What's coming in Apache Airflow 2.0

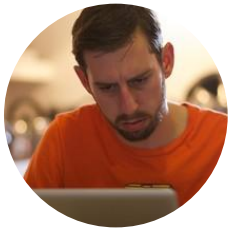
NYC Meetup
13th of May 2020

Who are we?

Tomek Urbaszek
Committer
Software Engineer @ Polidea

Jarek Potiuk
Committer, PMC member
Principal Software Engineer @ Polidea

Kamil Breguła
Committer
Software Engineer @ Polidea



Ash Berlin-Taylor
Committer, PMC member
Airflow Engineering Lead @ Astronomer

Daniel Imberman
Committer
Senior Data Engineer @ Astronomer

Kaxil Naik
Committer, PMC member
Senior Data Engineer @ Astronomer

High Availability

Scheduler High Availability

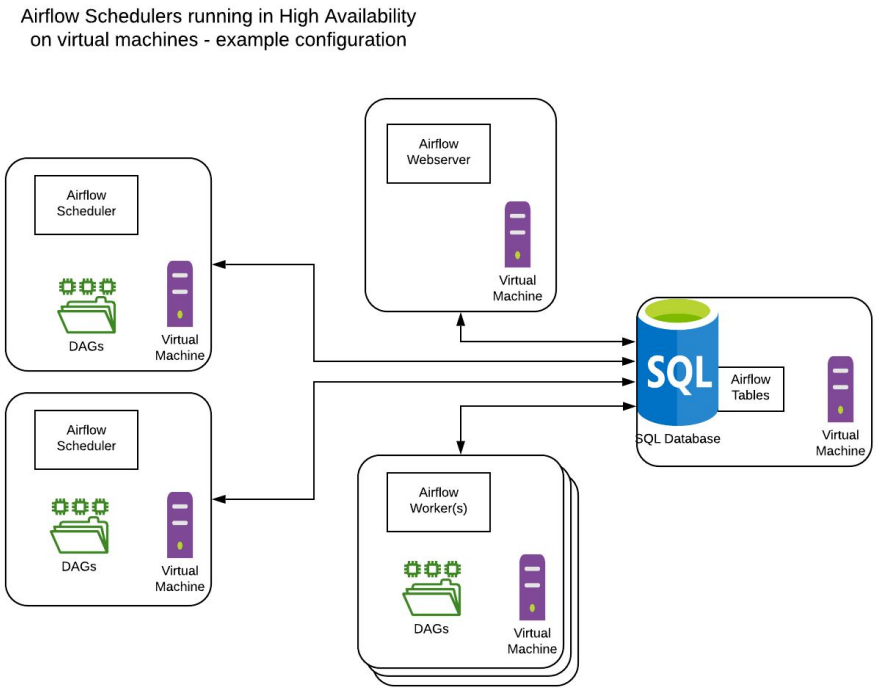
Goals:

- Performance - reduce task-to-task schedule "lag"
- Scalability - increase task throughput by horizontal scaling
- Resiliency - kill a scheduler and have tasks continue to be scheduled

Scheduler High Availability: Design

- Active-active model. Each scheduler does everything
- Uses existing database - no new components needed, no extra operational burden
- Plan to use row-level-locks in the DB
- Will re-evaluate if performance/stress testing show the need

Example HA configuration



Scheduler High Availability: Tasks

- Separate DAG parsing from DAG scheduling

  This removes the tie between parsing and scheduling that is still present

- Run a mini scheduler *in the worker* after each task is completed

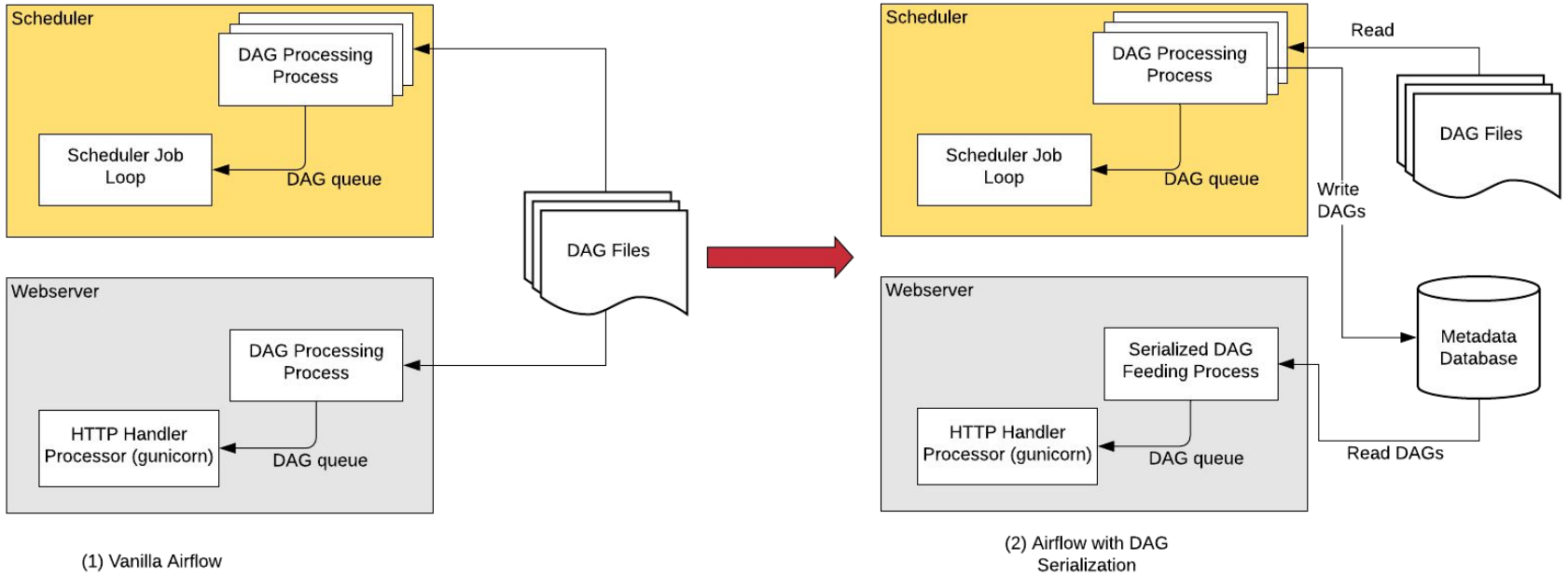
  A.K.A. "fast follow". Look at immediate down stream tasks of what just finished and see what we can schedule

- Test it to destruction

  This is a big architectural change, we need to be sure it works well.

DAG Serialization

Dag Serialization



Dag Serialization (Tasks Completed)

- **Stateless Webserver:** Scheduler parses the DAG files, serializes them in JSON format & saves them in the Metadata DB.
- **Lazy Loading of DAGs:** Instead of loading an entire DagBag when the Webserver starts we only load each DAG on demand. This helps **reduce Webserver startup time and memory.** This reduction is notable with large number of DAGs.
- Deploying new DAGs to Airflow - no longer requires long restarts of webserver (if DAGs are baked in Docker image)
- Feature to use the “JSON” library of choice for Serialization (default is inbuilt ‘json’ library)
- Paves way for **DAG Versioning** & **Scheduler HA**

Dag Serialization (Tasks In-Progress for Airflow 2.0)

- Decouple DAG Parsing and Serializing from the scheduling loop.
- Scheduler will fetch DAGs from DB
- DAG will be parsed, serialized and saved to DB by a separate component “Serializer”/ “Dag Parser”
- This should reduce the delay in Scheduling tasks when the number of DAGs are large

DAG Versioning

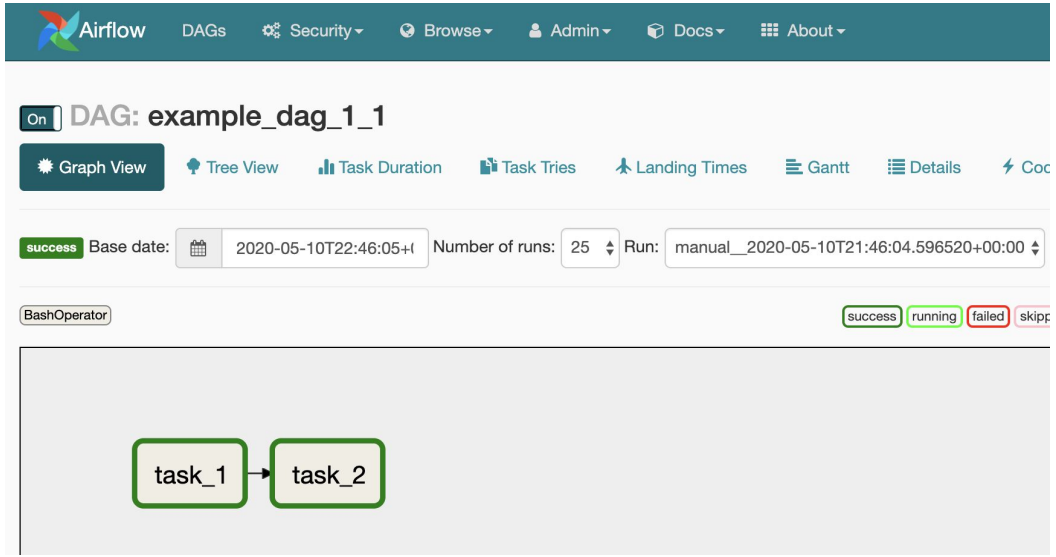
Dag Versioning

Current Problem:

- Change in DAG structure affects viewing previous DagRuns too
- Not possible to view the code associated with a specific DagRun

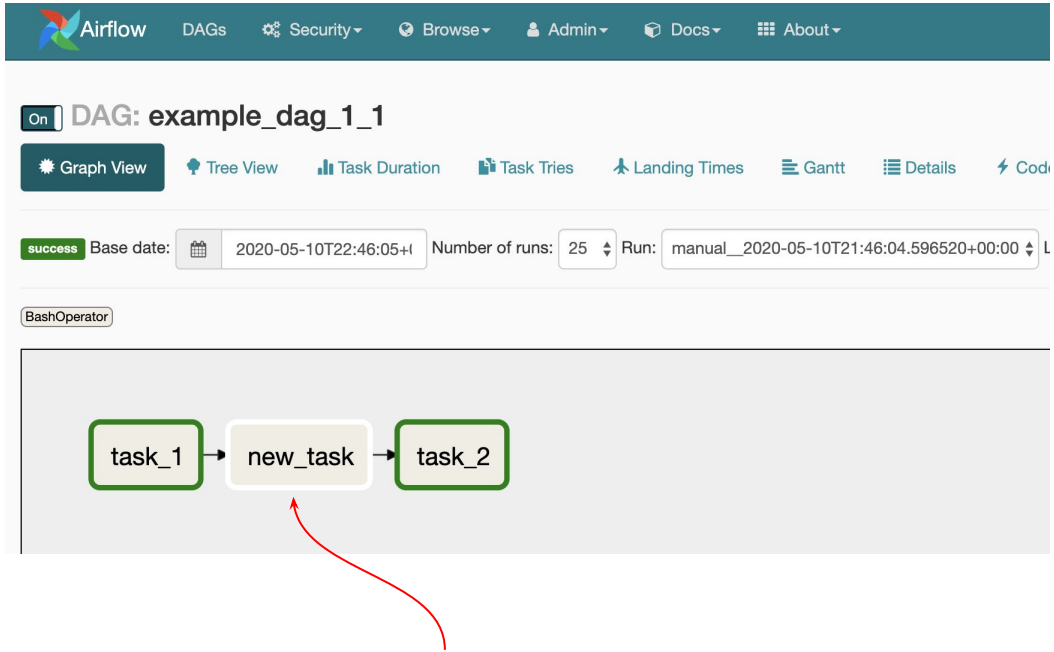
Dag Versioning (Current Problem)

```python
from airflow.models.dag import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime

with DAG('example_dag_1_1', schedule_interval=None,
         start_date=datetime(2020, 4, 25)) as example_dag_1_1:

    task_1 = BashOperator(
        task_id='task_1',
        bash_command='echo hello',
    )

    task_2 = BashOperator(
        task_id='task_2',
        bash_command='echo hello',
    )

    task_1 >> task_2
```



Dag Versioning (Current Problem)

```python
from airflow.models.dag import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime

with DAG('example_dag_1_1', schedule_interval=None,
         start_date=datetime(2020, 4, 25)) as example_dag_1_1:

    task_1 = BashOperator(
        task_id='task_1',
        bash_command='echo hello',
    )

    new_task = BashOperator(
        task_id='new_task',
        bash_command='echo hello',
    )

    task_2 = BashOperator(
        task_id='task_2',
        bash_command='echo hello',
    )

    task_1 >> new_task >> task_2
```



New task is shown in Graph View for older DAG Runs too with “no status”.

Dag Versioning

Current Problem:

- Change in DAG structure affects viewing previous DagRuns too
- Not possible to view the code associated with a specific DagRun

Goal:

- Support for storing multiple versions of Serialized DAGs
- Baked-In Maintenance DAGs to cleanup old DagRuns & associated Serialized DAGs
- Graph View shows the DAG associated with that DagRun

Performance Improvements

Performance improvements

- Review each component of scheduler in turn and its optimization.
- Perf kit
  - A set of tools that allows you to quickly check the performance of a component

Do you see a performance problem?

```
paused_dag_ids = {dag.dag_id for dag in dagbag.dags.values() if dag.is_paused}
```

```
class DAG:

    @provide_session
    def _get_is_paused(self, session=None):
        qry = session.query(DagModel).filter(
            DagModel.dag_id == self.dag_id)
        return qry.value(DagModel.is_paused)

    @property
    def is_paused(self):
        return self._get_is_paused()
```

Results for DagFileProcessor

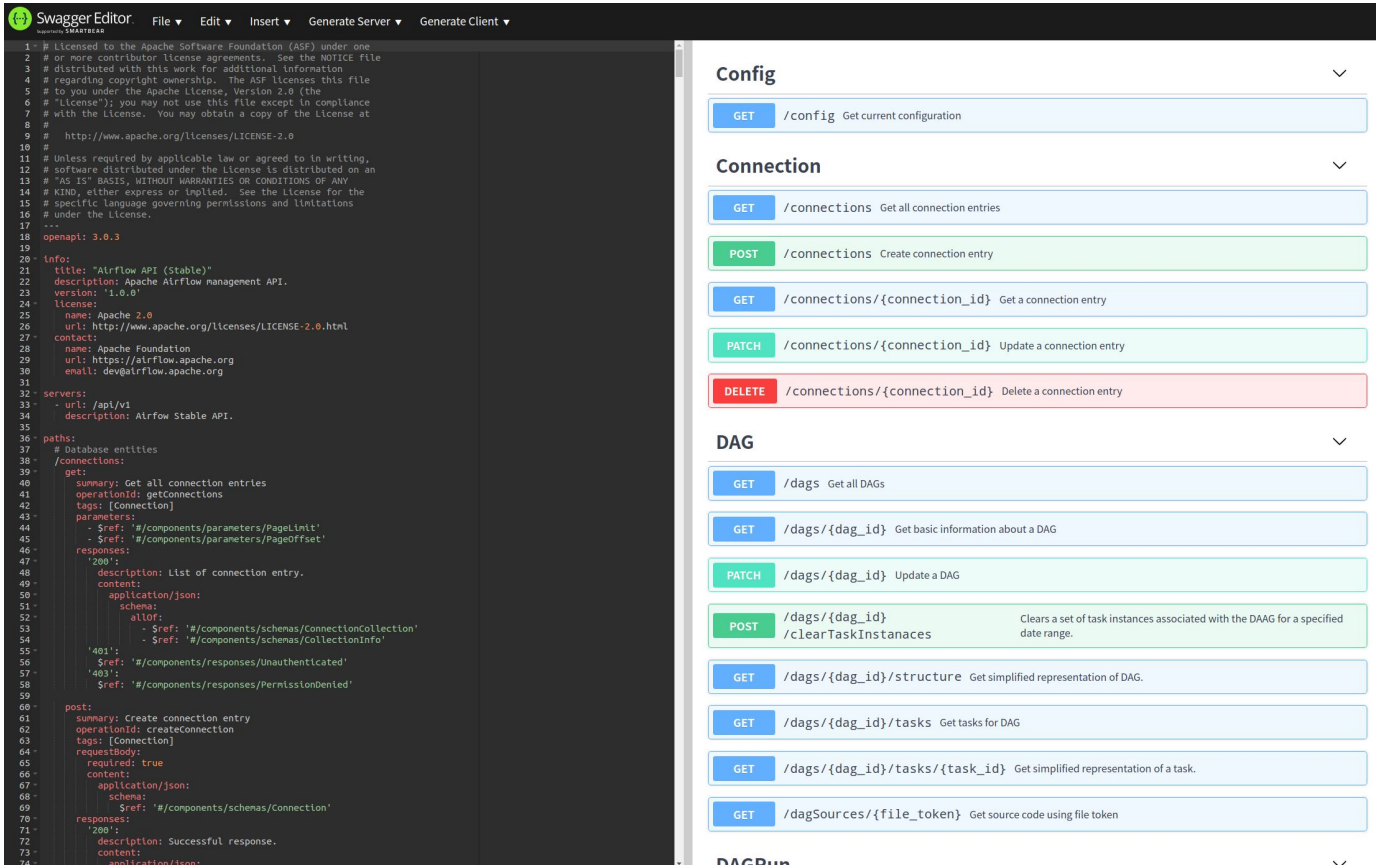
When we have one DAG file with 200 DAGs, each DAG with 5 tasks:

	Before	After	Diff
Average time:	8080.246 ms	628.801 ms	-7452 ms (92%)
Queries count:	2692	5	-2687 (99%)

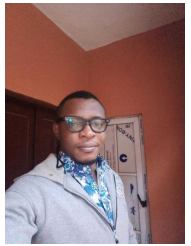
How to avoid regression?

REST API

API: follows Open API 3.0 specification



Outreachy interns



Ephraim Anierobi

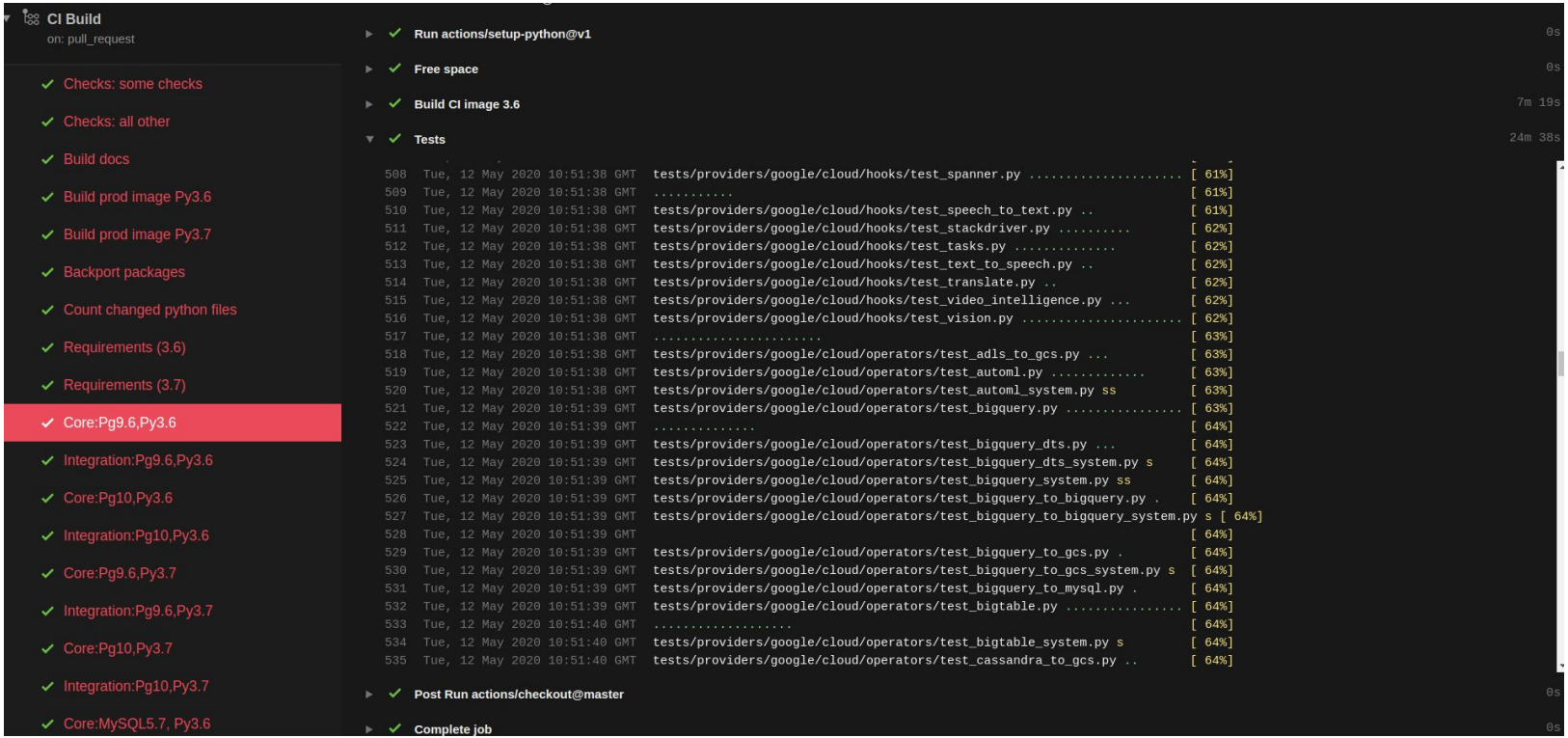
Omair Khan

Dev/CI environment

CI environment

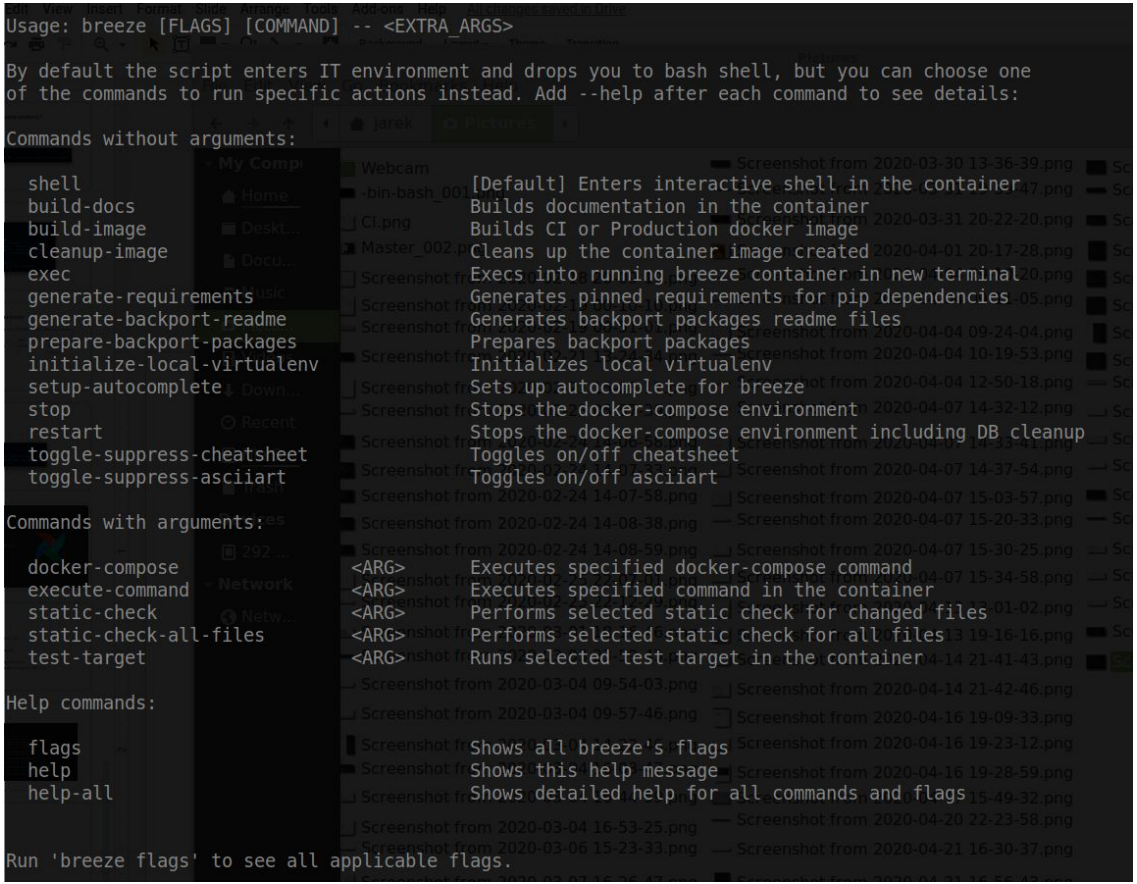
- Moving to GitHub Actions
  - Kubernetes Tests
  - Easier way to test Kubernetes Tests locally
- Quarantined tests
  - Process of fixing the Quarantined tests
- Thinning CI image
  - Move integrations out of the image (hadoop etc)
- Automated System Tests (AIP-21)

GitHub Actions



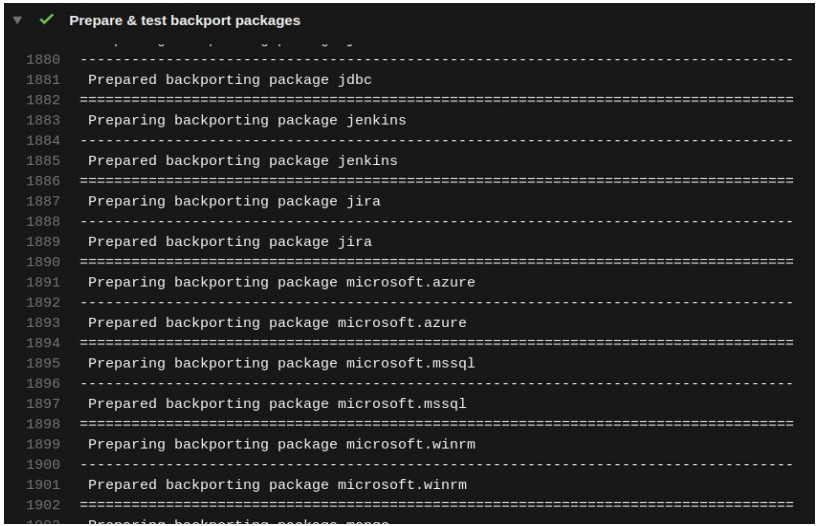
Dev environment

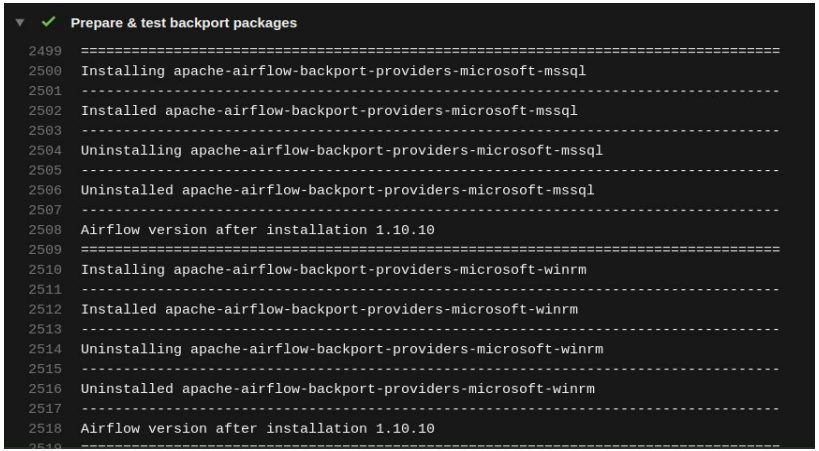
- Breeze
  - unit testing
  - package building
  - release preparation
  - refreshing videos
- CodeSpaces integration



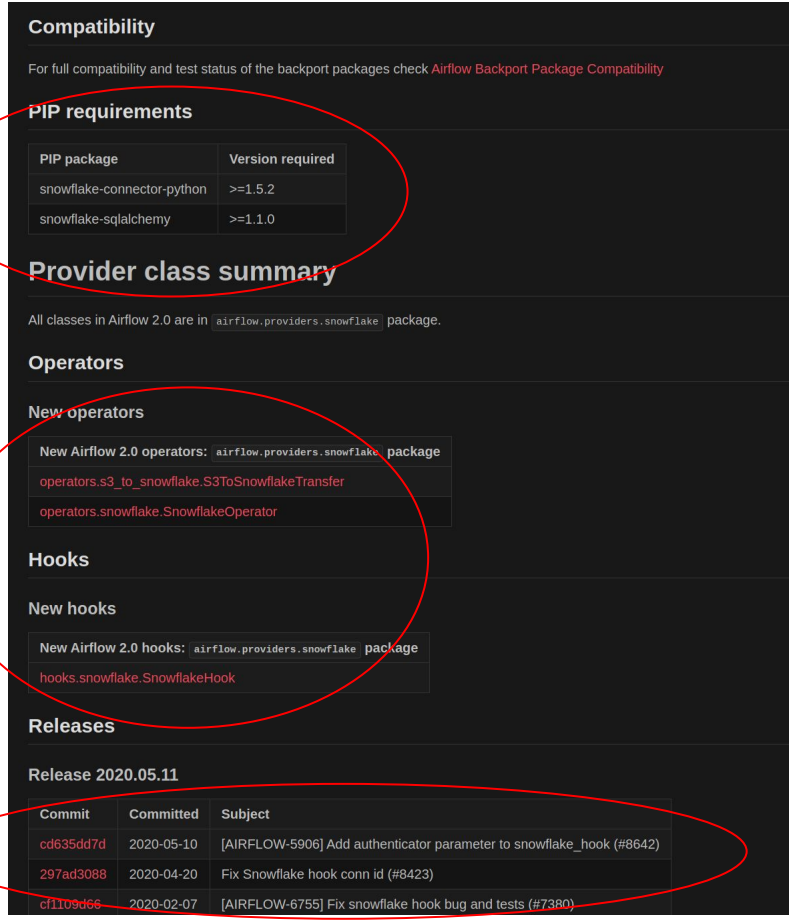
Backport Packages

- Bring Airflow 2.0 providers to 1.10.*
- Packages per-provider
- 58 packages (!)
- Python 3.6+ only(!)
- Automatically tested on CI
- Future
  - Automated System Tests (AIP-21)
  - Split Airflow (AIP-8)?





Automated release notes for backport packages



Compatibility

For full compatibility and test status of the backport packages check Airflow Backport Package Compatibility

PIP requirements

PIP package	Version required
snowflake-connector-python	>=1.5.2
snowflake-sqlalchemy	>=1.1.0

Provider class summary

All classes in Airflow 2.0 are in `airflow.providers.snowflake` package.

Operators

New operators

New Airflow 2.0 operators: `airflow.providers.snowflake` package
operators.s3_to_snowflake.S3ToSnowflakeTransfer
operators.snowflake.SnowflakeOperator

Hooks

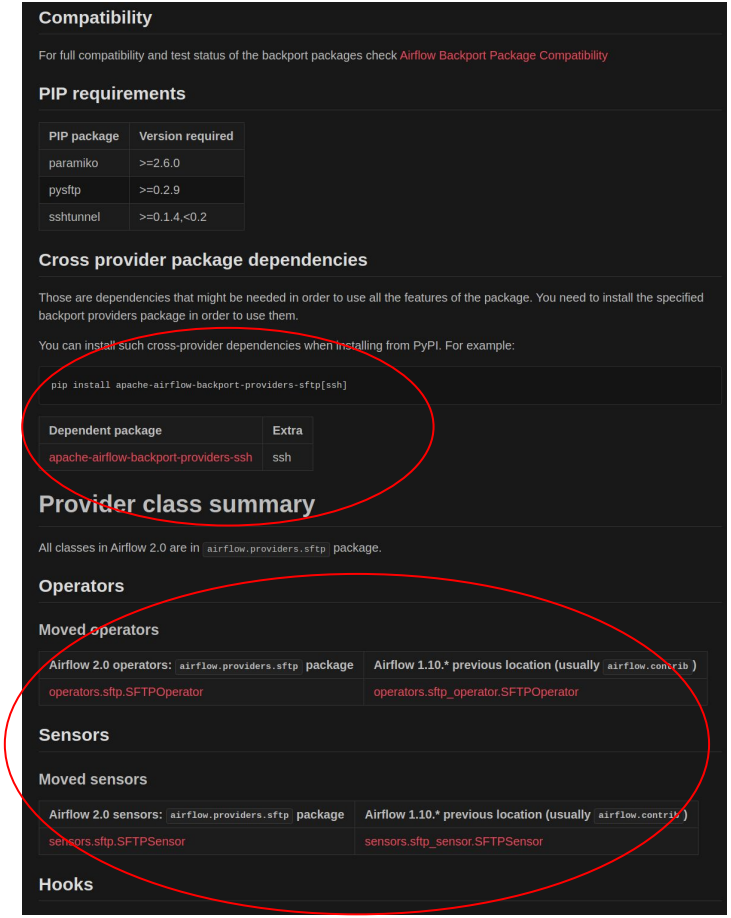
New hooks

New Airflow 2.0 hooks: `airflow.providers.snowflake` package
hooks.snowflake.SnowflakeHook

Releases

Release 2020.05.11

Commit	Committed	Subject
cd635dd7d	2020-05-10	[AIRFLOW-5906] Add authenticator parameter to snowflake_hook (#8642)
297ad3088	2020-04-20	Fix Snowflake hook conn id (#8423)
cf1109d66	2020-02-07	[AIRFLOW-6755] Fix snowflake hook bug and tests (#7380)



Compatibility

For full compatibility and test status of the backport packages check Airflow Backport Package Compatibility

PIP requirements

PIP package	Version required
paramiko	>=2.6.0
pysftp	>=0.2.9
sshtunnel	>=0.1.4,<0.2

Cross provider package dependencies

Those are dependencies that might be needed in order to use all the features of the package. You need to install the specified backport providers package in order to use them.

You can install such cross-provider dependencies when installing from PyPI. For example:

```
pip install apache-airflow-backport-providers-sftp[ssh]
```

Dependent package	Extra
apache-airflow-backport-providers-ssh	ssh

Provider class summary

All classes in Airflow 2.0 are in `airflow.providers.sftp` package.

Operators

Moved operators

Airflow 2.0 operators: `airflow.providers.sftp` package	Airflow 1.10.* previous location (usually `airflow.contrib`)
operators.sftp.SFTPOperator	operators.sftp_operator.SFTPOperator

Sensors

Moved sensors

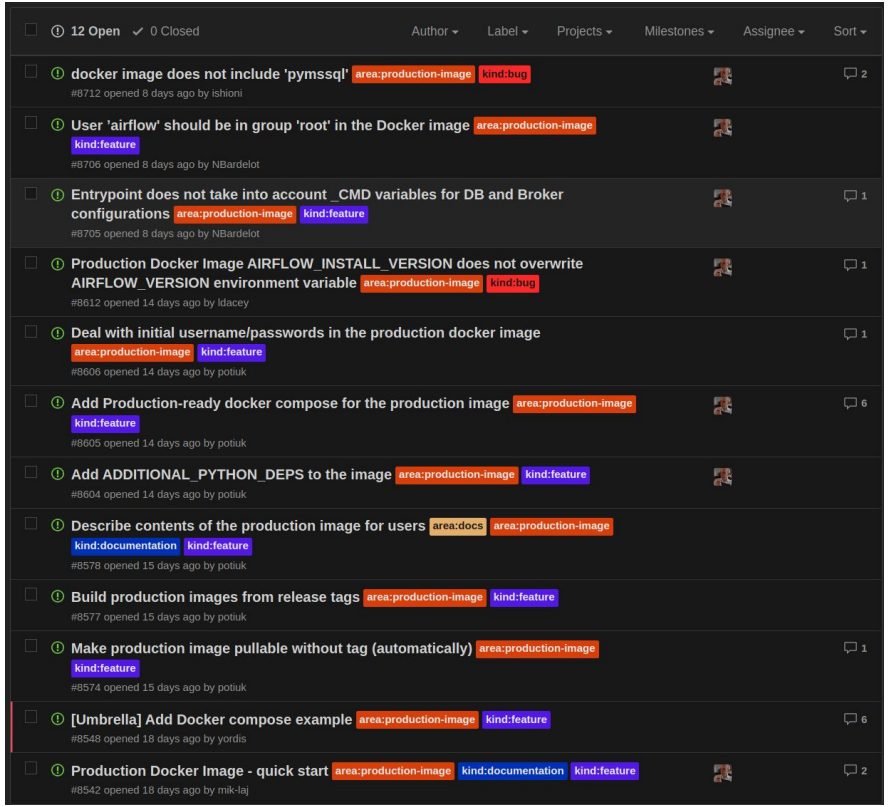
Airflow 2.0 sensors: `airflow.providers.sftp` package	Airflow 1.10.* previous location (usually `airflow.contrib`)
sensors.sftp.SFTPSensor	sensors.sftp_sensor.SFTPSensor

Hooks

Support for Production Deployments

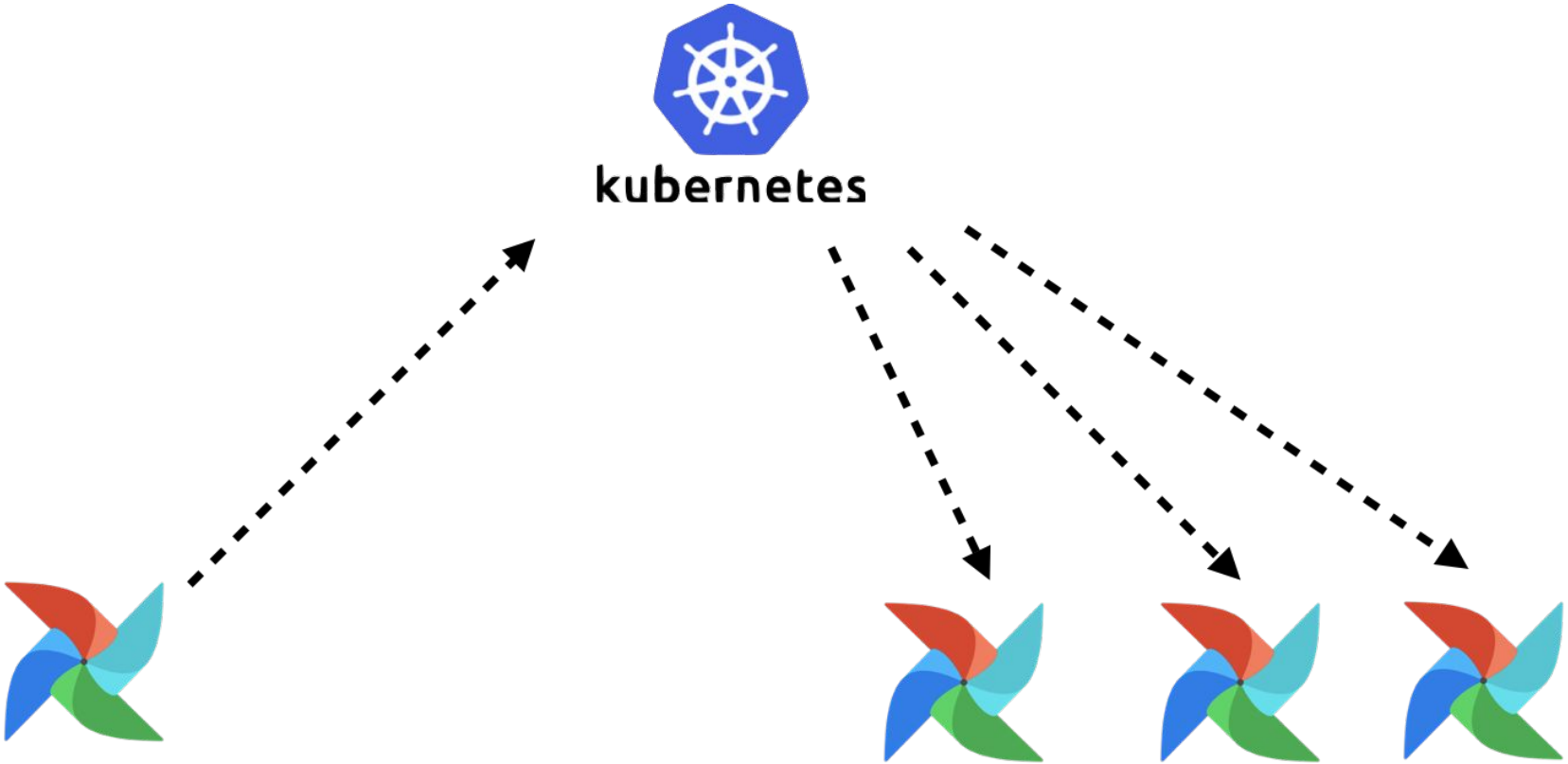
Production Image

- Alpha quality image is ready
- Gathering feedback
- Started with “bare image”
- Listening to use cases from users
- Integration with Docker Compose
- Integration with Helm Chart

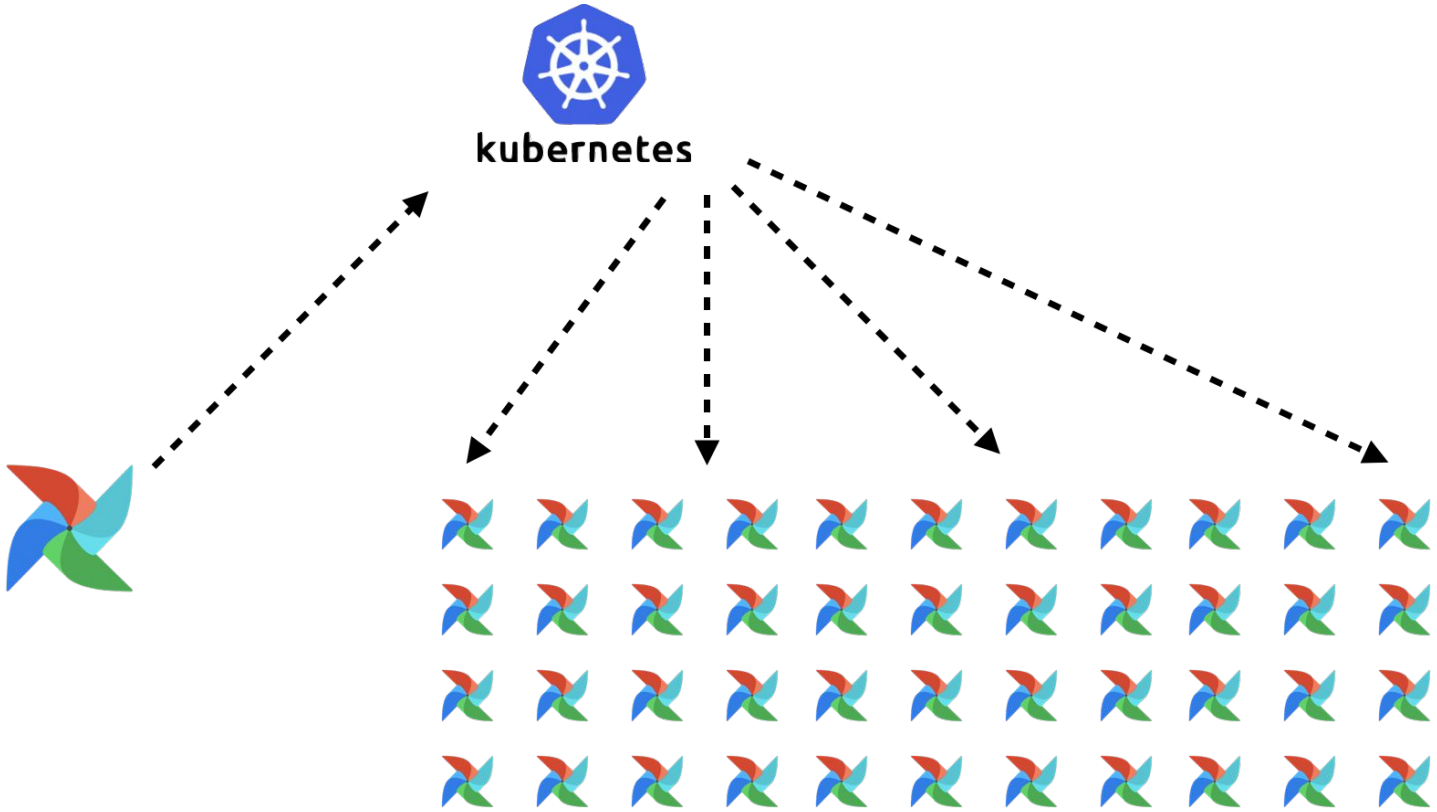


KEDA Autoscaling

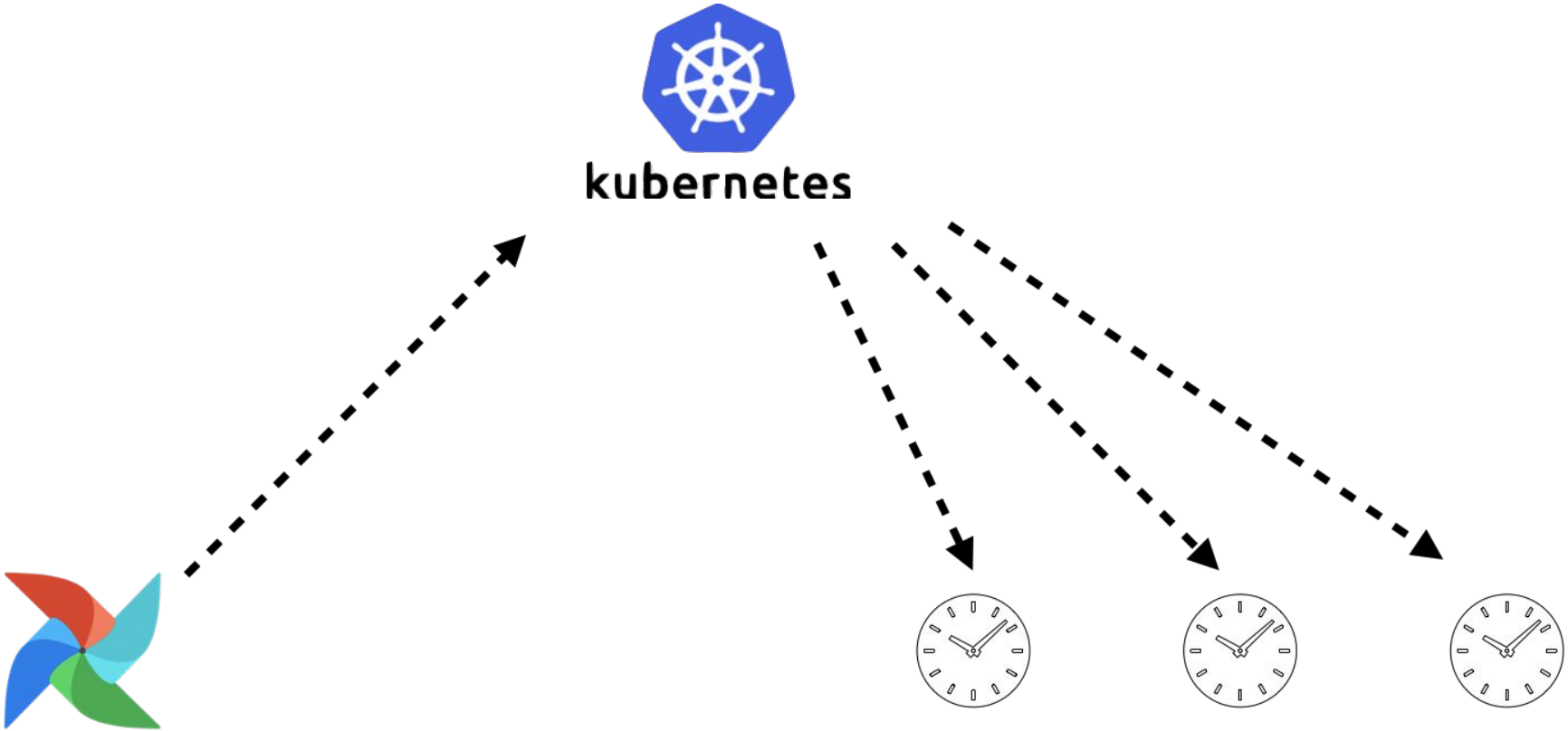
KubernetesExecutor



KubernetesExecutor



KubernetesExecutor



KubernetesExecutor vs. CeleryExecutor

KubernetesExecutor

- Dynamic Allocation
- executor_config

CeleryExecutor

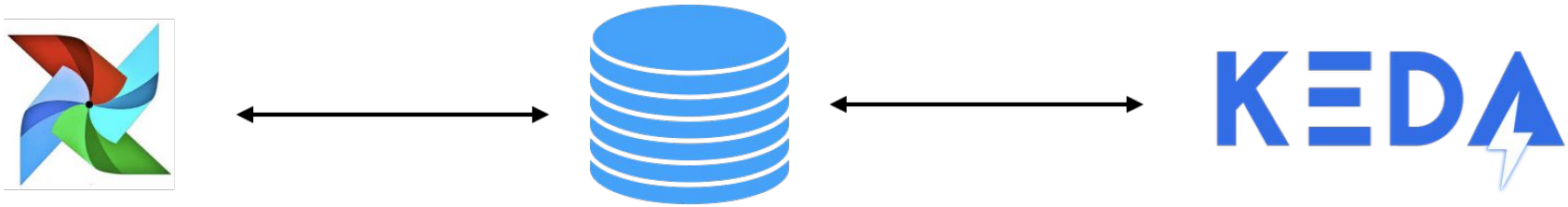
- Immediate SLAs
- Multiple tasks per-worker


KEDA

KEDA Autoscaling

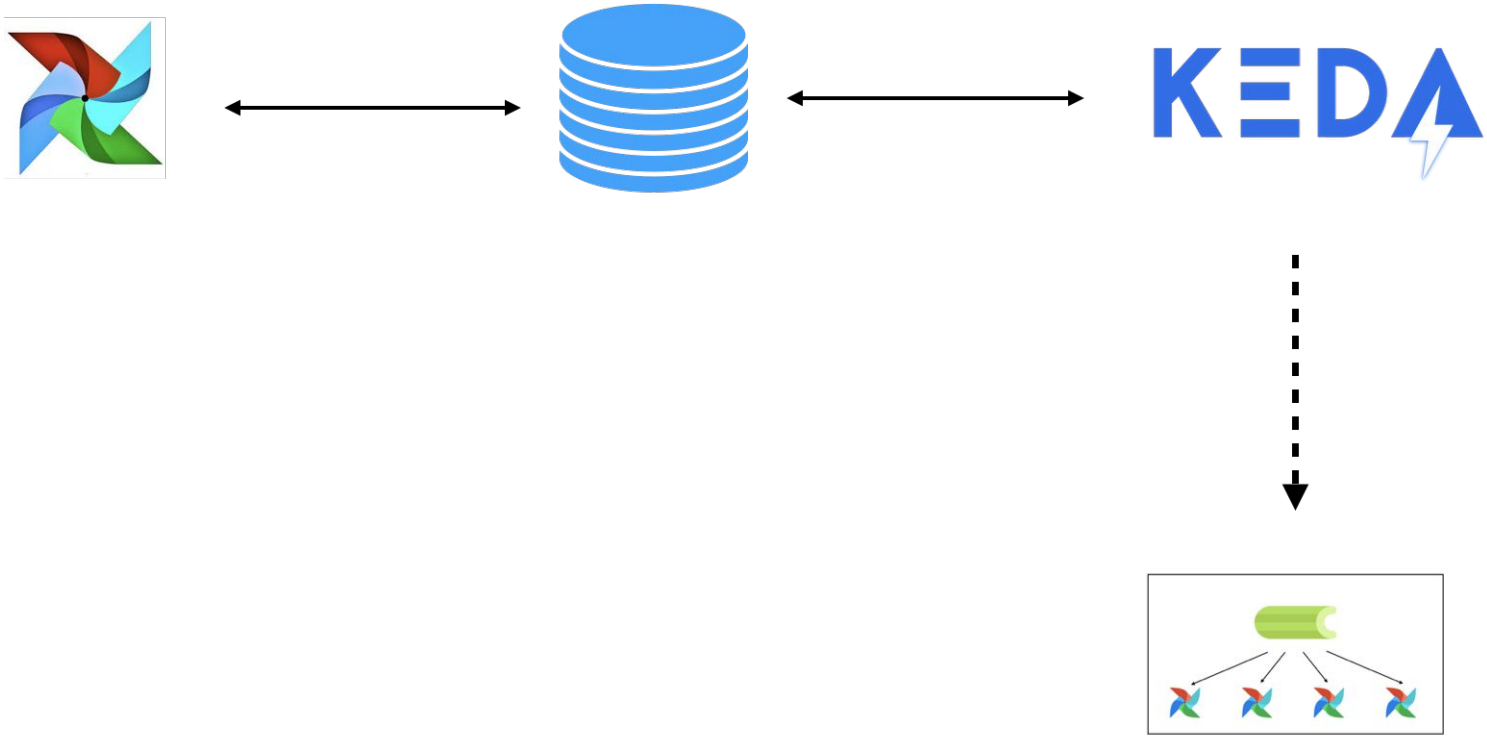
- Kubernetes Event-driven Autoscaler
- Scales based on # of RUNNING and QUEUED tasks in PostgreSQL backend

KEDA Autoscaling



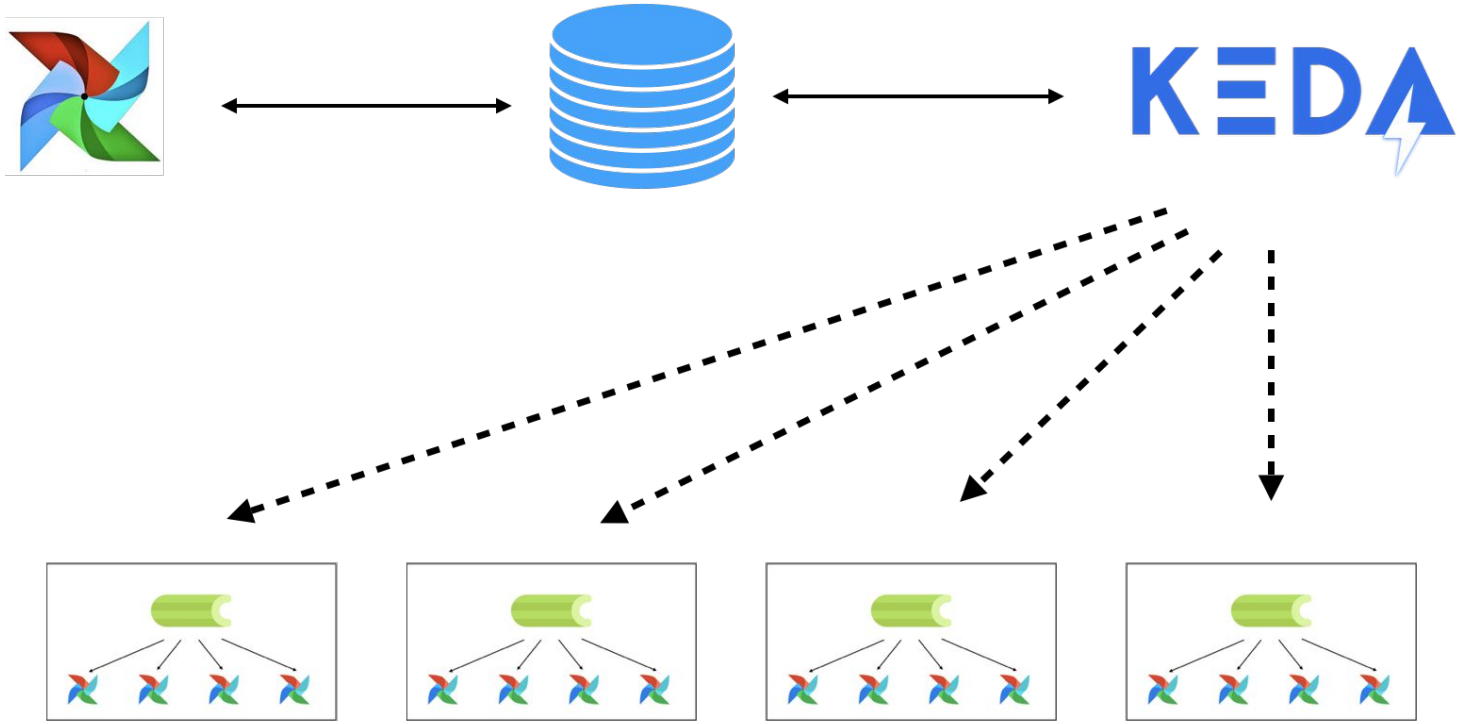
CEIL((0 RUNNING + 0 QUEUED)/16) = 0 workers

KEDA Autoscaling



CEIL((0 RUNNING + 1 QUEUED)/16) = 1 workers

KEDA Autoscaling



CEIL((20 RUNNING + 20 QUEUED)/16) = 4 workers

KEDA Queues

- Historically Queues were expensive and hard to allocate
- With KEDA, queues are free! (can have 100 queues)
- KEDA works with k8s deployments so any customization you can make in a k8s pod, you can make in a k8s queue (worker size, GPU, secrets, etc.)

KubernetesExecutor Pod Templating from YAML/JSON

KubernetesExecutor Pod Templating

- In the K8sExecutor currently, users can modify certain parts of the pod, but many features of the k8s API are abstracted away
- We did this because at the time the airflow community was not well acquainted with the k8s API
- We want to enable users to modify their worker pods to better match their use-cases

KubernetesExecutor Pod Templating

- Users can now set the pod_template_file config in their airflow.cfg
- Given a path, the KubernetesExecutor will now parse the yaml file when launching a worker pod
- Huge thank you to @davlum for this feature

Official Airflow Helm Chart

Helm Chart

- Donated by astronomer.io.
- This is the official helm chart that we have used both in our enterprise and in our cloud offerings (thousands of deployments of varying sizes)
- Users can turn on KEDA autoscaling through helm variables

Helm Chart

- Chart will cut new releases with each airflow release
- Will be tested on official docker image
- Significantly simplifies airflow onboarding process for Kubernetes users

DAG authoring "sugar"

Functional DAGs

```python
def get_cat_pictures(num: int) -> List[Dict]:
    response = requests.get("https://cat_pictures.com", params={"num": num})
    return response.json()["cats"]

def save_cats(list_of_cats: List[Dict]) -> None:
    for cat in list_of_cats:
        save_it_somehow(cat)

with DAG("cat_fetcher"):
    get_task = PythonOperator(
      task_id="get_task", python_callable=get_cat_pictures, op_args=[42]
    )
    cats = "{{ task_instance.xcom_pull('get_task') }}"
    save_task = PythonOperator(
      task_id="save_task", python_callable=save_cats, op_args=[cats]
    )
    get_task >> save_task
```

- ➔ PythonOperator boilerplate code
- ➔ Define order and data relation separately
- ➔ Writing jinja strings by hand

Functional DAGs

No PythonOperator boilerplate code!

```python
def get_cat_pictures(num: int) -> List[Dict]:
    response = requests.get("https://cat_pictures.com", params={"num": num})
    return response.json()["cats"]

def save_cats(list_of_cats: List[Dict]) -> None:
    for cat in list_of_cats:
        save_it_somehow(cat)

with DAG("cat_fetcher"):
    get_task = PythonOperator(
      task_id="get_task", python_callable=get_cat_pictures, op_args=[42]
    )
    cats = "{{ task_instance.xcom_pull('get_task') }}"
    save_task = PythonOperator(
      task_id="save_task", python_callable=save_cats, op_args=[cats]
    )
    get_task >> save_task
```

```python
@task
def get_cat_pictures(num: int) -> List[Dict]:
    response = requests.get("https://cat_pictures.com", params={"num": num})
    return response.json()["cats"]

@task
def save_cats(list_of_cats: List[Dict]) -> None:
    for cat in list_of_cats:
        save_it_somehow(cat)


with DAG("cat_fetcher"):
    get_task = get_cat_pictures(42)
    save_task = save_cats(get_task)
```

Data and order relationship are same!

And works for all operators

Functional DAGs

AIP-31: Airflow functional DAG definition

- ➔ Easy way to convert a function to an operator
- ➔ Simplified way of writing DAGs
- ➔ Pluggable XCom Storage engine

```python
@task
def get_cat_pictures(num: int) -> List[Dict]:
    response = requests.get("https://cat_pictures.com", params={"num": num})
    return response.json()["cats"]

@task
def save_cats(list_of_cats: List[Dict]) -> None:
    for cat in list_of_cats:
        save_it_somehow(cat)


with DAG("cat_fetcher"):
    get_task = get_cat_pictures(42)
    save_task = save_cats(get_task)
```

Example: store and retrieve DataFrames on GCS or S3 buckets without boilerplate code

Smaller changes

Other changes of note

- Connection IDs now need to be unique

  It was often confusing, and there are better ways to do load balancing

- Python 3 only

  Python 2.7 unsupported upstream since Jan 1, 2020

- "RBAC" UI is now the only UI.

  Was a config option before, now only option. Charts/data profiling removed due to security risks

Road to Airflow 2.0

When will Airflow 2.0 be available?

Airflow 2.0 – deprecate, but (try) not to remove

- Breaking changes should be avoided where we can – if upgrade is to difficult users will be left behind
- Release "backport providers" to make new code layout available "now":

```
pip install apache-airflow-backport-providers-aws  \
            apache-airflow-backport-providers-google
```

- Before 2.0 we want to make sure we've fixed everything we want to remove or break.

How to upgrade to 2.0 safely

- Install the latest 1.10 release
- Run `airflow upgrade-check` (doesn't exist yet)
- Fix any warnings
- Upgrade Airflow


Airflow
Summit 2020
July 6th-17th, 2020.
Join from anywhere.
https://airflowsummit.org

Thank you!

Time for Q & A