Databricks Workflows CICD and Automated Testing

…

Dustin Vannoy
dustinvannoy.com

This talk

Databricks Workflows (also known as Jobs) are a great choice for automating data pipelines. Once the code is ready comes the important step of promoting beyond your dev environment. Continuous Integration / Continuous Deployment (CI/CD) involves versioning, testing, and deploying your data processing jobs. Databricks provides tools that allow us to follow these DevOps best practices, but how do we put these together to ensure quality and manage workflow promotion across isolated environments? Join this session to learn some of the most common ways teams leverage Databricks to version, test, and deploy their automated data pipelines. In this session we cover some basic CI/CD concepts and the options within Databricks. Then we walk through an example of merging, testing, and deploying a workflow change.

Agenda

- Overview of CICD practices
- Databricks workflows
- Databricks asset bundles
- Testing and automation (Github Actions)

Overview of CICD practices

Why CICD?

Ensure best practices and easy release of new features

- *Code version control*
- *Automated tests*
- *Automated deploy (no manual steps)*
- *Faster innovation*

Continuous Integration

- Develop code
- Save to source control
- Run automated tests (pre-deploy)
- Build artifacts

Continuous Deployment

- Deploy code to stage and prod environments
- Run integration and system tests
- Schedule automated runs
- Re-install code and restart streaming jobs

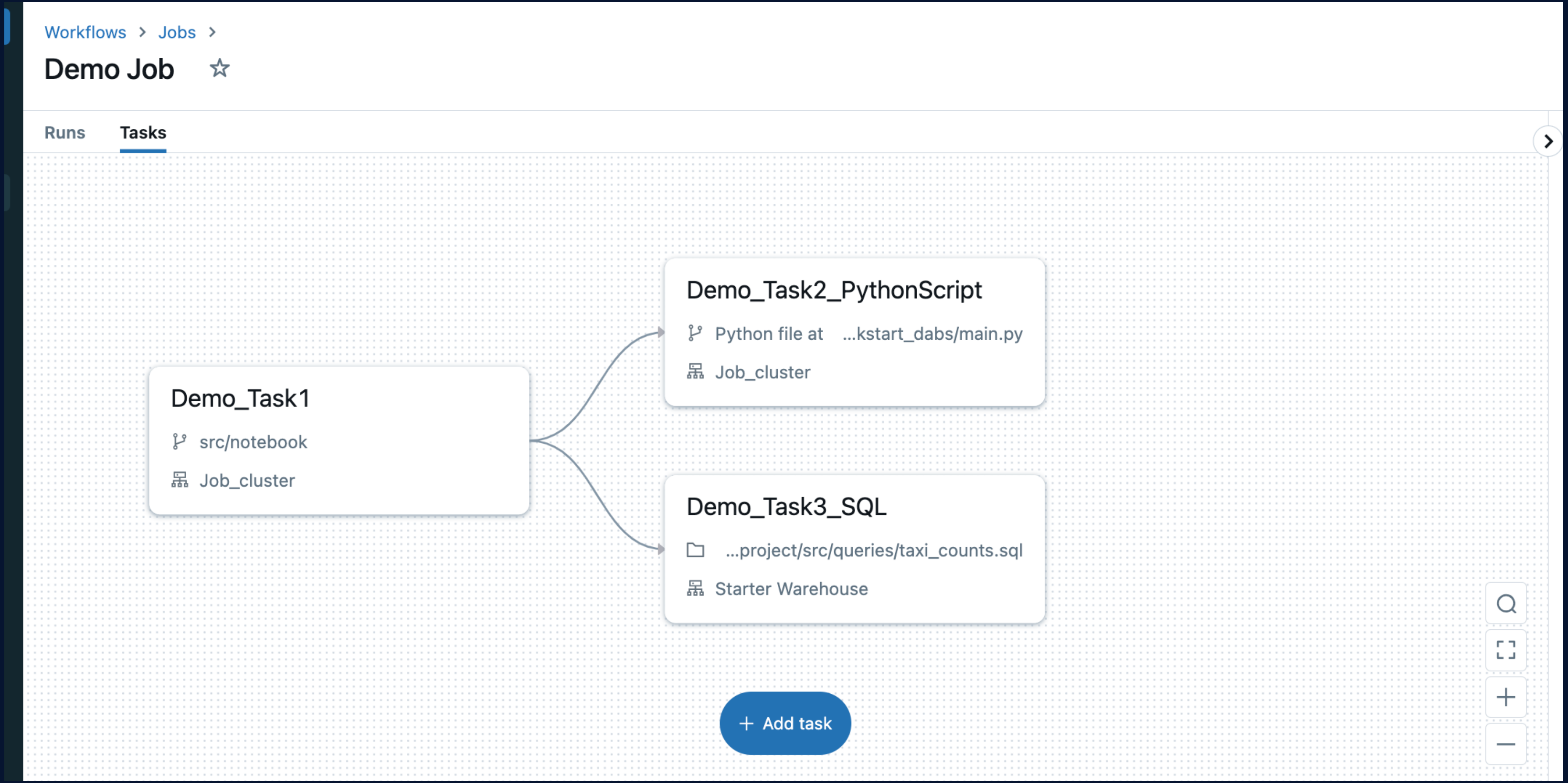
Databricks workflows

Why Databricks Workflows?

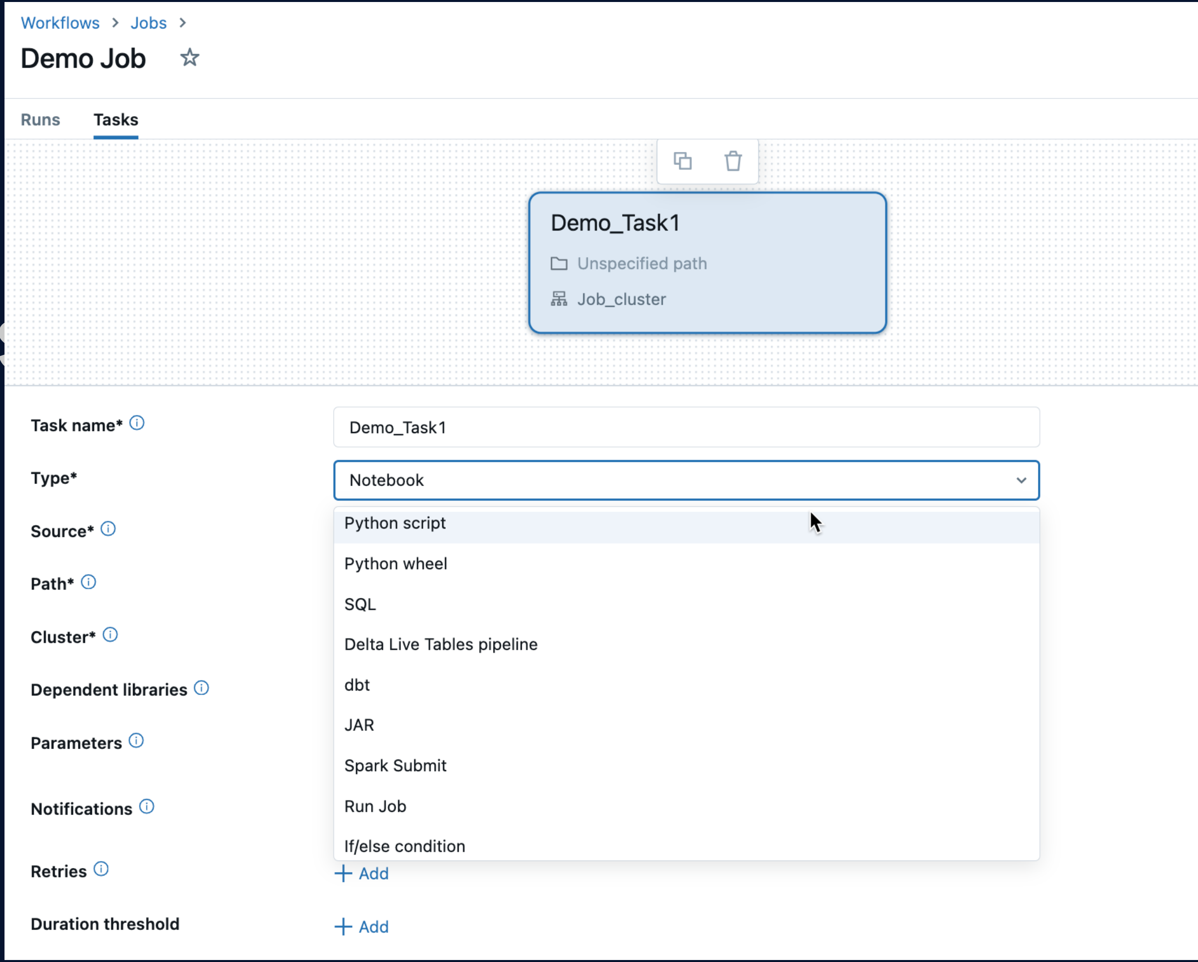
Automated jobs that support complex dependencies

- *Trigger on schedule, file arrival, or API call*
- *Set tasks with dependencies*
- *Task types:*
  - *Notebook*
  - *Python script*
  - *SQL*
  - *Etc.*

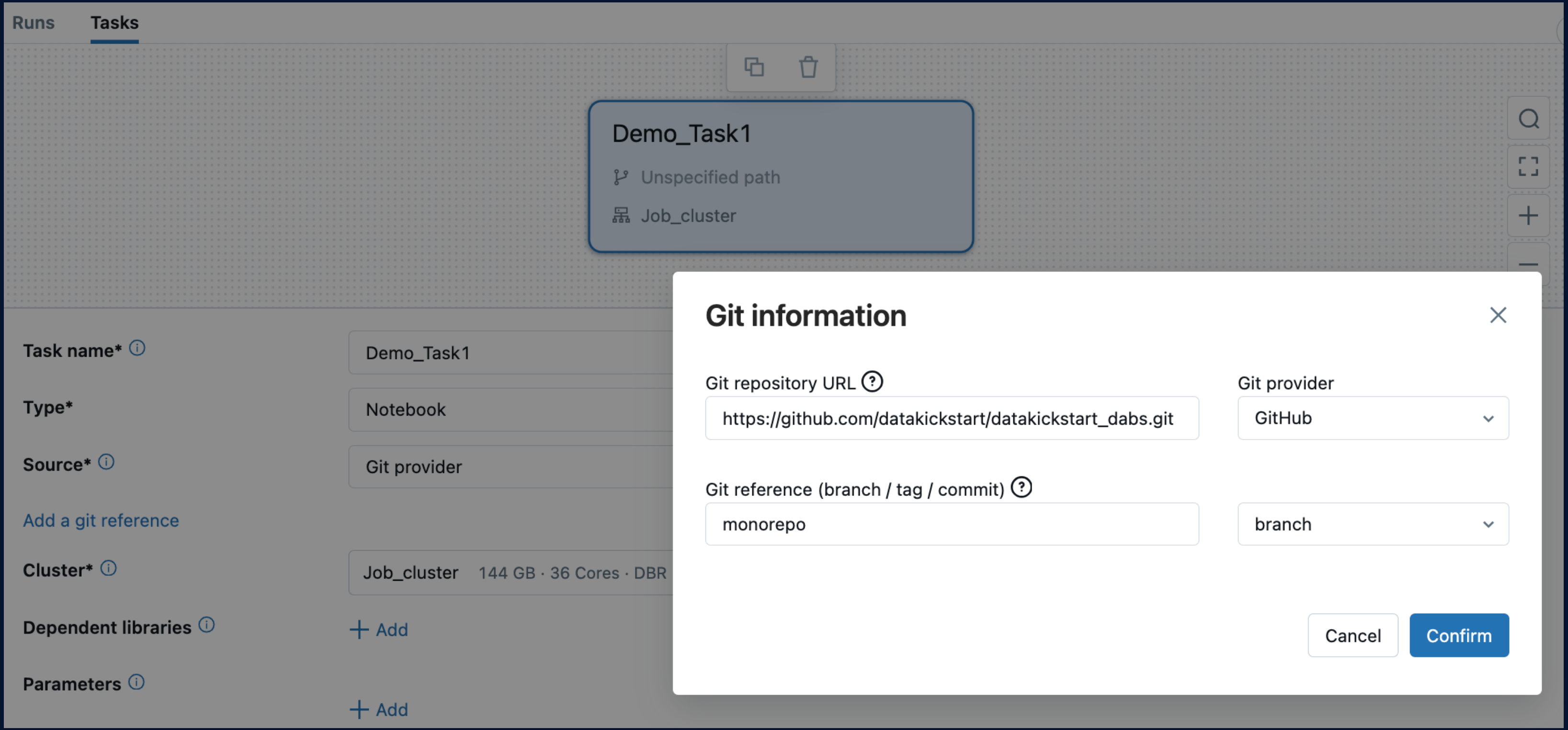
Orchestrate Databricks Tasks



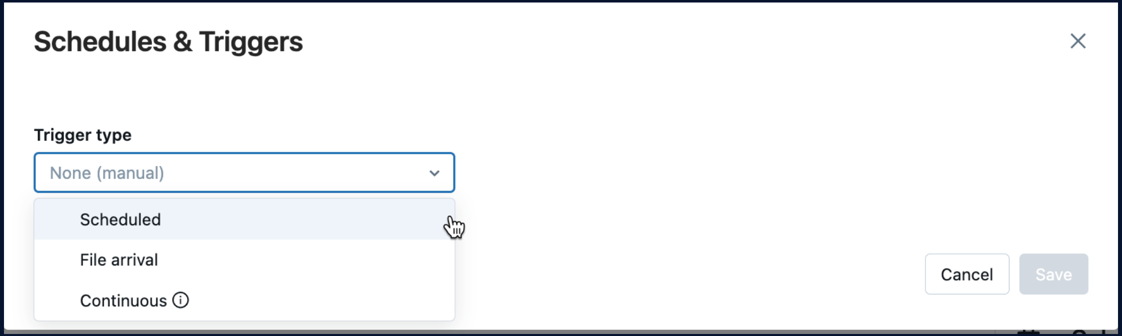
Variety of task types available

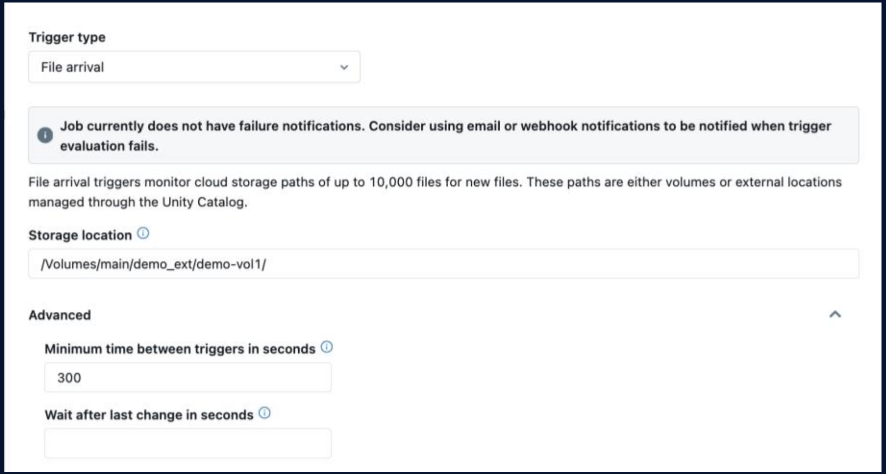


Source control integration (optional)



Setup trigger (optional)





Databricks asset bundles

Anatomy of your projects in Databricks

Let’s describe them

Consist of a variety of components

Code: Notebooks, Python .whl, JAR, dbt, etc.

Execution Environment: Databricks Workspace, compute configuration

Other resources: Databricks Workflows, MLflow Tracking Server and Registry, Delta Live Tables...

Produce a variety of data products

Create tables and pipelines, reports, machine learning models, dashboards, call external services, etc.

The task determines the components

A simple report might consist of a notebook running on single node compute

A full MLOps pipeline would require MLflow, Feature Store, and Model Serving components

Databricks Asset Bundles

Write code once, deploy everywhere

What are Databricks Asset Bundles?

YAML files that specify the artifacts, resources, and configurations of a Databricks project.

How do bundles work?

The **new databricks CLI** has functions to **validate, deploy and run** Databricks Asset Bundles using bundle.yml files

Where are bundles used?

Bundles are useful during **development and CI/CD** processes

A closer look

Name and default Workspace

Resource configurations

- Jobs, DLT pipelines, MLflow, etc.
- Follows REST API schema

Environment-based specs

- Control project behavior in different environments

bundle.yml

```yaml
bundle:
  name: shark_sightings

workspace:
  host: https://e2-dogfood.staging.cloud.databricks.com

resources:
  jobs:
    shark_sightings:
      name: "[${bundle.environment}] Shark sightings"
      tasks:
        - task_key: shark_sightings
          notebook_task:
            base_parameters:
              dbname: "shark_sightings_${bundle.environment}"
            notebook_path: ./shark_sightings.py
          new_cluster:
            spark_version: 10.4.x-scala2.12
            num_workers: 1
            node_type_id: i3.xlarge

environments:
  development:
    default: true
  production:
    workspace:
      host: https://e2-demo-west.cloud.databricks.com
    resources:
      jobs:
        shark_sightings:
          schedule:
            quartz_cron_expression: 14 8 14 * * ?
            timezone_id: UTC
```

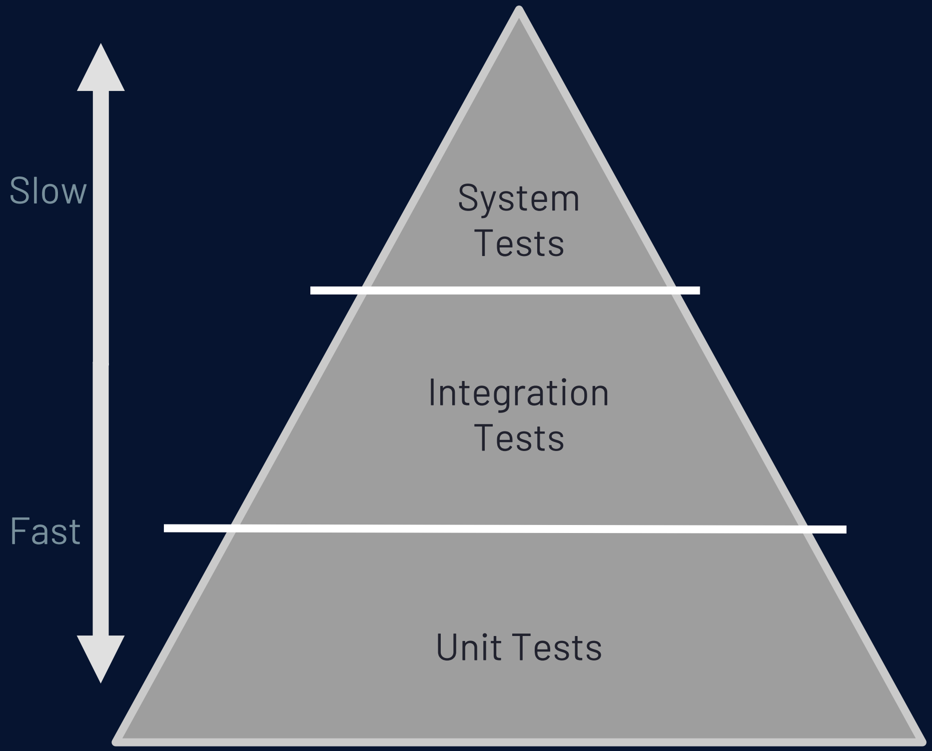
Line 34, Column 1 | Spaces: 2 | YAML

SHARK_SIGHTINGS
- .databricks
- .vscode
- .gitignore
- bundle.yml
- shark_sightings.py

Testing and automation

(Github Actions)

CI/CD for Databricks: Testing rationale



- Functional Tests
- Integration with other systems

- Spark Notebook / Job tests

- Core business logic / UDFs (dataframe in, dataframe out)

Native Testing for PySpark in Spark 3.5 / DBR 13.3+

pyspark.testing.assertDataFrameEqual

pyspark.testing.**assertDataFrameEqual**(*actual: Union[pyspark.sql.dataframe.DataFrame, pandas.DataFrame, pyspark.pandas.DataFrame, List[pyspark.sql.types.Row]], expected: Union[pyspark.sql.dataframe.DataFrame, pandas.DataFrame, pyspark.pandas.DataFrame, List[pyspark.sql.types.Row]], checkRowOrder: bool = False, rtol: float = 1e-05, atol: float = 1e-08*) [source]

A util function to assert equality between *actual* and *expected* (DataFrames or lists of Rows), with optional parameters *checkRowOrder*, *rtol*, and *atol*.

Supports Spark, Spark Connect, pandas, and pandas-on-Spark DataFrames. For more information about pandas-on-Spark DataFrame equality, see the docs for *assertPandasOnSparkEqual*.

New in version 3.5.0.

pyspark.testing.assertPandasOnSparkEqual

pyspark.testing.**assertPandasOnSparkEqual**(*actual: Union[pyspark.pandas.frame.DataFrame, pyspark.pandas.series.Series, pyspark.pandas.indexes.base.Index], expected: Union[pyspark.pandas.frame.DataFrame, pandas.core.frame.DataFrame, pyspark.pandas.series.Series, pandas.core.series.Series, pyspark.pandas.indexes.base.Index, pandas.core.indexes.base.Index], checkExact: bool = True, almost: bool = False, rtol: float = 1e-05, atol: float = 1e-08, checkRowOrder: bool = True*) [source]

A util function to assert equality between actual (pandas-on-Spark object) and expected (pandas-on-Spark or pandas object).

New in version 3.5.0.

pyspark.testing.assertSchemaEqual

pyspark.testing.**assertSchemaEqual**(*actual: pyspark.sql.types.StructType, expected: pyspark.sql.types.StructType*) [source]

A util function to assert equality between DataFrame schemas *actual* and *expected*.

New in version 3.5.0.

See Example in Spark Docs

Testing libraries for Spark

- chispa - Python version of spark-fast-tests
  - Authored by Matthew Powers
- spark-testing-base:
  - Scala & Python support
  - Supports RDD, Dataframe/Dataset, Streaming APIs
- spark-fast-tests - Scala, Spark 2 & 3
- pytest-spark - Python, native integration with pytest

```python
from chispa.dataframe_comparer import assert_df_equality

def test_remove_non_word_characters_long():
    source_data = [
        ("jo&&se",),
        ("**li**",),
        ("#::luisa",),
        (None,)
    ]
    source_df = spark.createDataFrame(source_data, ["name"])

    actual_df = source_df.withColumn(
        "clean_name",
        remove_non_word_characters(F.col("name"))
    )

    expected_data = [
        ("jo&&se", "jose"),
        ("**li**", "li"),
        ("#::luisa", "luisa"),
        (None, None)
    ]
    expected_df = spark.createDataFrame(expected_data, ["name", "clean_name"])

    assert_df_equality(actual_df, expected_df)
```

Github Action

- Script to handle CI and CD for the project
- For mono repo, separate action definition per project
- Run unit tests and integration tests
- Deploy DABs and run validation workflows

Additional resources

- DAIS 2023 Presentation: https://www.youtube.com/watch?v=9HOgYVo-WTM
- Code:
    - https://github.com/datakickstart/datakickstart_dabs

More Content

Website: dustinvannoy.com
LinkedIn: dustinvannoy
YouTube: Dustin Vannoy on YouTube

Thank you!

More Content

Website: [dustinvannoy.com]
Twitter: @dustinvannoy
YouTube: Dustin Vannoy on YouTube