2 Hours to Data Innovation with Cloudera Data Engineering

Feb 07, 2024

Step by Step Guide

Requirements

In order to execute the Innovation Lab you need:

- A Spark 3 and Iceberg-enabled CDE Virtual Cluster (Azure, AWS and Private Cloud).
- No script code changes are required other than entering your Username as a parameter for the jobs.
- Familiarity with Python and PySpark is highly recommended.
- The files contained in the data folder should be manually loaded in the Storage Location of choice. If you are attending an Innovation Lab Workshop, this will already have been done for you. Please validate this with your Cloudera Workshop Lead.

Project Setup

Clone this GitHub repository to your local machine or the VM where you will be running the script.

```
mkdir ~/Documents/cde_innovation_hol
cd ~/Documents/cde_innovation_hol
git clone https://github.com/pdefusco/CDE_Innovation_Lab.git
```

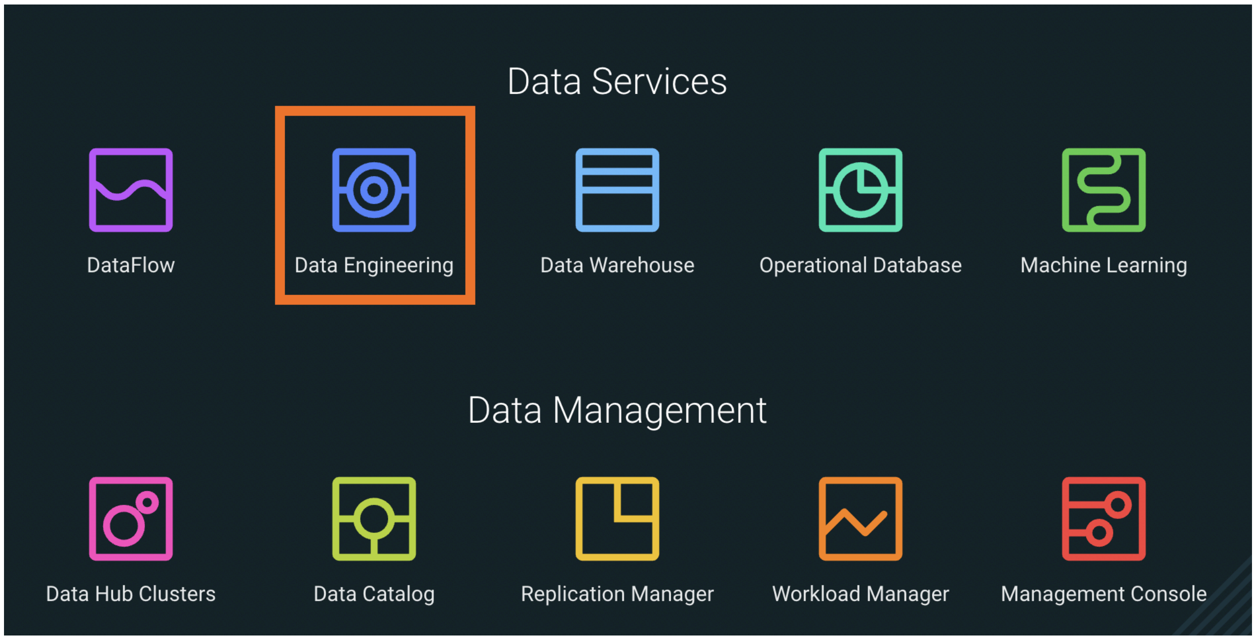
Alternatively, if you don't have `git` installed on your machine, create a folder on your local computer; navigate to this URL and manually download the files.

Introduction to the CDE Data Service

Cloudera Data Engineering (CDE) is a serverless service for Cloudera Data Platform that allows you to submit batch jobs to auto-scaling virtual clusters. CDE enables you to spend more time on your applications, and less time on infrastructure.

Cloudera Data Engineering allows you to create, manage, and schedule Apache Spark jobs without the overhead of creating and maintaining Spark clusters. With Cloudera Data Engineering, you define virtual clusters with a range of CPU and memory resources, and the cluster scales up and down as needed to run your Spark workloads, helping to control your cloud costs.

The CDE Service can be reached from the CDP Home Page by clicking on the blue "Data Engineering" icon.



The CDE Landing Page allows you to access, create and manage CDE Virtual Clusters. Within each CDE Virtual Cluster you can create, monitor and troubleshoot Spark and Airflow Jobs.

The Virtual Cluster is pegged to the CDP Environment. Each CDE Virtual Cluster is mapped to at most one CDP Environment while a CDP Environment can map to one or more Virtual Cluster.

These are the most important components in the CDE Service:

CDP Environment

A logical subset of your cloud provider account including a specific virtual network. CDP Environments can be in AWS, Azure, RedHat OCP and Cloudera ECS. For more information, see CDP Environments. Practically speaking, an environment is equivalent to a Data Lake as each environment is automatically associated with its own SDX services for Security, Governance and Lineage.

CDE Service

The long-running Kubernetes cluster and services that manage the virtual clusters. The CDE service must be enabled on an environment before you can create any virtual clusters.

Virtual Cluster

An individual auto-scaling cluster with defined CPU and memory ranges. Virtual Clusters in CDE can be created and deleted on demand. Jobs are associated with clusters.

Jobs

Application code along with defined configurations and resources. Jobs can be run on demand or scheduled. An individual job execution is called a job run.

Resource

A defined collection of files such as a Python file or application JAR, dependencies, and any other reference files required for a job.

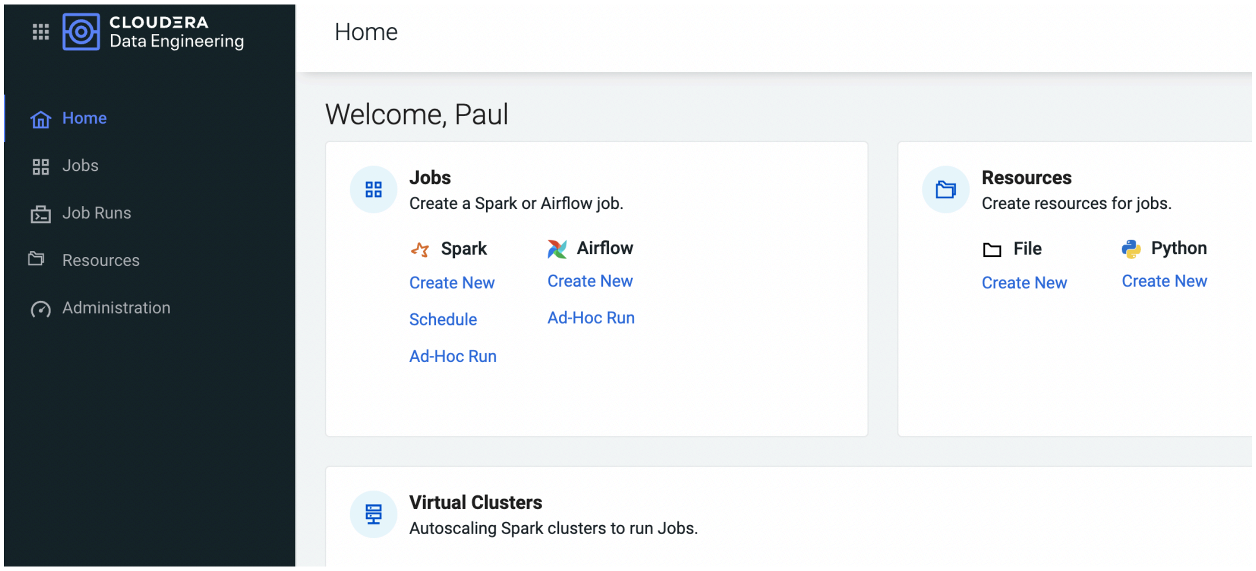
Job Run

An individual job run.

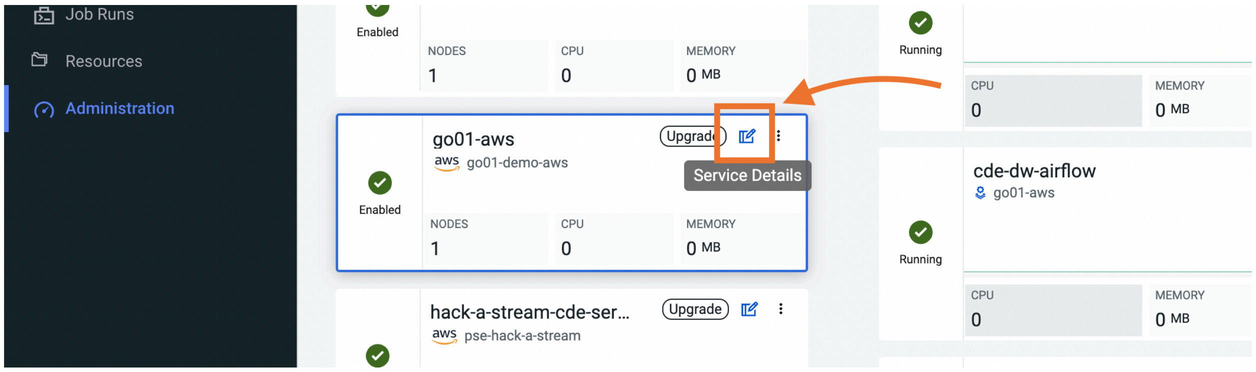
CDE User Interface

Now that you have covered the basics of CDE, spend a few moments familiarizing yourself with the CDE Landing page.

The Home Page provides a high level overview of all CDE Services and Clusters. At the top, you have shortcuts to creating CDE Jobs and Resources. Scroll down to the CDE Virtual Clusters section and notice that all Virtual Clusters and each associated CDP Environment / CDE Service are shown.

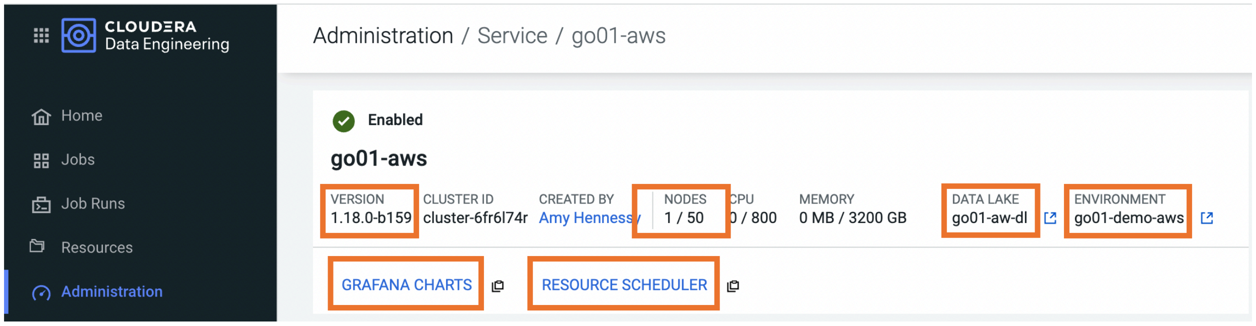


Next, open the Administration page on the left tab. This page also shows CDE Services on the left and associated Virtual Clusters on the right.



Open the CDE Service Details page and notice the following key information and links:

- CDE Version
- Nodes Autoscale Range
- CDP Data Lake and Environment
- Grafana Charts. Click on this link to obtain a dashboard of running Service Kubernetes resources.
- Resource Scheduler. Click on this link to view the Yunikorn Web UI.



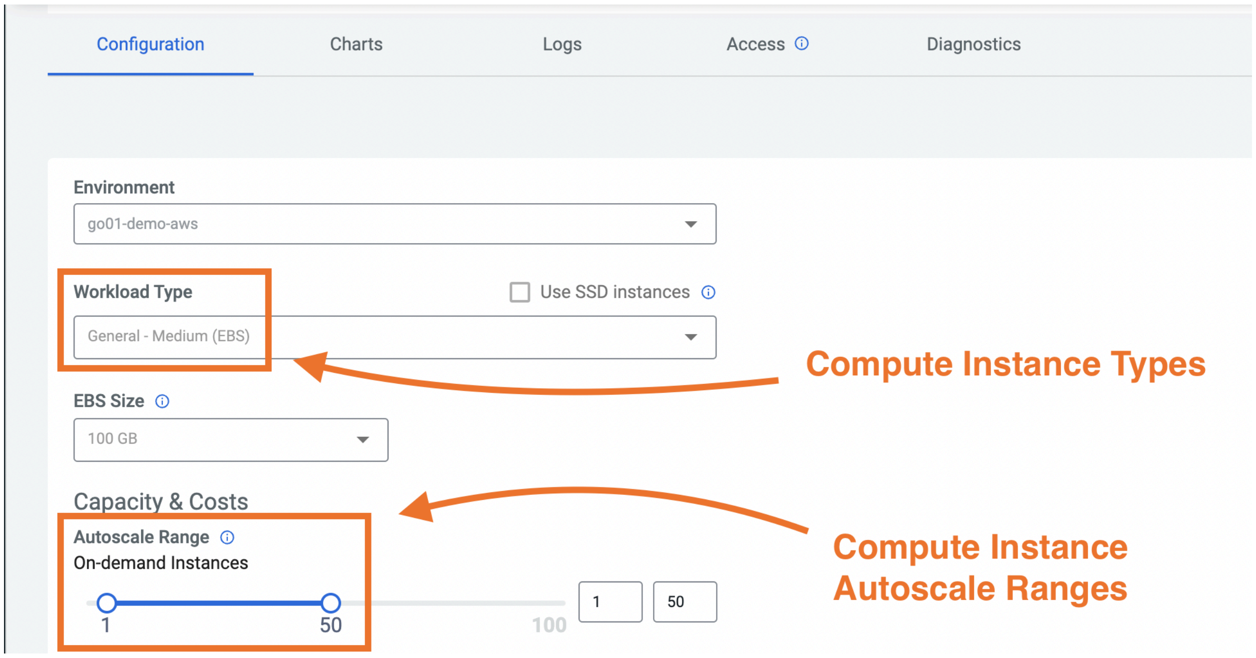
Administration / Service / go01-aws

Enabled

go01-aws

VERSION	CLUSTER ID	CREATED BY	NODES	CPU	MEMORY	DATA LAKE	ENVIRONMENT
1.18.0-b159	cluster-6fr6l74r	Amy Hennessy	1 / 50	0 / 800	0 MB / 3200 GB	go01-aw-dl	go01-demo-aws

Scroll down and open the Configurations tab. Notice that this is where Instance Types and Instance Autoscale ranges are defined.



Environment: go01-demo-aws

Workload Type: General - Medium (EBS)

Use SSD instances

EBS Size: 100 GB

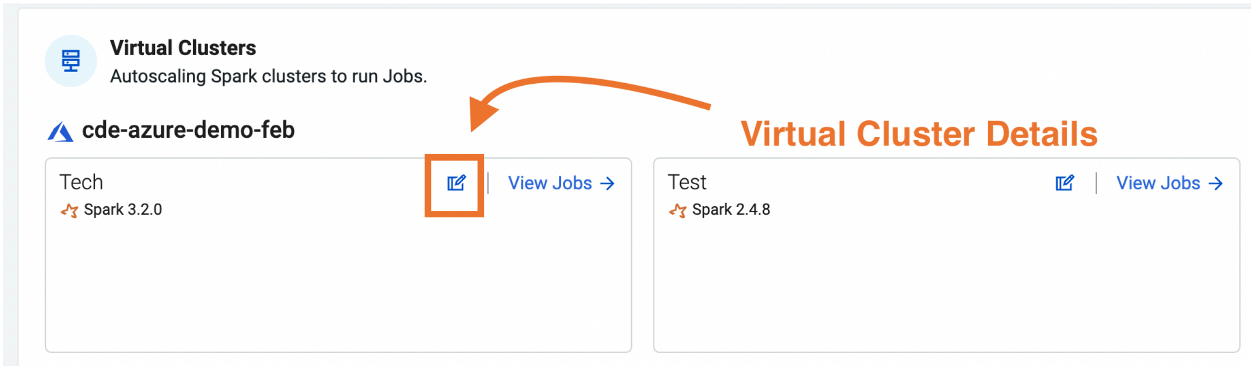
Capacity & Costs

Autoscale Range

On-demand Instances: 1 – 50 (max 100)

To learn more about other important service configurations please visit Enabling a CDE Service in the CDE Documentation.

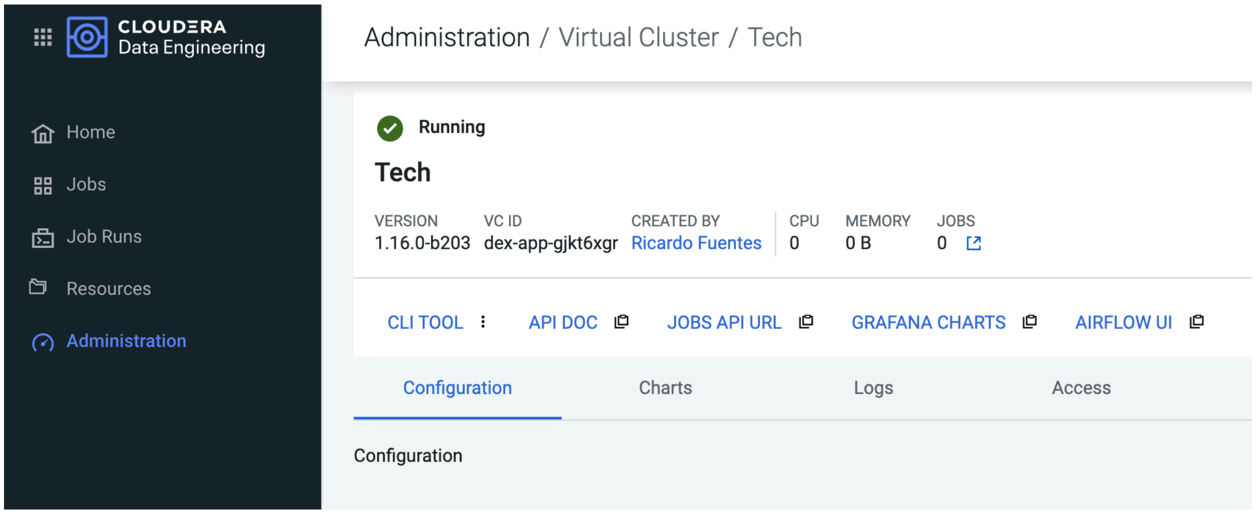
Navigate back to the Administration page and open a Virtual Cluster's Cluster Details page.



This view includes other important cluster management information. From here you can:

- Download the CDE CLI binaries. The CLI is recommended to submit jobs and interact with CDE. It is covered in Part 3 of this guide.
- Visit the API Docs to learn the CDE API and build sample requests on the Swagger page.
- Access the Airflow UI to monitor your Airflow Jobs, set up custom connections, variables, and more.

Open the Configuration tab. Notice that CPU and Memory autoscale ranges, Spark version, and Iceberg options are set here.



VERSION	VC ID	CREATED BY	CPU	MEMORY	JOBS
1.16.0-b203	dex-app-gjkt6xgr	Ricardo Fuentes	0	0 B	0

To learn more about CDE Architecture please visit Creating and Managing Virtual Clusters and Recommendations for Scaling CDE Deployments

Note
A CDE Service defines compute instance types, instance autoscale ranges and the

associated CDP Data Lake. The Data and Users associated with the Service are constrained by SDX and the CDP Environment settings.

Note
Within a CDE Service you can deploy one or more CDE Virtual Clusters. The Service Autoscale Range is a count of min/max allowed Compute Instances. The Virtual Cluster Autoscale Range is the min/max CPU and Memory that can be utilized by all CDE Jobs within the cluster. The Virtual Cluster Autoscale Range is naturally bounded by the CPU and Memory available at the Service level.

Note
This flexible architecture allows you to isolate your workloads and limit access within different autoscaling compute clusters while predefining cost management guardrails at an aggregate level. For example, you can define Services at an organization level and Virtual Clusters within them as DEV, QA, PROD, etc.

Note
CDE takes advantage of YuniKorn resource scheduling and sorting policies, such as gang scheduling and bin packing, to optimize resource utilization and improve cost efficiency. For more information on gang scheduling, see the Cloudera blog post Spark on Kubernetes – Gang Scheduling with YuniKorn.

Note
CDE Spark Job auto-scaling is controlled by Apache Spark dynamic allocation. Dynamic allocation scales job executors up and down as needed for running jobs. This can provide large performance benefits by allocating as many resources as needed by the running job, and by returning resources when they are not needed so that concurrent jobs can potentially run faster.

Part 1: Development & Deployment with CDE

Summary

In this section you will be able to run Spark code interactively using the feature CDE Sessions. Then you will run some useful Spark and Iceberg commands to query your data with ease. These commands will allow you to read data directly from S3 object store and interact with the Data Lakehouse tables to do some analytics.

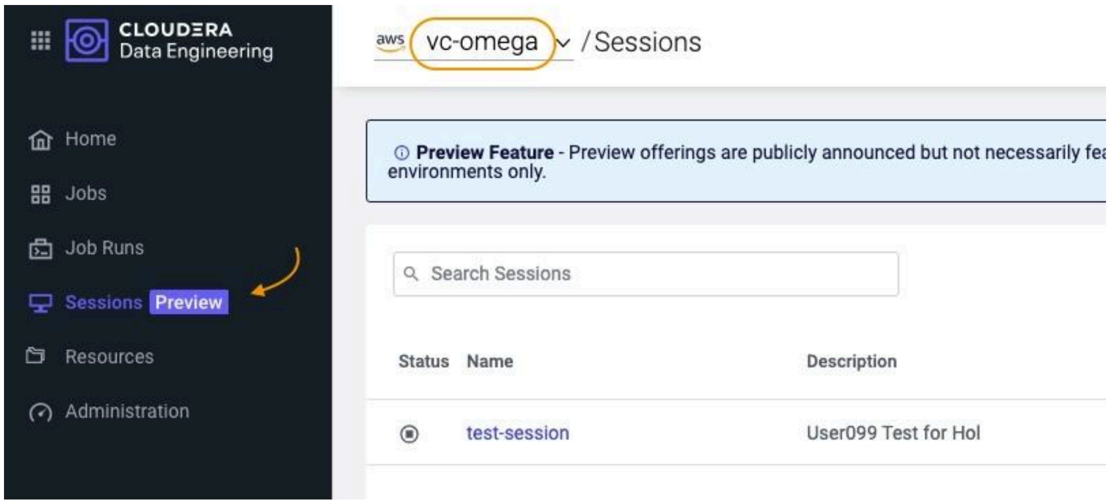
Recommendations Before you Start

⚠ Warning
CDE Sessions allow you to determine the timeout for the session. Set a timeout that will allow you to execute the workshop. When you start the session, it will be in "starting" state for a few moments, just wait until the session is ready to interact.

Lab 1: Using Interactive Sessions in the CDE

A CDE Session is an interactive short-lived development environment for running Spark commands to help you iterate upon and build your Spark workloads. You can launch CDE Sessions in two ways: from the CDE UI and from your terminal with the CLI.

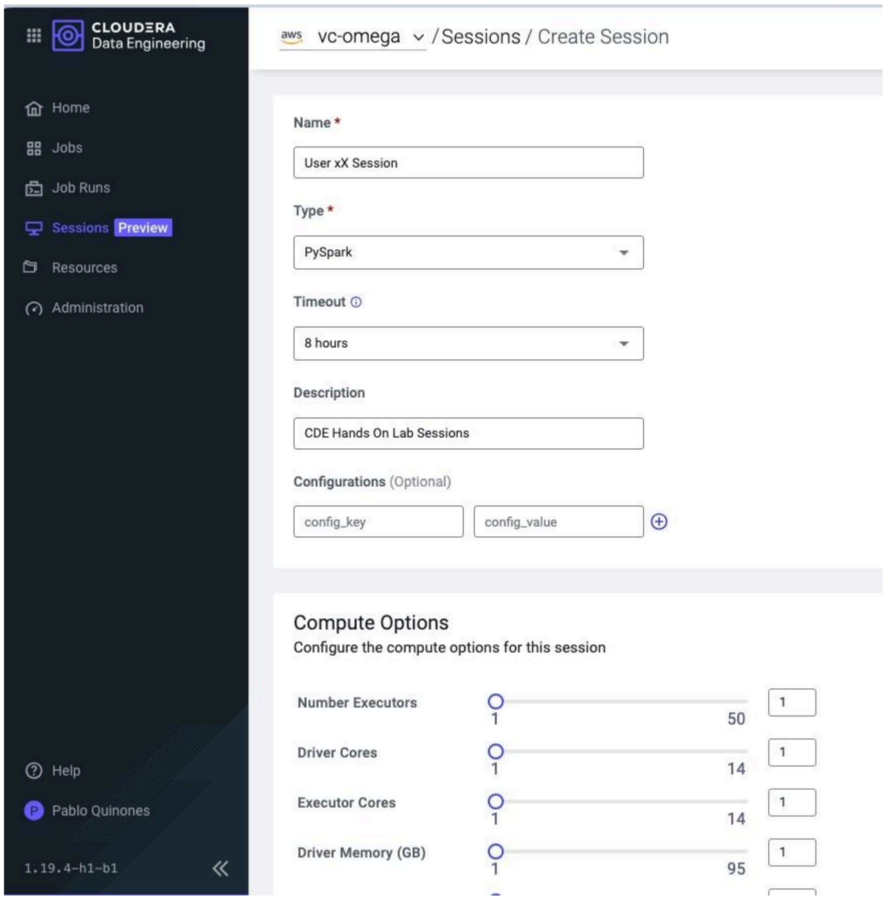
From the CDE Landing Page open "Sessions" on the left pane and then select the CDE Virtual Cluster where you want to run your CDE Interactive Session.



Preview Feature - Preview offerings are publicly announced but not necessarily fea
environments only.

Status	Name	Description
●	test-session	User099 Test for Hol

Click on “Create Session” to create a new CDE Session.



Name *

User xX Session

Type *

PySpark

Timeout

8 hours

Description

CDE Hands On Lab Sessions

Configurations (Optional)

config_key config_value

Compute Options

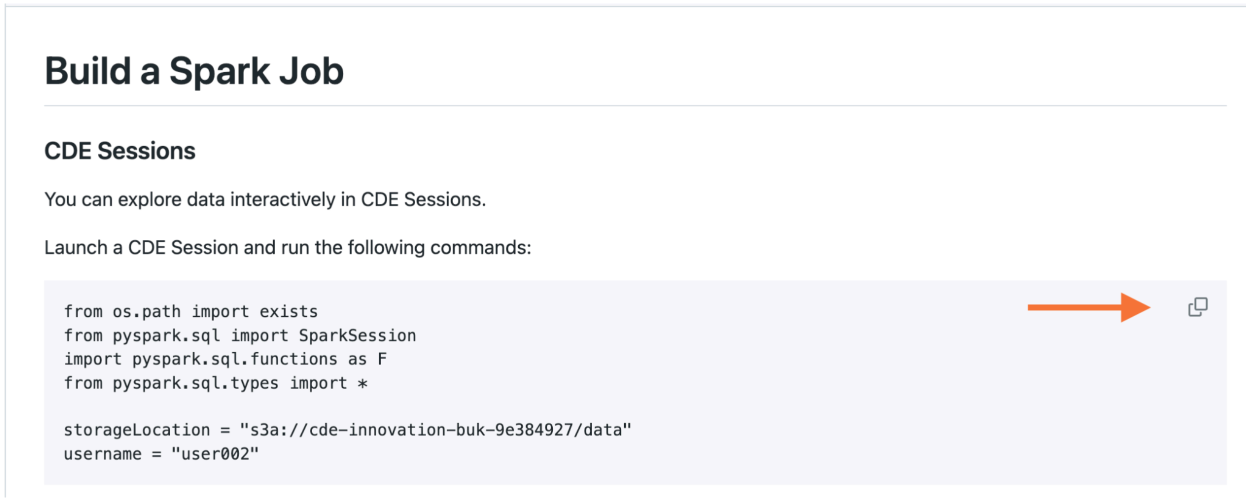
Configure the compute options for this session

Number Executors	1	50	1
Driver Cores	1	14	1
Executor Cores	1	14	1
Driver Memory (GB)	1	95	1

Fill out the name and select “pyspark” type. You can set the desired timeout, this could help optimize costs for the platform admins. For now, you can leave it on the default “8 hours”.

The session will be in "starting" state for a few moments. When it's ready, launch it and open the Spark Shell by clicking on the "Interact" tab.

Copy and paste the following code snippets in each cell and observe the output (no code changes required).



Build a Spark Job

CDE Sessions

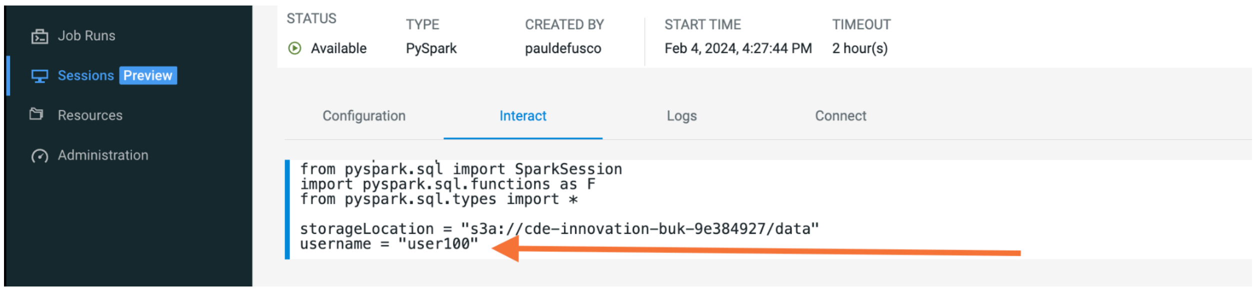
You can explore data interactively in CDE Sessions.

Launch a CDE Session and run the following commands:

```
from os.path import exists
from pyspark.sql import SparkSession
import pyspark.sql.functions as F
from pyspark.sql.types import *

storageLocation = "s3a://cde-innovation-buk-9e384927/data"
username = "user002"
```

Copy the following cell into the notebook. Before running it, ensure that you have edited the "username" variable with your assigned user.



STATUS	TYPE	CREATED BY	START TIME	TIMEOUT
Available	PySpark	pauldefusco	Feb 4, 2024, 4:27:44 PM	2 hour(s)

```
from pyspark.sql import SparkSession
import pyspark.sql.functions as F
from pyspark.sql.types import *

storageLocation = "s3a://cde-innovation-buk-9e384927/data"
username = "user100"
```

No more code edits are required. Continue running each code snippet below in separate cells in the notebook.

Note:
CDE Sessions do not require creating the SparkSession object. The shell has already been launched for you. However, if you need to import any types or functions you do have to import the necessary modules.

Import PySpark packages and set the “storageLocation” and “username” variables based on the user assigned to you.

Python
```python
from os.path import exists
from pyspark.sql import SparkSession
```

```
import pyspark.sql.functions as F
from pyspark.sql.types import *

storageLocation = "s3a://cde-innovation-buk-9e384927/data"
username = "user0xX"
```

First, you can use the provided location to read some raw transaction data from the object storage of the data lakehouse.

```
Python

### LOAD HISTORICAL TRANSACTIONS FILE FROM CLOUD STORAGE
transactionsDf =
spark.read.json("{0}/mkthol/trans/{1}/transactions".format(storageLocation,
username))
transactionsDf.printSchema()
```

You can create user defined functions and can create structures for the data frames and apply the functions:

```
Python
### CREATE PYTHON FUNCTION TO FLATTEN PYSPARK DATAFRAME NESTED STRUCTS
def flatten_struct(schema, prefix=""):
    result = []
    for elem in schema:
        if isinstance(elem.dataType, StructType):
            result += flatten_struct(elem.dataType, prefix + elem.name + ".")
        else:
            result.append(F.col(prefix + elem.name).alias(prefix + elem.name))
    return result
```

```
Python
### RUN PYTHON FUNCTION TO FLATTEN NESTED STRUCTS AND VALIDATE NEW SCHEMA
transactionsDf = transactionsDf.select(flatten_struct(transactionsDf.schema))
transactionsDf.printSchema()
```

Columns Renaming:

```
Python
### RENAME COLUMNS
transactionsDf =
transactionsDf.withColumnRenamed("transaction.transaction_amount",
"transaction_amount")
transactionsDf =
transactionsDf.withColumnRenamed("transaction.transaction_currency",
"transaction_currency")
transactionsDf =
transactionsDf.withColumnRenamed("transaction.transaction_type",
"transaction_type")
transactionsDf =
transactionsDf.withColumnRenamed("transaction_geolocation.latitude",
"latitude")
transactionsDf =
transactionsDf.withColumnRenamed("transaction_geolocation.longitude",
"longitude")
```

Type Casting:

```
Python
### CAST COLUMN TYPES FROM STRING TO APPROPRIATE TYPE
transactionsDf = transactionsDf.withColumn("transaction_amount",
transactionsDf["transaction_amount"].cast('float'))
transactionsDf = transactionsDf.withColumn("latitude",
transactionsDf["latitude"].cast('float'))
transactionsDf = transactionsDf.withColumn("longitude",
transactionsDf["longitude"].cast('float'))
transactionsDf = transactionsDf.withColumn("event_ts",
transactionsDf["event_ts"].cast("timestamp"))
```

After preparing the objects, you can start generating reports and analyzing the data:

```
Python
### CALCULATE MEAN AND MEDIAN CREDIT CARD TRANSACTION AMOUNT
```

```
transactionsAmountMean =
round(transactionsDf.select(F.mean("transaction_amount")).collect()[0][0],2)
transactionsAmountMedian =
round(transactionsDf.stat.approxQuantile("transaction_amount", [0.5],
0.001)[0],2)

print("Transaction Amount Mean: ", transactionsAmountMean)
print("Transaction Amount Median: ", transactionsAmountMedian)
```

List of 10 transactions:

Python
```
### CREATE SPARK TEMPORARY VIEW FROM DATAFRAME
transactionsDf.createOrReplaceTempView("trx")
spark.sql("SELECT * FROM trx LIMIT 10").show()
```

Average transaction amount per month:

Python
```
### CREATE SPARK TEMPORARY VIEW FROM DATAFRAME
transactionsDf.createOrReplaceTempView("trx")
spark.sql("SELECT * FROM trx LIMIT 10").show()
```

Average transaction amount by day of the week:

Python
```
### CALCULATE AVERAGE TRANSACTION AMOUNT BY MONTH
spark.sql("SELECT MONTH(event_ts) AS month, \
          avg(transaction_amount) FROM trx GROUP BY month ORDER BY
month").show()
```

Total number of transactions by credit card:

```
Python
### CALCULATE NUMBER OF TRANSACTIONS BY CREDIT CARD
spark.sql("SELECT CREDIT_CARD_NUMBER, COUNT(*) AS COUNT FROM trx \
            GROUP BY CREDIT_CARD_NUMBER ORDER BY COUNT DESC LIMIT 10").show()
```

Now, you will access a second set of data files from the data lakehouse for analytics:

```
Python
### LOAD CUSTOMER PII DATA FROM CLOUD STORAGE
piiDf = spark.read.options(header='True',
delimiter=',').csv("{0}/mkthol/pii/{1}/pii".format(storageLocation,
username))
piiDf.show()
piiDf.printSchema()
```

Similar to the other data objects, you can do data transformations and casting of data types:

```
Python
### CAST LAT LON TO FLOAT TYPE AND CREATE TEMPORARY VIEW
piiDf = piiDf.withColumn("address_latitude",
piiDf["address_latitude"].cast('float'))
piiDf = piiDf.withColumn("address_longitude",
piiDf["address_longitude"].cast('float'))
piiDf.createOrReplaceTempView("cust_info")
```

With the new dataset, you can perform multiple analytic operations:

```
Python
### SELECT TOP 100 CUSTOMERS WITH MULTIPLE CREDIT CARDS SORTED BY NUMBER OF
CREDIT CARDS FROM HIGHEST TO LOWEST
spark.sql("SELECT name AS name, \
          COUNT(credit_card_number) AS CC_COUNT FROM cust_info GROUP BY name
ORDER BY CC_COUNT DESC \
          LIMIT 100").show()
```

Top100 Credit cards with multiple names, sorted by highest amount:

```python
### SELECT TOP 100 CREDIT CARDS WITH MULTIPLE NAMES SORTED FROM HIGHEST TO
LOWEST
spark.sql("SELECT COUNT(name) AS NM_COUNT, \
          credit_card_number AS CC_NUM FROM cust_info GROUP BY CC_NUM ORDER BY
NM_COUNT DESC \
          LIMIT 100").show()
```

Top25 Customers with multiple addresses:

```python
# SELECT TOP 25 CUSTOMERS WITH MULTIPLE ADDRESSES SORTED FROM HIGHEST TO LOWEST
spark.sql("SELECT name AS name, \
          COUNT(address) AS ADD_COUNT FROM cust_info GROUP BY name ORDER BY
ADD_COUNT DESC \
          LIMIT 25").show()
```

More additional complex operations, like Joins, can be executed in the sessions:

```python
### JOIN DATASETS AND COMPARE CREDIT CARD OWNER COORDINATES WITH
TRANSACTION COORDINATES
joinDf = spark.sql("""SELECT i.name, i.address_longitude,
i.address_latitude, i.bank_country,
          r.credit_card_provider, r.event_ts, r.transaction_amount,
r.longitude, r.latitude
          FROM cust_info i INNER JOIN trx r
          ON i.credit_card_number == r.credit_card_number;""")
joinDf.show()
```

User defined function to calculate the distance between the transaction location and the address registered:

```
Python
### CREATE PYSPARK UDF TO CALCULATE DISTANCE BETWEEN TRANSACTION AND HOME
LOCATIONS
distanceFunc = F.udf(lambda arr:
(((arr[2]-arr[0])**2)+((arr[3]-arr[1])**2)**(1/2)), FloatType())
distanceDf = joinDf.withColumn("trx_dist_from_home",
distanceFunc(F.array("latitude", "longitude","address_latitude",
"address_longitude")
```

Use the function to analyze the transactions and perform a fraud detection process:

```
Python
### SELECT CUSTOMERS WHOSE TRANSACTION OCCURRED MORE THAN 100 MILES FROM HOME
distanceDf.filter(distanceDf.trx_dist_from_home > 100).show()
```

NOTE: You can keep the session open to execute some examples in “Part 2: Open Data Lakehouse with Iceberg on CDE”

Lab 2: Create CDE Resources and Run CDE Spark Job

Up until now you used Sessions to interactively explore data. CDE also allows you to run Spark Application code in batch as a CDE Job. There are two types of CDE Jobs: Spark and Airflow. In this lab we will create a CDE Spark Job and revisit Airflow later in part 3.

The CDE Spark Job is an abstraction over the Spark Submit. With the CDE Spark Job you can create a reusable, modular Spark Submit definition that is saved in CDE and can be modified in the CDE UI (or via the CDE CLI and API) before every run according to your needs. CDE stores the job definition for each run in the Job Runs UI so you can go back and refer to it long after your job has completed.

Furthermore, CDE allows you to directly store artifacts such as Python files, Jars and other dependencies, or create Python environments and Docker containers in CDE as "CDE Resources". Once created in CDE, Resources are available to CDE Jobs as modular components of the CDE Job definition which can be swapped and referenced by a particular job run as needed.

These features dramatically reduce the amount of work and effort normally required to manage and monitor Spark Jobs in a Spark Cluster. By providing a unified view over all your runs along with the associated artifacts and dependencies, CDE streamlines CI/CD pipelines and removes the need for glue code in your Spark cluster.

In the next steps we will see these benefits in actions.

Create CDE Python Resource

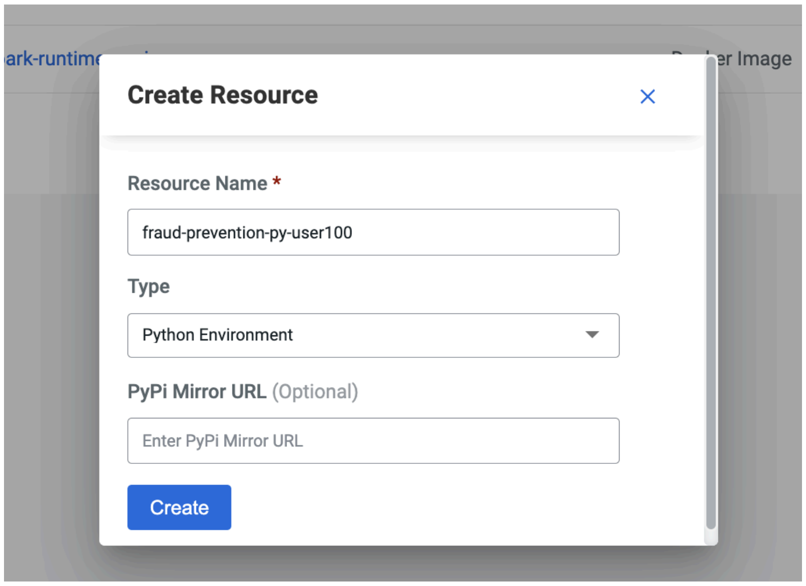
Navigate to the Resources tab and create a Python Resource. Make sure to select the Virtual Cluster assigned to you if you are creating a Resource from the CDE Home Page, and to name the Python Resource after your username

e.g. "fraud-prevention-py-user100" if you are *"user100".*

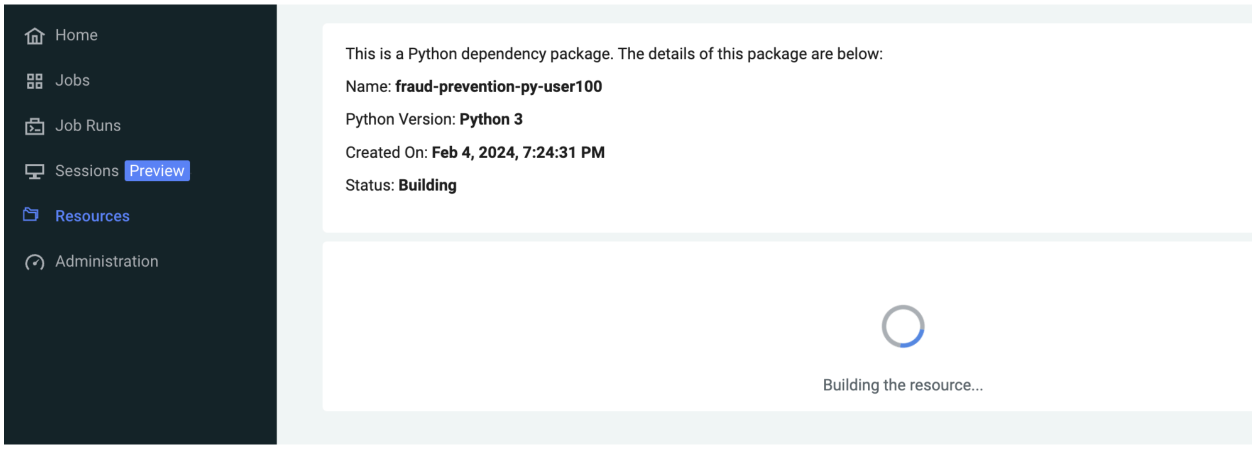
Upload the "requirements.txt" file located in the "cde_spark_jobs" folder. This can take up to a few minutes.

Please familiarize yourself with the contents of the "requirements.txt" file and notice that it contains a few Python libraries such as Pandas and PyArrow.

Then, move on to the next section even while the environment build is still in progress.



Resource Name: fraud-prevention-py-user100
Type: Python Environment



This is a Python dependency package. The details of this package are below:

Name: **fraud-prevention-py-user100**

Python Version: **Python 3**

Created On: **Feb 4, 2024, 7:24:31 PM**

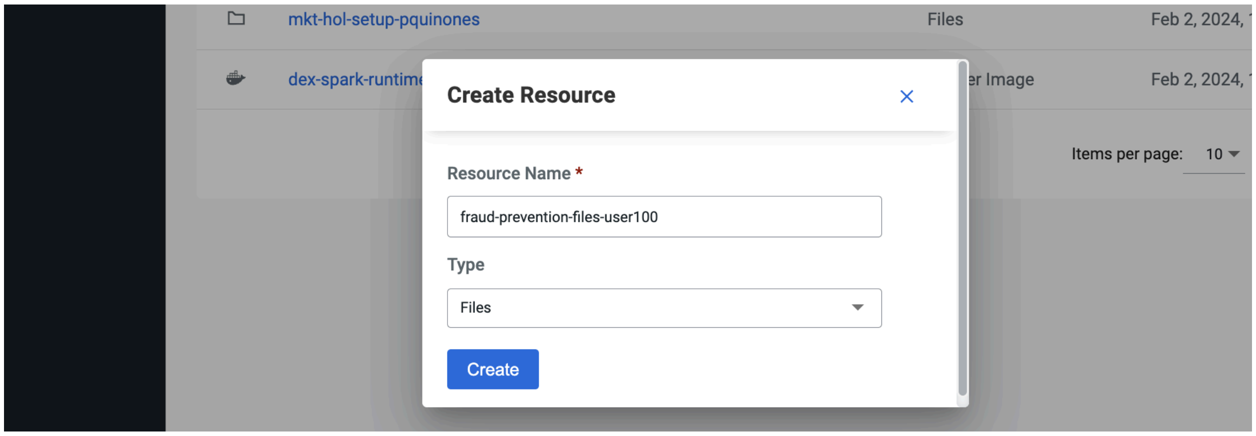
Status: **Building**

Building the resource...

Create CDE Files Resource

From the Resources page create a CDE Files Resource. Upload all files contained in the "*cde_spark_jobs*" folder. Again, ensure the Resource is named after your unique workshop username and it is created in the Virtual Cluster assigned to you.

e.g. "fraud-prevention-files-user100" if you are *"user100".*

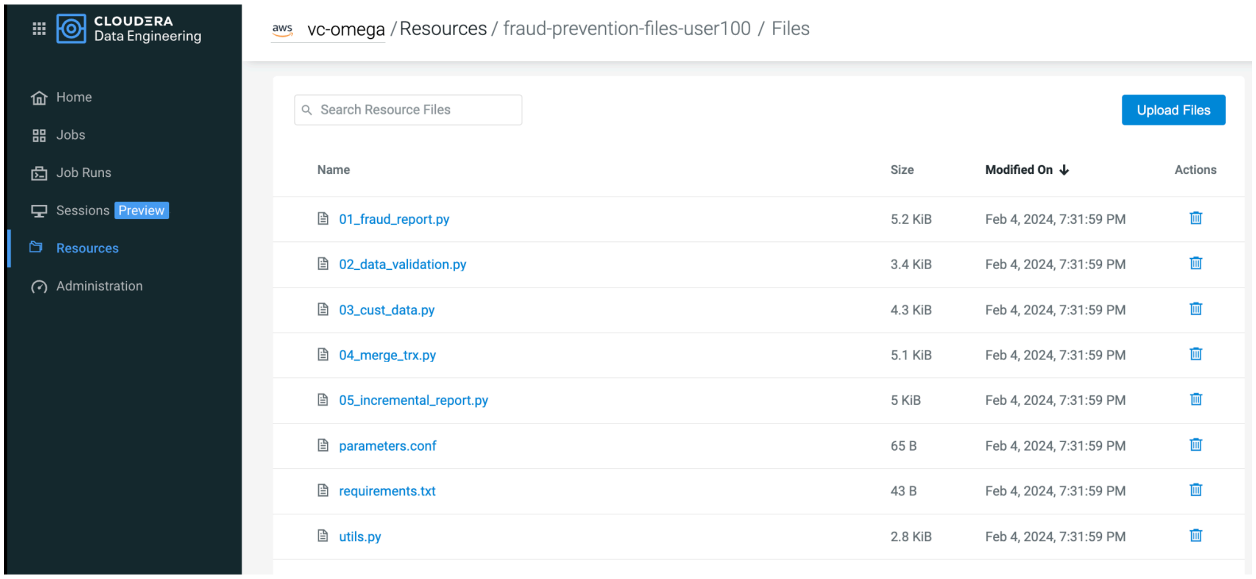


Resource Name *: fraud-prevention-files-user100

Type: Files

You should be able to see all the files (artifacts) uploaded in your file resource.

NOTE: Recommended to use the username in the resource name because you will reference these objects in the following labs.


CLOUDERA Data Engineering
Home
Jobs
Job Runs
Sessions Preview
Resources
Administration
aws vc-omega / Resources / fraud-prevention-files-user100 / Files
Search Resource Files
Upload Files

Name	Size	Modified On	Actions
01_fraud_report.py	5.2 KiB	Feb 4, 2024, 7:31:59 PM	
02_data_validation.py	3.4 KiB	Feb 4, 2024, 7:31:59 PM	
03_cust_data.py	4.3 KiB	Feb 4, 2024, 7:31:59 PM	
04_merge_trx.py	5.1 KiB	Feb 4, 2024, 7:31:59 PM	
05_incremental_report.py	5 KiB	Feb 4, 2024, 7:31:59 PM	
parameters.conf	65 B	Feb 4, 2024, 7:31:59 PM	
requirements.txt	43 B	Feb 4, 2024, 7:31:59 PM	
utils.py	2.8 KiB	Feb 4, 2024, 7:31:59 PM	

Before moving on to the next step, please familiarize yourself with the code in the "*01_fraud_report.py*", "*utils.py*", and "*parameters.conf*" files.

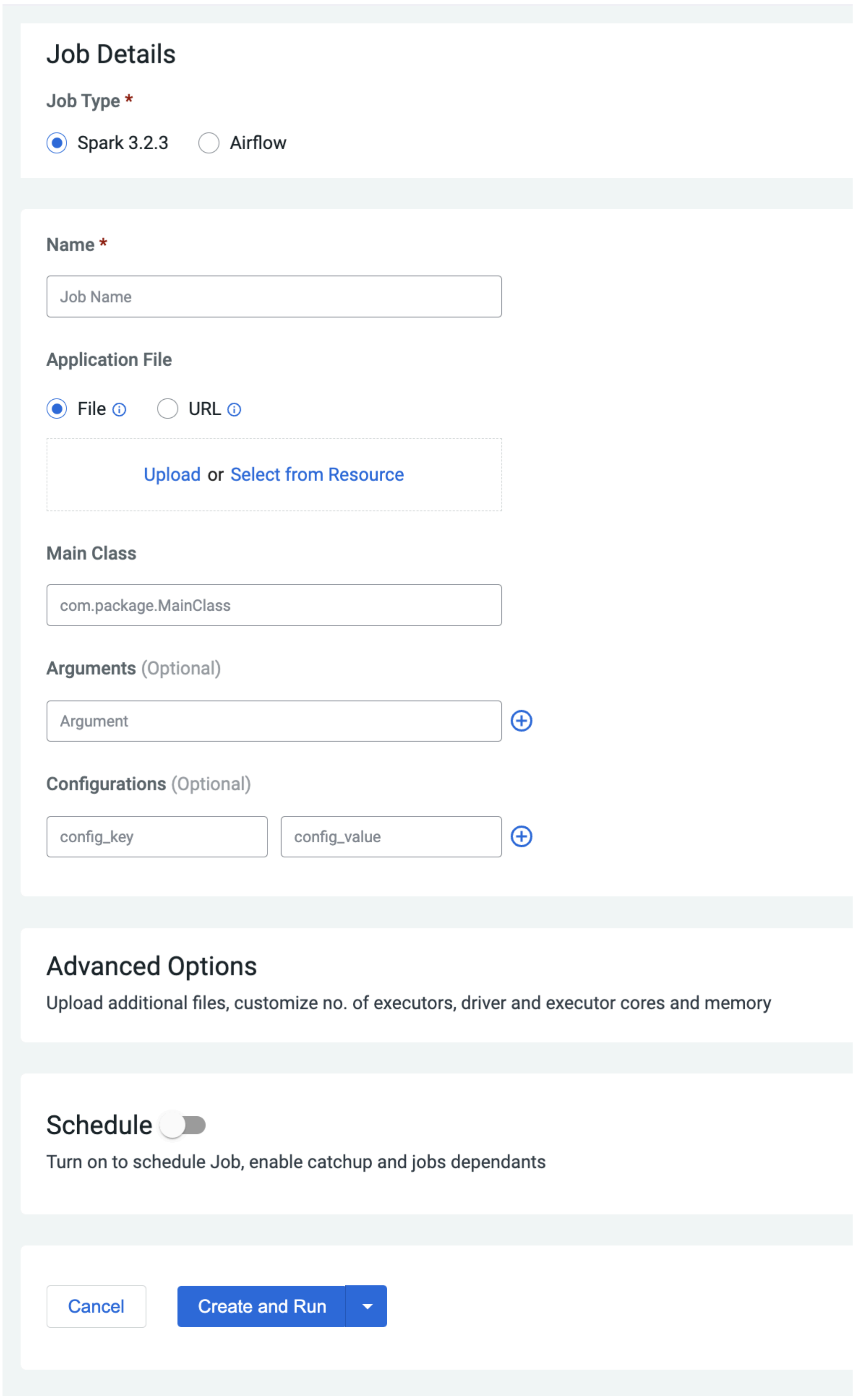
Notice that "*01_fraud_report.py*" contains the same PySpark Application code you ran in the CDE Session, with the exception that the column casting and renaming steps have been refactored into Python functions in the "*utils.py*" script.

Finally, notice the contents of "*parameters.conf*". Storing variables in a file in a Files Resource is one method used by CDE Data Engineers to dynamically parameterize scripts with external values.

Create CDE Spark Job

Now that the CDE Resources have been created you are ready to create your first CDE Spark Job.

Navigate to the CDE Jobs tab and click on "Create Job". The long form loaded to the page allows you to build a Spark Submit as a CDE Spark Job, step by step.



Job Details

Job Type *

[✓ Spark 3.2.3] Airflow

Name *

Job Name

Application File

[✓ File] URL

Upload or Select from Resource

Main Class

com.package.MainClass

Arguments (Optional)

Argument

Configurations (Optional)

config_key config_value

Advanced Options

Upload additional files, customize no. of executors, driver and executor cores and memory

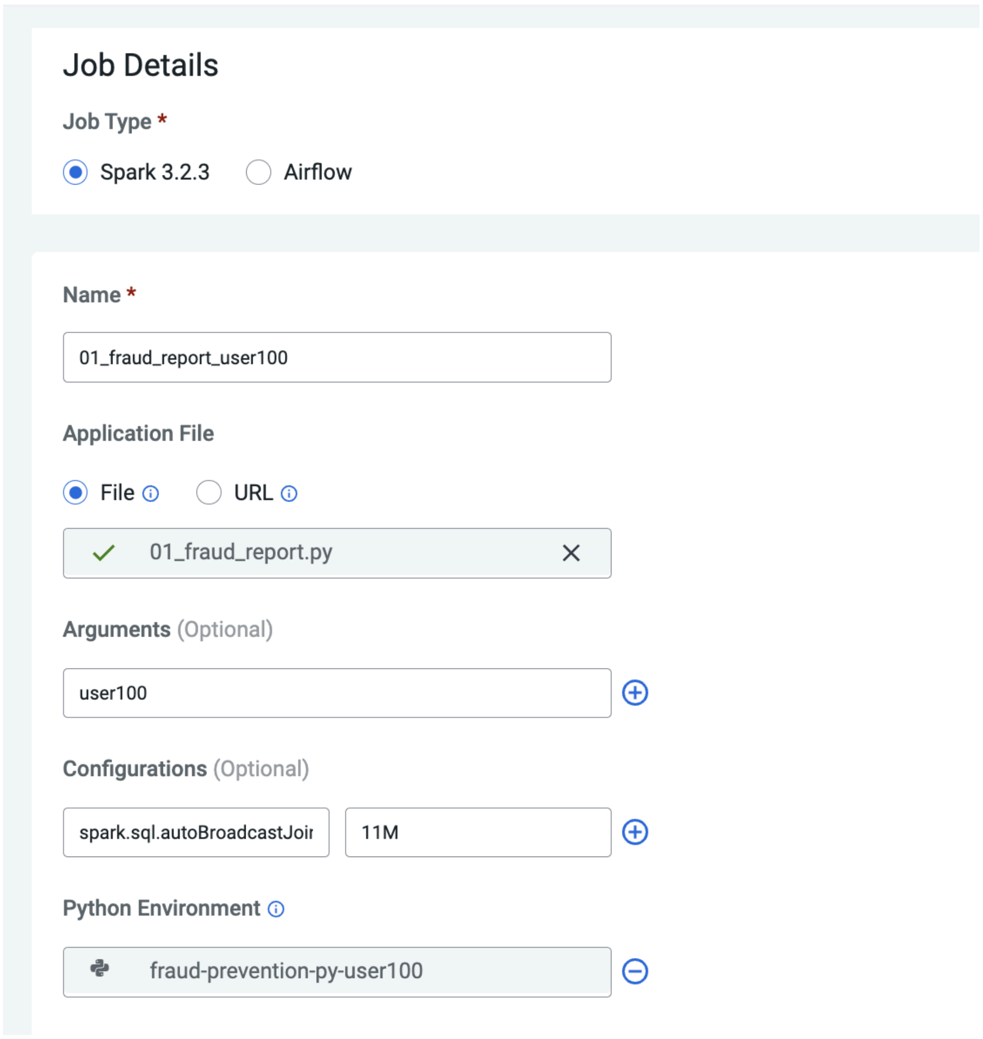
Schedule

Turn on to schedule Job, enable catchup and jobs dependants

Enter the following values without quotes into the corresponding fields. Make sure to update the username with your assigned user wherever needed:

- **Job Type**: Spark
- **Name:** 01_fraud_report_userxXx
- **File:** Select from Resource -> "01_fraud_report.py"
- **Arguments:** userxXx
- **Configurations:**
  - key: spark.sql.autoBroadcastJoinThreshold
  - value: 11M

The form should now look similar to this:


Job Details

Job Type *

[✓ Spark 3.2.3] Airflow

Name *

01_fraud_report_user100

Application File

[✓ File] URL

✓ 01_fraud_report.py

Arguments (Optional)

user100

Configurations (Optional)

spark.sql.autoBroadcastJoir	11M

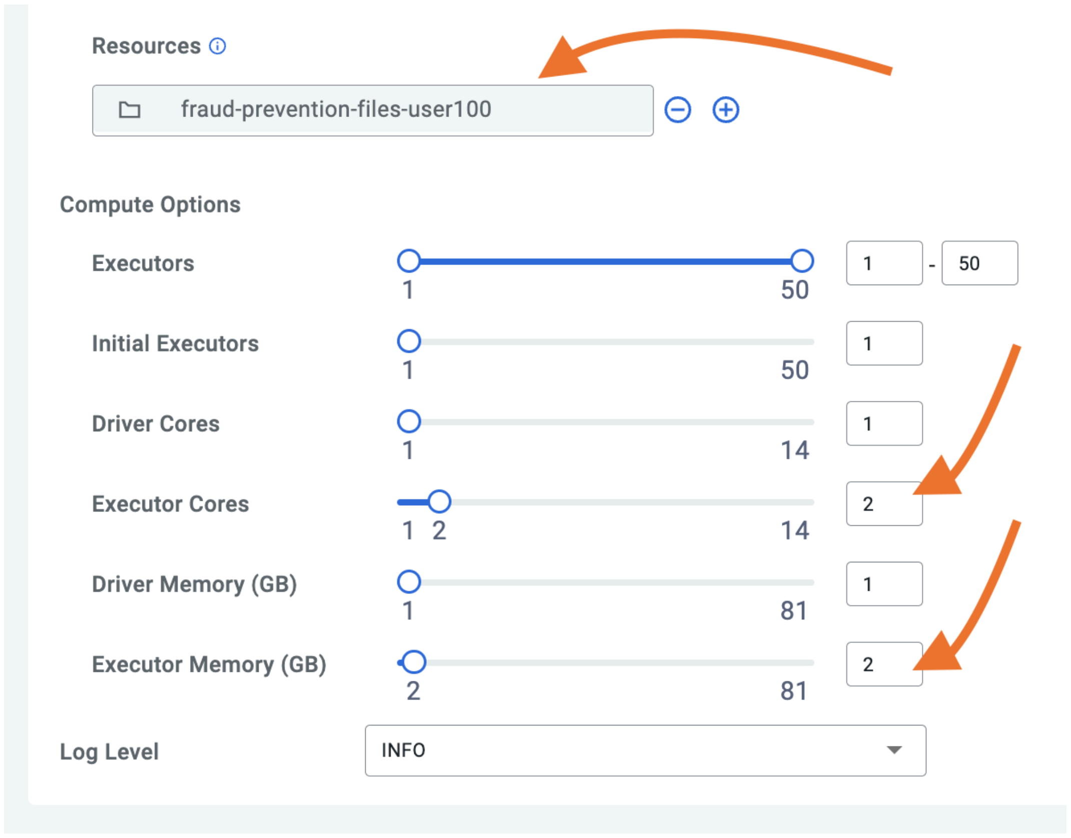
Python Environment

fraud-prevention-py-user100

Finally, open the "Advanced Options" section.

Notice that your CDE Files Resource has already been mapped to the CDE Job for you.

Then, update the Compute Options by increasing "Executor Cores" and "Executor Memory" from 1 to 2.



Resources

fraud-prevention-files-user100

Compute Options

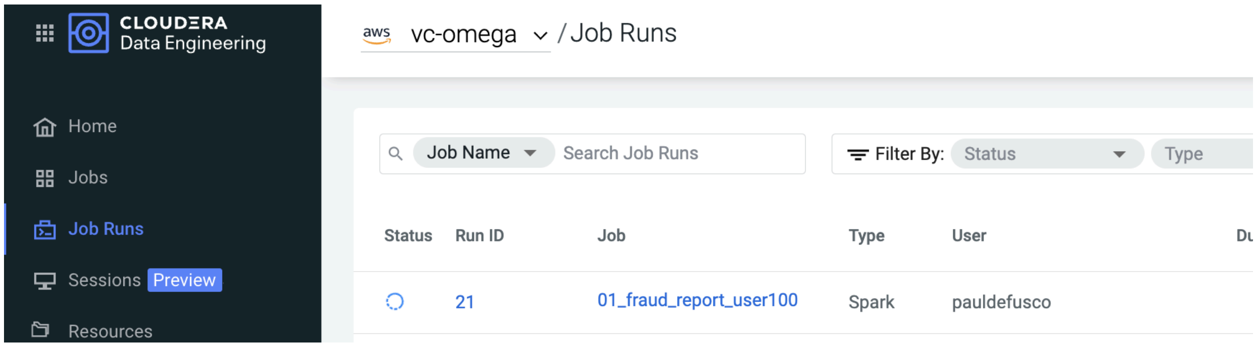
Executors	1 – 50	1 - 50
Initial Executors	1 – 50	1
Driver Cores	1 – 14	1
Executor Cores	1 2 – 14	2
Driver Memory (GB)	1 – 81	1
Executor Memory (GB)	2 – 81	2

Log Level: INFO

Finally, run the CDE Job by clicking the "**Create and Run"** icon.

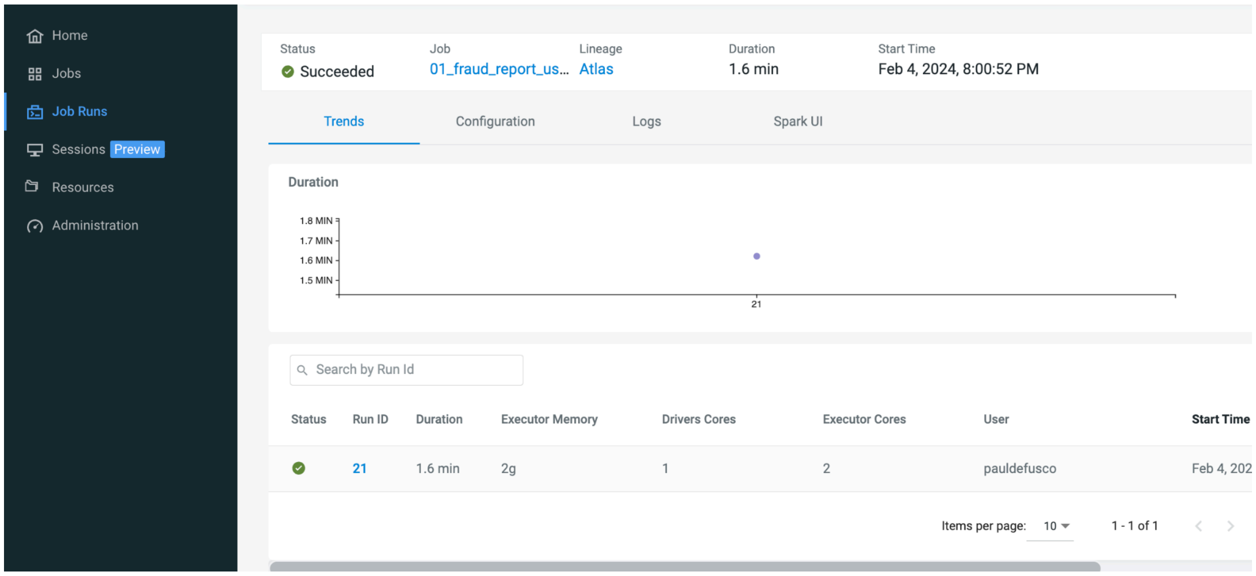
CDE Job Run Observability

Navigate to the Job Runs page in your Virtual Cluster and notice a new Job Run is being logged automatically for you.



Status	Run ID	Job	Type	User
	21	01_fraud_report_user100	Spark	pauldefusco

Once the run completes, open the run details by clicking on the Run ID integer in the Job Runs page.

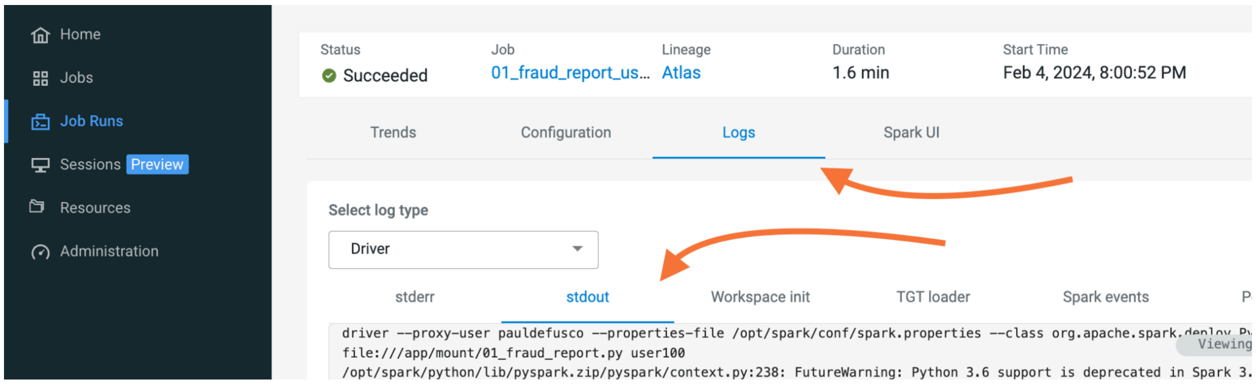


Status: Succeeded
Job: 01_fraud_report_us...
Lineage: Atlas
Duration: 1.6 min
Start Time: Feb 4, 2024, 8:00:52 PM

Status	Run ID	Duration	Executor Memory	Drivers Cores	Executor Cores	User	Start Time
✓	21	1.6 min	2g	1	2	pauldefusco	Feb 4, 202

Items per page: 10 1 - 1 of 1

Open the logs tab and validate output from the Job Run in the ***Driver -> Stdout*** tab.



Status: Succeeded
Job: 01_fraud_report_us...
Lineage: Atlas
Duration: 1.6 min
Start Time: Feb 4, 2024, 8:00:52 PM

```
driver --proxy-user pauldefusco --properties-file /opt/spark/conf/spark.properties --class org.apache.spark.deploy.Py
file:///app/mount/01_fraud_report.py user100
/opt/spark/python/lib/pyspark.zip/pyspark/context.py:238: FutureWarning: Python 3.6 support is deprecated in Spark 3.
```

Summary

In this section you first explored two datasets interactively with CDE Interactive sessions. This feature allowed you to run ad-hoc queries on large, structured and unstructured data, and prototype Spark Application code for batch execution.

Then, you created a batch Spark Job to turn your application prototype into scheduled execution.

In the process, you improved your code for reusability by modularizing your logic into functions, and stored those functions as a utils script in a CDE Files Resource. You also leveraged your Files Resource by storing dynamic variables in a parameters configurations file and applying a runtime variable via the Arguments field. In the context of more advanced Spark CI/CD pipelines both the parameters file and the Arguments field can be overwritten and overridden at runtime.

In order to improve performance you translated the PySpark UDF into a Pandas UDF. You created a CDE Python Resource and attached it to the CDE Job Definition in order to use Pandas and other Python libraries in your PySpark job.

Finally, you ran the job and observed outputs in the CDE Job Runs page. CDE stored Job Runs, logs, and associated CDE Resources for each run. This provided you real time job monitoring and troubleshooting capabilities, along with post-execution storage of logs, run dependencies, and cluster information.

Part 2: Open Data Lakehouse with Iceberg in CDE

A Brief Introduction to Apache Iceberg

Apache Iceberg is a cloud-native, high-performance open table format for organizing petabyte-scale analytic datasets on a file system or object store. Combined with Cloudera Data Platform (CDP), users can build an open data lakehouse architecture for multi-function analytics and to deploy large scale end-to-end pipelines.

Iceberg provides many advantages. First of all it is a table format that provides many advantages including the ability to standardize various data formats into a uniform data management system, a Lakehouse. In the Iceberg Lakehouse, data can be queries with different compute engines including Apache Impala, Apache Flink, Apache Spark and others.

In addition, Iceberg simplifies data analysis and data engineering pipelines with features such as partition and schema evolution, time travel, and Change Data Capture. In CDE you can use Spark to query Iceberg Tables interactively via Sessions or in batch pipelines via Jobs.

Lab 1: Working with Iceberg in CDE Sessions

In Part 1 we used CDE Sessions to explore two datasets and prototype code for a fraud detection Spark Application. In this brief Lab we will load a new batch of transactions from Cloud Storage and leverage Iceberg Lakehouse capabilities in order to test the SQL Merge Into with Spark.

NOTE: If you keep using the same session from the previous lab, you can skip the next step with the imports and variable definition.

```python
from pyspark.sql.types import Row, StructField, StructType, StringType,
IntegerType

storageLocation = "s3a://cde-innovation-buk-9e384927/data"
username = "user0Xx"
```

Migrate Spark Table to Iceberg Table

By default, a Spark table created in CDE is tracked in the Hive Metastore as an External table. Data is stored in Cloud Storage in one of many formats (parquet, csv, avro, etc.) and its location is tracked by the HMS.

When adopting Iceberg for the first time, tables can be copied or migrated to Iceberg format. A copy implies the reprocessing of the whole datasets into an Iceberg table while a Migration only involves the creation of Iceberg metadata in the Iceberg Metadata Layer.

CDE includes some functions to be able to perform the data lifecycle operations. One of the most important operations is the *in-place table migration* to Iceberg.

```python
## Migrate Table to Iceberg Table Format
spark.sql("ALTER TABLE {}.TRX_TABLE UNSET TBLPROPERTIES
('TRANSLATED_TO_EXTERNAL')".format(username))

spark.sql("CALL spark_catalog.system.migrate('{}.TRX_TABLE')".format(username))
```

The Iceberg Metadata Layer provides three layers of metadata files whose purpose is to track Iceberg tables as they are modified. The Metadata layer consists of Metadata Files, Manifest Lists and Manifest Files.

The Metadata Layer as a whole stores a version of each dataset by providing pointers to different versions of the data. Among other advantages, this enables Time Travel capabilities on Iceberg tables i.e. the ability to query data as of a particular snapshot or timestamp.

In the following example you will load a new batch of transactions from Cloud Storage. Then you will insert it into the historical transactions table via the SQL Merge Into statement which is not available in Spark SQL unless the Iceberg table format is used.

Finally, we will query Iceberg Metadata tables and query the data in its PRE-INSERT version with a simple Spark SQL command.

As in Part 1, you can just copy and paste the following code snippets into CDE Sessions Notebook cells.

```python
#PRE-INSERT TIMESTAMP
from datetime import datetime
now = datetime.now()
timestamp = datetime.timestamp(now)

print("PRE-INSERT TIMESTAMP: ", timestamp)
```

First, you will check the count of the records in the migrated table:

```python
# PRE-INSERT COUNT
spark.sql("SELECT COUNT(*) FROM
spark_catalog.{}.TRX_TABLE".format(username)).show()
```

Then, you will load a new data set and perform some transformations:

```python
# LOAD NEW TRANSACTION BATCH
trxBatchDf =
spark.read.json("{0}/mkthol/trans/{1}/trx_batch_1".format(storageLocatio
n, username))
trxBatchDf = trxBatchDf.withColumn("event_ts",
trxBatchDf["event_ts"].cast('timestamp'))
trxBatchDf.printSchema()
trxBatchDf.createOrReplaceTempView("BATCH_TEMP_VIEW".format(username))
```

Now, you can use the MERGE function to combine the two tables with the different conditions. By default the iceberg v2 tables will use Merge on Read mode.

```python

ICEBERG_MERGE_INTO_SYNTAX = """MERGE INTO spark_catalog.{}.TRX_TABLE t
USING (SELECT * FROM BATCH_TEMP_VIEW) s
   ON t.credit_card_number = s.credit_card_number WHEN MATCHED THEN
UPDATE SET * WHEN NOT MATCHED THEN INSERT *""".format(username)
```

```
spark.sql(ICEBERG_MERGE_INTO_SYNTAX)
```

NOTE: Iceberg allows you to also execute an APPEND only operation. To be able to do it, this is and example:

```python
### ALTERNATIVE SYNTAX VIA ICEBERG DF API IF MERGE INTO IS JUST APPEND
### trxBatchDf.writeTo("spark_catalog.{}.TRX_TABLE".format(username)).append()
```

After doing the merge operation, you can validate the actual number of records to confirm that some data was inserted:

```python
# POST-INSERT COUNT
spark.sql("SELECT COUNT(*) FROM
spark_catalog.{}.TRX_TABLE".format(username)).show()
```

Iceberg provides a set of functions to check the full history of the tables. To do that, you can use:

```python
# QUERY ICEBERG METADATA HISTORY TABLE
spark.sql("SELECT * FROM
spark_catalog.{}.TRX_TABLE.history".format(username)).show(20, False)
```

Also, there are some fields to check the full list of snapshots generated from the multiple transactions executed:

```python
# QUERY ICEBERG METADATA HISTORY TABLE
spark.sql("SELECT * FROM
spark_catalog.{}.TRX_TABLE.snapshots".format(username)).show(20, False)
```

To execute a time-travel operation, you can use the timestamp or the snapshot id. In this example, you will use the initial timestamp to query the initial version of the table:

```python
# TIME TRAVEL AS OF PREVIOUS TIMESTAMP
df = spark.read.option("as-of-timestamp",
int(timestamp*1000)).format("iceberg").load("spark_catalog.{}.TRX_TABLE".format
(username))

# POST TIME TRAVEL COUNT
print(df.count())
```

Summary

Open data lakehouse on CDP simplifies advanced analytics on all data with a unified platform for structured and unstructured data and integrated data services to enable any analytics use case from ML, BI to stream analytics and real-time analytics. Apache Iceberg is the secret sauce of the open lakehouse.

Apache Iceberg is an open table format designed for large analytic workloads. It supports schema evolution, hidden partitioning, partition layout evolution and time travel. Every table change creates an Iceberg snapshot, this helps to resolve concurrency issues and allows readers to scan a stable table state every time.

Iceberg lends itself well to a variety of use cases including Lakehouse Analytics, Data Engineering pipelines, and regulatory compliance with specific aspects of regulations such as GDPR (General Data Protection Regulation) and CCPA (California Consumer Privacy Act) that require being able to delete customer data upon request.

CDE Virtual Clusters provide native support for Iceberg. Users can run Spark workloads and interact with their Iceberg tables via SQL statements. The Iceberg Metadata Layer tracks Iceberg table versions via Snapshots and provides Metadata Tables with snapshot and other useful information. In this Lab we used Iceberg to access the credit card transactions dataset as of a particular timestamp.

Part 3: Pipeline Orchestration with Airflow in CDE

A Brief Introduction to Airflow

Apache Airflow is a platform to author, schedule and execute Data Engineering pipelines. It is widely used by the community to create dynamic and robust workflows for batch Data Engineering use cases.

The main characteristic of Airflow workflows is that all workflows are defined in Python code. The Python code defining the workflow is stored as a collection of Airflow Tasks organized in a DAG. Tasks are defined by built-in operators and Airflow modules. Operators are Python Classes that can be instantiated in order to perform predefined, parameterized actions.

CDE embeds Apache Airflow at the CDE Virtual Cluster level. It is automatically deployed for the CDE user during CDE Virtual Cluster creation and requires no maintenance on the part of the CDE Admin. In addition to the core Operators, CDE supports the CDEJobRunOperator and the CDWOperator in order to trigger Spark Jobs. and Data Warehousing queries.

Recommendations Before you Start

⚠ Warning
Throughout the labs, this guide will instruct you to make minor edits to some of the scripts. Please be prepared to make changes in an editor and re-upload them to the same CDE File Resource after each change. Having all scripts open at all times in an editor such as Visual Studio Code is highly recommended.

⚠ Warning
Each attendee will be assigned a username and cloud storage path. Each script will read your credentials from "parameters.conf" which you will have placed in your CDE File Resource. Before you start the labs, open the "parameters.conf" located in the "resources_files" folder.

⚠ Warning
If by mistake any of the jobs is run before the full airflow pipeline is created, you can go back to Lab 2 in Part 1, run the first job and then run the steps in Part 2 to do the in-place table conversion. WIth that you will have the tables required to execute Part 3.

Lab 1: Orchestrate Spark Pipeline with Airflow

In this lab you will build a pipeline of Spark Jobs to load a new batch of transactions, join it with customer PII data, and create a report of customers who are likely victims of credit card fraud.

At a high level, the workflow will be similar to Part 1 and 2 where you created two tables and loaded a new batch of transactions. However, there are two differences:

1. The workflow will leverage all the features used up to this point but in unison. For example, Iceberg Time Travel will be used to create an incremental report including only updates within the latest batch rather than the entire historical dataset.
2. The entire workflow will be orchestrated by Airflow. This will allow you to run your jobs in parallel while implementing robust error handling logic.

Create Spark Jobs

In this section you will create four CDE Spark Jobs via the CDE Jobs UI. It is important that you do not run the Spark Jobs when you create them. If you do run them by mistake, please raise your hand during the workshop and ask for someone to help you implement a workaround.

1. **Data Validation Job:**

Name: <name this after your user> *e.g. if you are user "user010" call it "02_data_val_user010"*

Application File: "02_data_validation.py" located in your CDE Files resource.

Arguments: enter your username here, without quotes (just text) e.g. if you are user "user010" enter "user010" without quotes

Python Environment: choose your CDE Python resource from the dropdown

Files & Resources: choose your CDE Files resource from the dropdown (this should have already been prefilled for you)

Leave all other settings to default values.

2. **Customer Data Load Job:**

Name: <name this after your user> *e.g. if you are user "user010" call it "03_cust_data_user010"*

Application File: "03_cust_data.py" located in your CDE Files resource.

Arguments: enter your username here, without quotes (just text) e.g. if you are user "user010" enter "user010" without quotes

Python Environment: choose your CDE Python resource from the dropdown

Files & Resources: choose your CDE Files resource from the dropdown (this should have already been prefilled for you)

Leave all other settings to default values.

3. **Merge Transactions Job:**

Name: <name this after your user> *e.g. if you are user "user010" call it "04_merge_trx_user010"*

Application File: "04_merge_trx.py" located in your CDE Files resource.

Arguments: enter your username here, without quotes (just text) e.g. if you are user "user010" enter "user010" without quotes

Files & Resources: choose your CDE Files resource from the dropdown (this should have already been prefilled for you)

Leave all other settings to default values.

4. **Incremental Report Job:**

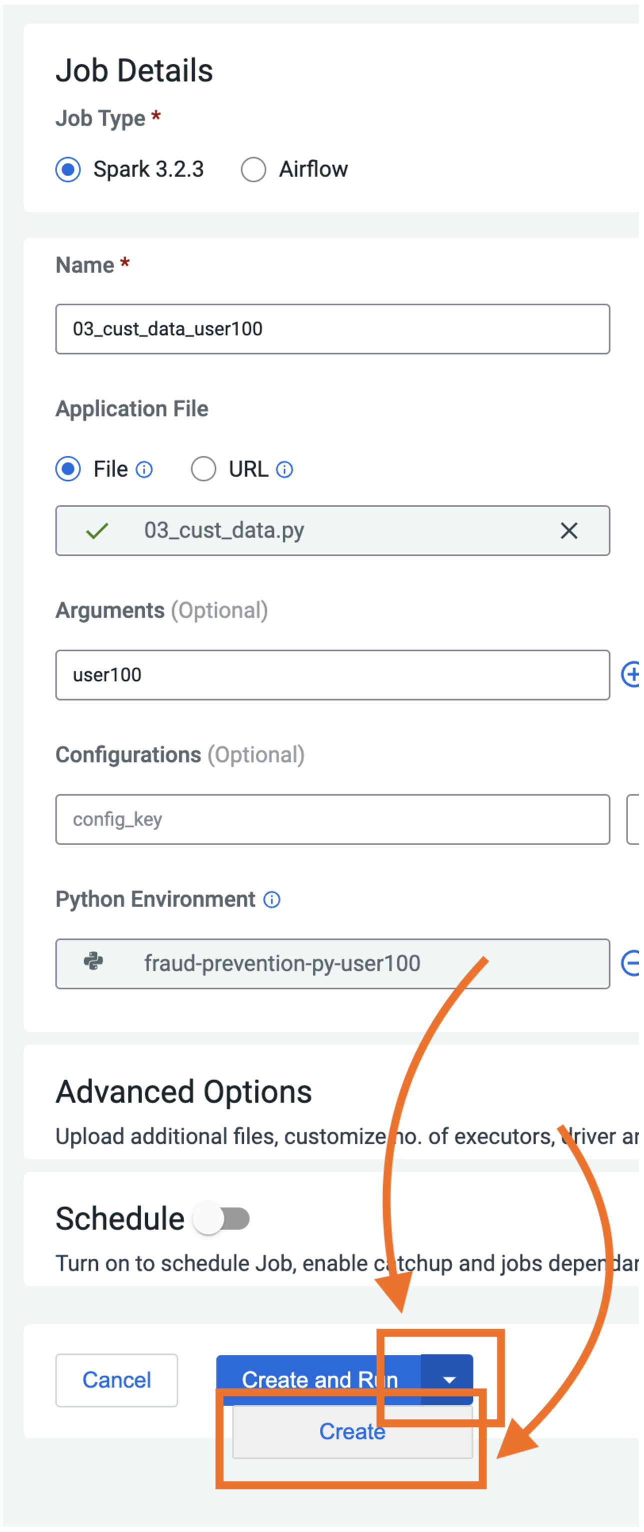
Name: <name this after your user> *e.g. if you are user "user010" call it "05_inc_report_user010"*

Application File: "05_incremental_report.py" located in your CDE Files resource.

Arguments: enter your username here, without quotes (just text) e.g. if you are user "user010" enter "user010" without quotes

Files & Resources: choose your CDE Files resource from the dropdown (this should have already been prefilled for you)

Leave all other settings to default values.



Job Details

Job Type *

[✓ Spark 3.2.3] Airflow

Name *

03_cust_data_user100

Application File

[✓ File] URL

03_cust_data.py

Arguments (Optional)

user100

Configurations (Optional)

config_key

Python Environment

fraud-prevention-py-user100

Advanced Options

Upload additional files, customize no. of executors, driver ar

Schedule

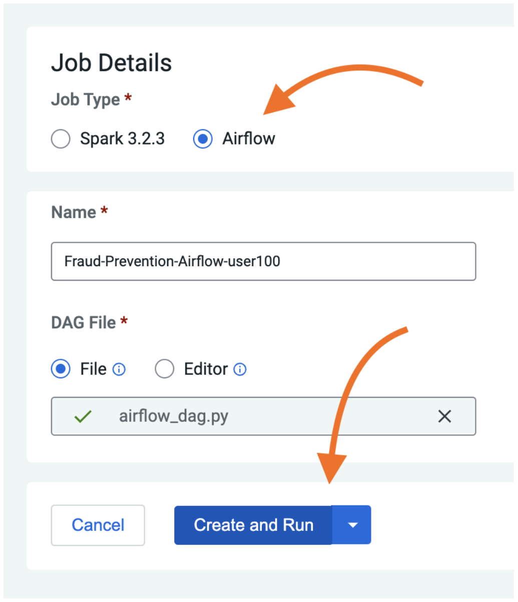
Turn on to schedule Job, enable catchup and jobs dependar

Create Airflow Job

Open the *"airflow_dag.py"* script located in the "cde_airflow_jobs" folder. Familiarize yourself with the code an notice:

- The Python classes needed for the DAG Operators are imported at the top. Notice the CDEJobRunOperator is included to run Spark Jobs in CDE.
- The "default_args" dictionary includes options for scheduling, setting dependencies, and general execution.
- Four instances of the CDEJobRunOperator object is declared with the following arguments:
    - **Task ID**: This is the name used by the Airflow UI to recognize the node in the DAG.
    - **DAG**: This has to be the name of the DAG object instance declared at line 16.
    - **Job Name**: *This has to be the name of the Spark CDE Job created in step 1 above.*
- Finally, at the bottom of the DAG, Task Dependencies are declared. With this statement you can specify the execution sequence of DAG tasks.

Edit the username variable at line 49. Then navigate to the CDE Jobs UI and create a new CDE Job.



Job Details

Job Type *

Spark 3.2.3 [✓ Airflow]

Name *

Fraud-Prevention-Airflow-user100

DAG File *

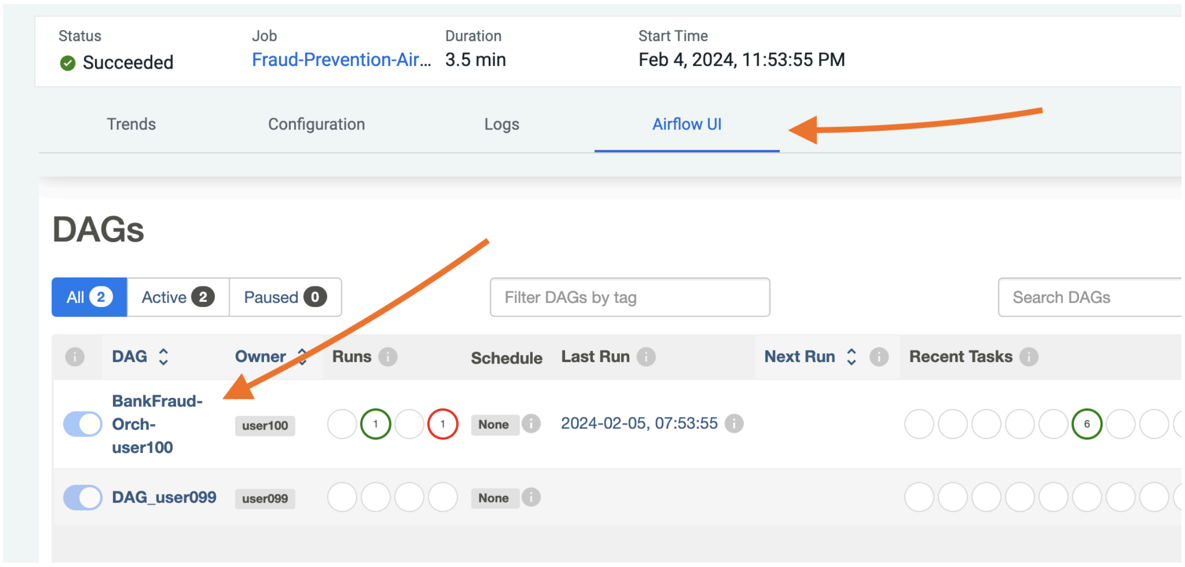
[✓ File] Editor

airflow_dag.py

NOTE: First you need to modify these files locally and then you can upload the modified version into the File Resource created in Part 2 to be able to be used by the Airflow jobs.

Select Airflow as the Job Type, assign a unique CDE Job name based on your user, and then run the Job

If you open the **Job Run.** Go to the **Airflow UI Tab.** You can see the status of the pipeline and the steps.



Status: Succeeded | Job: Fraud-Prevention-Air... | Duration: 3.5 min | Start Time: Feb 4, 2024, 11:53:55 PM

Trends | Configuration | Logs | Airflow UI

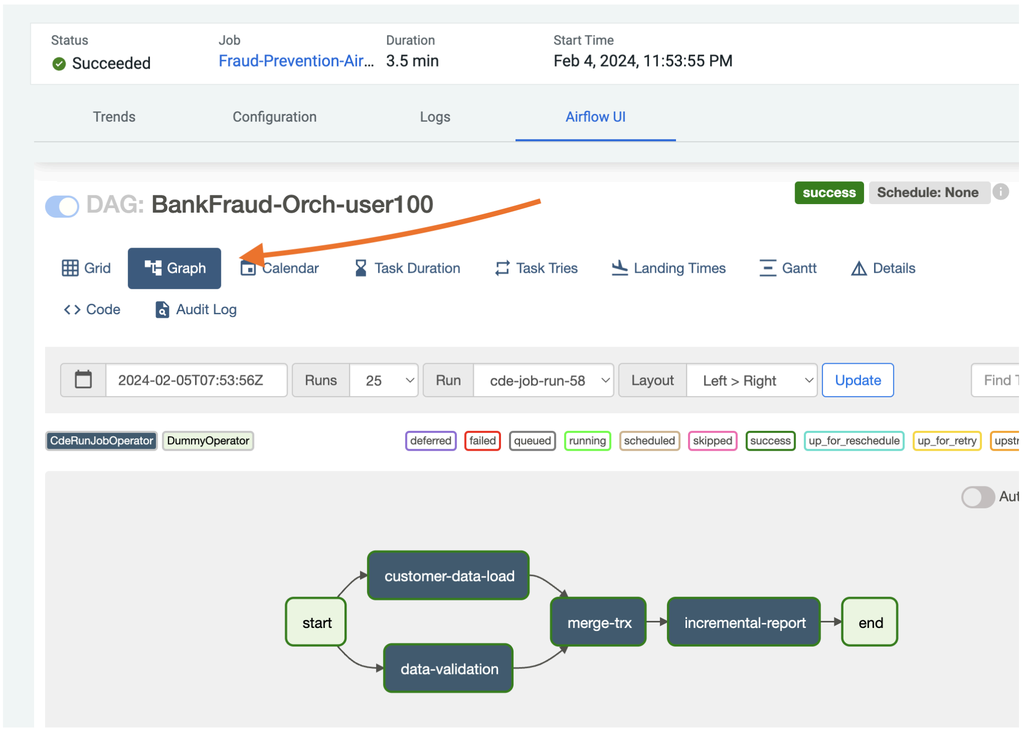
DAGs

All 2 | Active 2 | Paused 0

DAG	Owner	Runs	Schedule	Last Run	Next Run	Recent Tasks
BankFraud-Orch-user100	user100	1 1	None	2024-02-05, 07:53:55		6
DAG_user099	user099		None			

Monitor the execution of the pipeline from the Job Runs UI. Notice an Airflow Job will be triggered and successively the four CDE Spark Jobs will run one by one.

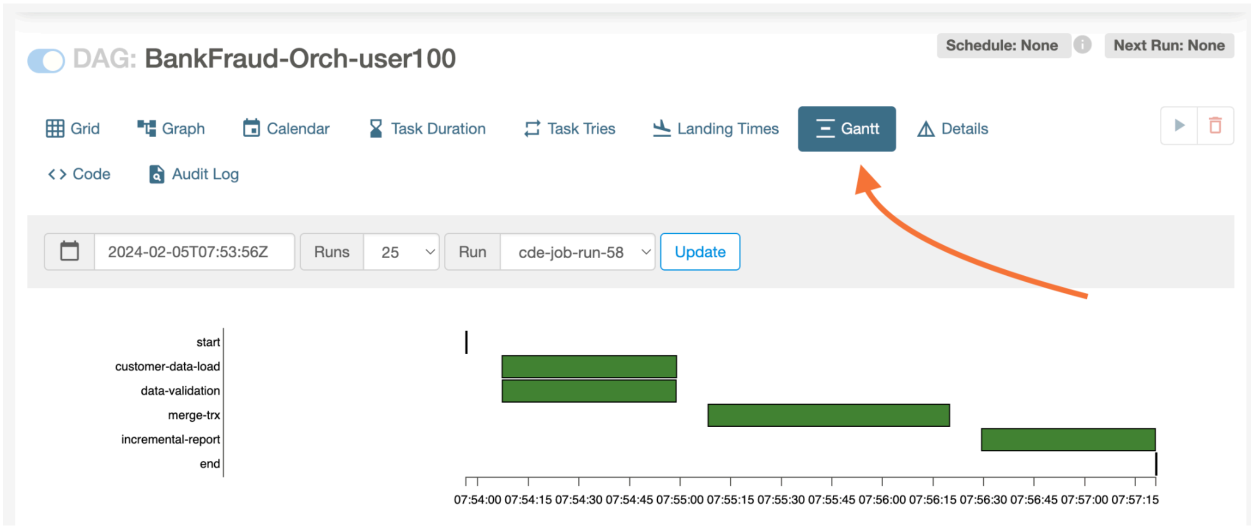
While the job is in-flight, open the Airflow UI and monitor execution.



Status: Succeeded | Job: Fraud-Prevention-Air... | Duration: 3.5 min | Start Time: Feb 4, 2024, 11:53:55 PM

DAG: BankFraud-Orch-user100 — success — Schedule: None

2024-02-05T07:53:56Z | Runs 25 | Run cde-job-run-58 | Layout Left > Right



Summary

Each CDE virtual cluster includes an embedded instance of Apache Airflow. With Airflow based pipelines users can specify their Spark pipeline using a simple python configuration file called the Airflow DAG.

A basic CDE Airflow DAG can be composed of a mix of hive and spark operators that automatically run jobs on CDP Data Warehouse (CDW) and CDE, respectively; with the underlying security and governance provided by SDX.

Conclusion

Congratulations for making it to the end of this tutorial! We hope you enjoyed using CDE first hand. We recommend visiting the Next Steps Section to continue your journey with CDE.

THANK YOU

CLOUDERA