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.

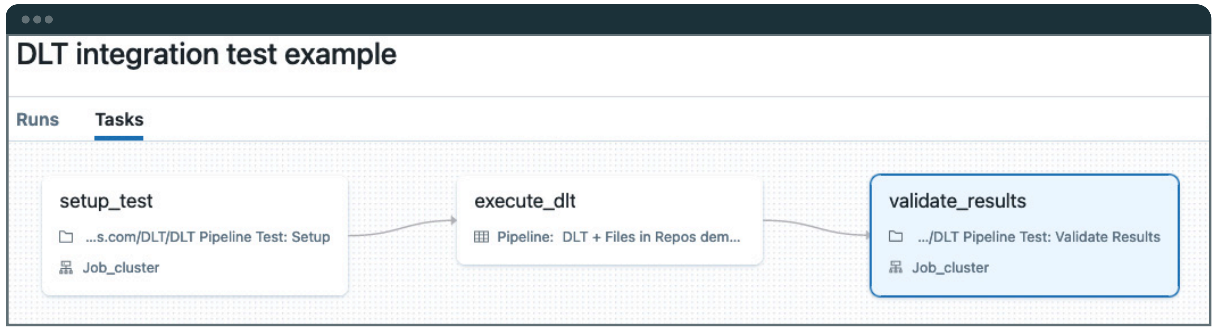

DLT integration test example
Runs
Tasks
setup_test
...s.com/DLT/DLT Pipeline Test: Setup
Job_cluster
execute_dlt
Pipeline: DLT + Files in Repos dem...
validate_results
.../DLT Pipeline Test: Validate Results
Job_cluster

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:

```python
@dlt.table(comment="Check type")
@dlt.expect_all_or_fail({"valid type": "type in ('link', 'redlink')",
                         "type is not null": "type is not null"})
def filtered_type_check():
  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):

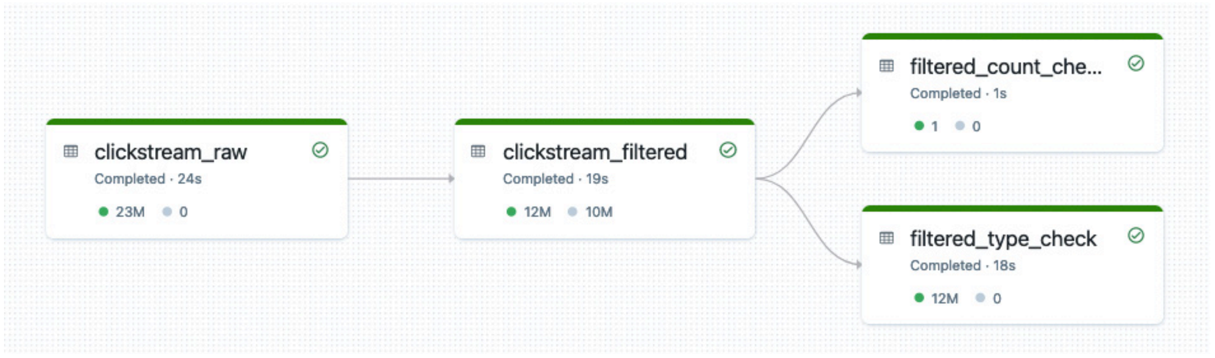


Figure: Implementing integration tests using DLT expectations

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.

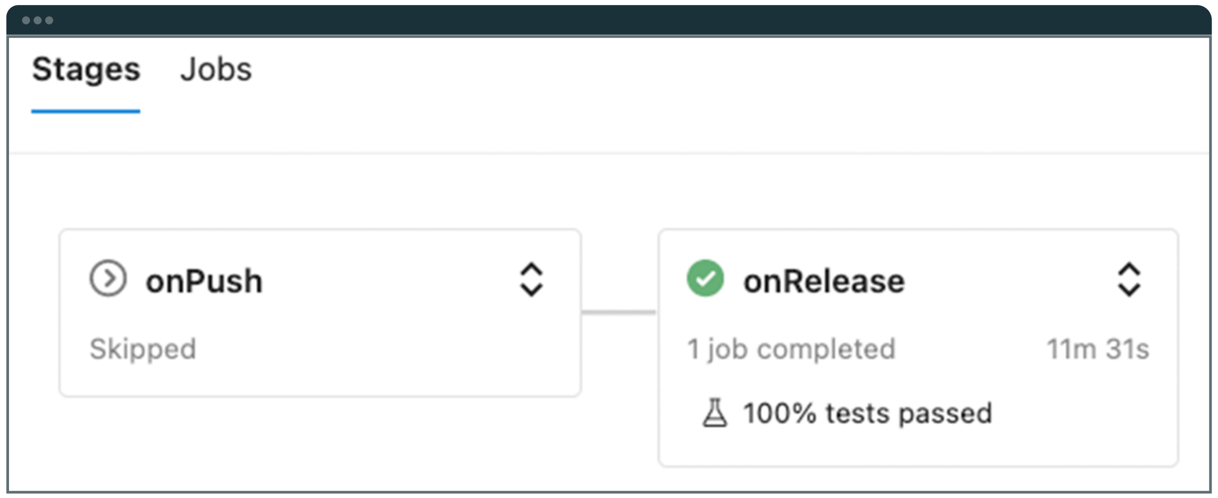
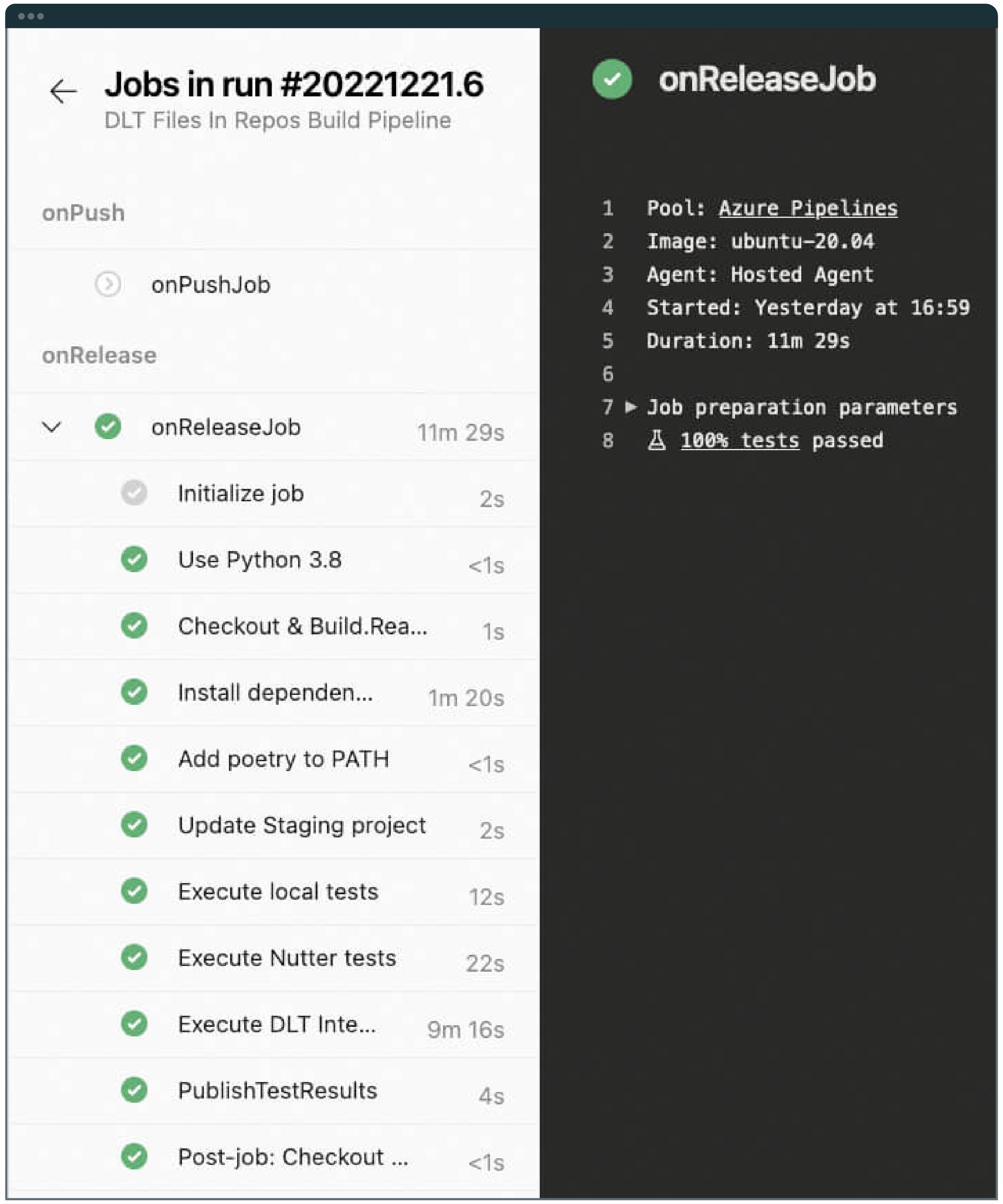


Figure: Structure of Azure DevOps build pipeline

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



Jobs in run #20221221.6
DLT Files In Repos Build Pipeline

onPush

onPushJob

onRelease

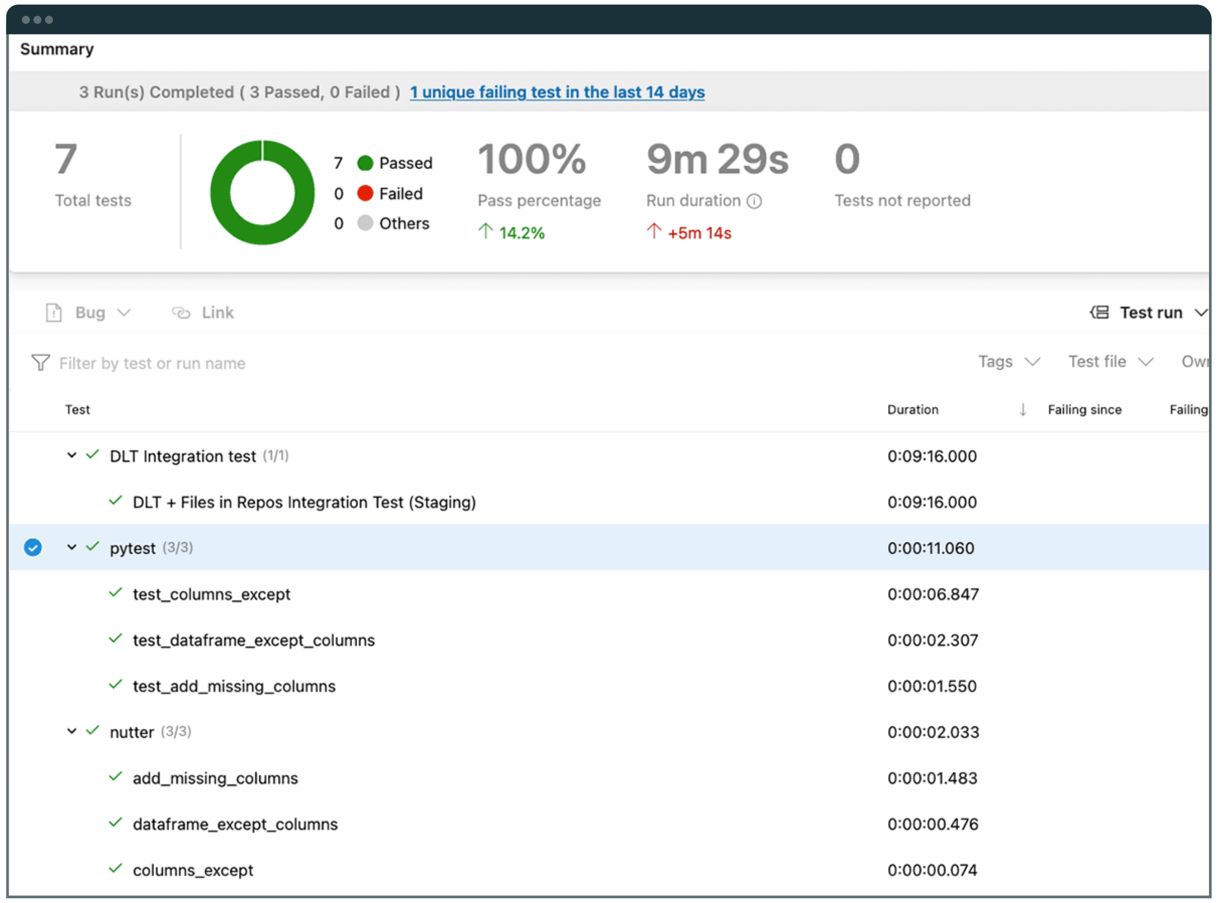
onReleaseJob	11m 29s
Initialize job	2s
Use Python 3.8	<1s
Checkout & Build.Rea...	1s
Install dependen...	1m 20s
Add poetry to PATH	<1s
Update Staging project	2s
Execute local tests	12s
Execute Nutter tests	22s
Execute DLT Inte...	9m 16s
PublishTestResults	4s
Post-job: Checkout ...	<1s

onReleaseJob

```
Pool: Azure Pipelines
Image: ubuntu-20.04
Agent: Hosted Agent
Started: Yesterday at 16:59
Duration: 11m 29s

▶ Job preparation parameters
100% tests passed
```

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



Summary

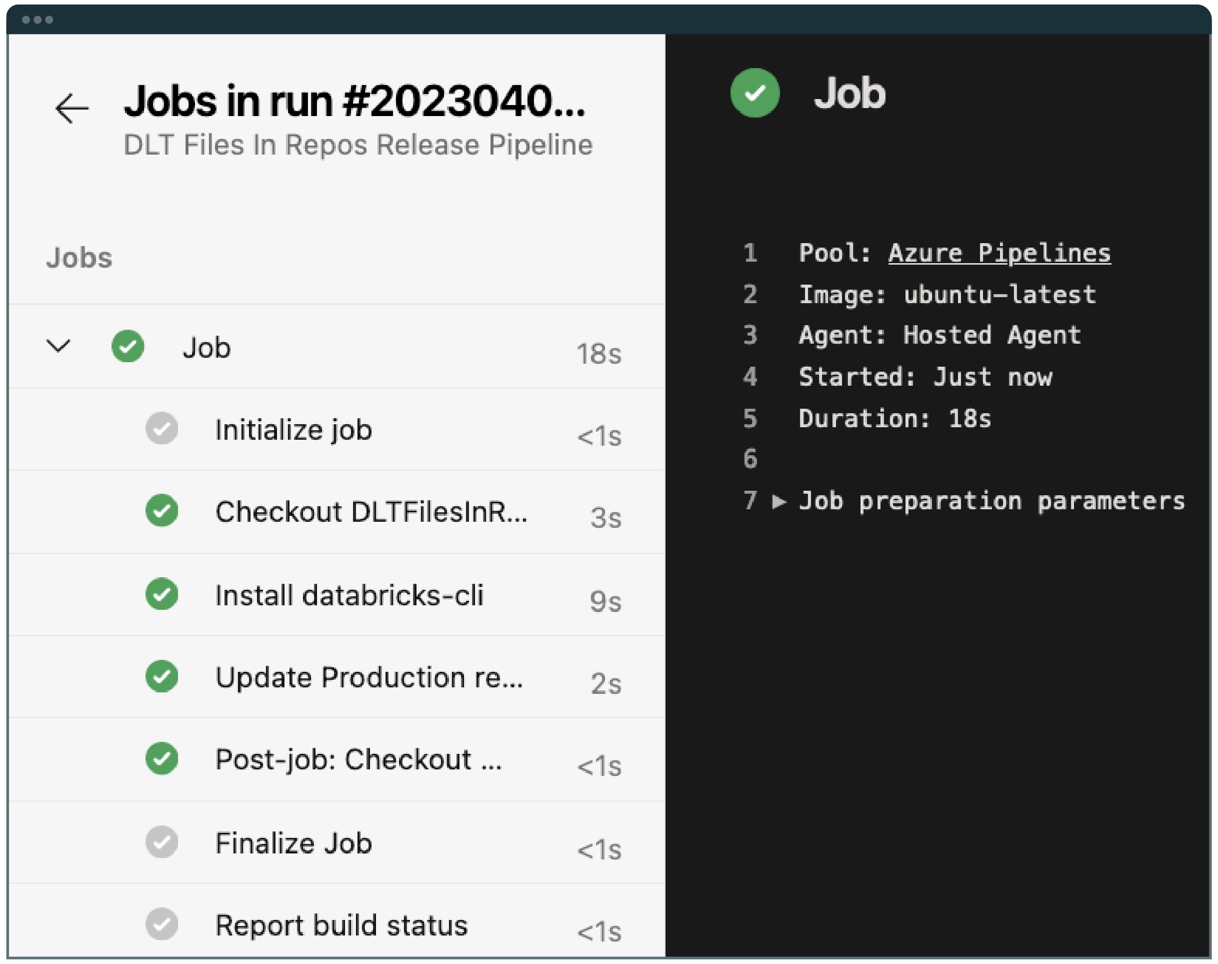
3 Run(s) Completed (3 Passed, 0 Failed) 1 unique failing test in the last 14 days

7 Total tests | 7 ● Passed, 0 ● Failed, 0 ● Others | 100% Pass percentage ↑ 14.2% | 9m 29s Run duration ↑ +5m 14s | 0 Tests not reported

Test	Duration	Failing since	Failing
DLT Integration test (1/1)	0:09:16.000		
DLT + Files in Repos Integration Test (Staging)	0:09:16.000		
pytest (3/3)	0:00:11.060		
test_columns_except	0:00:06.847		
test_dataframe_except_columns	0:00:02.307		
test_add_missing_columns	0:00:01.550		
nutter (3/3)	0:00:02.033		
add_missing_columns	0:00:01.483		
dataframe_except_columns	0:00:00.476		
columns_except	0:00:00.074		

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.



Jobs in run #2023040...

DLT Files In Repos Release Pipeline

Jobs

Job	18s
Initialize job	<1s
Checkout DLTFilesInR...	3s
Install databricks-cli	9s
Update Production re...	2s
Post-job: Checkout ...	<1s
Finalize Job	<1s
Report build status	<1s

Job

```
Pool: Azure Pipelines
Image: ubuntu-latest
Agent: Hosted Agent
Started: Just now
Duration: 18s

▶ Job preparation parameters
```

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.

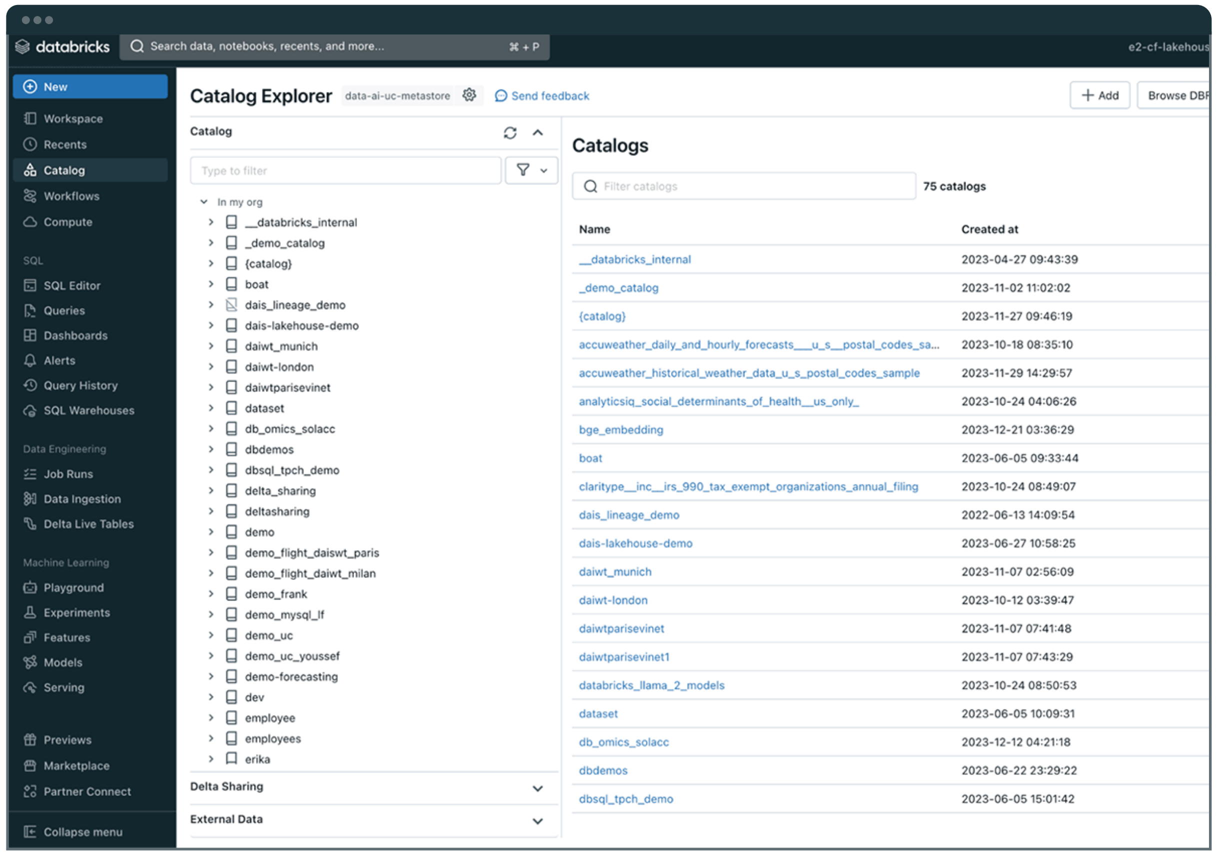
The major challenges discussed included weaker compliance and fractured data privacy controlled across multiple vendors; uncontrolled and siloed data and AI swamps; exponentially rising costs; loss of opportunities, revenue, and collaboration.

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.



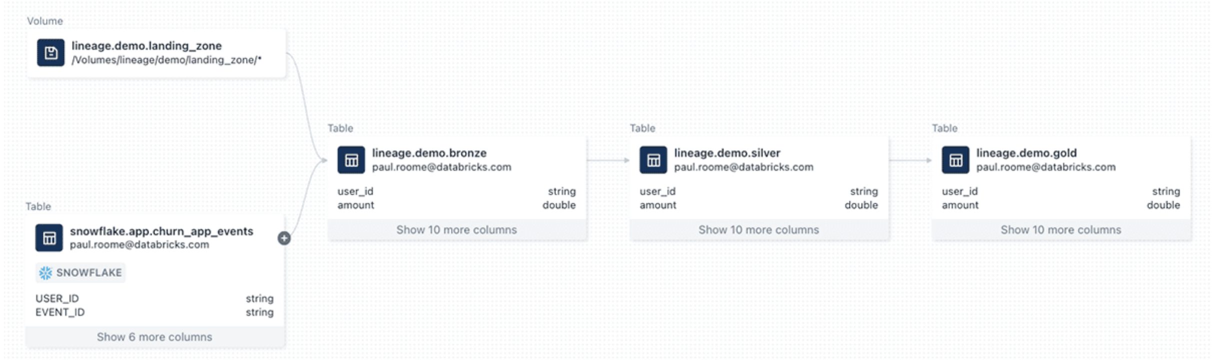
Catalogs

Filter catalogs — 75 catalogs

Name	Created at
__databricks_internal	2023-04-27 09:43:39
_demo_catalog	2023-11-02 11:02:02
{catalog}	2023-11-27 09:46:19
accuweather_daily_and_hourly_forecasts___u_s__postal_codes_sa...	2023-10-18 08:35:10
accuweather_historical_weather_data_u_s_postal_codes_sample	2023-11-29 14:29:57
analyticsiq_social_determinants_of_health__us_only_	2023-10-24 04:06:26
bge_embedding	2023-12-21 03:36:29
boat	2023-06-05 09:33:44
claritype__inc__irs_990_tax_exempt_organizations_annual_filing	2023-10-24 08:49:07
dais_lineage_demo	2022-06-13 14:09:54
dais-lakehouse-demo	2023-06-27 10:58:25
daiwt_munich	2023-11-07 02:56:09
daiwt-london	2023-10-12 03:39:47
daiwtparisevinet	2023-11-07 07:41:48
daiwtparisevinet1	2023-11-07 07:43:29
databricks_llama_2_models	2023-10-24 08:50:53
dataset	2023-06-05 10:09:31
db_omics_solacc	2023-12-12 04:21:18
dbdemos	2023-06-22 23:29:22
dbsql_tpch_demo	2023-06-05 15:01:42

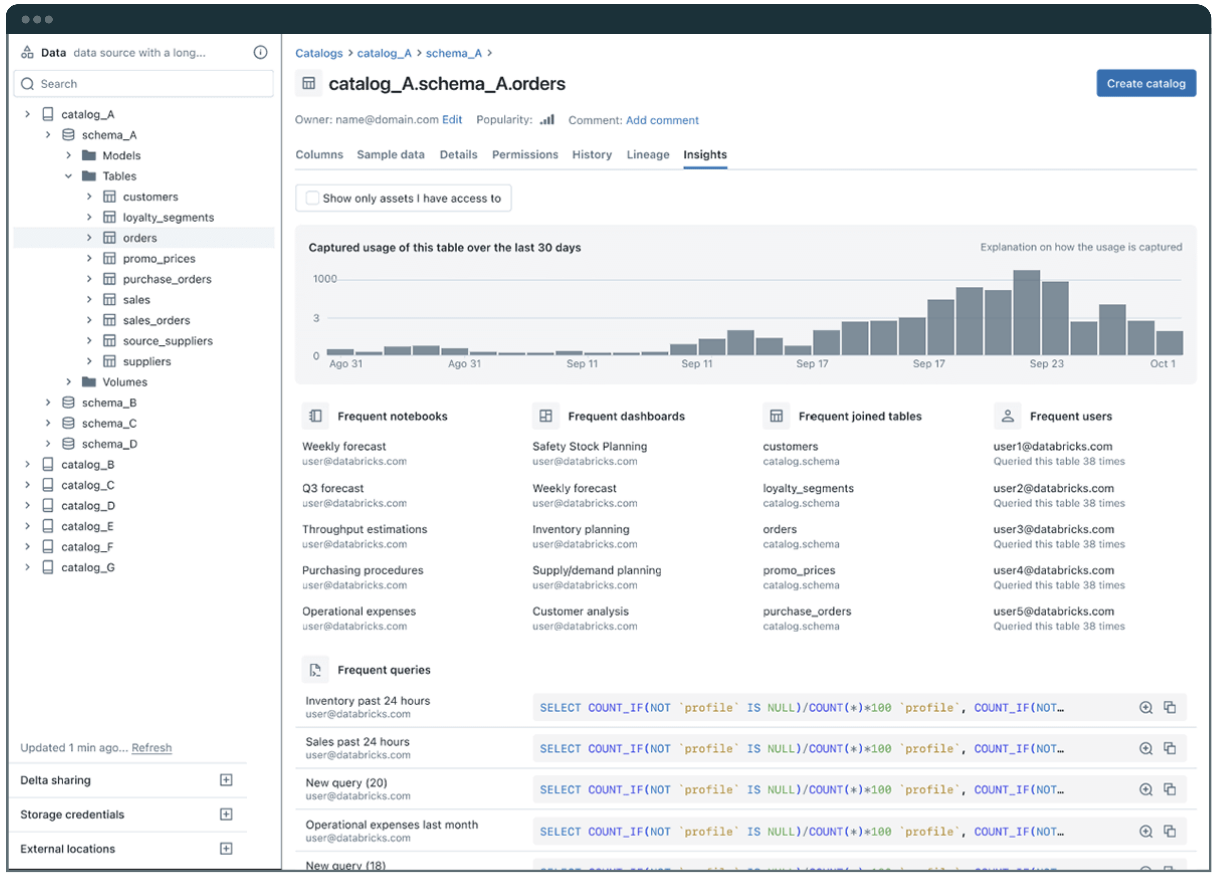
Catalog Explorer lets you discover and govern all your data and ML models

- 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:

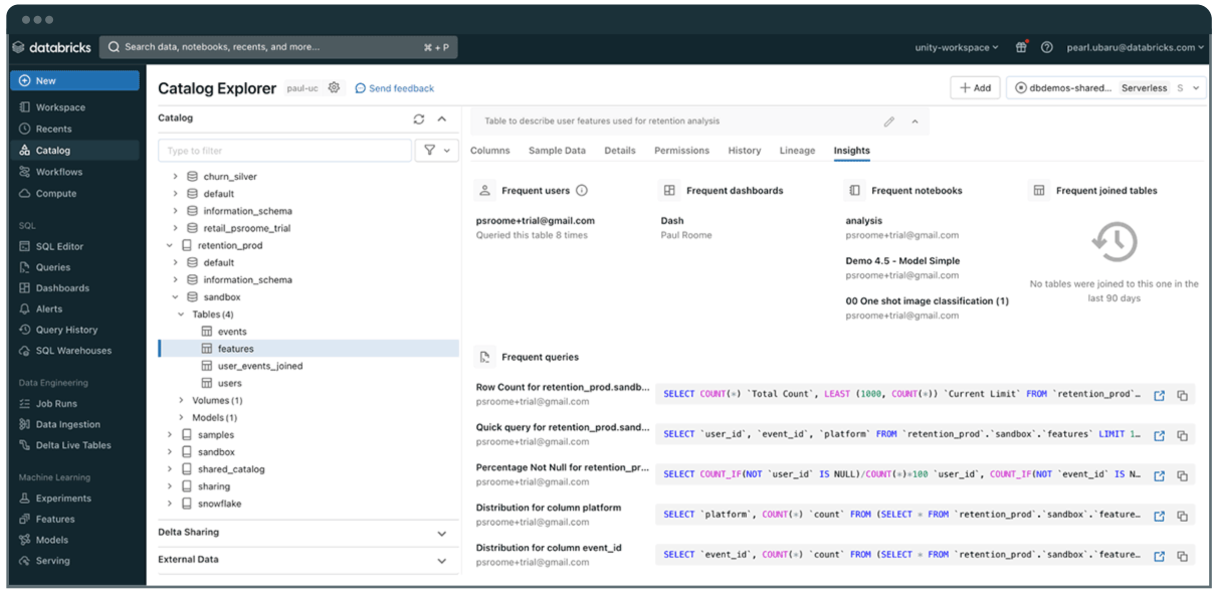

Catalogs > catalog_A > schema_A >
catalog_A.schema_A.orders
Create catalog
Owner: name@domain.com Edit Popularity: Comment: Add comment
Columns Sample data Details Permissions History Lineage Insights
Show only assets I have access to
Captured usage of this table over the last 30 days
Explanation on how the usage is captured

Frequent notebooks	Frequent dashboards	Frequent joined tables	Frequent users
Weekly forecast user@databricks.com	Safety Stock Planning user@databricks.com	customers catalog.schema	user1@databricks.com Queried this table 38 times
Q3 forecast user@databricks.com	Weekly forecast user@databricks.com	loyalty_segments catalog.schema	user2@databricks.com Queried this table 38 times
Throughput estimations user@databricks.com	Inventory planning user@databricks.com	orders catalog.schema	user3@databricks.com Queried this table 38 times
Purchasing procedures user@databricks.com	Supply/demand planning user@databricks.com	promo_prices catalog.schema	user4@databricks.com Queried this table 38 times
Operational expenses user@databricks.com	Customer analysis user@databricks.com	purchase_orders catalog.schema	user5@databricks.com Queried this table 38 times

Frequent queries

Inventory past 24 hours user@databricks.com	`SELECT COUNT_IF(NOT `profile` IS NULL)/COUNT(*)*100 `profile`, COUNT_IF(NOT…`
Sales past 24 hours user@databricks.com	`SELECT COUNT_IF(NOT `profile` IS NULL)/COUNT(*)*100 `profile`, COUNT_IF(NOT…`
New query (20) user@databricks.com	`SELECT COUNT_IF(NOT `profile` IS NULL)/COUNT(*)*100 `profile`, COUNT_IF(NOT…`
Operational expenses last month user@databricks.com	`SELECT COUNT_IF(NOT `profile` IS NULL)/COUNT(*)*100 `profile`, COUNT_IF(NOT…`
New query (18)	

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

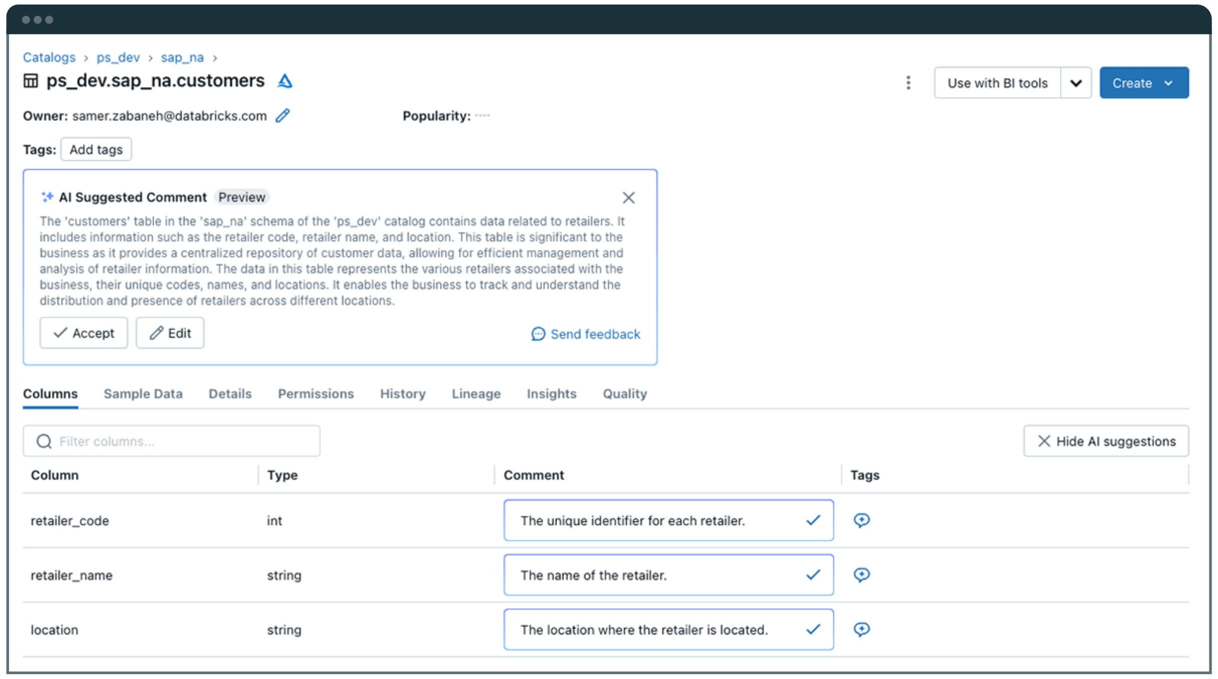

databricks
Catalog Explorer
Send feedback
+ Add
dbdemos-shared… Serverless
Table to describe user features used for retention analysis
Columns Sample Data Details Permissions History Lineage Insights

Frequent users	Frequent dashboards	Frequent notebooks	Frequent joined tables
psroome+trial@gmail.com Queried this table 8 times	Dash Paul Roome	analysis psroome+trial@gmail.com	No tables were joined to this one in the last 90 days
		Demo 4.5 - Model Simple psroome+trial@gmail.com	
		00 One shot image classification (1) psroome+trial@gmail.com	

Frequent queries

Row Count for retention_prod.sandb… psroome+trial@gmail.com	`SELECT COUNT(*) `Total Count`, LEAST (1000, COUNT(*)) `Current Limit` FROM `retention_prod`…`
Quick query for retention_prod.sand… psroome+trial@gmail.com	`SELECT `user_id`, `event_id`, `platform` FROM `retention_prod`.`sandbox`.`features` LIMIT 1…`
Percentage Not Null for retention_pr… psroome+trial@gmail.com	`SELECT COUNT_IF(NOT `user_id` IS NULL)/COUNT(*)*100 `user_id`, COUNT_IF(NOT `event_id` IS N…`
Distribution for column platform psroome+trial@gmail.com	`SELECT `platform`, COUNT(*) `count` FROM (SELECT * FROM `retention_prod`.`sandbox`.`feature…`
Distribution for column event_id psroome+trial@gmail.com	`SELECT `event_id`, COUNT(*) `count` FROM (SELECT * FROM `retention_prod`.`sandbox`.`feature…`

Metadata tagging insights: details on the “features” table


Catalogs > ps_dev > sap_na >
ps_dev.sap_na.customers
Use with BI tools
Create
Owner: samer.zabaneh@databricks.com
Popularity:
Tags: Add tags
AI Suggested Comment Preview
Accept
Edit
Send feedback
Columns Sample Data Details Permissions History Lineage Insights Quality
Filter columns...
Hide AI suggestions

The 'customers' table in the 'sap_na' schema of the 'ps_dev' catalog contains data related to retailers. It includes information such as the retailer code, retailer name, and location. This table is significant to the business as it provides a centralized repository of customer data, allowing for efficient management and analysis of retailer information. The data in this table represents the various retailers associated with the business, their unique codes, names, and locations. It enables the business to track and understand the distribution and presence of retailers across different locations.

Column	Type	Comment	Tags
retailer_code	int	The unique identifier for each retailer.	
retailer_name	string	The name of the retailer.	
location	string	The location where the retailer is located.	

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.

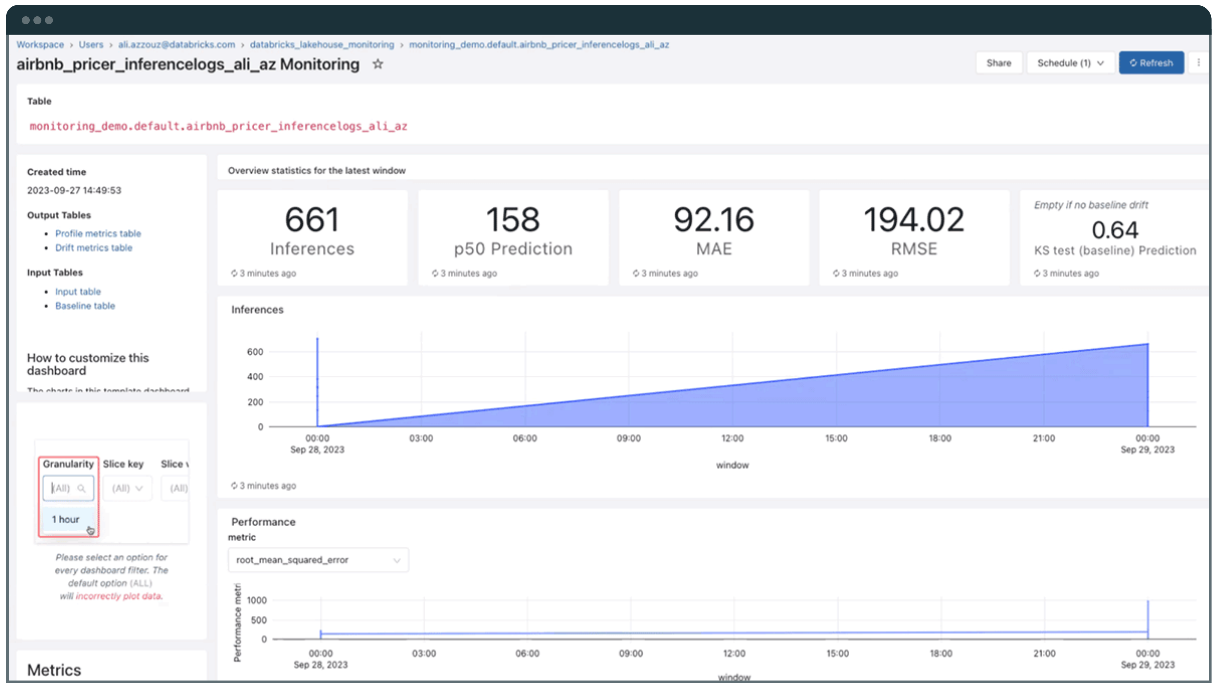
COMPREHENSIVE DATA AND AI MONITORING AND REPORTING

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:



Please select an option for every dashboard filter. The default option (ALL) will incorrectly plot data.

databricks