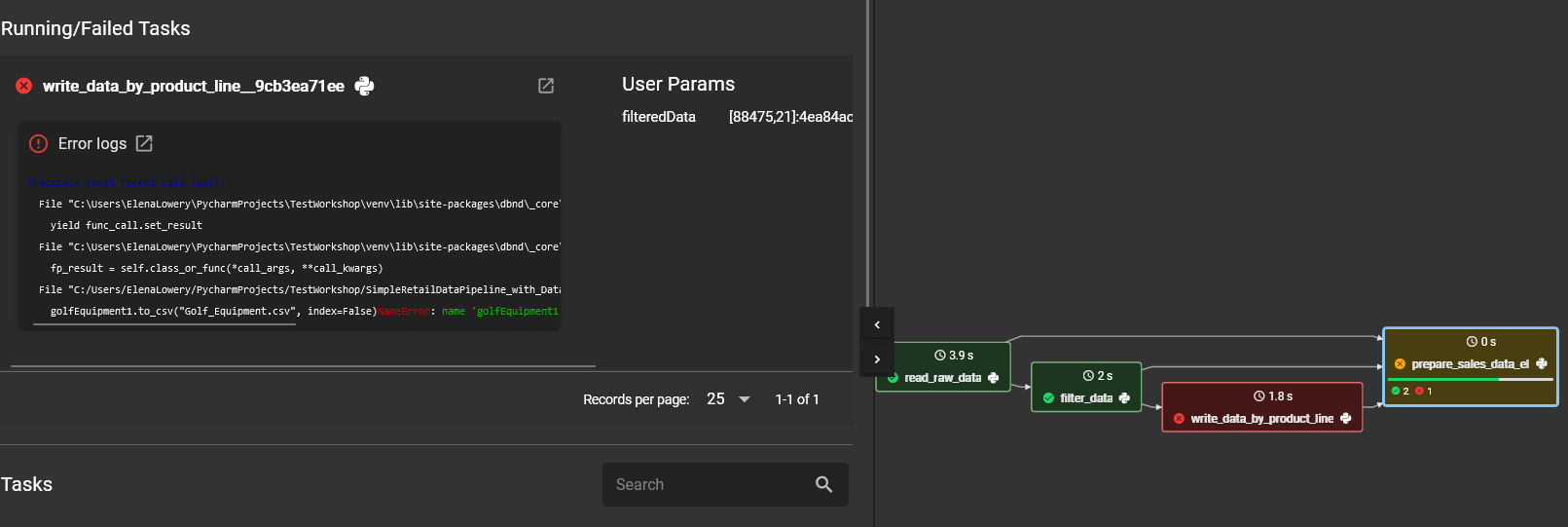
1. In the *write\_data\_by\_product\_Line()* function, find the line of code that writes the equipment csv and change it to the name of the pandas dataframe that doesn’t exist, for example, *golfEquipment1*.

*Note: While PyCharm shows the error, the code will still run because Python is not a compiled language.*



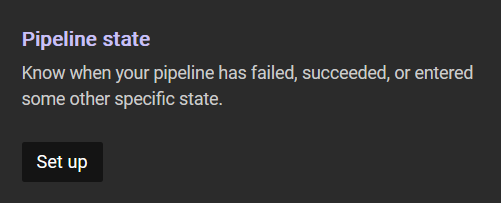
1. Save the change and run the script. Switch to Databand and find your pipeline in the **Pipelines** tab.

Databand shows that the 3rd step of the pipeline failed and displays the error that caused the failure.

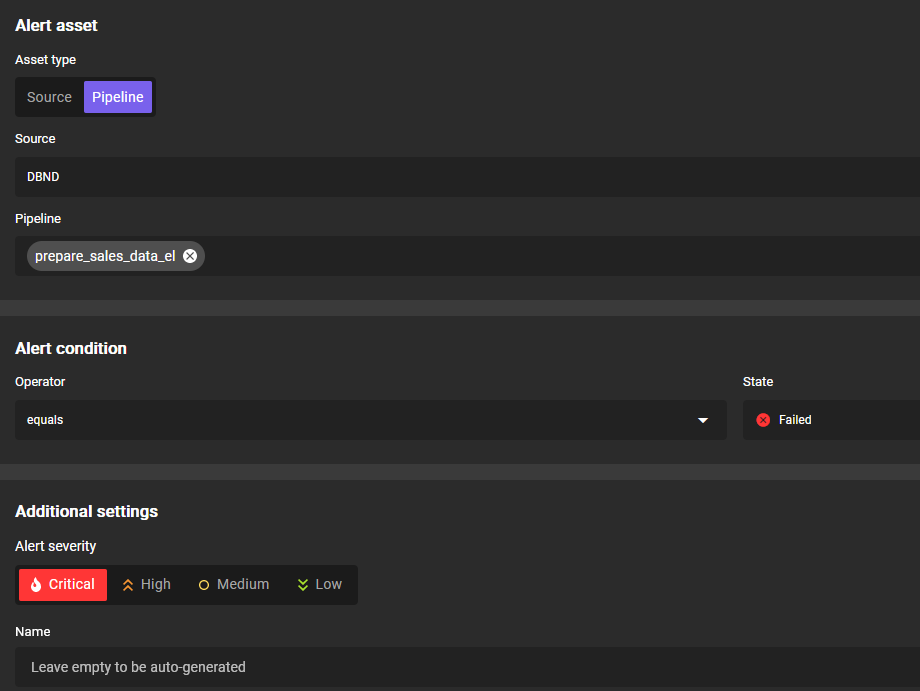


1. Switch to the **Alerts** view and click **Add Alert**.

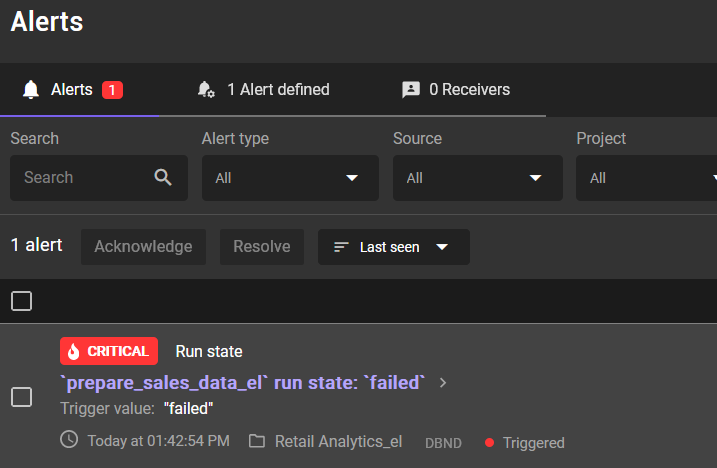
Create an alert for **Pipeline State** *Failed*, and make it a *Critical* severity alert.



*Note: The Source project for all pipelines in this workshop is DBND.*



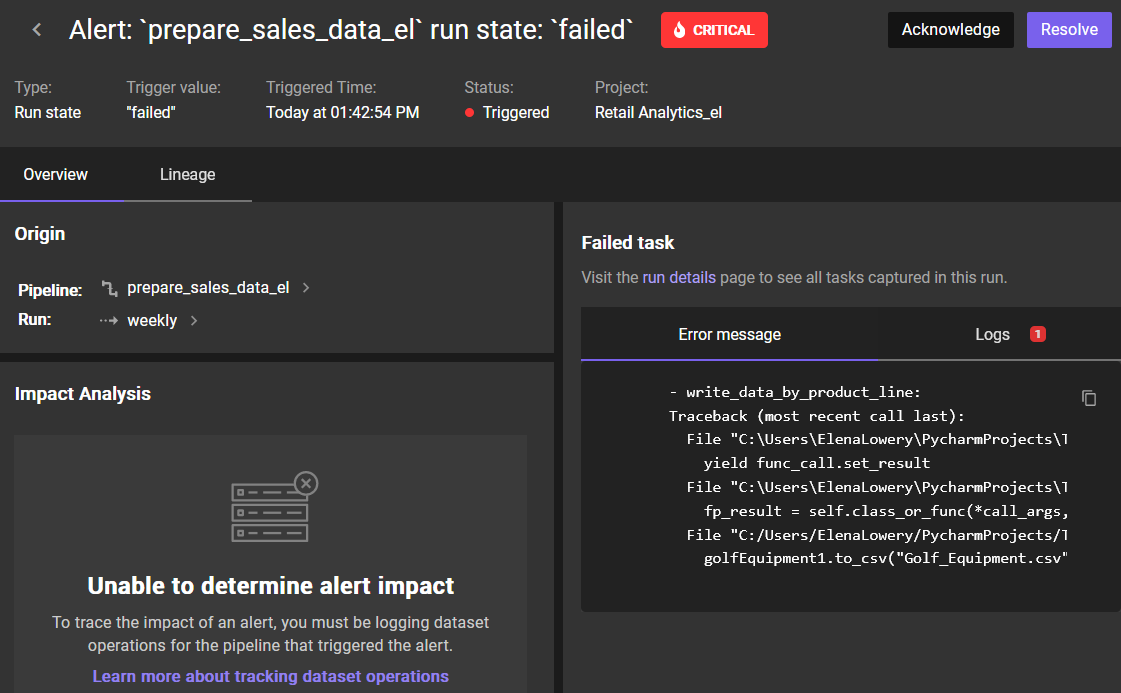
Run the pipeline again and refresh the **Alert** view. You should see a critical alert for your failed pipeline run.



1. Drill down to the alert and review the details.

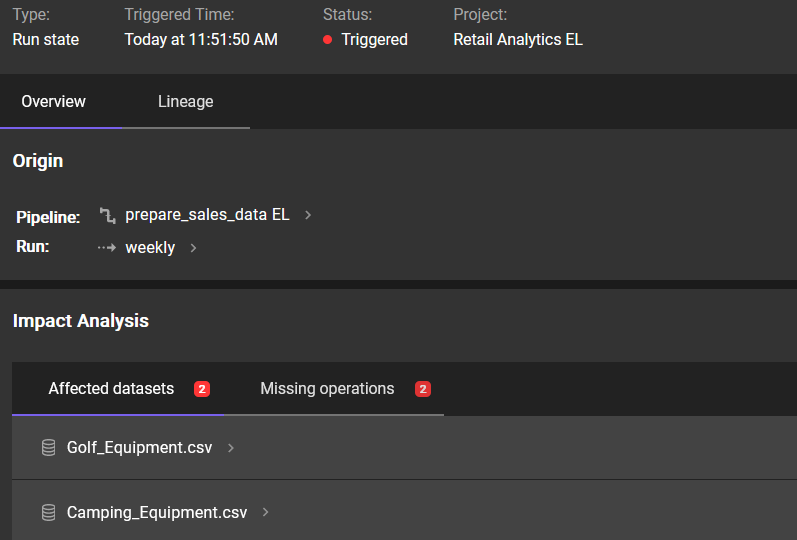
As on the **Pipelines** page, we can see the error that caused the failure. We can also review the datasets that could be potentially affected by the error in the pipeline step.

During the first few failed runs of a new pipeline that’s observed by Databand, you may see the message “*Unable to determine alert impact*.” This happens because the impact analysis job did not run yet. If you see this message, return to the alert after 10-15 minutes. You can continue with the rest of the instructions in the lab.



Databand is able to determine (infer) the list of affected datasets because

* We used the *logger.set()* function in our data pipeline
* We had several successful runs of the pipeline, and these datasets were read or written in the successful runs.

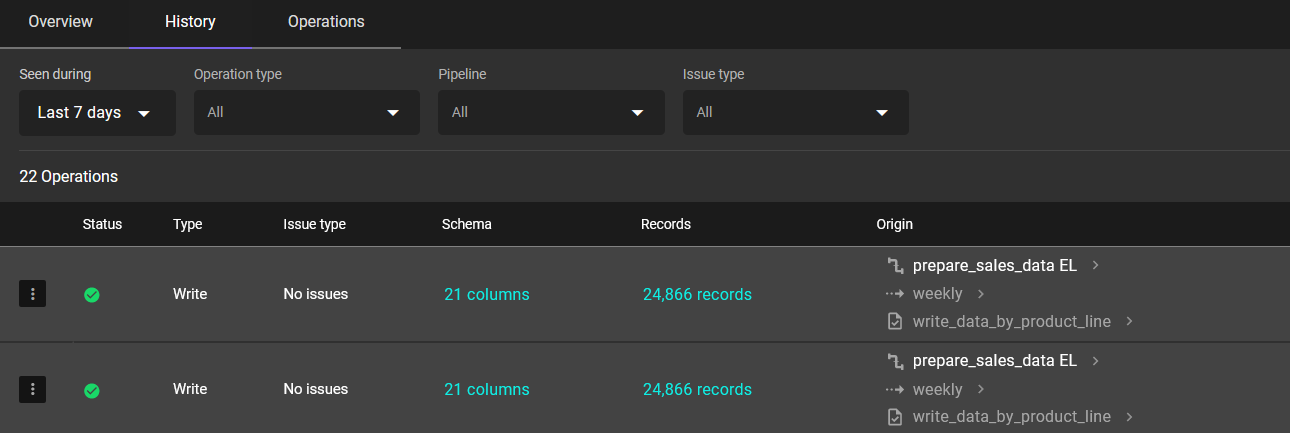


Since the error in a Python function can happen before or after the dataset is written, we need to investigate if the expected number of rows was written to the dataset.

1. Click on *Camping\_Equipment.csv*, then the **History** tab.

Based on this output, we have a consistent number of rows/columns that are being written to this dataset, including during the pipeline run that has failed (the last run shown on top of the table). That means that even though the pipeline run failed, data was written successfully.

This output aligns with the error we introduced in the code – it’s after we write the *CampingEquipment.csv*.

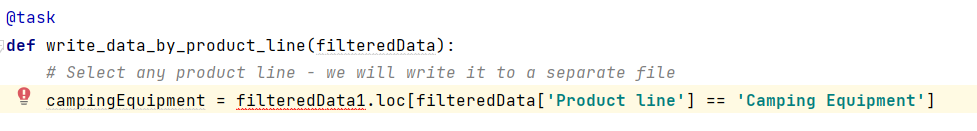


Next, we will introduce the error earlier in the code.

1. In the *write\_data\_by\_product\_line()* function find the line of code that references the pandas dataframe that was passed into the function, and change it to a different name, for example, *filteredData1*.

Since *filteredData1* does not exist in this function, we will get an error.

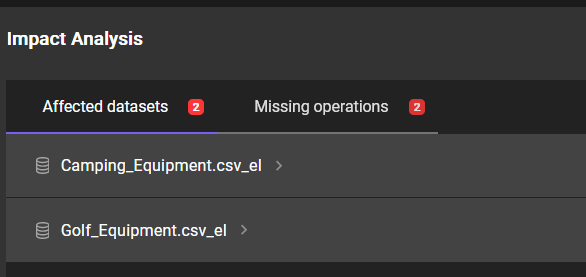
Run the script and switch back to Databand.



1. From the **Alerts** page, navigate to your failed pipeline run.

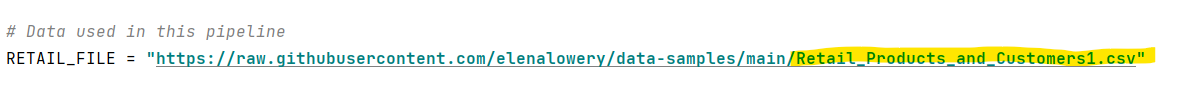
Notice that in addition to the **Affected datasets** tab, we now see **Missing operations** tab. A “missing operation” means that the code that writes datasets did not run. Databand knows that a successful execution should result in writing the *Camping Equipment* and *Golf Equipment* datasets because we had several successful runs of the pipeline.

In general, Databand *infers* affected datasets and pipelines by observing successful pipeline runs, which means that we should always ensure that a pipeline runs successfully several times after we configure it to be observed by Databand.



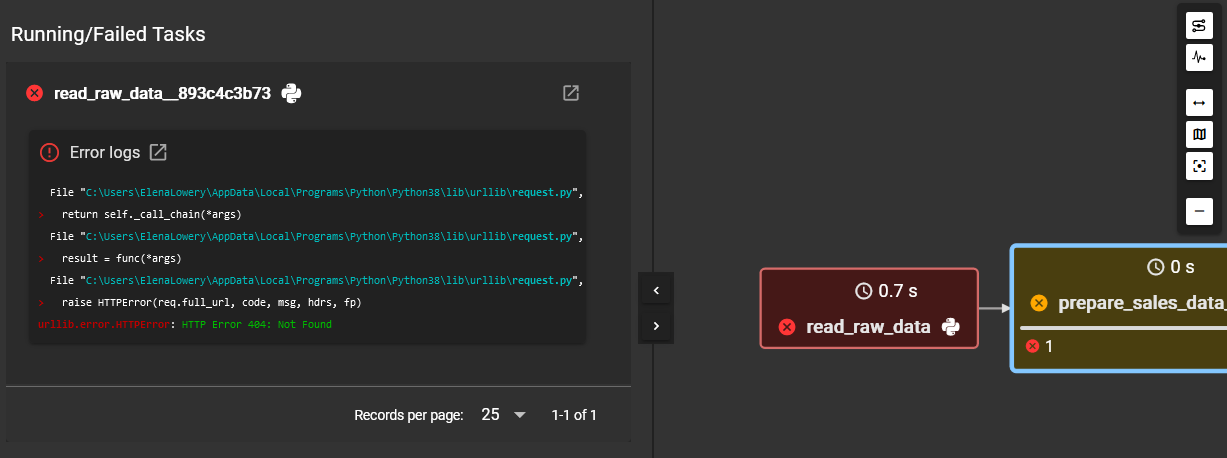
Next, we will review what happens if we have errors earlier in the pipeline.

1. Add an error in the 1st step of the pipeline. For example, change the name of the csv file to the name of the file that does not exist (global variable). Save the script and run it.

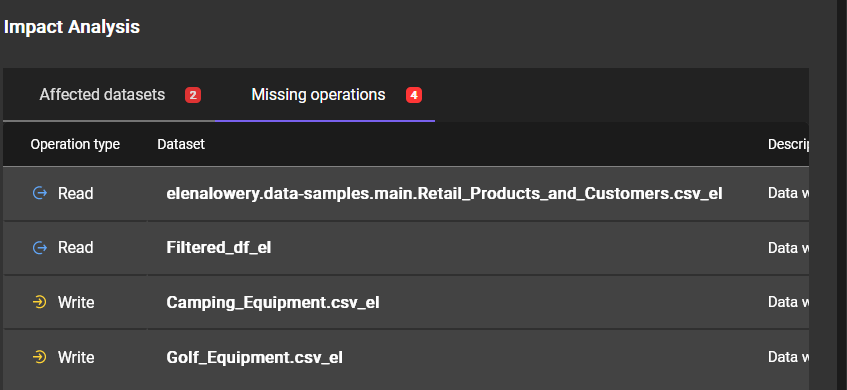


Switch to Databand and find your failed pipeline run (from the **Pipelines** view).

Since the pipeline failed on the first step, the other steps are not shown.



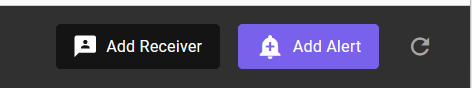
We can see that more datasets are shown in the **Missing Operations** tab of the pipeline alert.



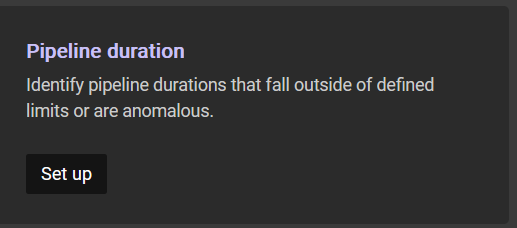
Next, we will configure 2 other types of alerts that are not pipeline failures, but nevertheless could signal a problem with the pipeline or data.

One of the most frequent alerts that a data engineering team is interested in is the *run duration* alert because it can indicate a problem with the pipeline.

1. In Databand switch to the **Alerts** panel and click **Add Alert**.



1. Select **Pipeline duration**.

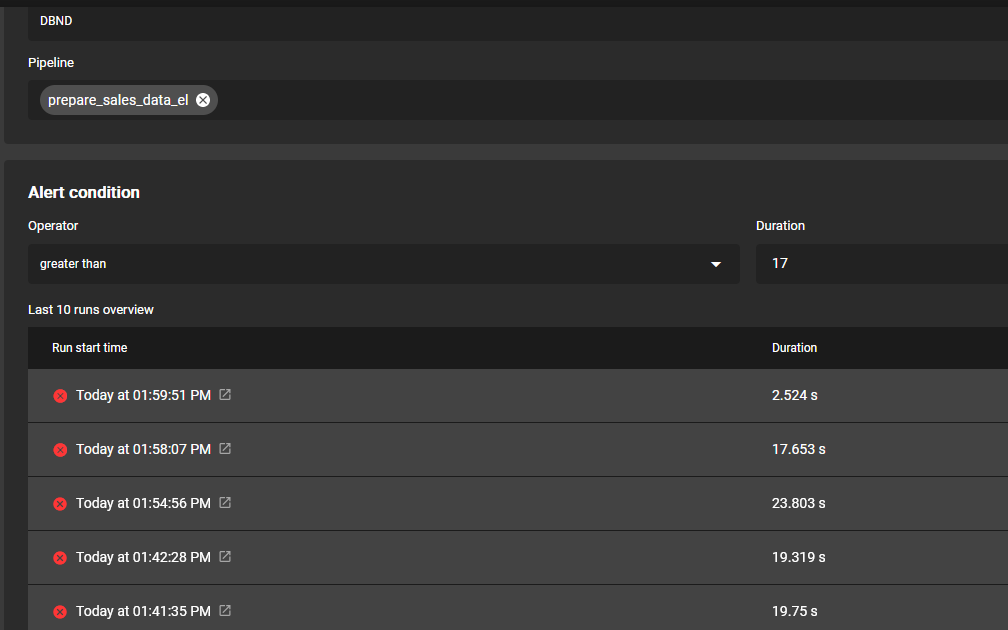


1. Find your pipeline and define the alert.

*Note: The Source project for all pipelines in this workshop is DBND.*

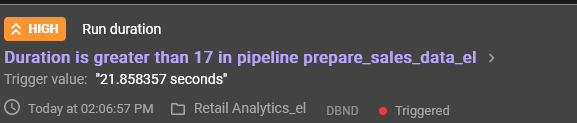
Select **Run Duration greater than**,and enter the value that’s smaller than the shortest duration of your pipeline runs.

In our example the longest successful run is *17.6* seconds, that’s why we are specifying *17* seconds because we want to “force” the alert. Select any alert severity, then add a name and a description.

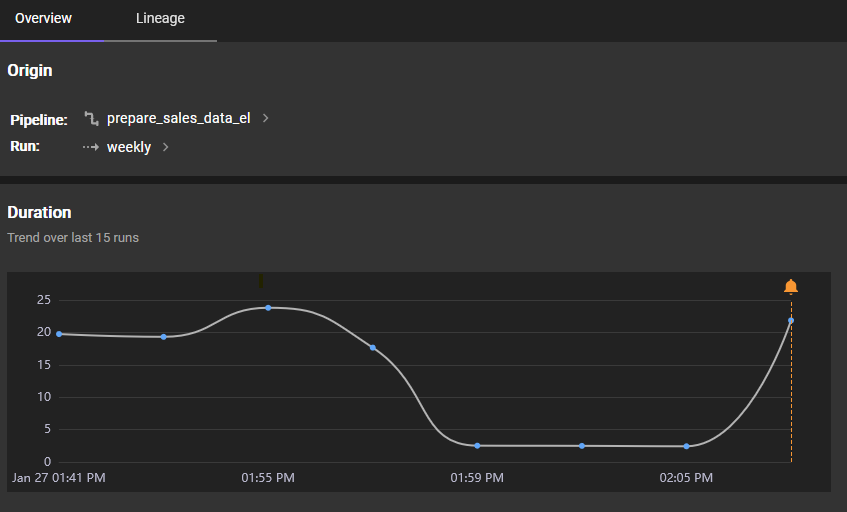


Save the alert.

1. Run your pipeline and check the **Alerts** page.



1. Drill down to the alert to view the details.

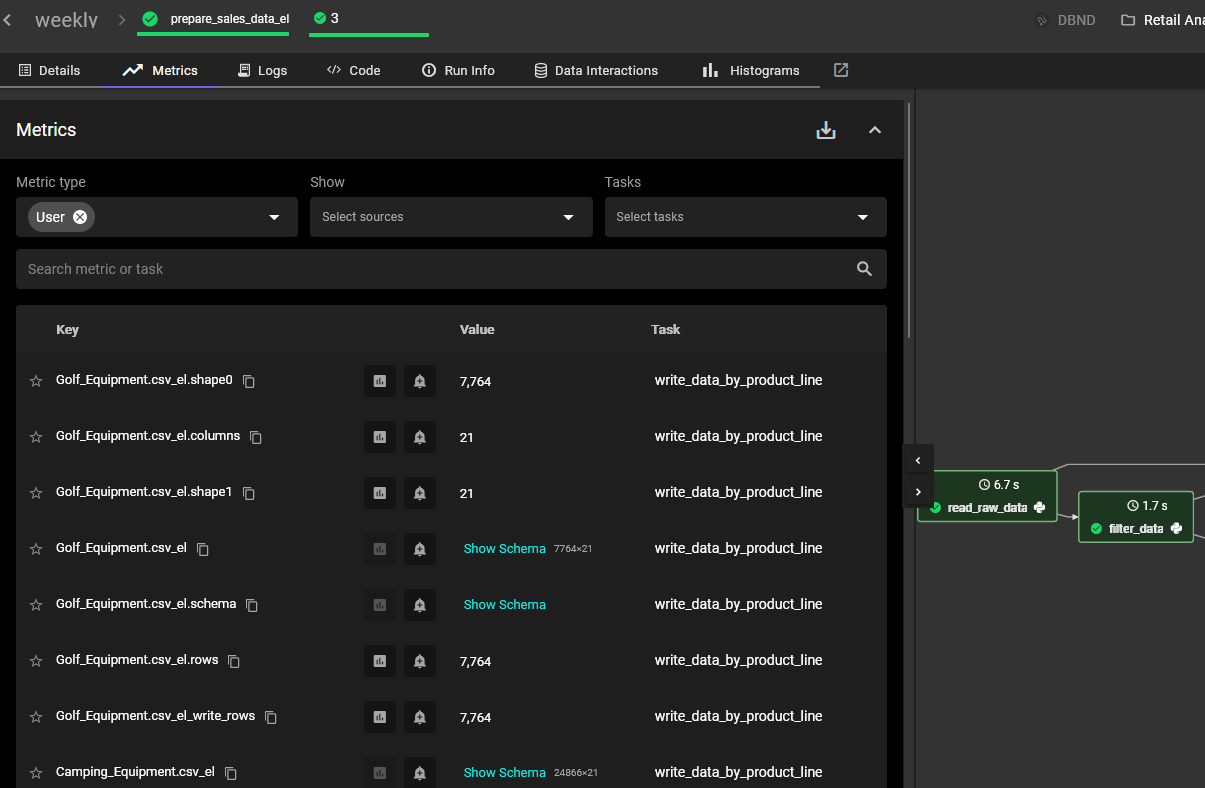


Now that you’ve tested the alert, either delete it or change the run duration to a higher value so that it does not generate too many unnecessary alerts in the workshop environment.

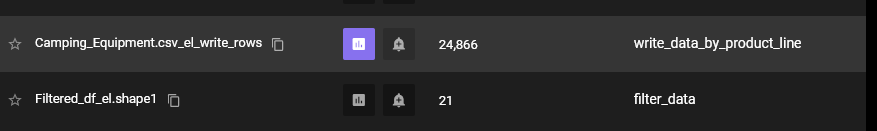
Next, we will create an alert for the *number of rows written* to the output dataset. We will use a different navigation approach to get to the alert definition page.

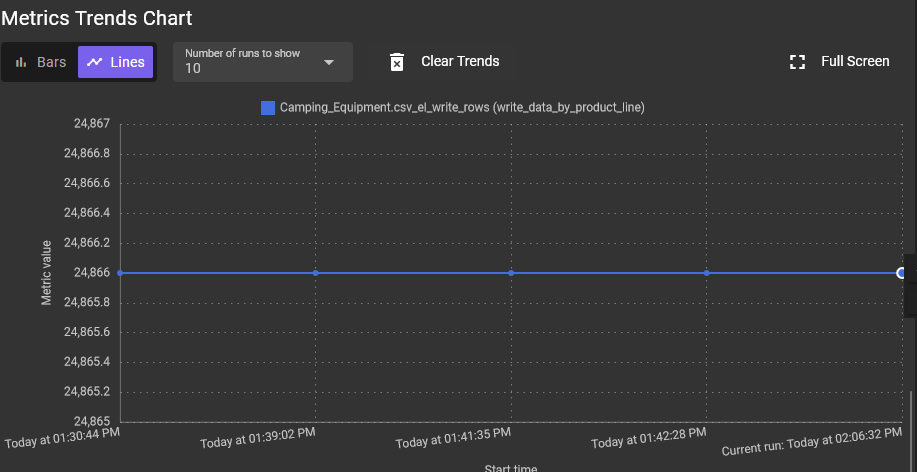
1. Navigate to your pipeline from the **Pipelines** panel, drill down to one of the successful runs.

Click on the **Metrics** tab.



Click on the **Show on chart** icon next to *Camping\_Equipment.csv\_write\_rows* and review the number of rows that has been written up to this point (scroll down to see the chart). Since we did not have runtime errors, the number of rows has consistently been 24,688. As a reminder, we created errors that resulted in a “missing operation”.



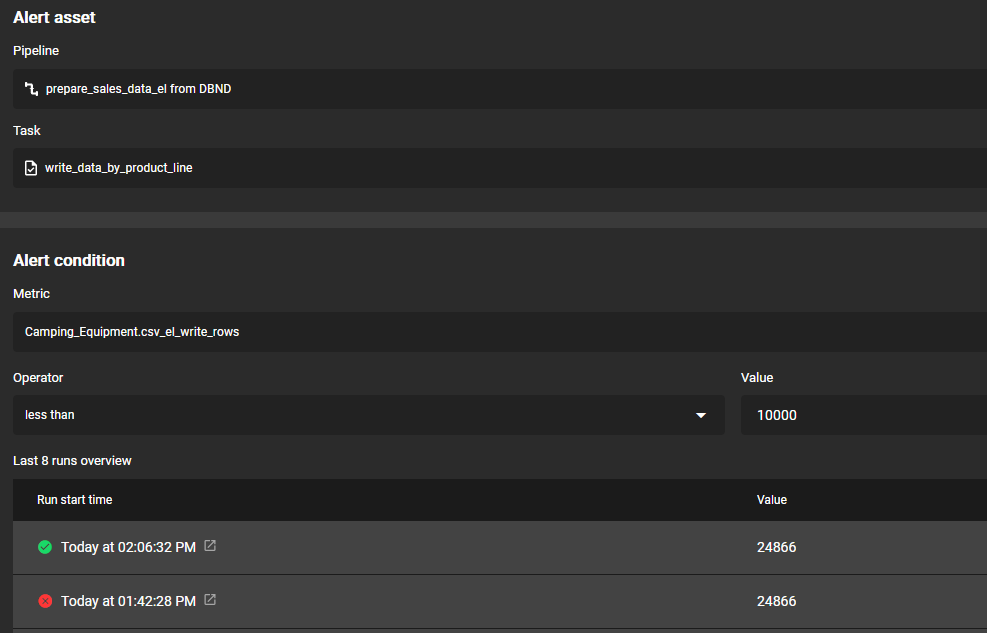


1. Click the **Create Alert** button next to *Camping\_Equipment.csv\_write\_rows* (the bell icon). Notice that when the **Create alert** panel is displayed, the number of rows is pre-selected.

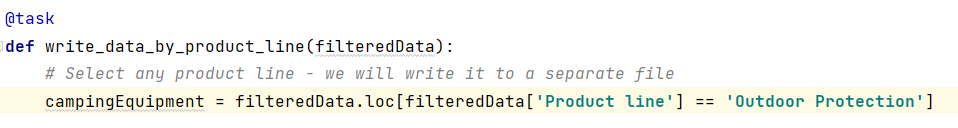
You can choose whether to create an alert based on a hardcoded value or an anomaly. We decided to create an alert for the number of rows less than 10,000.

If you don’t want to change the pipeline code to test this alert, then simply specify the number less than 24,688 to create the alert. However, we recommend that you change the code to test the alert.

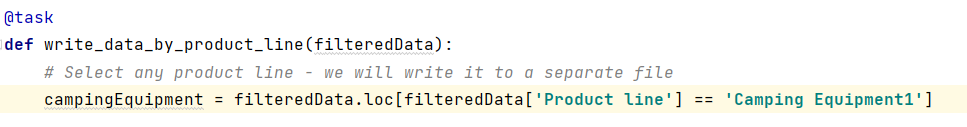
Select the *Camping\_Equipment.csv\_write\_rows* as the metric.



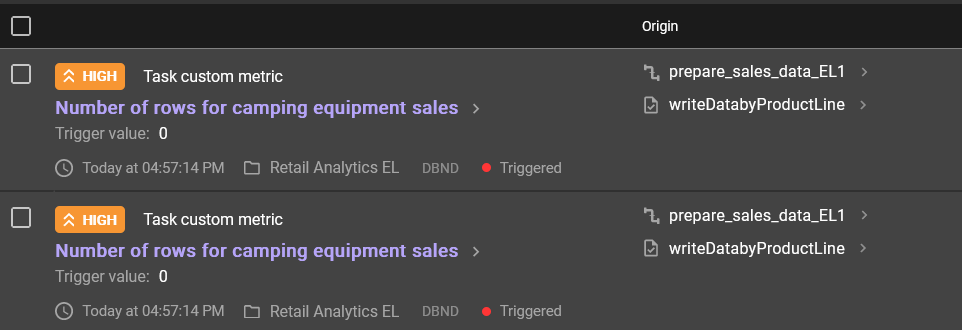
In the *write\_data\_by\_product\_line()* change the line of code that selects records from the pandas dataframe to a different valid value, for example, *Outdoor Protection*. The number of records for this filter will be different. Save the script and run it.



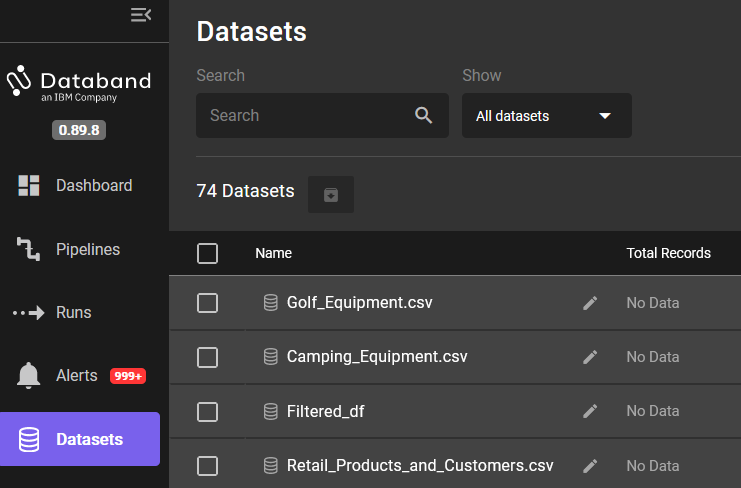
Next, change the value to an invalid value, for example, *Camping Equipment1*. The number or records should be 0. Save the script and run it.



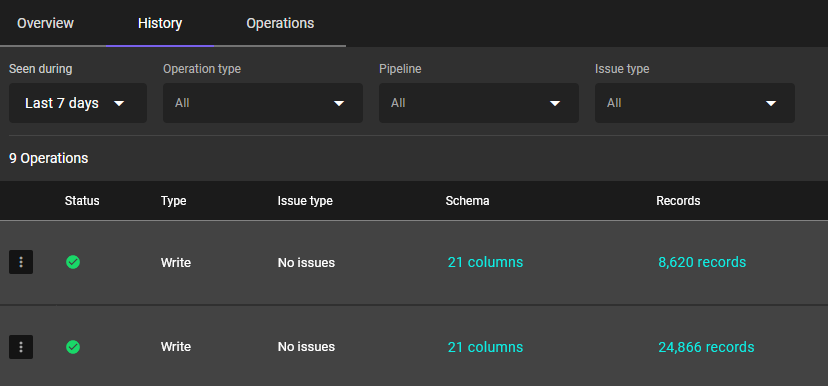
1. Switch to Databand and find the generated alerts.



1. Investigate the issue through **Datasets** view.



Find the dataset, drill down and click on the **History** tab. Notice that the number of records written in the last 2 pipeline runs is different.

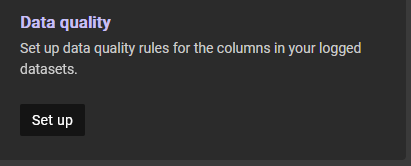


Next, we will add a *custom metric* to monitor data quality and set up an alert for it.

In Databand a *custom metric* is *business logic* that’s implemented in a data pipeline and registered in Databand. Custom metrics can be used to monitor/alert for data quality or simply patterns in data that need to be investigated.

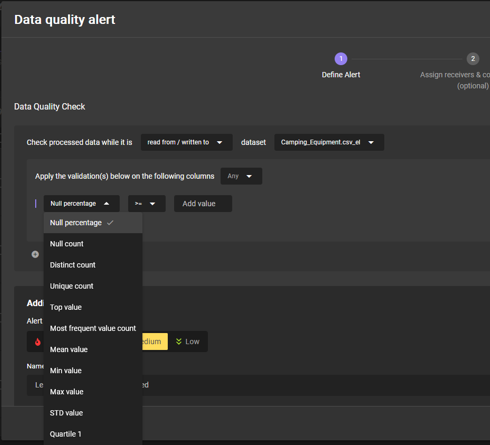
Let’s start by reviewing the built-in metrics for data quality.

1. In Databand navigate to the **Alerts** page. Click **Add Alert -> Data** and select **Data Quality**.



1. Select the *Camping Equipment* dataset and explore quality alerts that you can set up with the options available in the UI.

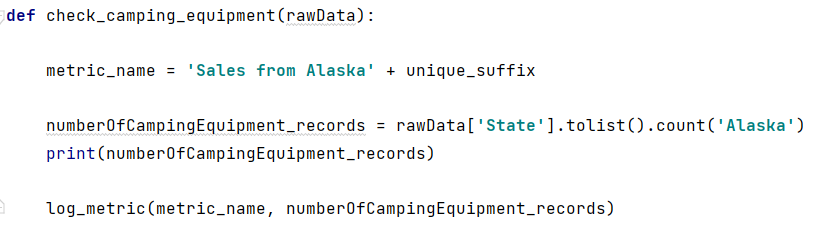
Notice that we can select columns in the dropdown and set up alerts for *null* records, *min* and *max* values, *distinct* *counts*, etc.



While the built-in options cover the most important data quality checks, we may have use cases that require custom logic.

For example, in our retail use case we want to issue an alert if no sales have been reported from a particular state. In order to do this, we will need to create a custom metric in our data pipeline.

We added this function to our data pipeline:



In this example the *log\_metric()* function sends the number of records for the specified state to Databand.

*Note: we hardcoded the state value for simplicity. In a production implementation, it’s possible to make the values configurable so that states are changed without changing the code.*

1. Add the metric logging code to your *SimpleRetailDataPipeline\_with\_Databand* script or open *SimpleRetailDataPipeline\_with\_CustomMetric.py* script (from the *Workshop/Pipelines* folder)

If you’re using *SimpleRetailDataPipeline\_with\_CustomMetric.py,* update the Databand URL, token, and add your initials like you’ve done for other scripts.

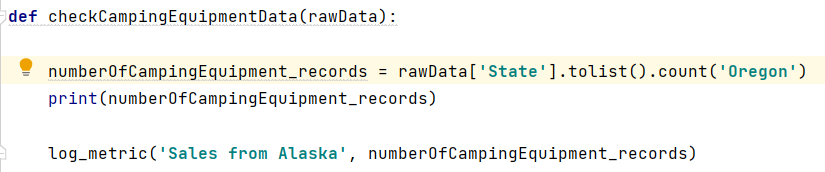
If you’re updating the original script, in addition to adding the function, add the function call to the *prepare\_retail\_data()* function (at the end).

**def** check\_camping\_equipment(rawData):  
  
 metric\_name = **'Sales from Alaska'** + unique\_suffix  
  
 numberOfCampingEquipment\_records = rawData[**'State'**].tolist().count(**'Alaska'**)  
 print(numberOfCampingEquipment\_records)  
  
 log\_metric(metric\_name, numberOfCampingEquipment\_records)

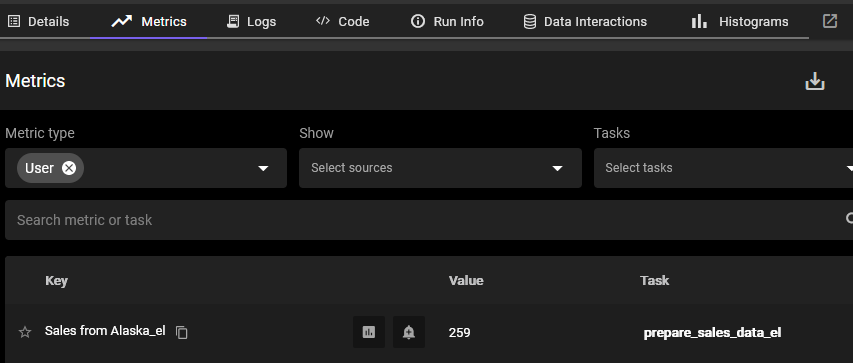


1. Run the updated script several times with different state values (keep the metric names the same).

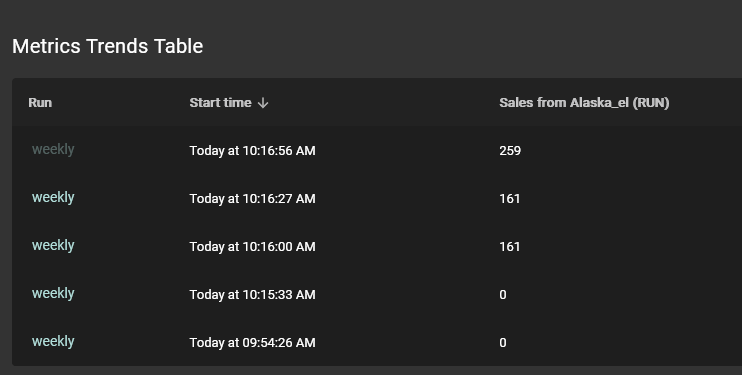
For example, filter for *Alaska* (no records), *Arizona*, and *Oregon*. We are providing different state names just to generate a different number of records in Databand.



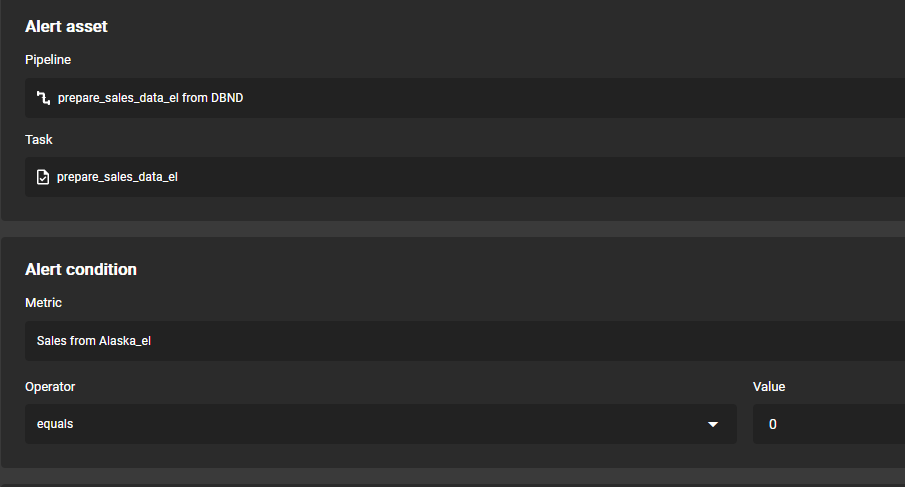
1. In Databand navigate to the **Pipeline -> Runs -> Metrics** page and find *Sales from Alaska* metric.



Next to the metric, click the Show on chart icon, then scroll down to the bottom and notice the number of records from different runs.

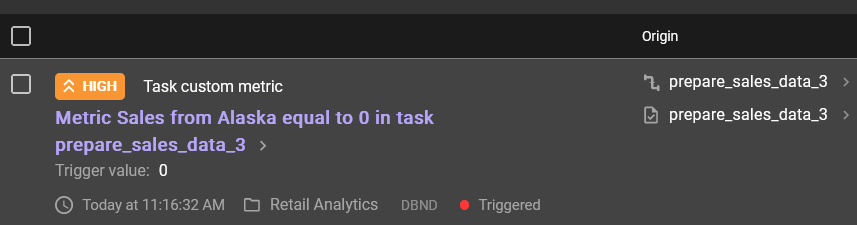


Create an alert for number of records *equals 0*.



1. Change the pipeline to count records from *Alaska* again and run it.

Switch to Databand and find the generated alert.



This concludes introduction to Databand Python SDK.

**Summary:**

In this section you learned how to use Databand SDK to:

* Track Python pipelines in Databand
* Create alerts for data quality
* Create custom metrics.

These tasks are the most frequently used tasks in a production implementation of Databand.

# Part 2: Use Databand SDK

In this section you will use the Databand SDK to configure monitoring for a different Python pipeline. Use the *SimpleRetailDataPipeline\_with\_Databand.py or SimpleRetailDataPipeline\_with\_CustomMetric.py* as an example.

You can use your own Python data pipeline or a pipeline that we created for you: *BankingDataPipeline.py*. The sample pipeline and the data are in the *Workshop/Pipelines* folder.

Complete the following steps to track the pipeline in Databand:

* Install the Databand SDK (*pip* commands)
* Add Databand *import* statements
* Start tracking in the main function that invokes the pipeline steps
* Add *@task* before each Python function to register it as a pipeline step
* Log datasets
* Log a custom metric

We also recommend that you “force errors” and create alerts for various data quality indicators.

# Part 3: Review a PySpark example

In this section you will use the Databand SDK to track execution of a Spark pipeline.

As you know, Cloud Park for Data includes a Spark runtime. Many customers use other distributions of Spark, such as *Databricks*, *Amazon EMR*, and *Google Cloud DataProc*.

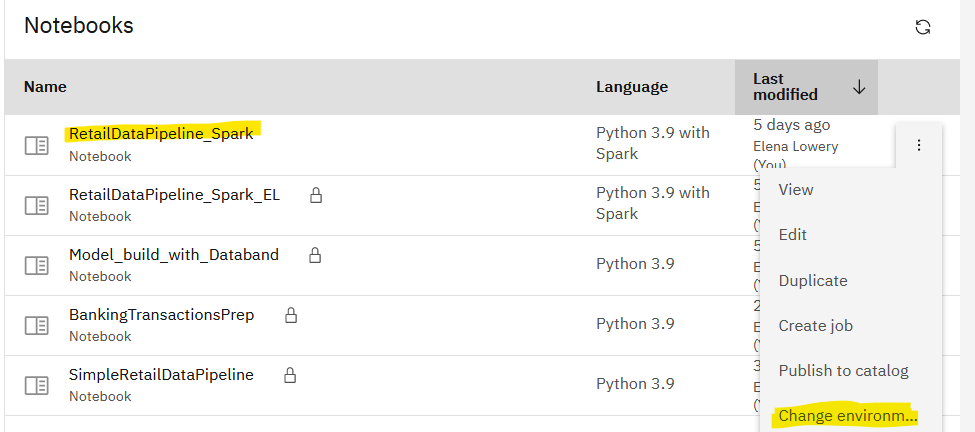
In this section we will review programmatic integration with Spark pipelines that’s similar to Python integration.

***Important note:*** *If a customer is using an external (non-CPD) Spark, then they can enable integration on the Spark cluster level. If this integration is enabled, then Spark pipelines will be automatically monitored (no additional code is required). This applies to starting/stopping tracking and logging datasets, which means that integration with Spark is “no-code” integration. See* [*documentation*](https://docs.databand.ai/docs/installing-dbnd-on-spark-cluster) *for configuration information.*

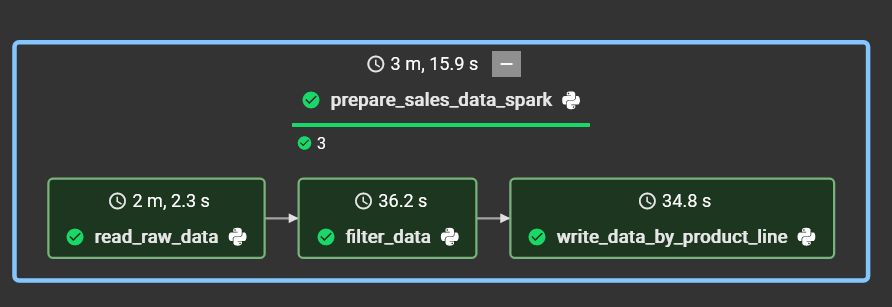
The PySpark example that we provided for this section is a PySpark notebook that performs ETL operations on the same retail dataset as Python example we reviewed in **Part 1**. We recommend that you use Watson Studio withi *Cloud Pak for Data as a Service* (*CPDaaS*) as the runtime environment.

1. Log in to CPDaaS and create a new project.
2. Import the *Retail\_Products\_and\_Customers.csv* from the */Pipelines* folder of the file downloaded from Box.
3. Create a notebook from file: *RetailDataPipeline\_Spark* (in the /*Pipelines* folder). Make sure to set the Environment to *Default Spark 3.3 &* ***Python 3.9 runtime****.*

If you forget to select the Spark environment, you can do it later by stopping the notebook and changing the default environment from the Project view (select the vertical ellipses menu next to the notebook). To unlock the notebook, click on the lock icon.

**

1. Follow the update instructions in the notebook:
   1. Generate the project token
   2. Replace code to read data from your object storage
   3. Provide Databand URL and token
   4. Rename the pipeline
2. Run the notebook and check Databand for pipeline status.



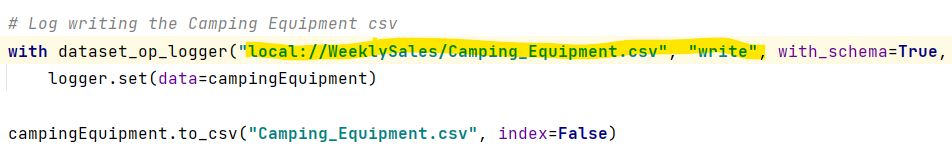
1. If you wish, introduce errors in the notebook.

# Part 4: Create pipelines with lineage

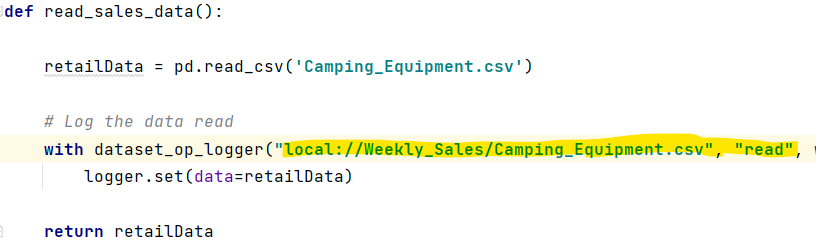
In this section you will run the pipelines that create a lineage graph.

Lineage in Databand is created when the same dataset is used by multiple pipelines. Datasets are identified by the value that’s provided to the *logger* function call.

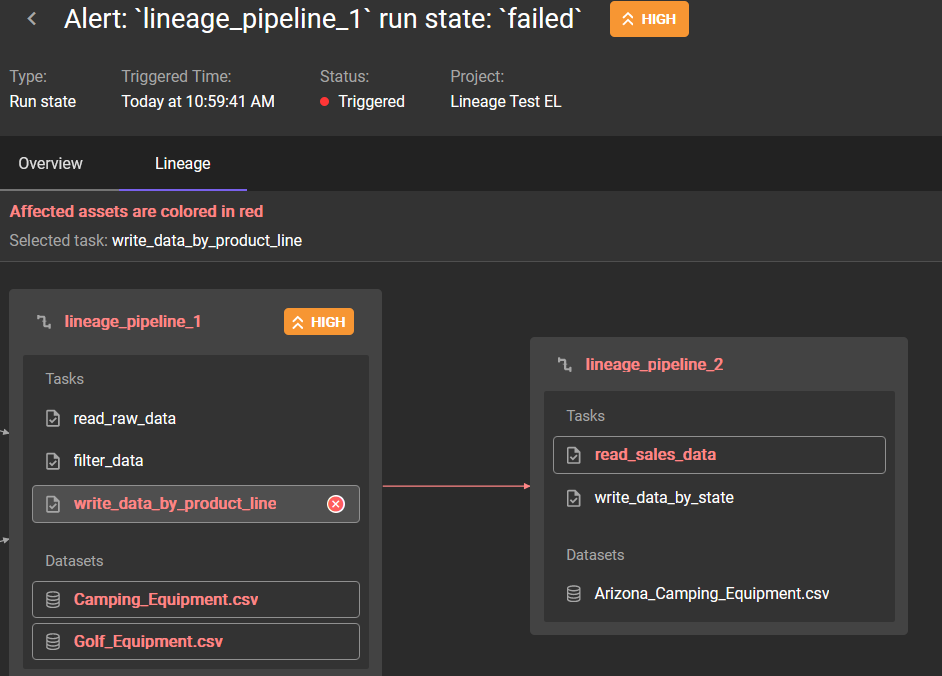
For example, pipeline 1 writes the *Camping\_Equipment.csv* file:



Pipeline 2 reads the same file:



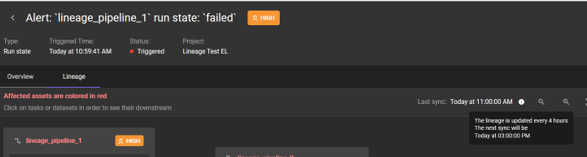
In Databand, the graphic lineage is represented like this:



Databand is able to create this lineage because we provided the same value, *local://Weekly\_Sales/Camping\_Equipment.csv,* to the logger.

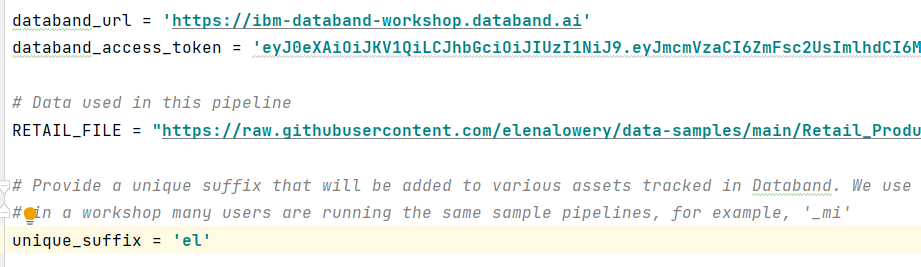
While this value can be *any string*, the recommendation is to make it as close to the actual source path as possible to simplify troubleshooting. We chose “local” to indicate that in our example we are reading from the local data source. If your data source is in DB2, you can include database name, schema, and table as a part of the string passed to the logger. For example “*db2://bludb/sales/camping\_equipment*”.

Unlike pipeline steps and dataset statistics, lineage is not determined at the time of a pipeline run. Lineage jobs run automatically every 4 hours, and read/write operations on datasets for the past 7 days are evaluated. At this time these settings are not configurable, however, they are displayed on the **Lineage** tab.



In this lab we will run 2 pipelines and check the lineage a few hours later.

1. Load *Lineage\_Pipeline1* and *Lineage\_Pipeline2* Python scripts to your Python environment. We will be using the same csv data source as in the *SimpleRetailDataPipeline\_with\_Databand* script, so we don’t need to copy it.
2. In the beginning of both scripts
   1. Provide the Databand URL and token
   2. Update your initials

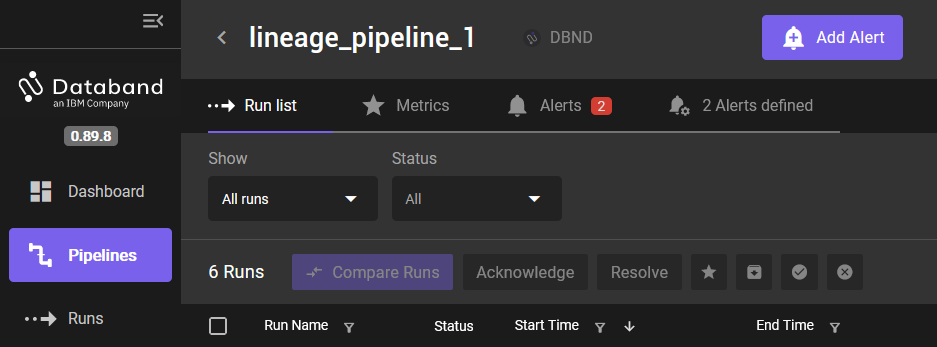


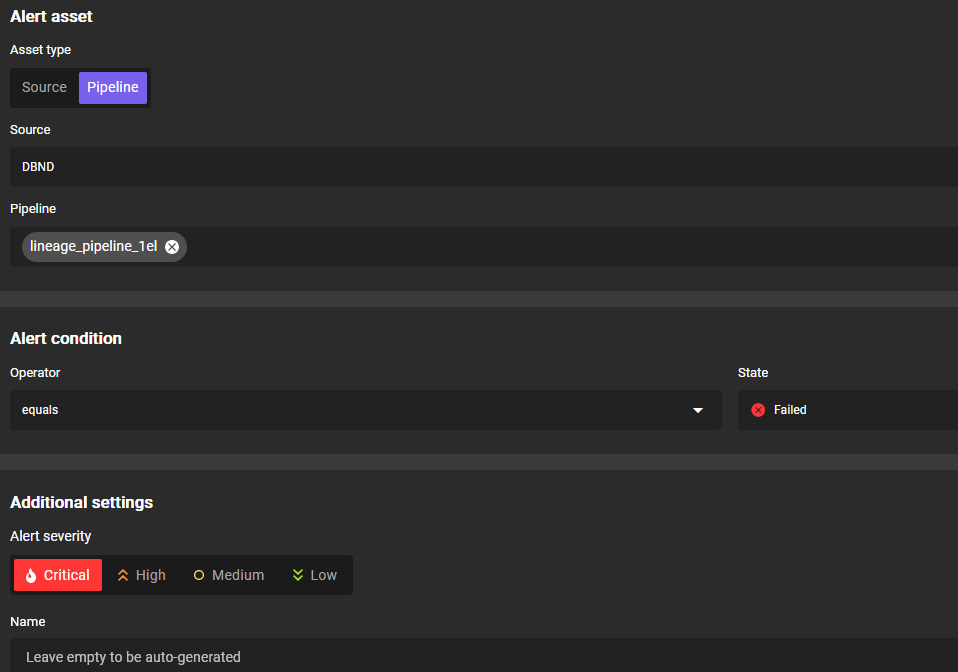
1. Run *Lineage\_Pipeline1* 3-5 times, then *Lineage\_Pipeline2 (*also3-5 times).

While the pipelines will be shown on the **Pipelines** page, currently the Lineage graph is accessible through the **Alerts** page only, which means that we have to create an alert for *Lineage\_Pipeline1*.

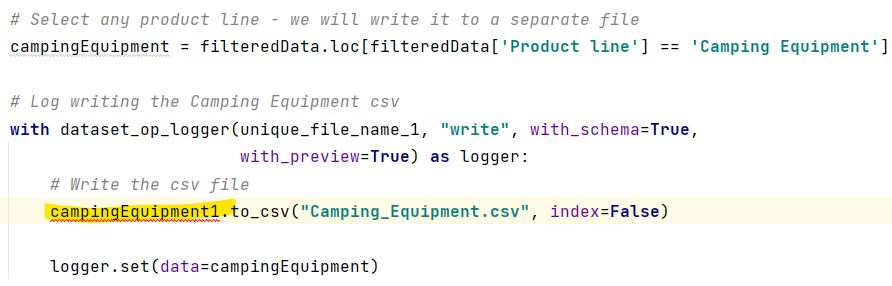
You can create an alert for a successful run, but in most cases data engineers want to investigate lineage for failed runs. We recommend that you “force” an error in the last step of *Lineage\_Pipeline1.*

1. In Databand **Pipelines** tab find your pipeline and create an alert for pipeline status *failed*.



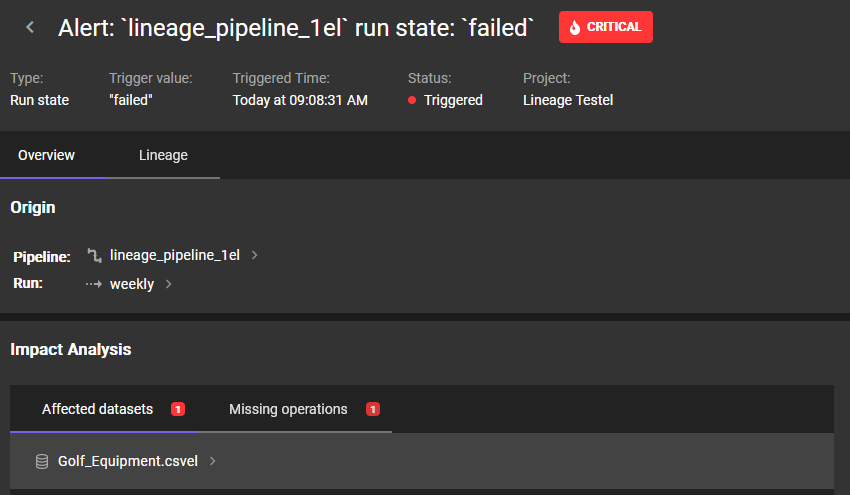


1. We will use the same approach as in **Part 1** to force an error in *LineagePipeline1* – we will change the code to the name of the dataframe that does not exist (*campingEquipment1*) in *write\_data\_by\_product\_line* function.

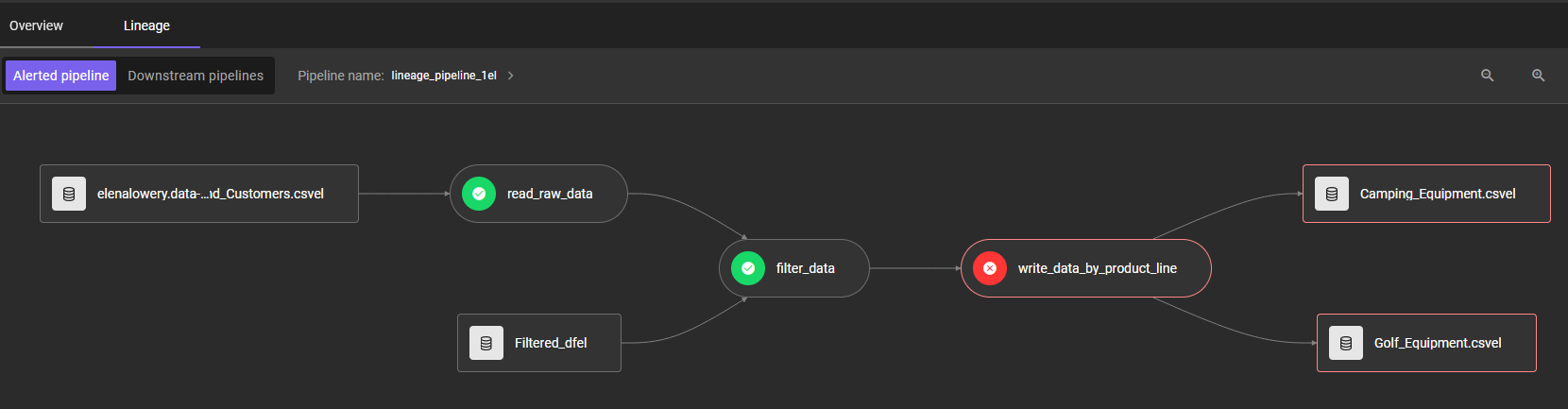


1. Run the modified pipeline to generate an alert.

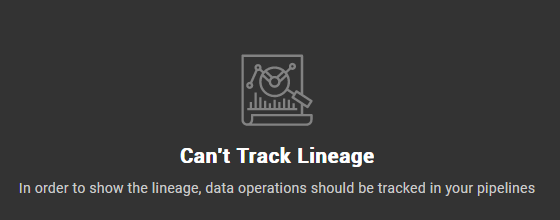
Drill down to the alert and notice that we have 2 tabs under **Impact Analysis: Affected datasets** and **Missing operations.** Since in our code we introduced the error before the *Golf\_equipment* write operation, it’s shown as the affected dataset and a missing operation.



If you click on the **Lineage** tab, you will see the lineage for the current pipeline.

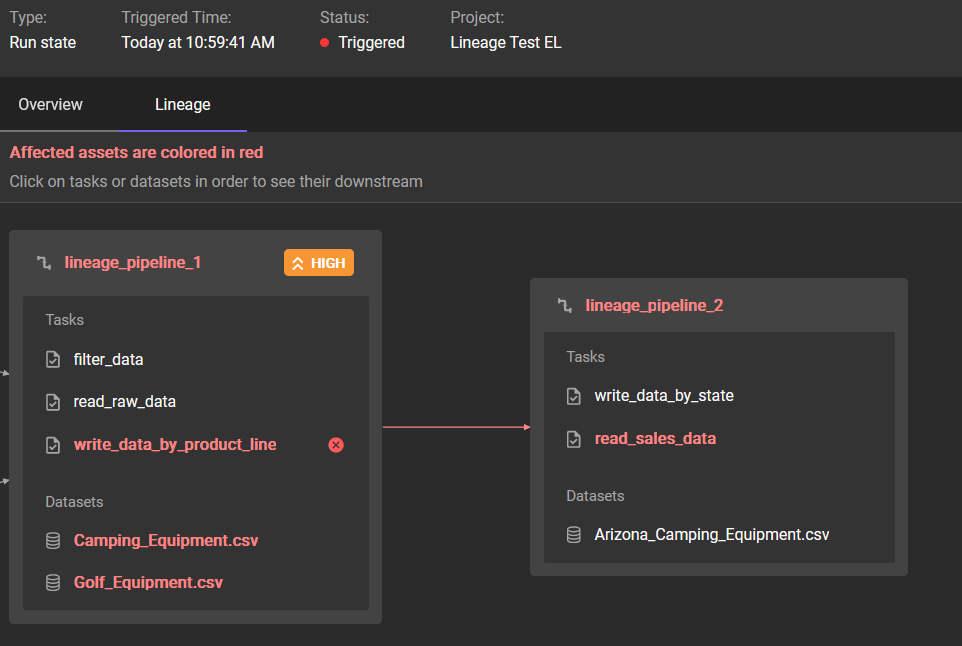


If you select the **Downstream pipeline** tab, you may see a message *“Can’t Track Lineage”.* However, after the automatic lineage jobs run, this tab as well as the Lineage tab will be automatically updated (including for the alerts that were already issued).



Check back on the alert and the lineage view after 1-3 ours to verify that lineage is displayed.

Lineage output:



You have completed the Lineage section of the lab.

# Part 5: Monitoring DataStage pipelines (optional)

Databand can monitor the following properties of DataStage NextGen pipelines:

* Run and task state alerts (e.g. running, successful, failed, etc.)
* Run duration alerts
  + anomaly detection
  + percent ranges (e.g. duration within 20% of 100 seconds)
  + basic comparison operators (e.g. duration > 100 seconds)
* Schema changes for inputs and outputs
  + new columns added
  + old columns removed
  + datatypes of existing columns changed
* Record counts for inputs and outputs
  + anomaly detection
  + percent ranges (e.g. record count within 20% of 100,000 rows)
  + basic comparison operators (e.g. record count > 100,000 rows)

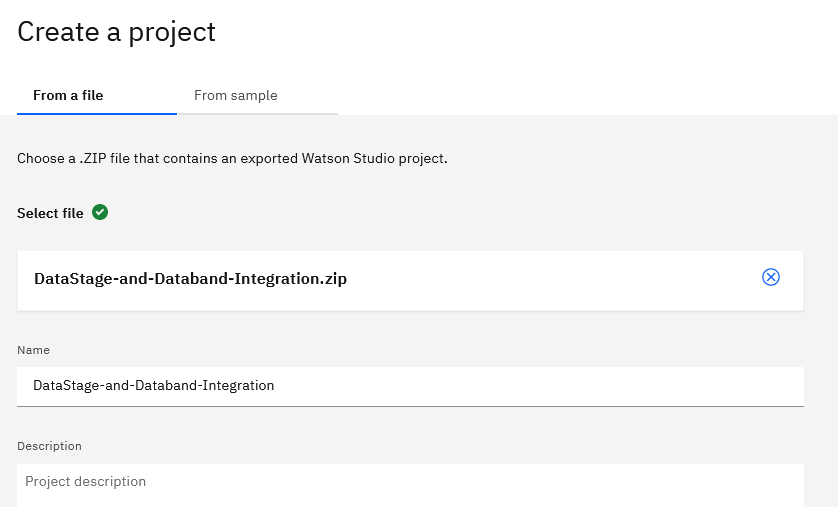
For latest information about DataStage integration, see [documentation](https://docs.databand.ai/docs/tracking-datastage).

In this section we will monitor DataStage pipelines that are running in **Cloud Pak for Data as a Service** (**CPDaaS**). We will use the sample DataStage pipeline created as a part of the Multicloud Data Integration [tutorial](https://dataplatform.cloud.ibm.com/docs/content/wsj/getting-started/df_data_integrate.html).

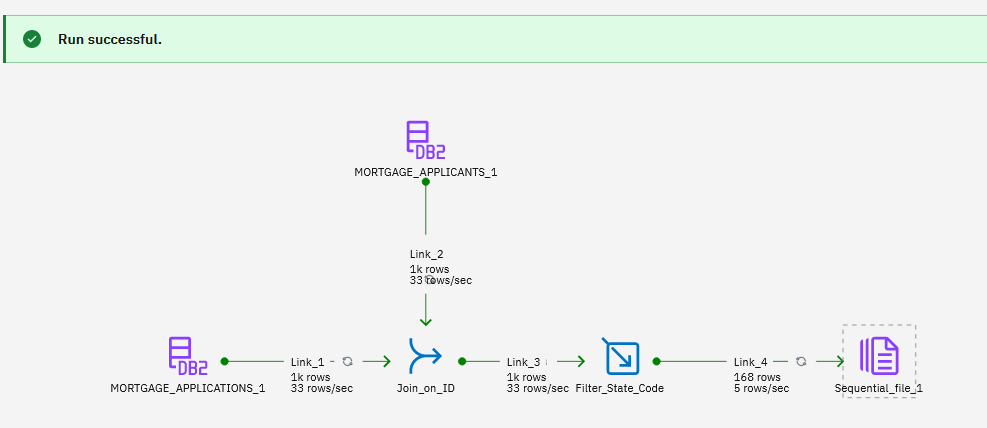
You can either choose to complete the [tutorial](https://dataplatform.cloud.ibm.com/docs/content/wsj/getting-started/df_data_integrate.html) to create the DataStage pipeline, or you can import the simple pipeline that’s provided with the tutorial. Completing the tutorial to create your own pipeline may take up to 1 hour.

If you want to import the pipeline:

1. In **CPDaaS** create a project from file: *DataStage-and-Databand-Integration.zip* located in the */CPDaaS Projects* folder of the file downloaded from Box.



1. Open the DataStage flow and run it to make sure that it can access data.



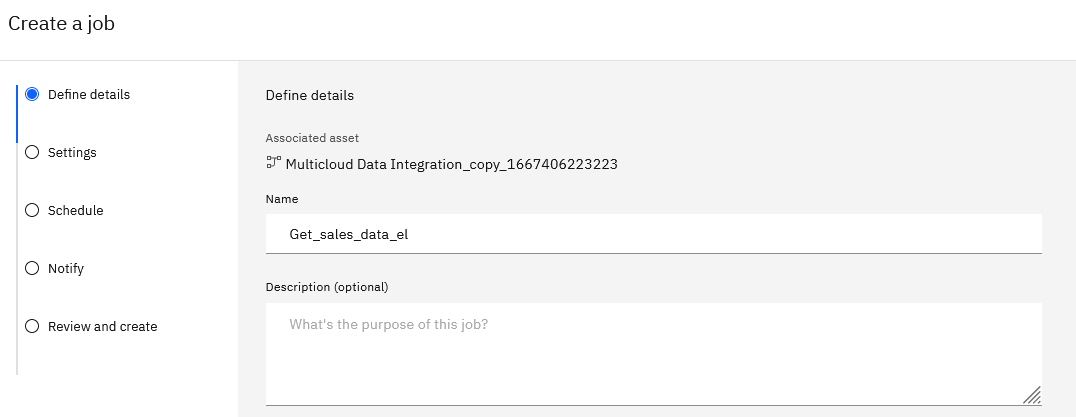
If you get errors while running the pipeline, verify that you can access data sources (use preview in the pipeline or in the project).

1. **Optional:** If you’re an expert in DataStage, we recommend that you change the data sources to read from your own instance of DB2 Cloud. Csv files for *Mortgage Applicants* and *Mortgage Application* tables can be found in the /Data folder downloaded from Box.

If you’re using the DataStage pipeline tutorial or the provided CPD project, you will be using a *shared* data source. While it will work, since all users access the same data source, lineage graphs will show multiple pipelines. If you switch to your own data source, you will see lineage for your pipeline only.

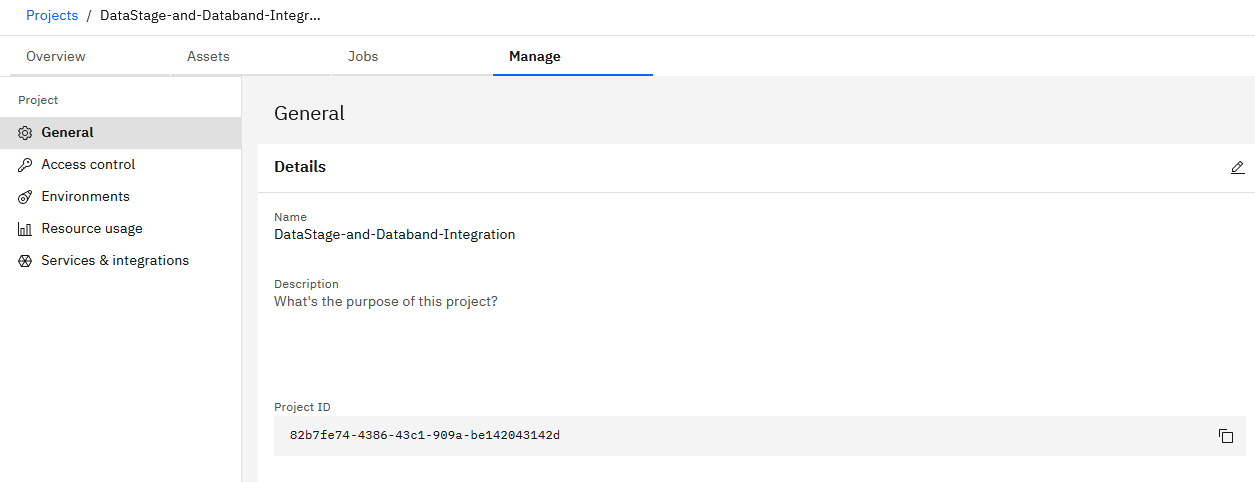
1. In CPDaaS create a job for the DataStage pipeline:

* In the **Assets** view of the project use the vertical ellipses menu next to the pipeline to select **Create Job**.
* Provide a unique name for the job, for example, add your initials.
* Accept all default settings and select **Create** on the last screen.

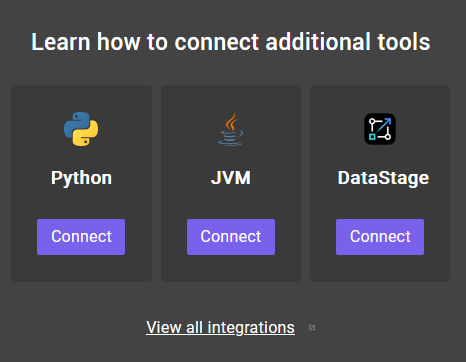


Next, we will configure DataStage/Databand integration. For integration with DataStage on CPDaaS, we need to get our *Cloud API key* and *project id*.

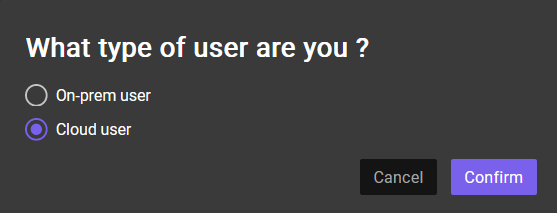
1. Log in to your IBM Cloud account and create an API key following these [instructions](https://cloud.ibm.com/docs/account?topic=account-userapikey&interface=ui#create_user_key). If you already have an IBM Cloud API key, you can reuse it.
2. Capture the project id. You can find it on the **Manage** tab of your project.

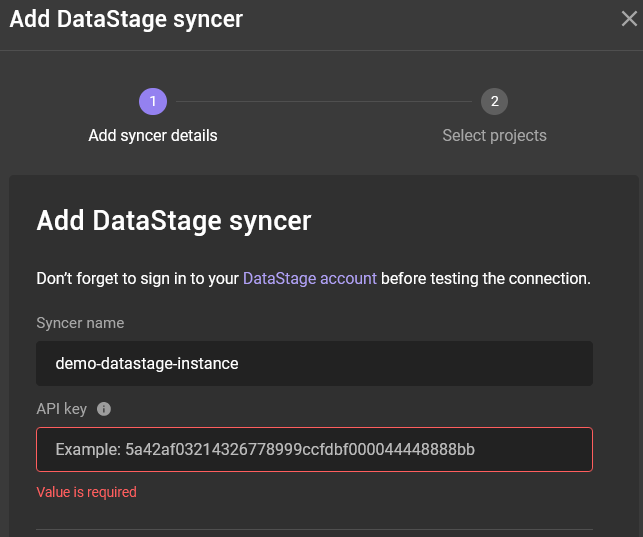


1. In **Databand** click on **Integrations**, then **DataStage -> Connect**.

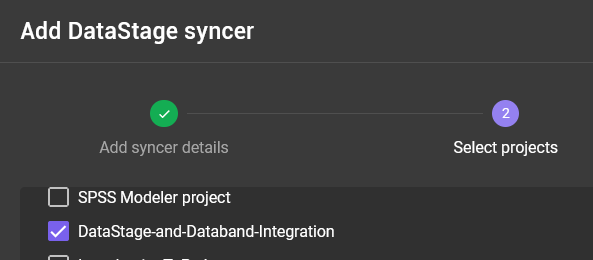


1. Provide the required configuration information.
   * *Source name:* use a unique name, for example, add your initials
   * *Cloud API key*

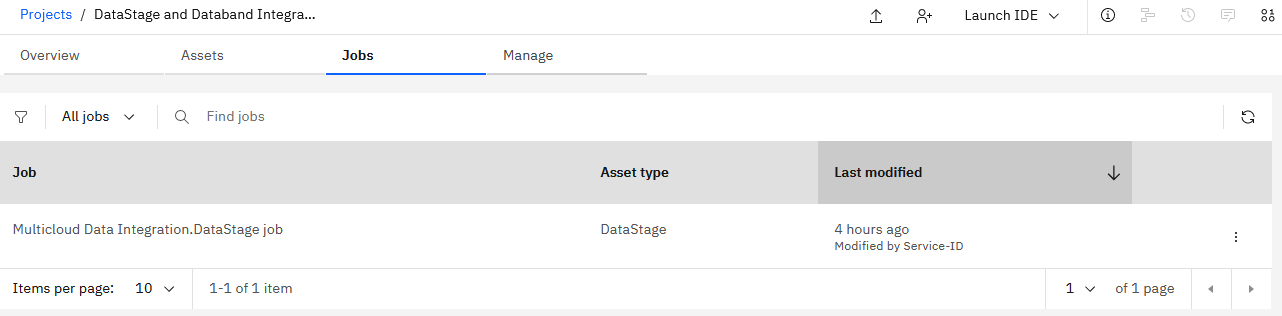


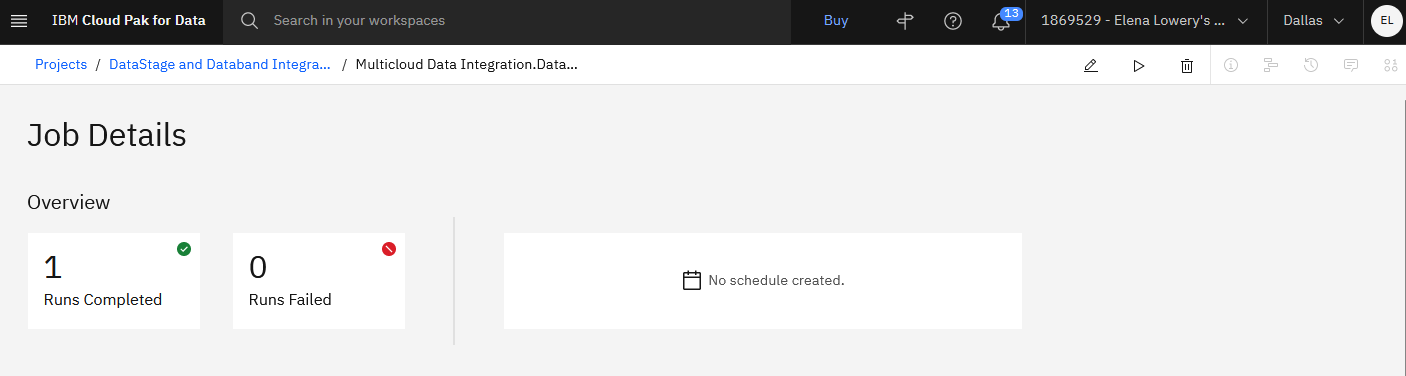


* On the next panel select the project that contains your pipeline.
* Save the configuration and return to IBM Cloud to run the DataStage pipeline.



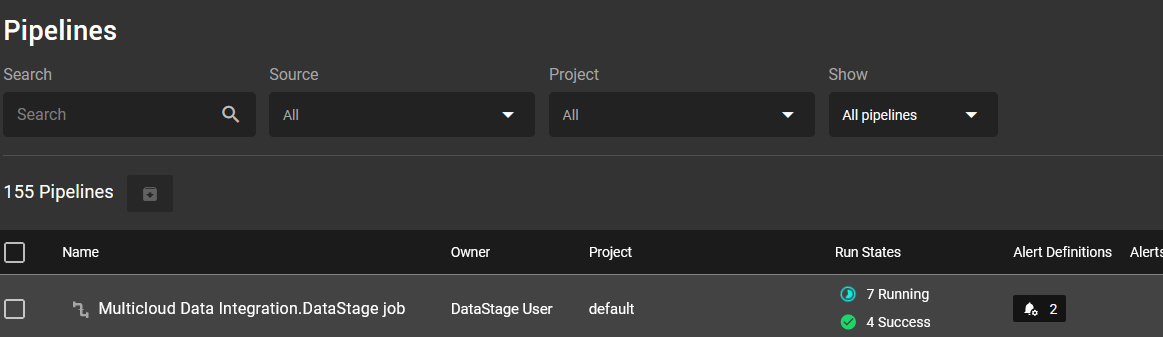
1. Navigate to the pipeline job in the project and run the pipeline.





1. Switch to the **Pipelines** view in Databand and review the run.

Explore pipeline steps and various metrics that are collected by Databand.



1. ***Optional:*** create an alert for the duration (pick a duration less than the first pipeline run). After you get an alert, explore lineage.

You have completed the DataStage integration section of the lab.

# Part 6: Monitoring other pipelines (optional)

While Databand is primarily used to monitor data pipelines, the SDK can also be used to monitor Machine Learning pipelines.

In addition to monitoring pipeline status, logging datasets that are used for model building and scoring, as well as capturing some statistics for models (such as feature importance or model accuracy) can be useful for creating different types of alerts. For example:

* Schema change in a model building pipeline
* Schema change in a model scoring pipeline
* Change in feature importance
* Values below threshold in a model building pipeline.

We created a sample notebook that demonstrates how the Databand SDK can be used in a model building pipeline.

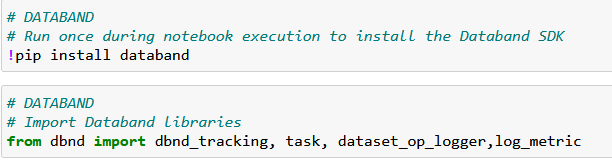
In this section you will review the notebook.

1. Log in to **Cloud Pak for Data as a Service** (**CPDaaS**) or Anaconda or another IDE supporting notebooks. Create a new project or open an existing project.
2. Create a notebook from file: *Model\_build\_with\_Databand* (in the /*Pipelines* folder downloaded from Box).

We do not need to import data because it’s read from a Git repo.

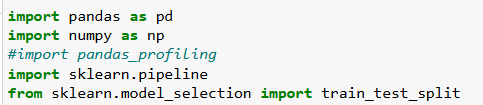
1. Review the notebook and make the changes as provided in the notebook instructions.

To understand the changes that were made for tracking, look for the # DATABAND tag



1. Run the notebook and check results in Databand.

*Note: if you get an error on import of pandas\_profiling library, you can comment it out*



1. If you wish, set up alerts for model accuracy threshold (make it less than 0.75).

# Part 7: Monitoring pipelines in Databricks

Databand can be used to monitor pipelines in *Databricks*.

Databricks is a comprehensive Data and AI platform that contains several data manipulation, data governance, and data science tools used in one of the cloud providers (Amazon, Azure, GCP), where it’s available aaS.

For integration with Databand, we will focus on the tools and features that support Python and Spark.

In Databricks customers typically have the following types of pipelines:

* *Spark scripts* (PySpark or Scala) that are submitted to a Databricks Spark cluster. The scripts are developed in a desktop IDE (for example, Visual Studio or PyCharm)
* *Notebooks* that are developed in Databricks UI and run in either Python or Spark environment in Databricks.

Scripts and notebooks can be scheduled to run as jobs, which are sometimes referred to as Workflows in Databricks. Customers also use various orchestrators, such as Airflow, to invoke scripts and notebooks that run on Databricks.

In general, integration with assets that run on Databricks is the same as covered in earlier sections of the lab:

* With the SDK in Python and PySpark scripts/notebooks
* Via Airflow configuration.

Since Databricks includes Spark, an additional integration option for Databricks is by using the Databand *Spark Listener*.

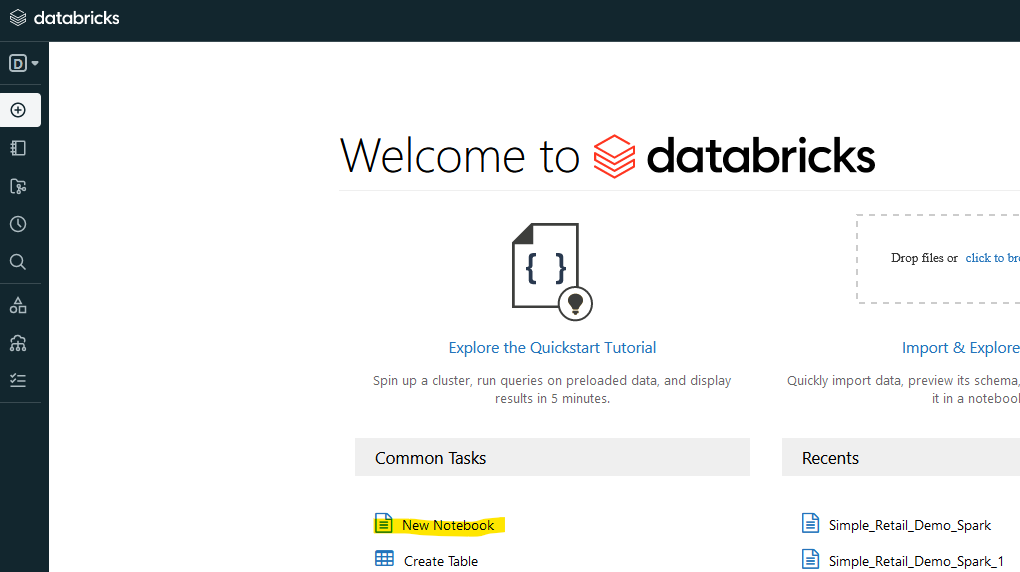
**Summary of Databand/Databricks integration:**

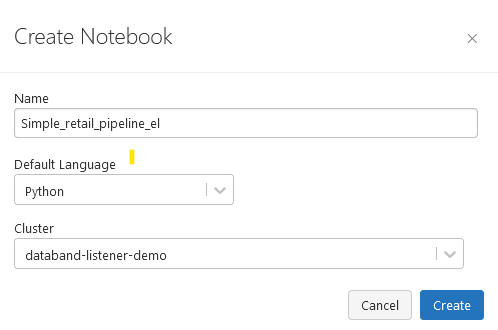
|  |  |  |
| --- | --- | --- |
| **Databricks artifacts** | **Integration** | **Notes** |
| 1. Spark *scripts* submitted directly to Databricks with the Spark Listener configured | No code for dataset tracking. SDK is needed to monitor pipeline status and steps | Dataset metadata is captured automatically by the Spark Listener |
| 1. Spark scripts invoked with Airflow | No code. Optional SDK | If the SDK is not used, then datasets are not tracked |
| 1. PySpark/Scala *notebooks* running in Databricks *with Spark Listener configured* | Databand SDK: start tracking and tasks | Dataset metadata is captured automatically by the Spark Listener |
| 1. Python Notebooks running in Databricks | Databand SDK | Standard SDK integration |

First, we will use the SDK to monitor Python and PySpark pipelines that are implemented in notebooks.

*Note: If you do not have access to Databricks environment, you can watch* [*this video*](https://ibm.box.com/s/7u9cyhxno12wvgt6dx72tk71shgm9a9t) *instead of completing sections 1-7.*

1. From the landing page in *Databricks* create a new notebook and name it *Simple\_Retail\_Pipeline\_Databricks\_<your\_initials>.*

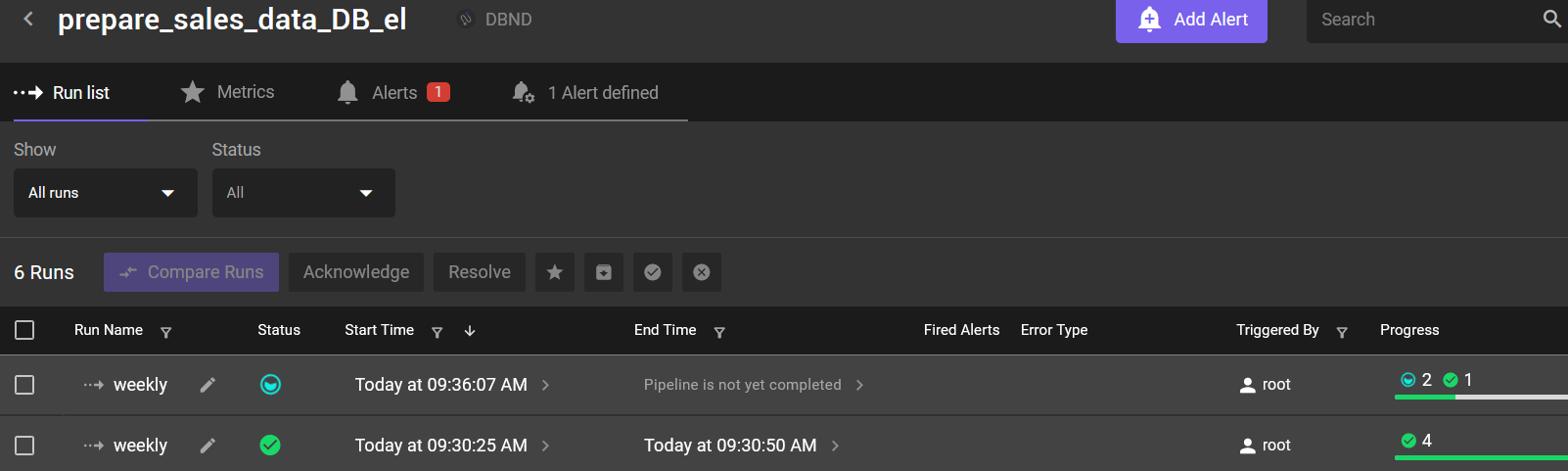




1. In the Notebook UI select **File -> Import** and import the *Simple\_Retail\_Pipeline\_Databricks.ipynb* notebook from the *Workshop*/Pipelines/*Databricks* folder (downloaded from git).
2. Review the notebook. Notice that the notebook contains the same code as the Python script and the Watson Studio notebook that we used earlier in the lab.
3. Add the Databand cluster URL and token. Update your initials so that you can find your pipeline in the Databand UI.



1. Run the notebook.
2. In the Databand UI, find your pipeline and review the captured metadata.



1. If you wish, set up alerts and repeat steps for causing errors as you’ve done earlier in the lab.

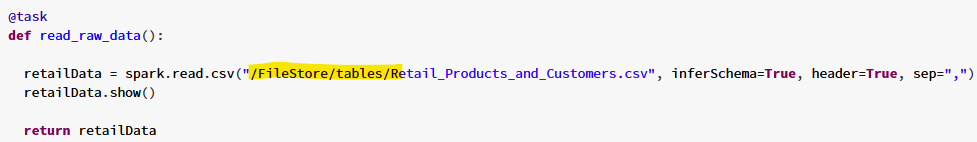
Next, we will run a Spark notebook that *does not use* the Spark Listener. While using the Spark Listener provides the “no code” monitoring of Spark pipelines (no changes are required in Spark scripts), we wanted to show you an example which you can use for testing in case you run into issues with the Spark Listener.

*Note: The advantages of using the Spark Listener is no-code integration and no impact of performance. The Spark Listener captures row-level metadata and schema, it does not capture column-level statistics. If you need column-level statistics, use the SDK. Also, the Spark Listener does not track pipeline execution status and completion of each task. If Airflow is used, then these items are captured with Airflow integration. If it’s not used, then use can use the SDK to track pipeline execution.*

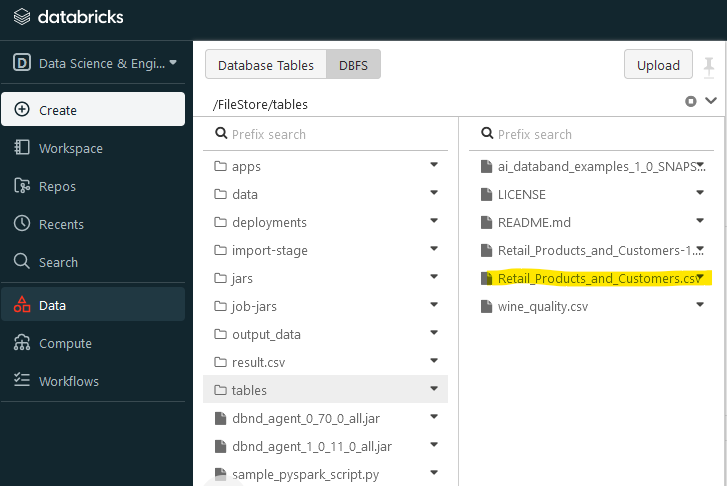
If you are in an instructor-led workshop, ask your instructor for the name of the cluster that does not have the Databand Spark Listener configured. If you’re using your own instance of Databricks, then you do not have the Spark Listener configured yet.

1. Load the data that will be used by the sample Spark notebook:

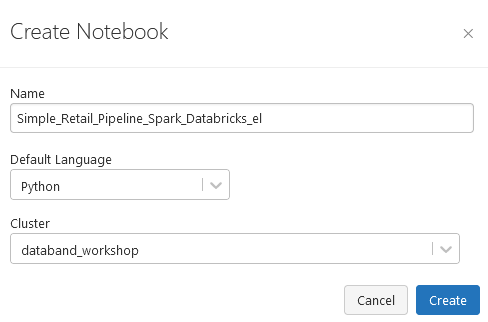
* The notebook code reads from this location:

**

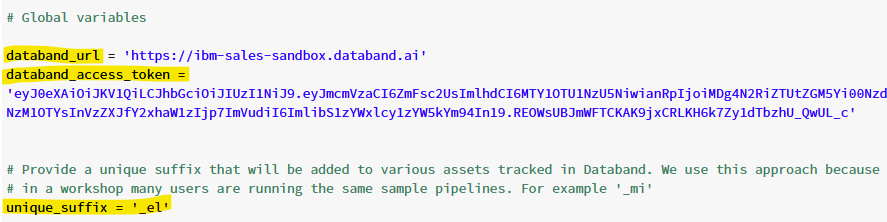
* You can either load the .csv file from the *Workshop/Data* folder (downloaded from Box) to this location or upload it to a different directory and update the notebook

**

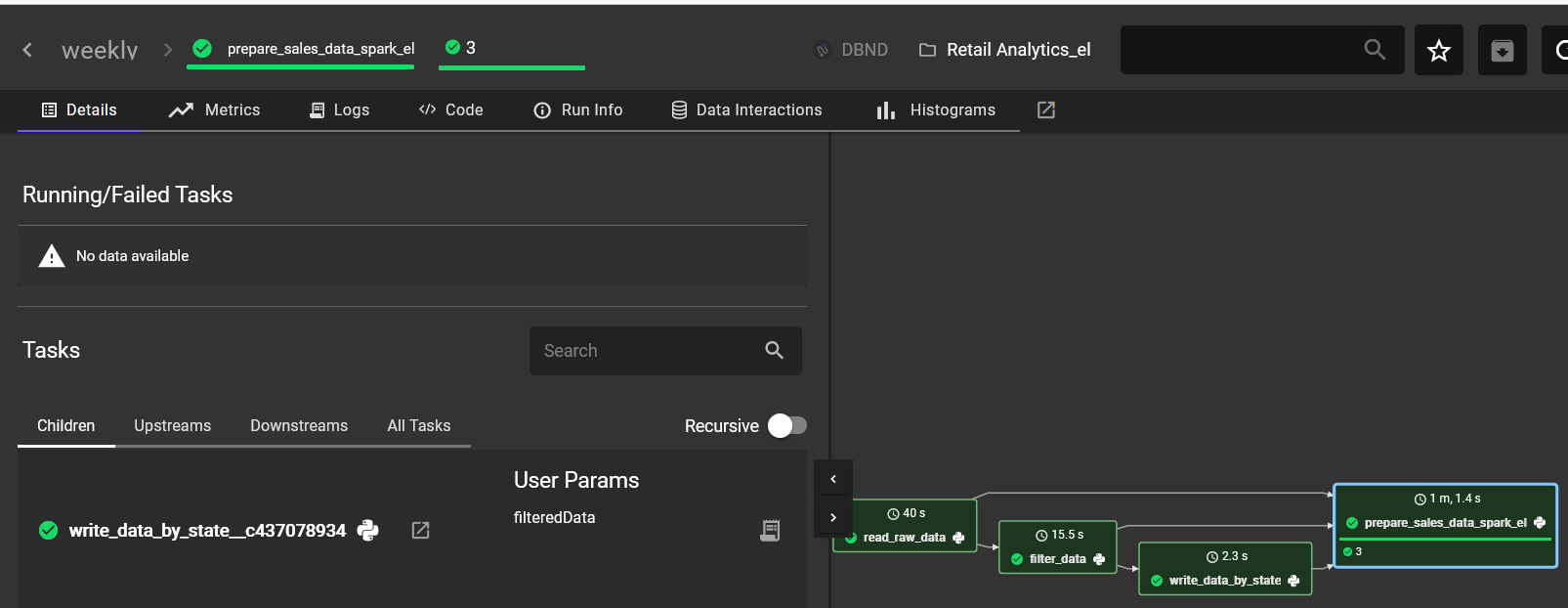
1. From the landing page in *Databricks* create a new notebook and name it *Simple\_Retail\_Pipeline\_Spark\_Databricks\_<your\_initials>.*

**

1. In the Notebook UI select **File -> Import** and import the *Simple\_Retail\_Pipeline\_Databricks\_Spark.ipynb* notebook from the *Workshop*/Pipelines/*Databricks* folder (downloaded from git).
2. Review the notebook. Notice that the notebook contains the same code as the Watson Studio Spark notebook that we used earlier in the lab.
3. Add the Databand cluster URL and token. Update your initials so that you can find your pipeline in the Databand UI.



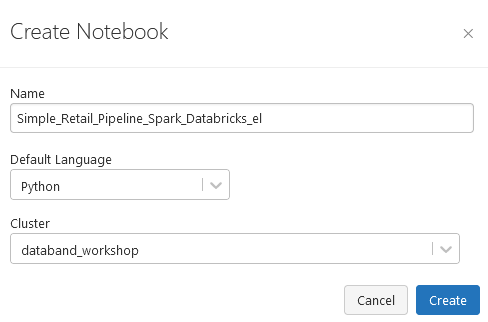
1. Run the notebook.
2. In the Databand UI, find your pipeline and review the captured metadata.



Next, we will review observability for pipelines that use the *Spark Listener*, and not the SDK for tracking pipeline execution and metadata.

If you’re working on your own cluster, set up the Spark Listener using instructions in [documentation](https://docs.databand.ai/docs/installing-dbnd-on-spark-cluster). If you do not have access to a Databricks cluster, you can watch this [video](https://ibm.box.com/s/0bu74i25ob4ictm9mwv95ehu6us4ekid).

1. From the landing page in *Databricks* create a new notebook and name it Simple\_Retail\_Pipeline\_Spark\_Listener*\_<your\_initials>.*

**

1. In the Notebook UI select **File -> Import** and import the *Simple\_Retail\_Pipeline\_Spark\_Listener.ipynb* notebook from the *Workshop*/*Databricks* folder (downloaded from Box).
2. Review the notebook. Notice that this notebook contains less code compared to the previous example:
   * We do not need to provide Databand URL and token because they are provided on the cluster level
   * We do not need to log datasets because they are logged automatically by observing Spark I/O operations.

When using PySpark/Scala notebooks in Databricks we still have to use some features of the Databand SKD:

* The *@task* decorator before each function in order for each step to be displayed as a separate task in Databand
* The *dbnd\_tracking* wrapper before we invoke the pipeline

1. Run the notebook.
2. In the Databand UI, find your pipeline and review the captured metadata.

**You have finished testing Databand integration with Databricks.**

# Part 8: Integration with Airflow – instructor led

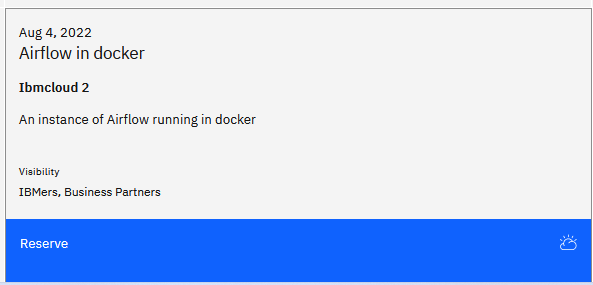
In this section you will learn how to integrate Databand with Airflow. While you do not need to be an Airflow expert, it’s helpful to understand the key concepts and usage. If you’re new to Airflow, take the time to review the resources referenced in the Databand [learning path](https://ibm.box.com/s/h7k9ewjgw3zq8fkn0snlgknfmry2pcsr).

Databand supports various deployments of Airflow, such as *Apache Airflow*, *Astronomer*, *Amazon Managed Workflows*, and *Google Cloud Composer*. See [documentation](https://docs.databand.ai/docs/tracking-airflow-dags) for more information.

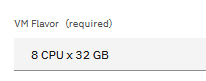
When we integrate Databand with Airflow, instead of configuring integration with Databand for each pipeline, we configure integration with the entire cluster. We can set up cluster-level alerts for pipeline status, duration, and schema changes. We also have to add less code to our pipelines – we do not have to include code to start and stop tracking (dataset tracking code is still needed).

For this exercise we will use Apache Airflow that was pre-installed and partially configured for Databand (because we are using it as a demo instance in TechZone). We will explain configuration, however, if you would like to complete configuration from scratch, you can delete “pre-configured” assets or you can set up a new Airflow instance in docker desktop or any environment of your choice. See **Appendix A: Airflow Installation** for more details.

1. Provision the pre-installed Airflow instance from the [Data Observability TechZone page](https://techzone.ibm.com/collection/2022-data-and-ai-tech-sales-resources-for-data-observability#tab-0).



When provisioning the instance, make sure to select *8 vCPUs* in the *VM Flavor* dropdown.



1. After you receive an e-mail that the VM has been provisioned, navigate to **My Reservations** page in TechZone.
2. Review the [*VM readme*](https://ibm.box.com/s/27n8qfy0v1aul8hvqxgw4qik1zejsjv5) file: it explains how to connect to the provisioned Airflow cluster from your laptop.

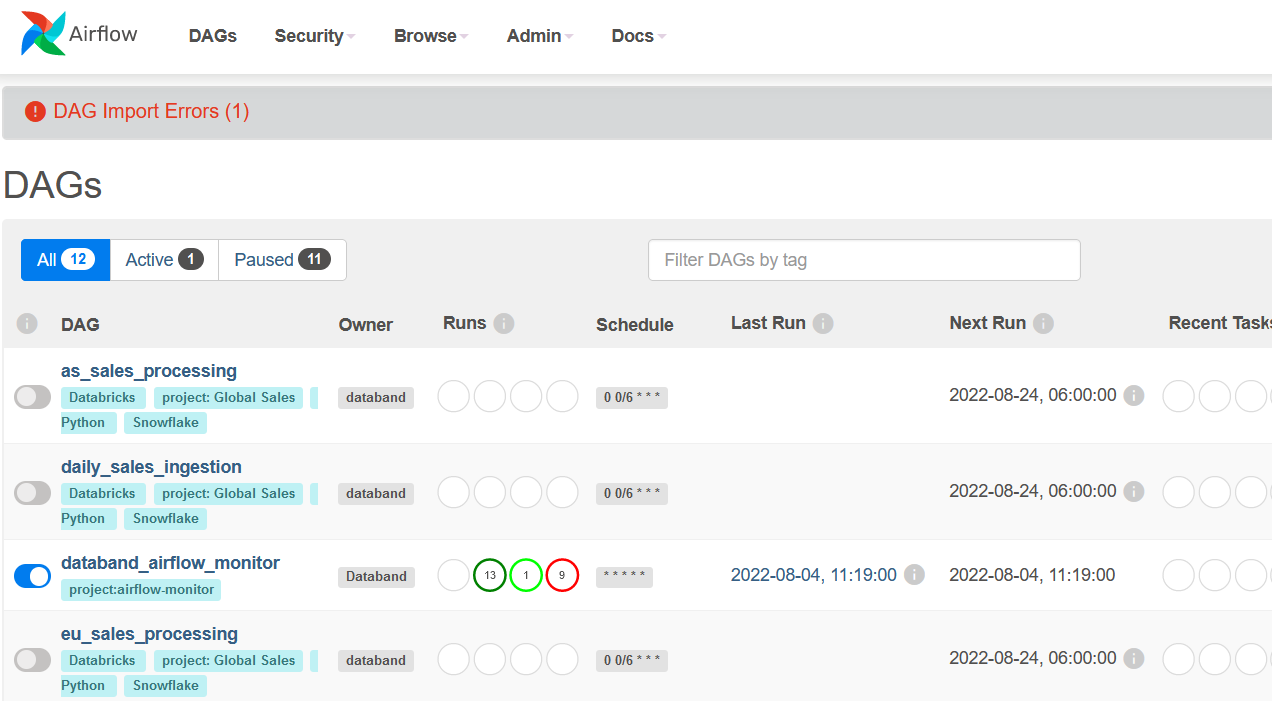
Some considerations:

* Airflow is installed in docker in a Linux VM
* This installation requires port forwarding, and it’s specific only to the Airflow installation in TechZone (customers do not need to perform these steps)

1. Log in to the Airflow instance from your laptop.

<http://localhost:8082/home>

Log in userid: *databand*, password – *databand*.



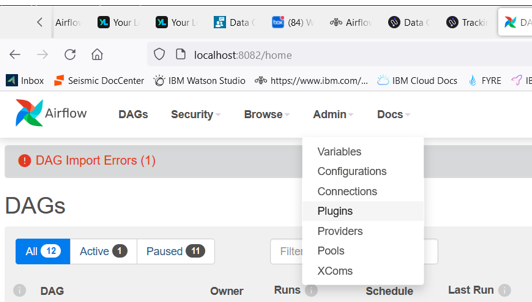
The following steps need to be performed when configuring an Airflow to be monitored by Databand:

* + Install the Databand SDK in the Airflow cluster
  + Add the Databand monitoring DAG (provided by Databand)
  + Configure a syncer in Databand UI. When creating a syncer, capture the connection details (will be entered in the next step)
  + In Airflow, create a connection to Databand, and paste the details from the previous step
  + Enable the monitoring DAG in Airflow (disabled by default)
  + Test connection to Airflow.

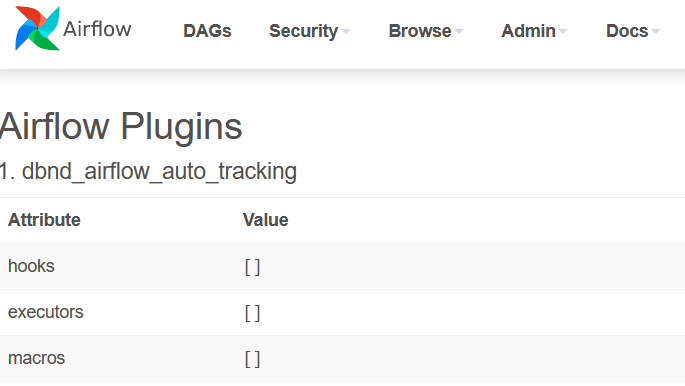
These steps are captured in Databand [documentation](https://docs.databand.ai/docs/tracking-airflow-dags).

In the demo cluster we completed steps *1* and *2*. Let’s review the outcome of completing these steps.

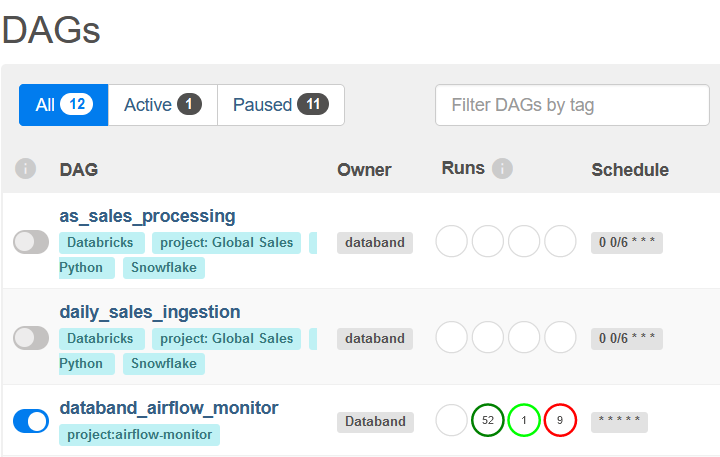
1. In Airflow navigate to **Admin -> Plugins**.



Notice that the Databand tracing sdk is showin as one of the plug-ins.



1. Navigate to the main admin page and find the Databand DAG – *databand\_airflow\_monitor*.



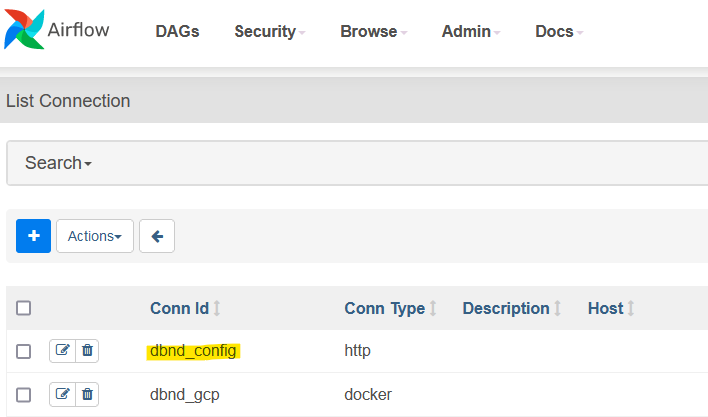
1. Drill down to the DAG and click on the **Code** section.

Notice that the code is generic, it does not contain configuration information to connect to the Databand cluster. However, the comment references that connection id *dbnd\_config*. This id contains the connection information.



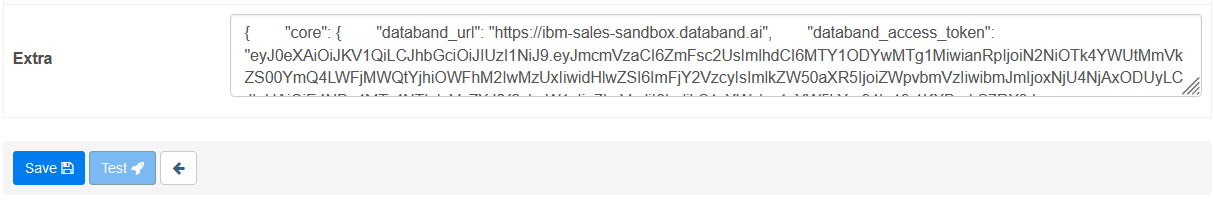
Let’s review the settings of the connection id.

1. In Airflow select **Admin -> Connections**.

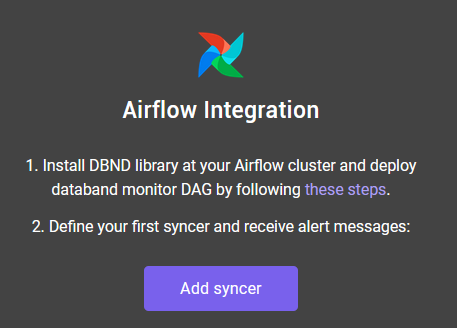


1. Click the **Edit** icon (pencil) next to the connection.

Scroll down to the **Extra** section. Value for this section is created in Databand after we configure a syncer. We will repeat this step in the lab to get a better understanding of configuration.



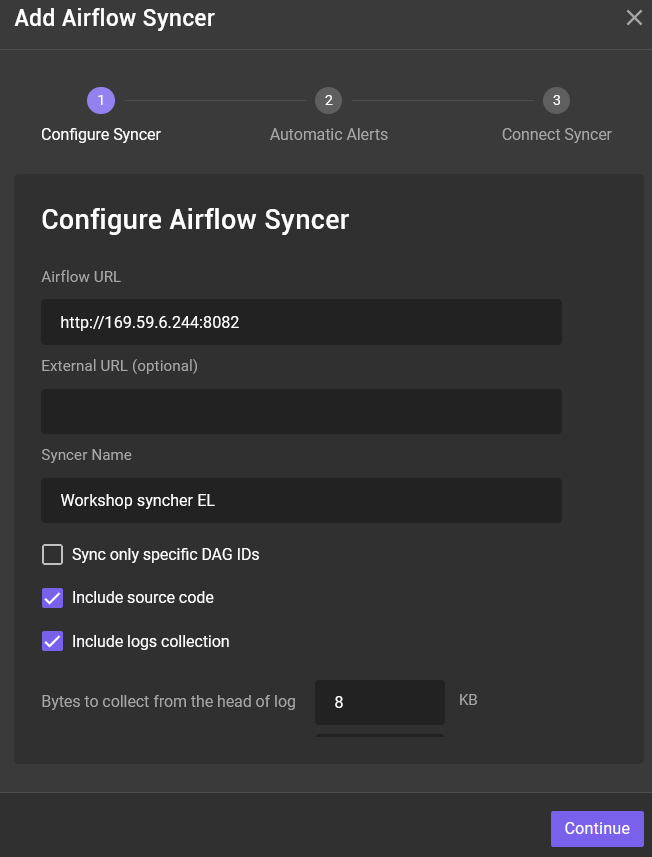
1. In Databand click **Integrations -> Add syncer**.



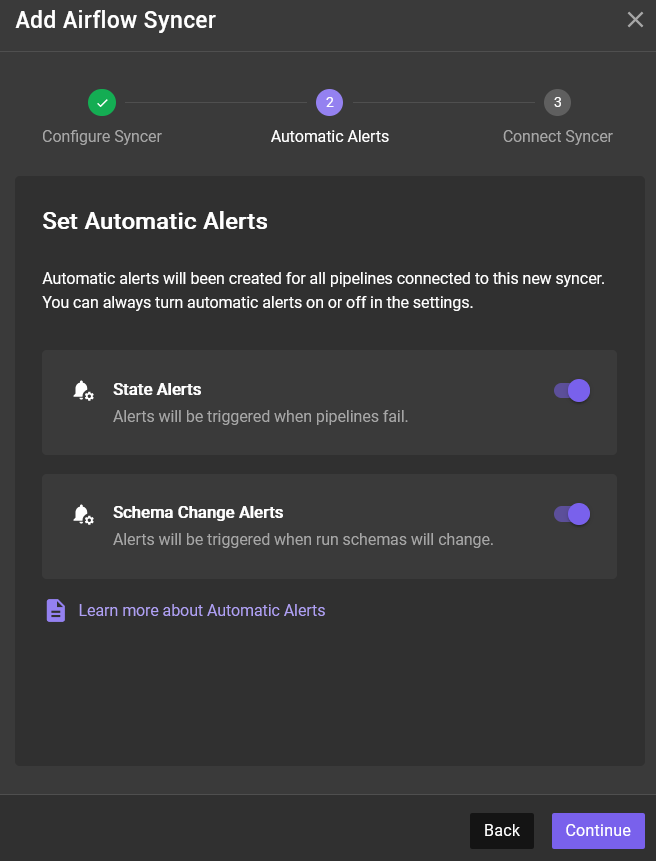
1. Fill out the required information
   * Airflow cluster URL: If you’re using the demo VM, then the URL is *<External IP>:8082*
   * Provide a unique syncer name (for example, add your initials)
   * Check the *Include source code* checkbox.

*Note: Airflow cluster URL in this configuration step is used for “ease of troubleshooting” , (clicking on a pipeline link in Databand will take you to the pipeline in Airflow). If you specify an incorrect Airflow URL in this step, Databand connectivity will still work. Connection and authentication are done from Airflow to Databand, and URL/token are generated during the last step of the wizard.*

Click **Next.**



1. Leave default values on the Set Automatic **Alerts** page and click **Next**.

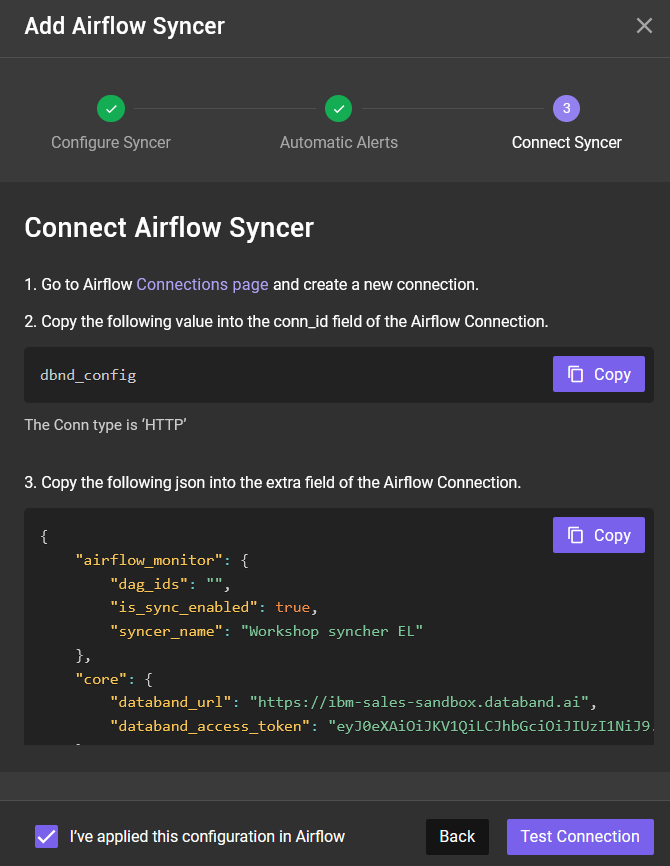


Next, we will copy Databand connection information from Databand to Airflow. We already have the *dbnd\_config* variable defined in Airflow connections, that’s why we don’t need to create it.

1. Copy the *json extra* field.

*Important Note: Like most tokens, this information will be displayed only once during configuration. Save this information in a notepad while you’re working on the lab.*

Do not click **Test Connection** at this time – you have not pasted the connectivity information to Airflow yet.

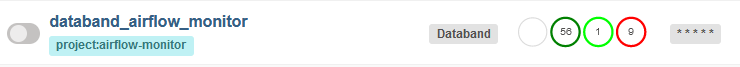


1. In Airflow navigate to **Admin -> Connections**, click on the **Edit** icon for the *dbnd\_config* variable, and paste the copied json into Extras field. Click **Save**.



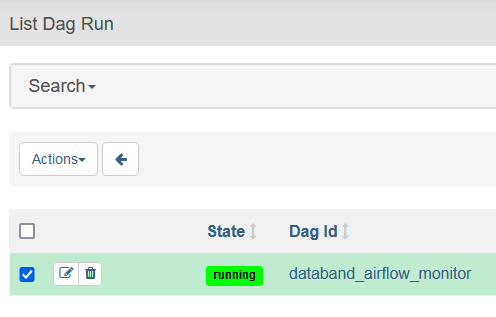
Since we updated the value, we need to restart the Databand DAG, which by default restarts every 3 hours. The easiest way to do this is to turn off the DAG, set the status of the current run to *success*, and to turn on the DAG.

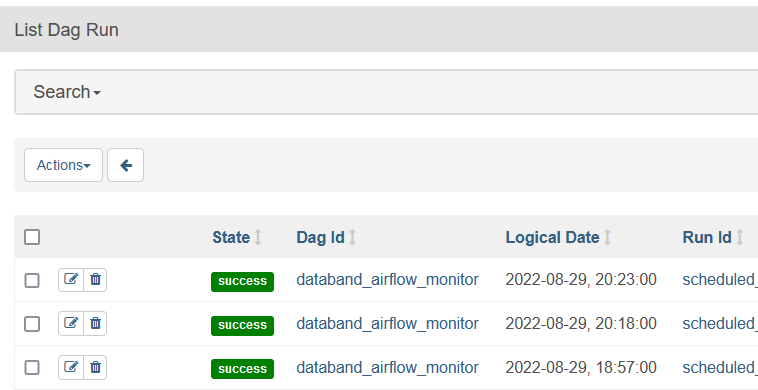
1. In Airflow navigate to the main admin page that displays all DAGs and turn off the *databand\_airflow\_monitor* DAG.



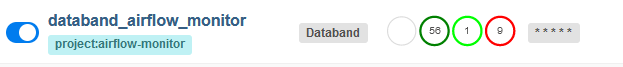
1. Click on the green circle in the same row as the DAG.

Select the current run by checking the checkbox. From the **Actions** dropdown select *Set state to success*. This step forces the completion of the run.



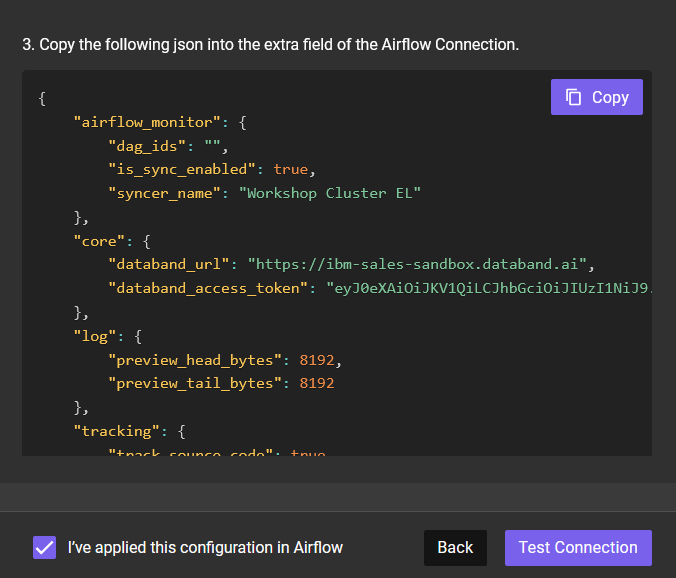


1. Return to the main DAG page and turn on the *databand\_airflow\_monitor* DAG.



Now that we updated connection information and restarted the monitoring DAG, we can test connection in Databand.

1. Return to the syncer configuration panel and click **Test Connection**.

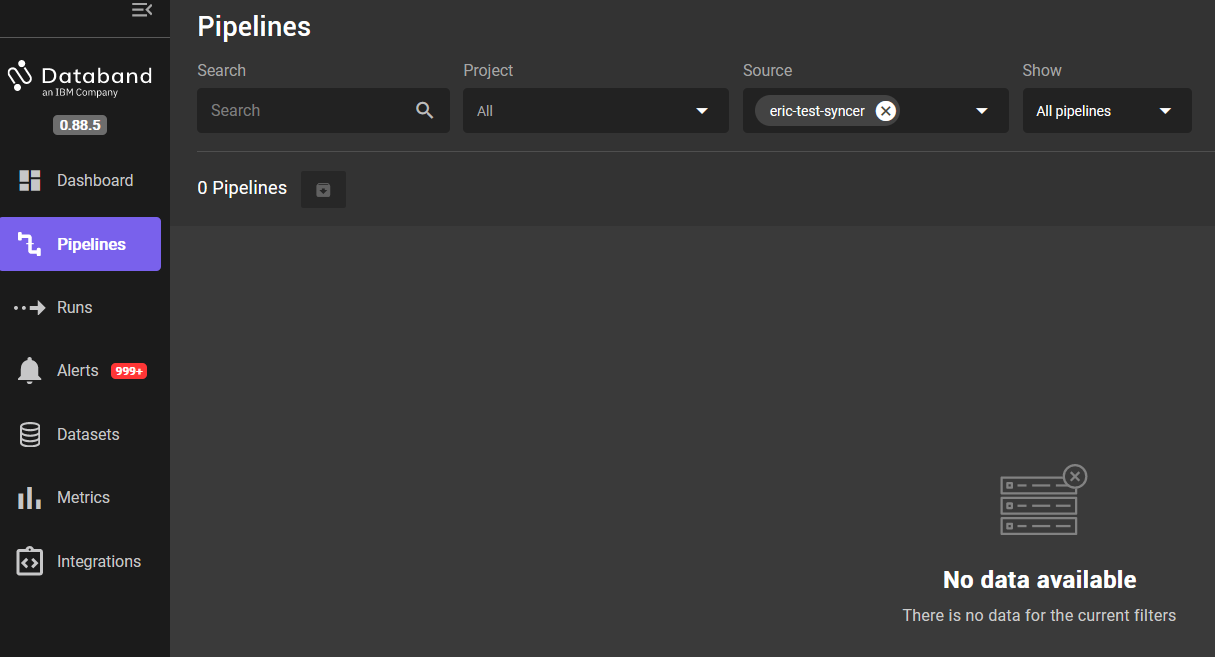


If your connection test is not successful, manually start the DAG run in Airflow and wait a few minutes till you test again.

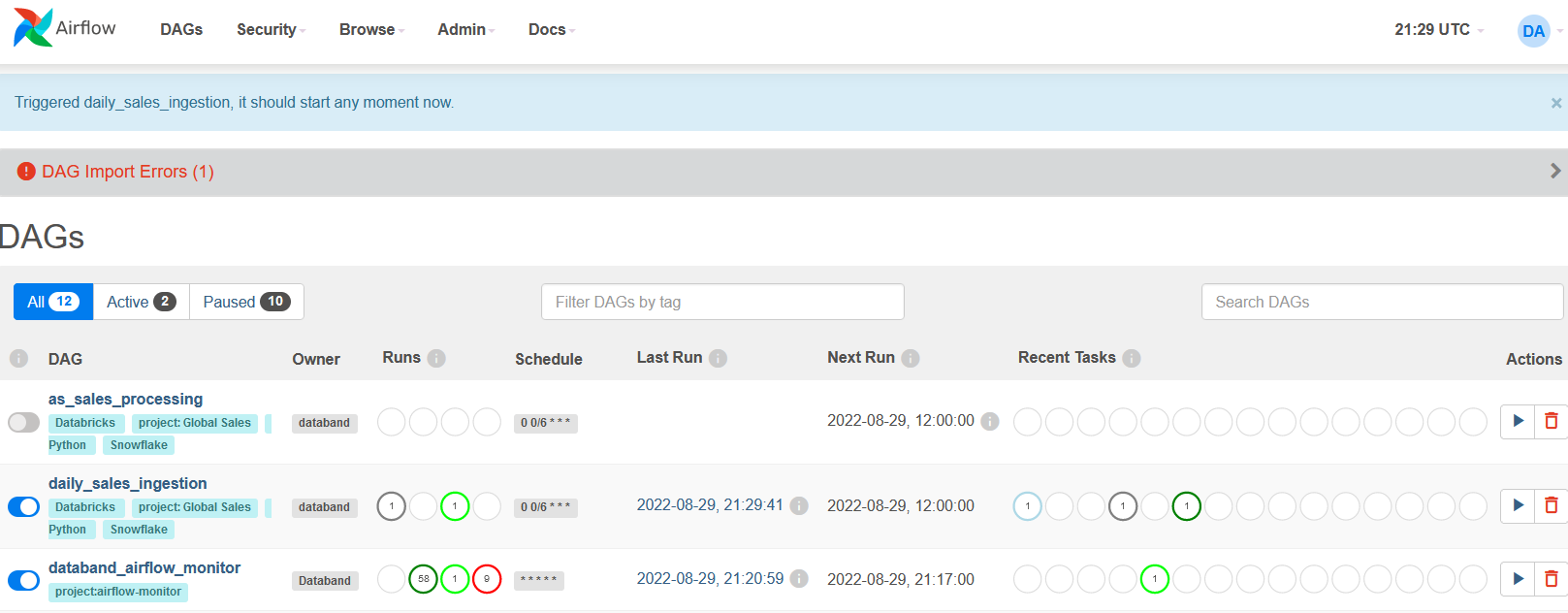
Summary of the steps that you completed so far:

* + Verified that the Databand SDK was installed on the cluster
  + Verified that the Databand monitoring DAG was installed on the cluster
  + Reviewed *dbnd\_config* connection that’s configured in Airflow
  + Created a syncer in Databand
  + As a part of the syncer creation process, copied the *Extras* section (which contains connection information from Airflow to Databand), and updated *dbnd\_config* variable.

In Databand navigate to the **Pipelines** view and filter it to display pipelines for your cluster. Since most DAGs are disabled in the Airflow cluster, we do not see any pipelines.

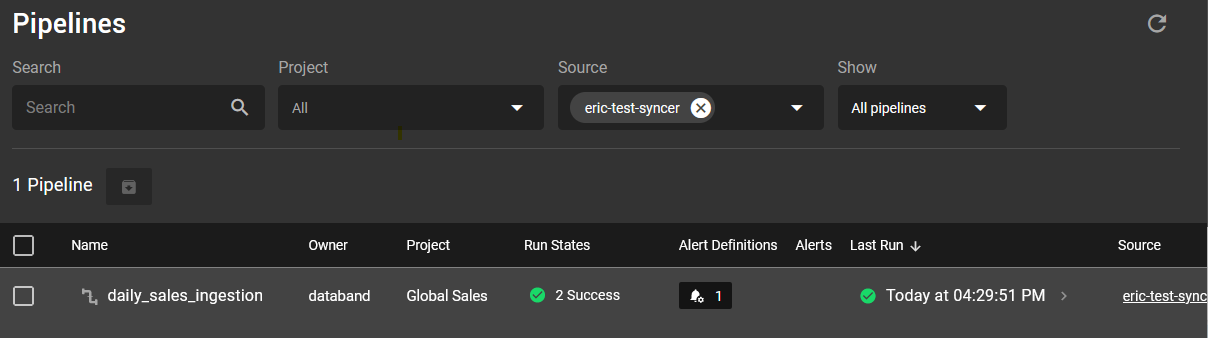


1. In Airflow enable one of the DAGs, for example, *daily\_sales\_ingestion*, and manually run it (use the arrow icon).



1. In Databand refresh the **Pipelines** view.

Now we see the pipeline and run status.



You have completed configuring the Databand/Airflow integration.

# Appendix A: Airflow Installation

See [Airflow documentation](https://airflow.apache.org/docs/apache-airflow/stable/start/docker.html) for installation instructions.