Distributed Data Systems

MAHESH CHAUDHARI, PH.D

Contents

Course Overview

Workflow Management

Apache Airflow

Airflow DAG Creation

Contents

Course Overview

Workflow Management

Apache Airflow

Airflow DAG Creation

Airflow

Apache Airflow

An open-source scalable workflow manager for developing and monitoring batch-oriented workflows (developed by Airbnb).

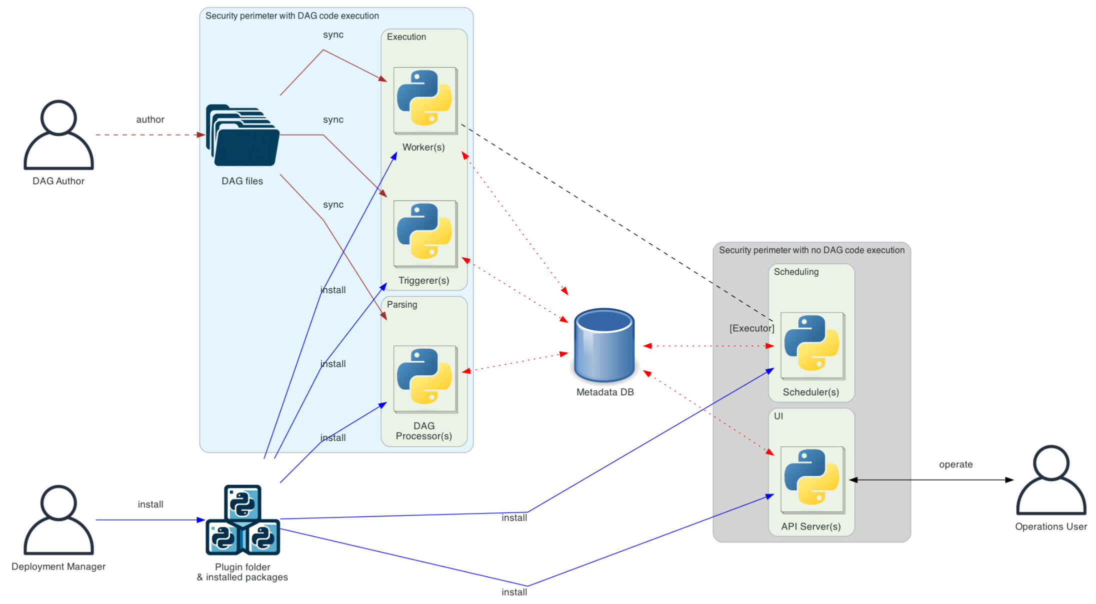
- You can define DAGs using Python.
  - Should include a set of tasks and their dependencies.
  - Flexible - you can dynamically generate optional tasks based on conditions.
- Main Components
  - Scheduler - monitors all tasks and DAGs, triggers the task instances once their dependencies are complete.
  - Workers - Executing tasks.
  - api-server - Visualize DAGs and provide main interface to monitor DAGs.
  - dag-processor - dedicated component responsible for continuously parsing Python files that define workflows (DAGs) and storing a serialized version of them in the metadata database

https://airflow.apache.org

Airflow

Apache Airflow

https://airflow.apache.org



1. User writes DAG
2. dag-processor detects the new DAG, parses it and serializes it into the airflow database.
3. Scheduler parses DAG and schedules tasks, considering schedules and dependencies.
4. Worker executes scheduled tasks.
5. User monitors workflow execution schedules and results displayed by the api-server.

* Note : Results/logs from each step is stored in metastores to track schedules and progress.

Contents

Course Overview

Workflow Management

Apache Airflow

Airflow DAG Creation

Airflow DAG definition

DAG

- Includes tasks and their dependencies.
- Declaration

```
with DAG(dag_id="dag_name",
         start_date=datetime(2022, 1, 1),
         end_date=datetime(2023, 12, 31),
         schedule="@daily") as dag:

     task1 = PythonOperator(task_id="task1",
                            python_callable=func_1,
                            op_kwargs={'arg1':val1, 'arg2':val2}
                            )

     task2 = BashOperator(task_id="task2",
                          bash_command="enter bash command")

     task1 >> task2
```

Str, ID of DAG

Datetime, timestamp from which the scheduler will attempt to run/backfill

Datetime, timestamp beyond which your DAG won't run, leave to None for open-ended.

Scheduling rules. You can use `@once`,`@hourly`, `@daily`, `@weekly`, `@monthly`, `@yearly` and also crontab-like scheduling

https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html

https://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/models/dag/index.html#airflow.models.dag.DAG https://ostechnix.com/a-beginners-guide-to-cron-jobs/

https://crontab.guru/

Crontab generator - https://crontab-generator.org/

Airflow DAG definition

Crontab
- Format
  - 1st : Minute (0-59)
  - 2nd: Hour (0-23)
  - 3rd : Day (1-31)
  - 4th : Month(1-12)
  - 5th : Day of week (0-7. 0/7-Sun. 1-Mon, 2-Tue..)
- Value : * means any value.

```
* * * * *
- - - - -
| | | | |
| | | | ----- Day of week (0 - 7) (Sunday=0 or 7)
| | | ------- Month (1 - 12)
| | --------- Day of month (1 - 31)
| ----------- Hour (0 - 23)
------------- Minute (0 - 59)
```

Airflow DAG definition

DAG

- Includes tasks and their dependencies.
- Declaration

Executes a Python callable

Id of task

Python Function Name

a dictionary of keyword arguments used in func_1

```
with DAG(dag_id="dag_name",
         start_date=datetime(2022, 1, 1),
         end_date=datetime(2023, 12, 31),
         schedule="@daily") as dag:

     task1 = PythonOperator(task_id="task1",
                            python_callable=func_1,
                            op_kwargs={'arg1':val1, 'arg2':val2}
                            )

     task2 = BashOperator(task_id="task2",
                          bash_command="enter bash command")

     task1 >> task2
```

https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html

https://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/models/dag/index.html#airflow.models.dag.DAG

Airflow DAG definition

DAG

- Includes tasks and their dependencies.
- Declaration

```
with DAG(dag_id="dag_name",
          start_date=datetime(2022, 1, 1),
          end_date=datetime(2023, 12, 31),
          schedule="@daily") as dag:

     task1 = PythonOperator(task_id="task1",
                            python_callable=func_1,
                            op_kwargs={'arg1':val1, 'arg2':val2}
                            )

     task2 = BashOperator(task_id="task2",
                          bash_command="enter bash command")

     task1 >> task2
```

Executes a Bash script

Id of task

Bash commands

Note : Other operator types in Airflow (https://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/operators/index.html)
https://airflow.apache.org/docs/apache-airflow-providers-standard/stable/operators/python.html
https://airflow.apache.org/docs/apache-airflow/stable/howto/operator/bash.html

Airflow DAG definition

DAG

- Includes tasks and their dependencies.
- Declaration

```
with DAG(dag_id="dag_name",
         start_date=datetime(2022, 1, 1),
         end_date=datetime(2023, 12, 31),
         schedule="@daily") as dag:

     task1 = PythonOperator(task_id="task1",
                            python_callable=func_1,
                            op_kwargs={'arg1':val1, 'arg2':val2}
                            )

     task2 = BashOperator(task_id="task2",
                          bash_command="enter bash command")

     task1 >> task2
```

Task dependency - The direction of edges between tasks.

```
first_task >> [second_task, third_task]
third_task << fourth_task
```

https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html

Task dependencies

Method 1:

- Use << and >> operators to indicate the task directionality and dependency.

```
task1 >> task2
```

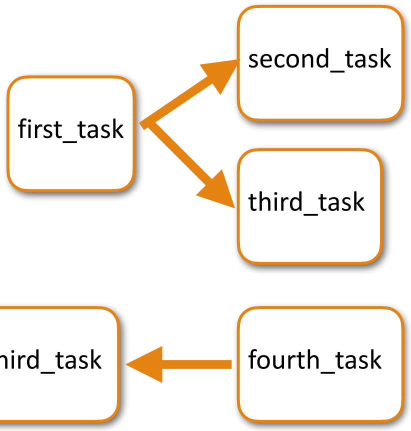
Task dependency - The direction of edges between tasks.

```
first_task >> [second_task, third_task]
third_task << fourth_task
```

Method 2:

- Use set_upstream and set_downstream methods:

```
task1.set_downstream(task2)
first_task.set_downstream([second_task, third_task])
third_task.set_upstream(fourth_task)
```



Task dependencies - Contd

More Complex chaining:

- Use cross_downstream

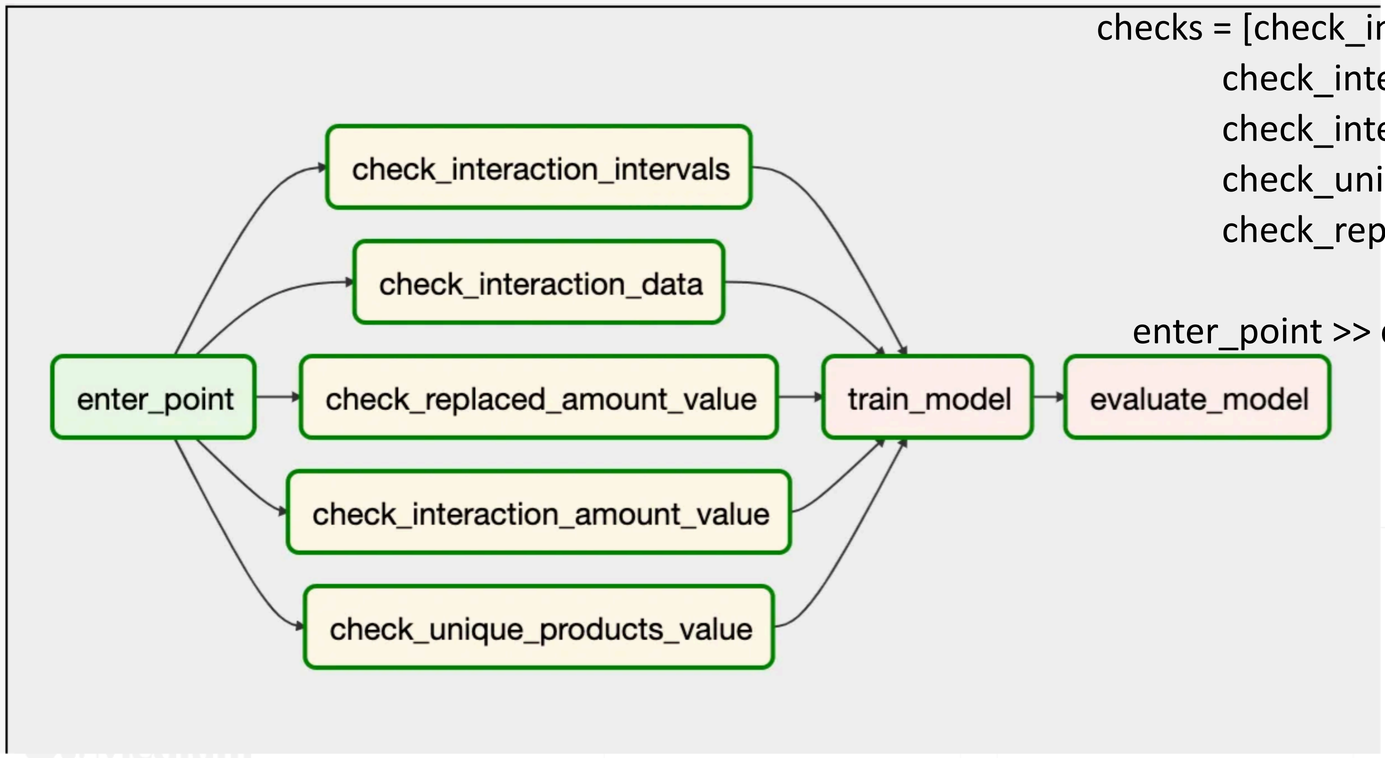
  ```
  # [op1, op2] >> op3
  # [op1, op2] >> op4
  cross_downstream([op1, op2], [op3, op4])
  ```

- Chain tasks together :

  ```
  # Replaces op1 >> op2 >> op3 >> op4
  chain(op1, op2, op3, op4)
  ```

https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html

More complex task dependencies



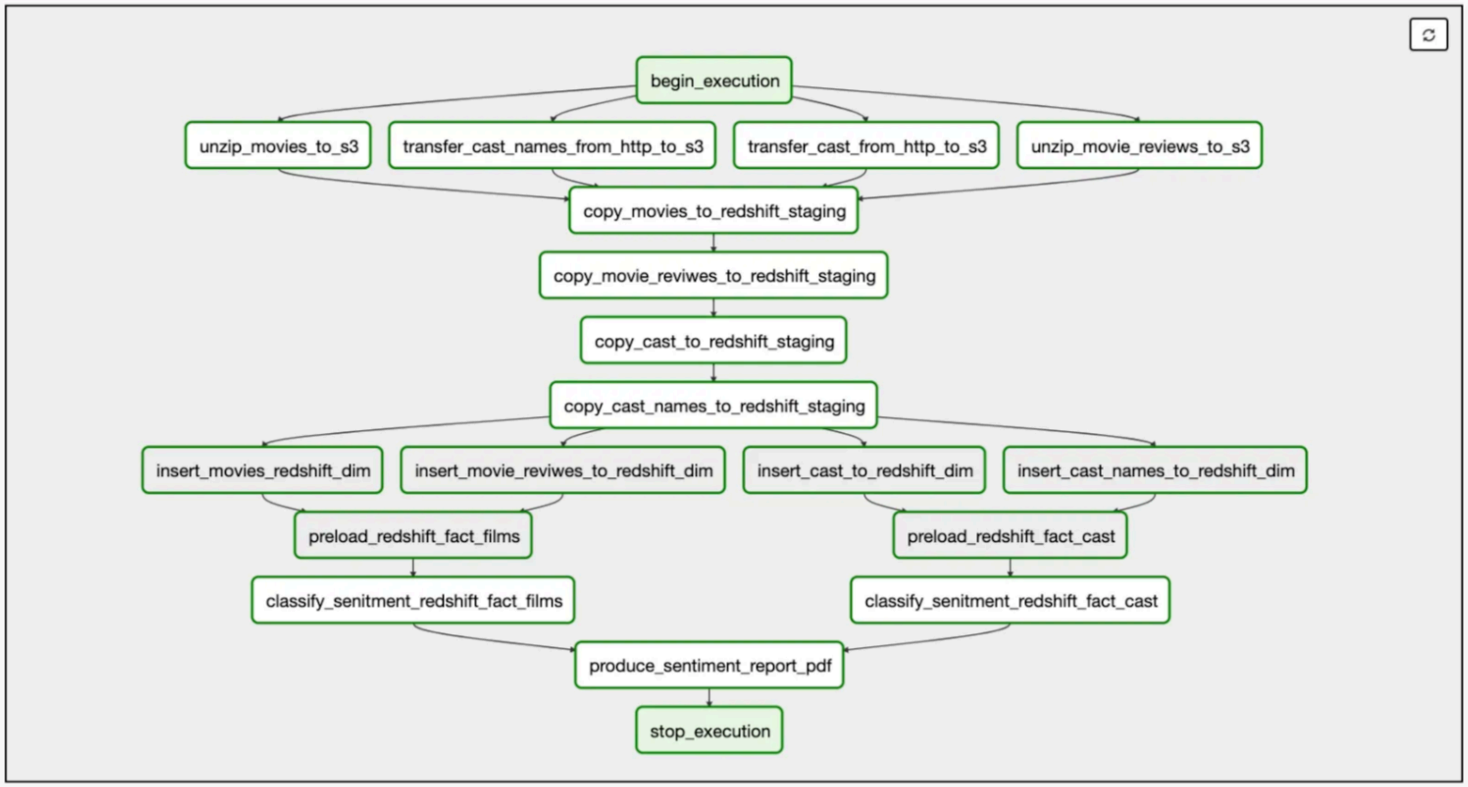
enter_point
check_interaction_intervals
check_interaction_data
check_replaced_amount_value
check_interaction_amount_value
check_unique_products_value
train_model
evaluate_model

```
checks = [check_interaction_data,
          check_interaction_intervals,
          check_interaction_amount_value,
          check_unique_products_value,
          check_replaced_amount_value]

enter_point >> checks >> train_model >> evaluate_model
```

https://towardsdatascience.com/machine-learning-in-production-using-apache-airflow-91d25a4d8152

Last example- task dependencies


begin_execution
unzip_movies_to_s3
transfer_cast_names_from_http_to_s3
transfer_cast_from_http_to_s3
unzip_movie_reviews_to_s3
copy_movies_to_redshift_staging
copy_movie_reviwes_to_redshift_staging
copy_cast_to_redshift_staging
copy_cast_names_to_redshift_staging
insert_movies_redshift_dim
insert_movie_reviwes_to_redshift_dim
insert_cast_to_redshift_dim
insert_cast_names_to_redshift_dim
preload_redshift_fact_films
preload_redshift_fact_cast
classify_senitment_redshift_fact_films
classify_senitment_redshift_fact_cast
produce_sentiment_report_pdf
stop_execution

https://towardsdatascience.com/machine-learning-in-production-using-apache-airflow-91d25a4d8152

Trigger Rules

Default behavior: Airflow will wait for all the upstream tasks to be successful before running the next task.

```
task1 >> task2
cross_downstream([op1, op2], [op3, op4])
chain(op1, op2, op3, op4)
```

Additional trigger rules for more control. But be careful of the behavior and unintentional side effects.

Customizable Trigger Rules

- `all_success` (default): All upstream tasks have succeeded
- `all_failed`: All upstream tasks are in a `failed` or `upstream_failed` state
- `all_done`: All upstream tasks are done with their execution
- `all_skipped`: All upstream tasks are in a `skipped` state
- `one_failed`: At least one upstream task has failed (does not wait for all upstream tasks to be done)
- `one_success`: At least one upstream task has succeeded (does not wait for all upstream tasks to be done)
- `one_done`: At least one upstream task succeeded or failed
- `none_failed`: All upstream tasks have not `failed` or `upstream_failed` - that is, all upstream tasks have succeeded or been skipped
- `none_failed_min_one_success`: All upstream tasks have not `failed` or `upstream_failed`, and at least one upstream task has succeeded.
- `none_skipped`: No upstream task is in a `skipped` state - that is, all upstream tasks are in a `success`, `failed`, or `upstream_failed` state
- `always`: No dependencies at all, run this task at any time

https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html

Operators

- `BashOperator - executes a bash command`
- `PythonOperator - calls an arbitrary Python function`
- `EmailOperator - sends an email`
- `HttpOperator`
- `MySqlOperator`
- `PostgresOperator`
- `MsSqlOperator`
- `OracleOperator`
- `JdbcOperator`

https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html

Airflow Operators Resources

https://airflow.apache.org/docs/

https://www.mongodb.com/developer/products/mongodb/mongodb-apache-airflow/

https://docs.astronomer.io/learn/airflow-mongodb

https://airflow.apache.org/docs/apache-airflow-providers-amazon/stable/operators/index.html

https://airflow.apache.org/docs/apache-airflow-providers-google/stable/operators/cloud/index.html

Airflow Installation

- Use Conda environment with python=3.12 Airflow with latest Python has some issues.
- Create conda environment with python version 3.12

```
conda create --name dds-spring-2026 python=3.12
```

- Activate the environment

```
conda activate dds-spring-2026
```

- Install Airflow on your local machine:

```
conda install conda-forge::airflow
```

- Initiate/Migrate the existing SQLLite database

```
airflow db migrate
```

Airflow Execution

Only for the first time

1. Initialize the metastore

```
$ airflow db migrate
```

2. Create a dags directory under ~/airflow

```
$ mkdir ~/airflow/dags
```

This is where your .py files should located.

3. Create a user

```
$ airflow users create --username admin --firstname FIRST_NAME --lastname LAST_NAME --role Admin --
email admin@example.org
```

Airflow requires users to authenticated to login to the web server.

3. Copy the DAG into the DAGs directory

```
$ cp *.py ~/airflow/dags/
```

4. Start the scheduler and web server (Open three separate terminals)

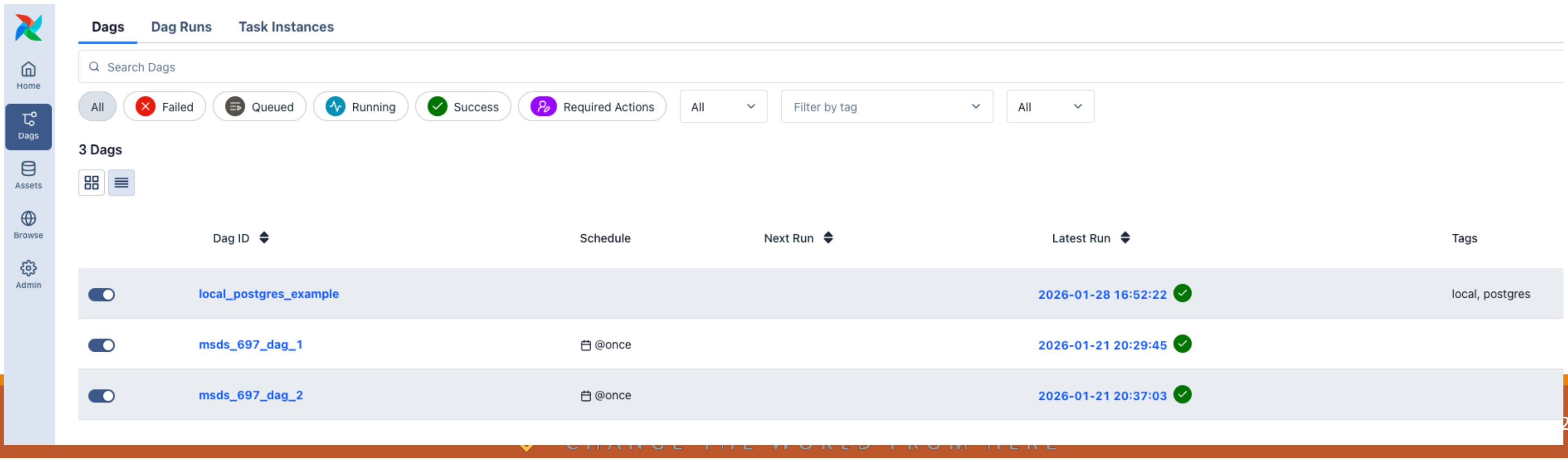
```
$ airflow scheduler
$ airflow api-server
$ airflow dag-processor
```

https://airflow.apache.org/docs/apache-airflow/stable/installation/setting-up-the-database.html

Airflow api-server

You can access the Airflow api-server via 0.0.0.0:8080 when runs locally.

- : Turn on/off the DAG
- Click the dag_id to see more details
- : Execute the DAG manually

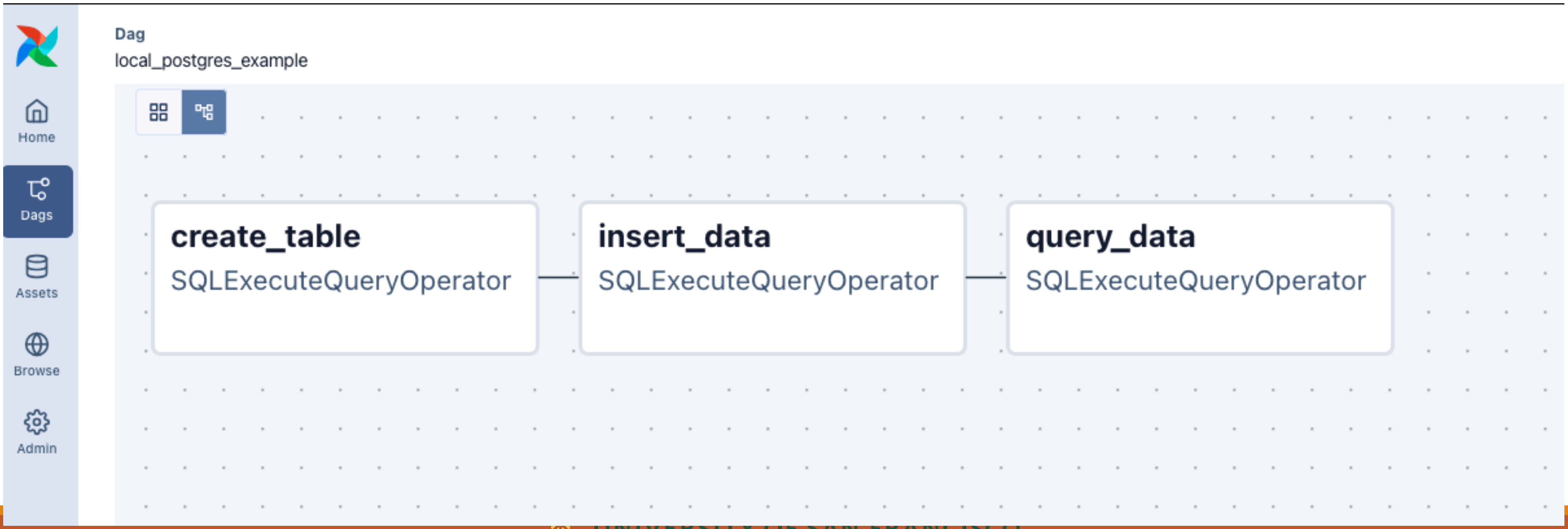


	Dag ID	Schedule	Next Run	Latest Run	Tags
	local_postgres_example			2026-01-28 16:52:22 ✓	local, postgres
	msds_697_dag_1	@once		2026-01-21 20:29:45 ✓	
	msds_697_dag_2	@once		2026-01-21 20:37:03 ✓	

Airflow api-server

DAG visualization

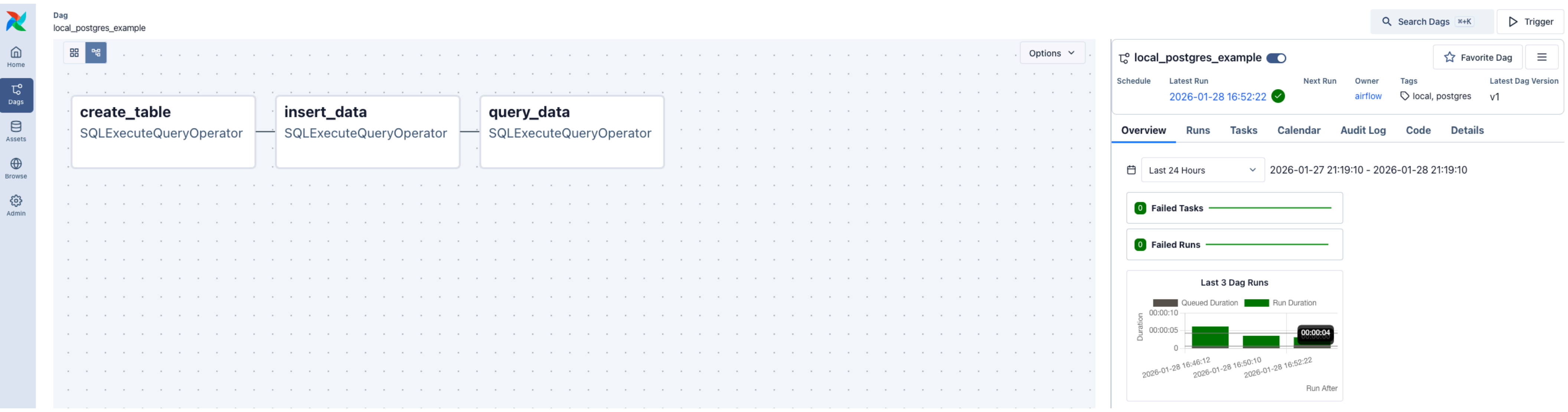
- dag_id ➡ Graph shows a dependency graph of tasks.



Airflow api-server

DAG visualization

- dag_id ➡ Graph view with stats.



Dag
local_postgres_example

create_table
SQLExecuteQueryOperator

insert_data
SQLExecuteQueryOperator

query_data
SQLExecuteQueryOperator

local_postgres_example

Schedule	Latest Run	Next Run	Owner	Tags	Latest Dag Version
	2026-01-28 16:52:22 ✓		airflow	local, postgres	v1

Last 24 Hours 2026-01-27 21:19:10 - 2026-01-28 21:19:10

0 Failed Tasks

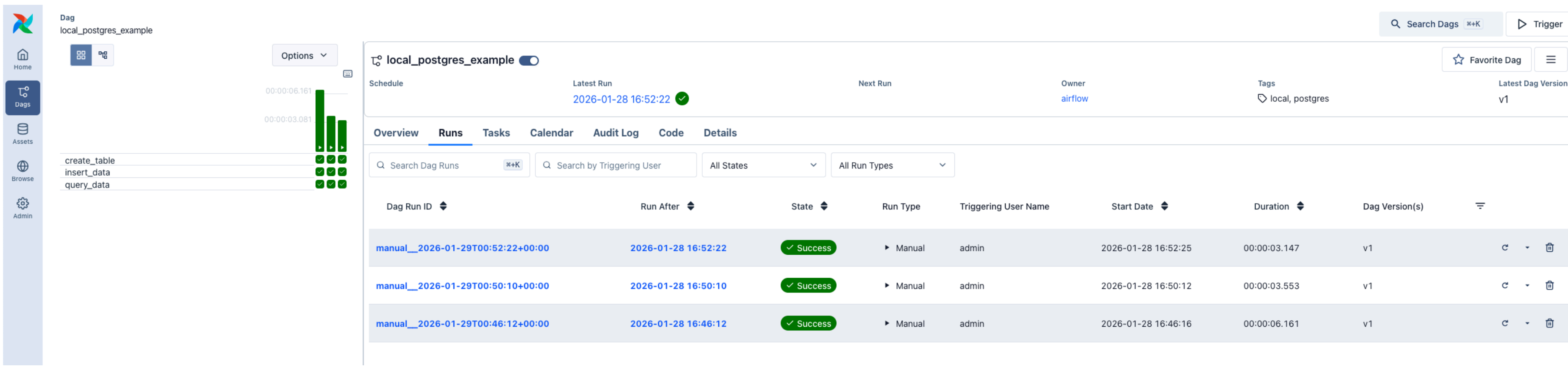
0 Failed Runs

00:00:10
00:00:05
0
00:00:04
2026-01-28 16:46:12
2026-01-28 16:50:10
2026-01-28 16:52:22

Airflow api-server

Debugging

- If fails and needs to debug, go to “Details” ➡ Click “Failed” ➡ Click “task_id” for the failed DAG ➡ Click “Logs”



Dag
local_postgres_example

00:00:06.161
00:00:03.081

create_table
insert_data
query_data

local_postgres_example

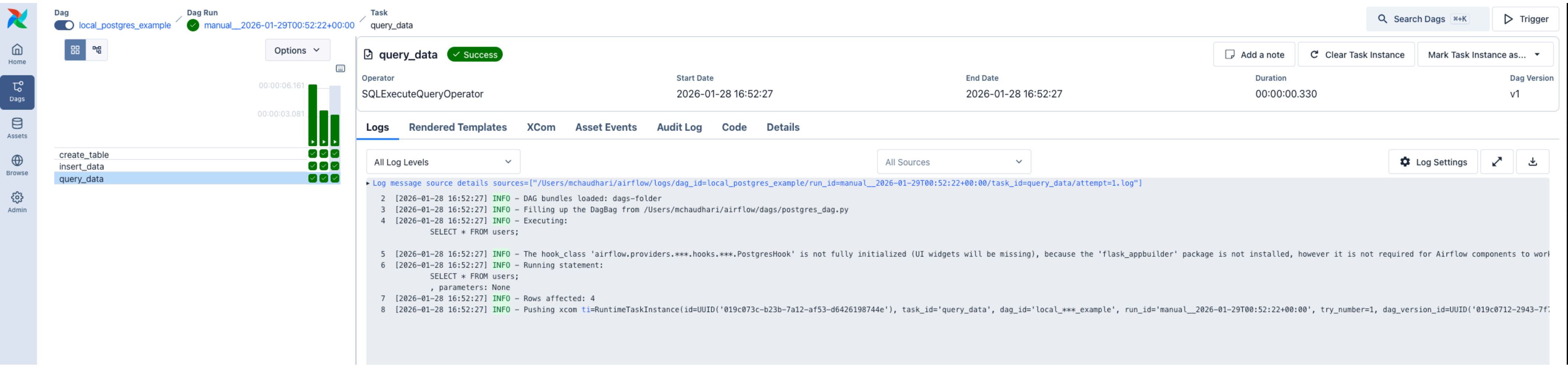
Schedule	Latest Run	Next Run	Owner	Tags	Latest Dag Version
	2026-01-28 16:52:22 ✓		airflow	local, postgres	v1

Dag Run ID	Run After	State	Run Type	Triggering User Name	Start Date	Duration	Dag Version(s)
manual__2026-01-29T00:52:22+00:00	2026-01-28 16:52:22	✓ Success	Manual	admin	2026-01-28 16:52:25	00:00:03.147	v1
manual__2026-01-29T00:50:10+00:00	2026-01-28 16:50:10	✓ Success	Manual	admin	2026-01-28 16:50:12	00:00:03.553	v1
manual__2026-01-29T00:46:12+00:00	2026-01-28 16:46:12	✓ Success	Manual	admin	2026-01-28 16:46:16	00:00:06.161	v1

Airflow api-server

Debugging

- If fails and needs to debug, go to “Details” ➡ Click “Failed” ➡ Click “task_id” for the failed DAG ➡ Click “Logs”



Dag: local_postgres_example
Dag Run: manual__2026-01-29T00:52:22+00:00
Task: query_data

create_table
insert_data
query_data

query_data ✓ Success

Operator	Start Date	End Date	Duration	Dag Version
SQLExecuteQueryOperator	2026-01-28 16:52:27	2026-01-28 16:52:27	00:00:00.330	v1

All Log Levels
All Sources

```
▸ Log message source details sources=["/Users/mchaudhari/airflow/logs/dag_id=local_postgres_example/run_id=manual__2026-01-29T00:52:22+00:00/task_id=query_data/attempt=1.log"]
[2026-01-28 16:52:27] INFO - DAG bundles loaded: dags-folder
[2026-01-28 16:52:27] INFO - Filling up the DagBag from /Users/mchaudhari/airflow/dags/postgres_dag.py
[2026-01-28 16:52:27] INFO - Executing:
        SELECT * FROM users;

[2026-01-28 16:52:27] INFO - The hook_class 'airflow.providers.***.hooks.***.PostgresHook' is not fully initialized (UI widgets will be missing), because the 'flask_appbuilder' package is not installed, however it is not required for Airflow components to wor
[2026-01-28 16:52:27] INFO - Running statement:
        SELECT * FROM users;
        , parameters: None
[2026-01-28 16:52:27] INFO - Rows affected: 4
[2026-01-28 16:52:27] INFO - Pushing xcom ti=RuntimeTaskInstance(id=UUID('019c073c-b23b-7a12-af53-d6426198744e'), task_id='query_data', dag_id='local_***_example', run_id='manual__2026-01-29T00:52:22+00:00', try_number=1, dag_version_id=UUID('019c0712-2943-7f
```

Google Cloud Composer

Google Cloud Composer Environment

- Self-contained Airflow deployed on Google Kubernetes Engine
- If possible, recorded tutorial will be posted.

https://cloud.google.com/composer/docs/how-to/managing/creating

Contents

Course Overview

Workflow Management

Apache Airflow

Airflow DAG Creation

References

Apache Airflow Online Documentation, https://airflow.apache.org/docs/apache-airflow/

Spark Online Documentation, http://spark.apache.org/docs/latest/

MongoDB. Online Documentation, https://docs.mongodb.com/

Google Cloud Platform Online Documentation, https://cloud.google.com/docs

Shannon Bradshaw, Eoin Brazil, Kristina Chodorow. MongoDB: The Definitive Guide. O'Reilly Media, Inc. 2019.

Jean-Georges Perrin. Spark in Action, O'Reilly Publications, 2020.

Sadalage, Pramod J., and Martin Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2012.