EECS E6893 Big Data Analytics
HW2: Steaming Big Data Analytics & Data Analytics Pipeline

TUTORIAL 1

Apurva Patel
amp2365@columbia.edu

October 04, 2024

Agenda

- Streaming Analytics on Polygon stock data
- Data pipeline orchestration using Apache Airflow

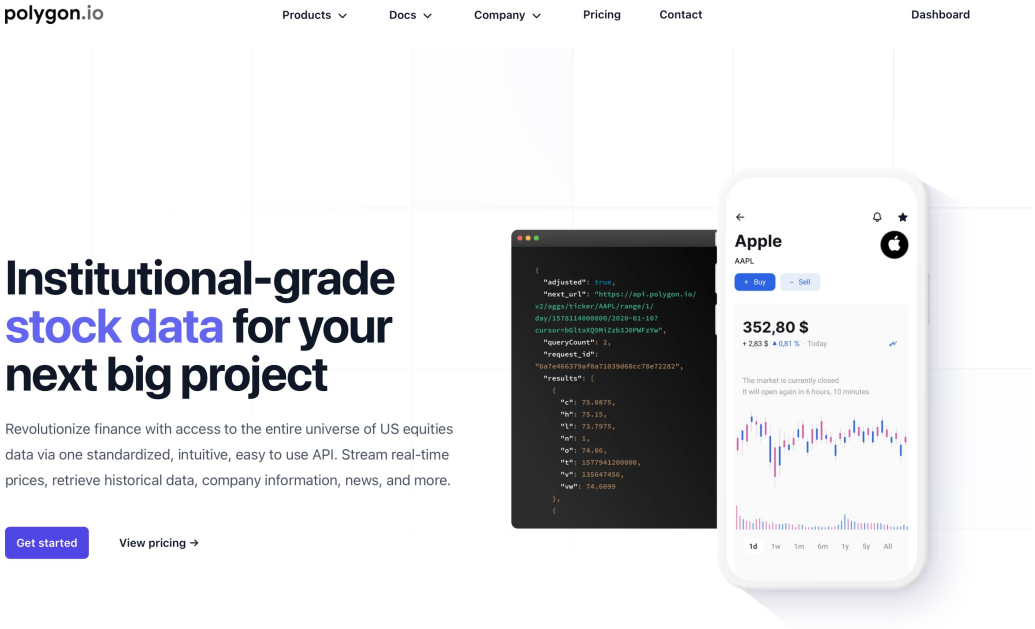
STREAMING ANALYTICS

Spark Streaming

https://spark.apache.org/docs/latest/streaming-programming-guide.html

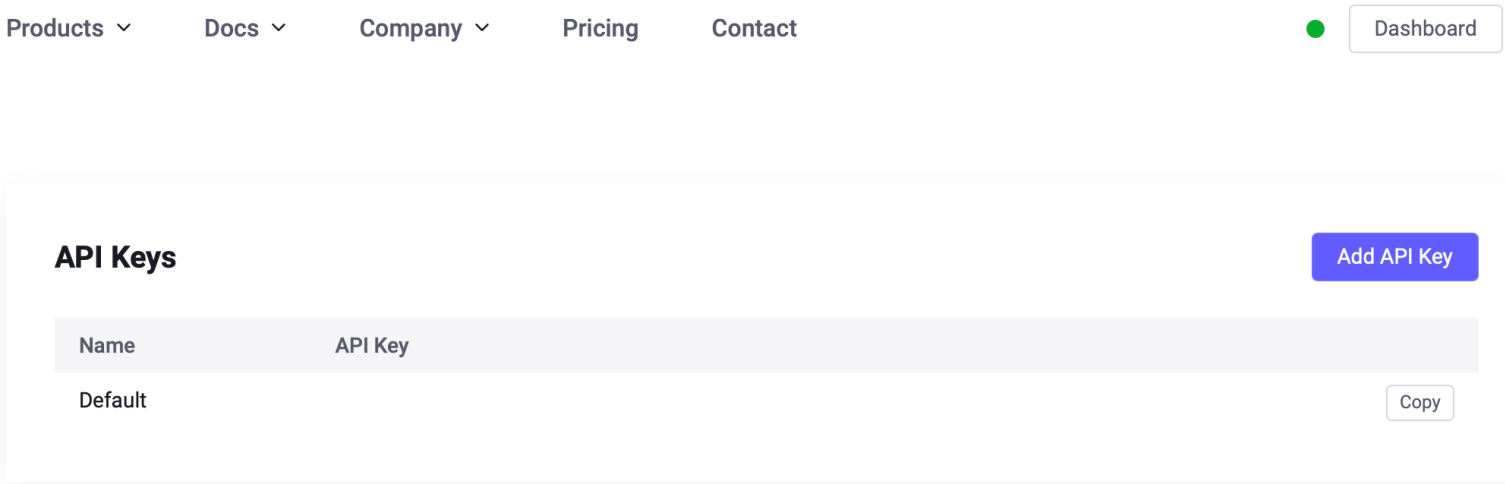
Streaming Analytics Setup

- Step 1: Create your account using your Columbia email ID on [Polygon]



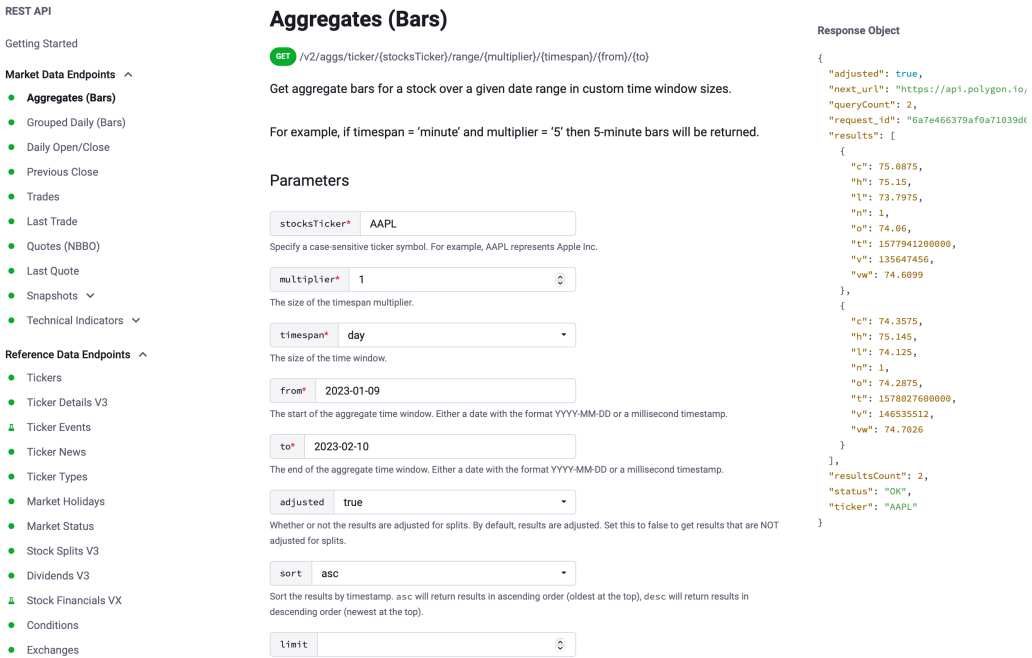
Streaming Analytics Setup

- Step 2: Login and Save your API Key from Dashboard (You will use your own unique API key in the following exercise)



Streaming Analytics Setup

- Step 3: Pull data from Polygon



Streaming Analytics Setup

- Step 4: Create your GCP Dataproc Cluster as done for previous assignments and setup the Polygon API

```
[1] !pip install polygon
```

```
[2] !pip install pandas pyspark requests polygon-api-client
```

Sample Python code to pull data from Polygon API

```
[10] !pip install requests pandas

     import pandas as pd
```

```
[33] BASE_URL = 'https://api.polygon.io'
```

```
[34] def get_histdata_polygon(ticker, start_date, end_date, timespan, multiplier):
         url = f"{BASE_URL}/v2/aggs/ticker/{ticker}/range/{multiplier}/{timespan}/{start_date}/{end_date}?apiKey={API_KEY}"
         response = requests.get(url)
         if response.status_code == 200:
             data = response.json()
             df = pd.DataFrame(data['results'])

             df['timestamp'] = pd.to_datetime(df['t'], unit='ms')

             columns = ['timestamp', 'o', 'h', 'l', 'c', 'v']

             df = df[['timestamp', 'o', 'h', 'l', 'c', 'v']]

             df.columns = ['timestamp', 'open', 'high', 'low', 'close', 'volume']

             df.set_index('timestamp', inplace=True)

             return df
         elif response.status_code == 403:
             print("Error 403: Forbidden. Check your API key and subscription plan.")
         else:
             print(f"Error: {response.status_code} - {response.text}")
         return pd.DataFrame()
```

```
# Example usage
ticker = 'AAPL'
start_date = '2024-10-03'
end_date = '2024-10-04'
timespan = 'minute'  # Can be 'minute', 'hour', 'day', etc.
multiplier = 1  # For 5-minute intervals

data = get_histdata_polygon(ticker, start_date, end_date, timespan, multiplier)
print(data)
```

```
                           open      high       low     close  volume
timestamp
2024-10-03 08:00:00  226.2000  226.2000  226.2000  226.2000  1143.0
2024-10-03 08:01:00  225.9100  225.9100  225.8400  225.8400  1235.0
2024-10-03 08:02:00  225.9800  226.0000  225.9800  226.0000   629.0
2024-10-03 08:03:00  226.0000  226.0000  225.9500  225.9500   493.0
2024-10-03 08:04:00  225.8000  226.0000  225.8000  226.0000   756.0
...                       ...       ...       ...       ...     ...
2024-10-04 23:53:00  226.4899  226.4899  226.4899  226.4899   152.0
2024-10-04 23:54:00  226.4600  226.4600  226.4600  226.4600   519.0
2024-10-04 23:55:00  226.4220  226.4500  226.4220  226.4500   423.0
2024-10-04 23:56:00  226.4200  226.4600  226.3700  226.3700  2134.0
2024-10-04 23:59:00  226.3745  226.3745  226.3745  226.3745   266.0

[1386 rows x 5 columns]
```

TO DO

1. Pull the stock data of “AAPL” at **1-minute level resolution every 5 minutes**. Each such pull made once every 5 minutes should have data from the (current timestamp - one hour) to the current timestamp when you hit the api i.e. **one hour's worth of data in each pull.** Let this entire process run for **30 minutes**. At the end of your program, your code should have generated **~1.5 hours** worth of data for the stock and the api should have been called 7 times (once every 5 minutes, over a period of 30 minutes).

TO DO

2. In each data pull, the data should be **incrementally** loaded into a Spark data frame where the data frame schema will be [‘Stock Name’, “UTC Timestamp”, “c”, “l”, ”h”, ”o”, ”v”]

Sample flow of incremental update operation:

Data Currently in dataframe

Stock	Datetime	c	l	h	o	v
AAPL	2023-10-02 21:45:00	173.7	173.7	173.7	173.7	507.0
AAPL	2023-10-02 21:48:00	173.72	173.72	173.72	173.72	340.0
AAPL	2023-10-02 21:49:00	173.7201	173.7201	173.7201	173.7201	110.0
AAPL	2023-10-02 21:51:00	173.74	173.74	173.74	173.74	111.0
AAPL	2023-10-02 21:54:00	173.66	173.66	173.7	173.7	2005.0
AAPL	2023-10-02 21:57:00	173.67	173.67	173.67	173.67	193.0
AAPL	2023-10-02 21:58:00	173.67	173.67	173.6799	173.6799	420.0
AAPL	2023-10-02 21:59:00	173.65	173.65	173.65	173.65	280.0
AAPL	2023-10-02 22:00:00	173.74	173.65	173.74	173.69	2389.0
AAPL	2023-10-02 22:02:00	173.74	173.735	173.74	173.735	536.0
AAPL	2023-10-02 22:03:00	173.7397	173.7397	173.7397	173.7397	231.0
AAPL	2023-10-02 22:09:00	173.7398	173.7398	173.7398	173.7398	3773.0
AAPL	2023-10-02 22:15:00	173.74	173.74	173.75	173.74	938.0
AAPL	2023-10-02 22:18:00	173.8	173.78	173.8	173.78	1623.0
AAPL	2023-10-02 22:31:00	173.86	173.75	173.86	173.75	738.0
AAPL	2023-10-02 22:32:00	173.87	173.87	173.87	173.87	434.0
AAPL	2023-10-02 22:35:00	173.83	173.83	173.83	173.83	441.0
AAPL	2023-10-02 22:37:00	173.8	173.8	173.82	173.82	341.0
AAPL	2023-10-02 22:38:00	173.8199	173.8199	173.8199	173.8199	527.0
AAPL	2023-10-02 22:39:00	173.75	173.75	173.75	173.75	553.0
AAPL	2023-10-02 22:42:00	173.75	173.75	173.8	173.8	215.0

Data from latest api pull

Stock	Datetime	c	l	h	o	v
AAPL	2023-10-02 21:49:00	173.7201	173.7201	173.7201	173.7201	110.0
AAPL	2023-10-02 21:51:00	173.74	173.74	173.74	173.74	111.0
AAPL	2023-10-02 21:54:00	173.66	173.66	173.7	173.7	2005.0
AAPL	2023-10-02 21:57:00	173.67	173.67	173.67	173.67	193.0
AAPL	2023-10-02 21:58:00	173.67	173.67	173.6799	173.6799	420.0
AAPL	2023-10-02 21:59:00	173.65	173.65	173.65	173.65	280.0
AAPL	2023-10-02 22:00:00	173.74	173.65	173.74	173.69	2389.0
AAPL	2023-10-02 22:02:00	173.74	173.735	173.74	173.735	536.0
AAPL	2023-10-02 22:03:00	173.7397	173.7397	173.7397	173.7397	231.0
AAPL	2023-10-02 22:09:00	173.7398	173.7398	173.7398	173.7398	3773.0
AAPL	2023-10-02 22:15:00	173.74	173.74	173.75	173.74	938.0
AAPL	2023-10-02 22:18:00	173.8	173.78	173.8	173.78	1623.0
AAPL	2023-10-02 22:19:00	173.8494	173.8494	173.86	173.86	3318.0
AAPL	2023-10-02 22:35:00	173.83	173.83	173.83	173.83	441.0
AAPL	2023-10-02 22:37:00	173.8	173.8	173.82	173.82	341.0
AAPL	2023-10-02 22:38:00	173.8199	173.8199	173.8199	173.8199	527.0
AAPL	2023-10-02 22:39:00	173.75	173.75	173.75	173.75	553.0
AAPL	2023-10-02 22:42:00	173.75	173.75	173.8	173.8	215.0
AAPL	2023-10-02 22:43:00	173.8	173.8	173.8	173.8	220.0
AAPL	2023-10-02 22:44:00	173.75	173.75	173.7501	173.7501	285.0
AAPL	2023-10-02 22:45:00	173.78	173.78	173.78	173.78	1815.0
AAPL	2023-10-02 22:46:00	173.79	173.79	173.8	173.8	789.0

Final dataframe state after incremental update

Stock	Datetime	c	l	h	o	v
AAPL	2023-10-02 21:45:00	173.7	173.7	173.7	173.7	507.0
AAPL	2023-10-02 21:48:00	173.72	173.72	173.72	173.72	340.0
AAPL	2023-10-02 21:49:00	173.7201	173.7201	173.7201	173.7201	110.0
AAPL	2023-10-02 21:51:00	173.74	173.74	173.74	173.74	111.0
AAPL	2023-10-02 21:54:00	173.66	173.66	173.7	173.7	2005.0
AAPL	2023-10-02 21:57:00	173.67	173.67	173.67	173.67	193.0
AAPL	2023-10-02 21:58:00	173.67	173.67	173.6799	173.6799	420.0
AAPL	2023-10-02 21:59:00	173.65	173.65	173.65	173.65	280.0
AAPL	2023-10-02 22:00:00	173.74	173.65	173.74	173.69	2389.0
AAPL	2023-10-02 22:02:00	173.74	173.735	173.74	173.735	536.0
AAPL	2023-10-02 22:03:00	173.7397	173.7397	173.7397	173.7397	231.0
AAPL	2023-10-02 22:09:00	173.7398	173.7398	173.7398	173.7398	3773.0
AAPL	2023-10-02 22:15:00	173.74	173.74	173.75	173.74	938.0
AAPL	2023-10-02 22:37:00	173.8	173.8	173.82	173.82	341.0
AAPL	2023-10-02 22:38:00	173.8199	173.8199	173.8199	173.8199	527.0
AAPL	2023-10-02 22:39:00	173.75	173.75	173.75	173.75	553.0
AAPL	2023-10-02 22:42:00	173.75	173.75	173.8	173.8	215.0
AAPL	2023-10-02 22:43:00	173.8	173.8	173.8	173.8	220.0
AAPL	2023-10-02 22:44:00	173.75	173.75	173.7501	173.7501	285.0
AAPL	2023-10-02 22:45:00	173.78	173.78	173.78	173.78	1815.0
AAPL	2023-10-02 22:46:00	173.79	173.79	173.8	173.8	789.0

TO DO

3. Every 5 minutes, after inserting data into the dataframe, compute the **30-minute moving averages** for the stock's "c", "l", "h", "o", and "v" values. Store these moving averages **incrementally** in a separate PySpark dataframe with the schema [“Datetime", “c_MA", "l_MA", "h_MA", "o_MA", "v_MA"], where “Datetime" represents the end timestamp of the moving average window.

Moving Averages for the data called till 2023-10-02 22:42:00

Stock	Datetime	c_ma	l_ma	h_ma	o_ma	v_ma
AAPL	2023-10-02 21:45:00	173.6999969482422	173.6999969482422	173.6999969482422	173.6999969482422	507.0
AAPL	2023-10-02 21:48:00	173.709999084472666	173.709999084472666	173.709999084472666	173.709999084472666	423.5
AAPL	2023-10-02 21:49:00	173.71336364746094	173.71336364746094	173.71336364746094	173.71336364746094	319.0
AAPL	2023-10-02 21:51:00	173.720024108888672	173.720024108888672	173.720024108888672	173.720024108888672	267.0
AAPL	2023-10-02 21:54:00	173.708020019531126	173.708020019531126	173.7160186767578	173.7160186767578	614.6
AAPL	2023-10-02 21:57:00	173.70168304443336	173.70168304443336	173.70834859212224	173.70834859212224	544.3333333333334
AAPL	2023-10-02 21:58:00	173.697156633649546	173.697156633649546	173.704284667968746	173.704284667968746	526.5714285714286
AAPL	2023-10-02 21:59:00	173.6912612915039	173.6912612915039	173.6974983215332	173.6974983215332	495.75
AAPL	2023-10-02 22:00:00	173.696677313910586	173.686676025390626	173.7022213406033	173.696665445963534	706.1111111111111
AAPL	2023-10-02 22:02:00	173.701010131835946	173.6915084838867	173.705999755859386	173.700498962402346	689.1
AAPL	2023-10-02 22:28:00	173.7573380903764	173.7468816583807	173.7601110284978	173.7523831454190	1258.909090909091
AAPL	2023-10-02 22:29:00	173.772792469371446	173.762336037375726	173.7746651389382	173.766937255859386	1239.090909090909
AAPL	2023-10-02 22:31:00	173.7970718383789	173.7835708618164	173.7991317749023	173.7846313476562	1169.9
AAPL	2023-10-02 22:32:00	173.8037012273615	173.7914276123047	173.8055738969282	173.7923916903409	1103.0
AAPL	2023-10-02 22:35:00	173.819100952148436	173.806100463867186	173.821160888671876	173.807160949707036	1180.7
AAPL	2023-10-02 22:37:00	173.817364779385666	173.805546153675446	173.821056019176136	173.808328801935376	1104.3636363636363
AAPL	2023-10-02 22:38:00	173.817576090494786	173.80674235026046	173.820959726969436	173.809293111165376	1056.25
AAPL	2023-10-02 22:39:00	173.81237792968756	173.802377554086556	173.815501286433336	173.804732102614256	1017.5384615384615
AAPL	2023-10-02 22:42:00	173.813161996647	173.803161621093756	173.820131742037256	173.809362558218146	743.8461538461538

Moving Averages for all the data present after next api call at 2023-10-02 22:46:00

Stock	Datetime	c_ma	l_ma	h_ma	o_ma	v_ma
AAPL	2023-10-02 21:45:00	173.6999969482422	173.6999969482422	173.6999969482422	173.6999969482422	507.0
AAPL	2023-10-02 21:48:00	173.709999084472666	173.709999084472666	173.709999084472666	173.709999084472666	423.5
AAPL	2023-10-02 21:49:00	173.7200927734375	173.7200927734375	173.7200927734375	173.7200927734375	110.0
AAPL	2023-10-02 21:51:00	173.730049133300786	173.730049133300786	173.730049133300786	173.730049133300786	110.5
AAPL	2023-10-02 21:54:00	173.706700642903666	173.706700642903666	173.720031738281256	173.720031738281256	742.0
AAPL	2023-10-02 21:57:00	173.697525024414066	173.697525024414066	173.707523345947276	173.707523345947276	604.75
AAPL	2023-10-02 21:58:00	173.692019653320326	173.692019653320326	173.701998901367186	173.701998901367186	567.8
AAPL	2023-10-02 22:37:00	173.817364779385666	173.805546153675446	173.821056019176136	173.808328801935376	1104.3636363636363
AAPL	2023-10-02 22:38:00	173.817576090494786	173.80674235026046	173.820959726969436	173.809293111165376	1056.25
AAPL	2023-10-02 22:39:00	173.81237792968756	173.802377554086556	173.815501286433336	173.804732102614256	1017.5384615384615
AAPL	2023-10-02 22:42:00	173.813161996647	173.803161621093756	173.820131742037256	173.809362558218146	743.8461538461538
AAPL	2023-10-02 22:43:00	173.812222072056366	173.802936008998336	173.818693978445886	173.808694022042456	706.4285714285714
AAPL	2023-10-02 22:44:00	173.808073933919276	173.799406941731776	173.814121500651056	173.804788208007856	678.3333333333334
AAPL	2023-10-02 22:45:00	173.806319236755376	173.798193935157966	173.811988830566406	173.803238868713386	749.375
AAPL	2023-10-02 22:46:00	173.809443473815926	173.801318168640146	173.815114021301276	173.806988716125506	740.0625

To Submit:

- PDF with screenshots of your code, brief explanation of the code worflow and the results i.e. dataframe holding the streamed data and the dataframe holding the moving averages.
- Code file for Streaming Analytics section

Important Notes:

- The data should be pulled anytime between 12pm – 4pm on Monday – Friday since that is the only time you will get real-time stock data when the markets are operational so plan your assignment timeline well. You will most probably not be able to pull data outside these timings accurately and on weekends.
- You may not get the data for each minute level timestep, there may be a few data points missing here and there which is fine but ensure that the majority of the expected timestamps are present in each pull.

AIRFLOW DATA PIPELINING

Workflow

- A sequence of tasks involved in moving from the beginning to the end of a working process
- Started on a schedule or triggered by an event

- A platform that lets you create, schedule, monitor and manage workflows

Principles:

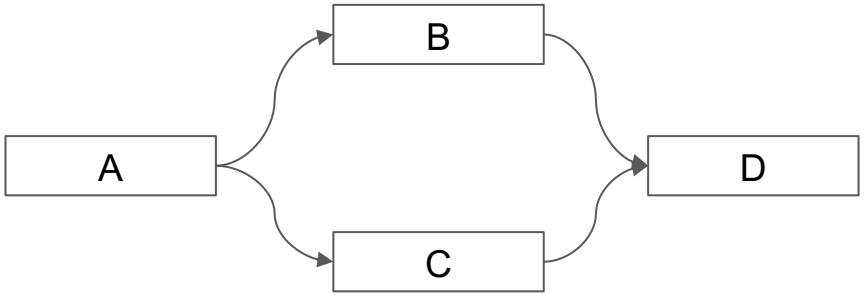
- Scalable
- Dynamic
- Extensible
- Elegant

Features:

- Pure Python
- Useful UI
- Robust Integrations
- Easy to Use
- Open Source

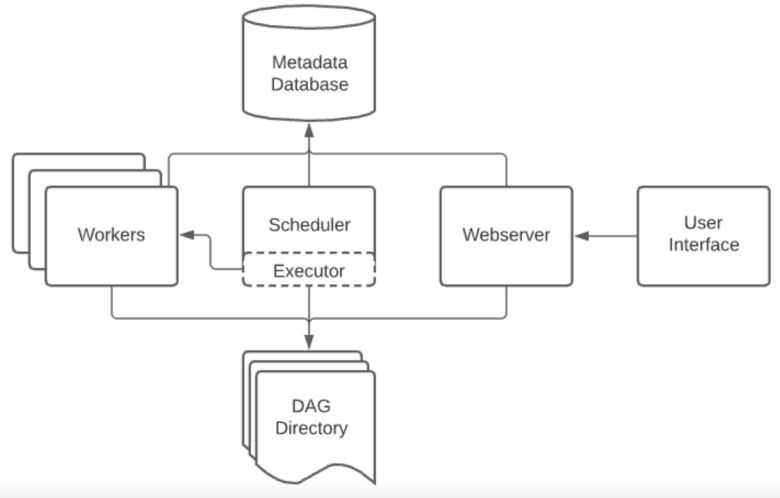
DAG (Directed Acyclic Graph)

- In Airflow, workflows are created using DAGs
- A DAG is a collection of tasks that you want to schedule and run, organized in a way that reflects their relationships and dependencies
- The tasks describe what to do, e.g., fetching data, running analysis, triggering other systems, or more
- A DAG ensures that each task is executed at the right time, in the right order, or with the right issue handling
- A DAG is written in Python



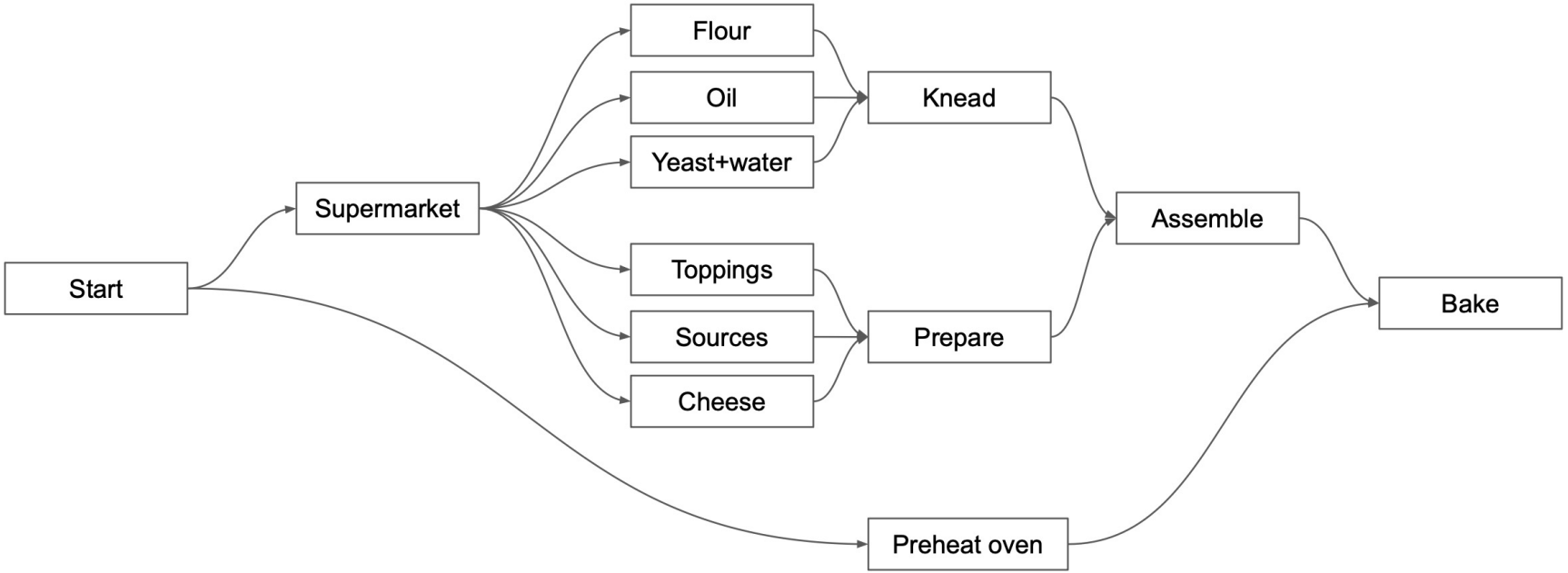
Airflow architecture

- **Scheduler:** handles both triggering scheduled workflows, and submitting tasks to the executor to run.
- **Executor**: handles running tasks.
- **Webserver**: a handy user interface to inspect, trigger and debug the behavior of DAGs and tasks.
- **A folder of DAG files**: read by the scheduler and executor
- **A metadata database**: used by the scheduler, executor and webserver to store state



Workflow

- Cook a pizza



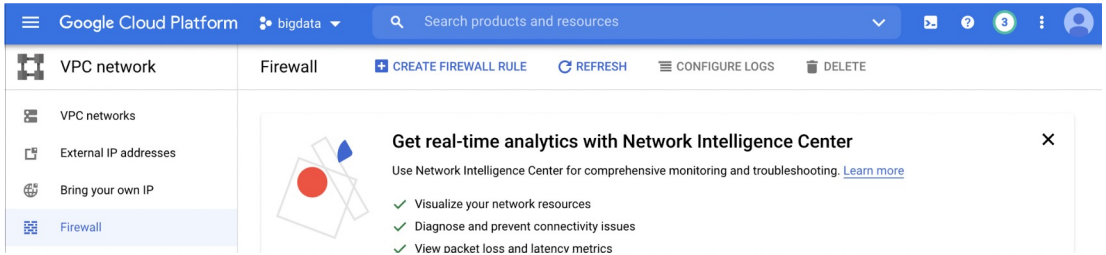
Airflow installation

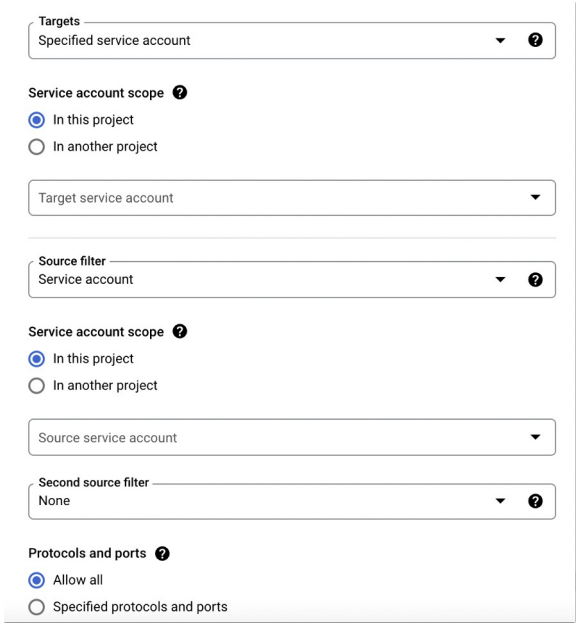
Three choices

1. **Install and use Airflow in the VM of GCP**
2. Install and use airflow in your local machines
3. Google composer

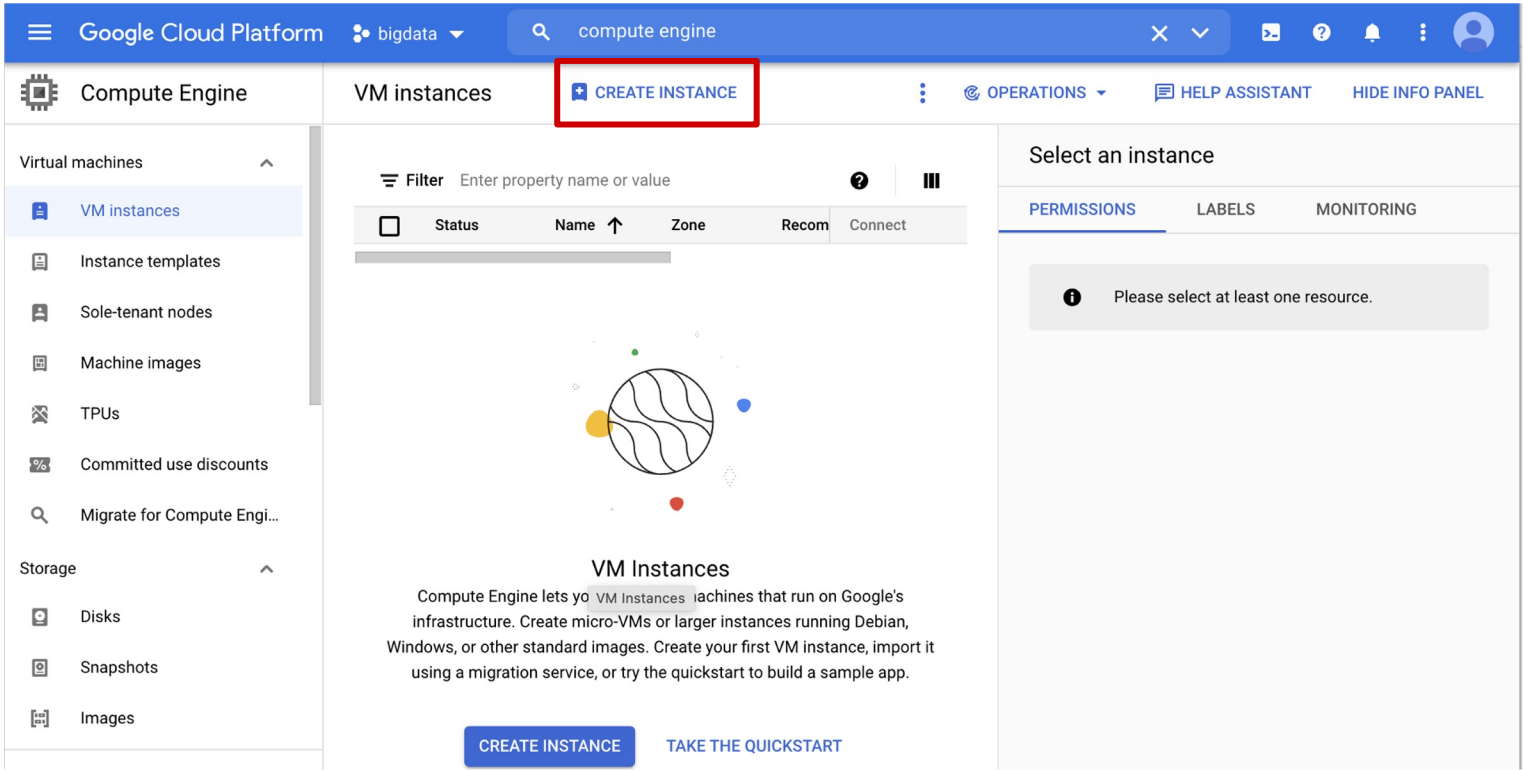
Set up the firewall

- VPC network → Firewall → Create Filewall rule
- Set service account scope and protocols and ports

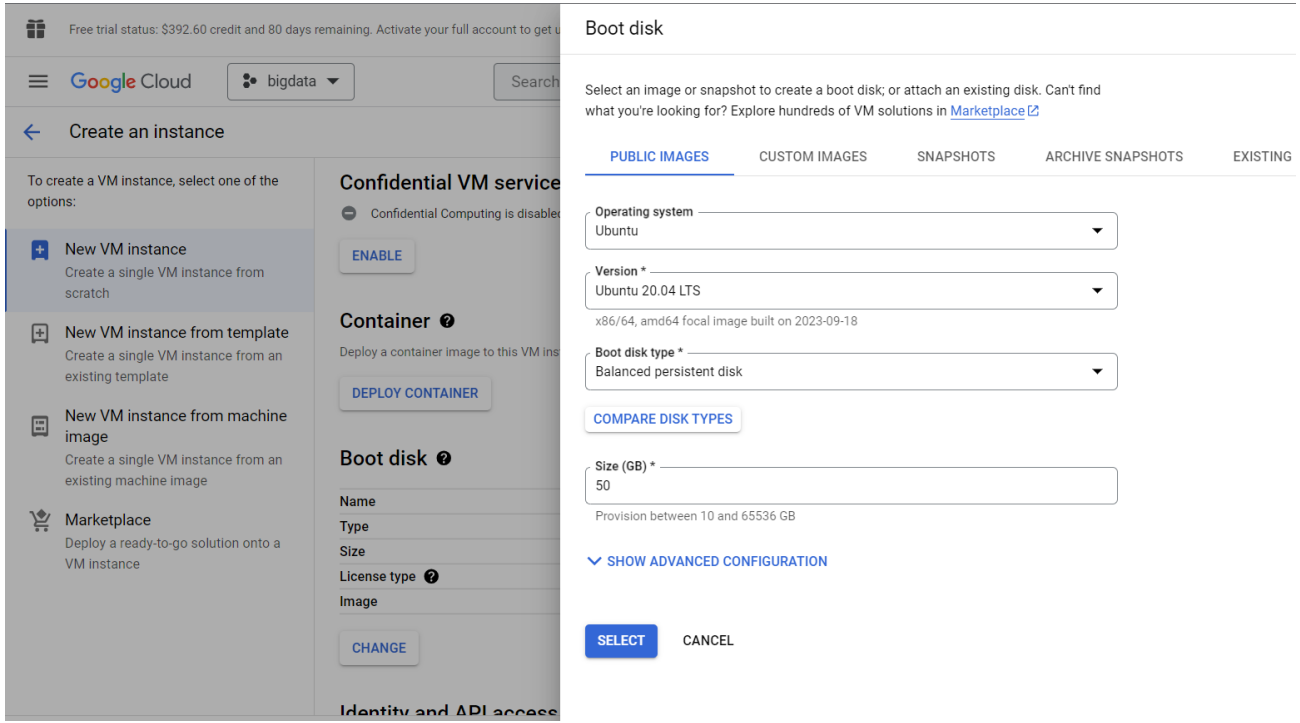




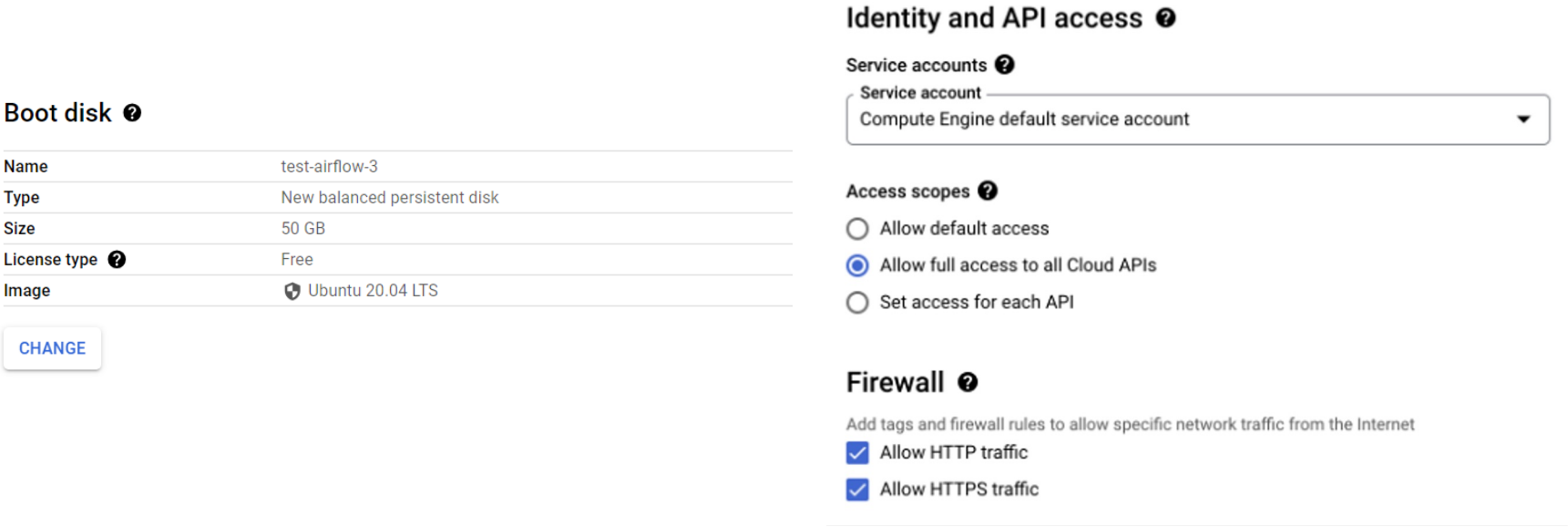
Create a VM instance



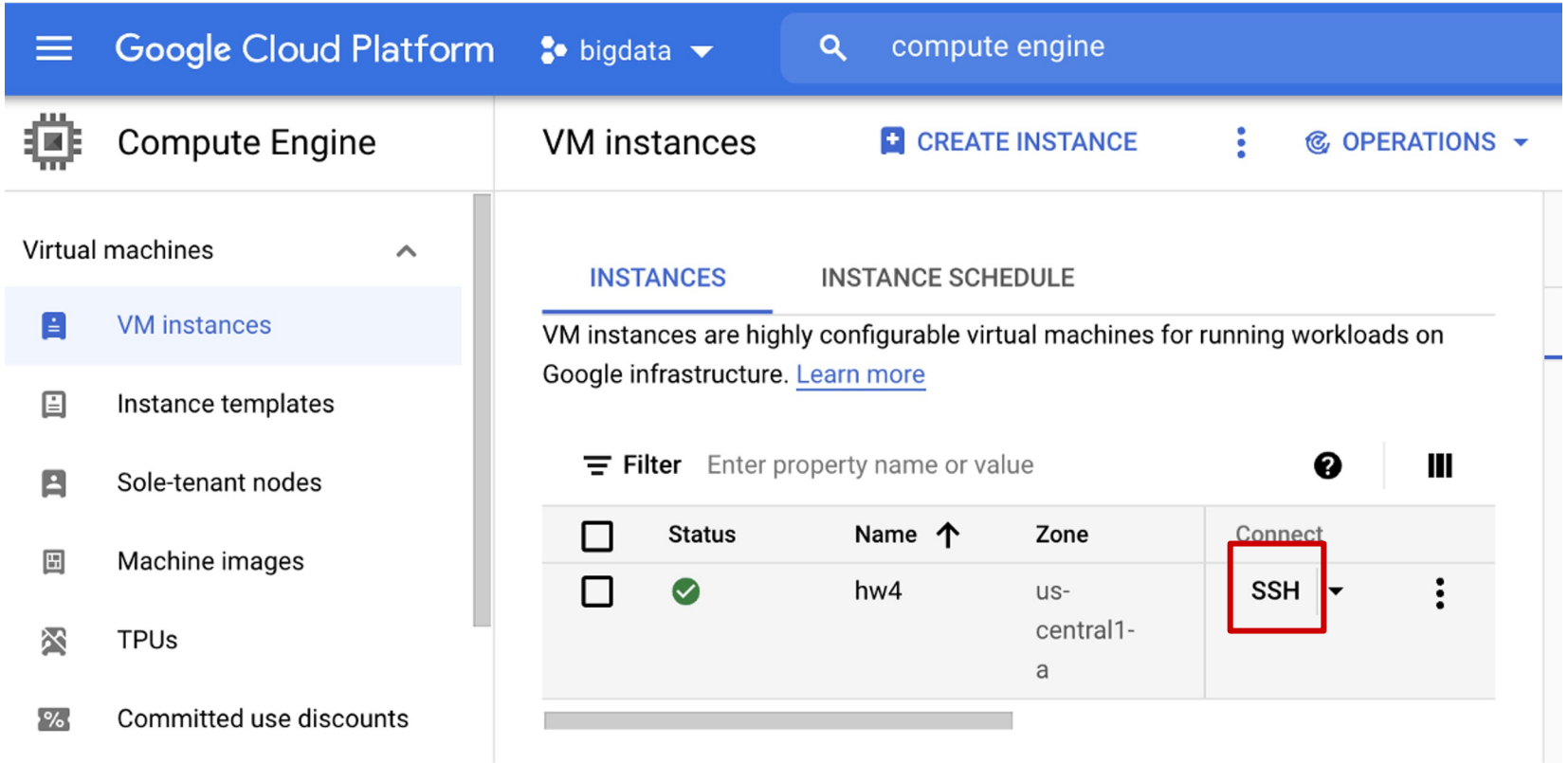
Create a VM instance



Create a VM instance



Connect to your VM using SSH



Connect to your VM using SSH

```
The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

ch3212@hw4:~$
```

Install and update packages

1. sudo apt update
2. sudo apt -y upgrade
3. sudo apt-get install wget
4. sudo apt install -y python3-pip

Download miniconda and create a virtual environment

1. mkdir -p ~/miniconda3
2. wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O ~/miniconda3/miniconda.sh
3. bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3
4. rm -rf ~/miniconda3/miniconda.sh
5. ~/miniconda3/bin/conda init bash
6. ~/miniconda3/bin/conda init zsh

reopen (or we say reconnect) your terminal and create a new environment

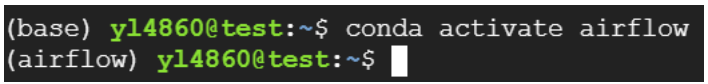
7. conda create --name airflow python=3.8

activate the environment (everytime you open a new terminal, you should run this)

8. conda activate airflow

optional but in case you don't like warnings

9. (optional) pip install virtualenv
10. (optional) pip install kubernetes



Install Airflow

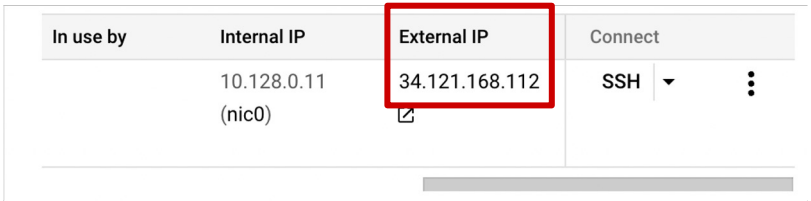
```
# Airflow needs a home. `~/airflow` is the default, but you can put it
# somewhere else if you prefer (optional)
export AIRFLOW_HOME=~/airflow
# Install Airflow using the constraints file
AIRFLOW_VERSION=2.2.1
PYTHON_VERSION=3.8
# For example: 3.8
CONSTRAINT_URL="https://raw.githubusercontent.com/apache/airflow/constraints-${AIRFLOW_VERSION}/constraints-${PYTHON_VERSION}.txt"
# For example: https://raw.githubusercontent.com/apache/airflow/constraints2.2.1/constraints-3.6.txt
pip install "apache-airflow==${AIRFLOW_VERSION}" --constraint "${CONSTRAINT_URL}"
# run airflow version to check if you install it successfully
airflow version
# The Standalone command will initialise the database, make a user,
# and start all components for you.
# airflow standalone
# Visit localhost:8080 in the browser and use the admin account details
# shown on the terminal to login.
# Enable the example_bash_operator dag in the home page
```

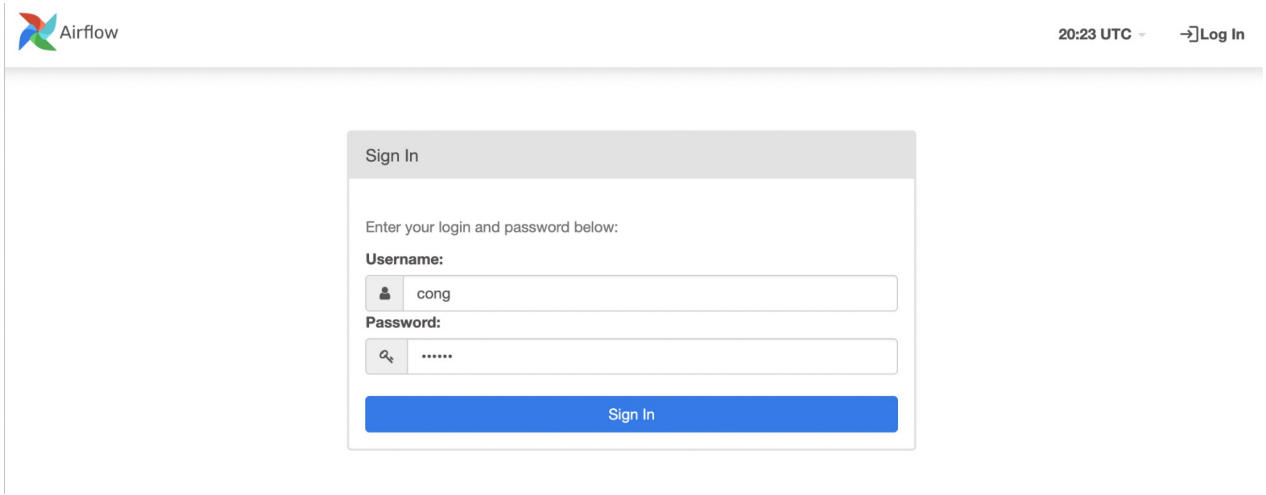
Initialize the database, make a user (enter your own details), and start webserver

```
# Initialize the database, after this you will see a new folder airflow in your
# $AIRFLOW_HOME which contains configuration file airflow.cfg
1.airflow db init
2.airflow users create \
        --username yunhang \
        --password 123456 \
        --firstname yunhang \
        --lastname lin \
        --role Admin \
        --email yl4860@columbia.edu
3.airflow webserver --port 8080
```

Airflow UI on your web browser

- Open your browser
- Visit “**http**://<External IP>:8080”
- login





Start scheduler

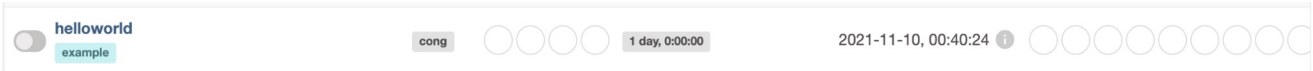
Open a new terminal (you can use screen if you prefer to open only one
terminal)

1. conda activate airflow
2. airflow db init
3. airflow scheduler

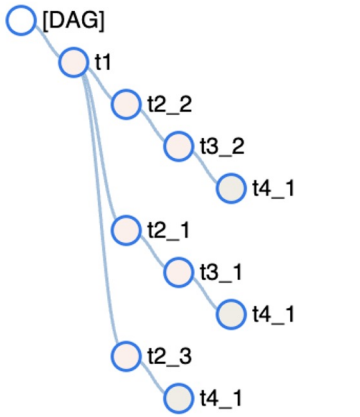
Airflow examples

Helloworld

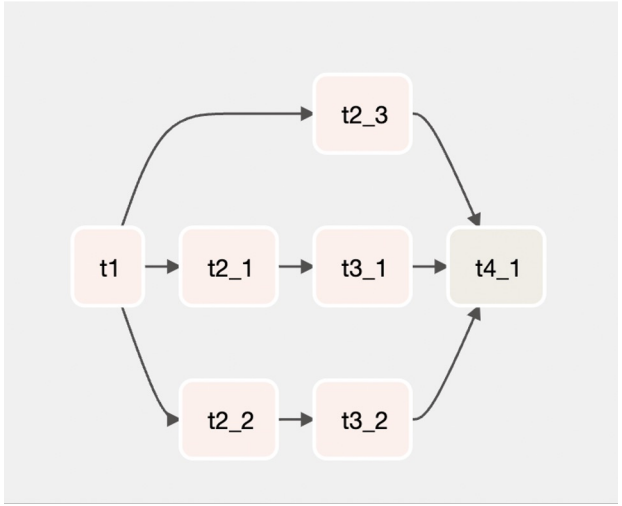
```
# Download helloworld.py from Coursework/Files
# Open a new terminal
conda activate airflow
# Create dags folders
cd airflow
mkdir dags
cd dags
# Upload helloworld.py here
# Check if the script is correct, no errors if it's correct
python helloworld.py
# Initialize db again and you will see "helloworld" on the website after refreshing it
airflow db init
```



Helloworld



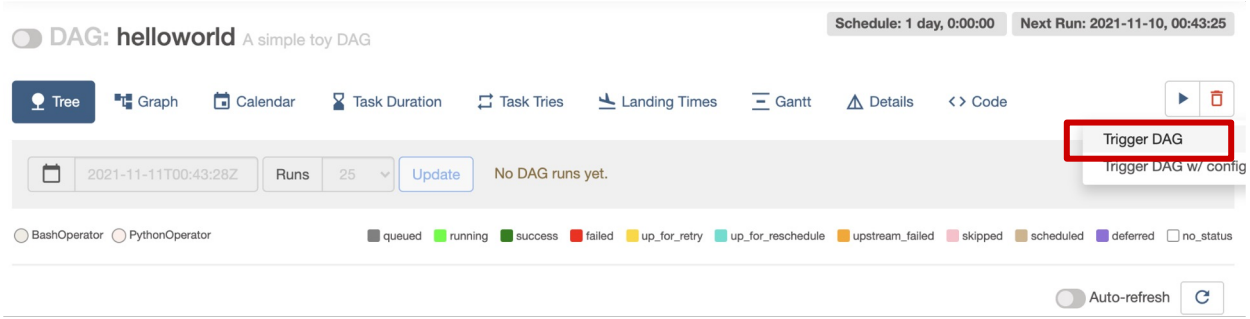
Tree



Graph

Two ways to trigger a DAG

1. Trigger manually
2. Trigger on a schedule



Trigger manually

Scheduler

- The scheduler won't trigger your tasks until the period it covers has ended.
- The scheduler runs your job one schedule_interval after the start date, at the end of the interval.

```python
from datetime import datetime, timedelta
from textwrap import dedent
import time

# The DAG object; we'll need this to instantiate a DAG
from airflow import DAG

# Operators; we need this to operate!
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
# These args will get passed on to each operator
# You can override them on a per-task basis during operator initialization
default_args = {
    'owner': 'cong',
    'depends_on_past': False,
    'email': ['ch3212@columbia.edu'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(seconds=30),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
    # 'wait_for_downstream': False,
    # 'dag': dag,
    # 'sla': timedelta(hours=2),
    # 'execution_timeout': timedelta(seconds=300),
    # 'on_failure_callback': some_function,
    # 'on_success_callback': some_other_function,
    # 'on_retry_callback': another_function,
    # 'sla_miss_callback': yet_another_function,
    # 'trigger_rule': 'all_success'
}
```

References

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

https://airflow.apache.org/docs/apache-airflow/stable/dag-run.html

https://cloud.google.com/composer/docs/triggering-dags

```python
with DAG(
    'helloworld',
    default_args=default_args,
    description='A simple toy DAG',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=['example'],
) as dag:
```

Tasks

```
with DAG(
    'helloworld',
    default_args=default_args,
    description='A simple toy DAG',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=['example'],
) as dag:

    # t* examples of tasks created by instantiating operators
    t1 = PythonOperator(
        task_id='t1',
        python_callable=correct_sleeping_function,
    )

    t2_1 = PythonOperator(
        task_id='t2_1',
        python_callable=correct_sleeping_function,
    )

    t2_2 = PythonOperator(
        task_id='t2_2',
        python_callable=correct_sleeping_function,
        retries=3,
    )

    t2_3 = PythonOperator(
        task_id='t2_3',
        python_callable=correct_sleeping_function,
    )

    t3_1 = PythonOperator(
        task_id='t3_1',
        python_callable=correct_sleeping_function,
    )

    t3_2 = PythonOperator(
        task_id='t3_2',
        python_callable=correct_sleeping_function,
    )

    t4_1 = BashOperator(
        task_id='t4_1',
        bash_command='sleep 2',
        retries=3,
    )
```

```
def correct_sleeping_function():
    """This is a function that will run within the DAG execution"""
    time.sleep(2)
```

Operators

1. **PythonOperator**
2. **BashOperator**
3. branch_operator
4. email_operator
5. mysql_operator
6. DataprocOperator

…

...

PythonOperator:

```
def function():
    print(123)

task = PythonOperator(
    task_id='task_id',
    python_callable=function,
  )
```

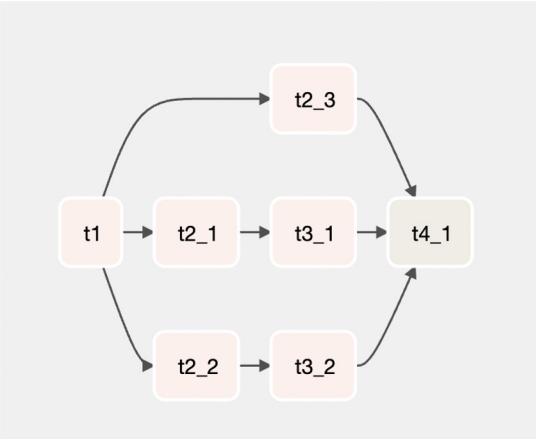
BashOperator:

```
task = BashOperator(
    task_id='task_id',
    bash_command='sleep 2',
  )
# other examples
bash_command='python python_code.py'
bash_command='bash bash_code.sh '
```

(must have a space to satisfy Jinja template !!)

Dependencies

```
# task dependencies

t1 >> [t2_1, t2_2, t2_3]
t2_1 >> t3_1
t2_2 >> t3_2
[t2_3, t3_1, t3_2] >> t4_1
```



t2_1 will depend on t1 running # successfully to run. The following ways # are equivalent:

t1 >> t2_1

t1 << t2_1

t1.set_downstream(t2_1)

t2_1.set_upstream(t1)

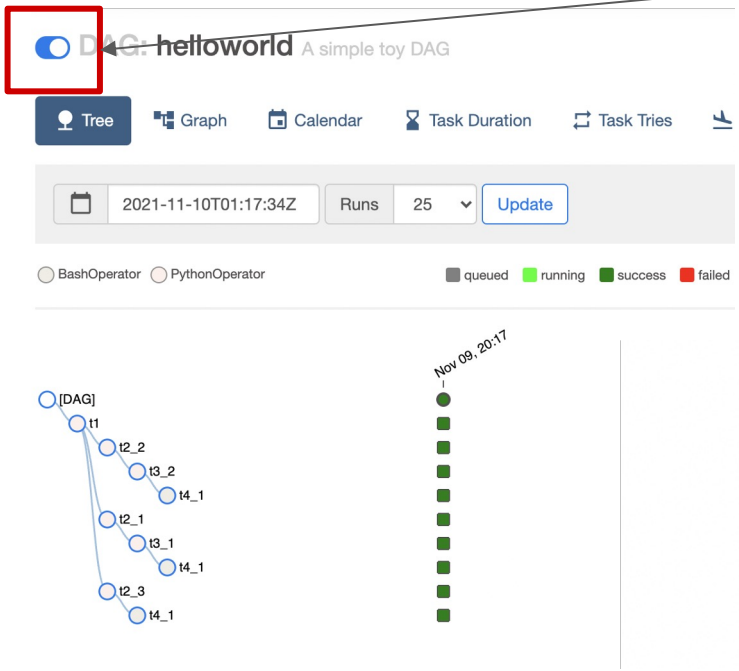
you can write in a chain

t1 >> t2_1 >> t3_1 >> t4_1

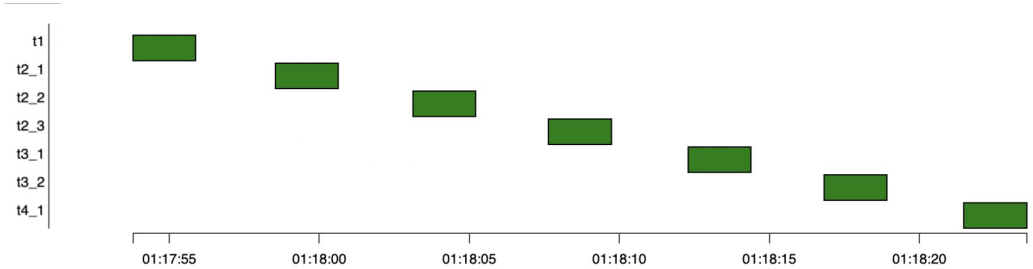
Trigger the dag

```
schedule_interval=timedelta(days=1),
start_date=datetime(2021, 1, 1),
```

Start scheduling the DAG



Tree

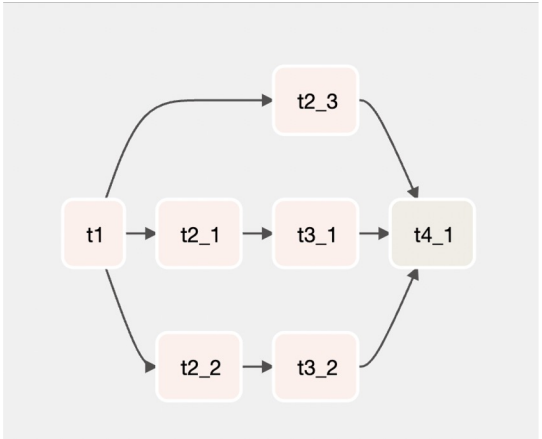


Gantt

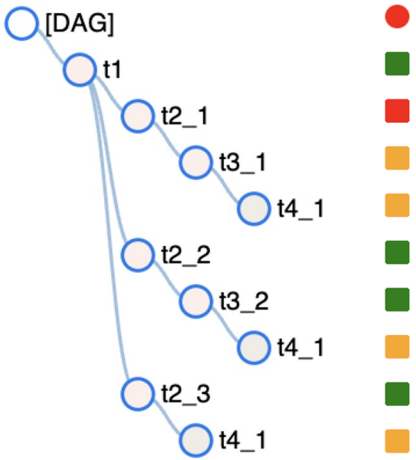
Example 2

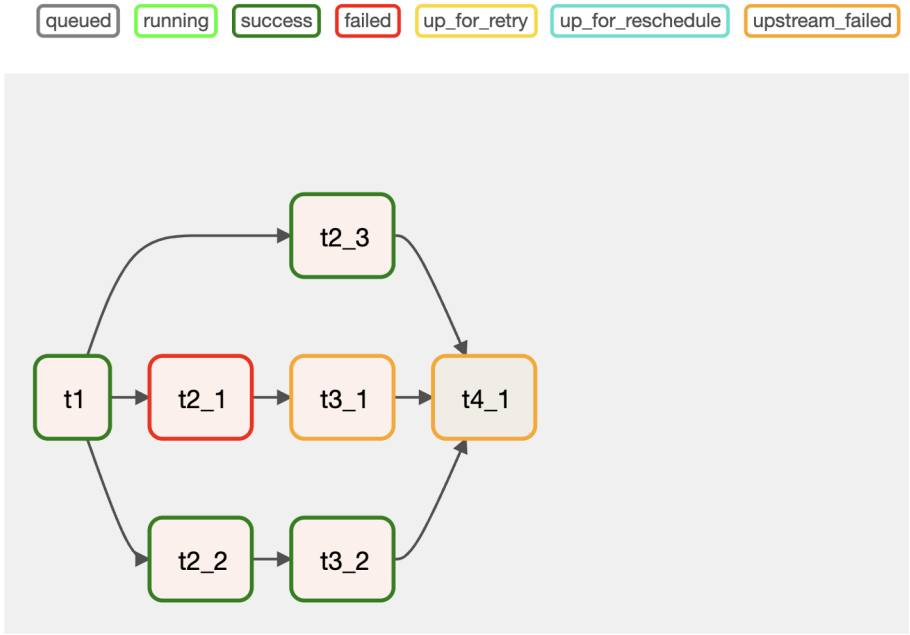
```python
t1 = PythonOperator(
    task_id='t1',
    python_callable=count_function,
)

t2_1 = PythonOperator(
    task_id='t2_1',
    python_callable=wrong_sleeping_function,
    retries=3,
)
```

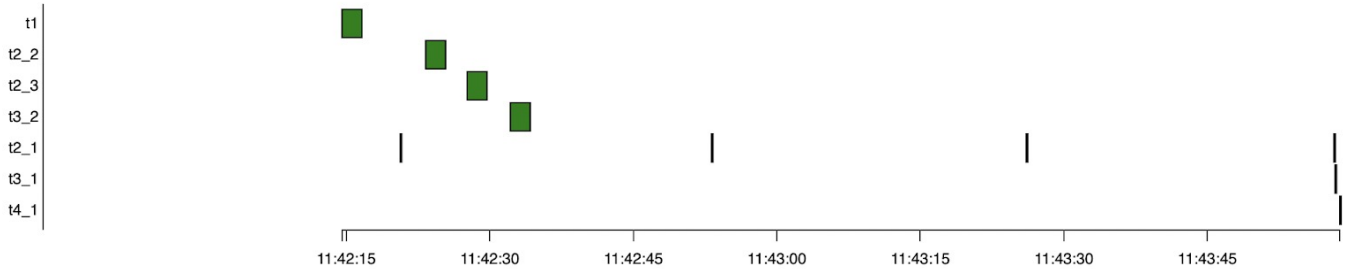
```python
count = 0

def count_function():
    # this task is t1
    global count
    count += 1
    print('count_increase output: {}'.format(count))
    time.sleep(2)

def wrong_sleeping_function():
    # this task is t2_1, t1 >> t2_1
    global count
    print('wrong sleeping function output: {}'.format(count))
    assert count == 1
    time.sleep(2)
```



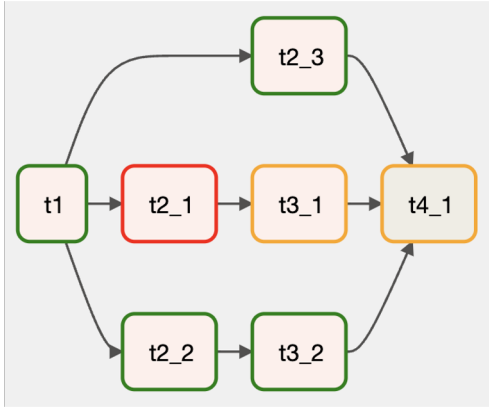
Example 2


[DAG]
t1
t2_1
t3_1
t4_1
t2_2
t3_2
t4_1
t2_3
t4_1


queued
running
success
failed
up_for_retry
up_for_reschedule
upstream_failed
t1
t2_1
t3_1
t4_1
t2_2
t3_2
t2_3


t1
t2_2
t2_3
t3_2
t2_1
t3_1
t4_1
11:42:15
11:42:30
11:42:45
11:43:00
11:43:15
11:43:30
11:43:45

Example 2



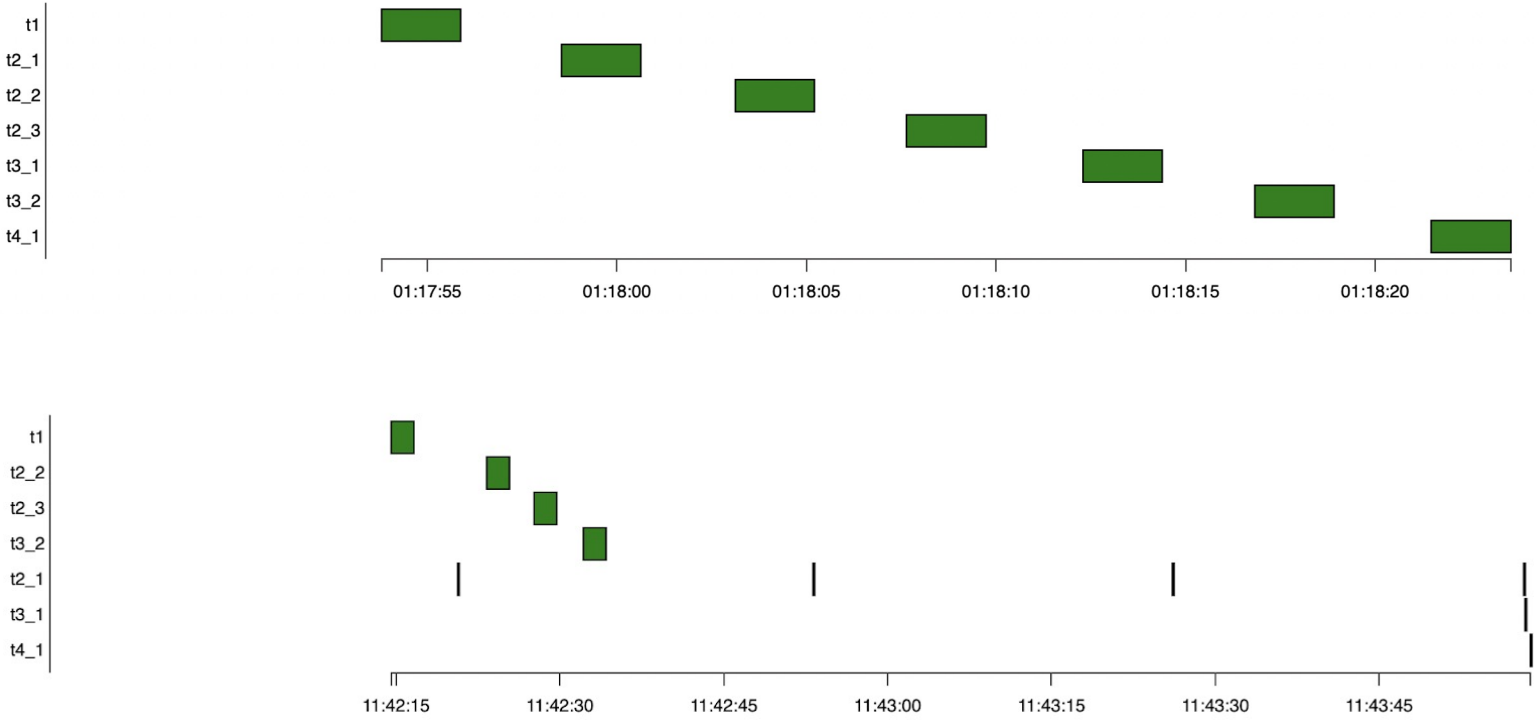

```python
count = 0

def count_function():
    # this task is t1
    global count
    count += 1
    print('count_increase output: {}'.format(count))
    time.sleep(2)

def wrong_sleeping_function():
    # this task is t2_1, t1 >> t2_1
    global count
    print('wrong sleeping function output: {}'.format(count))
    assert count == 1
    time.sleep(2)
```

```
{logging_mixin.py:109} INFO - count_function output: 1

...

{logging_mixin.py:109} INFO - wrong_sleeping_function output: 0

...

    assert count == 1

AssertionError
```

Why?

- Airflow Python script is just a configuration file specifying the DAG's structure as code.
- Different tasks run on different workers at different points in time
- **Script cannot be used to cross communicate between tasks (Xcoms can)**

Why sequential?



Executors

- **SequencialExecutor**
- **LocalExecutor**
- CeleryExecutor
- KubernetesExecutor

Take home

- DAG
- Scheduler
- Executor
- Database
- Operator

- Cross communication between tasks
- Schedule a job !! start data and schedule interval

Some potential error

1. airflow.exceptions.AirflowException: The webserver is already running under PID 20243.

Using the Command line

sudo lsof -i tcp:8080

```
(airflow) yl4860@test:~/airflow/dags$ sudo lsof -i tcp:8080
COMMAND     PID   USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
gunicorn: 28944 yl4860    5u  IPv4 106090      0t0  TCP *:http-alt (LISTEN)
[ready]   28946 yl4860    5u  IPv4 106090      0t0  TCP *:http-alt (LISTEN)
[ready]   28947 yl4860    5u  IPv4 106090      0t0  TCP *:http-alt (LISTEN)
[ready]   28948 yl4860    5u  IPv4 106090      0t0  TCP *:http-alt (LISTEN)
[ready]   28949 yl4860    5u  IPv4 106090      0t0  TCP *:http-alt (LISTEN)
```

Then, Kill all related PID

Sudo kill -9 xxxx

Or,

using the Command line

killall -9 airflow

Homework

Three tasks

- Helloworld
- Build sample workflow
- Design your own workflow for Yahoo Finance data

Task 1 Helloworld

Q1.1 Install Airflow (20 pts)

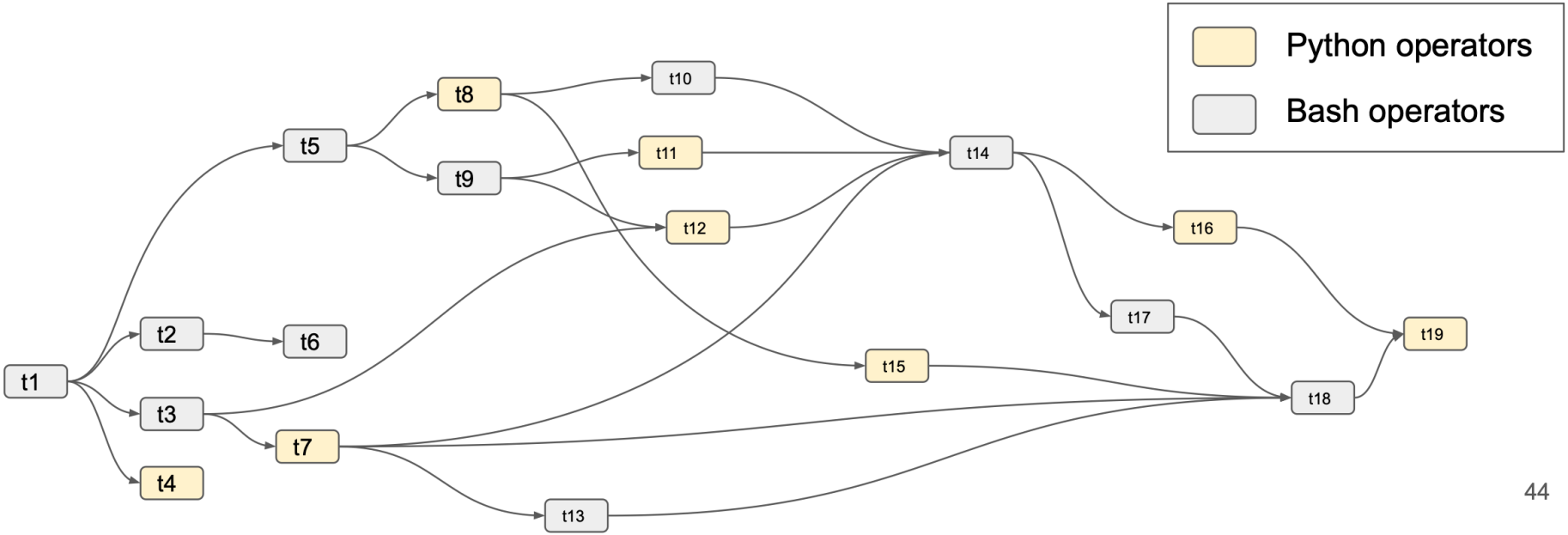
Q1.2 Run helloworld (15 pts)

- SequentialExecutor
- Explore other features and visualizations you can find in the Airflow UI. Choose two features/visualizations to explain their functions and how they help monitor and troubleshoot the pipeline, use helloworld as an example.

Task 2 Build workflows

Q2.1 Implement this DAG (20 pts)

- Tasks and dependencies (5 pts)
- Manually trigger it (10 pts)
- Schedule the first run immediately and then running the program every 30 minutes (5 pts)



Task 3 Build workflows

Q2.2 Stock price fetching, prediction, and storage every day (25 pts)

- Schedule fetching the stock price of [AAPL, GOOGL, FB, MSFT, AMZN].
- Preprocess data if you think necessary
- Train/update 5 linear regression models for stock price prediction for these 5 corporates. Each linear model takes the “open price”, “high price”, “low price”, “close price”, “volume” of the corporate in the current day as the feature and predicts the “high price” for the next day
- Calculate the relative errors for the last 5 days, i.e., (prediction made from yesterday’s data for today - actual price today) / actual price today, and update the prediction date and 5 errors into a table, e.g., a csv file.

```python
import yfinance as yf

ticker = 'AAPL'
aapl = yf.Ticker(ticker)
hist = aapl.history(period='max')
print(type(hist))
print(hist.shape)
print(hist)
```

```
<class 'pandas.core.frame.DataFrame'>
(10317, 7)
                Open        High         Low       Close     Volume  \
Date
1980-12-12    0.100453    0.100890    0.100453    0.100453  469033600
1980-12-15    0.095649    0.095649    0.095213    0.095213  175884800
1980-12-16    0.088661    0.088661    0.088224    0.088224  105728000
1980-12-17    0.090408    0.090845    0.090408    0.090408   86441600
1980-12-18    0.093029    0.093466    0.093029    0.093029   73449600
...                ...         ...         ...         ...        ...
2021-11-04  151.359097  152.207849  150.420465  150.740005   60394600
2021-11-05  151.889999  152.199997  150.059998  151.279999   65414600
2021-11-08  151.410004  151.570007  150.160004  150.440002   55020900
2021-11-09  150.199997  151.429993  150.059998  150.809998   56787900
2021-11-10  150.020004  150.130005  147.850006  147.919998   65076700
```

https://pypi.org/project/yfinance/

Pointers for Q2.2: (You have to think how to create the different tasks and the overall Airflow execution DAG based on the question requirements, below are pointers that may give you some ideas)

- Pull historical data for each corporate till the current date and store data for each in a csv. (Set period in history to "max" yf.Ticker(company_tag).history(period='max'))

- Use this csv to incrementally train the linear regression models for each corporate's data. The dataset for each model X and y should be created such that each X(d) = ["open price", "high price", "low price", "close price", "volume"] of **date d** and the corresponding y = "High price" for **date d+1** as stated in the question.

- Assume current date (the date till which you have pulled the data) is **d_current.**

  For each corporate,

  - Train/Update a model using data from [X,y] till (d_current – i days) and use this model to predict the y for the (d_current – (i-1) day). Repeat the training and prediction for i = 5,4,3,2,1 (make predictions for last 5 days and calculate relative errors for each prediction)
  - For each value of i keep storing the relative errors in a csv for each model for each corporate on each of the 5 testing days.
  - The final errors csv should look something like below:
    If the latest date till which you pulled data was 11/30/2022 then your final errors csv should look like:

	A	B	C	D	E	F	G
1		Date	APPLE	GOOGLE	META	MICROSOF	AMAZON
2	0	11/26/2022	-0.00501	-0.00083	-0.001	-0.00167	0.005777
3	1	11/27/2022	-0.00091	-0.00755	-0.00097	-0.00698	-0.00609
4	2	11/28/2022	0.026959	0.007073	0.005526	0.003579	0.008291
5	3	11/29/2022	0.019557	0.0088	0.008127	0.011455	-0.0204
6	4	11/30/2022	0.009074	0.010989	-0.00189	0.009092	0.014896

To give an example: the value in cell C2 represents the relative error produced by the linear regression model trained on Apple data till 11/25/2022 and the prediction made for 11/26/2022. Similarly, the value in cell C3 represents the relative error produced by the updated linear regression model trained on Apple data till 11/26/2022 and the prediction made for 11/27/2022. The value in cell F6 represents the relative error produced by the linear regression model trained on Microsoft data till 11/29/2022 and the prediction made for 11/30/2022.

References

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

https://cloud.google.com/composer/docs