1. ¿Qué es Apache Airflows?

- Plataforma para armar, monitorear y schedulear *workflows* de manera programática
- Inicio en 2014 en Airbnb y Open Source desde mediados de 2015. Fue utilizado por HBO, Twitter, ING, Paypal, Reddit, Yahoo y muchas más.
- Los workflows se escriben en Python lo que permite abstraer comportamiento, testear y usar las herramientas de desarrollo para Python.
- Tiene una UI que permite visualizar el estado de cada *workflow*, monitorear su progreso, métricas de cada tarea, realizar troubleshooting, manejo de credenciales y mucho más.

2. Problemas con CRON & otras opciones

CRON hace un trabajo deficiente en el manejo de dependencias de tareas y visualización.

- Estrategia deficiente o nula para reintentar tareas o reabastecimientos.
- Datos limitados sobre tiempos de tareas, duraciones de ejecución y fallas.
- Necesita ingresar al servidor para verificar los registros e interactuar.
- No hay una forma fácil de escalar más allá de una máquina.
- Algunas preguntas que son difíciles de responder:
  - ¿Sabe cuándo fallan sus trabajos de CRON?
  - ¿Puede detectar cuándo sus tareas se vuelven 3 veces más lentas?
  - ¿Puedes visualizar lo que se está ejecutando actualmente? ¿Qué está en cola?
  - ¿Tiene componentes reutilizables que pueda usar en los flujos de trabajo?

3. Terminología

DAG: grafo dirigido acíclico de las tareas que uno quiere correr (*workflow*). El DAG es simplemente un archivo de python que define la estructura del grafo.

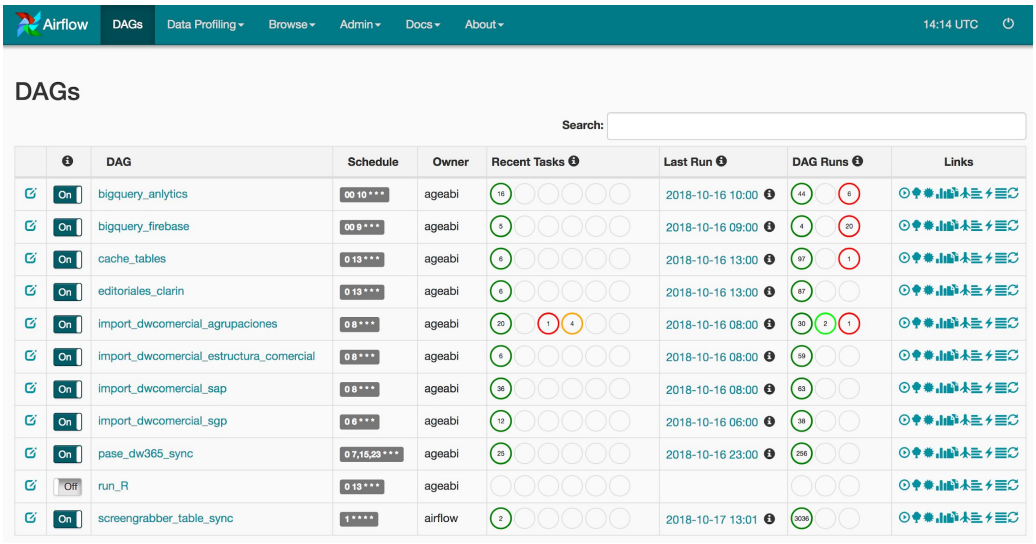
Operator: definen que se va a ejecutar (por ejemplo un comando bash, un insert a una tabla, etc).

Task: instancia de un operador que define un nodo del grafo.

DAG run: instancia de un DAG. Cuando un DAG inicia una corrida entonces Airflow orquesta la ejecución de los operadores respetando dependencias y asignando recursos.

Task instance: corrida particular de una tarea particular para una corrida particular del DAG para un periodo temporal particular.

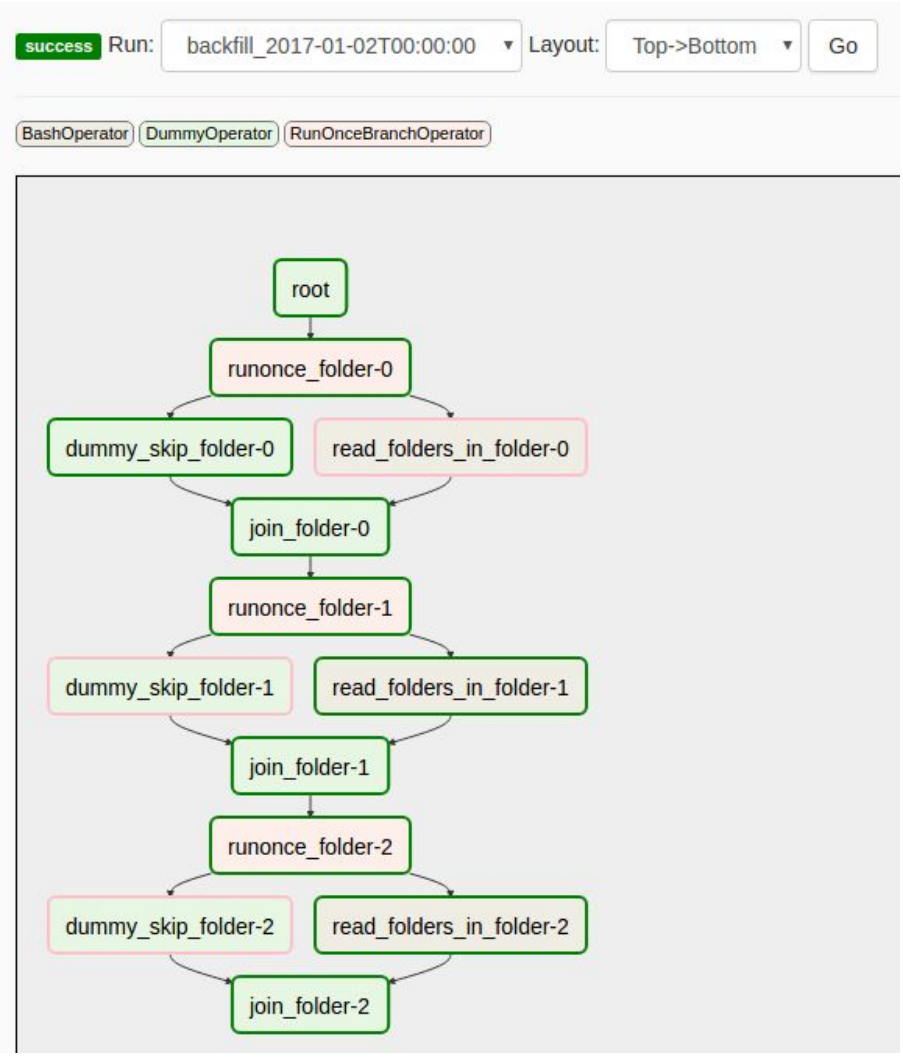
4. DAGs y Ejecuciones



Archivos de definición *DAGs*

Los DAGs son solo archivos de configuración que definen la estructura del DAG usando Python como código. Los DAGs no ejecutan ningún procesamiento de datos, solamente la ejecución real de un DAG.

Las tareas definidas se ejecutarán en diferentes contextos, diferentes *workers*, diferentes puntos en el tiempo y en su mayoría no se comunican entre sí Deberían ejecutarse rápidamente (cientos de milisegundos) porque serán evaluados a menudo por el *Scheduler* de Airflow.



5. Ejemplo de un *DAG*

```python
from datetime import datetime, timedelta

from airflow.utils import dates
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator

import ibis


default_args = {
    'owner': 'mutt',
    'depends_on_past': True,
    'start_date':  datetime(2018, 8, 15),
    'email': ['juan@muttdata.ai'],
    'email_on_failure': True,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # 'catchup': False,
}


def do_impala_query(query):
    client = ibis.impala.connect(host='host',user='user')
    cursor = client.raw_sql(query, results=True)
    print(cursor.fetchall())
```

```python
dag = DAG(
        'import_anomaly_detection_data',
        default_args=default_args,
        schedule_interval='30 * * * *')

q = """
SELECT * from prod_ar.{{ params.table_name }} limit 100
"""
with dag:
    run_this = PythonOperator(
        task_id='do_impala_query',
        provide_context=True,
        python_callable=do_impala_query,
        op_kwargs={'query': q.format(table_name='GSM_FILE_DETAILS')},
        dag=dag)

    then_this = BashOperator(
        task_id='templated_bash_command',
        bash_command='echo "{{ macros.ds_add(ds, 7)}}"',
        dag=dag)

    run_this >> then_this
```

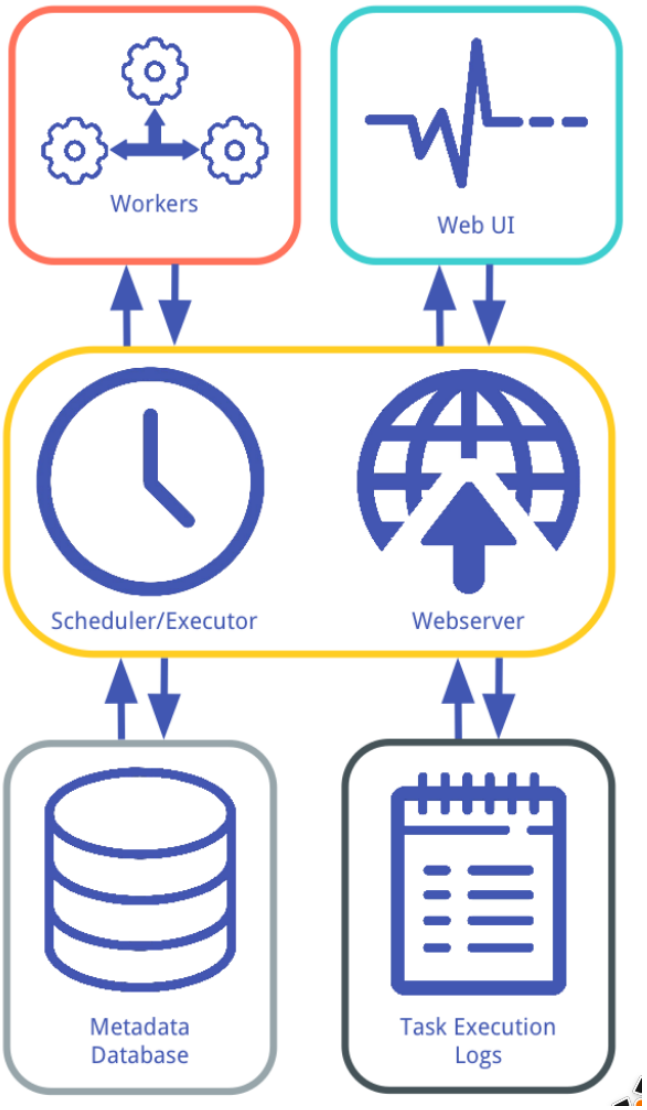
Instalación Local

```
# airflow needs a home, ~/airflow is the default,
# but you can lay foundation somewhere else if you prefer
# (optional)
export AIRFLOW_HOME=~/airflow

# install from pypi using pip
pip install apache-airflow

# initialize the database
airflow initdb

# start the web server, default port is 8080
airflow webserver -p 8080

# start the scheduler
airflow scheduler

# visit localhost:8080 in the browser and enable the example dag in the home page
```

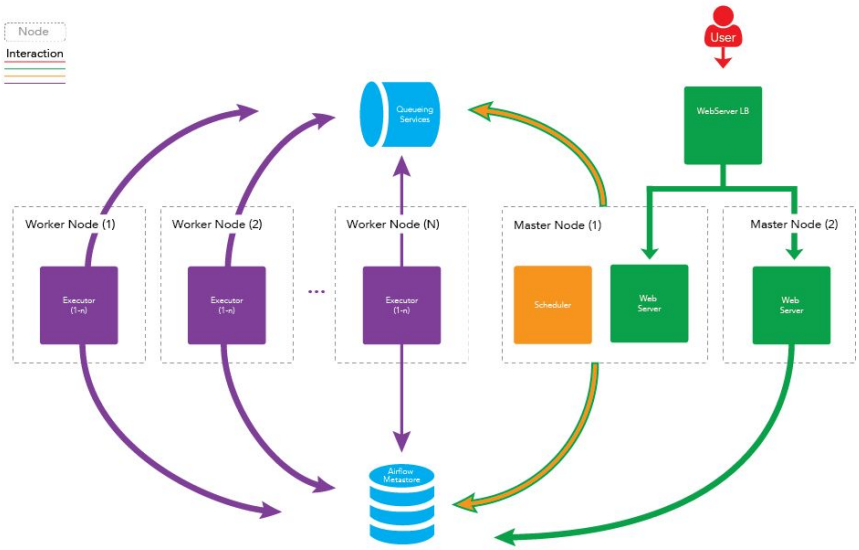
Corra un DAG con:

- `airflow backfill example_bash_operator -s 2015-01-01 -e 2015-01-02`
- O ingrese a la interfaz de usuario de Airflow, actualice el DAG y Airflow lo activará cuando sea necesario.

6. Arquitectura Distribuida



- SequentialExecutor
- LocalExecutor
- CeleryExecutor
- DaskExecutor
- MesosExecutor
- KubernetesExecutor

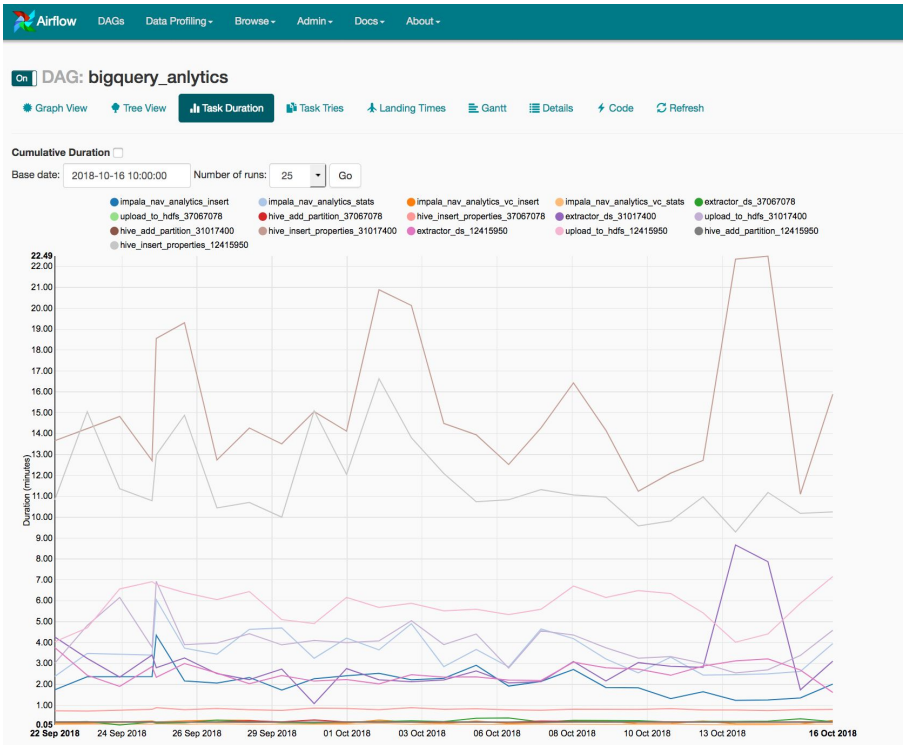


7. El *Scheduler*

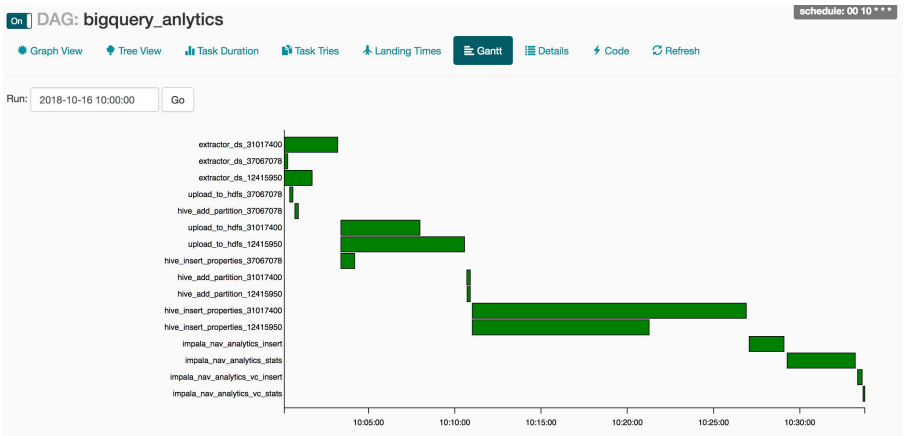
Los procesos del *scheduler* iteran sobre todos los DAGs continuamente y *disparan* las ejecuciones del DAG.

- **execution_date:** el período de tiempo durante el cual se procesarán los datos.
- **start_date:** la fecha de ejecución de la primera ejecución del DAG.
- **end_date:** última fecha de ejecución en la que se ejecutará un DAG.
- **execution_timeout:** tiempo máximo que tardará una tarea en fallar.
- **retries**: cantidad de veces que se reintentará una tarea antes de fallar.
- **retry_delay:** tiempo mínimo entre la ejecución de una tarea y la siguiente después de una falla.

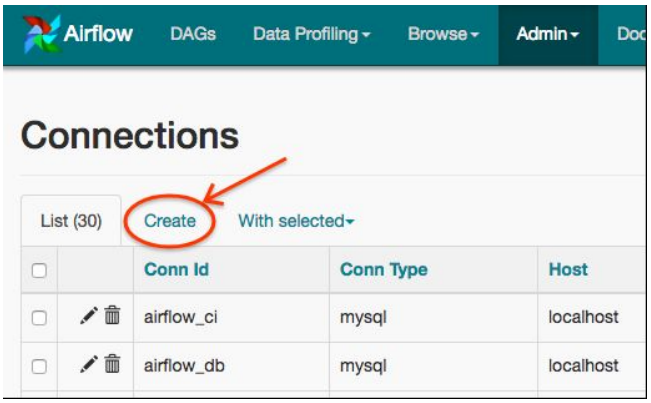
8. Métricas



9. Diagrama de Gantt



10. Conexiones y variables




```python
from airflow.models import Variable
foo = Variable.get("foo")
bar = Variable.get("bar", deserialize_json=True)
```

11. Características Adicionales

- ***Hooks:*** interfaces a plataformas y bases de datos externas como Hive, S3, MySQL, Postgres, etc.
- ***Pools:*** ayudan a limitar el paralelismo de ejecución en conjuntos arbitrarios de tareas. Las tareas se pueden asignar a grupos y tienen un peso de prioridad.
- ***Queues:*** cuando se usa *CeleryExecutor*, las tareas se pueden asignar a una queue y un trabajador puede escuchar y ejecutar tareas en una o varias *queues*.
- ***XComs:*** es una abreviatura de "*cross-communication*", son almacenamientos de claves, valores y *timestamps* destinadas a la comunicación entre tareas. Los *XComs* se almacenan en la base de datos de metadatos de Airflow.

```python
# inside a PythonOperator called 'pushing_task'
def push_function():
    return value

# inside another PythonOperator where provide_context=True
def pull_function(**context):
    value = context['task_instance'].xcom_pull(task_ids='pushing_task')
```

- ***Sensors:*** operadores que esperan que se cumpla una determinada condición y tener éxito. (por ejemplo, esperar a que aparezca un determinado archivo en un directorio)
- **Autenticación:** existen complementos para habilitar la autenticación y autorización a través de LDAP / Kerberos / otros métodos.
- ***Queries ad hoc:*** permite crear gráficos y consultar fuentes de datos configuradas.

12. Jinga Templating

El *Jinga Templating* pone a disposición múltiples variables y macros útiles para ayudar en la manipulación de las fechas.

```
task = BashOperator(
    task_id='bash_script',
    bash_command='./run.sh {{ ds }}',
    dag=dag)
```

Los corchetes indican a Airflow que se trata de una *Jinga template* y que `ds` es una variable creada por Airflow que se reemplaza por la fecha de ejecución en el formato AAAA-MM-DD. Así, en la ejecución del *DAG*:

```
./run.sh 2018-06-04
```

Otra variable útil es `ds_nodash`, donde `'./run.sh ds_nodash'` se representa en:

```
./run.sh 20180604
```

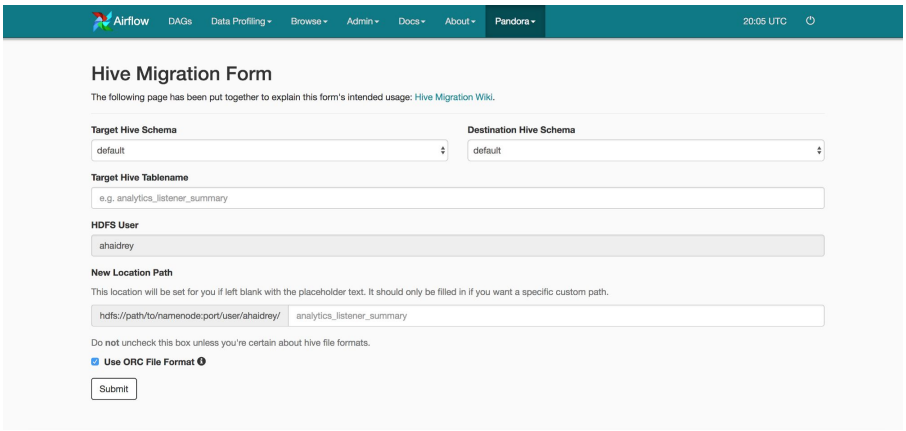
La variable `execution_date` es útil, ya que es un objeto `datetime` de Python y no una cadena como `ds`.

```
'./run.sh {{ (execution_date - macros.timedelta(days=5)).strftime("%Y-%m-%d") }}'
```

```
./run.sh 2018-05-30
```

13. Plugins

Habilita la definición de *hooks*, operadores, sensores, ejecutores de macros y vistas web personalizados. Se usa en muchas empresas para generar DAGs automáticamente para ETLS, ML, pruebas A / B, etc.



14. *Testing*

Los DAG son código, por lo que existen diferentes opciones para testearlos.

- Testear la importación de DAG: iterar en el directorio **dags** de *DAG* y verificar que cada DAG pueda importar o ejecutar el archivo .py desde la línea de comandos.
- Testear los parámetros de DAG: asegúrese de que todos los DAGs tengan parámetros obligatorios como correos electrónicos, *catchup*, etc.
- Prueba unitaria de la lógica de Python: dado que el código ejecutado por PythonOperator es una función de Python, pueden utilizar pruebas unitarias normales.

15. Reglas de Activación de los Operadores

- Los operadores tienen un argumento `trigger_rule` que define la regla por la cual la tarea generada se activa. El valor predeterminado de `trigger_rule` es `all_success`
- Otras opciones:
  - `all_failed`: todos los padres están en un estado de falla o *upstream _failed*.
  - `all_done`: todos los padres finalizaron con su ejecución.
  - `one_failed`: se activa tan pronto como al menos uno de los padres falla, no espera que todos los padres terminen.
  - `one_success`: se activa tan pronto como al menos uno de los padres tiene éxito, no espera que todos los padres terminen.

16. Mejores prácticas de Instalación

- Instale el **package apache-airflow**.
- **LocalExecutor** está bien para empezar.
- Utilice **CeleryExecutor** o **Dask / Kubernetes** para escalar.
- Utilice https://github.com/puckel/docker-airflow si desea utilizar **Docker**.
- Utilice **PostgreSQL** o **MySQL** para metadatos.
- Ajuste las propiedades del *scheduler* para reducir el consumo de CPU.
- Recuerde copiar todos los archivos de configuración y *DAG* en el directorio del **worker/executor**.

17. Mejores prácticas

- Intente equilibrar la legibilidad del DAG y el código de abstracción.
- Utilice `depends_on_past` y `wait_for_downstream` por motivos de seguridad.
- Cambie el nombre del *DAG* cuando cambie la fecha_inicio.
- Las tareas son procesos que se ejecutan en *workers*, limite el tamaño de los datos del proceso localmente.
- Recuerde borrar los registros de tareas después de cierto tiempo.
- Genere vistas personalizadas para personas no técnicas.
- ¡Abstraiga Lógica duplicada!

18. Links Interesantes para Leer

- Beyond CRON: an introduction to Workflow Management Systems
- Why Quizlet chose Apache Airflow for executing data workflows
- Understanding Apache Airflow's key concepts
- How Quizlet uses Apache Airflow in practice
- Get started developing workflows with Apache Airflow
- Airflow: Tips, Tricks, and Pitfalls
- Awesome Apache Airflow
- https://gtoonstra.github.io/etl-with-airflow/principles.html
- Customising Airflow: Beyond Boilerplate Settings