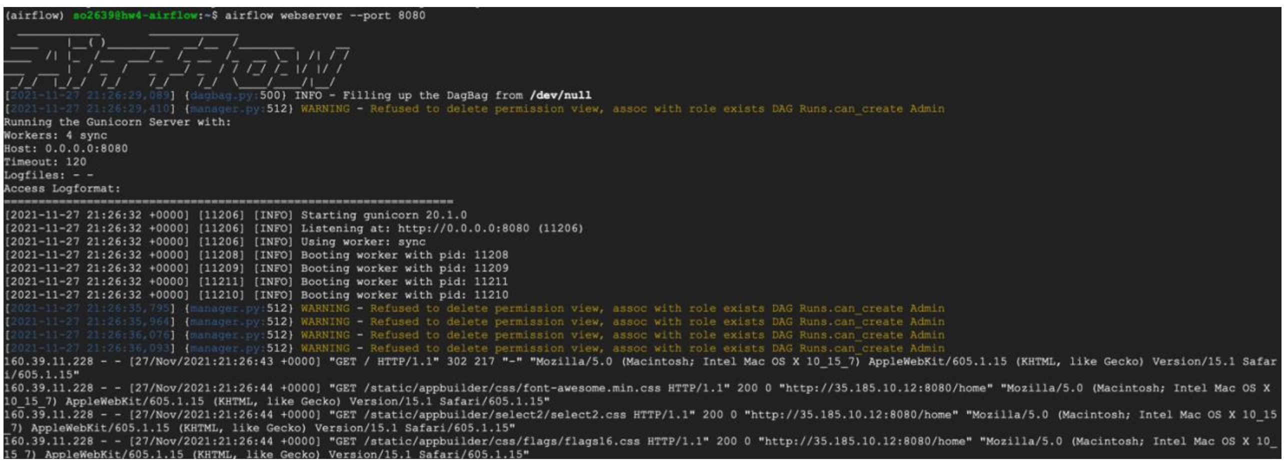
EECS E6893 Big Data Analytics
Homework Assignment 4

Submitted by:
Shivam Ojha
UNI: so2639

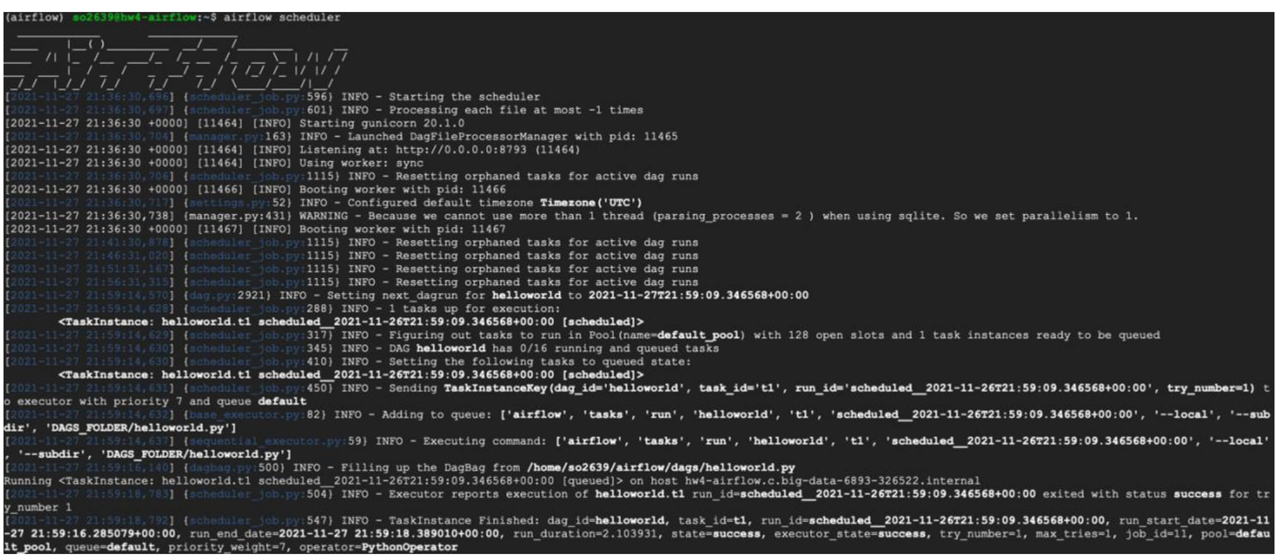
Task 1: (Helloworld)

1.1 Please find below the screenshots for the installation of Airflow on a virtual machine on Google Cloud Platform.

1. Terminal after successfully starting the webserver

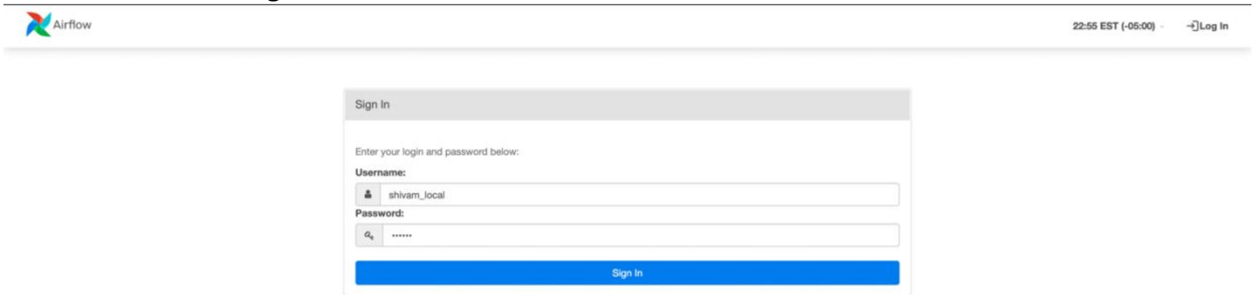


Terminal after successfully starting the scheduler

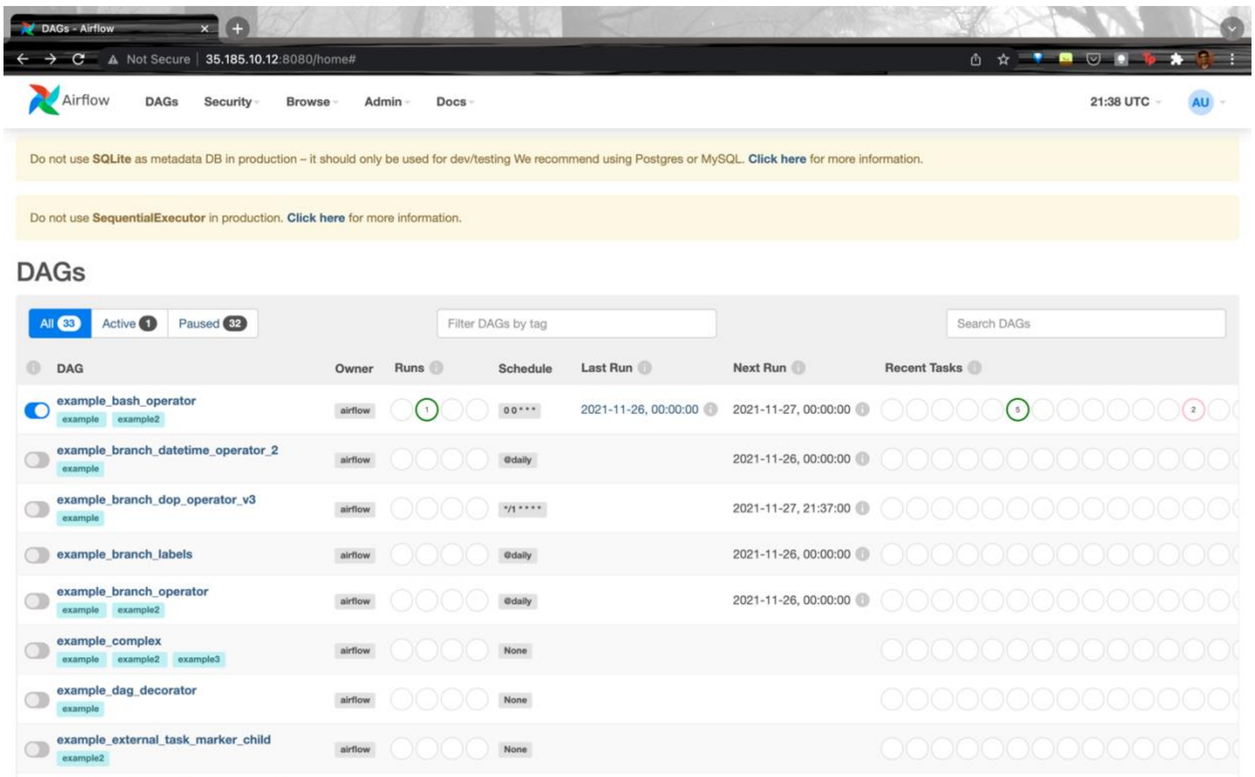


2. Screenshots of the web browser:

a. Before login:



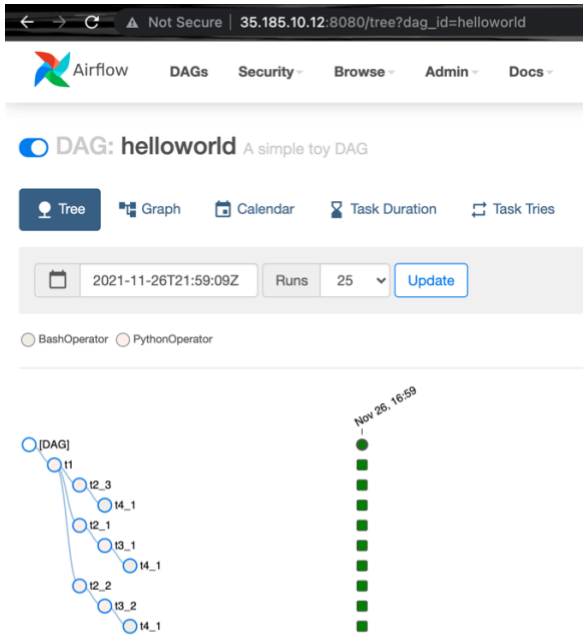
b. Showing the DAGs after successful login:



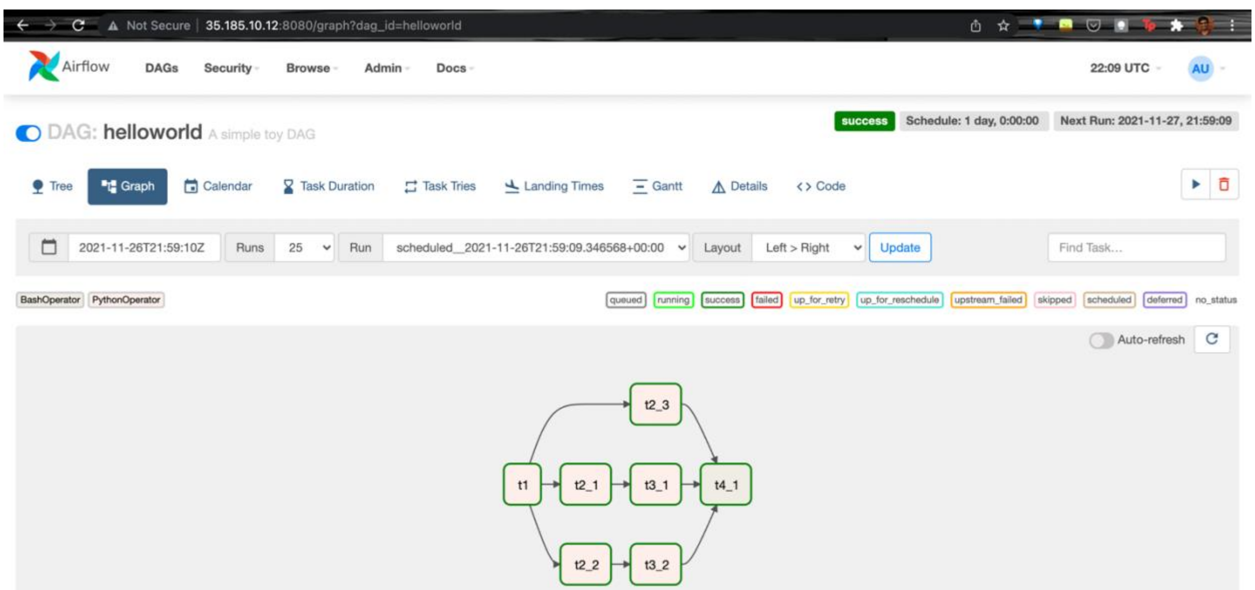
1.2 Helloworld with SequentialExecutor and LocalExecutor.

(1) <u>SequentialExecutor:</u>

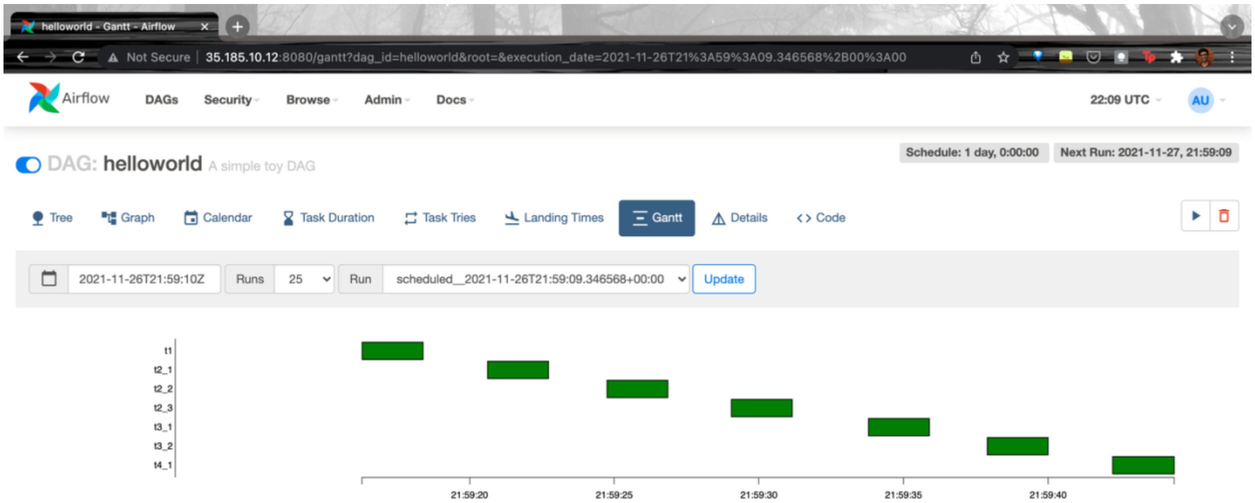
Screenshot of the tree for helloworld for Sequential Executor:



Screenshot of the graph for helloworld for Sequential Executor:



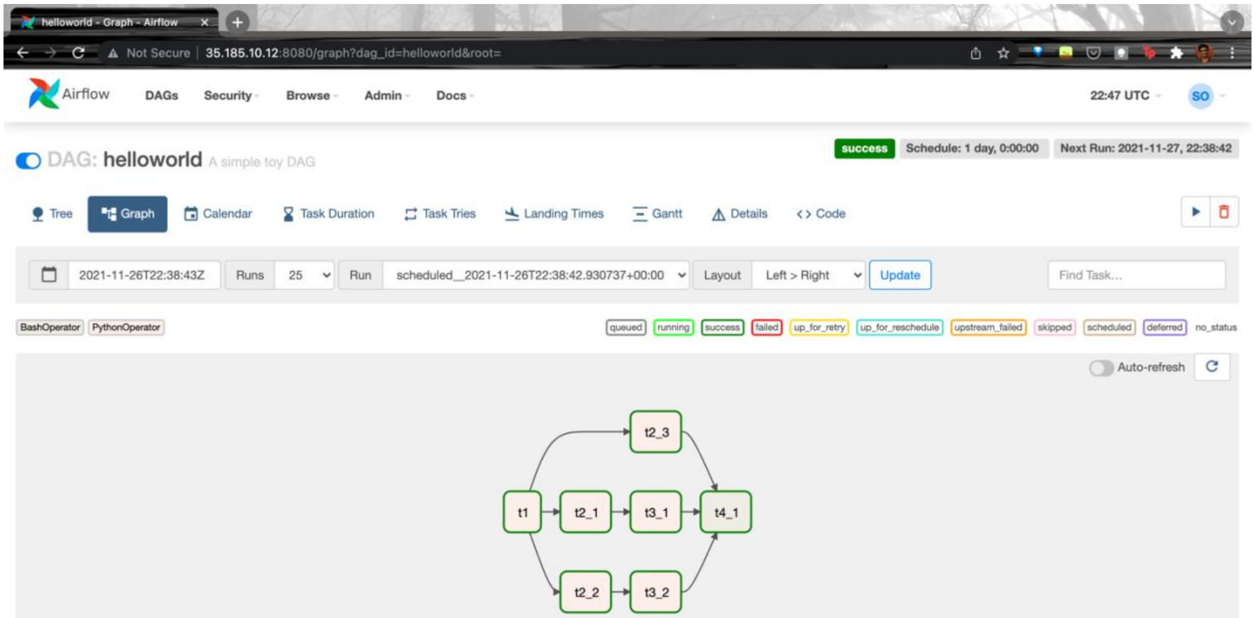
Screenshot of the gantt for helloworld for Sequential Executor:



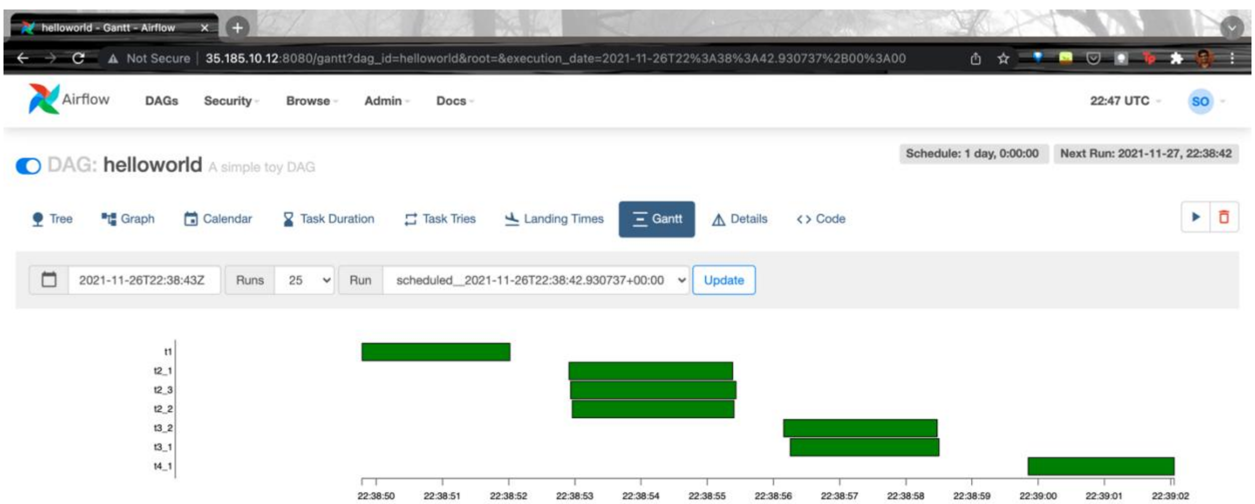
LocalExecutor:

Screenshot of the tree for helloworld for Local Executor:

Screenshot of the graph for helloworld for Local Executor:



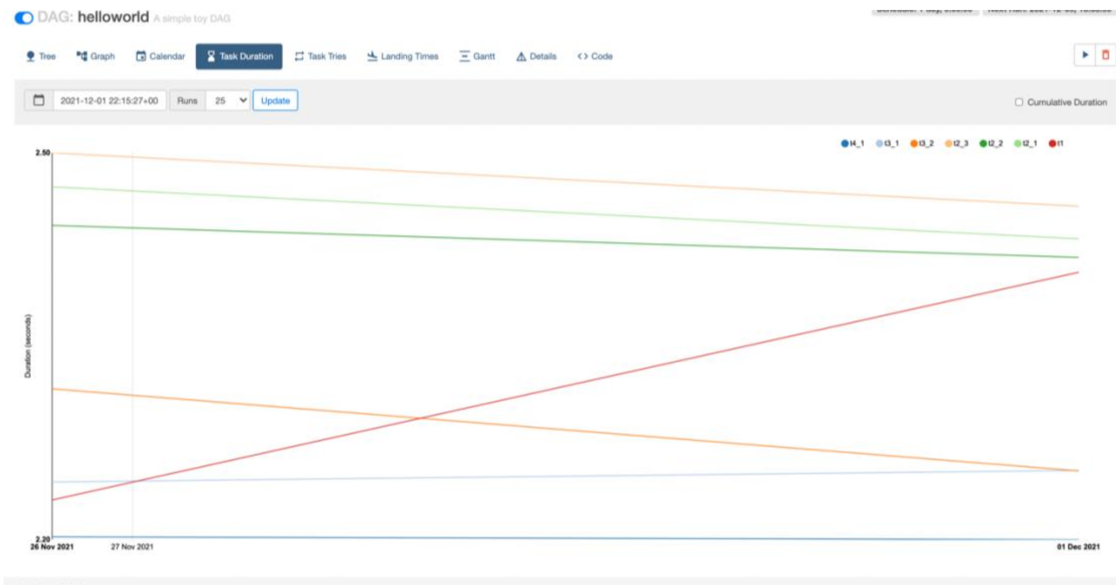
Screenshot of the gantt for helloworld for Local Executor:



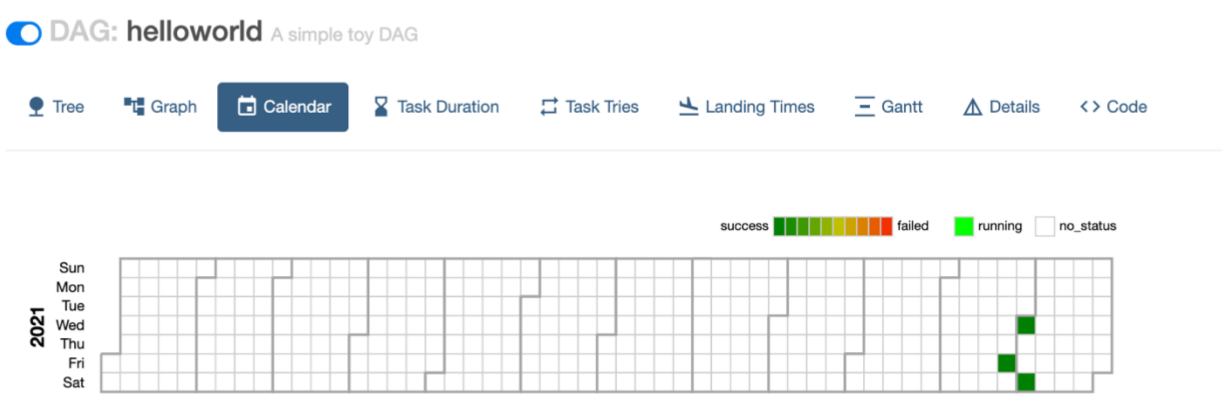
(2) Two other visualizations/features in the Airflow UI are:

1. <u>Task Duration:</u> It's a visualization that shows how much time a task took to execute over the past N runs. It's a useful feature for monitoring the task execution time as it lets us find outliers and quickly understand where the time is spent in your DAG over several runs.

Screenshot of the task duration of helloworld over multiple runs is shown below:



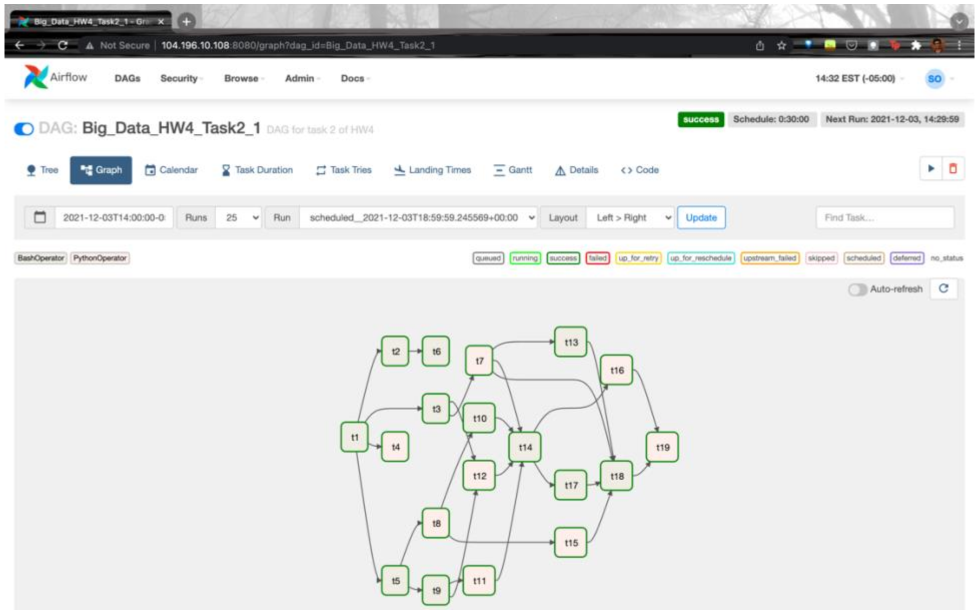
2. Calendar: The calendar view gives us an overview of the entire history of the DAG's over months or years. It helps us in the troubleshooting process as we can we can understand the trends of the overall success/failure rate of runs over time from the visualisation. In the screenshot below, we can see that the helloworld DAG has tried executing three times in the past 10 days and has successfully executed all three times.
Note: The daily schedule isn't depicted here as the DAG is deployed on a virtual machine, and the virtual machine was turned off for the rest of the days.



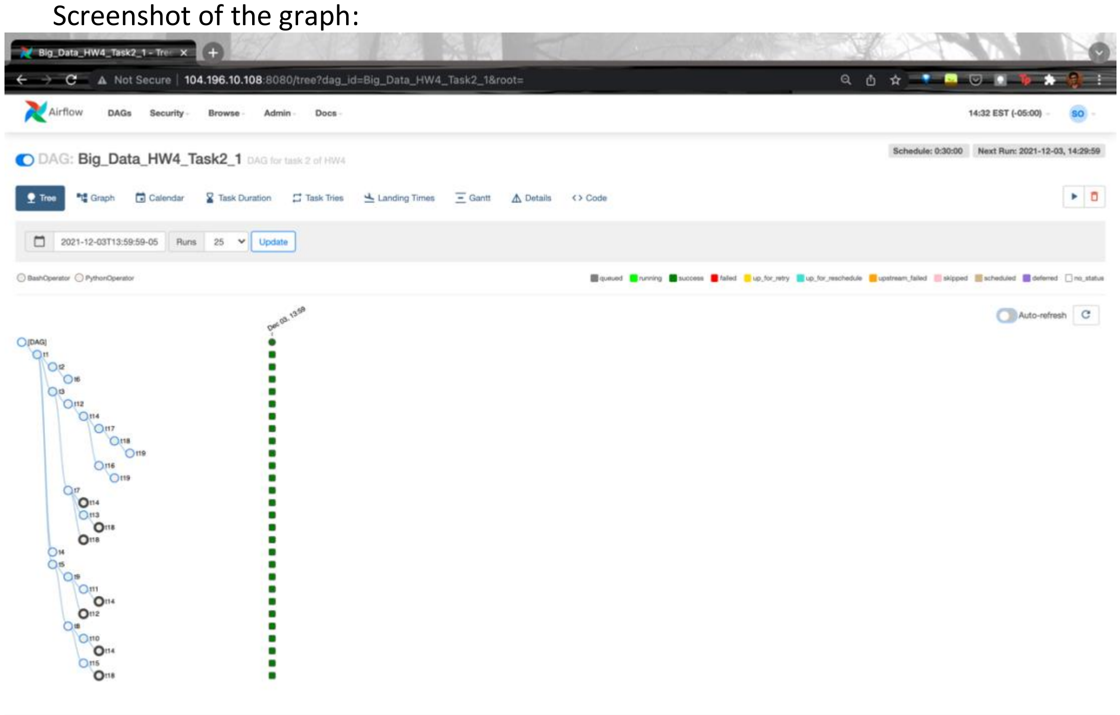
Task 2: (Build Workflows)

2.1 The DAG is implemented with a Local Executor.

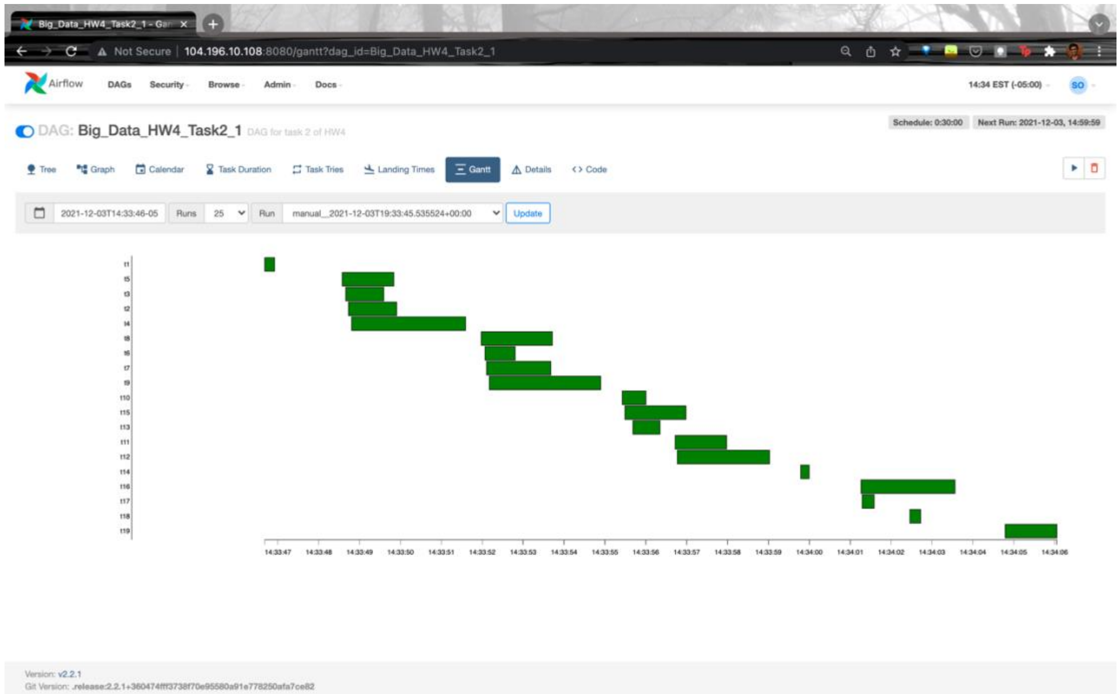
1. Screenshot of the tree:



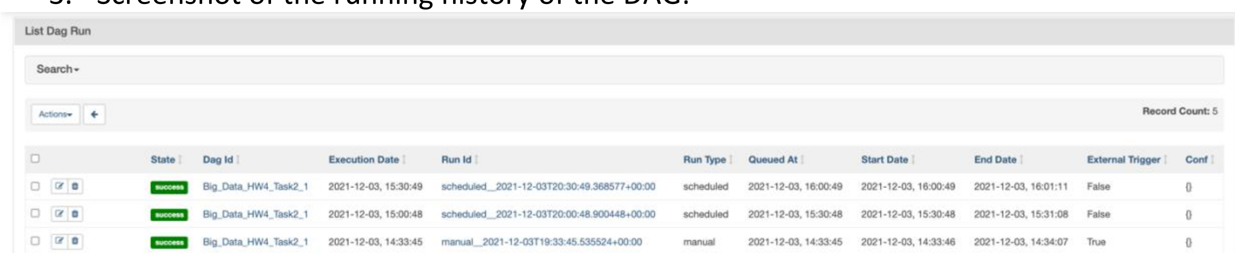
Screenshot of the graph:



2. Screenshot pf the gantt for the manual trigger:



3. Screenshot of the running history of the DAG:



List Dag Run

Search

Actions

Record Count: 5

	State	Dag Id	Execution Date	Run Id	Run Type	Queued At	Start Date	End Date	External Trigger	Conf
	success	Big_Data_HW4_Task2_1	2021-12-03, 15:30:49	scheduled__2021-12-03T20:30:49.368577+00:00	scheduled	2021-12-03, 16:00:49	2021-12-03, 16:00:49	2021-12-03, 16:01:11	False	{}
	success	Big_Data_HW4_Task2_1	2021-12-03, 15:00:48	scheduled__2021-12-03T20:00:48.900448+00:00	scheduled	2021-12-03, 15:30:48	2021-12-03, 15:30:48	2021-12-03, 15:31:08	False	{}
	success	Big_Data_HW4_Task2_1	2021-12-03, 14:33:45	manual__2021-12-03T19:33:45.535524+00:00	manual	2021-12-03, 14:33:45	2021-12-03, 14:33:46	2021-12-03, 14:34:07	True	{}

As you can see above, the DAG is manually triggered at 2:33 pm EST on 3rd December. Then the next two runs are the scheduled runs, which happen every 30 minutes.
The date at which the DAG starts being scheduled is called the <u>start date</u>. <u>Schedule interval</u> refers to the interval of time at which your DAG gets triggered. In the code, I have taken the start date as 2021-12-03 3:30 pm and the schedule has been set to minutes = 30, using timedelta functionality of datetime package in python. Another option to schedule would have been to use a cron job. This 30 minutes schedule interval ensures that the scheduler in the airflow triggers (manually/scheduled) the task at every 30 minutes after the set start date.

<u>Code for Task 2.1:</u>

Task2_1.py > ...

```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 initializatio
default_args = {
    'owner': 'Shivam',
    'depends_on_past': False,
    'email': ['so2639@columbia.edu'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=30)
}


count = 0

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

def count_function():
    global count
    count += 1
    print('count_increase output: {}'.format(count))
    time.sleep(1)

def print_function():
    print('This task is a python operator')
    time.sleep(1)

```

Task2_1.py > ...

```python

with DAG(
    'Big_Data_HW4_Task2_1',
    default_args=default_args,
    description='DAG for task 2 of HW4',
    schedule_interval=timedelta(minutes=30),
    start_date=datetime(2021, 12, 3, 3, 30, 00),
    catchup=False,
    tags=['Task2'],
) as dag:

    # t* examples of tasks created by instantiating operators

    t1 = BashOperator(
        task_id='t1',
        bash_command= 'echo "This task is part of Question 2"',
    )

    t2 = BashOperator(
        task_id='t2',
        bash_command= 'python3 /home/so2639/airflow/dags/py_script.py',
        retries=2,
    )

    t3 = BashOperator(
        task_id='t3',
        bash_command= 'date',
        retries=3,
    )

    t4 = PythonOperator(
        task_id='t4',
        python_callable=sleeping_function,
    )

    t5 = BashOperator(
        task_id='t5',
        bash_command= 'python3 /home/so2639/airflow/dags/py_script.py',
    )

    t6 = BashOperator(
```

Task2_1.py > ...

```python

    t6 = BashOperator(
        task_id='t6',
        bash_command= 'echo "This task is part of Question 2"',
    )

    t7 = PythonOperator(
        task_id='t7',
        python_callable=print_function,
        retries=2,
    )

    t8 = PythonOperator(
        task_id='t8',
        python_callable=count_function,
        retries=2,
    )

    t9 = BashOperator(
        task_id='t9',
        bash_command='sleep 2',
    )

    t10 = BashOperator(
        task_id='t10',
        bash_command= 'date',
    )

    t11 = PythonOperator(
        task_id='t11',
        python_callable=print_function,
        retries=2,
    )

    t12 = PythonOperator(
        task_id='t12',
        python_callable=sleeping_function,
        retries=2,
    )

    t13 = BashOperator(
```

Task2_1.py > ...

```python
    t13 = BashOperator(
        task_id='t13',
        bash_command= 'date',
    )

    t14 = BashOperator(
        task_id='t14',
        bash_command= 'echo "This task is part of Question 2"',
    )

    t15 = PythonOperator(
        task_id='t15',
        python_callable=count_function,
        retries=2,
    )

    t16 = PythonOperator(
        task_id='t16',
        python_callable=sleeping_function,
        retries=2,
    )

    t17 = BashOperator(
        task_id='t17',
        bash_command= 'echo "This task 17 is part of Question 2"',
    )

    t18 = BashOperator(
        task_id='t18',
        bash_command= 'python3 /home/so2639/airflow/dags/py_script.py',
    )

    t19 = PythonOperator(
        task_id='t19',
        python_callable=print_function,
        retries=1,
    )


    # task dependencies
    t1 >> [t2, t3, t4, t5]
```

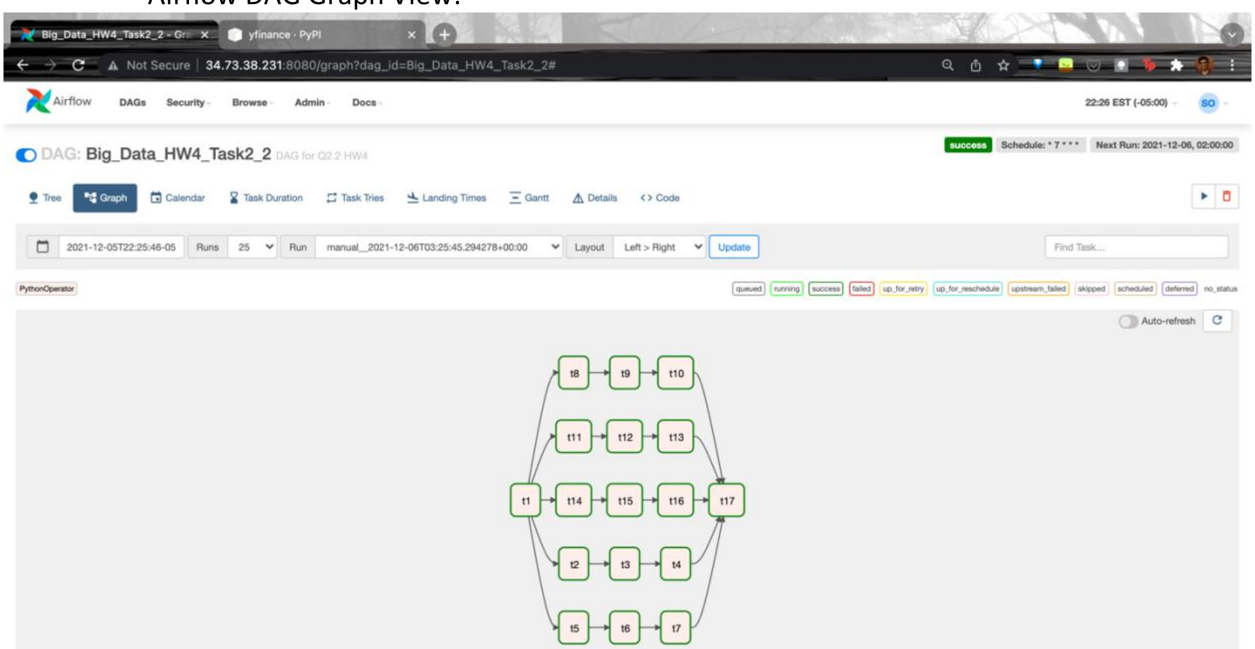
```python
Task2_1.py > ...
    )

    t19 = PythonOperator(
        task_id='t19',
        python_callable=print_function,
        retries=1,
    )


    # task dependencies
    t1 >> [t2, t3, t4, t5]
    t2 >> t6
    t3 >> [t7, t12]
    t5 >> [t8, t9]
    t7 >> [t13, t14, t18]
    t8 >> [t10, t15]
    t9 >> [t11, t12]
    [t10, t11, t12] >> t14
    [t13, t15, t17] >> t18
    t14 >> [t16, t17]
    [t16, t18] >> t19
```

2.2 Stock price fetching, prediction and storage:

Airflow DAG Graph View:



1. I have used the yfinance python package to fetch the data and created a DAG on airflow and set a cron job '* 7 * * *' to schedule it to run at 7:00 AM every day (shown above).

2. The data is pre-processed by dropping the nan values, if any, with the help of dropna method in pandas, so that the model training later on isn't affected.

3. As per the question, 5 linear regression models are trained(each for one company) with the the features as ‘high price’, ‘low price’, ‘close price’, open price’, ‘volume’] and it predicts the ‘high price’ for the next day.

4. An if condition checks if the values received are same for the last 2 days(They can be same in case the market was closed). If they are different, the relative error for each stock is predicted and stored in a output csv. A final compiled csv is created at the end that shows all the error values for all the companies for each day the DAG has been running. The relative error is calculated from 30th November to 5th December for each company, shown in the csv table screenshot below:

	Date	AAPL	GOOGL	FB	MSFT	AMZN
0	11/30/21	0.01833223	0.0178464	0.02154729	0.006656	-0.0011046
0	12/1/21	-0.018877	-0.0186309	-0.0090627	-0.0187762	-0.0092226
0	12/2/21	-0.0229995	0.00372933	0.02005463	0.00508727	0.00354409
0	12/3/21	-0.0196708	-0.0153219	-0.0053576	-0.0123388	-0.0057598
0	12/4/21	0.01860972	-0.0068104	0.00302498	0.00552341	-0.0008707
0	12/5/21	0.01860972	-0.0068104	0.00302498	0.00552342	-0.0008707

5. The workflow is setup using a DAG. Each functionality of the code is defined as a function, and different tasks in the DAG call these functions one by one. The stock data for the five companies is fetched and processed in parallel (Task 2, 5, 8, 11 and 14 start in parallel, and the subsequent pre-processing and prediction tasks for each company also happen in parallel accordingly). Once the model is compiled and the predictions are ready, the final output csv is generated with the relative errors for all the companies(Task 17). The task dependencies are mentioned below:
   t1 >> [t2, t5, t8, t11, t14]
   t2 >> t3
   t3 >> t4
   t5 >> t6
   t6 >> t7
   t8 >> t9
   t9 >> t10
   t11 >> t12
   t12 >> t13
   t14 >> t15
   t15 >> t16
   [t4, t7, t10, t13, t16] >> t17

For the Linear regression model training, the training data for the model is taken till two days before the day for which we have to predict the stock price. For example, consider we fetch data on 30th November 7 AM, hence we will fetch the stock prices till 29th November, when the market closed. So as per the question, we will use the data till 28th November to train the model, and as we have to make predictions for the high price for the next day (29th November), we will push the index of predictions by 1. Hence, we will be able to calculate the relative error for the last record i.e. 29th November.

The csv and json files are used for <u>cross task communication.</u> For each company the data is fetched, pre-processed and stored as a csv initially. This data is used to calculate the relative error in a later task, and the error is stored in a dict initially. Once the dict has accumulated the error values for all the companies, then the last task(t17) calls the generate_output_csv function, which stores the error values as a dataframe for the given date. Then all the date error values are compiled to create a single cdv file 'compiled_error_results.csv'. The <u>scheduler</u> is setup with the help of cron job, so that the DAG executes every day at 7 am and fetches the stock data till previous day (As at 7 am, the stock price for the current day will not be available).

<u>Screenshots for code for Task 2.2:</u>

so2639_6893_hw4_Task2_2.py 6 ×

so2639_6893_hw4_Task2_2.py > preprocess_data

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import os
import json
import yfinance as yf
import numpy as np
import pandas as pd
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.python import PythonOperator
from sklearn.linear_model import LinearRegression

default_args = {
    'owner': 'Shivam',
    'depends_on_past': False,
    'email': ['so2639@columbia.edu'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(seconds=30)
}

data_dir = '/home/so2639/airflow/dags/task2_2/'
json_file = 'errors_dict.json'
date = datetime.now().strftime("%Y-%m-%d")
#strt_date = datetime.strptime(date, "%Y-%m-%d") - timedelta(weeks=27)

def fetch_data(ticker):
  input_company_data = yf.Ticker(ticker)
  # Fetching data for the last 6 months
  # df = input_company_data.history(start=strt_date, end=date)
  df = input_company_data.history(period = '6mo')
  df.to_csv(data_dir + ticker + '.csv')

def preprocess_data(ticker):
  df = pd.read_csv(data_dir + ticker + '.csv')
  df['Date'] = pd.to_datetime(df['Date'])
  # The data doesn't necessarily need to be normalised
  # Cleaning the data by dropping null values and any date values greater than today
  df = df[df.Date <= pd.to_datetime(date)].reset_index(drop = True)
  df = df.dropna()
  df.to_csv(data_dir + ticker + '.csv')
```

so2639_6893_hw4_Task2_2.py 6 ×

so2639_6893_hw4_Task2_2.py > preprocess_data

```python
  df.to_csv(data_dir + ticker + '.csv')

def init_df_dict():
  errors_dict = {'Date': '', 'AAPL': '', 'GOOGL': '', 'FB': '', 'MSFT': '', 'AMZN': ''}
  errors_dict['Date'] = [date]

  with open(data_dir + json_file, 'w') as jsonfile:
    json.dump(errors_dict, jsonfile)

def predict_and_compute_error(ticker):
  df = pd.read_csv(data_dir + ticker + '.csv')
  df = df.iloc[:-1, :]

  # Generate training data
  X = np.array(df[['Open', 'High', 'Low', 'Close', 'Volume']])
  y = np.array(df['High'])

  # When we fetch data till 30th November morning, we get stock values till market close on 29th.
  # So using data till 28th November to train. Pushing the labels value ahead by 1 index as well.
  # error = (prediction yesterday - actual price today)/actual price today
  # example: error = (prediction on 28th for 29th - price on 29th)/ price on 29th
  X_train = X[:-2, :]
  y_train = y[1:-1]

  # Run Linear Regression Model
  lr_model = LinearRegression().fit(X_train, y_train)
  print("Score: " + str(lr_model.score(X_train, y_train)))

  # Make predictions
  X_test = X[-2:-1, :] # Last value in dataframe
  y_test = y[-1] # Last High Value

  preds = lr_model.predict(np.array(X_test))
  preds = preds[0]

  # Read error json
  with open(data_dir + json_file) as j:
    errors = json.load(j)

```

so2639_6893_hw4_Task2_2.py 6 ×

so2639_6893_hw4_Task2_2.py > preprocess_data

```python
  # Compute error
  if not(np.array_equal(X[-2:-1, :], X[-3:-2, :], equal_nan=False)):
      er = (preds - y_test) / y_test
      errors[ticker] = [er]

      # Update error json
      with open(data_dir + json_file, 'w') as j:
        json.dump(errors, j)

def generate_output_csv():
  # Read error json
  with open(data_dir + json_file) as j:
    errors = json.load(j)

  df = pd.DataFrame(errors)
  df.to_csv(data_dir + 'csv_files/' + 'results_' + date + '.csv')

  csv_list = []
  for file in os.listdir(data_dir+'csv_files/'):
    if (file.startswith('results_') and file.endswith('.csv')):
      csv_list.append(os.path.join(data_dir + 'csv_files/', file))

  # Compute compiled csv output
  out = pd.DataFrame(columns=['Date','AAPL','GOOGL','FB', 'MSFT','AMZN'])
  for item in csv_list:
      df1 = pd.read_csv(item, index_col=[0])
      out = out.append(df1)
  out.sort_values(by=['Date'], inplace=True, ascending=True)
  out.to_csv(data_dir + 'compiled_error_results.csv')

with DAG(
    'Big_Data_HW4_Task2_2',
    default_args = default_args,
    description = 'DAG for Q2.2 HW4',
    start_date = datetime(2021, 11, 30, 7, 0, 0),
    schedule_interval = '* 7 * * *',
    catchup = False,
    tags = ['example'],
) as dag:
```

so2639_6893_hw4_Task2_2.py 6

so2639_6893_hw4_Task2_2.py > ...

```python

t1 = PythonOperator(
  task_id = 't1',
  python_callable = init_df_dict,
)

t2 = PythonOperator(
  task_id = 't2',
  python_callable = fetch_data,
  op_kwargs = {'ticker': 'AAPL'}
)

t3 = PythonOperator(
  task_id = 't3',
  python_callable = preprocess_data,
  op_kwargs = {'ticker': 'AAPL'}
)

t4 = PythonOperator(
  task_id = 't4',
  python_callable = predict_and_compute_error,
  op_kwargs = {'ticker': 'AAPL'}
)

t5 = PythonOperator(
  task_id = 't5',
  python_callable = fetch_data,
  op_kwargs = {'ticker': 'GOOGL'}
)

t6 = PythonOperator(
  task_id = 't6',
  python_callable = preprocess_data,
  op_kwargs = {'ticker': 'GOOGL'}
)

t7 = PythonOperator(
  task_id = 't7',
  python_callable = predict_and_compute_error,
  op_kwargs = {'ticker': 'GOOGL'}
)
```

so2639_6893_hw4_Task2_2.py 6 ×

so2639_6893_hw4_Task2_2.py > ...

```
    )

    t8 = PythonOperator(
      task_id = 't8',
      python_callable = fetch_data,
      op_kwargs = {'ticker': 'FB'}
    )

    t9 = PythonOperator(
      task_id = 't9',
      python_callable = preprocess_data,
      op_kwargs = {'ticker': 'FB'}
    )

    t10 = PythonOperator(
      task_id = 't10',
      python_callable = predict_and_compute_error,
      op_kwargs = {'ticker': 'FB'}
    )


    t11 = PythonOperator(
      task_id = 't11',
      python_callable = fetch_data,
      op_kwargs = {'ticker': 'MSFT'}
    )

    t12 = PythonOperator(
      task_id = 't12',
      python_callable = preprocess_data,
      op_kwargs = {'ticker': 'MSFT'}
    )

    t13 = PythonOperator(
      task_id = 't13',
      python_callable = predict_and_compute_error,
      op_kwargs = {'ticker': 'MSFT'}
    )

```

so2639_6893_hw4_Task2_2.py

```python

    t14 = PythonOperator(
      task_id = 't14',
      python_callable = fetch_data,
      op_kwargs = {'ticker': 'AMZN'}
    )

    t15 = PythonOperator(
      task_id = 't15',
      python_callable = preprocess_data,
      op_kwargs = {'ticker': 'AMZN'}
    )

    t16 = PythonOperator(
      task_id = 't16',
      python_callable = predict_and_compute_error,
      op_kwargs = {'ticker': 'AMZN'}
    )

    t17 = PythonOperator(
      task_id = 't17',
      python_callable = generate_output_csv,
    )

    # Task dependencies
    t1 >> [t2, t5, t8, t11, t14]
    t2 >> t3
    t3 >> t4
    t5 >> t6
    t6 >> t7
    t8 >> t9
    t9 >> t10
    t11 >> t12
    t12 >> t13
    t14 >> t15
    t15 >> t16
    [t4, t7, t10, t13, t16] >> t17

```

Task 3: (Written Parts)

3.1 Sequential Executor:

Pros: Sequential Executor allows to run Airflow without setting up many dependencies. So, we can even work directly with SQLite using the SequentialExecutor. It also identifies a single point of failure and is hence easy for debugging.

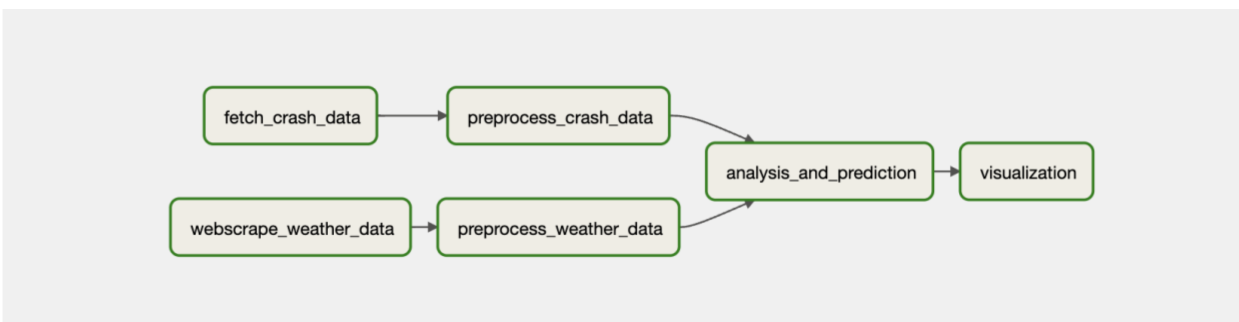
Cons: It is not recommended for any use cases that require more than a single task execution at a time, and hence usually it cannot be used in production.

Local Executor:
Pros: Local Executor is easy to set up and is resource efficient. It allows testing multiple jobs in parallel & hence offers parallelism.
Cons: It is not sufficiently scalable, and hence is only used in small-scale production environments. It is also affected by a single point of failure.

Celery Executor:
Pros: Celery Executor quickly adapts and is able to assign that allocated task or tasks to another worker, if a worker node is ever down or goes offline. It is built for horizontal scaling and has high availability.
Cons: It is expensive and worker maintenance is required. It's set up is also comparatively complicated.

Kubernetes Executor:
Pros: Kubernetes Executor is highly resource & cost efficient. Here, each task instance is run in its own pod on a Kubernetes cluster. It runs as a process in the Airflow Scheduler. When the traffic is high, we can scale up. Conversely, when the traffic is low, we can scale to zero. It also has high level of fault tolerance and has task-level configurations. Also, if a deployment is pushed, there are no interruptions to running tasks.
Cons: There is an overhead of a few extra seconds per task for a pod to spin up. Also lack of Kubernetes familiarity can potentially act as a barrier to use this.

3.2 DAG of group project "NYC Vehicle Crash Analysis":



1) The DAG for our group project "NYC Vehicle Crash Analysis" is sequenced into 6 tasks which is shown in the graph from the airflow UI above. As depicted, the tasks have been divided based on the various steps in our ETL pipeline and the analysis and visualization part of our project.
2) Tasks:
   a. fetch_crash_data: The complete historical and incremental data load of the crash dataset is done using the SODA API available on the NYC Open Data website. This formulates the first task of the DAG, which runs the python script to do the operations mentioned above.
   b. fetch_weather_data: This task executes the web scraping script to fetch the weather data. The fetched raw data is stored in the Cloud SQL postgres server

and this task executes in parallel to the crash dataset to reduce the total execution time of the pipeline.

c. preprocess_crash_data: This step involves taking the raw crash data and backfilling the missing zip code and latitude/longitude values.
d. preprocess_weather_data: This step involves taking the raw weather data and cleaning up the relevant columns that will be used in the recurrent neural network later on to predict the future weather conditions.
e. analysis_and_prediction: This step involves making correlations of crash and weather data and making weather predictions for the next two months. These predictions are used along with the crash data to predict the chances of crashes happening in a borough in New York city in the next two months.
f. visualization: The analysis done in the previous step is visualized using metabase in this task. The data used here so be fully processed so that we are able to make a better analysis. Hence this is the last task of our project.

Task Dependencies:

The dependencies for the DAG are depicted below:

fetch_crash_data >> preprocess_crash_data
fetch_weather_data >> preprocess_weather_data
[preprocess_crash_data, preprocess_weather_data] >> analysis_and_prediction
analysis_and_prediction >> visualization

As mentioned earlier, the tasks have been divided based on the several steps in the ETL pipeline and the analysis and visualization part of our project. Analysis and visualization tasks have upstream dependencies, as mentioned above, to make sure that the data used for the final analysis and visualization is clean and processed.

3) The airflow DAG is scheduled to run daily at 7pm, so that it is able to fetch the incremental crash and weather data and update the analysis on a daily basis. This schedule is set in the airflow arguments in the code, where the schedule_interval is mentioned as days = 1, and start time is at 15th November 2021 at 7 pm.