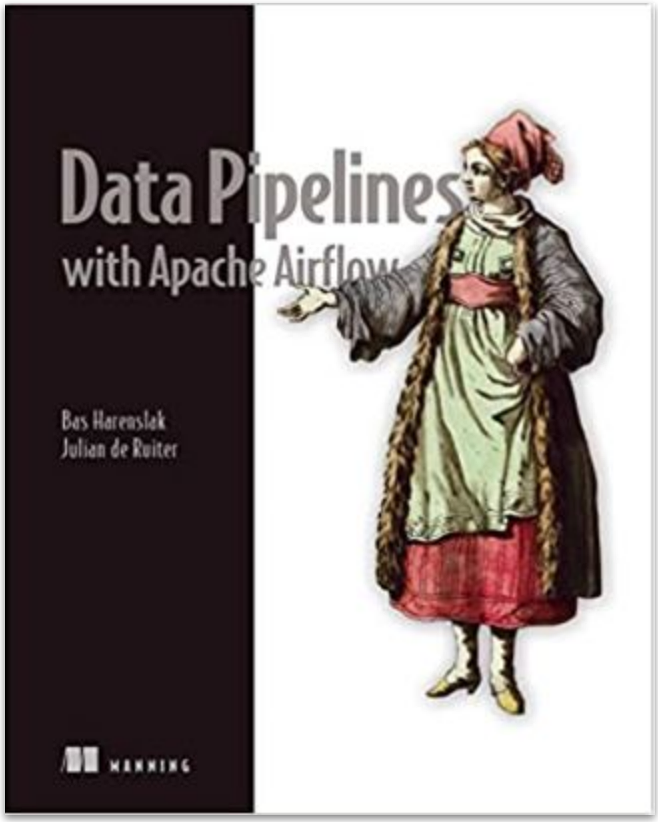
UMD DATA605 - Big Data Systems
Orchestration with Airflow
Deploying an Application

Dr. GP Saggese
gsaggese@umd.edu

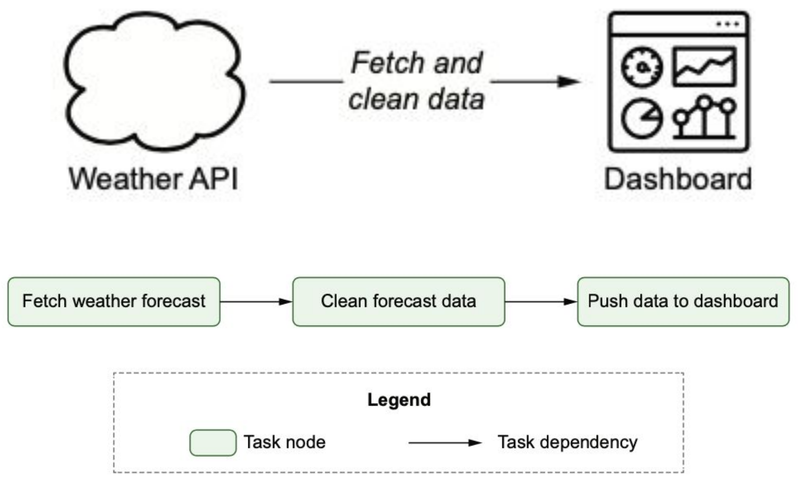
Orchestration - Resources

- Concepts in the slides
- Airflow tutorial
- Class project
- Web resources
  - [Documentation]
  - [Tutorial]
- Mastery
  - [Data Pipelines with Apache Airflow]



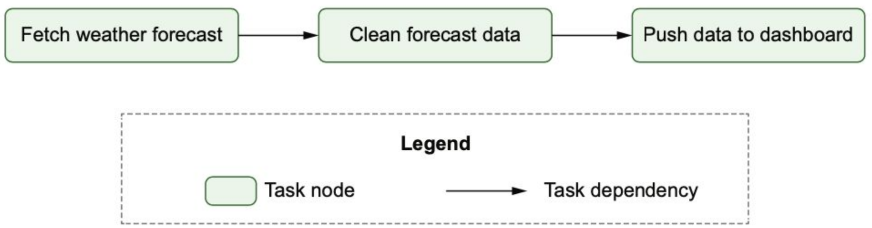
Workflow Managers

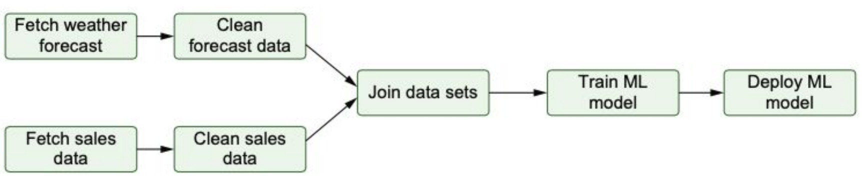
- **Orchestration problem** = data pipelines require to coordinate jobs across systems
  - Tasks run on a certain schedule
  - Tasks need to be executed in a specific order
  - Notify if a job fails
  - Retry on failure
  - Monitor: track how long it takes to run
- **E.g., live weather dashboard**
  - Fetch the weather data from API
  - Clean / transform the data
  - Push data to the dashboard / website



Workflow Managers

- Workflow managers address the orchestration problem
  - E.g., Airflow, Luigi, Metaflow, make, cron ...
- Represent data pipelines as DAGs
  - Nodes are tasks
  - Direct edges are dependencies
  - A task is executed only when all the ancestors have been executed
  - Independent tasks can be executed in parallel
  - Re-run failed tasks incrementally
- How to describe data pipelines
  - Static files (e.g., XML, YAML)
  - Workflows-as-code (e.g., Airflow and Python)
- Provide scheduling
  - How to describe what and when to run
- Provide backfilling and catch-up
  - Horizontally scalable (e.g., multiple runners)
- Provide monitoring web interfaces





(Apache) Airflow

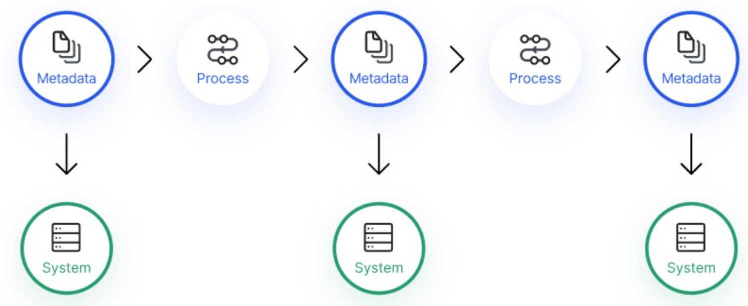
- Developed at AirBnB in 2015
  - Open-sourced as part of Apache
- **Batch oriented** framework for building data pipelines
- **Data pipelines**
  - Represented as DAGs
  - Described as code in Python
- **Scheduler with rich semantics**
- Web-interface for monitoring
- Large ecosystem
  - Support many DBs
  - Many actions (e.g., emails, pages)
- **Hosted and managed solution**
  - Run Airflow on your laptop (e.g., in class project)
  - Managed solution (e.g., AWS)

Airflow: Execution Semantics

- **Scheduling semantic**
  - Describe when the next scheduling interval is
    - E.g., every day at midnight, every 5 minutes on the hour
  - Similar to `cron`
- **Retry**
  - If a task fails, it can be re-run (after a wait time) to recover from intermittent failures
- **Incremental processing**
  - Time is divided in intervals given the schedule
  - Execute DAG for data only in that interval, instead of processing the entire data set
- **Catch-up**
  - Run all the schedule intervals up to now
- **Backfilling**
  - Execute DAG for historical schedule intervals that occurred in the past
  - E.g., if the data pipeline has changed one needs to re-process data from scratch

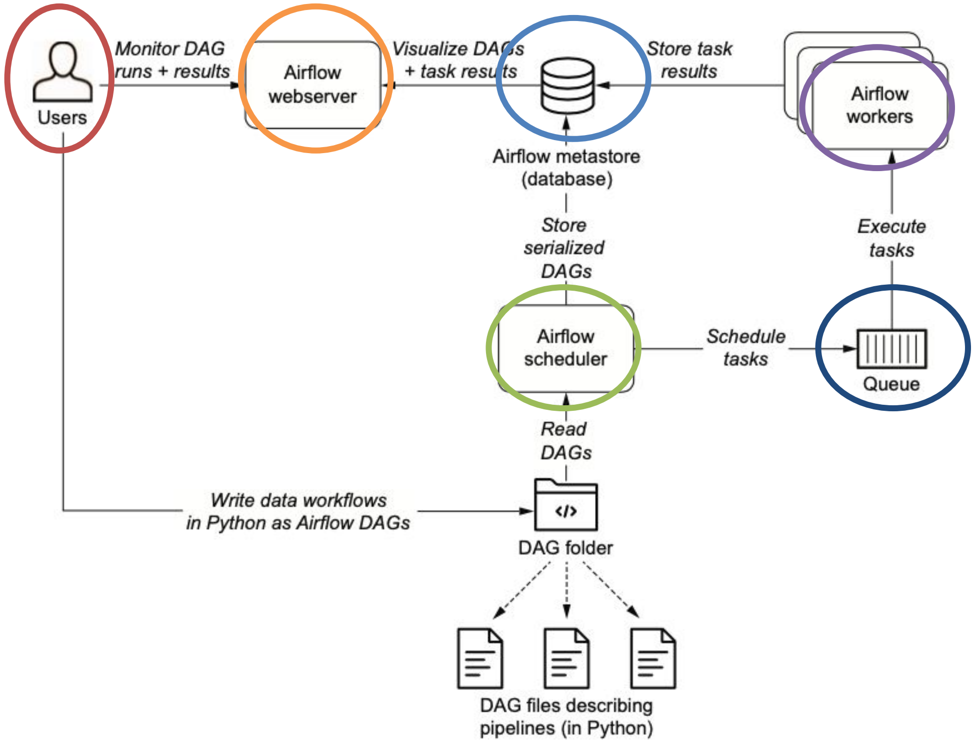
Airflow: What Doesn't Do Well

- **Not great for streaming pipelines**
  - Better for recurring or batch-oriented tasks
  - Time is assumed to be discrete and not continuous
    - E.g., schedule every hour (instead of process data as it comes)
- **Prefer static pipelines**
  - DAGs should not change (too much) between runs
- **No data lineage**
  - No tracking of how data is transformed through the pipeline
  - Need to be implemented manually
- **No data versioning**
  - No tracking of updates to the data
  - Need to be implemented manually



Airflow: Components

- **Users**
- **Web-server**
  - Visualize DAGs
  - Monitor DAG runs and results
- **Scheduler**
  - Parse DAGs
  - Keep track of completed dependencies
  - Add tasks to the execution queue
  - Schedule tasks when time comes
- **Metastore**
  - Keep the state of the system
  - E.g., what DAG nodes have been executed
- **Queue**
  - Tasks ready for execution
  - Tasks picked up by a pool of Workers
- **Workers**
  - Pick up tasks from Queue
  - Execute the tasks
  - Register task outcome in Metastore



Airflow: Tutorial

- From the tutorial for Airflow 2.2.2
- Follow Airflow Tutorial in README from https://github.com/sorrentum/sorrentum/tree/master/sorrentum_sandbox

Airflow: Tutorial

- The script describes the DAG structure as Python code
  - There is no computation inside the DAG code
  - It only defines the DAG structure and metadata (e.g., about scheduling)
- The `Scheduler` executes the code to build DAG
- `BashOperator` creates a task wrapping a Bash command

airflow/example_dags/tutorial.py

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

# 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
```

Airflow: Tutorial

- Dict with various default params to pass to the DAG constructor
  - Can have different set-ups for dev vs prod
- Instantiate the DAG

airflow/example_dags/tutorial.py [view source]

```python
# These args will get passed on to each operator
# You can override them on a per-task basis during operator initialization
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # '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'
}
```

airflow/example_dags/tutorial.py [view source]

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

Airflow: Tutorial

- DAG defines tasks by instantiating `Operator` objects
  - The default params are passed to all the tasks
  - Can be overridden explicitly

airflow/example_dags/tutorial.py view source

```
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
)

t2 = BashOperator(
    task_id='sleep',
    depends_on_past=False,
    bash_command='sleep 5',
    retries=3,
)
```

- One can use a Jinja template

airflow/example_dags/tutorial.py view source

```
templated_command = dedent(
    """
{% for i in range(5) %}
    echo "{{ ds }}"
    echo "{{ macros.ds_add(ds, 7)}}"
    echo "{{ params.my_param }}"
{% endfor %}
"""
)

t3 = BashOperator(
    task_id='templated',
    depends_on_past=False,
    bash_command=templated_command,
    params={'my_param': 'Parameter I passed in'},
)
```

- Add tasks to the DAG with dependencies

```
t1 >> [t2, t3]
```

Airflow: Concepts

- Each DAG run represents a data interval, i.e., an interval between two times
  - E.g., A DAG scheduled @daily
  - Each data interval starts at midnight for each day, ends at midnight of next day
- DAG scheduled after data interval has ended
- Logical date
  - Simulate the scheduler running DAG / task for a specific date
  - Even if it is physically run now