# Task Parameters and Values in Databricks Workflows

Ryan Chynoweth

11 min read · Dec 7, 2022



Databricks provides a set of powerful and dynamic orchestration capabilities that are leveraged to build scalable pipelines supporting data engineering, data science, and data warehousing workloads. Databricks Workflows allow users to create jobs that can have many tasks. Tasks can be executed in parallel isolation and can set to follow specific dependencies. Below are the items that we will discuss while focusing on creating parameterized jobs using the user interface and APIs.

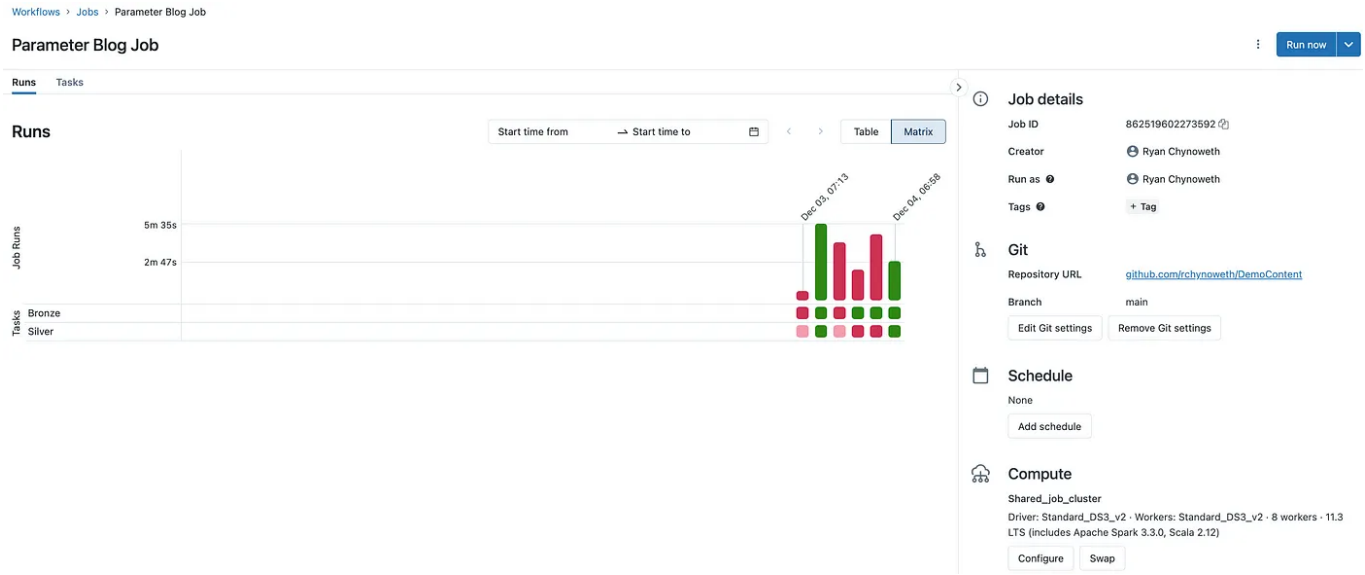
- Ephemeral job clusters
- Reusing job clusters
- Linking jobs to external git repositories
- Passing values between tasks within a job
- Parameterizing tasks

Please note that we will focus on notebook tasks in Databricks, however, much of what we discuss will be similar for other task types.

## Overview

Databricks Workflows is the name of the product that is used to create and schedule jobs. A job is a top level namespace containing tasks, compute definitions, and other required metadata. Workflows contain jobs, jobs contain tasks, and tasks are linked to an individual compute cluster that can be shared by many tasks to run data applications.

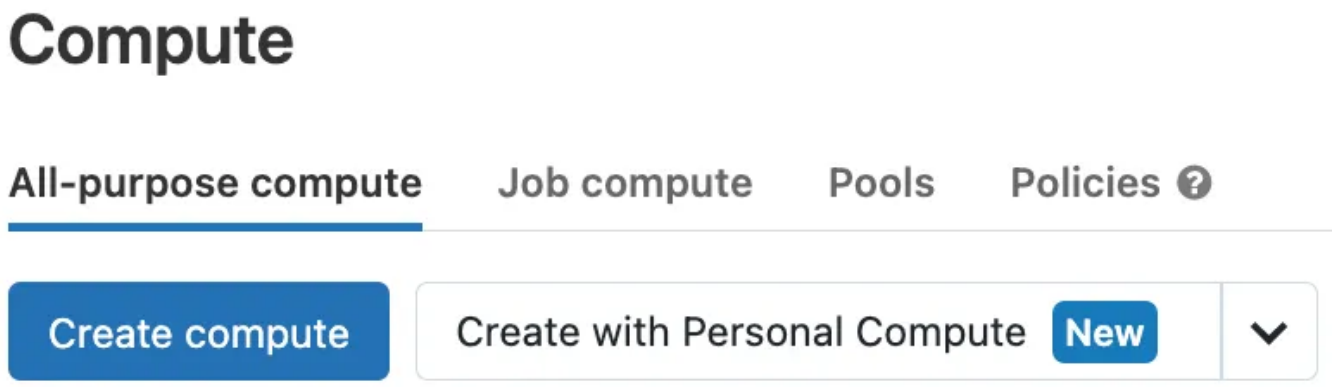
In the image below you can see that each execution of a job, a job run, is linked to the job allowing users to view the statuses of previous job runs. If your job is triggered externally, then it is recommended to create the job in Databricks then use the run now endpoint so that each job run is associated with a single job. This is in contrast to the one-time run which creates a new job with each execution, which in turn does not allow for easy tracking of multiple runs for the same set of code.




Jobs User Interface

## Clusters

Ephemeral job clusters are compute instances that are used to run specific workloads and once complete the cluster is terminated. Running jobs on automated clusters allows users to take advantage of a lower price per unit when compared to all purpose clusters. All purpose clusters are typically used during development and give engineers the ability to leverage many of the development tools available on the platform. Once done developing, a users can schedule the code on a job cluster for reduced cost and automation. Below is an image showing the different tabs for the various compute options. Please note that pools are a way to maintain a set of warm machines for faster start time and policies are used to enforce specification requirements for clusters.




Compute Options in Databricks

Historically, each task within a job required a separate cluster. Each time a cluster was requested there would be a wait time for virtual machines to be assigned by the cloud provider then Databricks could install the appropriate software. These wait times could be burdensome for many customers. Databricks pools are a way to maintain a set of virtual machines on idle so that customers could eliminate the cloud provisioning time, and are still a great way to ensure faster start up. To further eliminate the pain point of compute creation, users have the ability to reuse clusters between tasks

eliminating start up times entirely for subsequent tasks while maintaining isolation and dedicated task compute.

### Git

Databricks Repos allow users to connect to external git repositories such as GitHub and Azure DevOps. They are heavily used throughout the development process to commit and pull code changes in Databricks. They can be used for automation and deployment as highlighted in option two.

Repos are great and to take them once step further, jobs in Databricks can be linked directly to git repositories eliminating the requirement that the repo needs to be created in the Databricks environment to run a job. Engineers can set jobs to use a specific branch to ensure seamless deployment processes between environments. Providing the git repository in the job definition simply reduces the required steps and can ensure that solutions are always up to date. We will highlight this feature in our example pipeline.

### Task Parameters

A common requirement for data applications is to run the same set of code multiple times with a different set of parameters. This is easily done in Databricks using parameters provided at runtime or environment variables, however, the nuance of providing these parameters using the REST APIs or CLI requires familiarity of the API object definitions which we will cover in more detail later in the article.

When you create a task in a Databricks job, you can assign parameters to that task which will isolated to that task alone. Job level parameters are linked to the job which means if you have multiple tasks that rely on job parameters (not task parameters) with the same parameter name, then the value provided at runtime is the same for all tasks in that job. In the example

we will discuss later in this article, we have two tasks that both have `schema` and `table` parameters. If you provide job parameters at runtime then both the tasks that leverage job parameters will use the same new value. However, the default values for each parameter is at the task level, therefore, the default values for each task can be different even if the parameter name is the same. Lastly, if leveraging task parameters then this does not apply as each task will have a set of isolated parameters that are not shared between tasks.

In addition to runtime parameters, Databricks released the ability to pass values from task to task within a job in August 2022. To leverage this capability engineers will use Databricks utility functions. To set and get values see below.

```
dbutils.job.taskValues.set(key='my_key', value='key_value')
dbutils.job.taskValues.get(taskKey='my_key', key='key_value',
                           default='default_value', debugValue='debug_value')
```

When using with task values, it is important to set the `debugValue` as that is used interactively to provide value in a notebook since there is no context of the job during development.

Task values are the best way to pass variable values from one task to another. These variables are often dynamic and depend on the execution of code to determine their values. Runtime parameters for notebooks can be set using Databricks Widgets for each task a job. Unlike task values, widget parameters are set when the job is triggered, scoped to each notebook, and are not typically altered throughout the execution of a given job. See below for an example of creating and getting parameters using widgets.

```
dbutils.widgets.text("my_parameter", "my_default_value")

param_value = dbutils.widgets.get("my_parameter")
```

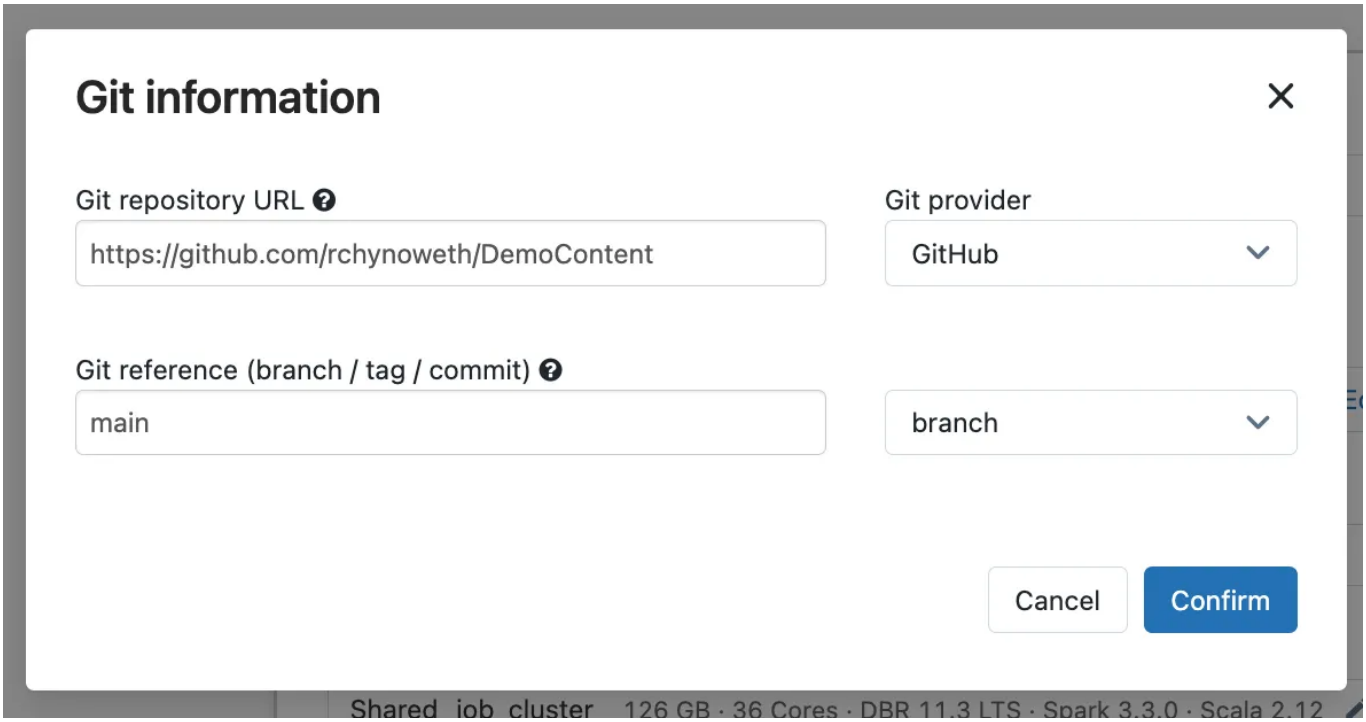
We will walk through how to create jobs with multiple tasks and have tasks pass values between one another. To start we will create the task through the user interface to see how simple that process is, then go into detail around the job definition to understand how to use these features programmatically.

Please note that there are Task Parameter Variables available at the task level to programmatically obtain: `job_id`, `run_id`, `start_date`, `start_time`, `task_retry_count`, `parent_run_id`, and `task_key`.

## Defining a Job

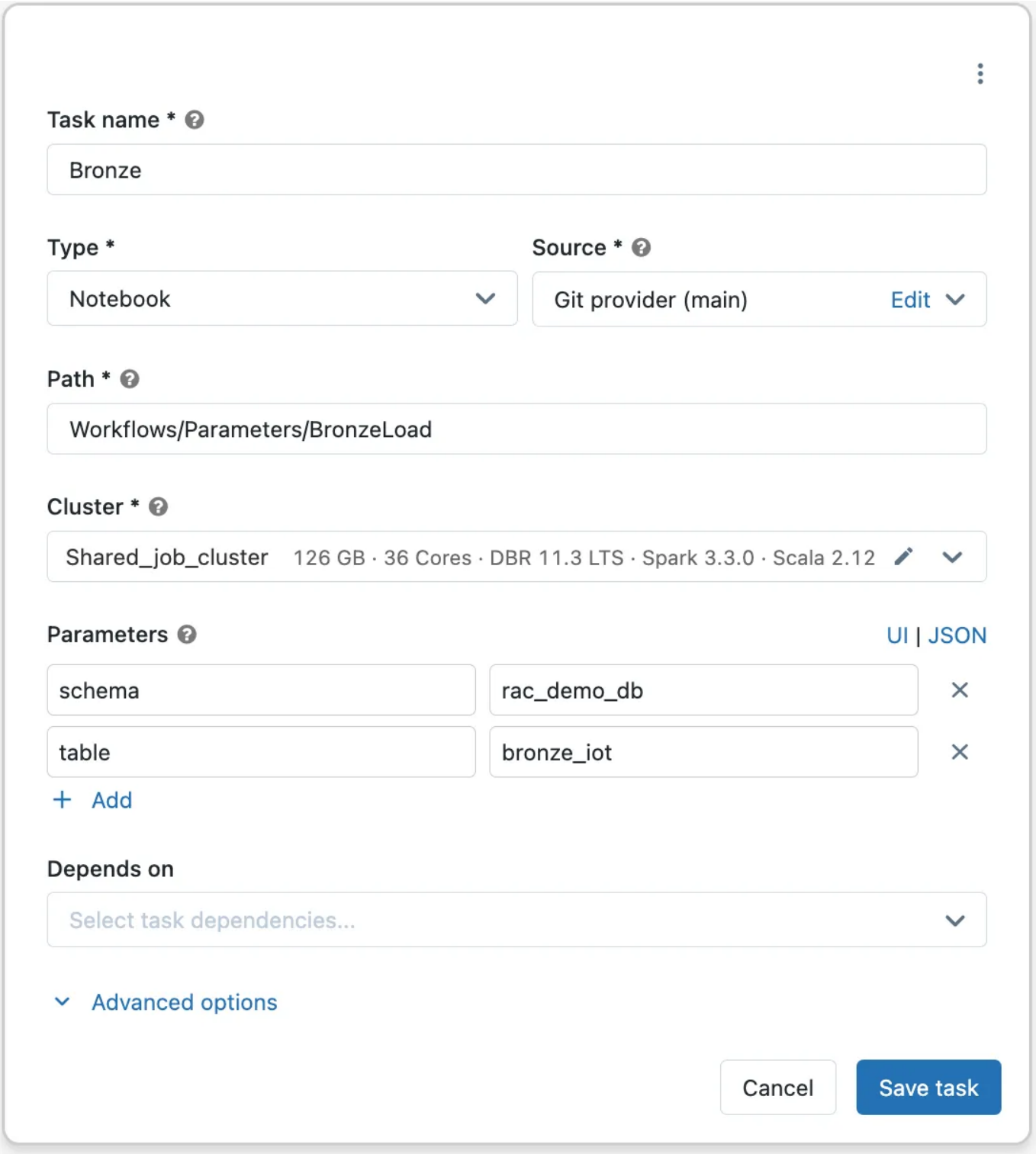
In our example we will create a job with two notebook tasks. Both tasks will have set parameters specifying table object values using widgets. The tasks will have a dependency to run in isolation on a shared job cluster and will connect directly to the git repository to acquire the executable code. Lastly, the bronze task will also pass a variable value to the silver task highlighting the task value feature.

To start, navigate to the Workflows tab in Databricks UI and create a new job. While creating your first task, set the task type to “Notebook” and the source to “Git provider”. Please confirm your git information is correct.



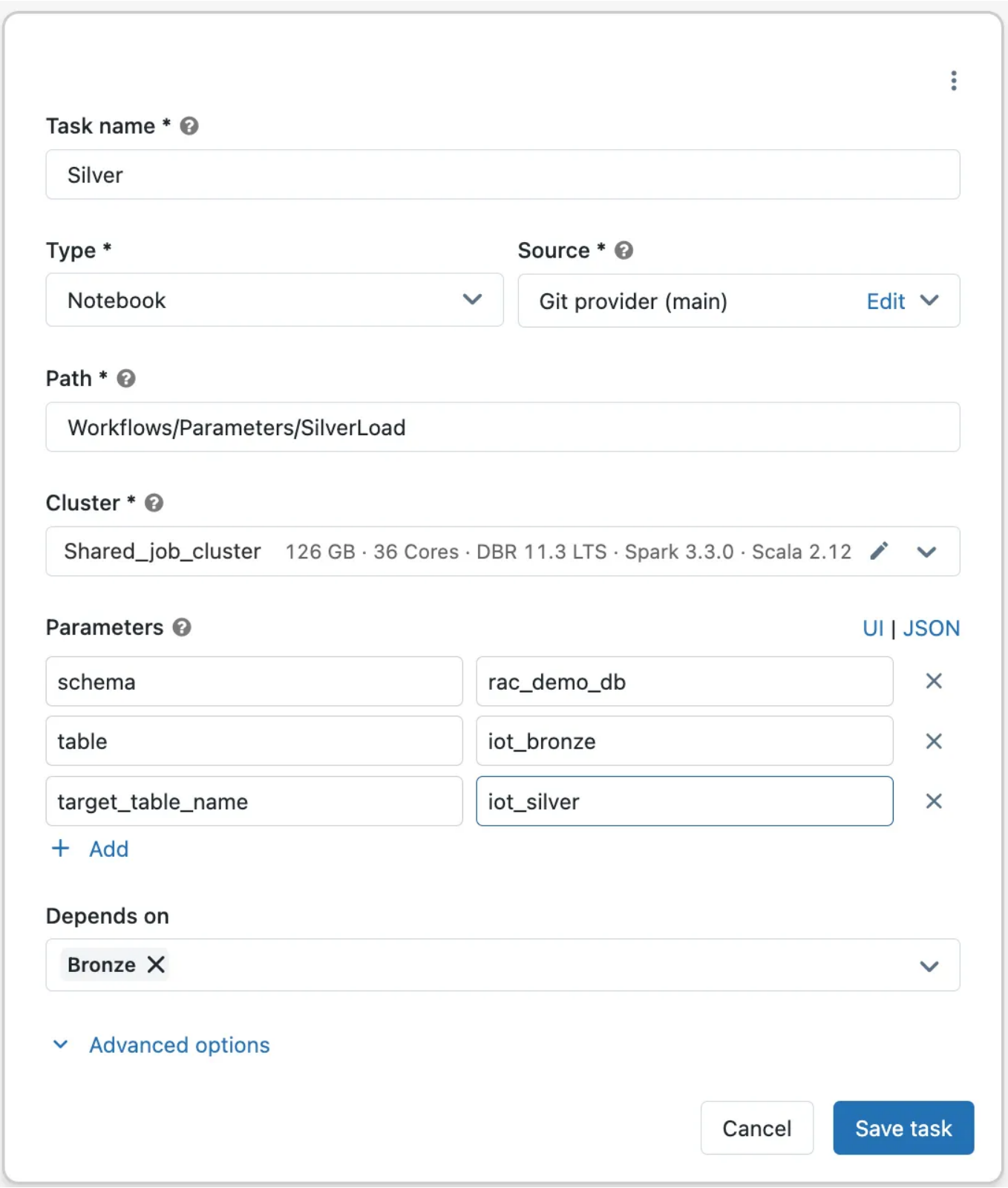

Configure Git Repository for Databricks Job

Next create a task definition as shown below. Notice that the widgets defined in the notebook are passed using the parameters section in the task definition.




Defining the Data Ingestion Task

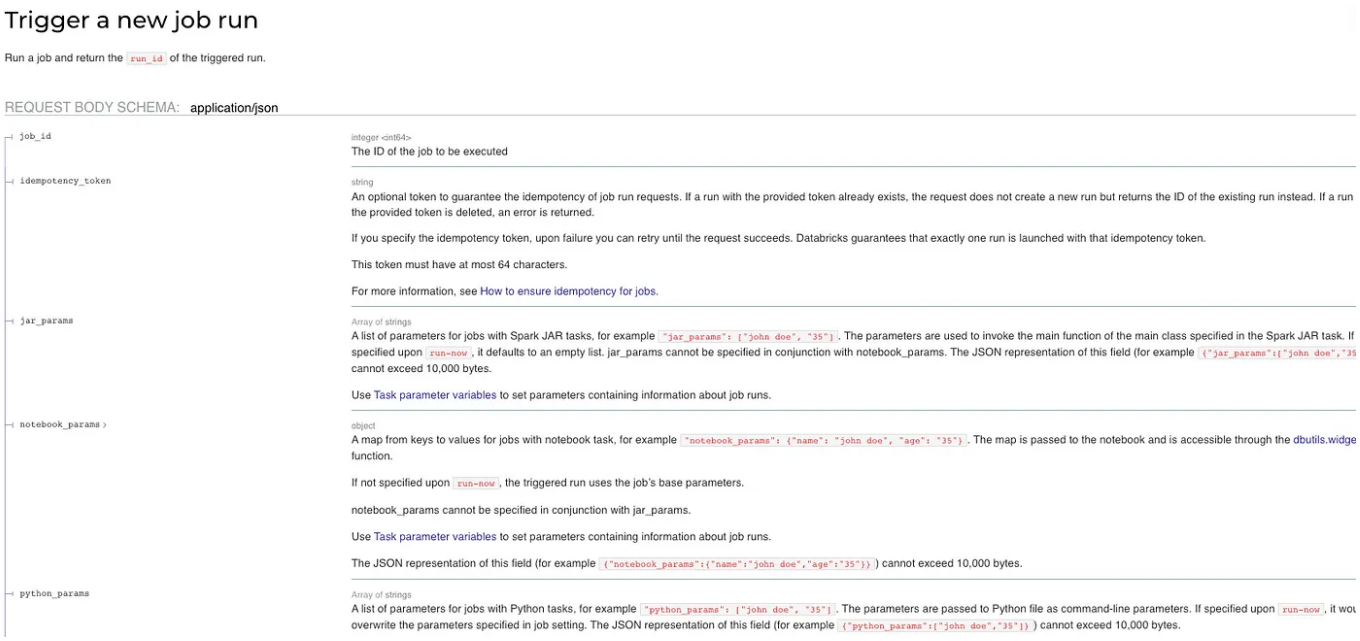
Creating the second task you will notice that we again connect directly to the git repository, reuse the “Shared_job_cluster” from the bronze task, set parameter values, and set the silver task to be dependent on the bronze task.




Defining the Aggregation Task

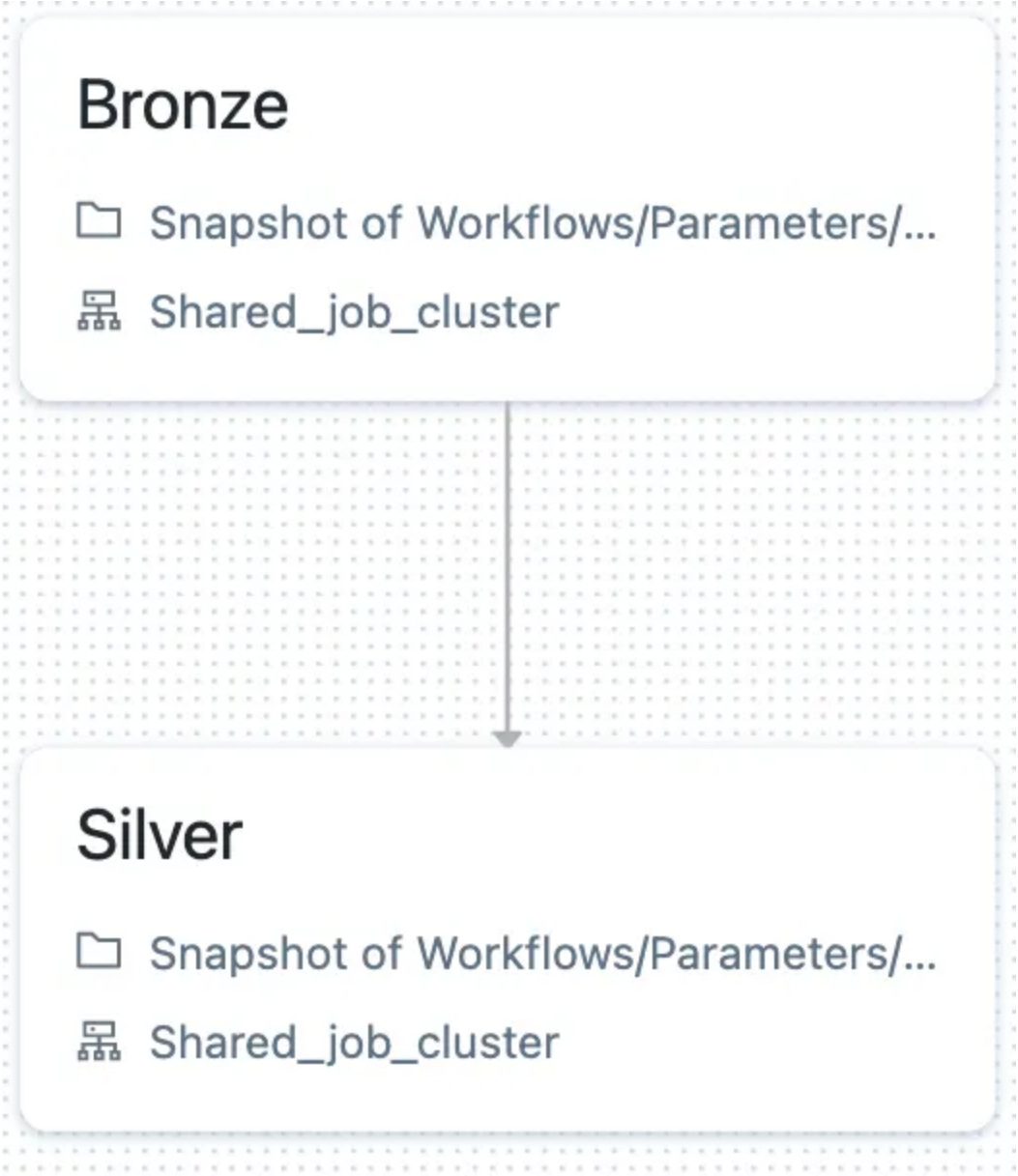
In both of the defined tasks we have set the parameter keys and default values. These values can be overridden when you send a request to run the

job. Please refer to the jobs run now documentation which shows the various parameter objects.




Job Run Now — Runtime Parameters

We end up with a simple Databricks Workflow with two tasks. The bronze task loads raw json files into a bronze Databricks table, then the silver task loads the bronze table to output aggregations using a time window for each device id. Please refer to the notebooks for more detail.




DAG of Databricks Job

While we created this job using the Databricks UI, it is also possible to create and run this job using the Databricks Jobs API. The API is most commonly used for automation and release processes which we will discuss the structure of the job definition now. The simple job shown above results in the following definition.

```
{
    "job_id": 862519602273592,
    "creator_user_name": "ryan.chynoweth@databricks.com",
```

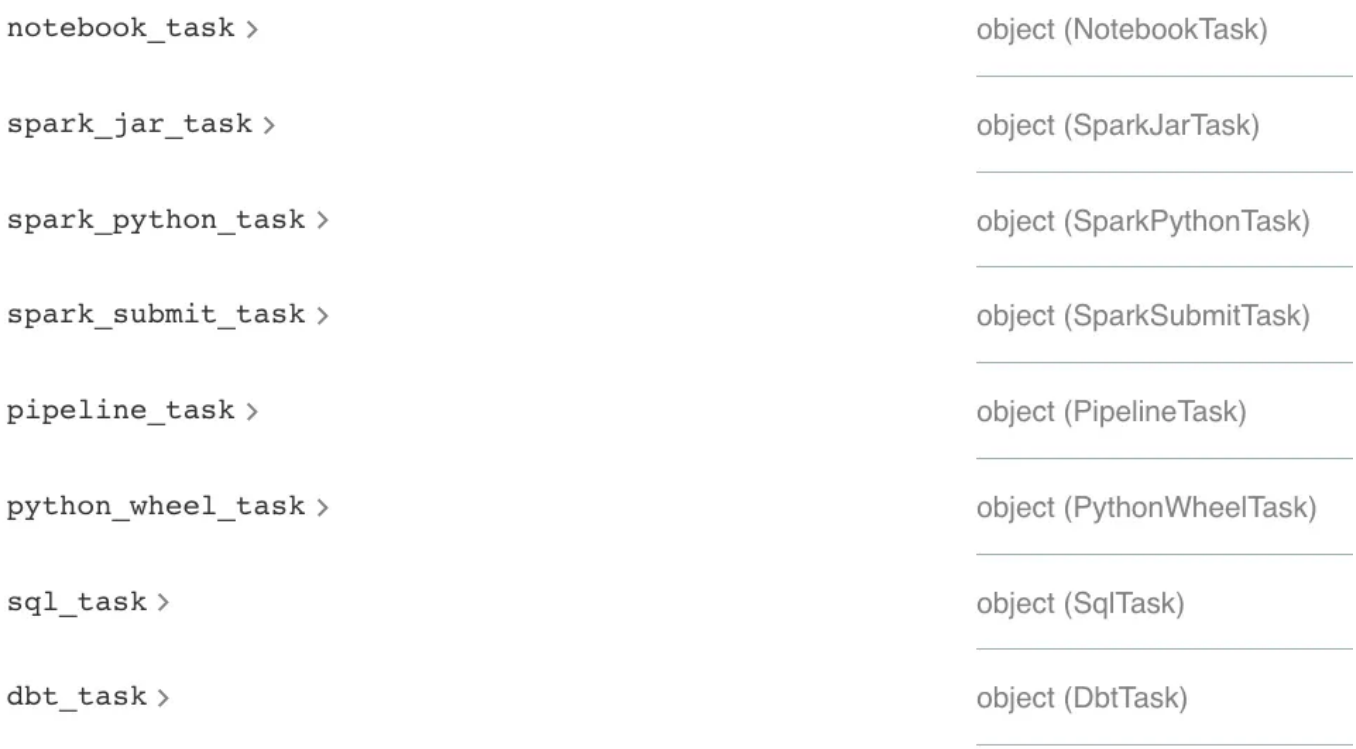
```
"run_as_user_name": "ryan.chynoweth@databricks.com",
"run_as_owner": true,
"settings": {
    "name": "Parameter Blog Job ",
    "email_notifications": {
        "no_alert_for_skipped_runs": false
    },
    "timeout_seconds": 0,
    "max_concurrent_runs": 1,
    "tasks": [
        {
            "task_key": "Bronze",
            "notebook_task": {
                "notebook_path": "Workflows/Parameters/BronzeLoad",
                "base_parameters": {
                    "schema": "rac_demo_db",
                    "table": "bronze_iot"
                },
                "source": "GIT"
            },
            "job_cluster_key": "Shared_job_cluster",
            "timeout_seconds": 0,
            "email_notifications": {}
        },
        {
            "task_key": "Silver",
            "depends_on": [
                {
                    "task_key": "Bronze"
                }
            ],
            "notebook_task": {
                "notebook_path": "Workflows/Parameters/SilverLoad",
                "base_parameters": {
                    "schema": "rac_demo_db",
                    "table": "iot_bronze",
                    "target_table_name": "iot_silver"
                },
                "source": "GIT"
            },
            "job_cluster_key": "Shared_job_cluster",
            "timeout_seconds": 0,
            "email_notifications": {}
        }
    ],
    "job_clusters": [
        {
            "job_cluster_key": "Shared_job_cluster",
            "new_cluster": {
                "cluster_name": "",
```

```
                    "spark_version": "11.3.x-scala2.12",
                    "spark_conf": {
                        "spark.databricks.delta.preview.enabled": "true"
                    },
                    "azure_attributes": {
                        "first_on_demand": 1,
                        "availability": "ON_DEMAND_AZURE",
                        "spot_bid_max_price": -1
                    },
                    "node_type_id": "Standard_DS3_v2",
                    "spark_env_vars": {
                        "PYSPARK_PYTHON": "/databricks/python3/bin/python3"
                    },
                    "enable_elastic_disk": true,
                    "data_security_mode": "SINGLE_USER",
                    "runtime_engine": "STANDARD",
                    "num_workers": 8
                }
            }
        ],
        "git_source": {
            "git_url": "https://github.com/rchynoweth/DemoContent",
            "git_provider": "gitHub",
            "git_branch": "main"
        },
        "format": "MULTI_TASK"
    },
    "created_time": 1670079972376
}
```


The job definition is fairly straightforward but it is important that we highlight a few areas. In the definition of the job there are two objects that are very important: `tasks` and `job_clusters`.

The task object is a json array of task items. Each item has a `task_key` which is the name of the task that we provided in the user interface, a list of dependencies, and task definition by type. Databricks supports running tasks of type: Notebook, Python script, Python wheel, SQL, Delta Live Table pipeline, dbt, jar, and spark-submit. Each type of task has slightly different settings which are defined within the REST API documentation, we will

discuss general guidance that applies to all the different types of tasks with some degree of variation. In the screenshot below you will see a list of the supported tasks. If you are looking at them on the web page then you can expand each object type to view the specific definition for that task type.




List of Databricks Task Objects

Looking at the task object in more detail you will see that the notebook task simply requires a path, a source, a cluster, and parameters. In our case we point the task to a git repository to provide the path and source. Then we define our parameters and set their values which makes these values available using Databricks Widgets. The values provided for the parameters can be left blank or can be overridden at runtime. Lastly, we need to assign compute to our task in order to run the workload.

Focusing on the job clusters object in our definition, you will see that this is simply a list of all clusters that are available within the Databricks job. These clusters are scoped to the individual job and can be used for many tasks.

Clusters can be automated or all-purpose. Automated clusters are created and terminated for the lifetime of the job, and have a lower cost when compared to all-purpose clusters. All-purpose clusters are used for interactive development and provide an abundance of developer features to improve experience and shorten the time to production. It is recommended to use automated clusters for Databricks jobs.

The definition of clusters in a job is straightforward. Simply define the specifications and provide a name for the cluster. It is important to note that the cluster must be present in the list of clusters in order to assign it to a task. Even if you are using an existing all-purpose cluster you will need to ensure that it is within scope.

### Creating a Job Using the API

The definition above is what is provided *after* the job was created using the Databricks UI. If required the job could have been created programmatically using the job create API endpoint.

Here is the definition that we will use to create the same job as above.

```
{
    "name": "Parameter Blog Job ",
    "email_notifications": {
        "no_alert_for_skipped_runs": false
    },
    "timeout_seconds": 0,
    "max_concurrent_runs": 1,
    "tasks": [
        {
            "task_key": "Bronze",
            "notebook_task": {
                "notebook_path": "Workflows/Parameters/BronzeLoad",
                "base_parameters": {
                    "schema": "rac_demo_db",
                    "table": "bronze_iot"
```

```
                    },
                    "source": "GIT"
                },
                "job_cluster_key": "Shared_job_cluster",
                "timeout_seconds": 0,
                "email_notifications": {}
            },
            {
                "task_key": "Silver",
                "depends_on": [
                    {
                        "task_key": "Bronze"
                    }
                ],
                "notebook_task": {
                    "notebook_path": "Workflows/Parameters/SilverLoad",
                    "base_parameters": {
                        "schema": "rac_demo_db",
                        "table": "iot_bronze",
                        "target_table_name": "iot_silver"
                    },
                    "source": "GIT"
                },
                "job_cluster_key": "Shared_job_cluster",
                "timeout_seconds": 0,
                "email_notifications": {}
            }
        ],
        "job_clusters": [
            {
                "job_cluster_key": "Shared_job_cluster",
                "new_cluster": {
                    "spark_version": "11.3.x-scala2.12",
                    "spark_conf": {
                        "spark.databricks.delta.preview.enabled": "true"
                    },
                    "azure_attributes": {
                        "first_on_demand": 1,
                        "availability": "ON_DEMAND_AZURE",
                        "spot_bid_max_price": -1
                    },
                    "node_type_id": "Standard_DS3_v2",
                    "spark_env_vars": {
                        "PYSPARK_PYTHON": "/databricks/python3/bin/python3"
                    },
                    "enable_elastic_disk": true,
                    "data_security_mode": "SINGLE_USER",
                    "runtime_engine": "STANDARD",
                    "num_workers": 8
                }
```

```
            }
        ],
        "git_source": {
            "git_url": "https://github.com/rchynoweth/DemoContent",
            "git_provider": "gitHub",
            "git_branch": "main"
        },
        "format": "MULTI_TASK"
    }
```

Below is a Python script that can be used to create and run the job with a different set of parameters using the Databricks REST API.

```
import requests
import json



create_job_endpoint = "/api/2.1/jobs/create"
job_definition_file = "Workflows/Parameters/APIJobDefinition.json"
pat_token = "<DATABRICKS PERSONAL ACCESS TOKEN>"
workspace_url = "https://adb-123456789123456.78.azuredatabricks.net" # i.e. http


auth = {"Authorization": "Bearer {}".format(pat_token)}


with open(job_definition_file) as f:
    payload = json.load(f)


### Create the job
response = requests.post("{}{}".format(workspace_url, create_job_endpoint), json
assert response.status_code == 200


### Run the job now with DIFFERENT parameters
job_id = json.loads(response.content.decode('utf-8')).get('job_id')
print(f"JOB ID ------------> {job_id}")

job_params = {
    "job_id": job_id,
    "notebook_params": {
```

```
            "schema": "rac_demo_db",
            "table": "iot_bronze_2",
            "target_table_name": "iot_silver_2"
            }
        }


run_now_endpoint = "/api/2.1/jobs/run-now"

response = requests.post("{}{}".format(workspace_url, run_now_endpoint), json=jo
assert response.status_code == 200

json.loads(response.content.decode('utf-8'))
```

If you navigate to the job run in the Databricks UI and select the bronze task, then you will see that the default parameter values were overridden and the task values were set. Notice that we did not name a `target_table_name` in our definition for the task but it is possible to still pass new parameters.


Parameters and Task Values in the Bronze Task

## Conclusion

Databricks Workflows provide an excellent first-party orchestration service with a number of powerful features. In this brief discussion we clearly showed the relationship between jobs, tasks, parameters, and clusters within a Databricks job. Parameterizing tasks on Databricks is an extremely useful

feature that allows developers to reduce the amount of code required and deploy dynamic pipelines that can operate depending on the values provided at runtime.

For reference to the code shown here please refer to this GitHub repository.

*Disclaimer: these are my own thoughts and opinions and not a reflection of my employer.*

Databricks Databricks Workflows

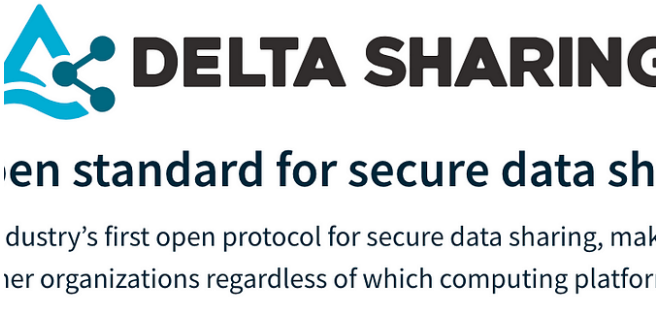
## Written by Ryan Chynoweth

Edit profile

312 Followers

Senior Solutions Architect Databricks — anything shared is my own thoughts and opinions

### More from Ryan Chynoweth



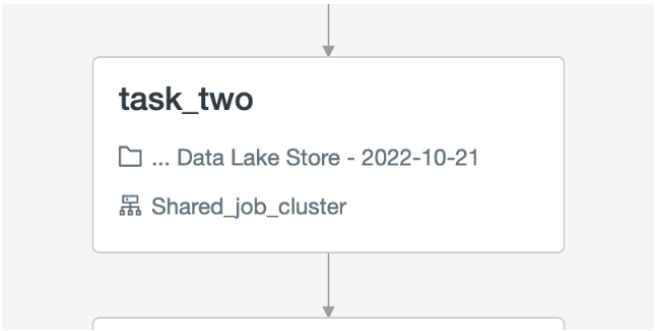

Ryan Chynoweth

## Delta Sharing: An Implementation Guide for Multi-Cloud Architecture

Introduction

8 min read · 3 days ago

3 2




Ryan Chynoweth

## Converting Stored Procedures to Databricks

Special thanks to co-author Kyle Hale, Sr. Specialist Solutions Architect at Databricks.

14 min read · Dec 29, 2022

116 5

Ryan Chynoweth

## Recursive CTE on Databricks

Introduction

3 min read · Apr 20, 2022

33

Ryan Chynoweth

## SQL Variables in Databricks

Last December we published a blog providing an overview of Converting Stored Procedure...

2 min read · Oct 6, 2023

21

See all from Ryan Chynoweth

## Recommended from Medium

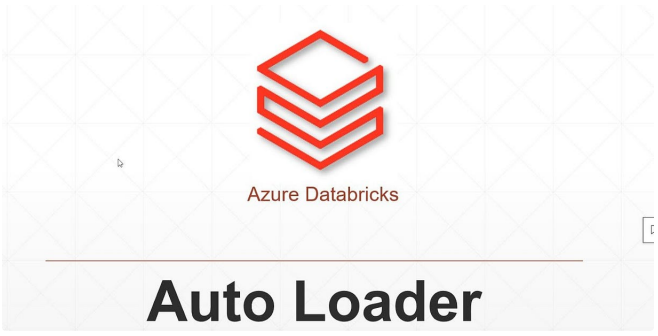
Daan Rademaker

### Do-it-yourself, building your own Databricks Docker Container

In my previous LinkedIn article, I aimed to persuade you of the numerous advantages o...

7 min read · Oct 16, 2023

26




SIRIGIRI HARI KRISHNA in Towards Dev

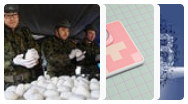
### Auto Loader

Autoloader simplifies reading various data file types from popular cloud locations like...

5 min read · Dec 9, 2023

4 1

## Lists



**Staff Picks**

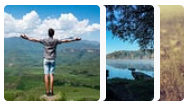
547 stories · 597 saves

**Stories to Help You Level-Up at Work**

19 stories · 395 saves

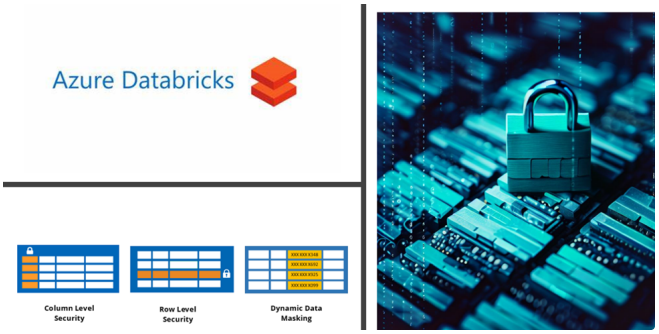
**Self-Improvement 101**

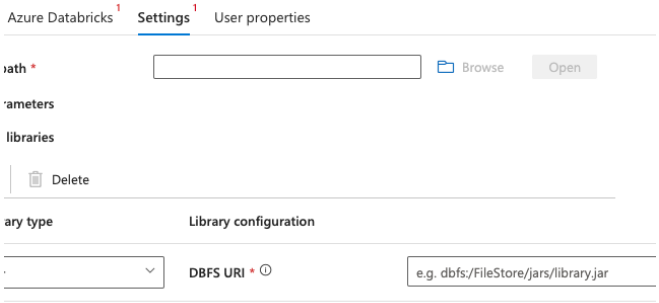
20 stories · 1146 saves



**Productivity 101**

20 stories · 1047 saves




Samarendra Panda

## Dynamic Row Level Filtering and Column Level Masking in Azure...

Background

5 min read · Sep 23, 2023

29

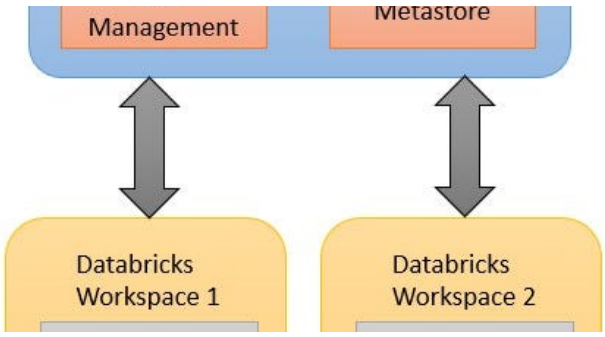
Matt Bradley

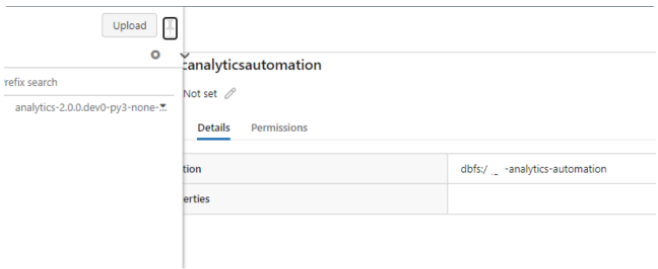
## Azure Data Factory tips for running Databricks jobs

Following on from an earlier blog around using spot instances with Azure Data...

4 min read · Nov 2, 2023

14




Oindrila Chakraborty

## Introduction to “Unity Catalog” in Databricks

What is “Unity Catalog”?

10 min read · Aug 14, 2023

Prashanth Kumar

## Automating .whl File Deployment to Azure Databricks with GitHub...

Introduction

6 min read · Oct 12, 2023

 18  2    2   

See more recommendations