Get In Control Of Your Workflows With Airflow

Christian Trebing, Apache Big Data 2016

@ctrebing

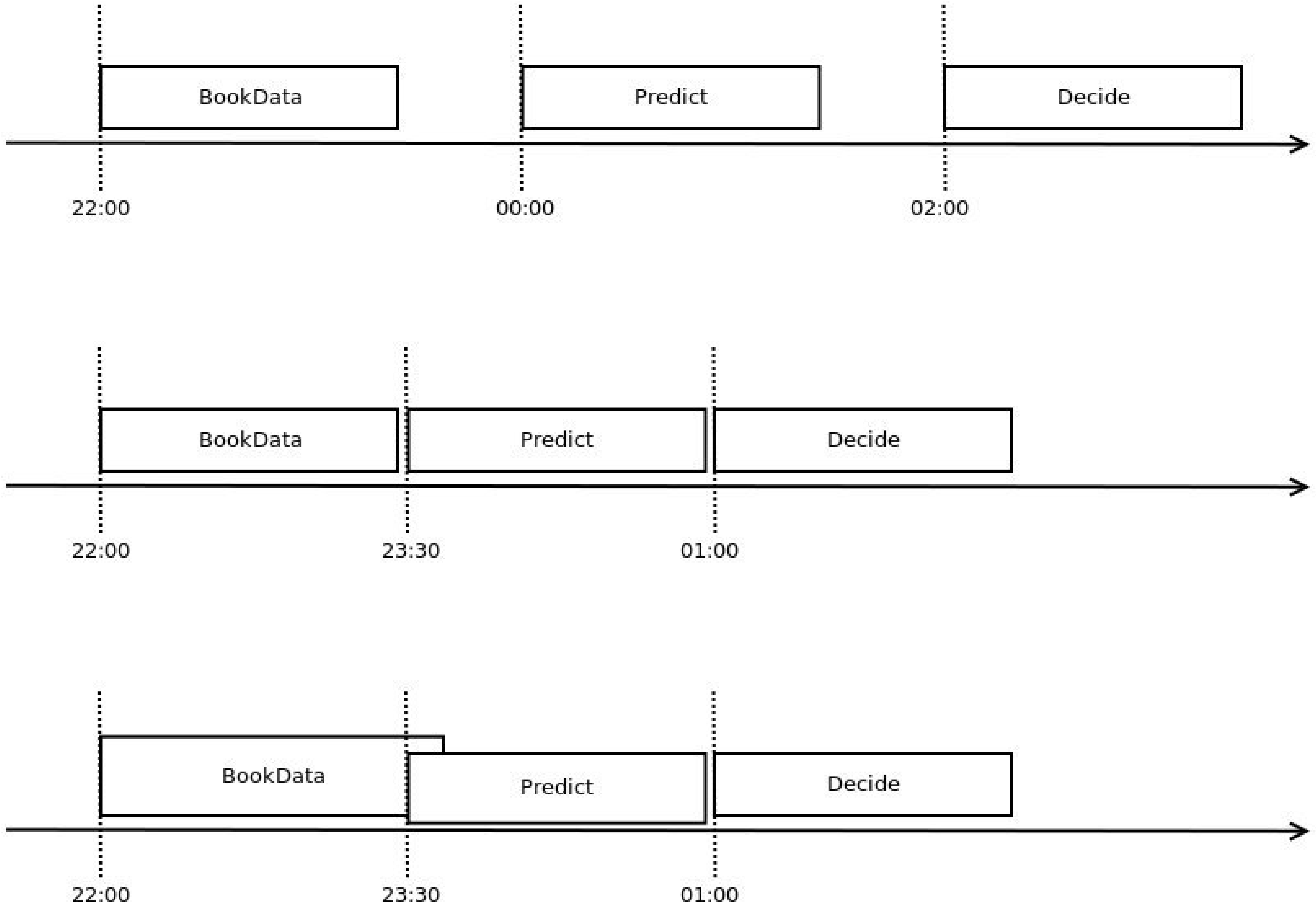
Imagine:

- you are a data driven company
- each night you get data from your customers and this data wants to be processed
- processing happens is done in separate steps (for example booking, machine learning, decision taking)
- if errors happen, you want to get an overview on what happened when
- as you might have already guessed: you have a tight time schedule each night

What options do you have?

Doing it with cron

- works for the start
- only time triggers possible, no dependency
- error handling is hard



Writing a workflow processing tool

- we did that for the start. and not just one.
- start is easy, everything is great.
- soon you reach the limits. Then you either have to invest much more than you thought initially or live with the limits
  - some ideas: concurrency, traceability, manual triggers, external interfaces, ui

Using an open source workflow processing tool

- we evaluated multiple ones and decided for airflow

Why did we decide for airflow?

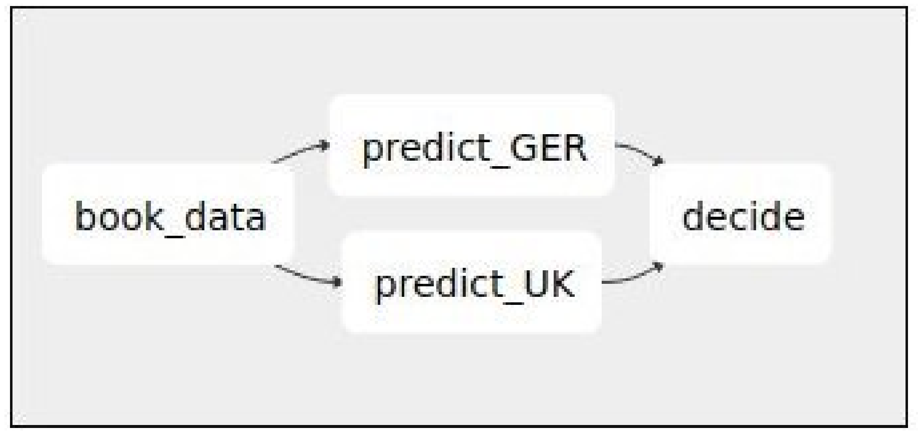
- written in python. we know that and we like it.
- also workflows are defined in python code
- view of present and past runs, logging features
- extensible through plugins
- active development (apache incubator project)
- nice ui, possibility to define a REST interface
- relatively lightweight: two processes on a server + some database

In [ ]:

```python
from airflow import DAG
from airflow.operators import BookData, Predict, Decide

dag_id = "daily_processing"
schedule_interval = '0 22 * * *'

default_args = {
    'retries': 2,
    'retry_delay': timedelta(minutes=5)
}

dag = DAG(
    dag_id,
    start_date=datetime.date(2016, 12, 7),
    schedule_interval=schedule_interval,
    default_args=default_args)

book = BookData(dag=dag)

predict = Predict(dag=dag)
predict.set_upstream(book)

decide = Decide(dag=dag)
decide.set_upstream(predict)
```

In [ ]:

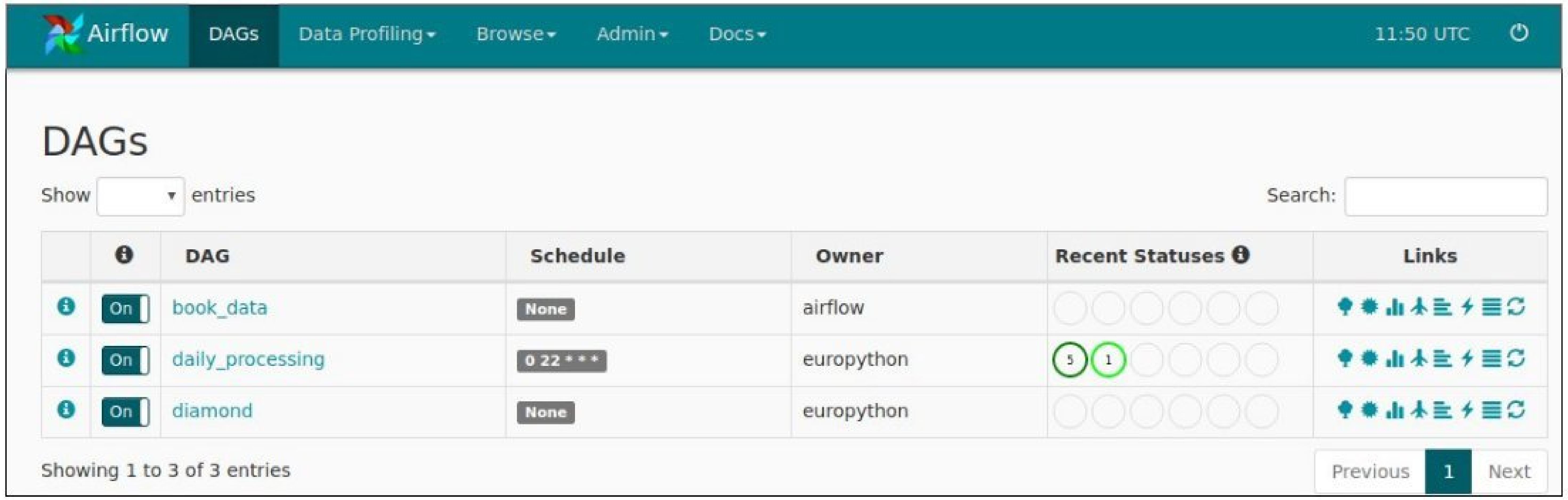
```
# Fan In / Fan Out

book = BookData(dag=dag)

predict_ger = Predict(dag=dag, country='GER')
predict_ger.set_upstream(book)

predict_uk = Predict(dag=dag, country='UK')
predict_uk.set_upstream(book)

decide = Decide(dag=dag)
decide.set_upstream(predict_ger)
decide.set_upstream(predict_uk)
```



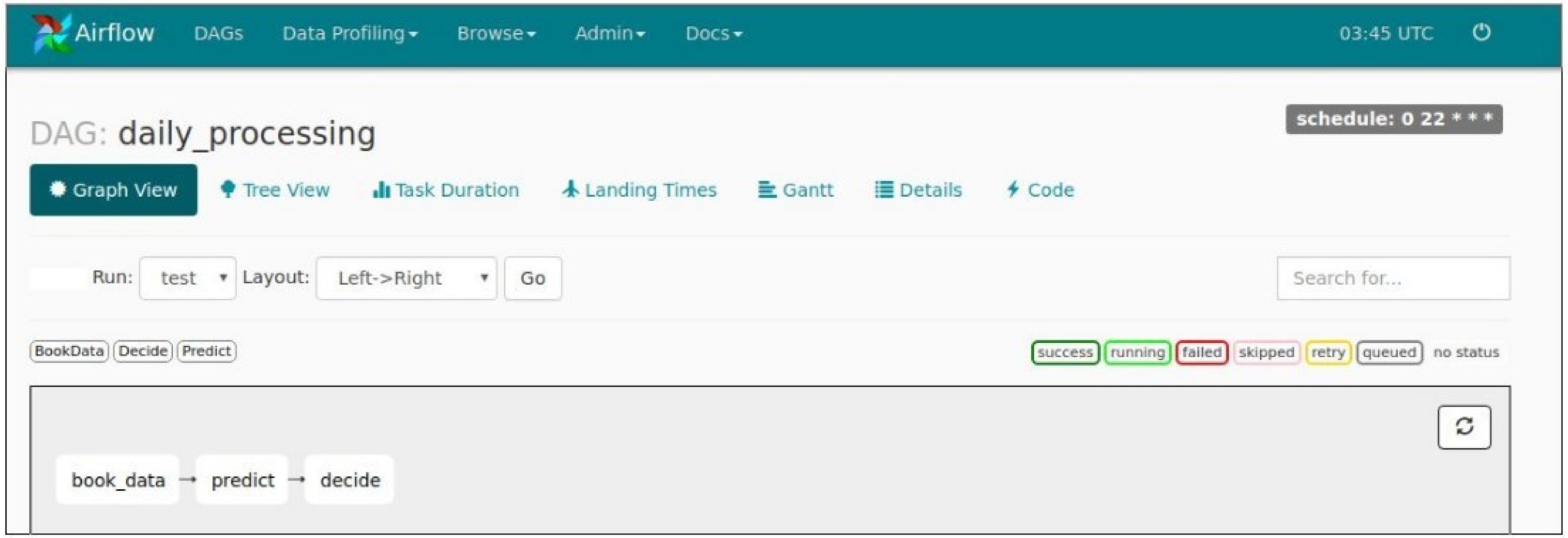
On the UI

- DAG overview (start screen)
- run view
- tree view
- runtimes
- gantt chart
- log view

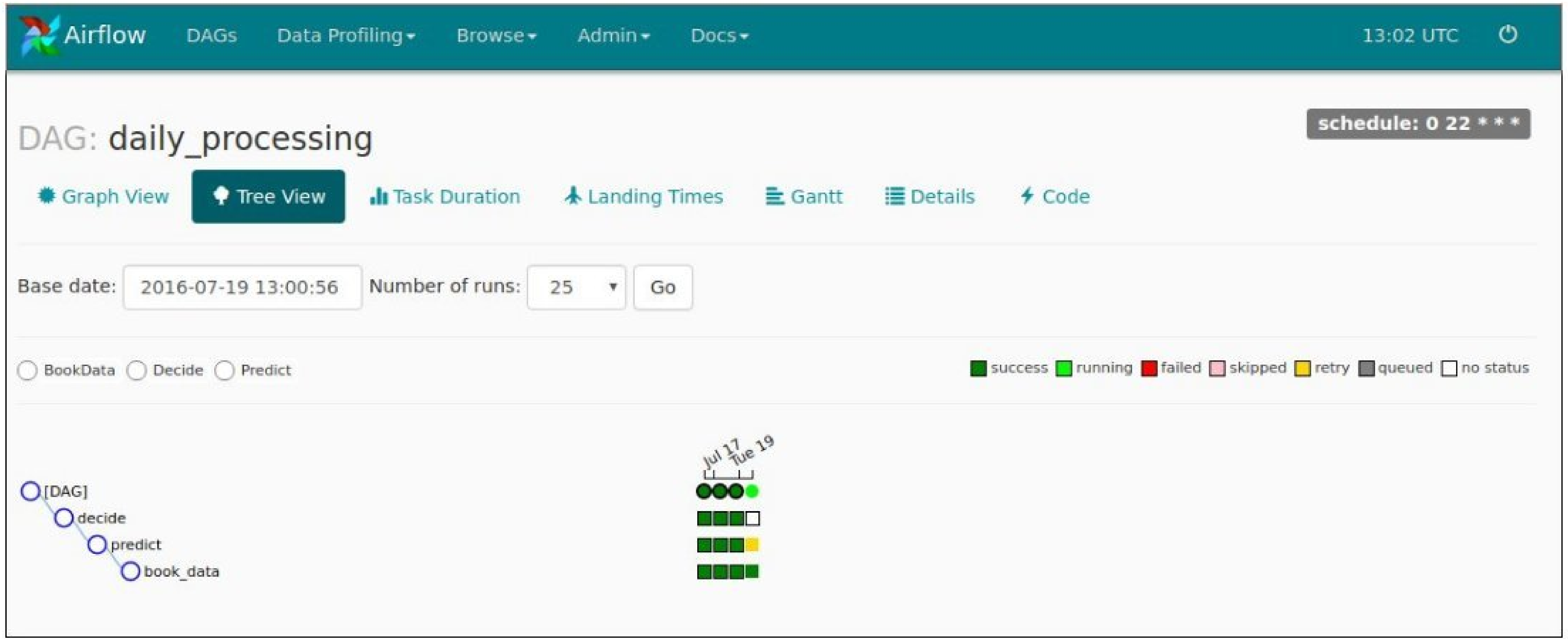
DAG Overview (start screen)



DAG Run View

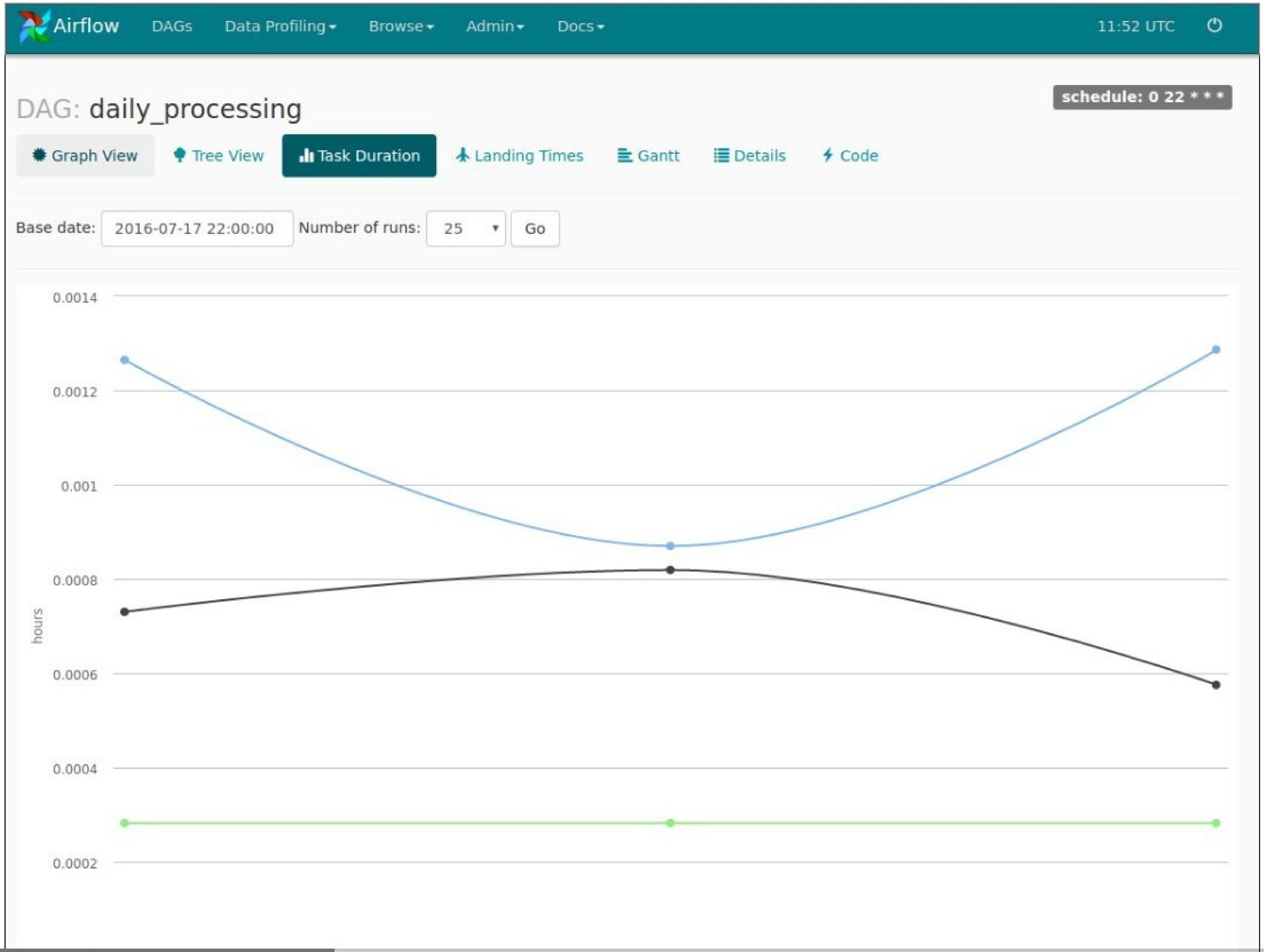

Airflow
DAGs
Data Profiling
Browse
Admin
Docs
03:45 UTC
DAG: daily_processing
schedule: 0 22 * * *
Graph View
Tree View
Task Duration
Landing Times
Gantt
Details
Code
Run: test
Layout: Left->Right
Go
Search for...
BookData
Decide
Predict
success
running
failed
skipped
retry
queued
no status
book_data
predict
decide

Tree View



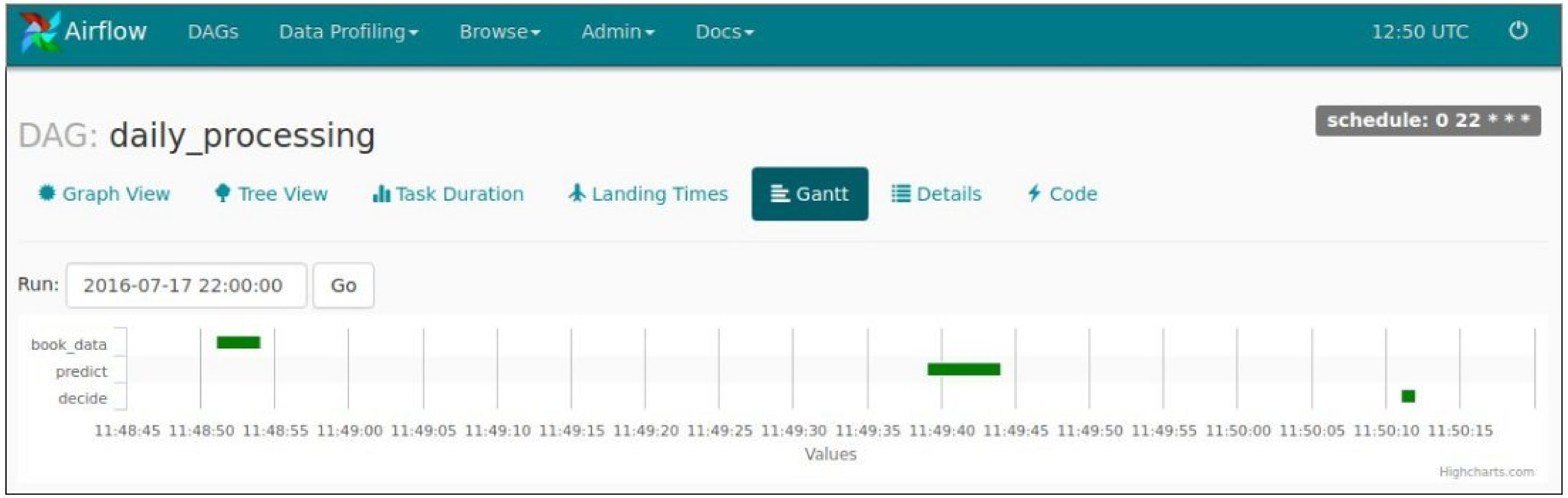
Airflow
DAGs
Data Profiling
Browse
Admin
Docs
13:02 UTC
DAG: daily_processing
schedule: 0 22 * * *
Graph View
Tree View
Task Duration
Landing Times
Gantt
Details
Code
Base date:
2016-07-19 13:00:56
Number of runs:
25
Go
BookData
Decide
Predict
success
running
failed
skipped
retry
queued
no status
Jul 17
Tue 19
[DAG]
decide
predict
book_data

Runtimes

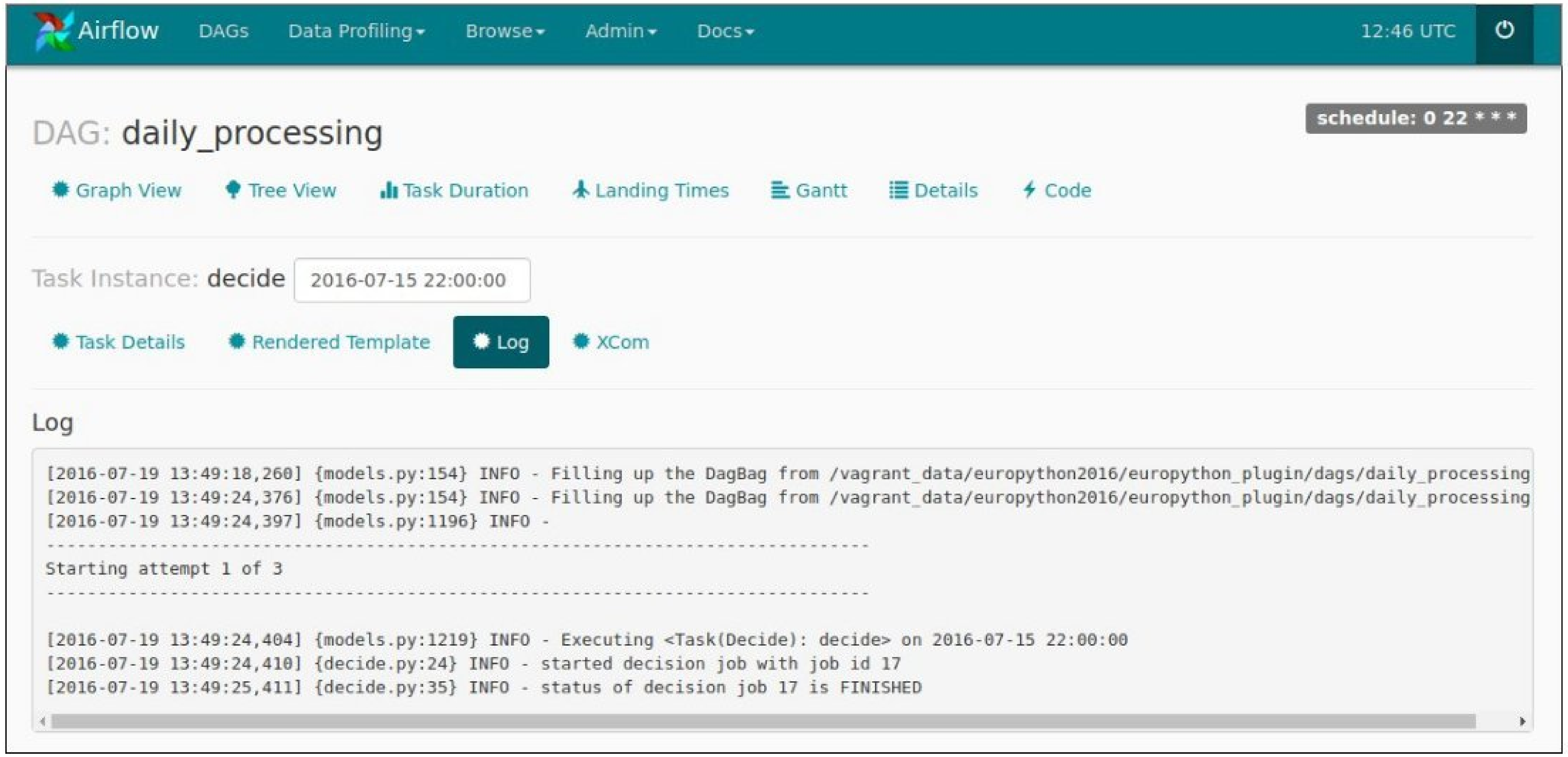


Airflow
DAGs
Data Profiling
Browse
Admin
Docs
11:52 UTC
DAG: daily_processing
schedule: 0 22 * * *
Graph View
Tree View
Task Duration
Landing Times
Gantt
Details
Code
Base date: 2016-07-17 22:00:00
Number of runs: 25
Go
0.0014
0.0012
0.001
0.0008
0.0006
0.0004
0.0002
hours

Gantt Chart


Airflow
DAGs
Data Profiling
Browse
Admin
Docs
12:50 UTC
DAG: daily_processing
schedule: 0 22 * * *
Graph View
Tree View
Task Duration
Landing Times
Gantt
Details
Code
Run:
2016-07-17 22:00:00
Go
book_data
predict
decide
11:48:45 11:48:50 11:48:55 11:49:00 11:49:05 11:49:10 11:49:15 11:49:20 11:49:25 11:49:30 11:49:35 11:49:40 11:49:45 11:49:50 11:49:55 11:50:00 11:50:05 11:50:10 11:50:15
Values
Highcharts.com

Log View


Airflow
DAGs
Data Profiling
Browse
Admin
Docs
12:46 UTC
DAG: daily_processing
schedule: 0 22 * * *
Graph View
Tree View
Task Duration
Landing Times
Gantt
Details
Code
Task Instance: decide
2016-07-15 22:00:00
Task Details
Rendered Template
Log
XCom
Log
[2016-07-19 13:49:18,260] {models.py:154} INFO - Filling up the DagBag from /vagrant_data/europython2016/europython_plugin/dags/daily_processing
[2016-07-19 13:49:24,376] {models.py:154} INFO - Filling up the DagBag from /vagrant_data/europython2016/europython_plugin/dags/daily_processing
[2016-07-19 13:49:24,397] {models.py:1196} INFO -

Starting attempt 1 of 3

[2016-07-19 13:49:24,404] {models.py:1219} INFO - Executing <Task(Decide): decide> on 2016-07-15 22:00:00
[2016-07-19 13:49:24,410] {decide.py:24} INFO - started decision job with job id 17
[2016-07-19 13:49:25,411] {decide.py:35} INFO - status of decision job 17 is FINISHED

Operators

Many basic operators are included in airflow:

- BashOperator
- SimpleHttpOperator
- PostgresOperator / SqliteOperator
- PythonOperator
- EmailOperator
- ...

Also there are sensors to wait for things:

- HttpSensor
- HdfsSensor
- SqlSensor
- ...

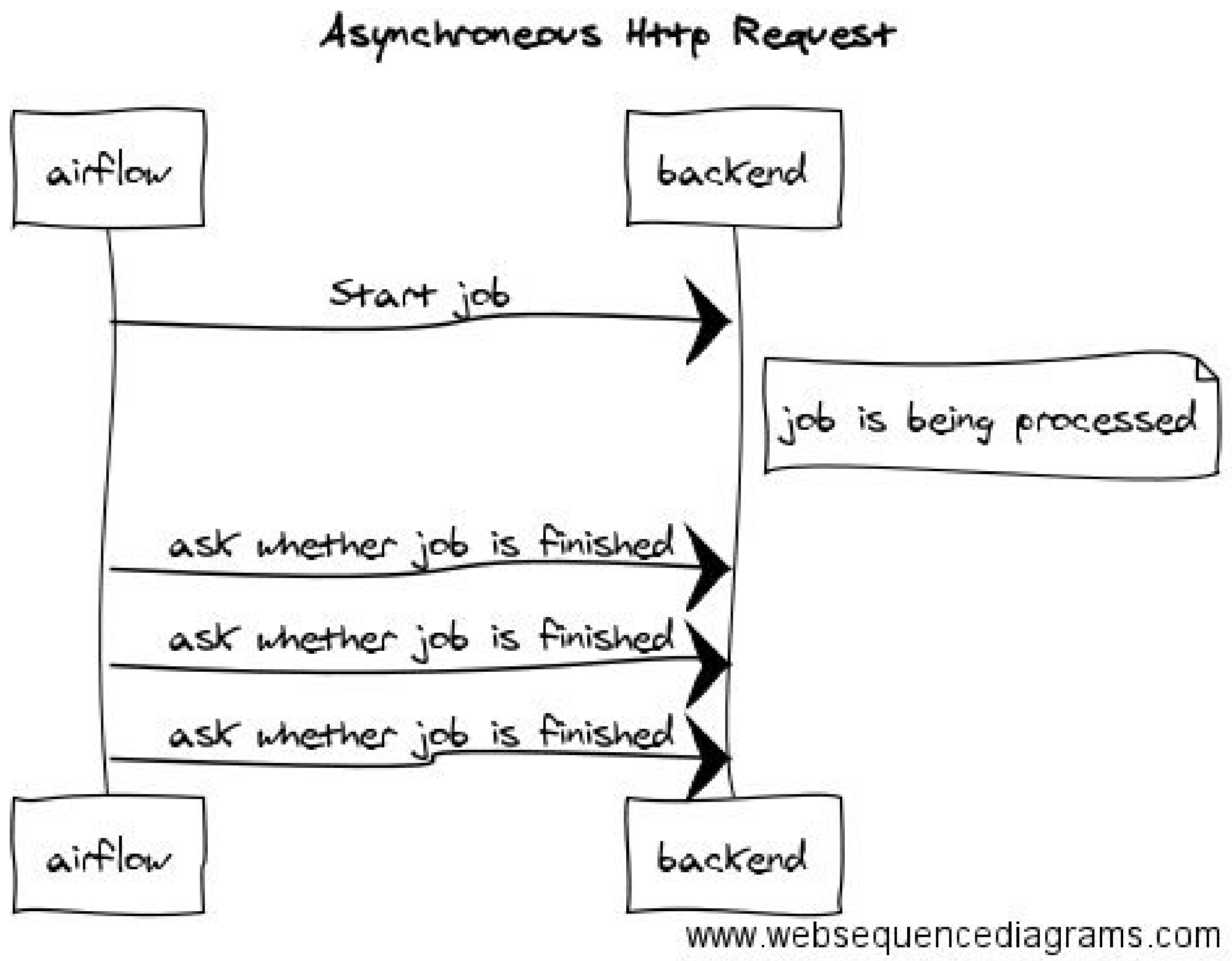
Python Operator

Within your DAG definition, you can define arbitrary python code that can be run by a PythonOperator

In [ ]:
```
def print_context(ds, **kwargs):
    pprint(kwargs)
    print(ds)
    return 'Whatever you return gets printed in the logs'

run_this = PythonOperator(
    task_id='print_the_context',
    provide_context=True,
    python_callable=print_context,
    dag=dag)
```

- Great flexibility
- be aware of dependencies: all imported python packages have to be available on all worker nodes

But I need a different operator...



1. I could use a SimpleHttpOperator and afterwards an HttpSensor
    - would work functional wise
    - but wouldn't it be nice to see the execution time directly as the operator run time?
2. Time for a new operator!

In [ ]:

```
# operator implementation

import time, logging
from airflow import models, hooks

class Decide(models.BaseOperator):
    @airflow_utils.apply_defaults
    def __init__(self, **kwargs):
        super(Decide, self).__init__(
            task_id='decide',
            **kwargs)
        self.http_conn_id = 'DECISION_SERVER'
        self.endpoint_job_start = 'decide/'
        self.endpoint_job_status = 'job_status/'

    def execute(self, context):
        http = hooks.HttpHook(method='POST', http_conn_id=self.http_conn
        response = http.run(endpoint=self.endpoint_job_start)
        job_id = response.json()['job_id']
        logging.info('started decision job with job id {}'.format(job_id
        self.wait_for_job(job_id)

    def wait_for_job(self, job_id):
        job_status = None
        http = hooks.HttpHook(method='GET', http_conn_id=self.http_conn_
        while not job_status == 'FINISHED':
            time.sleep(1)
            response = http.run(endpoint=self.endpoint_job_status + str(
            job_status = response.json()['status']
            logging.info('status of decision job {} is {}'.format(job_id
```

State Handling

Variables

- per airflow instance

XCOMs

- per DAG run / task

These two types of states are persisted in two database tables

- does not get lost on scheduler restart

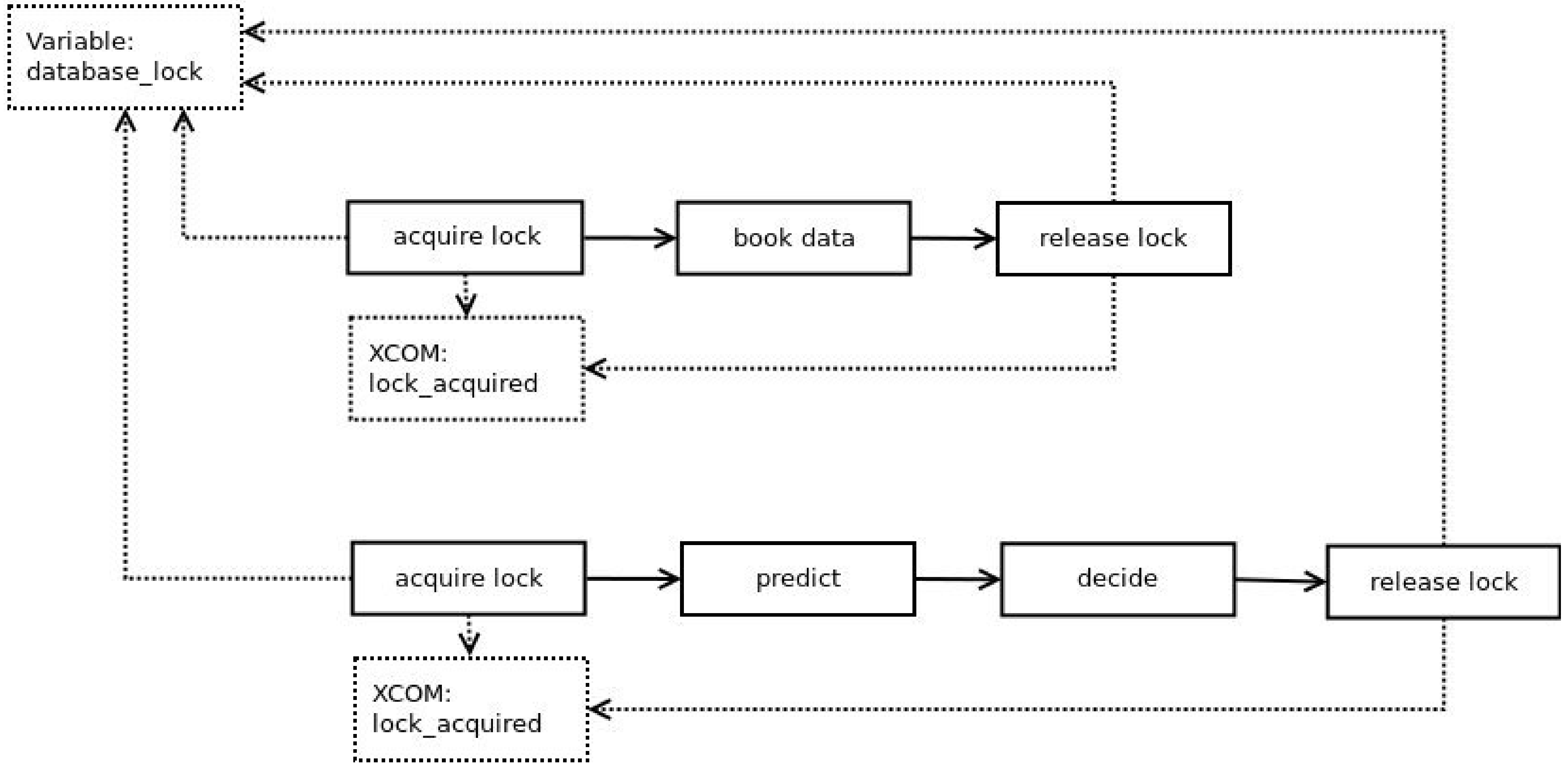
Example: Resource Reservation

Requirements:

- Some of the tasks inside my DAG require exclusive access to resource
- multiple DAGs exist that require this exclusive access

Solution:

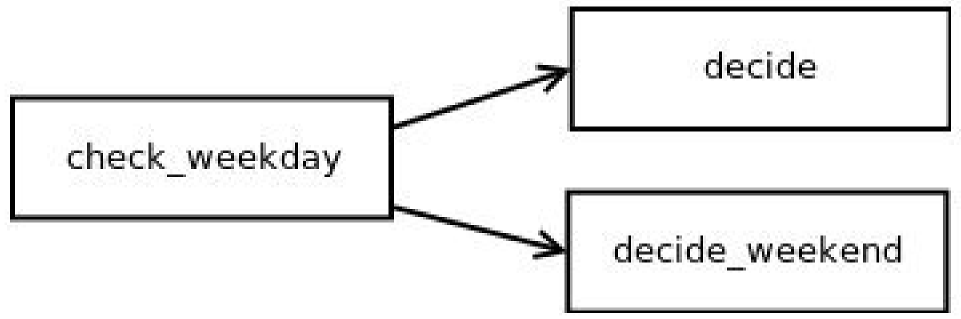
- before each task block requiring the resource insert a new task that acquires a lock for this resource
- after each task block requiring the resource insert a new task that returs the lock for this resource
- only return the lock if it has been acquired during this DAG


Variable:
database_lock
acquire lock
book data
release lock
XCOM:
lock_acquired
acquire lock
predict
decide
release lock
XCOM:
lock_acquired

Branching

- use BranchPythonOperator
- implement decision logic in python function
- return value of python function is task_id to be done next

example: do different processing on weekend



In [ ]:

```python
def check_weekday_python():
    weekday = datetime.now().weekday()
    if weekday in [5, 6]:
        return 'decide_weekend'
    else:
        return 'decide'

check_weekday = BranchPythonOperator(
    task_id='check_weekday',
    python_callable=check_weekday_python,
    dag=dag
)
```

Plugin Concept

- own operators
- own blueprints
- in the airflow configuration, give path to plugin

Plugin Implementation

In [ ]:
```
from airflow.plugins_manager import AirflowPlugin

from plugins import blueprints
from plugins import operators


# Defining the plugin class
class EuropythonPlugin(AirflowPlugin):
    name = "europython_plugin"
    operators = [
        operators.BookData,
        operators.Predict,
        operators.Decide
    ]
    flask_blueprints = [blueprints.TriggerBlueprint]
```

Defining own Blueprints

Extends the web server

For example: currently, no REST API exists to ask trigger dags or ask for the state of a dag run

you can add your own blueprints that run within the webserver and can access all airflow functionality

- add as a flask blueprint
- we defined endpoints for the above (trigger dags/ask for state of a dag run)
- need to be careful of maintaining them through an airflow version upgrade

for implementation, see the example git repo

In [ ]:

```
curl -X POST localhost:8080/trigger/daily_processing
{ "dag_id": "daily_processing", "run_id": "external_trigger_2016-07-19T1
```

In [ ]:

```
curl localhost:8080/trigger/daily_processing/external_trigger_2016-07-19
{ "dag_id": "daily_processing",
  "execution_date": "2016-07-19T15:12:28",
  "run_id": "external_trigger_2016-07-19T15:12:28.398352",
  "state": "running"}
```

Deployment / What happens inside

Two processes and a database

- scheduler
- webserver
- database: postgres, sqlite(with restrictions), ...

Executor: different possibilities exist

- SequentialExecutor (within scheduler process)
- LocalExecutor (with subprocesses)
- Celery Framework (multiple worker nodes)

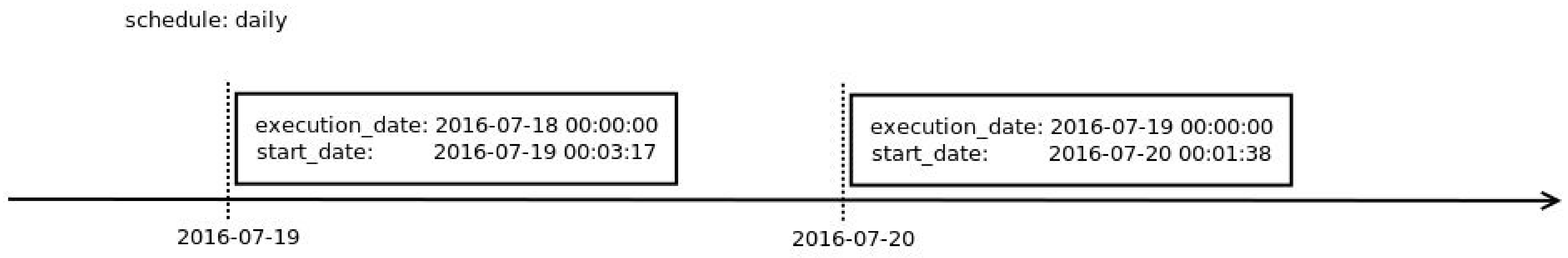
How we use it

- automatic and manual triggers
- one airflow instance per system we manage
- database: sometimes postgres, sometimes sqlite
- lightweight executors, only triggers http requests
- contributing to airflow with pull requests
    - external_triggers functionality (PR 503/540)
    - plugin detection mechanism (PR 730)

Challenges / Pitfalls

- scheduling
- start time and backfill

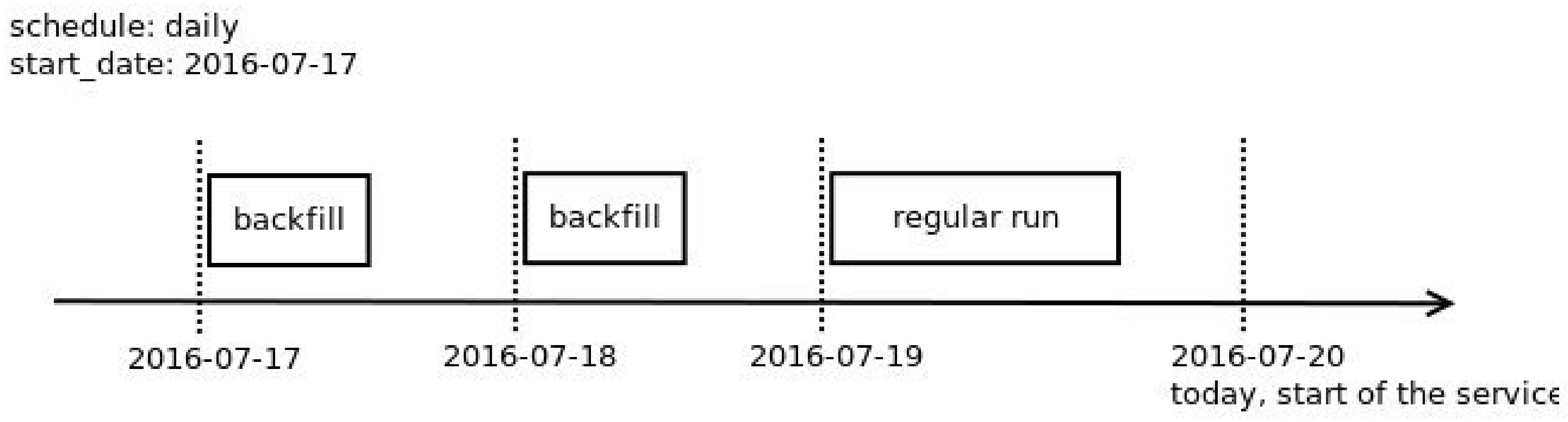
Scheduling



- start date: when did it start really
- execution date:
  - more like a description for that run
  - always one iteration back in time
  - comes from ETL scenarios where data was available only on the next day

Start Time and Backfill

- for every dag, you have to define a start time
- if the dag has a schedule, the scheduler will trigger a backfill to that date



When you know the start time at design time of your dag, this is fine.

If not, you have to take care what date to enter:

- it should not be too much in the past, otherwise backfill will be triggered
- ideally it should be one iteration before your first intended run

blueyonder

If you want to dig deeper:

https://github.com/apache/incubator-airflow

airflow documentation http://pythonhosted.org/airflow/

common pitfalls (from airflow wiki)
https://cwiki.apache.org/confluence/display/AIRFLOW/Common+Pitfalls

plugin example from this talk: https://github.com/blue-yonder/airflow-plugin-demo