The Godfather of Talent | Futurense

Airflow Cheat Sheet

Airflow CLI (Command Line Interface)

GENERAL COMMANDS

- `airflow version` :Shows the version of Airflow
- `airflow info` :Provides sys. and dependencies info.
- `airflow webserver` :Starts Airflow webserver
- `airflow scheduler` :Starts the Airflow scheduler
- `airflow worker` :Starts a new Airflow worker

DAG COMMANDS

- `airflow dags list` :List all the DAGs
- `airflow dags delete DAG_ID` :Deletes a DAG
- `airflow dags pause DAG_ID` :Pauses a DAG
- `airflow dags unpause DAG_ID` :Unpauses a DAG
- `airflow dags trigger -d DAG_ID` :Manually triggers a DAG run
- `airflow dags test DAG_ID EXECUTION_DATE` :Runs a specific DAG

TASK COMMANDS

- `airflow tasks list DAG_ID` :List all the tasks in a DAG
- `airflow tasks test DAG_ID TASK_ID EXECUTION_DATE` :Test a task instance. This will not produce any database entries or trigger handlers
- `airflow tasks run DAG_ID TASK_ID EXECUTION_DATE` :Run a single task instance
- `airflow tasks clear -t TASK_REGEX -s START_DATE -e END_DATE DAG_ID` :Clears the task instances matching the filters from the history.

POOL COMMANDS

- `airflow pools list` :List all pools
- `airflow pools delete POOL_NAME` :Delete a pool
- `airflow pools get POOL_NAME` :Get pool size and used slots count
- `airflow pools set POOL_NAME POOL_SLOT_COUNT POOL_DESCRIPTION` :Create a new pool with the given parameters.

CONNECTION COMMANDS

- `airflow connections list` :List all connections
- `airflow connections add 'CONN_ID' --conn-type 'CONN_TYPE' --conn-host 'CONN_HOST' --conn-login 'CONN_LOGIN' --conn-password 'CONN_PASSWORD' --conn-schema 'CONN_SCHEMA' --conn-port 'CONN_PORT'` :Add new connection
- `airflow connections delete 'CONN_ID'` :Delete a connection

VARIABLE COMMANDS

- `airflow variables list` :List all variables
- `airflow variables get 'VAR_KEY'` :Get value of a variable
- `airflow variables set 'VAR_KEY' 'VAR_VALUE'` :Set a variable and its value
- `airflow variables delete 'VAR_KEY'` :Delete a variable

Creating a DAG (Directed Acyclic Graph)

BASIC DAG DECLARATION

```
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from datetime import datetime, timedelta
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2022, 1, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}
with DAG('my_first_dag', default_args=default_args, description='A simple tutorial DAG',
schedule_interval=timedelta(days=1),) as dag:
    t1 = DummyOperator(task_id='task_1')
    t2 = DummyOperator(task_id='task_2')
    t1 >> t2
```

DAG ARGUMENTS

- `owner` :Owner of the DAG
- `start_date` :The start date of the DAG
- `email` :The email(s) to notify on task failure
- `retries` :Number of times a failed task is retried
- `retries` :Number of times a failed task is retried
- `retry_delay` :Time delay between retries
- `depends_on_past` :If True, a task can't run unless the previous schedule succeeded
- `email_on_failure` :If True, emails are sent on task failure
- `email_on_retry` :If True, emails are sent on each task retry
- `description` :Description of the DAG
- `schedule_interval` :The interval of DAG execution

ORDERING AND DEPENDENCIES

- `t1 >> t2` :t1 precedes t2
- `t1 << t2` :Get value of a variable
- `t1.set_upstream(t2)` : t1 follows t2.
- `t1.set_downstream(t2)` :t1 precedes t2
- `chain(t1, t2, t3)` :Chains the tasks in sequence (t1 >> t2 >> t3).
- `cross_downstream([t1, t2], [t3])` :Creates dependencies t1 >> t3 and t2 >> t3

Using Operators

BASIC OPERATOR USAGE

```
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from datetime import datetime
with DAG('my_dag', start_date=datetime(2022, 1, 1)) as dag:
    task = DummyOperator(task_id='my_task')
```

COMMON OPERATORS AND THEIR PARAMETERS

- **DummyOperator** :Does nothing, but useful for testing or if you need a no-op operator. Its main parameter is **task_id**

```
DummyOperator(task_id='dummy_task')
```

- **PythonOperator** :Executes a Python function. Main parameters are **task_id** and **python_callable** (the function to run)

```
from airflow.operators.python_operator import PythonOperator
def my_func():
    print('Hello, World!')
PythonOperator(task_id='python_task', python_callable=my_func)
```

- **BashOperator** :Executes a bash command. Main parameters are **task_id** and **bash_command**

```
from airflow.operators.bash_operator import BashOperator
BashOperator(task_id='bash_task', bash_command='echo Hello, World!')
```

- **EmailOperator** :Sends an email. Main parameters are **task_id**, **to** (recipient address), **subject**, and **html_content**

```
from airflow.operators.email_operator import EmailOperator
EmailOperator(task_id='email_task', to='test@example.com', subject='Hello',
html_content='<h1>World!</h1>')
```

- **SubDagOperator** :Executes a sequence of tasks in a sub-DAG.

```
from airflow.operators.subdag_operator import SubDagOperator
SubDagOperator(task_id='subdag_task', subdag=subdag())
```

CREATING A CUSTOM OPERATOR

Creating a custom operator involves defining a new Python class that inherits from the BaseOperator class.

```
from airflow.models import BaseOperator
from airflow.utils.decorators import apply_defaults
class MyFirstOperator(BaseOperator):
    @apply_defaults
```

```
    def __init__(self, my_param, *args, **kwargs):
        super(MyFirstOperator, self).__init__(*args, **kwargs)
        self.my_param = my_param
    def execute(self, context):
        log.info('My first operator, param: %s', self.my_param)
```

In this example, MyFirstOperator takes a parameter my_param. The execute method is called when the operator is run.

Tasks

BASIC TASK DECLARATION

- A task is an instance of an operator. Here's a basic task declaration using the PythonOperator:

```
from airflow.operators.python_operator import PythonOperator
def hello_world():
    print('Hello, World!')
task = PythonOperator(task_id='hello_world_task', python_callable=hello_world)
```

In this example, task_id is a unique identifier for the task, and python_callable is the Python function that the task will execute

TASK ARGUMENTS

- `depends_on_past` : If set to True, the task instance will only run if the previous task instance succeeded. Defaults to False
- `wait_for_downstream` : If set to True, this task instance will wait for tasks downstream of the previous task instance to finish before running.
- `retries` : The number of times to retry failed tasks. Defaults to 0.
- `retry_delay` : The time to wait before retrying the task. Defaults to five minutes. This must be a datetime.timedelta object.
- `queue` : The name of the queue to use for task execution. This is used with the CeleryExecutor and KubernetesExecutor.
- `priority_weight` : Defines the priority of the task in relation to other tasks. This can be used to prioritize certain tasks in the scheduler.
- `pool` : The name of the pool to use, if any. This can be used to limit parallelism for a set of tasks.
- `trigger_rule` : Defines the rule by which the task gets triggered (all_success, all_failed, all_done, one_success, one_failed, none_failed, none_skipped, etc.)

Example:-

```
task = PythonOperator(
    task_id='hello_world_task',
    python_callable=hello_world,
    depends_on_past=True,
    wait_for_downstream=True,
    retries=3,
    retry_delay=timedelta(minutes=1),
    queue='my_queue',
    priority_weight=10,
    pool='my_pool'
)
```

TaskFlow API

THE @task DECORATOR

```
from airflow.decorators import task
@task
def my_task():
    print('Hello, World!')
```

DYNAMIC OUTPUT TYPES

```
from airflow.decorators import task
from typing import Tuple
@task
def my_task() -> Tuple[int, str]:
    return 42, 'Hello, World!'
```

USING XComArg FOR INTER-TASK COMMUNICATION

```
from airflow.decorators import task
from airflow.models import XComArg
@task
def my_task() -> str:
    return 'Hello, World!'
@task
def greet(msg: XComArg):
    print(f"Greeting: {msg}")
#Use my_task's output in greet task
greet(my_task())
```

Conditional Execution and Branching

The BranchPythonOperator

- The BranchPythonOperator is used when you want to follow different paths in your DAG based on a condition. It takes a Python callable that returns a task id (or a list of task ids).

```
from airflow.operators.python import BranchPythonOperator
def choose_path():
    return 'first_task' if CONDITION else 'second_task'
branch_task = BranchPythonOperator(
    task_id='branch_task',
    python_callable=choose_path
)
#Define first_task and second_task here
```

The ShortCircuitOperator

- The ShortCircuitOperator is used to bypass (short-circuit) a section of the DAG based on a condition. It is a PythonOperator that evaluates a condition and short-circuits the workflow if the condition is False.

```
from airflow.operators.python import ShortCircuitOperator
def check_condition():
    return True if CONDITION else False
condition_task = ShortCircuitOperator(
    task_id='condition_task',
    python_callable=check_condition
)
#Define further tasks here. They won't run if check_condition returns False
```

Dynamic DAGs

CREATING DYNAMIC DAGs

```python
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.utils.dates import days_ago
#Function to create a DAG
def create_dag(dag_id, schedule):
    dag = DAG(
        dag_id=dag_id,
        schedule_interval=schedule,
        start_date=days_ago(2),
        default_args={'retries': 1},
    )
    with dag:
        DummyOperator(task_id='task_1')
        DummyOperator(task_id='task_2')
    return dag
#Loop to create multiple DAGs
for i in range(1, 6):
    dag_id = f"my_dag_{i}"
    globals()[dag_id] = create_dag(dag_id,
                                   '@daily')
```

DYNAMIC TASK GENERATION

```python
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.utils.dates import days_ago
dag = DAG(
    'my_dag',
    start_date=days_ago(2),
    default_args={'retries': 1},
)
#Loop to create multiple tasks
for i in range(1, 6):
    DummyOperator(task_id=f'task_{i}', dag=dag)
```

Error Handling

CALLBACKS

- `on_failure_callback` : Function called on task failure. Use for custom failure behavior (e.g., notifications or cleanup).
- `on_retry_callback` : Function called before task retry. Use for pre-retry adjustments or logging.
- `on_success_callback` : Function called on task success. Use for post-success actions (e.g., notifications or downstream triggers).

Example:-

```python
#Define your callback functions
def task_fail_callback(context): #Write body to handle failure below
def task_retry_callback(context): #Write body to handle retry below
def task_success_callback(context): #Write body to handle success below
#Assign callbacks to your task
task = PythonOperator(
    task_id='task',
    python_callable=my_function,
    on_failure_callback=task_fail_callback,
    on_retry_callback=task_retry_callback,
    on_success_callback=task_success_callback,
    dag=dag,
)
```

TRIGGERRULE CLASS FOR HANDLING TASK DEPENDENCIES

- `TriggerRule.ALL_SUCCESS` : All parent tasks must have succeeded. This is the default rule.
- `TriggerRule.ALL_FAILED` : All parent tasks are in the 'failed' or 'upstream_failed' state.
- `TriggerRule.ALL_DONE` : All parent tasks are done with their execution.
- `TriggerRule.ONE_SUCCESS` : At least one parent task has succeeded.
- `TriggerRule.ONE_FAILED` : At least one parent task is in the 'failed' or 'upstream_failed' state.
- `TriggerRule.NONE_FAILED` : No parent task is in the 'failed' or 'upstream_failed' state.

Example:-

```
from airflow.utils.trigger_rule import TriggerRule
#Create tasks with different trigger rules
task1 = DummyOperator(task_id='task1', dag=dag)
task2 = DummyOperator(task_id='task2', dag=dag, trigger_rule=TriggerRule.ALL_SUCCESS)
task3 = DummyOperator(task_id='task3', dag=dag, trigger_rule=TriggerRule.ONE_FAILED)
task4 = DummyOperator(task_id='task4', dag=dag, trigger_rule=TriggerRule.ALL_DONE)
#Setting dependencies
task1 >> task2 >> [task3, task4]
```

Integration with Other Tools

EXAMPLE FOR FILE TRANSFER OPERATORS

- **S3ToRedshiftOperator**

```
from airflow.providers.amazon.aws.transfers.s3_to_redshift import S3ToRedshiftOperator
transfer_s3_redshift = S3ToRedshiftOperator(
 task_id='transfer_s3_redshift',
   schema='SCHEMA_NAME',
   table='TABLE_NAME',
   s3_bucket='S3_BUCKET',
   s3_key='S3_KEY',
   copy_options=['csv'],
   dag=dag,
)
```

- **GoogleCloudStorageToBigQueryOperator**

```
from airflow.providers.google.cloud.transfers.gcs_to_bigquery import GoogleCloudStorageToBigQueryOperator
transfer_gcs_bq = GoogleCloudStorageToBigQueryOperator(
    task_id='gcs_to_bq_example',
    bucket='BUCKET_NAME',
    source_objects=['SOURCE_FILE_NAME.csv'],
    destination_project_dataset_table='DESTINATION_DATASET.TABLE_NAME',
    schema_fields=[{'name': 'column1', 'type': 'STRING'}, {'name': 'column2', 'type': 'STRING'}],
    write_disposition='WRITE_TRUNCATE',
    dag=dag,
 )
```

EXAMPLE FOR DATABASE OPERATORS

- **PostgresOperator**

```
from airflow.providers.postgres.operators.postgres import PostgresOperator
t1 = PostgresOperator(
    task_id='run_sql',
    sql='sql_file.sql',
    postgres_conn_id='postgres_default',
    dag=dag,
)
```

- **MySqlOperator**

```
from airflow.providers.mysql.operators.mysql import MySqlOperator
sql_op = MySqlOperator(
    task_id='run_sql',
    sql='sql_file.sql',
    mysql_conn_id='mysql_default',
    dag=dag,
)
```

Testing and Debugging

- **TESTING A TASK :** `airflow tasks test DAG_ID TASK_ID EXECUTION_DATE`
- **BACKFILLING DATA :** `airflow dags backfill -s START_DATE -e END_DATE DAG_ID`
- **RUNNING A TASK INSTANCE :** `airflow tasks run DAG_ID TASK_ID EXECUTION_DATE`

TriggerRule Class

- **all_success** : All parent tasks must have succeeded. This is the default trigger rule.

```
task3 = DummyOperator(task_id='task3', trigger_rule=TriggerRule.ALL_SUCCESS,dag=dag)
```

- **all_failed** : All parent tasks must have failed

```
task3 = DummyOperator(task_id='task3',trigger_rule=TriggerRule.ALL_FAILED, dag=dag)
```

- **all_done** : All parent tasks are done with their execution.

```
task3 = DummyOperator(task_id='task3',trigger_rule=TriggerRule.ALL_DONE, dag=dag)
```

- **one_success** : Fires if at least one parent has succeeded, it allows triggering the task soon after one parent succeeds.

```
task3 = DummyOperator(task_id='task3',trigger_rule=TriggerRule.ONE_SUCCESS, dag=dag)
```

- **one_failed** : Fires if at least one parent has failed, it allows triggering the task soon after one parent fails.

```
task3 = DummyOperator(task_id='task3',trigger_rule=TriggerRule.ONE_FAILED, dag=dag)
```

- **none_failed** : Fires as long as no parents have failed, i.e. all parents have succeeded or been skipped.

```
task3 = DummyOperator(task_id='task3',trigger_rule=TriggerRule.NONE_FAILED, dag=dag)
```

The Godfather of Talent | Futurense

How to use ChatGPT Effectively for Airflow Workflow

ChatGPT and CodeGPT Prompts for Airflow

- **SETTING UP A DAG IN APACHE AIRFLOW**
  - **ChatGPT:** *"Explain the step-by-step process to setup a Directed Acyclic Graph (DAG) in Apache Airflow."*
  - **CodeGPT:** *"Generate a simple DAG in Apache Airflow with 3 tasks that run in sequence."*
- **IMPLEMENTING CONDITIONAL EXECUTION IN APACHE AIRFLOW**
  - **ChatGPT:** *"Detail the method for implementing conditional execution in Apache Airflow. What operators are commonly used?"*
  - **CodeGPT:** *"Create an example DAG in Apache Airflow demonstrating conditional execution using the BranchPythonOperator."*
- **HANDLING ERRORS IN APACHE AIRFLOW**
  - **ChatGPT:** *"How should I handle errors in Apache Airflow tasks? What are the best practices?"*
  - **CodeGPT:** *"Generate a sample task in Apache Airflow with on_failure_callback, on_retry_callback, and on_success_callback."*
- **INTERACTING WITH DATABASES IN APACHE AIRFLOW**
  - **ChatGPT:** *"Explain the process of setting up a connection to a PostgreSQL database in Apache Airflow."*
  - **CodeGPT:** *"Generate a sample PostgresOperator task in an Airflow DAG that executes a simple SQL command."*
- **AIRFLOW WITH KUBERNETES**
  - **ChatGPT:** *"Explain how to leverage Kubernetes in Airflow? What are the benefits and complexities involved?"*
  - **CodeGPT:** *"Generate a simple KubernetesPodOperator task in an Airflow DAG."*
- **BACKFILLING DATA IN AIRFLOW**
  - **ChatGPT:** *"How to backfill data in Airflow? What considerations should be kept in mind when doing so?"*
  - **CodeGPT:** *"Show me an example of an Airflow command for backfilling a DAG from a specific date."*
- **SCALING AIRFLOW**
  - **ChatGPT:** *"What strategies can be applied to scale Airflow? How do I ensure my Airflow setup remains robust as data volume increases?"*
  - **CodeGPT:** *"Provide an example of how to set up the LocalExecutor in Airflow for better scalability."*
- **EXCEPTION HANDLING IN AIRFLOW**
  - **ChatGPT:** *"Explain how to handle exceptions in Apache Airflow effectively. What are the best practices?"*
  - **CodeGPT:** *"Provide a Python snippet demonstrating the use of on_failure_callback in Apache Airflow for exception handling."*
- **CUSTOM AIRFLOW PLUGINS**
  - **ChatGPT:** *"How can I develop and use custom plugins in Airflow? What are the steps and important considerations?"*
  - **CodeGPT:** *"Generate a skeleton Python code for a custom Airflow plugin."*

- **SECURE CONNECTIONS IN AIRFLOW**
  - **ChatGPT:** *"What are the best practices to handle secure connections in Airflow? How to use connection hooks?"*
  - **CodeGPT:** *"Generate a Python snippet demonstrating the use of connection hooks for secure database connection."*
- **AIRFLOW WITH AWS: S3 AND REDSHIFT**
  - **ChatGPT:** *"How can Apache Airflow be used with AWS services, such as S3 and Redshift? Describe with use-cases."*
  - **CodeGPT:** *"Write an example DAG using **S3ToRedshiftOperator** to load data from S3 to Redshift."*
- **AIRFLOW IN CI/CD PIPELINES**
  - **ChatGPT:** *"How can Apache Airflow be incorporated into a CI/CD pipeline? What are the benefits?"*
  - **CodeGPT:** *"Generate a Python snippet showing an example DAG, which could be part of a CI/CD pipeline."*
- **COMPLEX DEPENDENCY MANAGEMENT IN AIRFLOW**
  - **ChatGPT:** *"How to manage complex dependencies in Apache Airflow? Share tips for handling inter-DAG and intra-DAG dependencies."*
  - **CodeGPT:** *"Generate a Python snippet showing an example DAG, which could be part of a CI/CD pipeline."*
- **AIRFLOW WITH GOOGLE CLOUD PLATFORM SERVICES**
  - **ChatGPT:** *"How can Apache Airflow be integrated with Google Cloud Platform services, such as GCS and BigQuery? Describe use-cases."*
  - **CodeGPT:** *"Generate a Python script showing an example DAG using **GoogleCloudStorageToBigQueryOperator** to load data from GCS to BigQuery."*
- **ADVANCED ERROR HANDLING IN AIRFLOW**
  - **ChatGPT:** *"Explain strategies for advanced error handling in Apache Airflow, including fallbacks and failovers."*
  - **CodeGPT:** *"Create a Python snippet demonstrating the use of **on_retry_callback** and **on_failure_callback** for advanced error handling."*
- **AIRFLOW SCALABILITY AND HIGH AVAILABILITY**
  - **ChatGPT:** *"What strategies can be applied to ensure high availability and scalability of Apache Airflow deployments?"*
  - **CodeGPT:** *"Create a sample Airflow configuration demonstrating optimal settings for a highly available and scalable deployment."*
- **AIRFLOW WITH MACHINE LEARNING PIPELINES**
  - **ChatGPT:** *"How can Apache Airflow be leveraged for building and managing Machine Learning pipelines?"*
  - **CodeGPT:** *"Write a Python script showing a simple example DAG for a machine learning pipeline using scikit-learn or TensorFlow."*
- **MONITORING AND ALERTING IN AIRFLOW**
  - **ChatGPT:** *"What are the best practices for setting up monitoring and alerting for Apache Airflow workflows?"*
  - **CodeGPT:** *"Create an example of how to set up a custom alert using **email_on_failure** and **email_on_retry** parameters in a task."*