

Airflow training

Thursday 8th July 2020

Fokko Driesprong



GO 
DATA
DRIVEN

About Fokko Driesprong

- Master Distributed System at University of Groningen
- Principal Code Connoisseur at GoDataDriven for 4 years
- Member of the Apache Software Foundation
- Committer on Apache {Avro, Parquet, Druid, Airflow}
- Started with Airflow over 4 years ago



Fokko Driesprong

About you (1 sentence per question)

- Background?
- Knowledge of Airflow?
- Plans to apply Airflow?
- Expectations of the training?



First remote edition

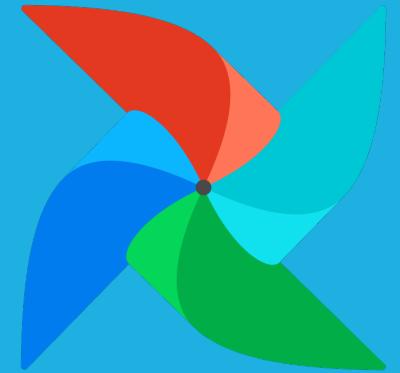
- Did the training many times (in RL)
- Balance between questions and progress
- Feel free to interrupt me
- Let's embrace the awkwardness



Program

- Introduction into Airflow
- Setup training environment
- Writing a first DAG
- Scheduling
- Templating
- Branching
- Putting it together





Airflow - Introduction

GO 
DATA
DRIVEN

What is Apache Airflow?

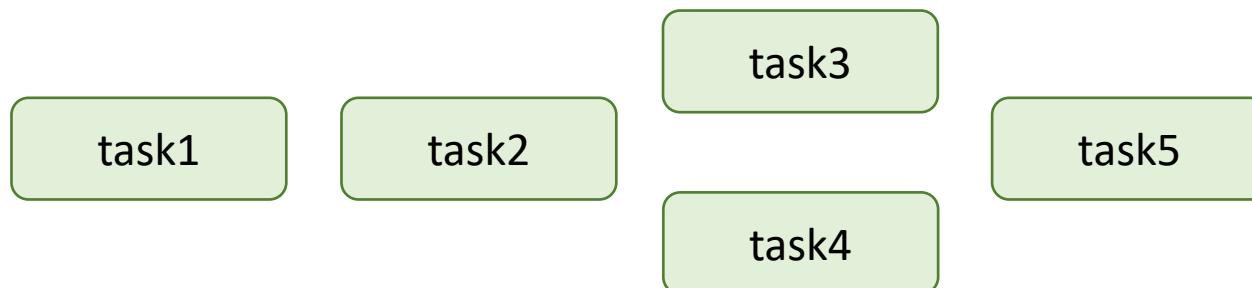


- Open source platform for creating, scheduling and monitoring workflows
- Started at AirBnB in 2015
 - Developed by Maxime Beauchemin
 - (<https://medium.com/@maximebeauchemin>)
 - Now used by 200+ companies (ING, LinkedIn, Paypal, HBO, ...)
 - Contributions from 600+ developers
- Written in Python

Why use Airflow?



- Scenario
 - We have a workflow consisting of 5 tasks
 - Some tasks need to run before others
 - Some tasks can run in parallel (tasks 3 + 4)
 - We want to run this workflow every day
 - We would like to have independent logs and be able to re-run tasks if they fail



Scenario – Single script



- We could write a simple (bash) script the executes tasks linearly, scheduled with CRON

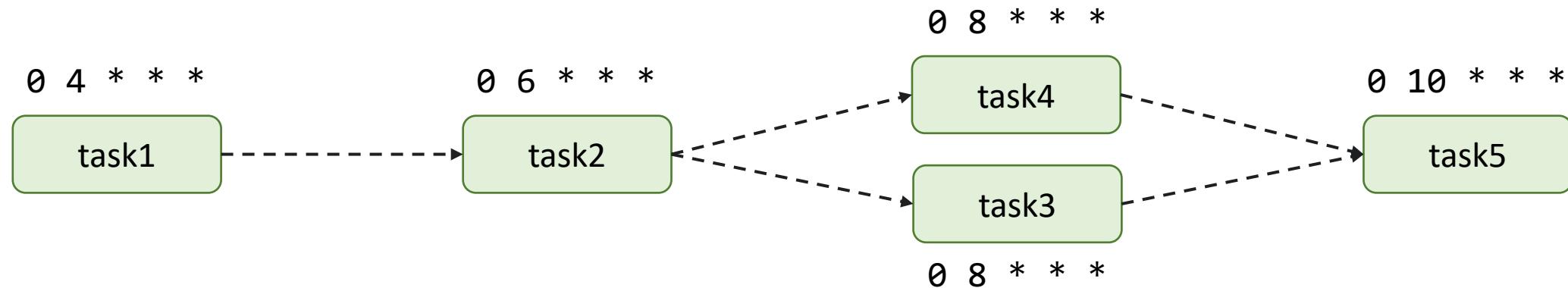


- Drawbacks
 - No parallelism between tasks 3 + 4
 - No independent logs + re-running of individual tasks

Scenario – Multiple scripts + cron



- Alternatively – split the tasks into separate scripts and schedule these using cron



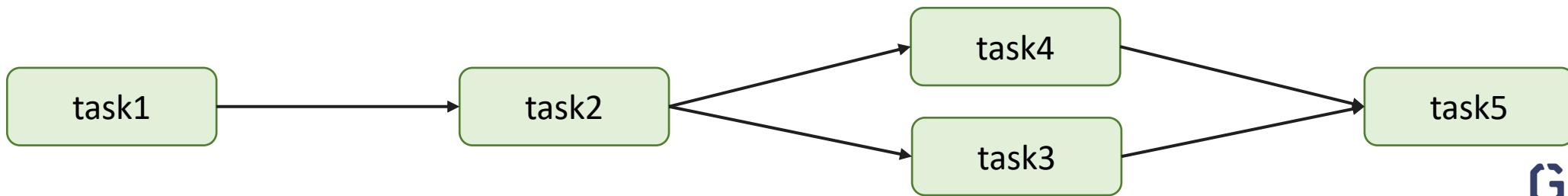
- But what if a task fails?
What if a task takes longer than expected?

Scenario - Airflow



- Workflows built as graphs (DAG) of tasks
 - Dependencies between tasks as edges
 - Workflow runs on a given schedule (or can be triggered)
 - Results/logs monitored per-task by Airflow

DAG interval: 0 4 * * *



When to use Airflow

- Running complex (data) pipelines
 - Batch-oriented data processing
 - Running at regular intervals
- Workflows as code
- Everything is Python (Airflow and Dags)
- Reusable components



When NOT to use Airflow

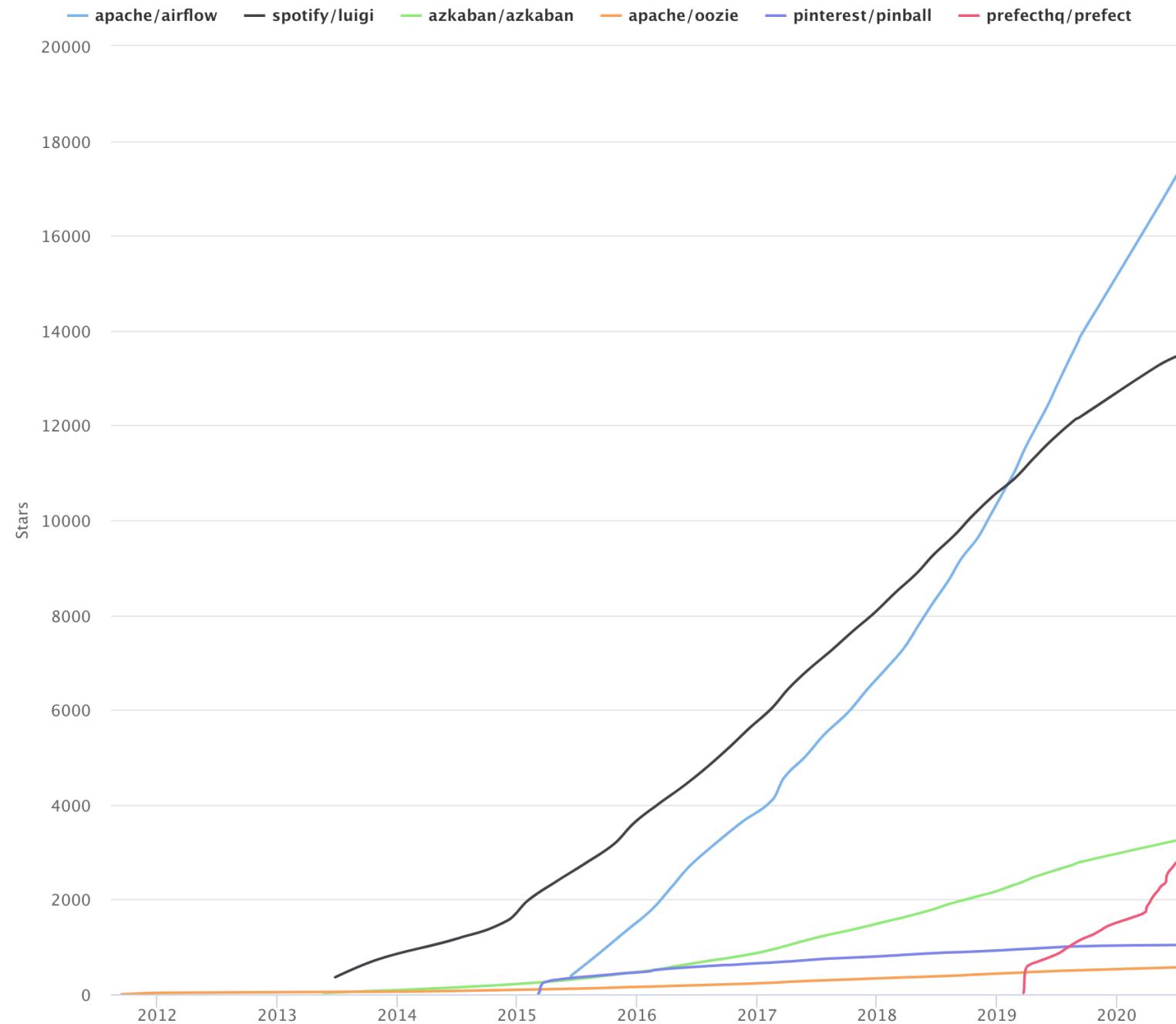
- Airflow is not a (data) processing framework (such as Spark), focusses on orchestration
- Not for streaming data solutions



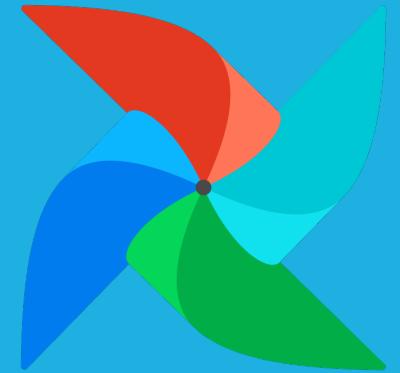
Airflow alternatives?

- There are many, many workflow managers:
 - Azkaban (LinkedIn)
 - Oozie (Apache)
 - Luigi (Spotify)
 - Pinball† (Pinterest)
 - Snakemake
 - Argo
- So why use Airflow?
 - Define workflows in Python
 - Easy to integrate + extend
 - Strong scheduling semantics





GO
DATA
DRIVEN



Use case – Recommendations

Recommendations

The screenshot shows the NPO Start website interface. At the top, there is a navigation bar with the NPO Start logo, 'Live', 'Programma's', 'Gids', a user profile icon labeled 'Bas', a search icon, and a 'Zoeken' (Search) button.

Gesprek van de dag >

Five news items are displayed in a grid:

- Honderden doden na tsunami op Sulawesi** (6 min) - A news anchor in a red dress stands next to a screen showing a destroyed building. A man in glasses is speaking.
- Terrorisme-expert over verijdelde aanslag** (57 min) - A man in a suit is speaking.
- Aanpak radicalisering in Nederland** (49 min) - A news anchor in a suit is speaking.
- Veel bomgrondstoffen bij terrorismeverdachten** (27 min) - A news anchor in a suit is speaking.
- Koos Alberts overleden** - A man in a blue jacket is standing next to a yellow car.

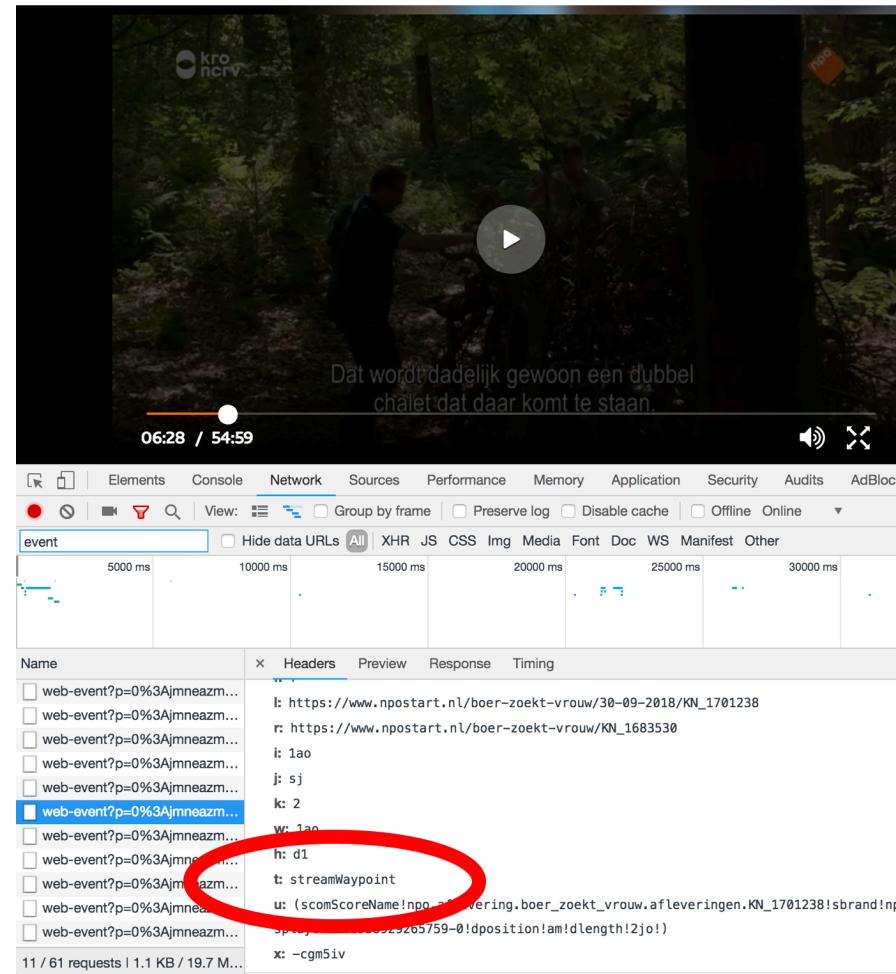
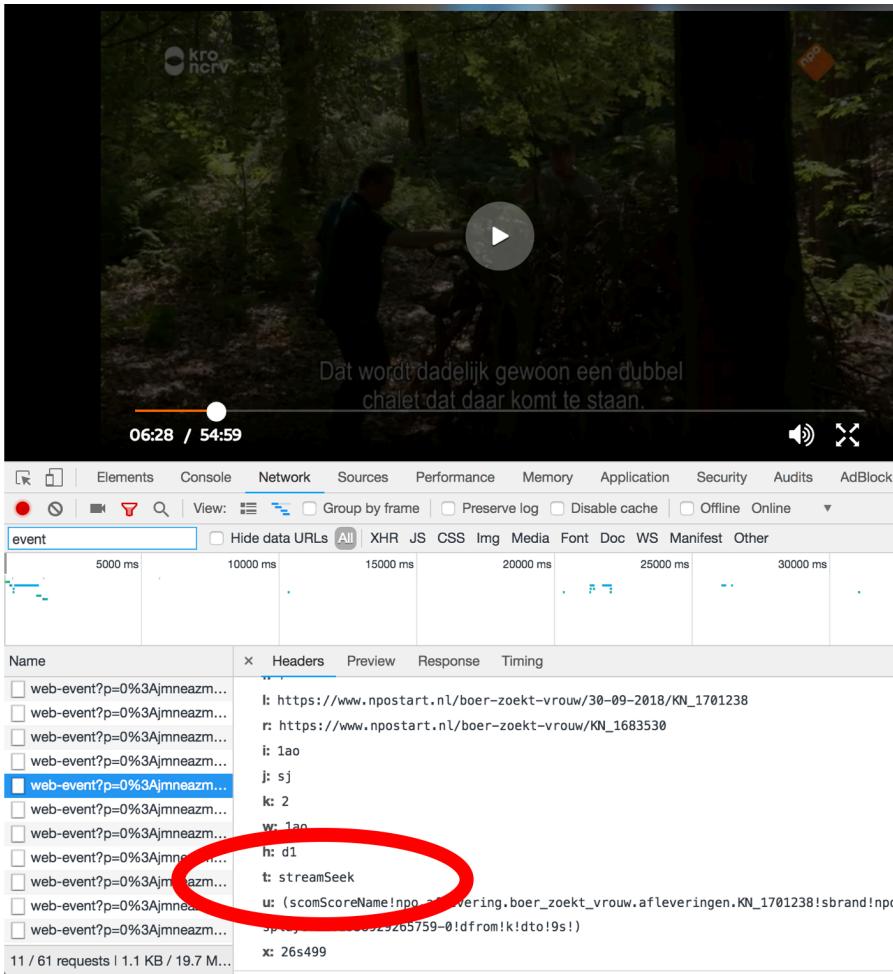
Reis wereld rond met beste reisseries >

Five travel series are displayed in a grid:

- Filemon Buiten Westen** - A man with a large afro hairstyle is smiling.
- Reizen Waes** - A group of people are on a boat on a river.
- Trippers** - A man is walking through a tunnel.
- De kruisvaarder en de sultan** - Two men are walking along a path.
- Break Free** - A man is snowboarding down a snowy slope.

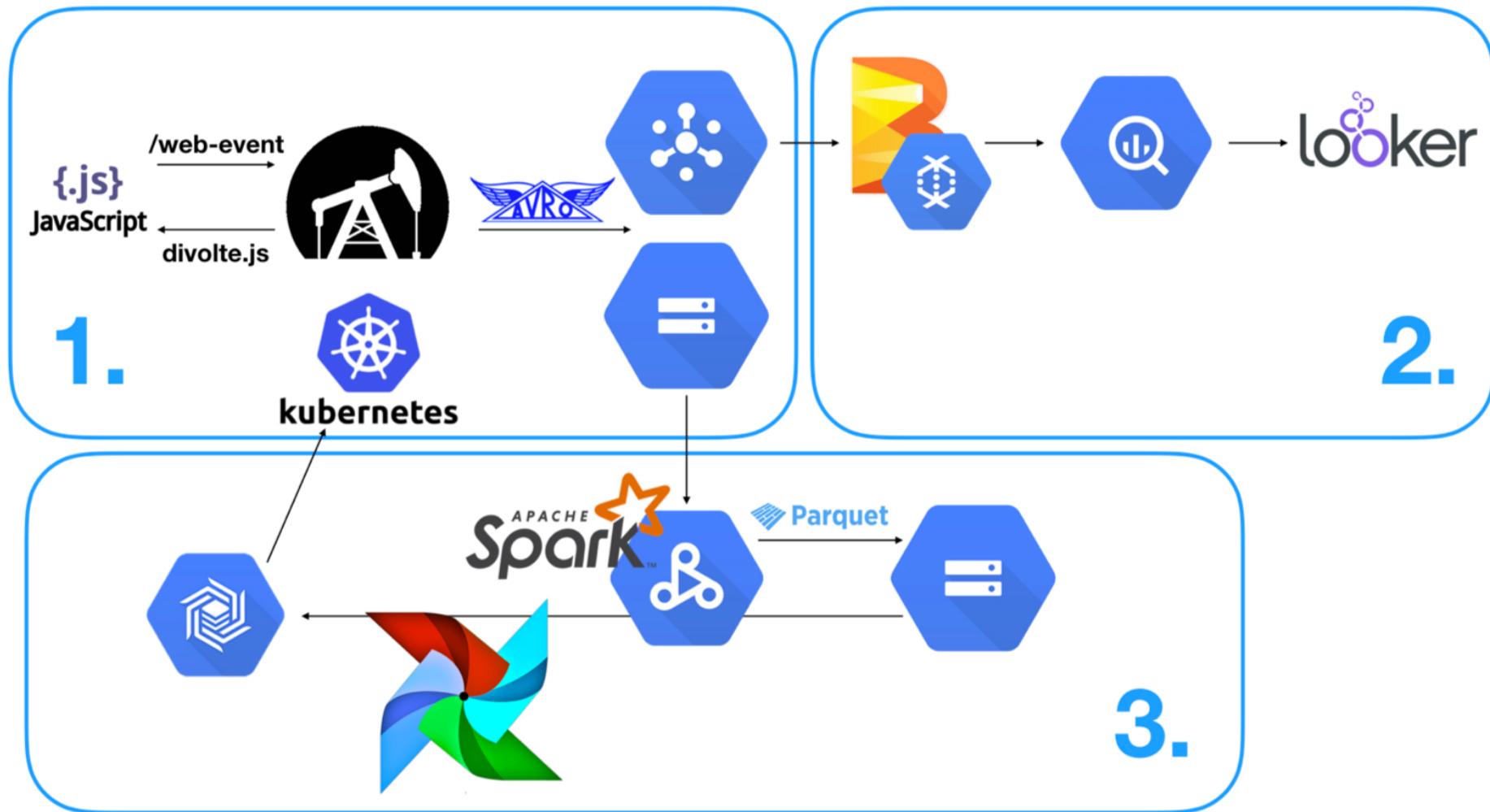
GO
DATA
DRIVEN

User streaming events

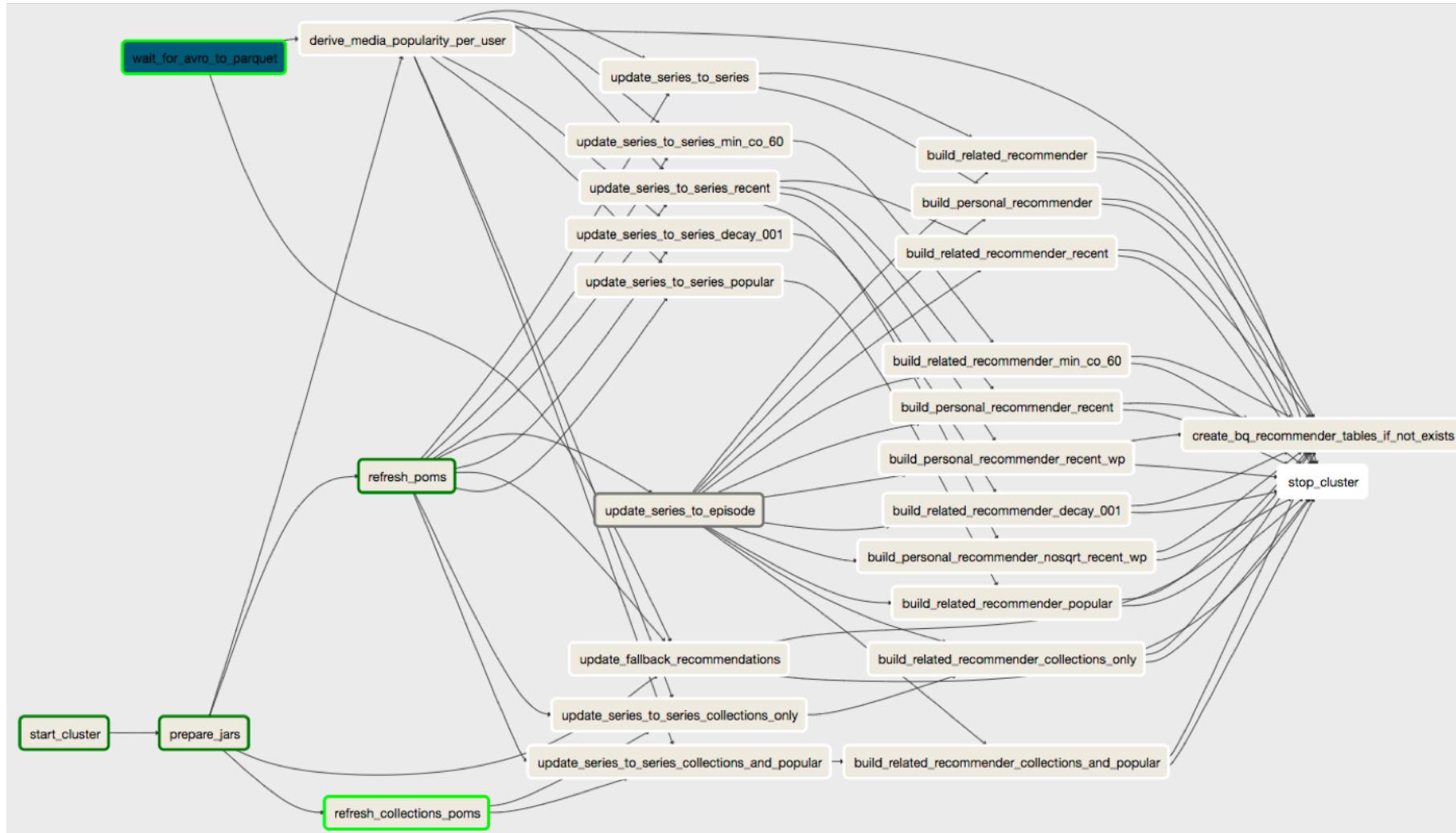


The logo consists of the words "GO", "DATA", and "DRIVEN" stacked vertically. The letters are bold and sans-serif. "GO" is dark blue, "DATA" is light blue, and "DRIVEN" is dark blue. To the right of the text is a stylized teal arrow pointing upwards and to the right.

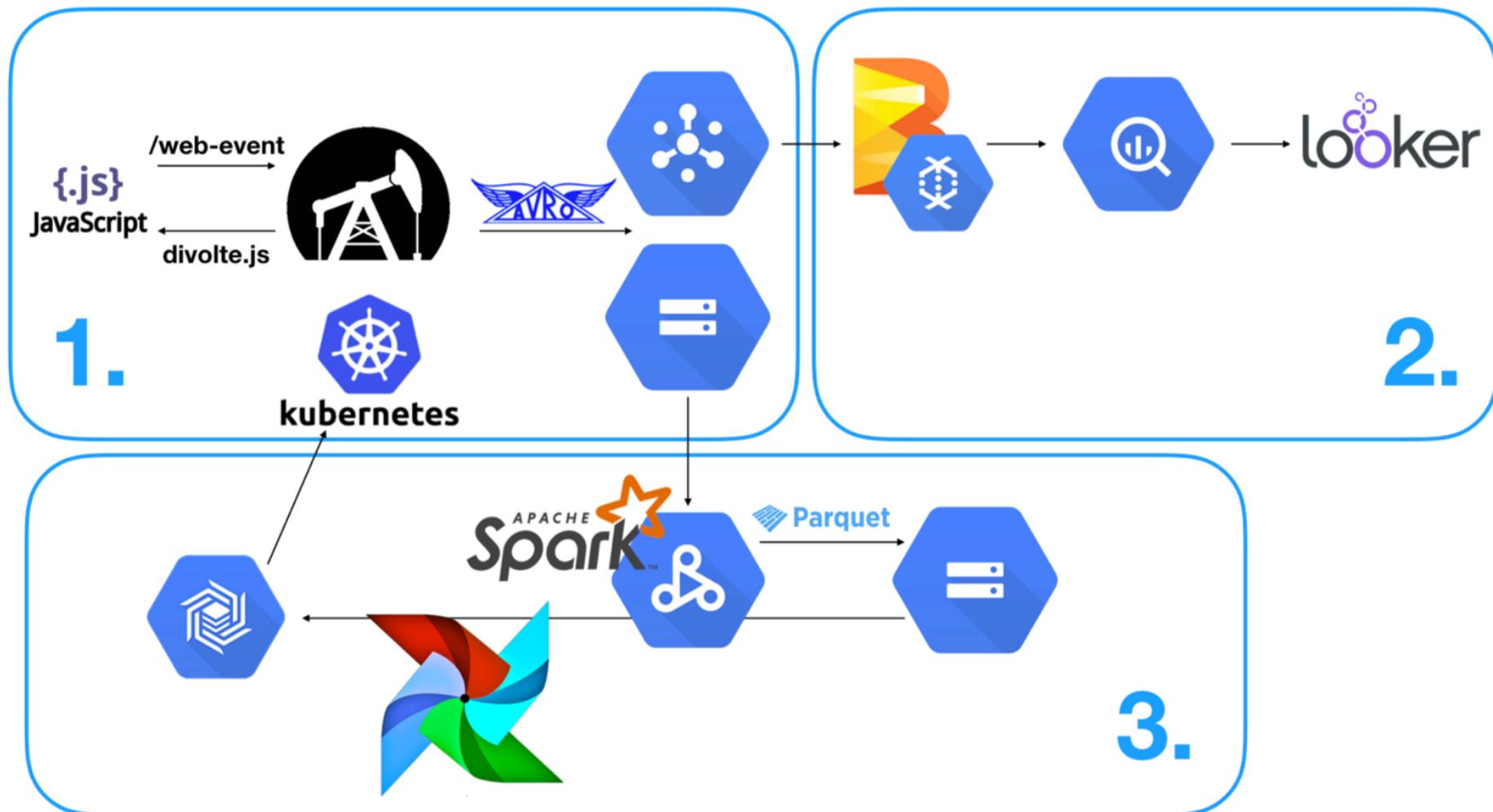
Architecture

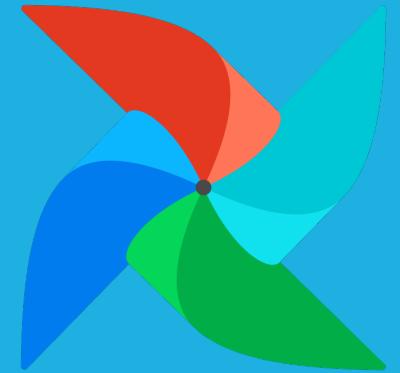


Recommender DAG



Architecture





Setting up the environment

GO 
DATA
DRIVEN

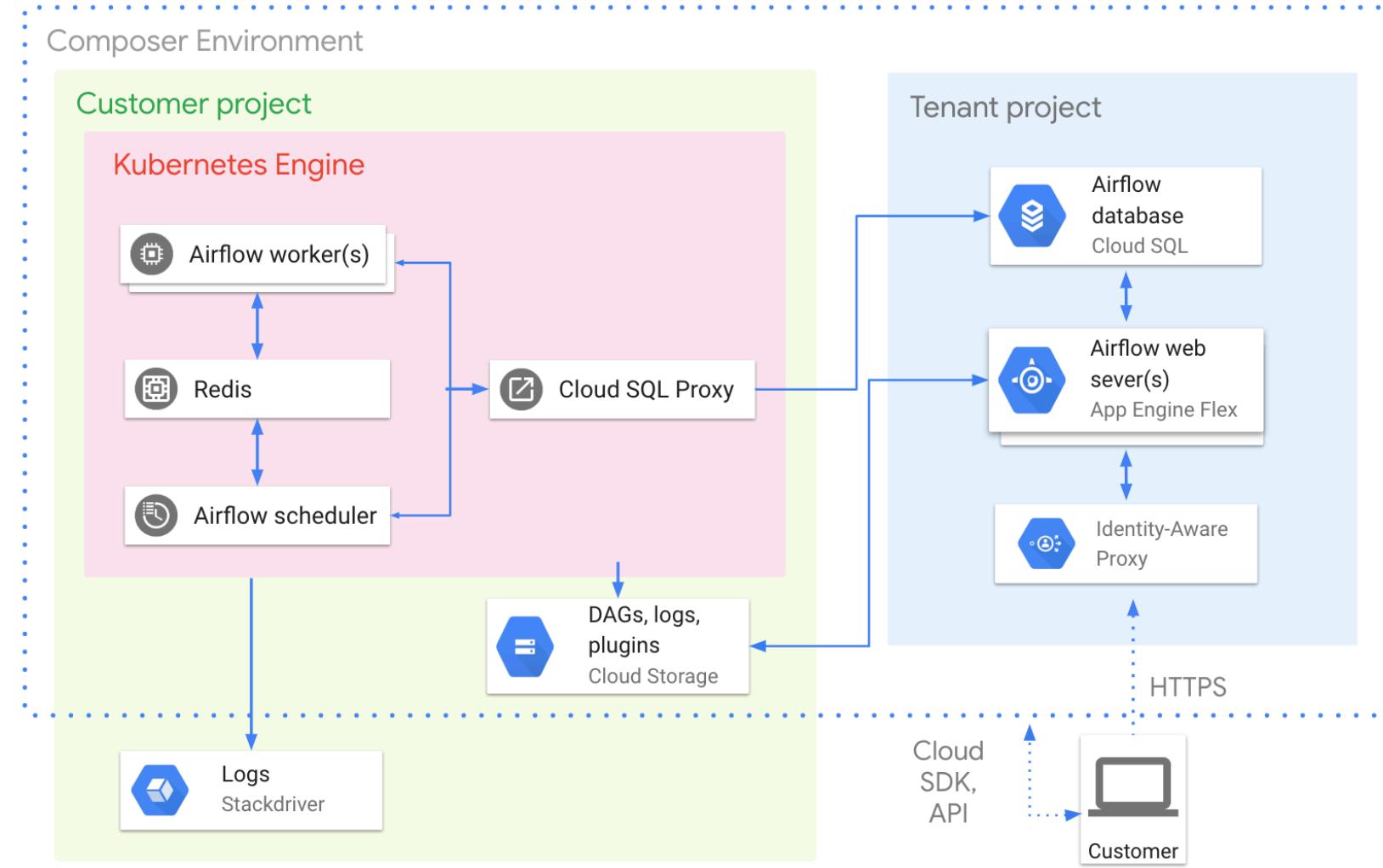
Google Cloud Composer

- Managed Apache Airflow on Google Cloud Platform
- <https://cloud.google.com/composer/docs?hl=nl>
- Takes some time to spin up



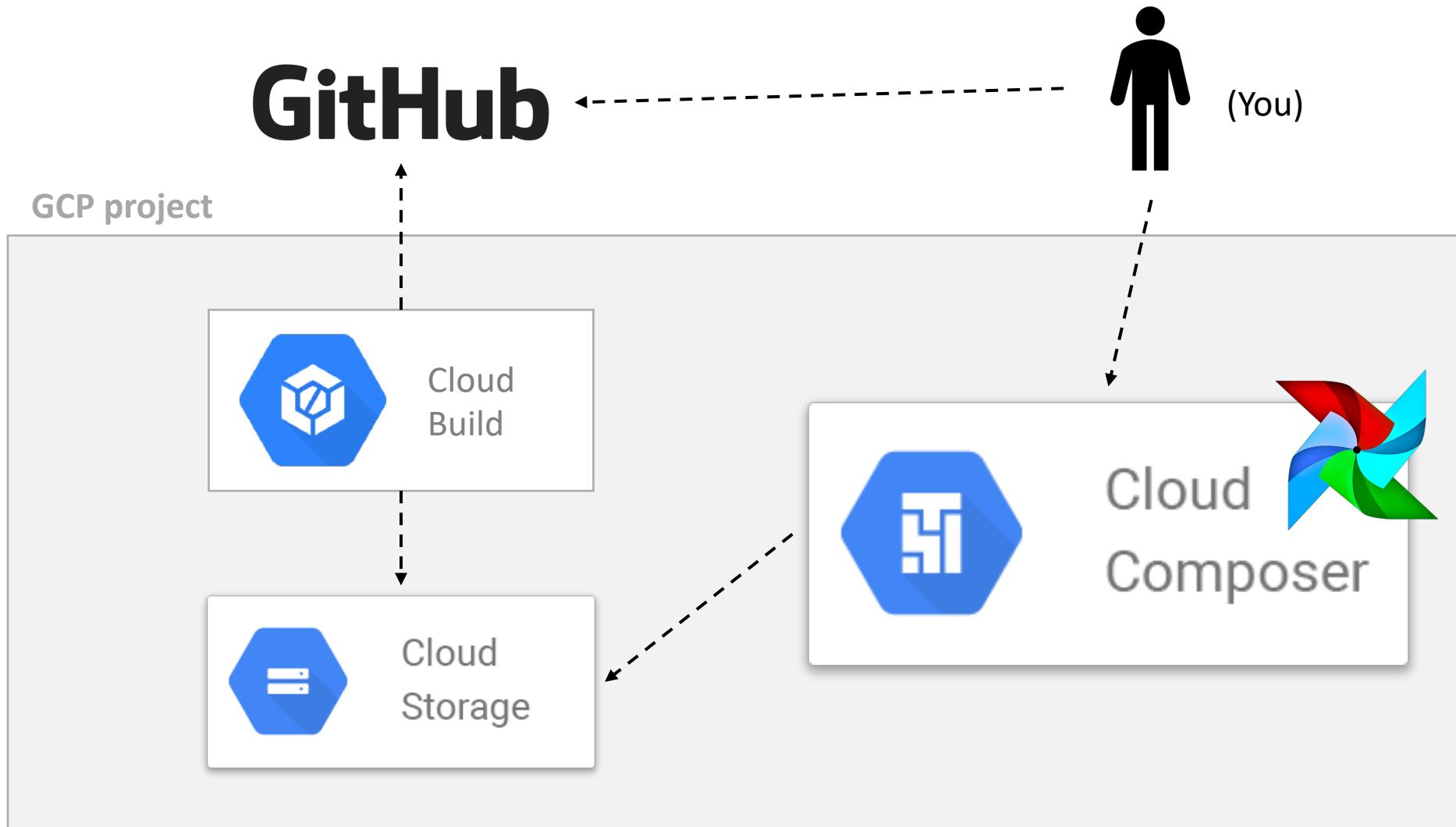
GO 
DATA
DRIVEN

Google Cloud Composer



GO
DATA
DRIVEN

Training environment



Opening Google Composer



- Go to <https://console.cloud.google.com>
- Open the project named “Training Airflow - [name]”
- Go to the Composer service (Google-managed Airflow)
- Docs: <https://cloud.google.com/composer/docs>

Screenshot of the Google Cloud Platform Composer interface:

Google Cloud Platform Training Airflow - bas ▾

Composer Environments + CREATE - DELETE

Filter environments

<input type="checkbox"/>	Name	Location	Creation time	Update time	Airflow webserver	Logs	DAGs folder	Labels
<input checked="" type="checkbox"/>	training-airflow	europe-west1	30/01/2020, 21:10	30/01/2020, 21:33	Airflow	Logs	DAGs	None

GO
DATA
DRIVEN

Setting up code deployment

- Fork <https://github.com/godatadriven/airflow-training-skeleton>

The screenshot shows the GitHub repository page for 'godatadriven / airflow-training-skeleton'. The page includes the repository name, a fork button (circled in red), and various navigation links like Code, Issues, Pull requests, Projects, Wiki, Security, Insights, and Settings. Below the navigation bar, there's a dropdown for the 'master' branch, a 'Code' button (also circled in red), and a list of recent commits. On the right side, there's an 'About' section describing the repository as a skeleton project for Apache Airflow training participants.

godatadriven / **airflow-training-skeleton**

Unwatch 7 | Star 8 | Fork 81

Code Issues Pull requests Projects Wiki Security Insights Settings

master ▾ Go to file Add file ▾ Code ▾

BasPH committed a91365d on 30 Jan 107 commits 2 branches 0 tags

dags Remove useless files 5 months ago

exercise_launch Provide launch dag because copy from presentatio... 5 months ago

k8s Link PVs to PVCs in a simpler way 8 months ago

other Add code for the spark and dataflow job for the use... 6 months ago

About

Skeleton project for Apache Airflow training participants to work on.

Readme Apache-2.0 License

Setting up code deployment

- Open the Cloud Build service in GCP
 - <https://console.cloud.google.com/cloud-build/>
- Connect Cloud Build to your GitHub account (Triggers).
- Create a build trigger linked to your repo.



Screenshot of the Google Cloud Platform Dashboard showing the Cloud Build service.

The dashboard shows a successful build trigger named "Geslaagd: Fokko/airflow-training-skeleton - default-push-trigger-1".

Nieuwste build	Duur	Beschrijving van de trigger	Bron	Toewijzi...
08-07-2020 19:25	00:00:52	Push to any branch	Fokko/airflow-training-skel...	964da7d...

Connect to your GitHub account

The screenshot shows the 'Cloud Build' interface with the 'Triggers' tab selected. The main title is 'Connect repository'. Below it, a four-step process is shown: 1 Select source, 2 Authenticate, 3 Select repository, 4 Create a push trigger (optional). The first step, 'Select source', is active. It displays three options: 'GitHub (Cloud Build GitHub App)' (selected, indicated by a blue radio button), 'Bitbucket (mirrored)', and 'Cloud Source Repositories'. A note below the GitHub option says 'Pull request triggers are supported. Build status will be posted back to GitHub.' A 'BETA' badge is next to the GitHub option. A link 'here' is provided for Cloud Source Repositories. A 'Show more options' link is also present. A note at the bottom states: 'You will be asked to authorize the Google Cloud Build GitHub App to access your GitHub Account to proceed. You may revoke access through GitHub at any time.' At the bottom are 'Continue' and 'Cancel' buttons.

Cloud Build

← Connect repository

History Triggers Settings

1 Select source 2 Authenticate 3 Select repository 4 Create a push trigger (optional)

Select your source

GitHub (Cloud Build GitHub App) **BETA**
Pull request triggers are supported. Build status will be posted back to GitHub.

Bitbucket (mirrored)

Cloud Source Repositories
These repositories are already connected. Create triggers for them [here](#).

[>Show more options](#)

You will be asked to authorize the Google Cloud Build GitHub App to access your GitHub Account to proceed. You may revoke access through GitHub at any time.

Continue Cancel

Select repository to connect

The screenshot shows the 'Cloud Build' interface with the 'Triggers' tab selected. The main title is 'Connect repository'. A progress bar indicates steps 1 ('Select source') and 2 ('Authenticate') are completed, while step 3 ('Select repository') is currently active. Step 4 ('Create a push trigger (optional)') is shown as the next step. A message box states: 'The GitHub App is not installed on any of your repositories' and contains a blue 'Install Google Cloud Build' button. Below the message are 'Connect repository' and 'Cancel' buttons.

Cloud Build

History

Triggers

Settings

← Connect repository

1 Select source 2 Authenticate 3 Select repository 4 Create a push trigger (optional)

Select repository

Select the GitHub repositories to connect to Cloud Build. Members of this GCP project will be able to create and run triggers on these repositories.

The GitHub App is not installed on any of your repositories

Install Google Cloud Build

Connect repository Cancel

Select repository to connect



Install Google Cloud Build

Install on your personal account Bas Harenslak

All repositories
This applies to all current *and* future repositories.

Only select repositories

Select repositories ▾

Selected 1 repository.

BasPH/airflow-training-skeleton X

...with these permissions:

- ✓ Read access to code
- ✓ Read access to issues, metadata, and pull requests
- ✓ Read and write access to checks and commit statuses

Install Cancel

Next: you'll be directed to the GitHub App's site to complete setup.



GO DATA DRIVEN

Select repository to connect

The screenshot shows the 'Cloud Build' interface with the 'Triggers' tab selected. The main title is 'Connect repository'. The steps are: 1 Select source, 2 Authenticate, 3 Select repository (the current step), 4 Create a push trigger (optional). The 'Select repository' section shows a GitHub account dropdown set to 'BasPH', a 'Filter repositories' input field, and two checked checkboxes: 'Select all repositories' and 'airflow-training-skeleton'. A note states: 'I understand that GitHub content for the selected repositories will be transferred to this GCP project to provide the connected service. Members of this GCP project with sufficient permissions will be able to create and run triggers on these repositories, based on transferred GitHub content.' At the bottom are 'Connect repository' and 'Cancel' buttons.

Cloud Build

History

Triggers

Settings

← Connect repository

1 Select source 2 Authenticate 3 Select repository 4 Create a push trigger (optional)

Select repository

Select the GitHub repositories to connect to Cloud Build. Members of this GCP project will be able to create and run triggers on these repositories.

GitHub Account

BasPH

Filter repositories

Select all repositories [Edit repositories on GitHub](#)

airflow-training-skeleton

I understand that GitHub content for the selected repositories will be transferred to this GCP project to provide the connected service. Members of this GCP project with sufficient permissions will be able to create and run triggers on these repositories, based on transferred GitHub content.

Connect repository Cancel

Create a push trigger

The screenshot shows the Google Cloud Build interface. On the left, there's a sidebar with icons for Cloud Build, History, Triggers (which is selected and highlighted in blue), and Settings. The main area is titled "Connect repository" and shows four steps completed: "Select source", "Authenticate", "Select repository", and "Create a push trigger (optional)". The "Create a push trigger (optional)" section contains a table with one row:

Description	Event
Push to any branch	Push to branch

Below this, there's a section titled "Create a push trigger for these repositories:" with a "Filter repositories" input field and two checked checkboxes: "Select all repositories" and "airflow-training-skeleton". At the bottom are two buttons: "Create push trigger" (in blue) and "Skip for now".

Done!

Cloud Build

Triggers BETA CONNECT REPOSITORY + CREATE TRIGGER

History Triggers ACTIVE INACTIVE Filter repositories

BasPH/airflow-training-skeleton

Description	Event	Filter	Build configuration	Status	Run trigger
Push to any branch	Push to branch	.*	Auto-detected	Enabled	⋮

Cloud Build GitHub App ⋮ ^

Cloud Build Edit trigger DISABLE DELETE

History Triggers ACTIVE INACTIVE

Name Must be unique within the project Push-to-any-branch

Description Push to any branch

Trigger type Branch (radio button selected)

Branch (regex) Matches 2 branches: launch-dag, master

Invert Regex

* Filter builds by changed files

Build configuration Auto-detected (radio button selected)

A cloudbuild.yaml or Dockerfile will be detected in the repository

Dockerfile Specify the path within the Git repo

Cloud Build configuration file (yaml or json) Specify the path to a Cloud Build configuration file in the Git Repo Learn more

Save Cancel

GO 
DATA
DRIVEN

Done!

- We now trigger a pipeline on every push
- cloudbuild.yaml is automatically detected
- You can test by triggering manually
- Or, make a commit and push:



Cloud Build	Build history STOP STREAMING BUILDS						
History	Filter builds						
Triggers	Build	Source	Ref	Commit	Trigger name	Created	Duration
	e7aeee82	BasPH/airflow-training-skeleton	master	ce59c83	Push-to-any-branch	30/01/2020, 21:53	8 sec

GO
DATA
DRIVEN

Note



- I'll leave the environment running a hour after the training
- Don't waste resources
- Free 300\$ credits for 12 months:
 - <https://cloud.google.com/free/docs/gcp-free-tier>

Exercise

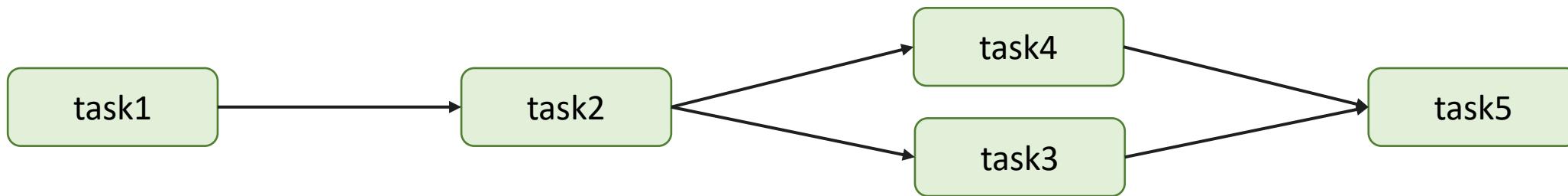
- Configure the trigger
- Open the Airflow webserver from Composer
- Do you see the freshly deployed example_bash_operator dag?
- Try exploring the Airflow web UI
- What do you find?

Airflow – Getting started

GO 
DATA
DRIVEN

Airflow concepts - DAGs

- Airflow workflows are created using DAGs
 - DAG = Directed Acyclic Graph
 - Tasks as nodes, dependencies as edges
- DAGs are defined in Python code



Airflow concepts - Operators

- Tasks are executed using Operators
 - Allow you to execute tasks on different systems
 - Abstract logic of interacting with these systems
- Many built-in operators
 - BashOperator – execute a Bash command
 - PythonOperator – execute a Python function
 - EmailOperator – send an e-mail

An example DAG

```
Imports {  
    import airflow  
    from airflow.models import DAG  
    from airflow.operators.bash_operator import BashOperator  
  
Configuration {  
    args = {  
        "owner": "airflow",  
        "start_date": airflow.utils.dates.days_ago(2),  
    }  
  
DAG initialization {  
    dag = DAG(  
        dag_id="hello_dag",  
        default_args=args,  
        description="Demo DAG showing a hello world example.",  
    )  
  
Tasks {  
    t1 = BashOperator(task_id="sleep_a_bit", bash_command="sleep 5", dag=dag)  
    t2 = BashOperator(task_id="print_date", bash_command="date", dag=dag)  
  
Dependencies {  
    t1 >> t2
```

DAGs – Two different styles

- Explicit reference

```
dag = DAG(...)  
t1 = DummyOperator(task_id="task1", dag=dag)  
t2 = DummyOperator(task_id="task2", dag=dag)
```

- Implicit reference

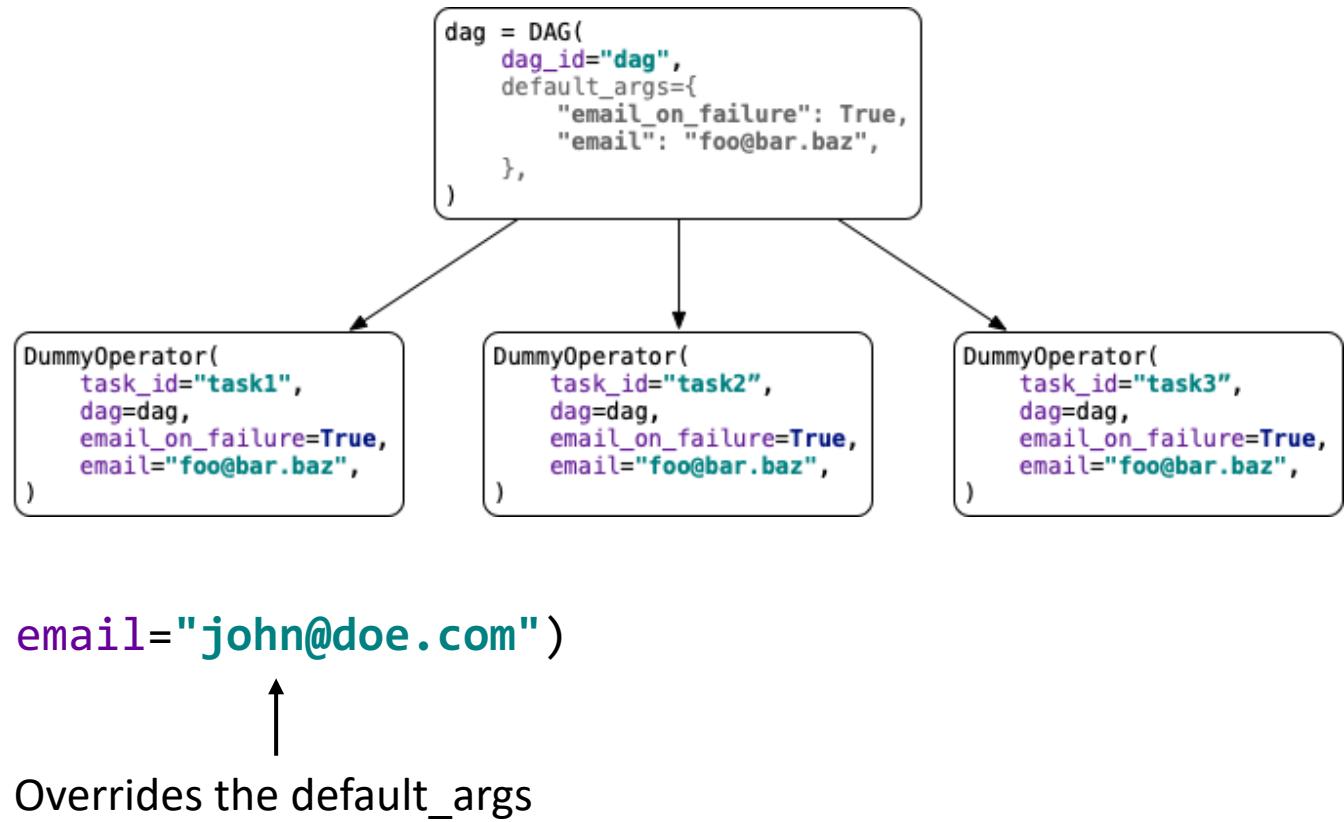
```
with DAG(...) as dag:  
    t1 = DummyOperator(task_id="task1")  
    t2 = DummyOperator(task_id="task2")
```

DAGs – default_args

```
from airflow.models import DAG
from airflow.operators.dummy_operator import DummyOperator
```

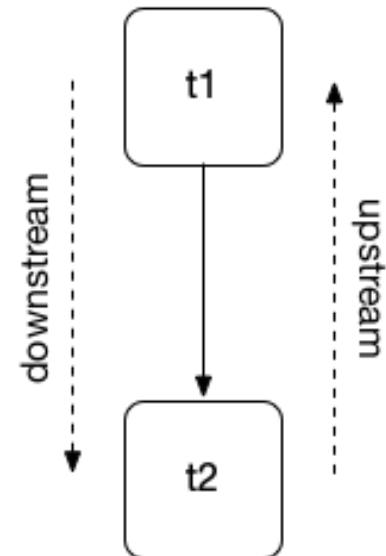
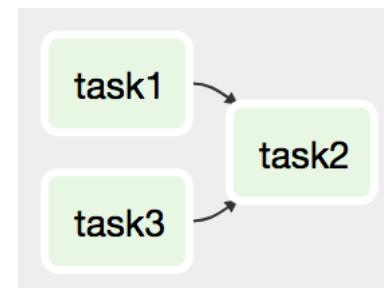
```
dag = DAG(
    dag_id="dag",
    default_args={
        "email_on_failure": True,
        "email": "foo@bar.baz",
    },
)
```

```
DummyOperator(task_id="task1", dag=dag)
DummyOperator(task_id="task2", dag=dag)
DummyOperator(task_id="task3", dag=dag, email="john@doe.com")
```



DAGs – defining dependencies

- Chaining multiple tasks:
 - `task1 >> task2 >> task3`
- Other direction is also possible:
 - `task3 << task2 << task1`
- Combinations are also possible:
 - `task1 >> task2 << task3`
- Is the same as:
 - `task1 >> task2`
 - `task3 >> task2`



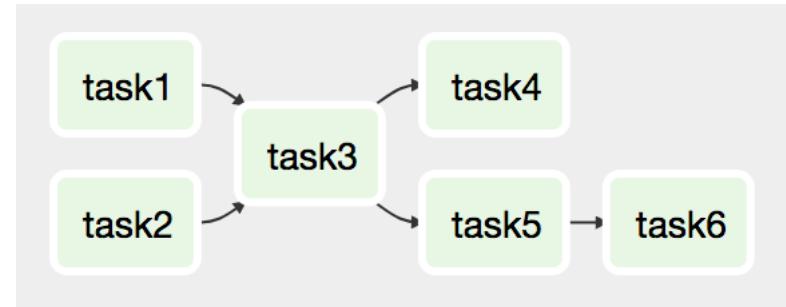
DAGs – defining dependencies

- Is also the same as:
 - [t1, t3] >> t2

- More complex examples require multiple lines:

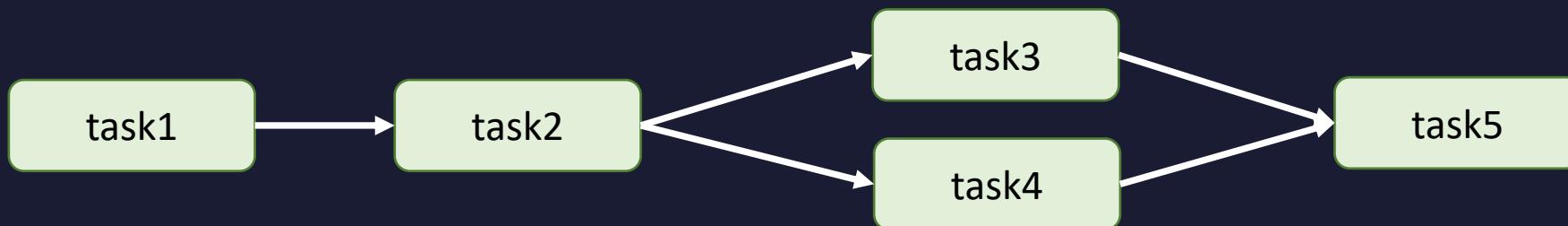
```
t1 = DummyOperator(task_id="task1", dag=dag)
t2 = DummyOperator(task_id="task2", dag=dag)
t3 = DummyOperator(task_id="task3", dag=dag)
t4 = DummyOperator(task_id="task4", dag=dag)
t5 = DummyOperator(task_id="task5", dag=dag)
t6 = DummyOperator(task_id="task6", dag=dag)
```

```
[t1, t2] >> t3 >> t4
t3 >> t5 >> t6
```

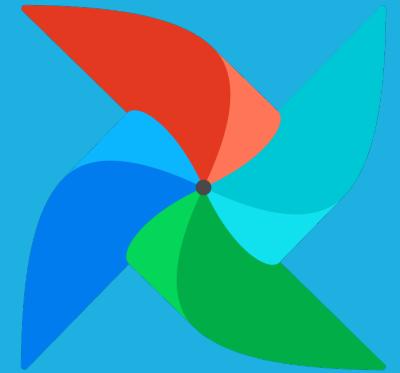


First Exercise

- Create a first DAG, that looks as follows:



- Use the DummyOperator + schedule_interval = None
- Try triggering a (manual) run from the Airflow UI

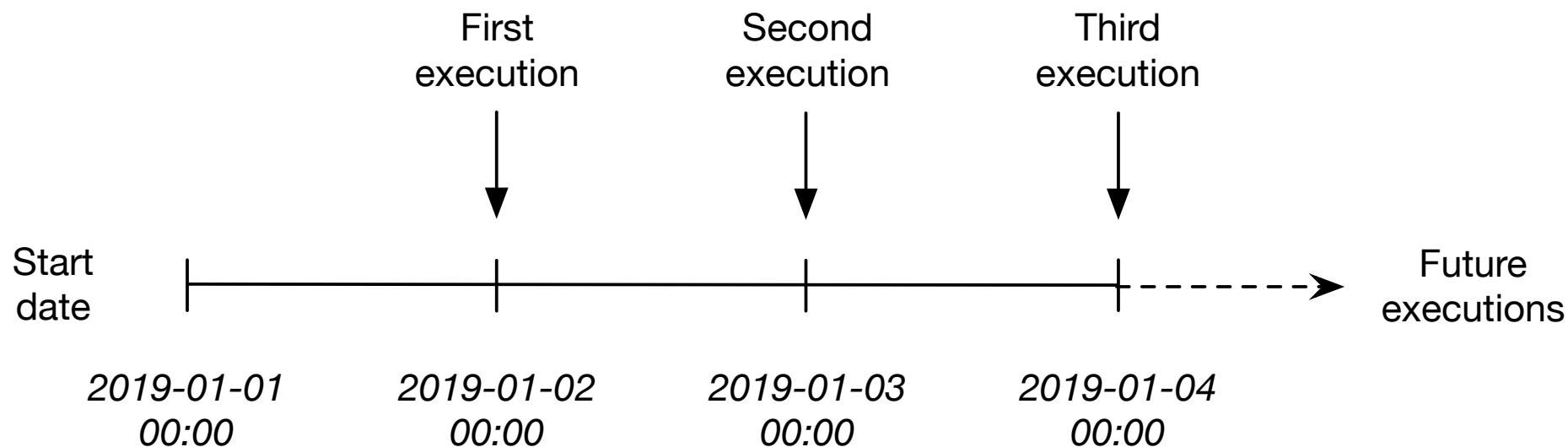


Airflow – Scheduling

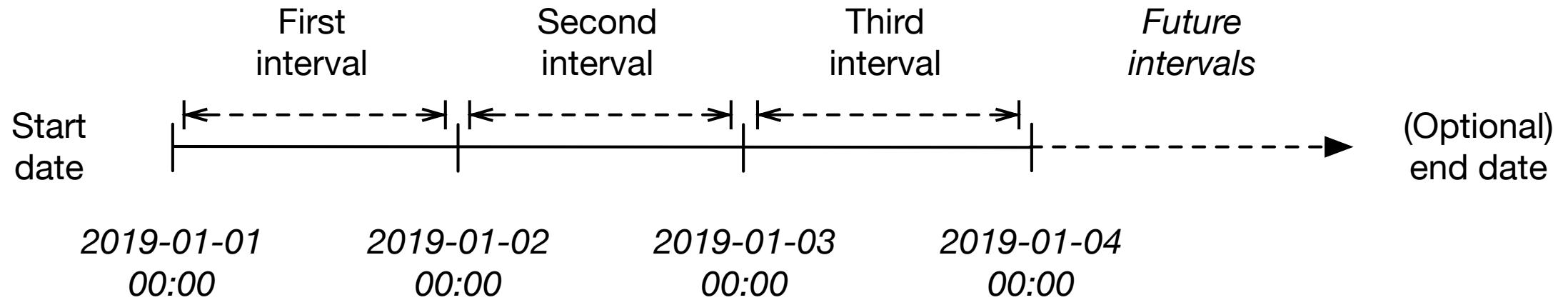
GO 
DATA
DRIVEN

Schedule intervals

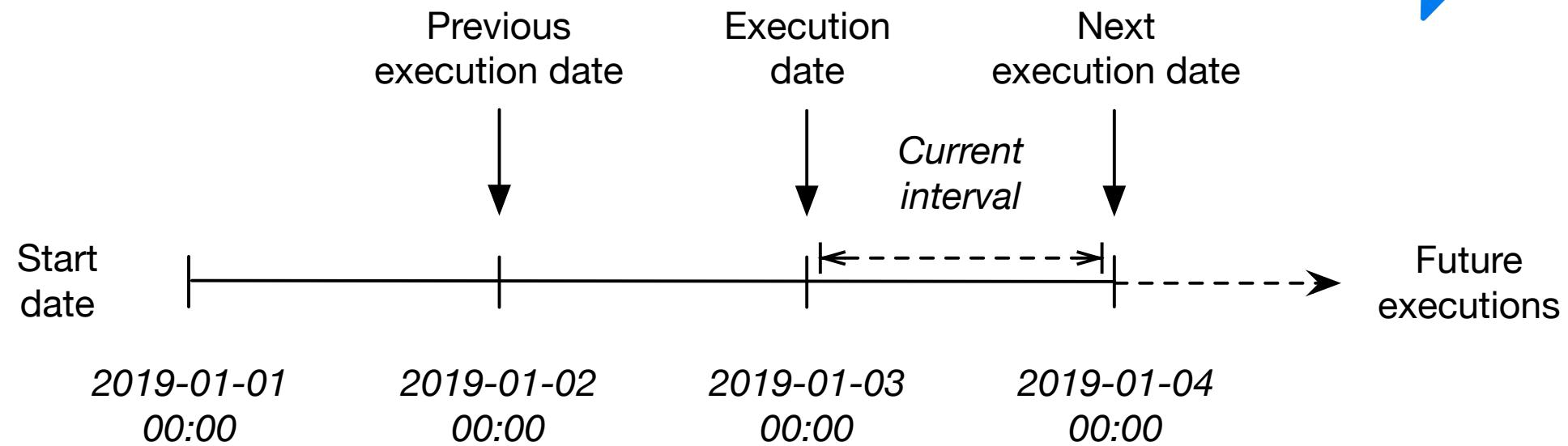
```
dag = DAG(  
    dag_id="demo",  
    description="Showing execution intervals.",  
    start_date=datetime.datetime(2019, 1, 1),  
    schedule_interval="@daily",  
)
```



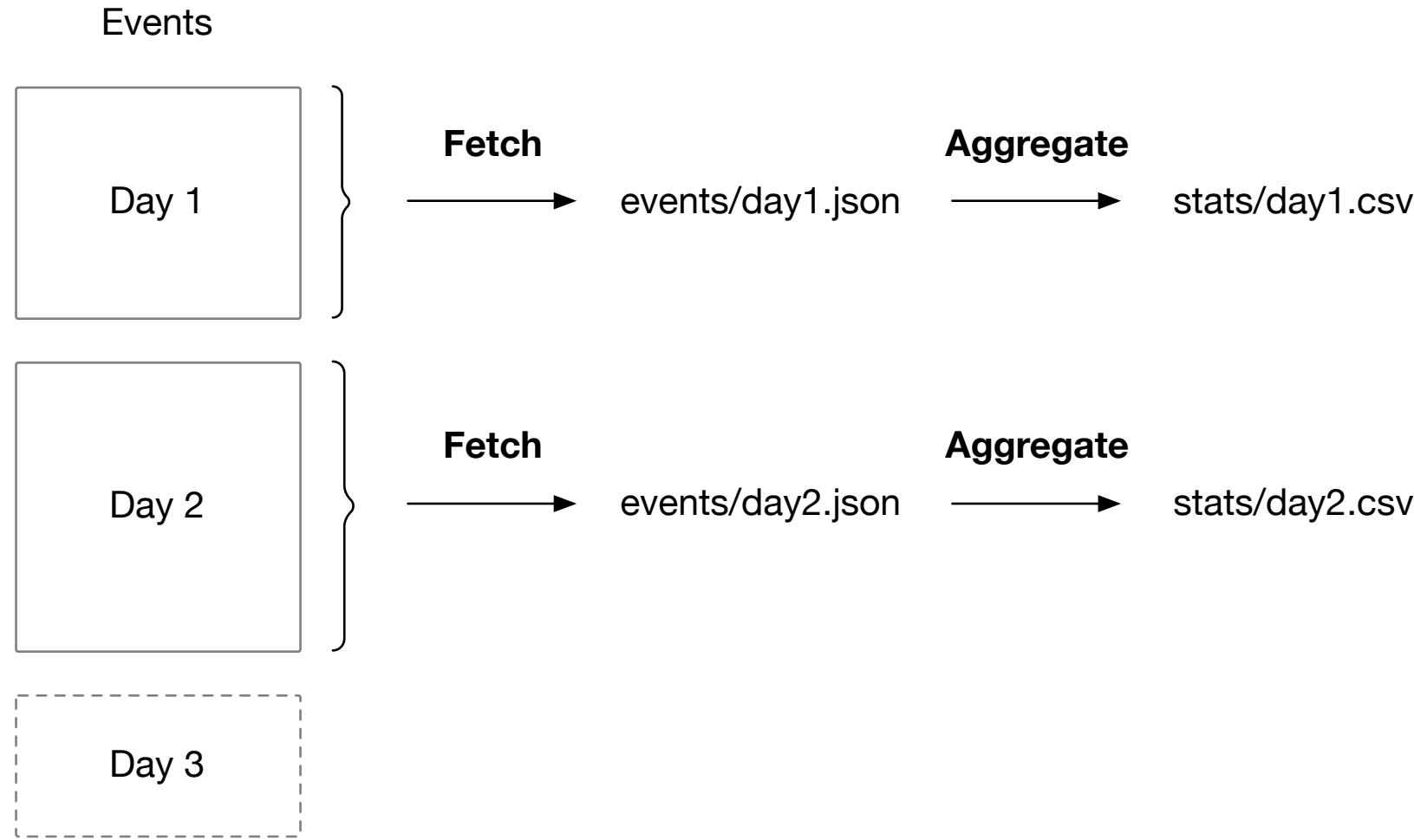
Schedule intervals



Schedule intervals - dates



Using intervals for incremental loads



Things to avoid

- Changing start date or schedule interval
 - Scheduler will get confused
 - Better to create new version of DAG
- Dynamic dates
 - Dates move along as time progresses
 - May come to never pass, or not line up with existing runs

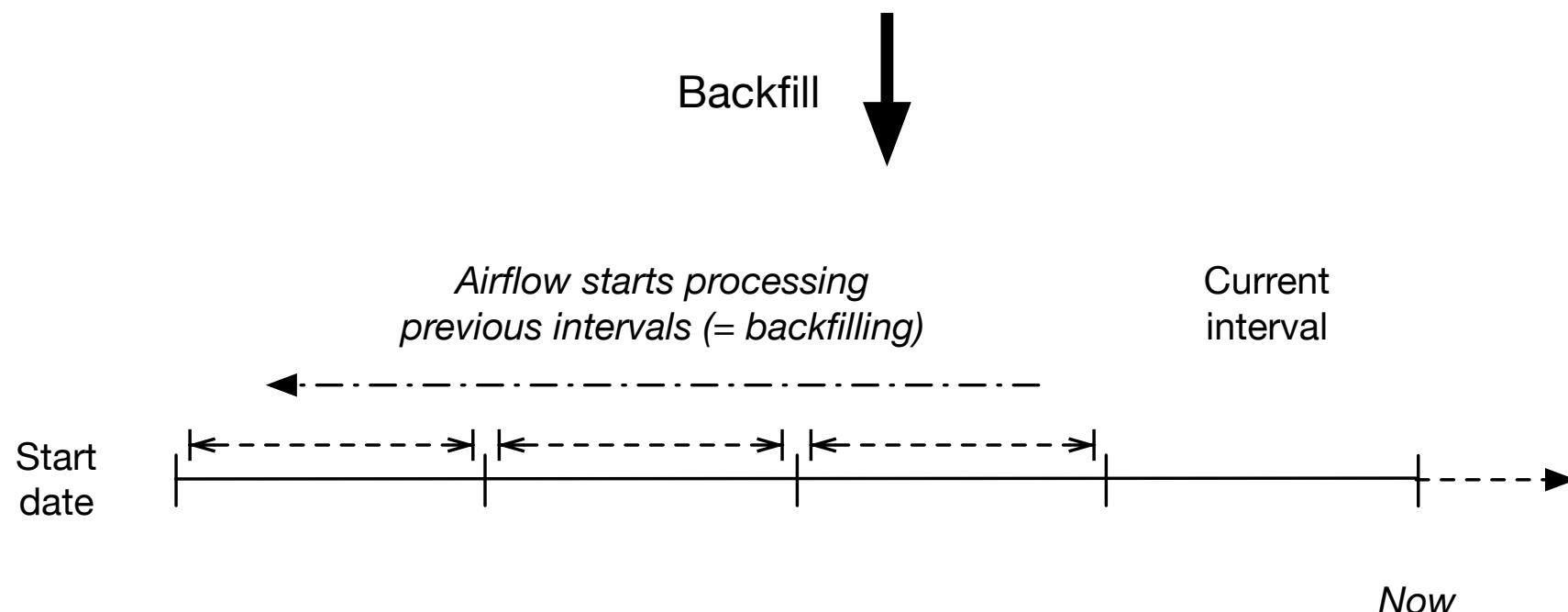
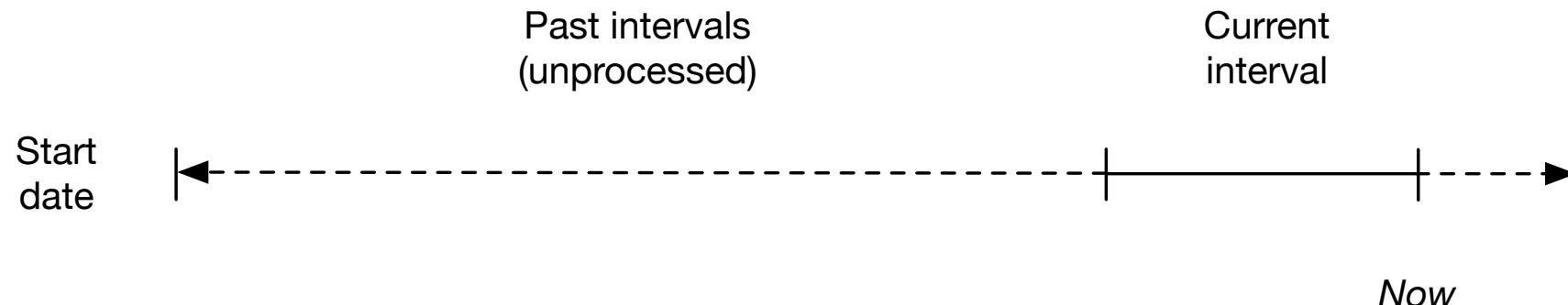


Backfilling

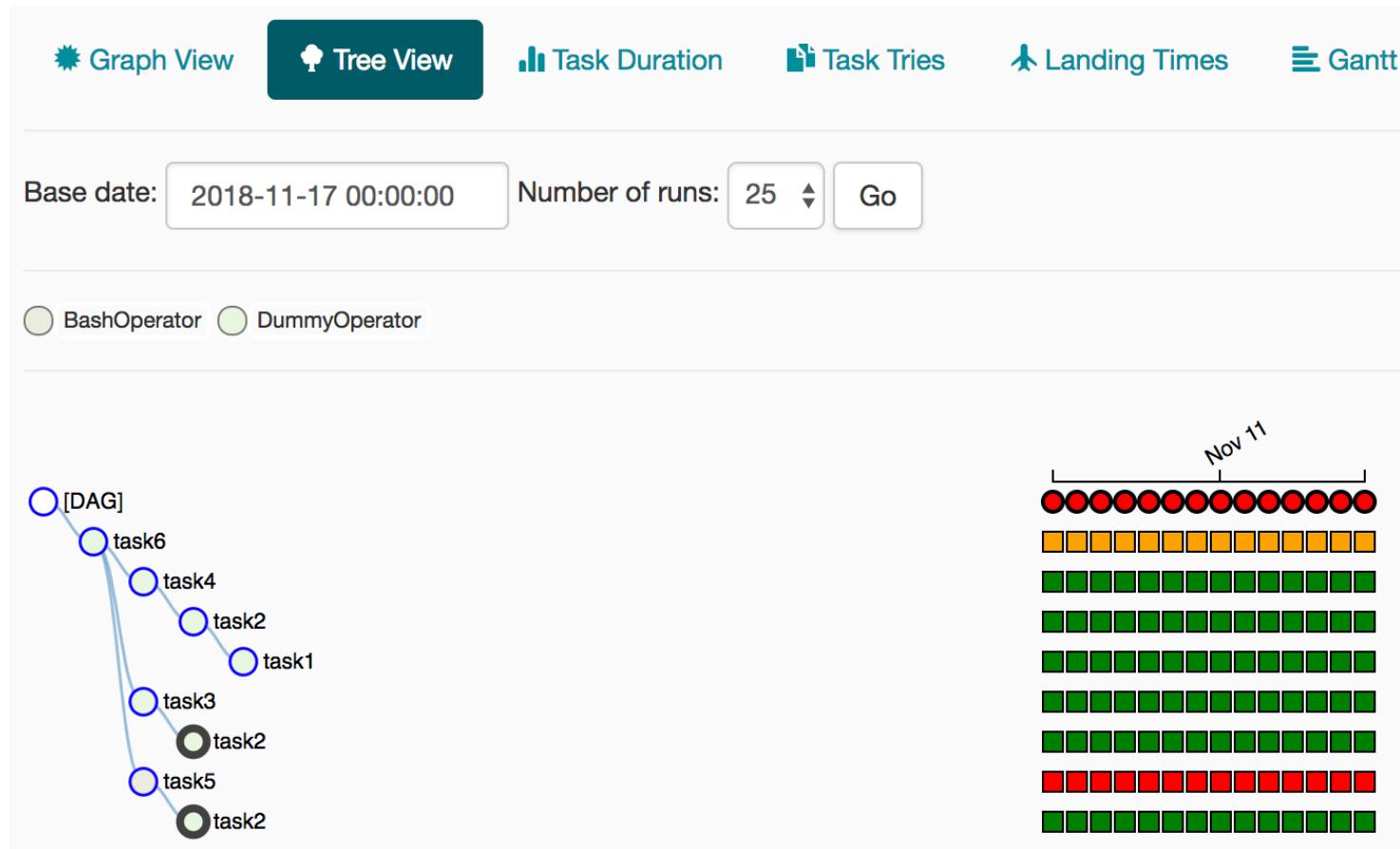
- Backfilling is the concept of (re-)running Airflow tasks back in time.
- For example:
 - You created a daily job and want to run it 1 year back.
 - A task failed due to a missing file. You placed the file manually and now you want to re-run the task.
 - You changed code in a DAG and want to re-run specific tasks with the changed code.



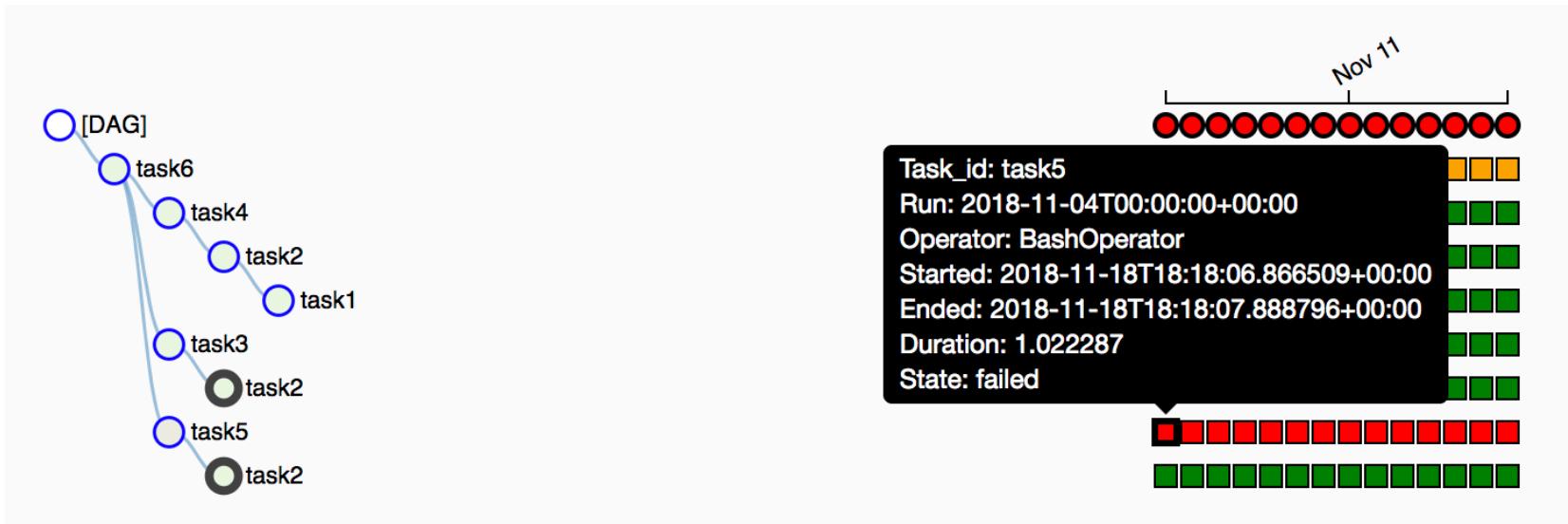
Backfilling



Backfilling - Example



Backfilling - Example



Backfilling - Example

task5  on 2018-11-04T00:00:00+00:00

Task Instance Details

Rendered

Task Instances

View Log

Run

Ignore All Deps

Ignore Task State

Ignore Task Deps

Clear

Past

Future

Upstream

Downstream

Recursive

Mark Success

Past

Future

Upstream

Downstream

Backfilling - Example

Wait a minute.

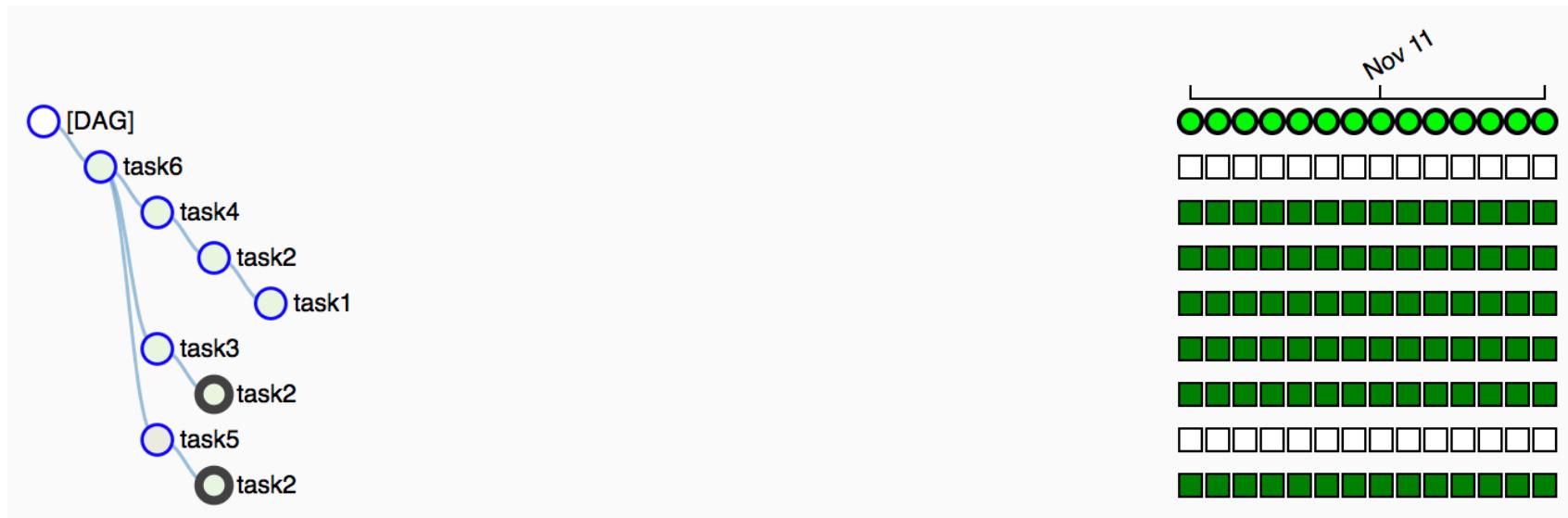
Here's the list of task instances you are about to clear:

```
<TaskInstance: hello_dag.task5 2018-11-04 00:00:00+00:00 [failed]>
<TaskInstance: hello_dag.task5 2018-11-05 00:00:00+00:00 [failed]>
<TaskInstance: hello_dag.task5 2018-11-06 00:00:00+00:00 [failed]>
<TaskInstance: hello_dag.task5 2018-11-07 00:00:00+00:00 [failed]>
<TaskInstance: hello_dag.task5 2018-11-08 00:00:00+00:00 [failed]>
<TaskInstance: hello_dag.task5 2018-11-09 00:00:00+00:00 [failed]>
<TaskInstance: hello_dag.task5 2018-11-10 00:00:00+00:00 [failed]>
<TaskInstance: hello_dag.task5 2018-11-11 00:00:00+00:00 [failed]>
<TaskInstance: hello_dag.task5 2018-11-12 00:00:00+00:00 [failed]>
<TaskInstance: hello_dag.task5 2018-11-13 00:00:00+00:00 [failed]>
<TaskInstance: hello_dag.task5 2018-11-14 00:00:00+00:00 [failed]>
<TaskInstance: hello_dag.task5 2018-11-15 00:00:00+00:00 [failed]>
<TaskInstance: hello_dag.task5 2018-11-16 00:00:00+00:00 [failed]>
<TaskInstance: hello_dag.task5 2018-11-17 00:00:00+00:00 [failed]>
<TaskInstance: hello_dag.task6 2018-11-04 00:00:00+00:00 [upstream_failed]>
<TaskInstance: hello_dag.task6 2018-11-05 00:00:00+00:00 [upstream_failed]>
<TaskInstance: hello_dag.task6 2018-11-06 00:00:00+00:00 [upstream_failed]>
<TaskInstance: hello_dag.task6 2018-11-07 00:00:00+00:00 [upstream_failed]>
<TaskInstance: hello_dag.task6 2018-11-08 00:00:00+00:00 [upstream_failed]>
<TaskInstance: hello_dag.task6 2018-11-09 00:00:00+00:00 [upstream_failed]>
<TaskInstance: hello_dag.task6 2018-11-10 00:00:00+00:00 [upstream_failed]>
<TaskInstance: hello_dag.task6 2018-11-11 00:00:00+00:00 [upstream_failed]>
<TaskInstance: hello_dag.task6 2018-11-12 00:00:00+00:00 [upstream_failed]>
<TaskInstance: hello_dag.task6 2018-11-13 00:00:00+00:00 [upstream_failed]>
<TaskInstance: hello_dag.task6 2018-11-14 00:00:00+00:00 [upstream_failed]>
<TaskInstance: hello_dag.task6 2018-11-15 00:00:00+00:00 [upstream_failed]>
<TaskInstance: hello_dag.task6 2018-11-16 00:00:00+00:00 [upstream_failed]>
<TaskInstance: hello_dag.task6 2018-11-17 00:00:00+00:00 [upstream_failed]>
```

OK!

bail.

Backfilling - Example



Designing tasks for backfilling

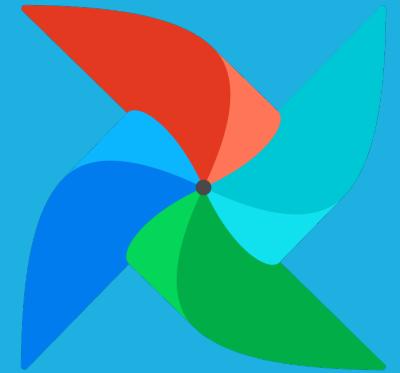
- Tasks should be atomic and idempotent
- Atomic
 - Tasks either succeed fully or not at all (no partial result)
- Idempotent
 - Re-running a task gives the same result
 - (Assuming other circumstances have not changed)



Second Exercise

- Set a daily schedule for the previously created DAG
- Check if your DAG ran successfully using the tree view
 - See if you can inspect individual task results
 - Do you understand the execution dates for each run?
- Can you re-run (backfill) tasks in the past?
- Bonus questions – schedule (a copy of) the DAG to run:
 1. At 13:45 every Mon/Wed/Fri
 2. Every 2,5 hours





Airflow – Templating

GO 
DATA
DRIVEN

The Airflow 'context'

- Airflow provides information about the current execution in the Airflow 'context'
- This includes:
 - The `execution_date` of the DAG run
 - Previous and next execution dates



Jinja templating (string parameters)

- Many operators allow you to include context variables in their (string) parameters using templating:

```
print_exec_date = BashOperator(  
    task_id="demo_templating",  
    bash_command="echo {{ execution_date }}",  
    dag=dag,  
)
```

- Other examples:
 - SQL (database operators)
 - Input/output file paths

Context kwargs (Python)



- The Python operator allows you to access the context using the context keyword-arguments:

```
def _print_exec_date(**context):      # Same result, different notation:  
    print(context["execution_date"])  def _print_exec_date(execution_date: Datetime, **context):  
                                         print(execution_date)
```

```
print_exec_date = PythonOperator(  
    task_id="demo_templating",  
    python_callable=_print_exec_date,  
    provide_context=True,  
    dag=dag,  
)
```

What's in the context?

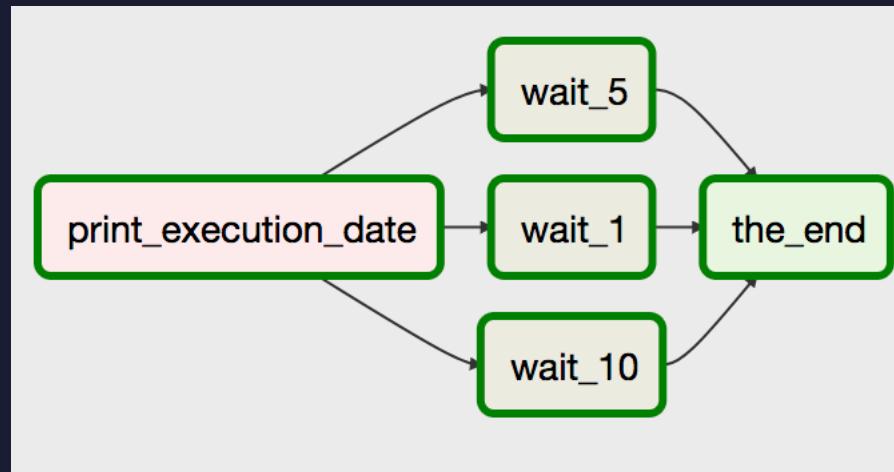
```
{'END_DATE': '2018-01-01',
'conf': <module 'airflow.configuration' from '/opt/conda/lib/python3.6/site-packages/airflow/configuration.py'>,
'dag': <DAG: templated_task_dag>,
'dag_run': None,
'ds': '2018-01-01',
'ds_nodash': '20180101',
'end_date': '2018-01-01',
'execution_date': <Pendulum [2018-01-01T00:00:00+00:00]>,
'inlets': [],
'latest_date': '2018-01-01',
'macros': <module 'airflow.macros' from '/opt/conda/lib/python3.6/site-packages/airflow/macros/__init__.py'>,
'next_ds': '2018-01-02',
'next_execution_date': datetime.datetime(2018, 1, 2, 0, 0, tzinfo=<TimezoneInfo [UTC, GMT, +00:00:00, STD]>),
'outlets': [],
'params': {},
'prev_ds': '2017-12-31',
'prev_execution_date': datetime.datetime(2017, 12, 31, 0, 0, tzinfo=<TimezoneInfo [UTC, GMT, +00:00:00, STD]>),
'run_id': None,
'tables': None,
'task': <Task(PythonOperator): demo_templating>,
'task_instance': <TaskInstance: templated_task_dag.demo_templating 2018-01-01T00:00:00+00:00 [None]>,
'task_instance_key_str': 'templated_task_dag_demo_templating_20180101',
'templates_dict': None,
'test_mode': True,
'ti': <TaskInstance: templated_task_dag.demo_templating 2018-01-01T00:00:00+00:00 [None]>,
'tomorrow_ds': '2018-01-02',
'tomorrow_ds_nodash': '20180102',
'ts': '2018-01-01T00:00:00+00:00',
'ts_nodash': '20180101T000000+0000',
'ver': {'json': None, 'value': None},
'yesterday_ds': '2017-12-31',
'yesterday_ds_nodash': '20171231'}
```

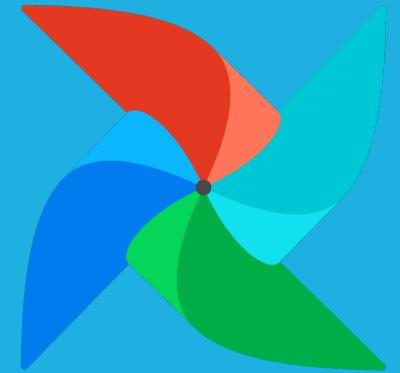


Third Exercise



- Write a DAG with the following stages:
 1. PythonOperator printing execution date
 2. BashOperator doing sleep 1, sleep 5 and sleep 10
 3. DummyOperator to finish it off
- Desired result:



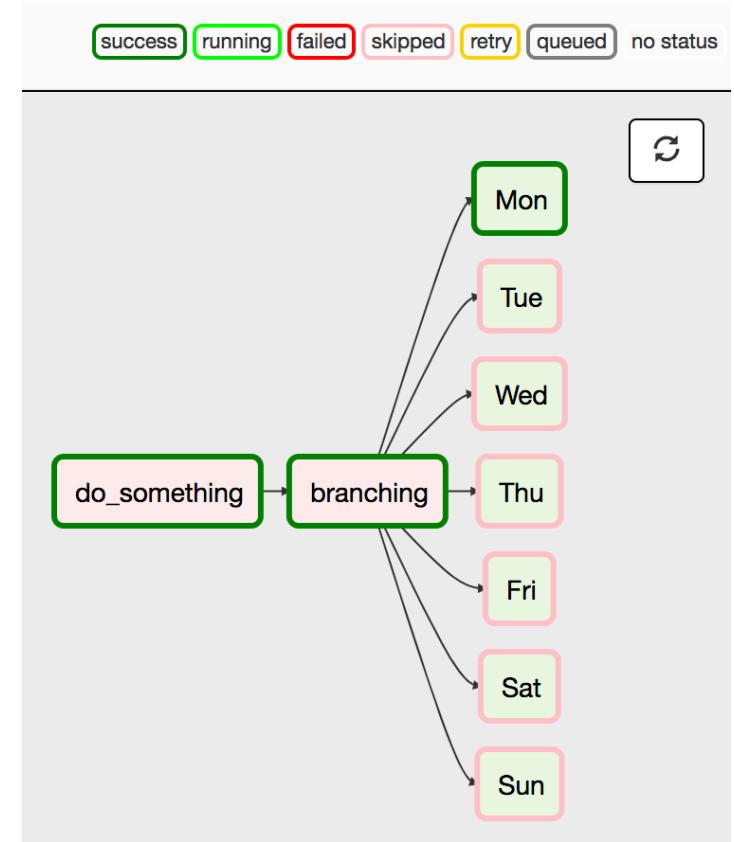


Airflow – Branching

GO 
DATA
DRIVEN

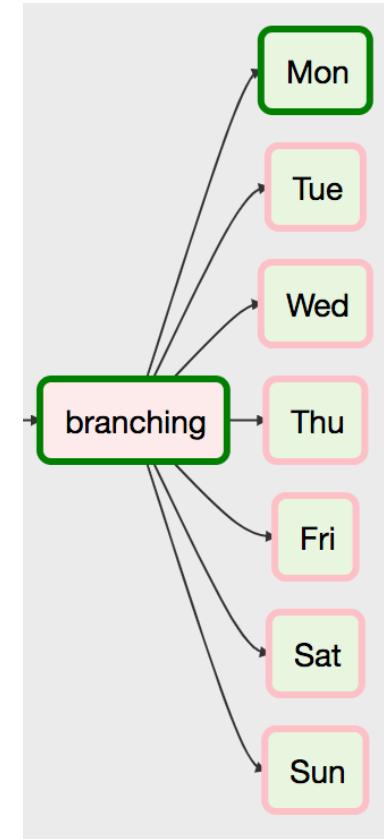
Branching

- Sometimes you'd like to execute some tasks based on a specific condition
- Examples:
 - Running tasks on certain days of the week
 - Running a different set of tasks after a specific date (e.g. to account for a schema change)



The PythonBranchOperator

- Branches on a certain condition
 - Accepts a callable, which when called should return the name of downstream task(s) to run
 - Other tasks are automatically skipped
- Note that tasks should be a (direct) downstream dependency of the branch task

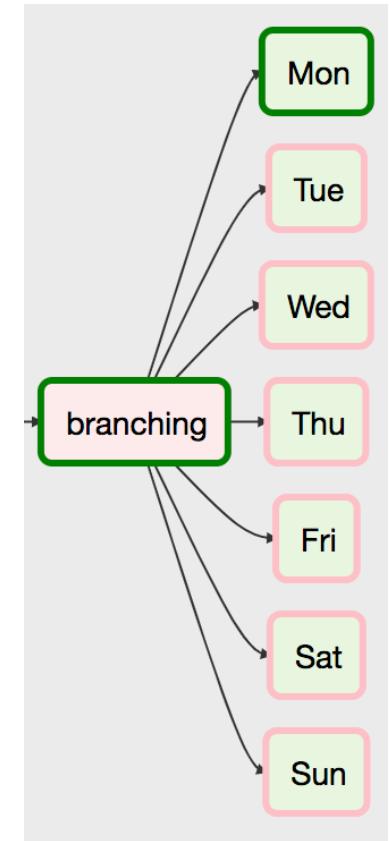


The PythonBranchOperator

```
def _get_weekday(execution_date, **context):
    return execution_date.strftime("%a")

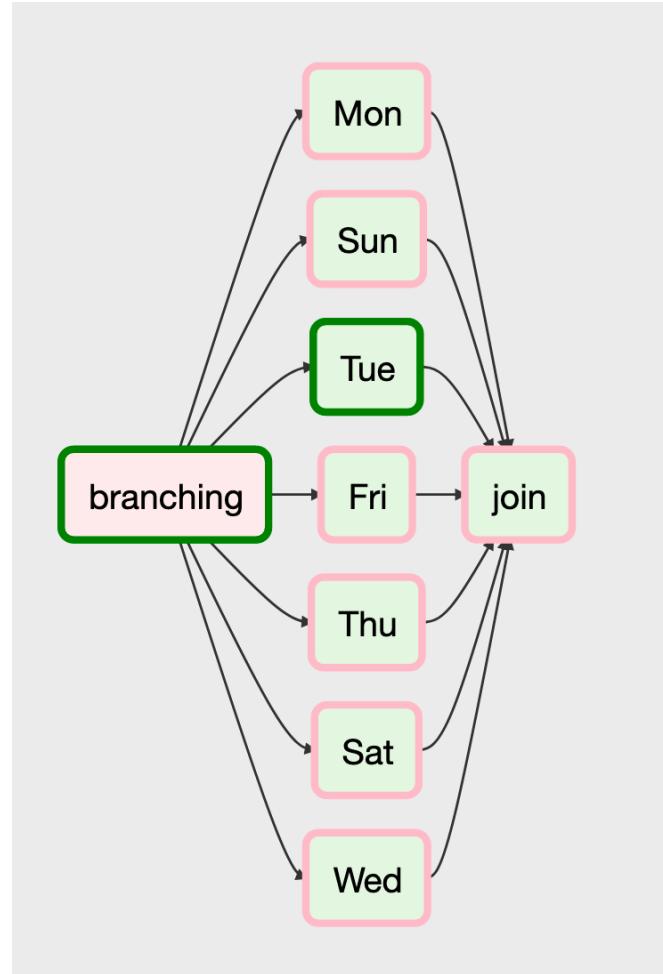
branching = BranchPythonOperator(
    task_id="branching",
    python_callable=_get_weekday,
    provide_context=True,
    dag=dag
)

days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
for day in days:
    branching >> DummyOperator(task_id=day, dag=dag)
```



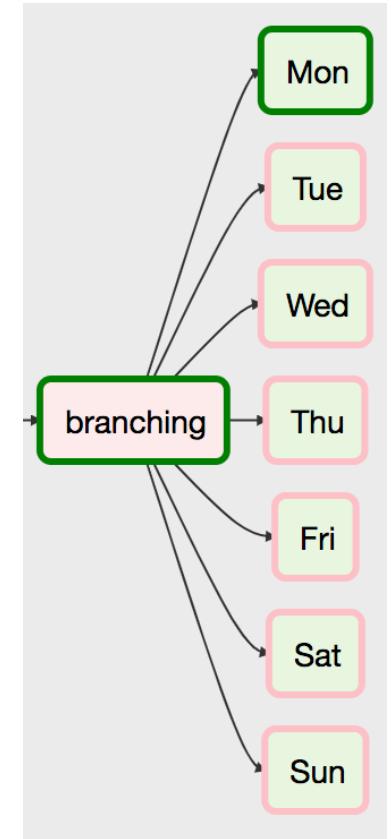
Continuing after a branch

- Typically, execution after a branch is continued after a (dummy) join task
- This allows downstream tasks to be ‘unaware’ of the branch
- However, naively adding a join does not work.
Any idea why?



Trigger rules

- Trigger rules determine when a task is executed
 - Defined using the `trigger_rule` Operator argument
 - Default rule is `all_success` (all upstream tasks must have completed successfully)



Trigger rules

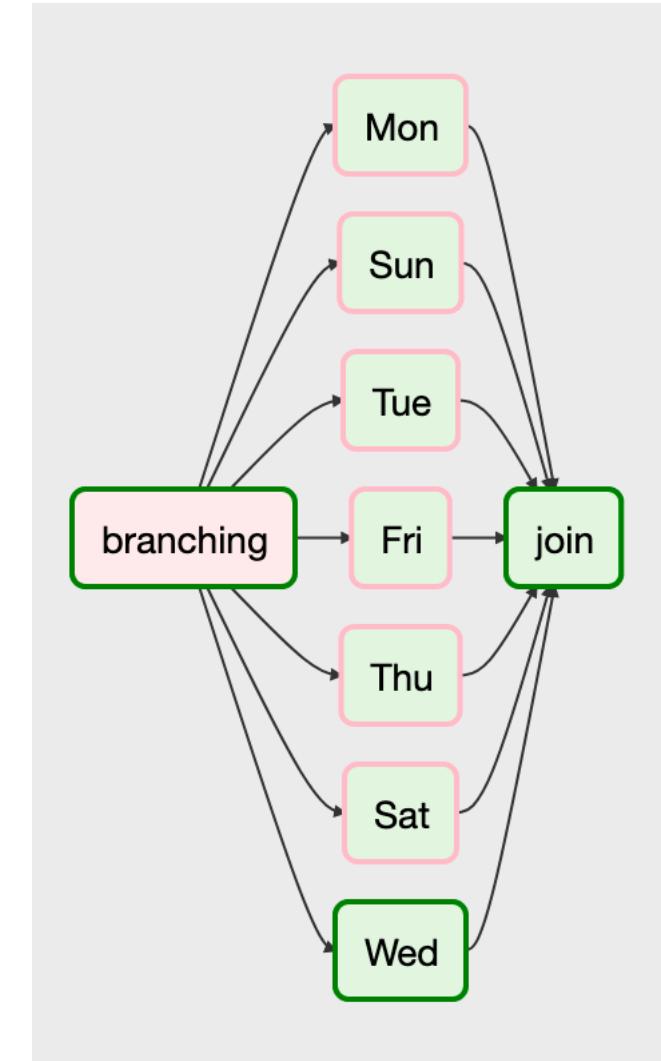
TriggerRule.ALL_SUCCESS	(Default) all parents have succeeded.
TriggerRule.ALL_FAILED	All parents are in a failed or upstream_failed state.
TriggerRule.ALL_DONE	All parents are done with their execution.
TriggerRule.ONE_FAILED	Fires as soon as at least one parent has failed, it does not wait for all parents to be done.
TriggerRule.ONE_SUCCESS	Fires as soon as at least one parent succeeds, it does not wait for all parents to be done.
TriggerRule.DUMMY	Dependencies are just for show, trigger at will.

Fixing the join

- Changing the trigger rule is enough:

```
join = DummyOperator(  
    task_id="join",  
    trigger_rule="none_failed"  
)
```

- Ensures join will run if none of the upstream tasks fail



Skipping some tasks

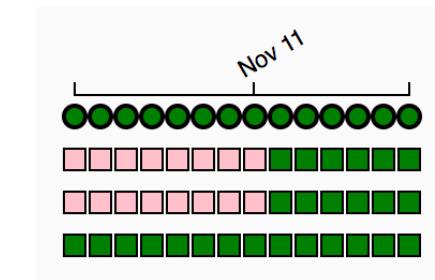
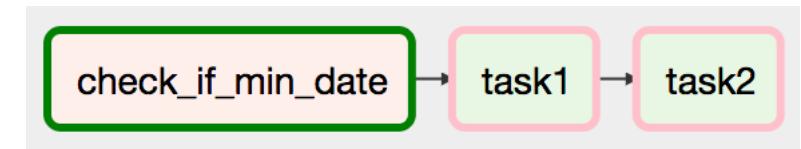
- Sometimes you might like to skip tasks based on a condition
- For example
 - Only running tasks for the most recent run
 - Skipping load if there is no data
- For this you can use the ShortCircuitOperator

The ShortCircuitOperator

```
def check_date(execution_date, **context):
    return execution_date > (datetime.datetime.now() - relativedelta(weeks=1))

check_date = ShortCircuitOperator(
    task_id="check_if_min_date",
    python_callable=check_date,
    provide_context=True,
    dag=dag,
)
task1 = DummyOperator(task_id="task1", dag=dag)
task2 = DummyOperator(task_id="task2", dag=dag)

check_date >> task1 >> task2
```



Another option – Skip exceptions

```
def _check_date(execution_date, **context):
    min_date = datetime.datetime.now() - relativedelta(weeks=1)
    if execution_date < min_date:
        raise AirflowSkipException(f"No data available on this execution_date ({execution_date}).")

check_date = PythonOperator(
    task_id="check_if_min_date",
    python_callable=_check_date,
    provide_context=True,
    dag=dag,
)

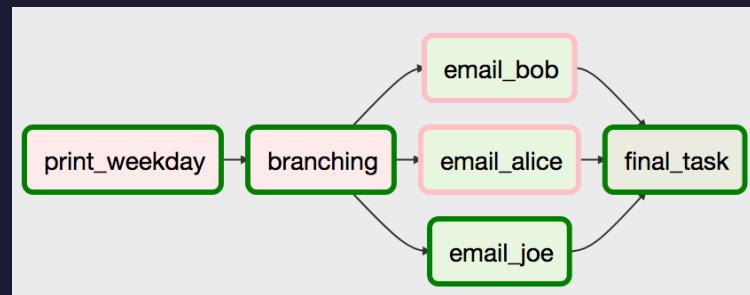
task1 = DummyOperator(task_id="task1", dag=dag)
task2 = DummyOperator(task_id="task2", dag=dag)

check_date >> task1 >> task2
```

Fourth Exercise

- Write a DAG that does the following:
 1. Start by printing the weekday using the PythonOperator
 2. Run a DummyOperator for given person, based on weekday
 3. Finish DAG with BashOperator

- Desired result:



```
weekday_person_to_email = {  
    0: "Bob", # Monday  
    1: "Joe", # Tuesday  
    2: "Alice", # Wednesday  
    3: "Joe", # Thursday  
    4: "Alice", # Friday  
    5: "Alice", # Saturday  
    6: "Alice", # Sunday  
}
```

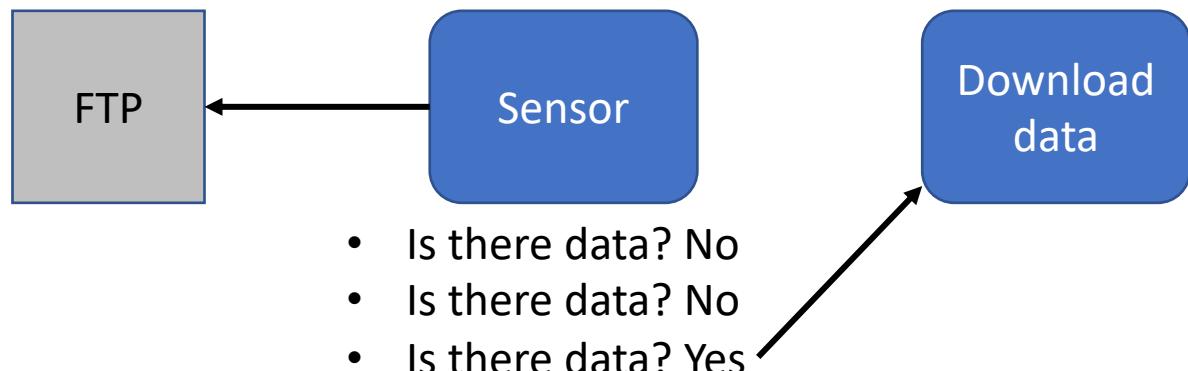


Airflow – ‘Advanced’ components

GO 
DATA
DRIVEN

Sensors

- Subclass of Operators
- Main method -> Poke
 - Checks if certain condition is met
 - Starts once that condition is True (or timed out).
- Useful if you want to process directly after an event, such as delivery of the data



Sensors

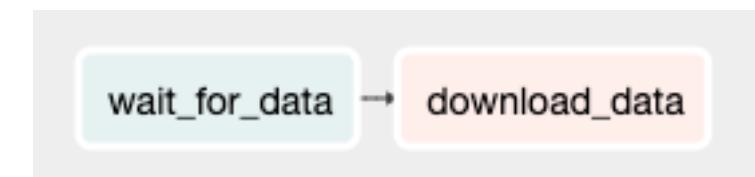
```
import airflow.utils.dates
from airflow.contrib.sensors.ftp_sensor import FTPSensor
from airflow.models import DAG

dag = DAG(
    dag_id="example_sensor",
    default_args={"owner": "Bob", "start_date": airflow.utils.dates.days_ago(5)},
    schedule_interval=None,
)

wait_for_data = FTPSensor(
    task_id="wait_for_data",
    path="/path/to/data",
    ftp_conn_id="ftp_connection",
    dag=dag,
)

download_data = PythonOperator(
    task_id="download_data",
    python_callable=lambda: print("do magic"),
    dag=dag,
)

wait_for_data >> download_data
```



Sensors – custom check with PythonSensor

```
from datetime import datetime

import airflow.utils.dates
from airflow.contrib.sensors.python_sensor import PythonSensor
from airflow.models import DAG
from airflow.operators.bash_operator import BashOperator

dag = DAG(
    dag_id="example_python_sensor",
    default_args={"owner": "GoDataDriven", "start_date": airflow.utils.dates.days_ago(3)},
    schedule_interval="0 0 * * *",
    description="Is it time for coffee?",
)

def _time_for_coffee():
    """I drink coffee between 6 and 12"""
    if 6 <= datetime.now().hour < 12:
        return True
    else:
        return False

time_for_coffee = PythonSensor(
    task_id="time_for_coffee", python_callable=_time_for_coffee, mode="reschedule", dag=dag
)

make_coffee = BashOperator(task_id="make_coffee", bash_command="echo 'Time for coffee!', dag=dag)

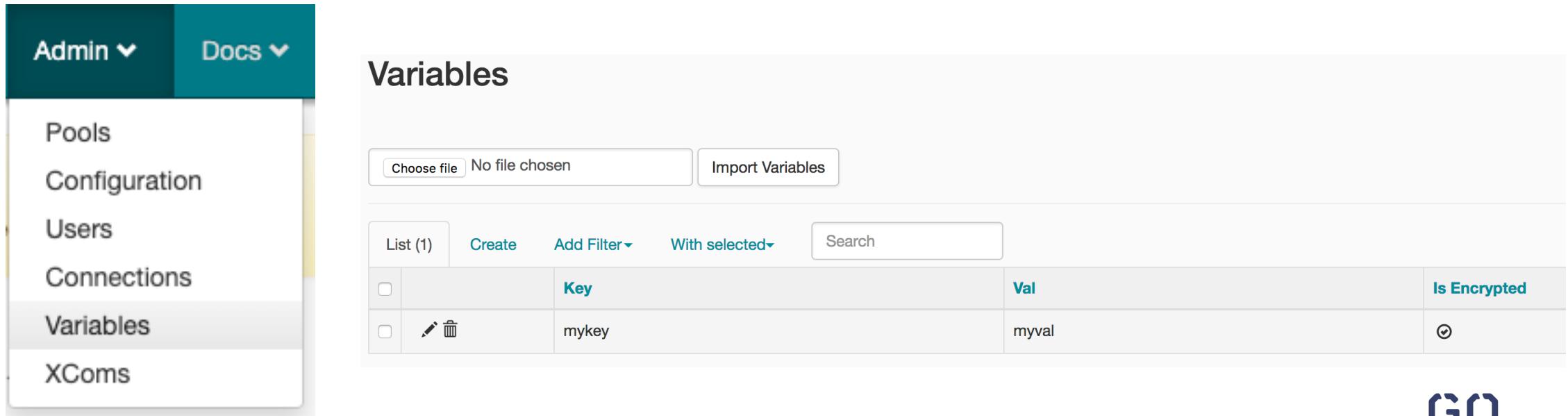
time_for_coffee >> make_coffee
```

True = continue
False = retry



Variables

- Just like connections, variables (non-secret) can also be stored in Airflow



The screenshot shows the Airflow web interface with the 'Admin' menu selected. The left sidebar includes options for Pools, Configuration, Users, Connections, Variables (which is currently selected), and XComs. The main content area is titled 'Variables' and displays a table with one row. The table has columns for Key, Val, and Is Encrypted. The single entry is 'mykey' with value 'myval' and the 'Is Encrypted' checkbox checked.

	Key	Val	Is Encrypted
<input type="checkbox"/>	mykey	myval	<input checked="" type="checkbox"/>

Variables

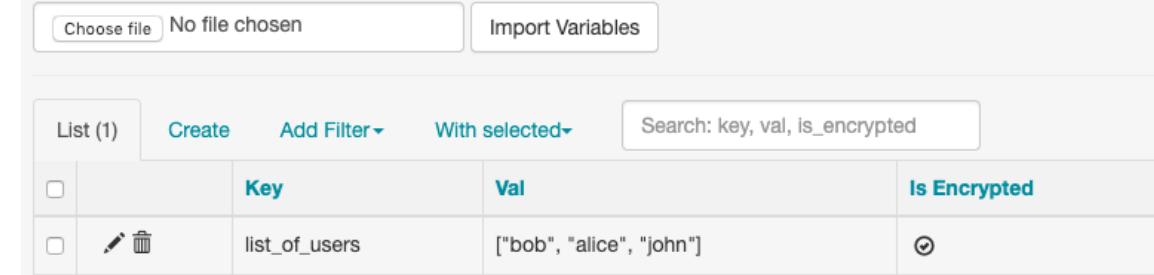
```
import airflow.utils.dates
from airflow.models import DAG, Variable
from airflow.operators.python_operator import PythonOperator

dag = DAG(
    dag_id="example_variable_usage",
    default_args={"owner": "GoDataDriven", "start_date": airflow.utils.dates.days_ago(5)},
    schedule_interval=None,
)

def _do_magic():
    users = Variable.get("list_of_users", deserialize_json=True)
    for user in users:
        ...

do_magic = PythonOperator(task_id="do_magic", python_callable=_do_magic, dag=dag)
```

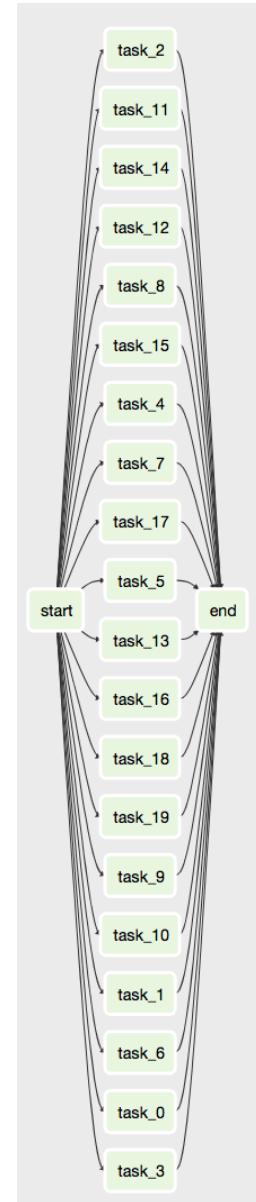
Variables



Choose file		No file chosen	Import Variables
List (1)	Create	Add Filter ▾	With selected ▾
	Key	Val	Is Encrypted
<input type="checkbox"/>	<input type="text"/> list_of_users	<input type="text"/> ["bob", "alice", "john"]	<input checked="" type="checkbox"/>

Pools

- Pools are used to limit parallelism in Airflow tasks
- There is a global config for maximum parallel tasks
- Pools can be used to limit parallelism within that global maximum

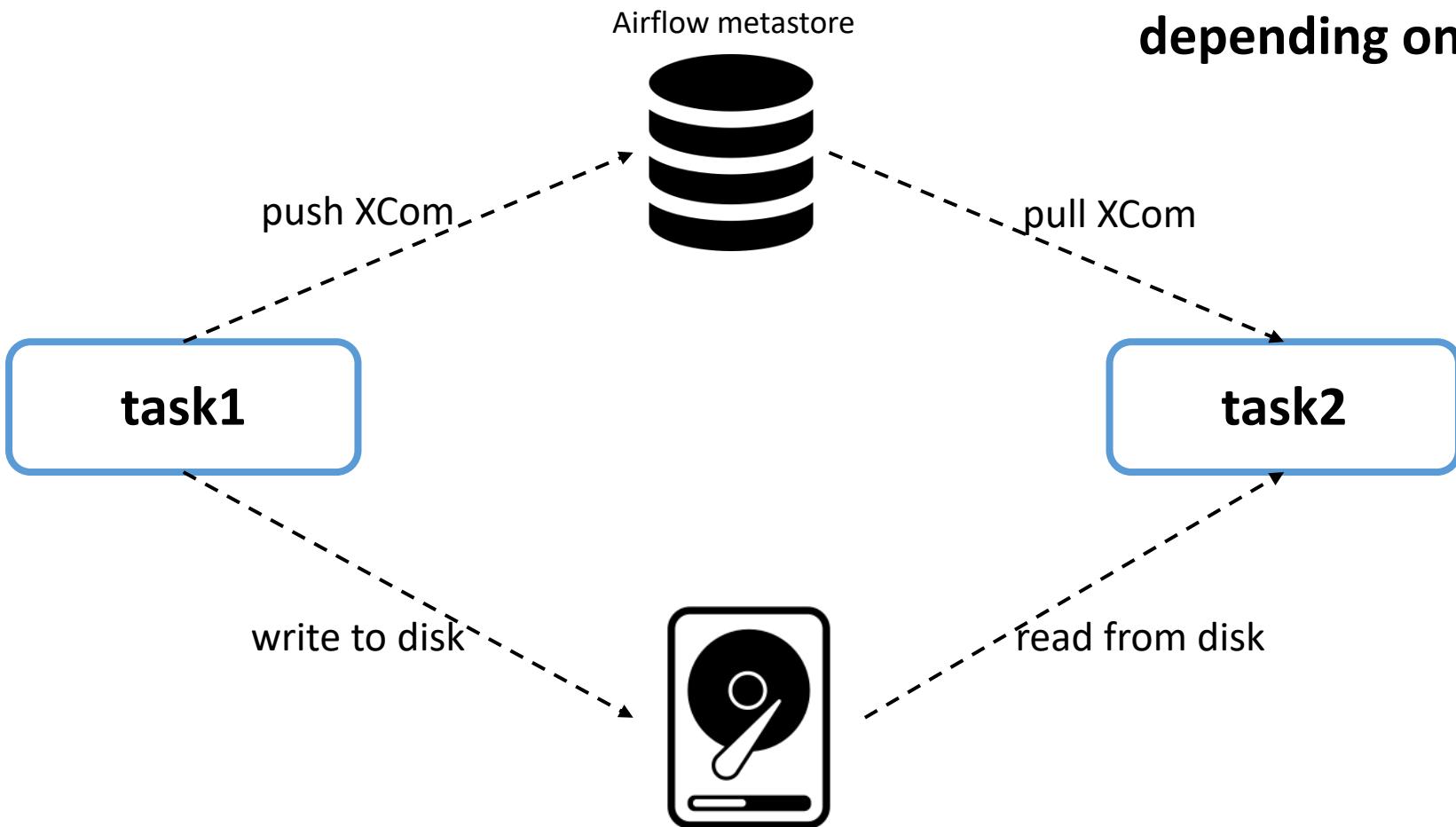


XComs

- Data can be communicated between tasks with the help of XComs (cross-communication)
- Works for anything that can be pickled
- Pickled objects are stored in the Airflow metastore

XComs

Decide XCom/other
depending on size of data



XComs

```
import airflow.utils.dates
from airflow.models import DAG
from airflow.operators.python_operator import PythonOperator

dag = DAG(
    dag_id="example_xcom",
    default_args={"owner": "GoDataDriven", "start_date": airflow.utils.dates.days_ago(5)},
    schedule_interval=None,
)
pushtask = BashOperator(task_id="pushtask", bash_command="echo 'Hoi Fokko'", xcom_push=True)
```

```
def _pushtask(task_instance, **_):
    return "Hoi Fokko"
```



```
pushtask = PythonOperator(task_id="pushtask", python_callable=_pushtask, xcom_push=True)
```

```
def _pulltask(task_instance, **_):
    print(task_instance.xcom_pull(task_ids="pushtask"))
    # prints "Hoi Fokko"
```



```
pulltask = PythonOperator(task_id="pulltask", python_callable=_pulltask, xcom_push=True)
```

```
pushtask >> pulltask
```

XComs

```
def _pushtask():
    print("do stuff")
    return "foobar" # automatic XCom push (be aware of this!)

pushtask = PythonOperator(
    task_id="pushtask",
    python_callable=_pushtask
)
```

Airflow - Custom components

GO 
DATA
DRIVEN

Re-cap – Operators and hooks

- Hooks
 - Used to interact with (external) systems
- Operators
 - Actually execute tasks on (remote) systems
 - Typically use one or more hooks

Building your own Operator

```
class MyOwnOperator(BaseOperator):  
    @apply_defaults  
    def __init__(self, *args, **kwargs):  
        super().__init__(*args, **kwargs)  
  
    def execute(self, context):  
        pass
```

All variables in default_args will
be applied to this operator

implement this

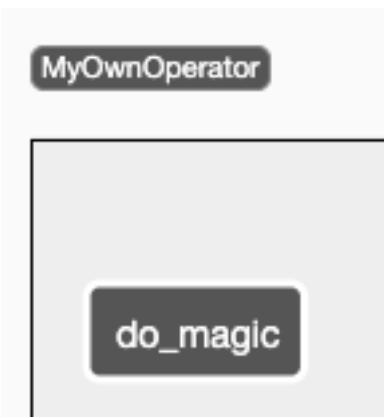
Building your own Operator

```
class MyOwnOperator(BaseOperator):  
  
    ui_color = '#555'  
    ui_fgcolor = '#fff'  
  
    @apply_defaults  
    def __init__(self, *args, **kwargs):  
        super().__init__(*args, **kwargs)  
  
    def execute(self, context):  
        pass
```

Change the look of the operator in the UI

Building your own Operator

```
class MyOwnOperator(BaseOperator):  
    template_fields = ('_myvar1',) Determines which fields are templated (note the underscore!)  
    ui_color = '#555'  
    ui_fgcolor = '#fff'  
  
    @apply_defaults  
    def __init__(self, myvar1, myvar2, *args, **kwargs):  
        super().__init__(*args, **kwargs)  
        self._myvar1 = myvar1 ←  
        self._myvar2 = myvar2 ←  
  
    def execute(self, context):  
        pass  
  
task1 = MyOwnOperator(  
    myvar1="it is now {{ execution_date }}",  
    myvar2="i won't be templated :("  
)
```



Building your own Hook

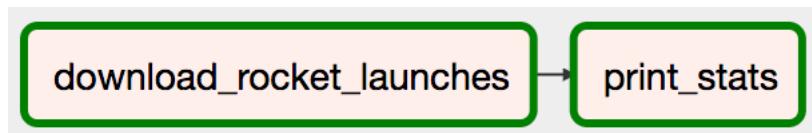
```
class MyOwnHook(BaseHook):
    def __init__(self, conn_id):
        super().__init__(source=None)
        self._conn_id = conn_id
        self._conn = None

    def get_conn(self):
        if self._conn is None:
            self._conn = object() # create connection instance here
        return self._conn

    def do_stuff(self, arg1, arg2, **kwargs):
        session = self.get_conn()
        session.do_stuff(# perform some action...)
```

Use case - Fetching rocket launches

- <https://launchlibrary.net>
- Fetch data about rocket launches from an API
 - Hook -> Interact with API
 - Operator -> Fetch + write results



```
import json
import pathlib
import posixpath

import airflow
import requests
from airflow.models import DAG
from airflow.operators.python_operator import PythonOperator

args = {"owner": "godatadriven", "start_date": airflow.utils.dates.days_ago(10)}

dag = DAG(
    dag_id="download_rocket_launches",
    default_args=args,
    description="DAG downloading rocket launches from Launch Library.",
    schedule_interval="@0 * * * *",
)

def _download_rocket_launches(ds, tomorrow_ds, **_):
    query = f"https://launchlibrary.net/1.4/launch?startdate={ds}&enddate={tomorrow_ds}"
    result_path = f"/tmp/rocket_launches/ds={ds}"
    pathlib.Path(result_path).mkdir(parents=True, exist_ok=True)
    response = requests.get(query)
    print(f"response was {response}")

    with open(posixpath.join(result_path, "launches.json"), "w") as f:
        print(f"Writing to file {f.name}")
        f.write(response.text)

def _print_stats(ds, **_):
    with open(f"/tmp/rocket_launches/ds={ds}/launches.json") as f:
        data = json.load(f)
        rockets_launched = [launch["name"] for launch in data["launches"]]

        if rockets_launched:
            rockets_str = f" ({' & '.join(rockets_launched)})"
            print(f"{len(rockets_launched)} rocket launch(es) on {ds}{rockets_str}.")
        else:
            print(f"No rockets found in {f.name}")

download_rocket_launches = PythonOperator(
    task_id="download_rocket_launches",
    python_callable=_download_rocket_launches,
    provide_context=True,
    dag=dag,
)

print_stats = PythonOperator(
    task_id="print_stats", python_callable=_print_stats, provide_context=True, dag=dag
)

download_rocket_launches >> print_stats
```

Exercise: Refactor the previous DAG

1. Implement a LaunchOperator which fetches launches and stores them in xcom
2. Rebuild the DAG from the previous slide using your own operator (`exercise_launch/launch-dag.py`)
3. Bonus: Store the data in GCS instead

End goal

```
if response.status_code not in (200, 404):  
    # Launch Library returns 404 if no rocket launched in given interval.  
    response.raise_for_status()
```

On DAG: fifth_exercise

Graph View Tree View Task Duration Task Tries Landing Times

success Base date: 2020-07-06 00:00:01 Number of runs: 25 Run: scheduled_2

PythonOperator RocketLaunchOperator

```
graph LR; A[download_rocket_launches] --> B[print_stats]
```

End goal

```
args = {"owner": "Airflow", "start_date": airflow.utils.dates.days_ago(10)}
```

```
dag = DAG(  
    dag_id="fifth_exercise",  
    default_args=args,  
    schedule_interval="@daily",  
    dagrun_timeout=timedelta(minutes=60),  
)
```

```
download_rocket_launches = RocketLaunchOperator(  
    task_id="download_rocket_launches", dag=dag  
)
```

```
def _print_stats(ds, task_instance, **context):  
    data = task_instance.xcom_pull(task_ids="download_rocket_launches")  
    print("Date from xcom: {}".format(data))  
    rockets_launched = [launch["name"] for launch in data["launches"]]  
    rockets_str = ""
```

```
    if rockets_launched:  
        rockets_str = f" ({' & '.join(rockets_launched)})"  
        print(f"{len(rockets_launched)} rocket launch(es) on {ds}{rockets_str}.")  
    else:  
        print(f"No rockets found in {f.name}")
```

```
print_stats = PythonOperator(  
    task_id="print_stats", python_callable=_print_stats, provide_context=True, dag=dag  
)
```

```
download_rocket_launches >> print_stats
```

Airflow - Testing

GO 
DATA
DRIVEN

Testing in Airflow

- Unit test your custom Airflow components
- Testing full DAGs is inherently difficult; often external systems are not accessible from your development machine
 - Mock external system calls/responses
 - Create a local Docker development environment
 - Check if development endpoints are available
- <https://blog.godatadriven.com/testing-and-debugging-apache-airflow>

7 Circles of Data Testing Hell with Airflow



<https://medium.com/wbaa/datas-inferno-7-circles-of-data-testing-hell-with-airflow-cef4adff58d8>

Testing in Airflow

- Verifies correctness of your DAGs
 - Don't contain cycles
 - No duplicate tasks
 - Etc.
- Run on your local machine (requires pytest installed):
 - `pytest tests/`

Testing operators with Pytest

```
import pytest
from airflow import DAG

pytest_plugins = ["helpers_namespace"]

@pytest.fixture
def test_dag():
    """Airflow DAG for testing."""
    return DAG(
        "test_dag",
        default_args={"owner": "airflow", "start_date": datetime.datetime(2018, 1, 1)},
        schedule_interval=datetime.timedelta(days=1),
    )

@pytest.helpers.register
def run_task(task, dag):
    """Run an Airflow task."""
    dag.clear()
    task.run(
        start_date=dag.default_args["start_date"],
        end_date=dag.default_args["start_date"],
    )
```

Testing operators with Pytest

```
class TestOperator:

    def test_operator(self, test_dag, mocker, tmpdir):
        # Mock get_connection because we have no metastore and no connection
        mocker.patch.object(BaseHook, "get_connection", return_value=Connection())

        # TODO: Mock any external calls in hooks.

        # Force run task with Operator
        task = MyOperator(...)
        pytest.helpers.run_task(task=task, dag=test_dag)

        # TODO: Check result.
```

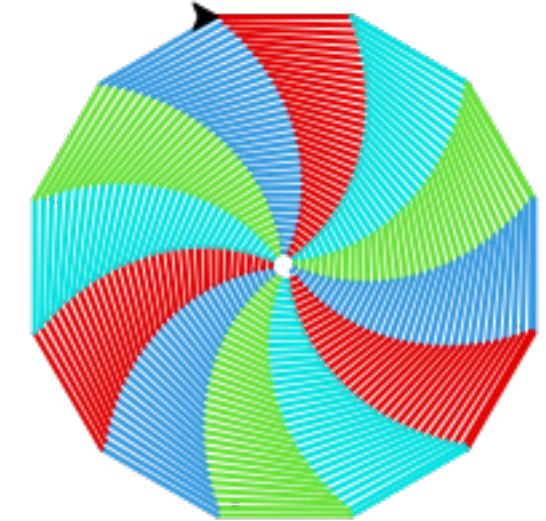
Testing tasks from the CLI

```
airflow test dag-name task-name 2018-01-01
```

```
$ gcloud composer environments run --project airflowbolcom-102b0f55fa0b9585 training-airflow --location europe-west1 test --exercise1 print_exec_date 2018-11-20

kubeconfig entry generated for europe-west1-training-airfl-24c2c786-gke.
[2018-11-21 23:09:54,285] {models.py:1569} INFO - Executing <Task(BashOperator): print_exec_date> on 2018-11-20T00:00:00+00:00@-@{}
[2018-11-21 23:09:54,389] {bash_operator.py:74} INFO - Tmp dir root location:
/tmp
[2018-11-21 23:09:54,393] {bash_operator.py:87} INFO - Temporary script location: /tmp/airflowtmp9eek03qx/print_exec_date7ck3860q
[2018-11-21 23:09:54,394] {bash_operator.py:97} INFO - Running command: echo 2018-11-20T00:00:00+00:00
[2018-11-21 23:09:54,400] {bash_operator.py:106} INFO - Output:
[2018-11-21 23:09:54,404] {bash_operator.py:110} INFO - 2018-11-20T00:00:00+00:00
[2018-11-21 23:09:54,405] {bash_operator.py:114} INFO - Command exited with return code 0
```

Testing DAGs with Whirl



- *“Fast iterative local development and testing of Apache Airflow workflows”*
- Uses Docker to run Airflow + simulate other components of your workflow (MySQL, SFTP, S3, APIs, etc.)
- <https://github.com/godatadriven/whirl>

Other useful tools

- Black (others: YAPF & autopep8)
 - Code formatter according to PEP8 standards
- PyLint (others: Flake8, pep8 & pyflakes)
 - Checks syntactical issues and potential problems but does not improve code
 - E.g. unused function arguments/Python 2 usage

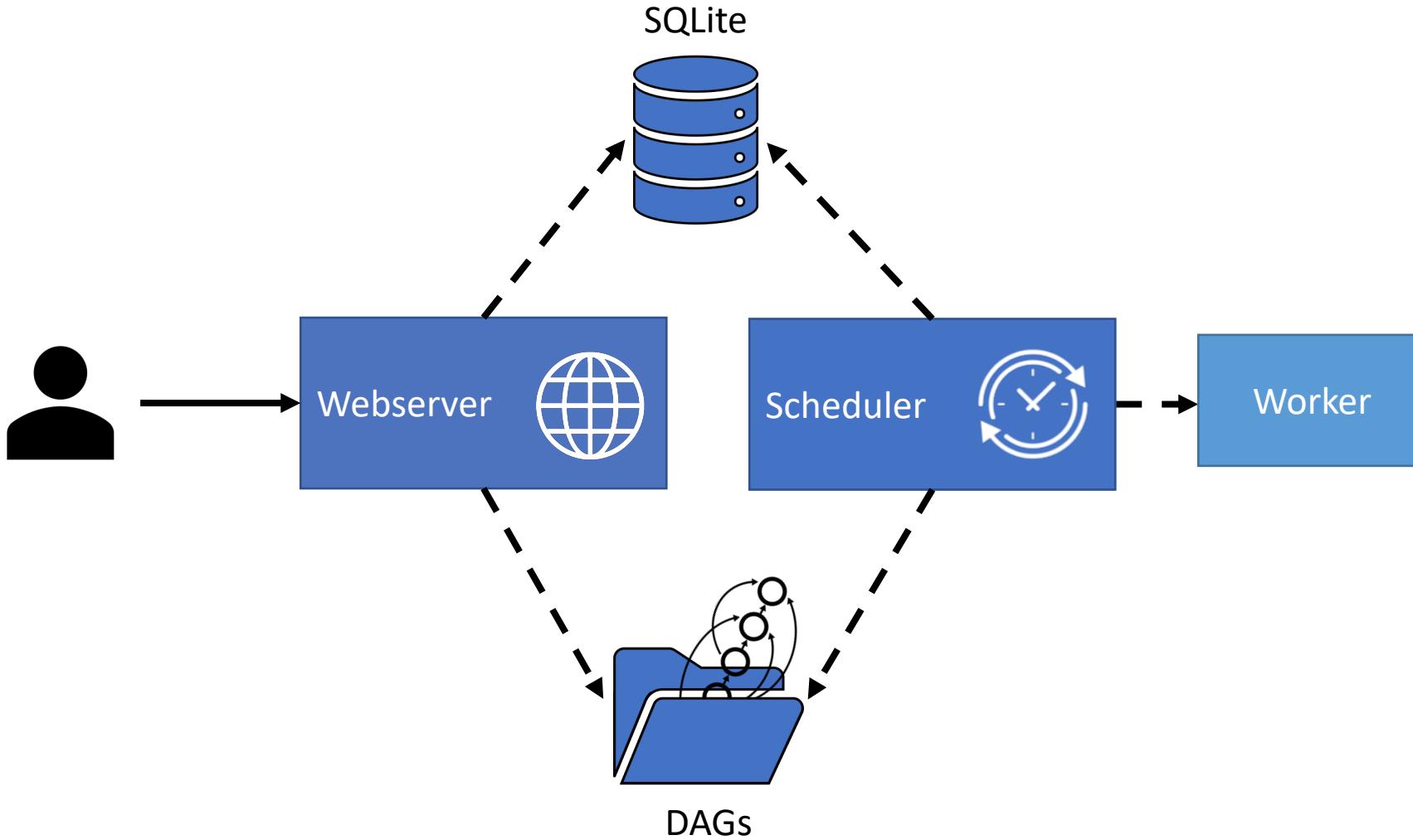
Exercise

- Write the following tests for our previous DAG (using pytest)
 - The basic DAG integrity test
 - Unit test(s) for the LaunchHook
 - Unit test(s) for the LaunchOperator
 - Bonus: try running Black/Pylint
- Tip: for the LaunchHook tests, you might consider mocking responses from the API

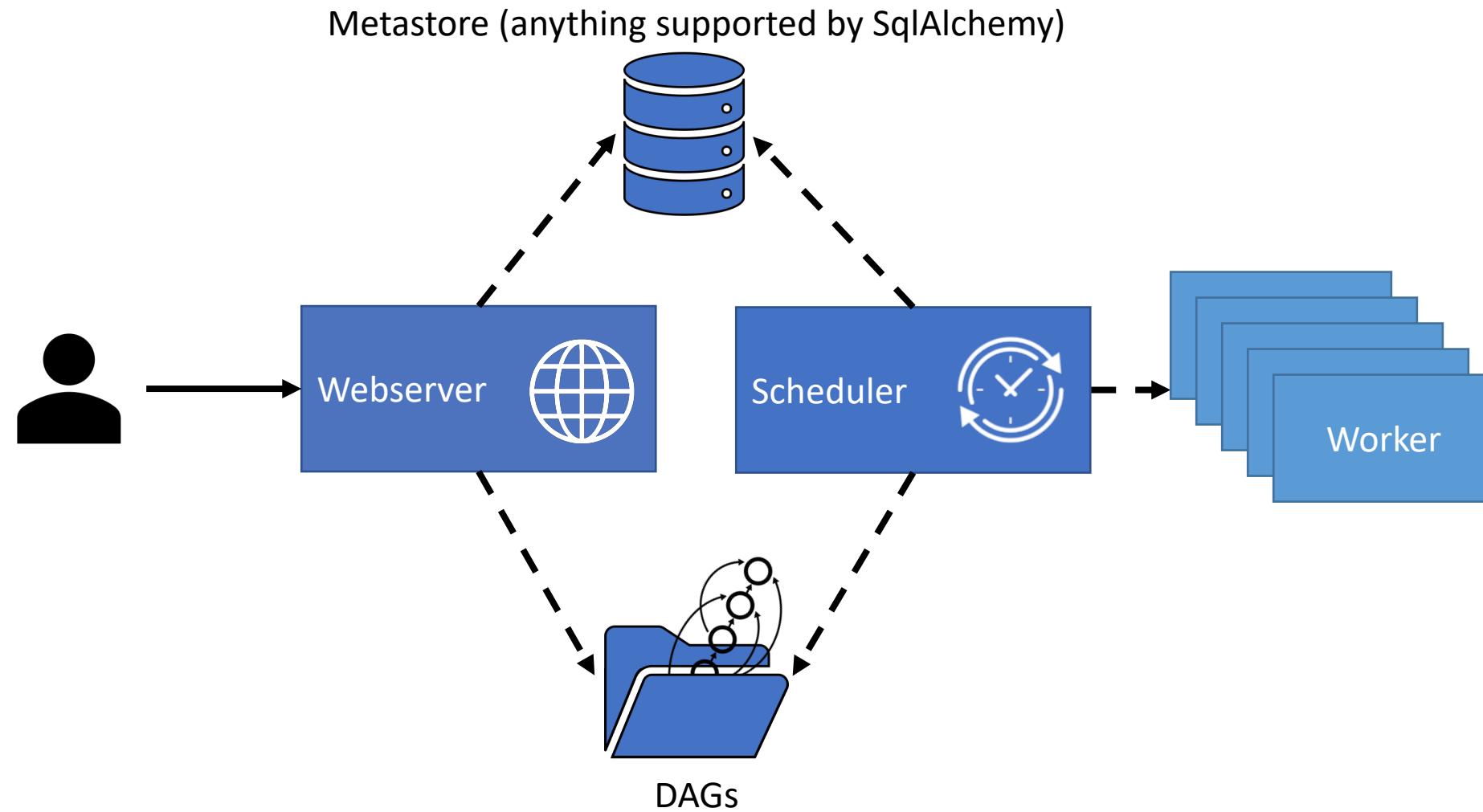
Airflow - Architectures

GO 
DATA
DRIVEN

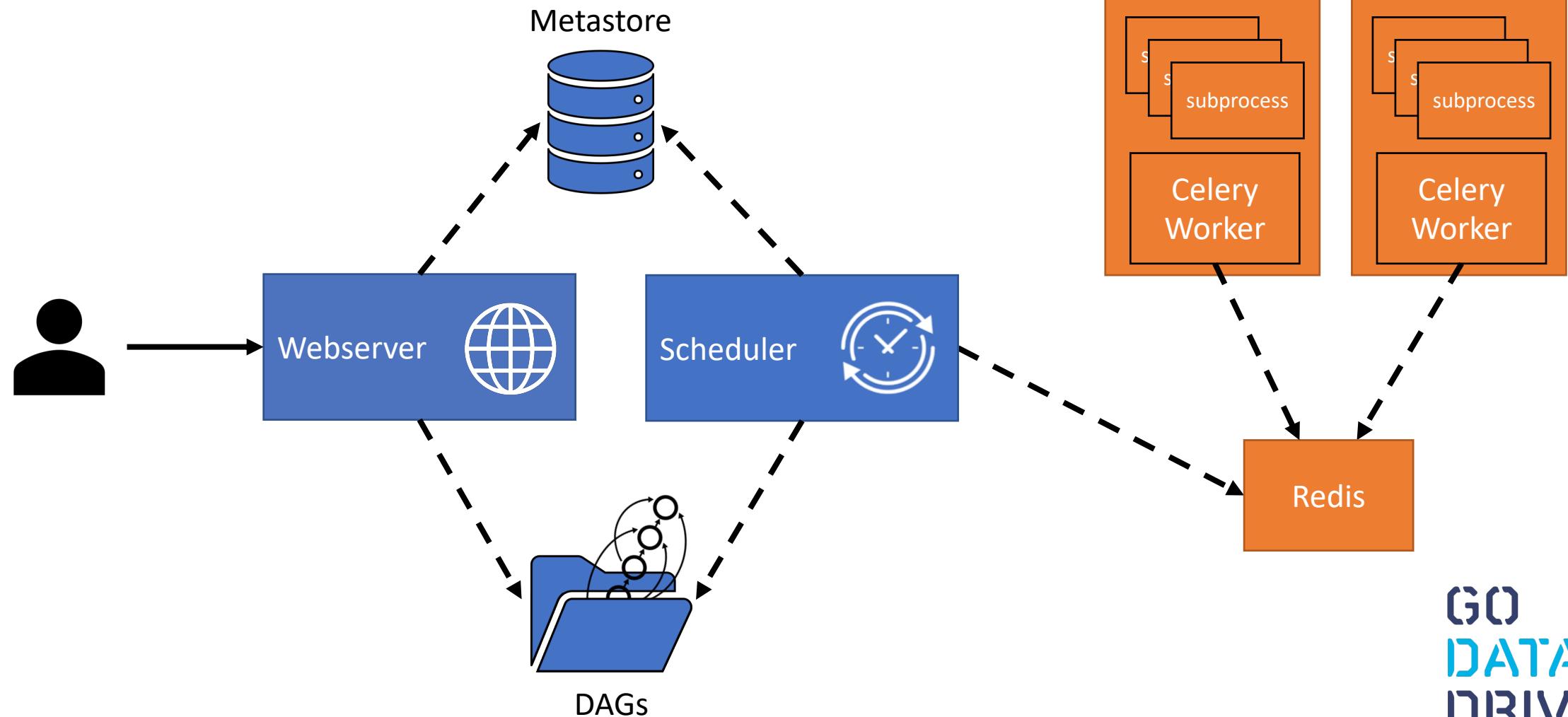
Sequential Executor



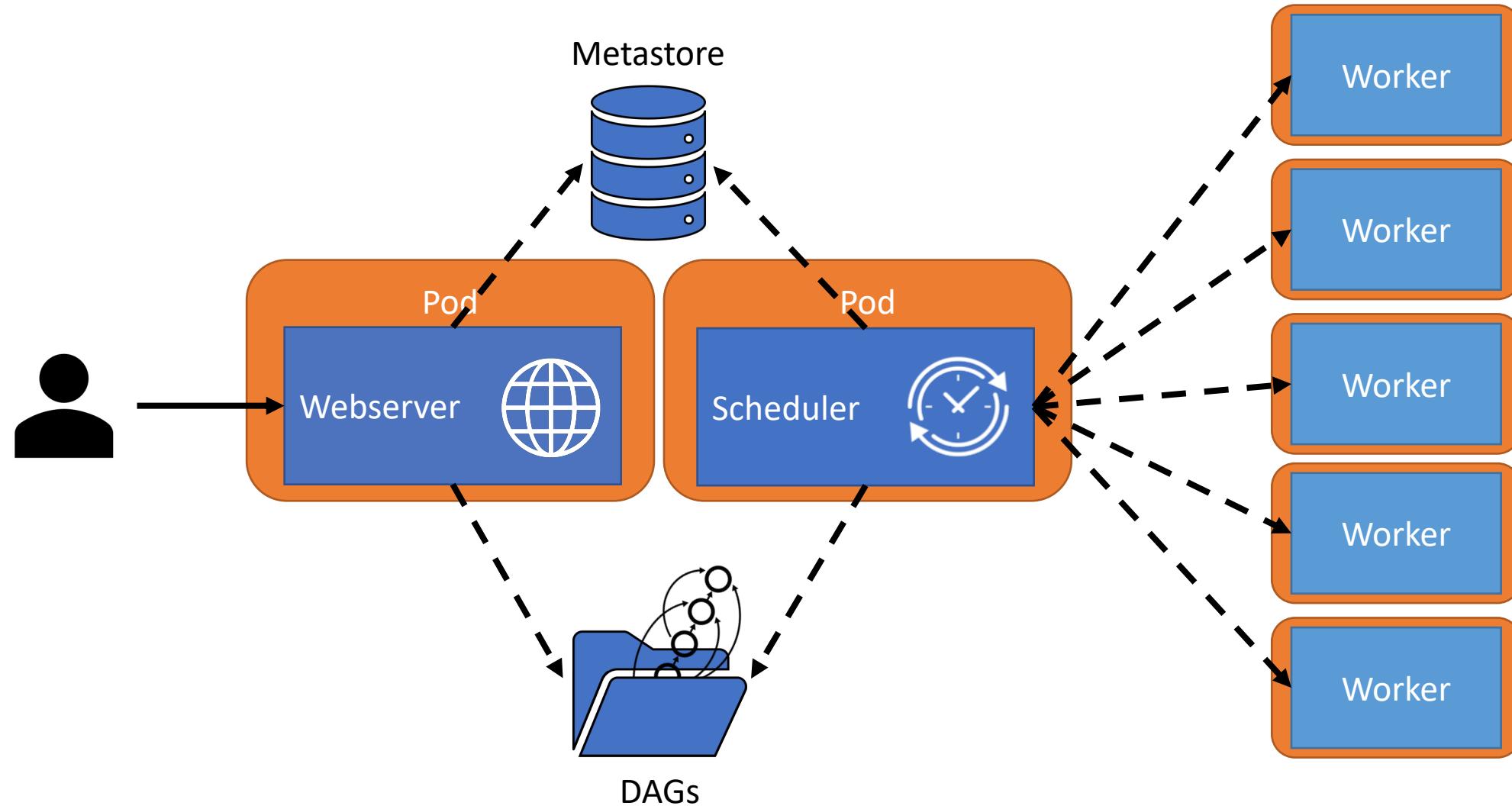
Local Executor



Celery Executor



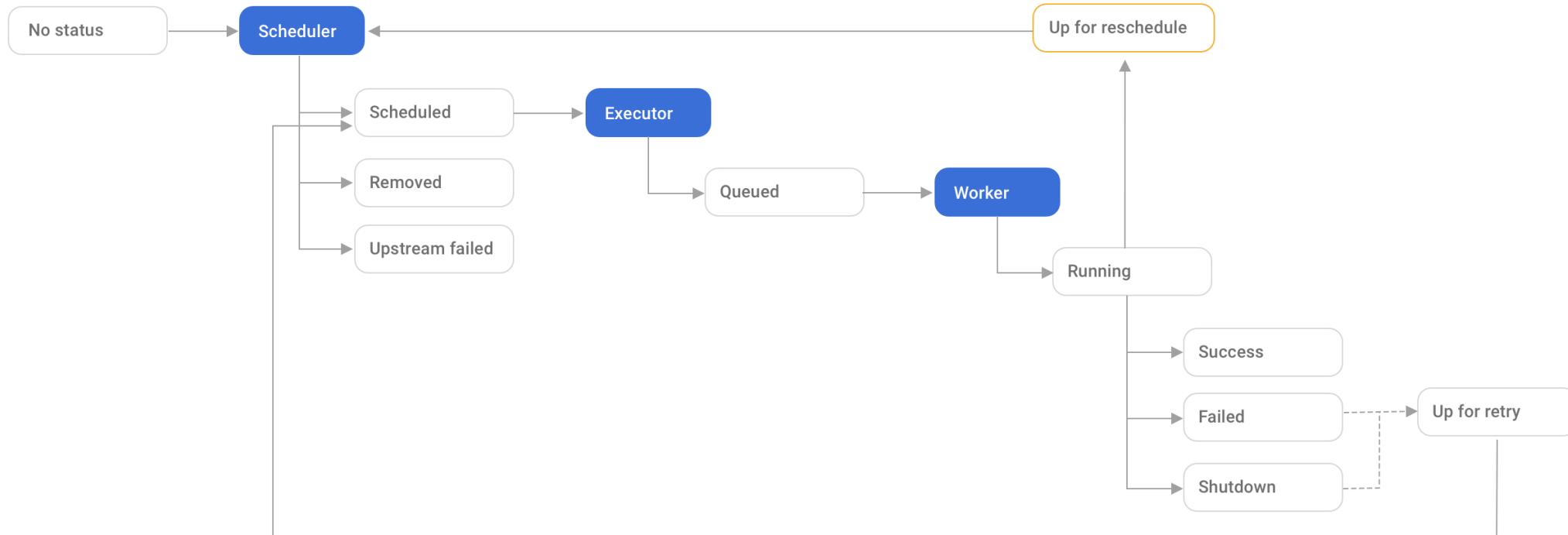
Kubernetes Executor



Extras

GO 
DATA
DRIVEN

Task lifecycle



Questions?

- Email me at fokko@apache.org

- Solutions at:

<https://github.com/Fokko/airflow-training-skeleton>

