

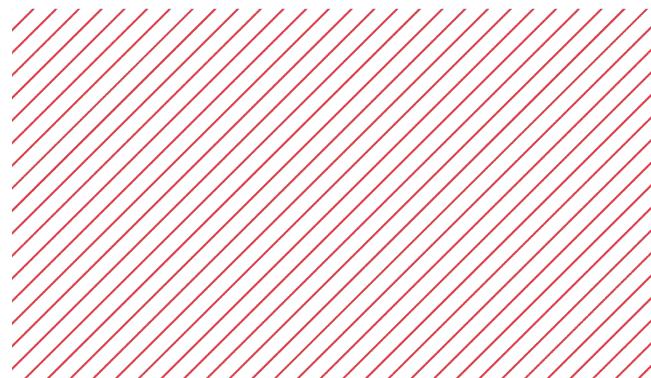
академия  
больших  
данных



mail.ru  
group

## Data Pipelines

Михаил Марюфич, MLE





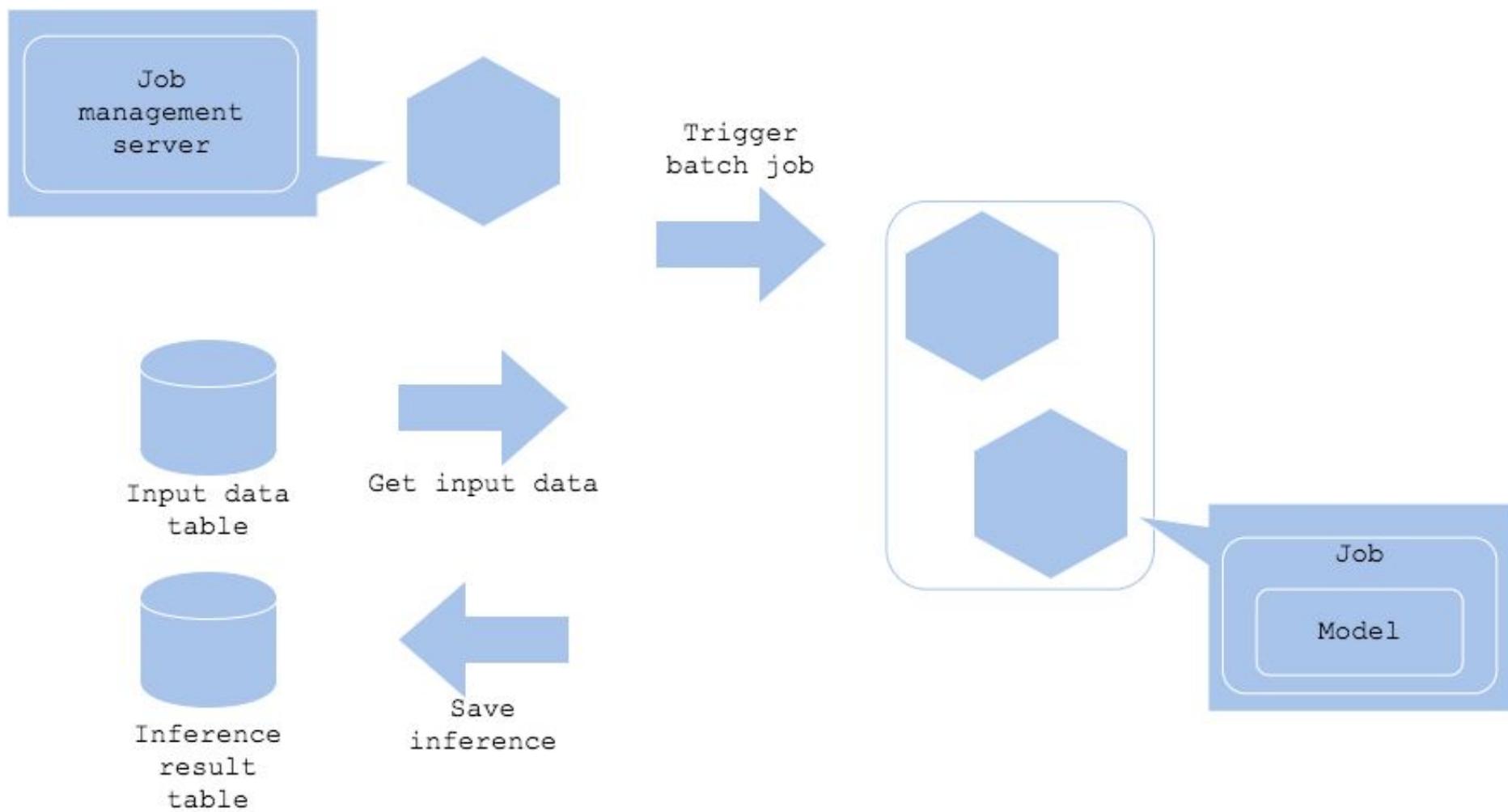
# Содержание занятия

---

- Зачем нужны и что такое оркестраторы данных
- Что такое Airflow
- Разворачиваем Airflow
- Пишем даги на airflow
- Best practice написание дагов на airflow

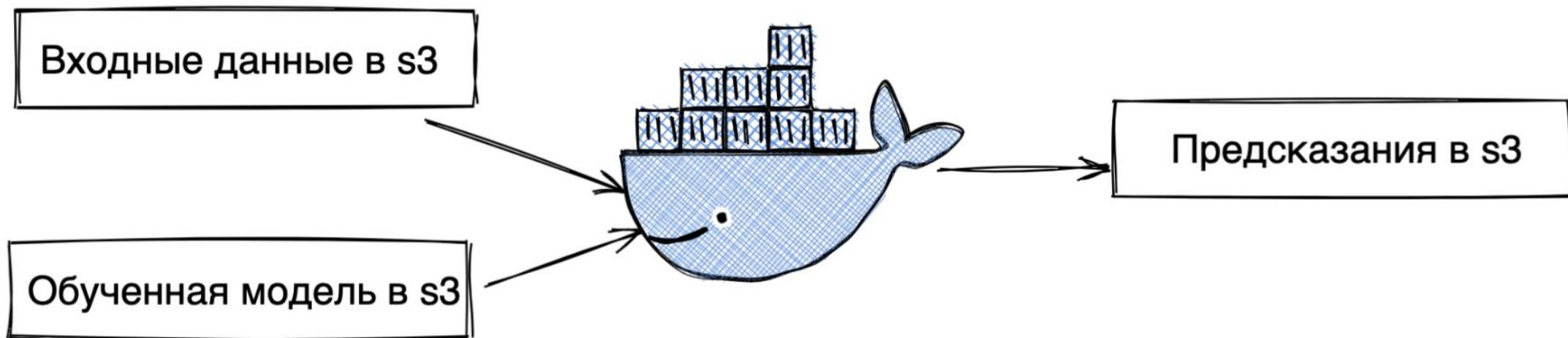
# О пакетном паттерне

# Пакетный паттерн



# Batch инференс модели в docker

---



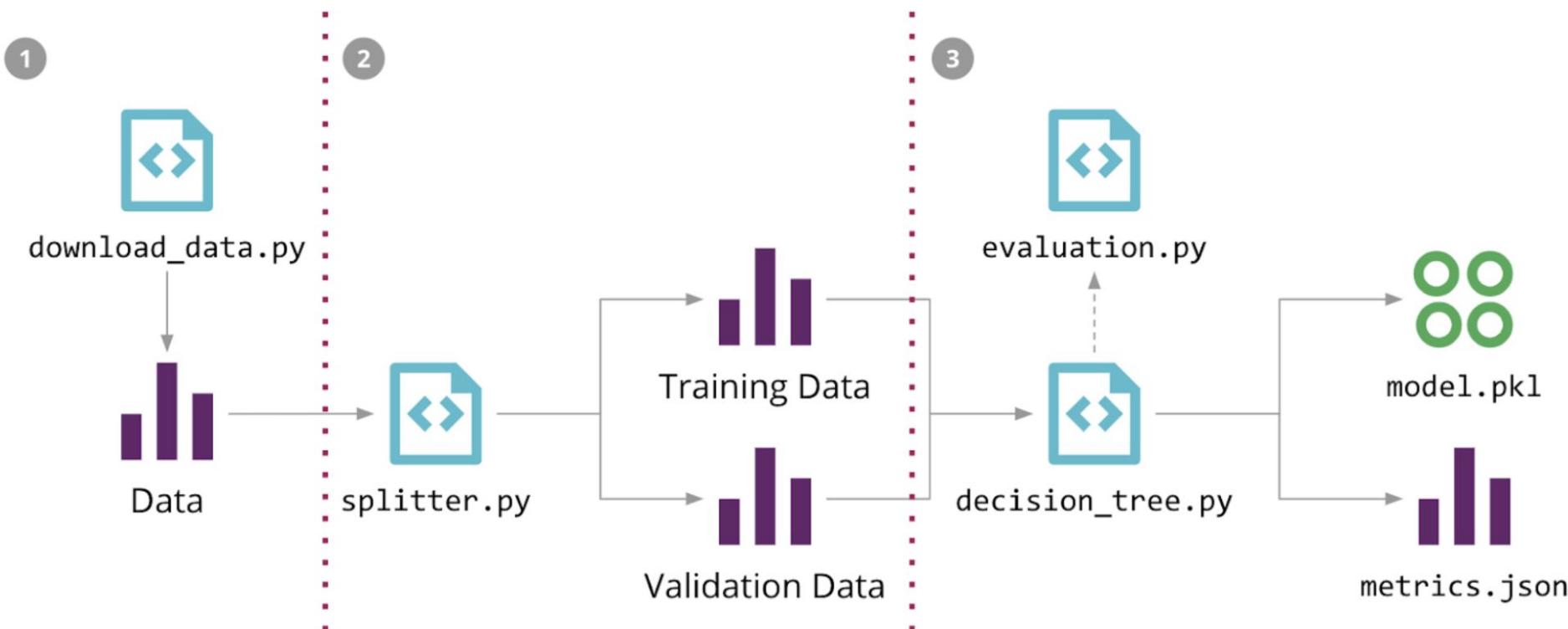


# Как это обычно выглядит?

---

- 1) Считываются фичи по данным за предыдущий день (**много джоб**)
- 2) Берем модель (обучаем модель) и делаем предсказание
- 3) Смотрим на предсказания, если их распределение норм, то заливаем в **прод**, если нет, то посылаем предупреждение, о том, что-то пошло не так.
- 4) Делать надо **каждый день**

# ML PIPELINE



# CRON

---

**Cron** - unix утилита, используемая для периодического выполнения задач в определенное время.

Регулярные действия описываются инструкциями в **crontab** и специальных каталогах

# CRONTAB

```
worker1: ~ cronitor list
```

```
▶ Reading user "ubuntu" crontab
```

SCHEDULE	COMMAND
0 1 * * *	/var/app/edi/send_batch_invoices.py
0 2 * * *	/var/app/edi/send_batch_settlement.py
*/20 * * * *	/var/app/edi/reconciler.sh --full
* * * * *	/var/app/reporting/rollup minute
0 * * * *	/var/app/reporting/rollup hour
0 0 * * *	/var/app/reporting/rollup day
5 0 * * *	/var/app/reporting/rollup archive-ancient-data

```
▶ Reading /etc/crontab
```

SCHEDULE	COMMAND
0 * * * *	/tmp/foo.sh --systemcrontab

```
worker1: ~ █
```

# CRONTAB





# Как можно применить CRON?

---

Помним, что нам нужно запустить несколько задач подряд

- 1) определяем время старта и конца каждой таски и запускаем несколько cron-job
- 2) запихиваем все таски в один cronjob

spark-submit job1

spark-submit job2

etc

# Минусы CRON

---

- трудно мониторить
- нет автоматических перезапусков
- нельзя явно задавать связи между задачами

```
#30 2 * * * spark-submit --master yarn --queue root.daily --executor-cores=8 --executor-memory=16G --class BanReport /mnt/h  
#30 4 * * * spark-submit --master yarn --queue root.daily --executor-cores=4 --executor-memory=32G --class BanGenerator /mn  
30 5 * * * spark2-submit --master yarn --queue root.daily --executor-cores=4 --executor-memory=8G --class etl.RequestOtahot  
#30 6 * * * spark-submit --master yarn --queue root.daily --executor-cores=8 --executor-memory=16G --class etl.AvailScore /  
#10 * * * * spark-submit --master yarn --executor-cores=4 --executor-memory=8G --class etl.DAPIToParquet /mnt/hadoop/spark-  
#31 * * * * spark-submit --master yarn --executor-cores=4 --executor-memory=8G --class DiscrepancyHunter /mnt/hadoop/spark-  
#33 * * * * spark-submit --master yarn --executor-cores=8 --executor-memory=16G --class DiscrepancyInStep /mnt/hadoop/spark-  
20 * * * * . /etc/profile; /mnt/hdfs/tools/spark-jars/exchange.py >> /root/exchange.log 2>&1  
#11 * * * * spark2-submit --master yarn --queue root.hourly --executor-cores=4 --executor-memory=8G --class etl.RawDataConv  
#26 * * * * spark-submit --master yarn --queue root.hourly --executor-cores=4 --executor-memory=8G --class etl.MinRates /mn  
#32 * * * * spark2-submit --master yarn --queue root.hourly --executor-cores=4 --executor-memory=8G --class etl.Discrepancy  
32 * * * * spark-submit --master yarn --queue root.hourly --executor-cores=4 --executor-memory=8G --class etl.DiscrepancyHu  
#40 * * * * spark-submit --master yarn --queue root.hourly --executor-cores=4 --executor-memory=8G --class etl.RAIndex /mnt
```



# Чего нам хотелось бы?

---

- возможность запускаться по расписанию
- запускать задачи друг за другом в зависимости от чего-либо
- мониторинг, если что-то упало и перезапуск



# Оркестраторы данных

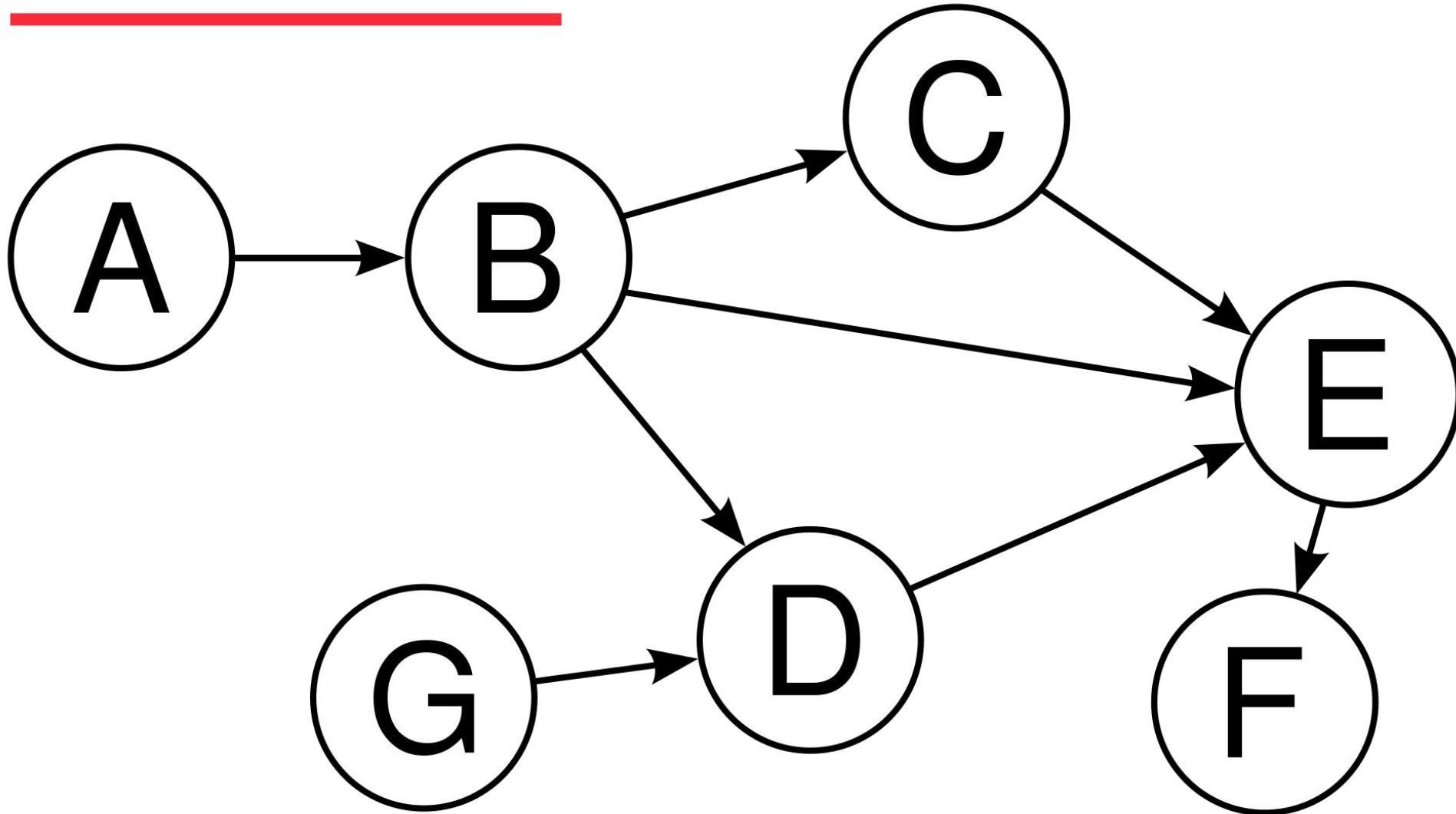
---

Оркестратор -- это штука, которая отвечает за:

- **Планирование задач** (когда запустить)
- **Управление зависимостями** (ждать пока исполняются другие задачи перед запуском)
- **Репроцессинг** - легко перезапускать упавшие задачи и зависящие от неё
- **Мониторинг** - если задача упала, то нужно об этом уведомить

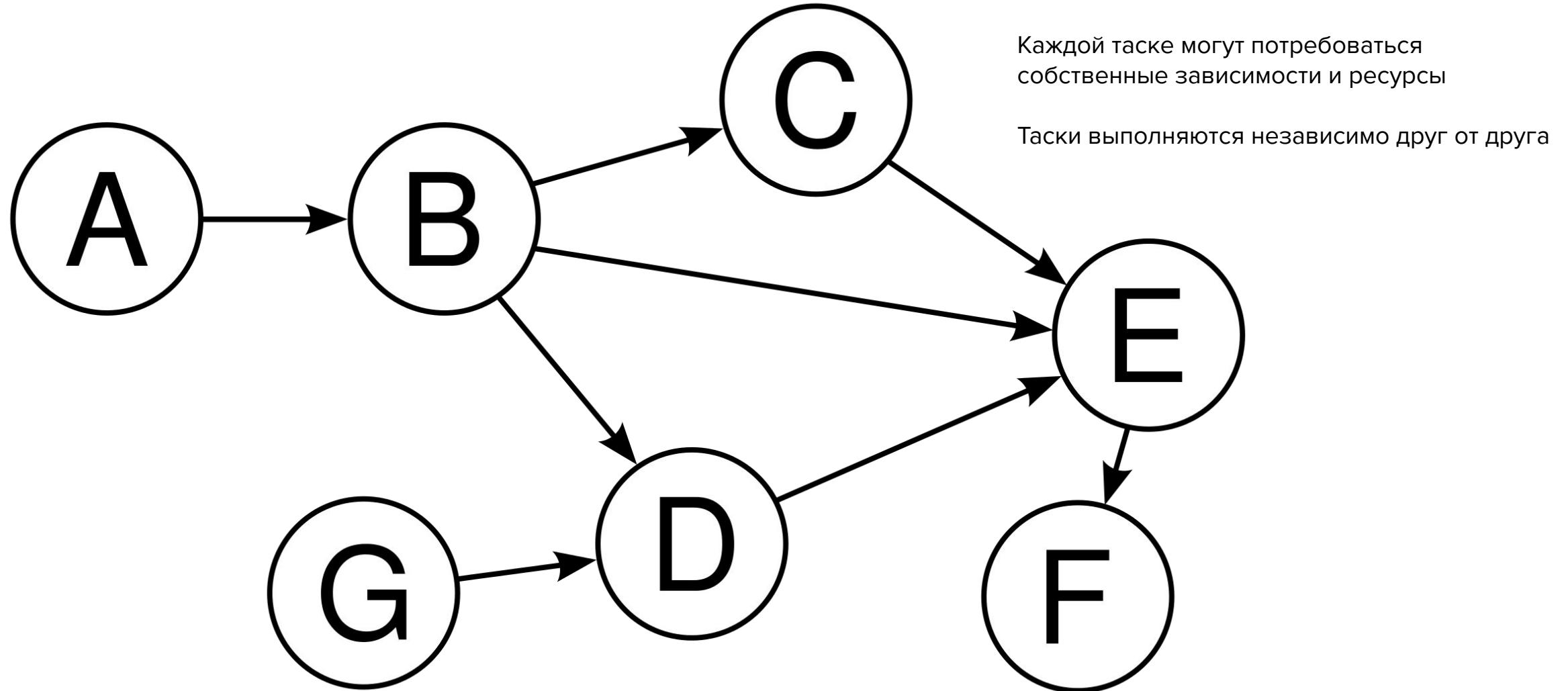
# DAG

---



# DAG

---





# Оркестраторы

---

- Apache Oozie
- Apache Airflow
- Luigi
- Dagster



# Итоги

---

1. Посмотрели на типичную batch ML задачу
2. Рассмотрели применимость(и неприменимость) cron
3. Пришли к идее оркестратора

# Apache Airflow

# Apache Airflow

---

Apache Airflow - платформа для автоматического управления задачами, их расписанием и мониторингом

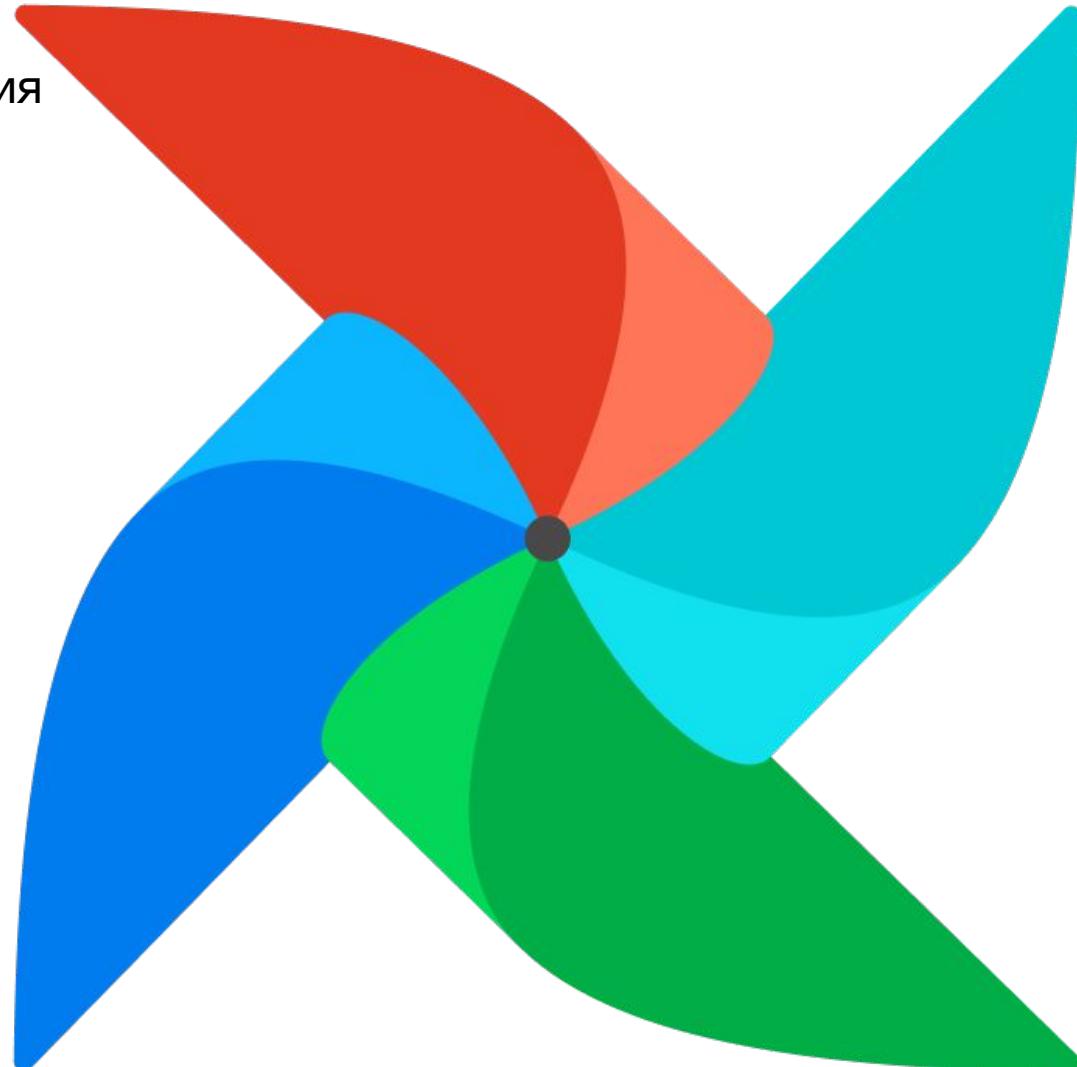
Разрабатывается с 2014 года(изначально в Airbnb  
<https://github.com/apache/airflow>

★ Star

18.6k

🍴 Fork

7.3k



# Как задаются даги в Airflow

- Даги в airflow задаются питоновским **кодом**
- Поддерживаются сложные штуки, вроде ветвлений и описания зависимостей из нескольких задач
- Легко расширяется кастомными задачами

```
dag = DAG(  
    "easy pipeline",  
    default_args=default_args,  
    description="An easy airflow DAG",  
    schedule_interval=timedelta(days=1),  
)  
  
task_1 = BashOperator(  
    task_id="print_date",  
    bash_command="date",  
    dag=dag)  
  
task_2 = BashOperator(  
    task_id="sleep",  
    depends_on_past=False,  
    bash_command="sleep 5",  
    retries=3,  
    dag=dag  
)  
  
task_1 >> task_2
```

# Интерфейс Airflow — даги

Airflow DAGs Security Browse Admin Docs 13:52 UTC PP

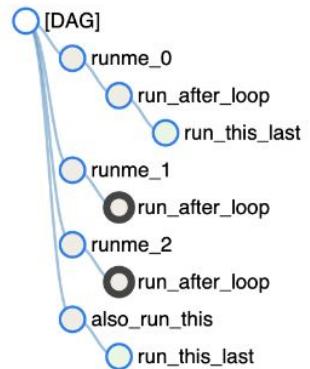
## DAGs

All 26	Active 3	Paused 23	Filter DAGs by tag	Search DAGs				
DAG		Owner	Runs	Schedule	Last Run	Recent Tasks	Actions	Links
example_bash_operator	example example2	airflow	2	0 0 * * *	2021-05-04, 00:00:00	6	▶ C ⚡	...
example_branch_dop_operator_v3	example	airflow	919	*1 * * * *	2021-05-03, 15:19:00	4	▶ C ⚡	...
example_branch_operator	example example2	airflow	0	@daily		0	▶ C ⚡	...
example_complex	example example2 example3	airflow	0	None		0	▶ C ⚡	...
example_dag_decorator	example	airflow	0	None		0	▶ C ⚡	...

# Интерфейс Airflow — задачи

● BashOperator ● DummyOperator

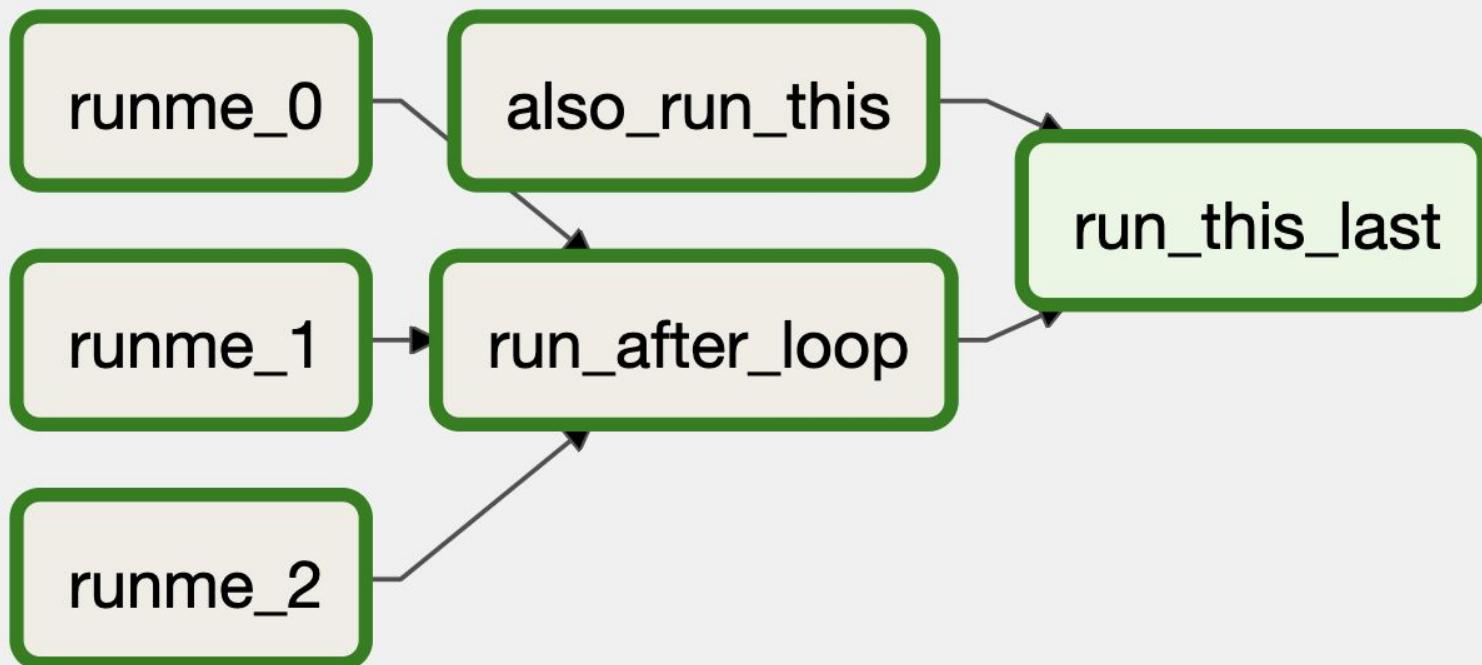
■ queued ■ running ■ success ■ failed ■ up\_for\_retry ■ up\_for\_reschedule ■ upstream\_failed ■



12:58:04



# Интерфейс Airflow — таски



DEMO: смотрим Airflow UI

# Из чего состоит AIRFLOW

# Airflow Scheduler

---

Парсит даги, заданные питоновским кодом, проверяет расписание, назначает задачам статус “scheduled”.



# Airflow Webserver

---

Обрабатывает пользовательские запросы, отрисовывает даги.



**Web Server**

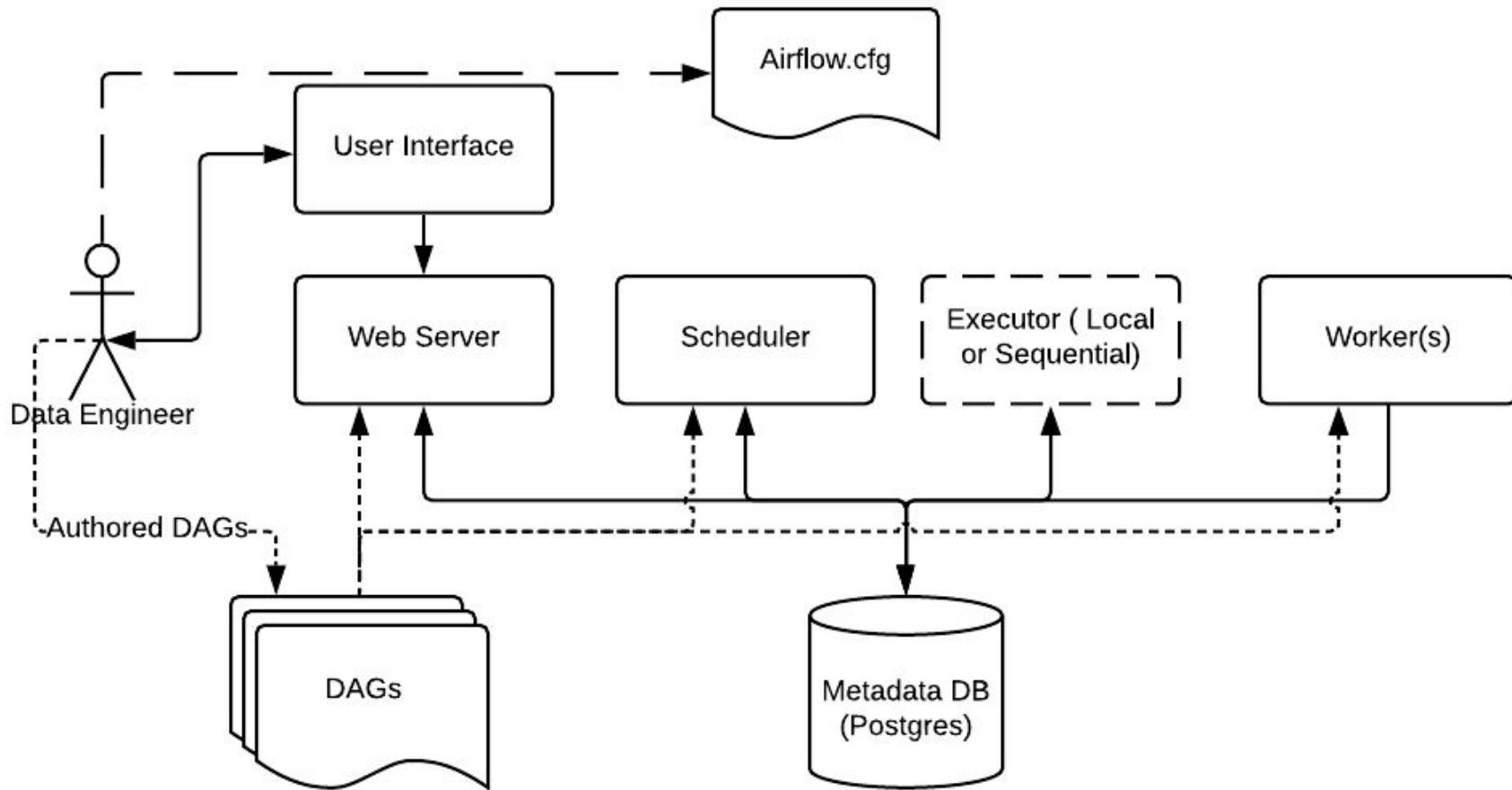
# Airflow Workers

---

Подбирают задачи, помеченный статусом “scheduled”. Несут ответственность за фактическое выполнение работы



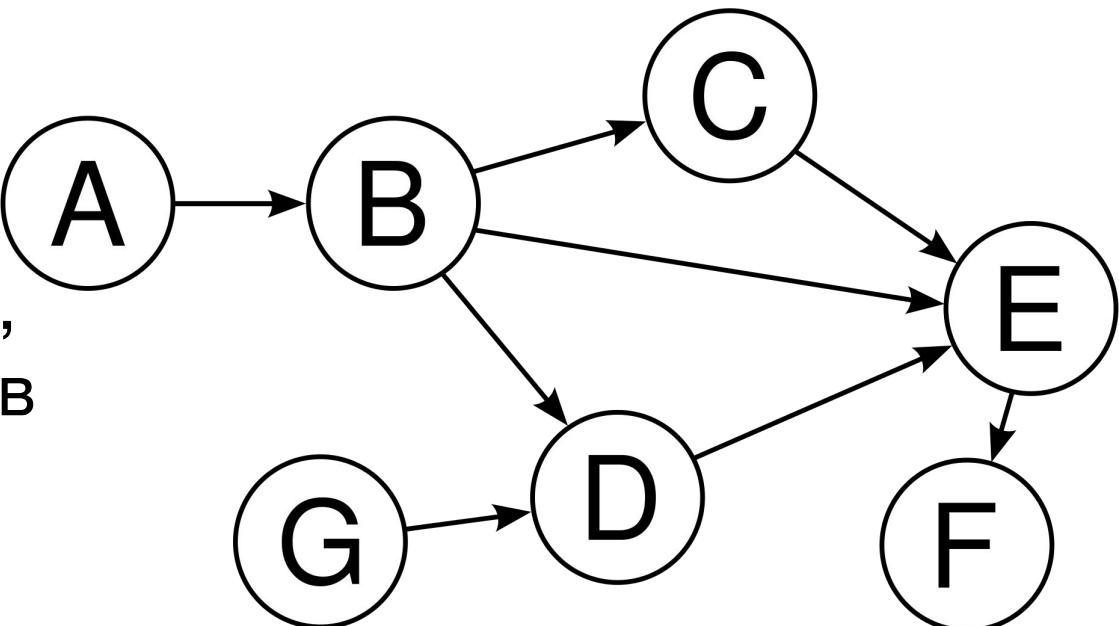
# Взаимодействие компонент



# Как работает scheduler

---

- 1) считать из папки /opt/airflow/dags/ питоновские модули – спарсить их в граф исполнения задач
- 2) Для каждого dag проверяет, не пришло ли время его выполнить, если пришло, то переводит dag в статус scheduled





# Как работает scheduler

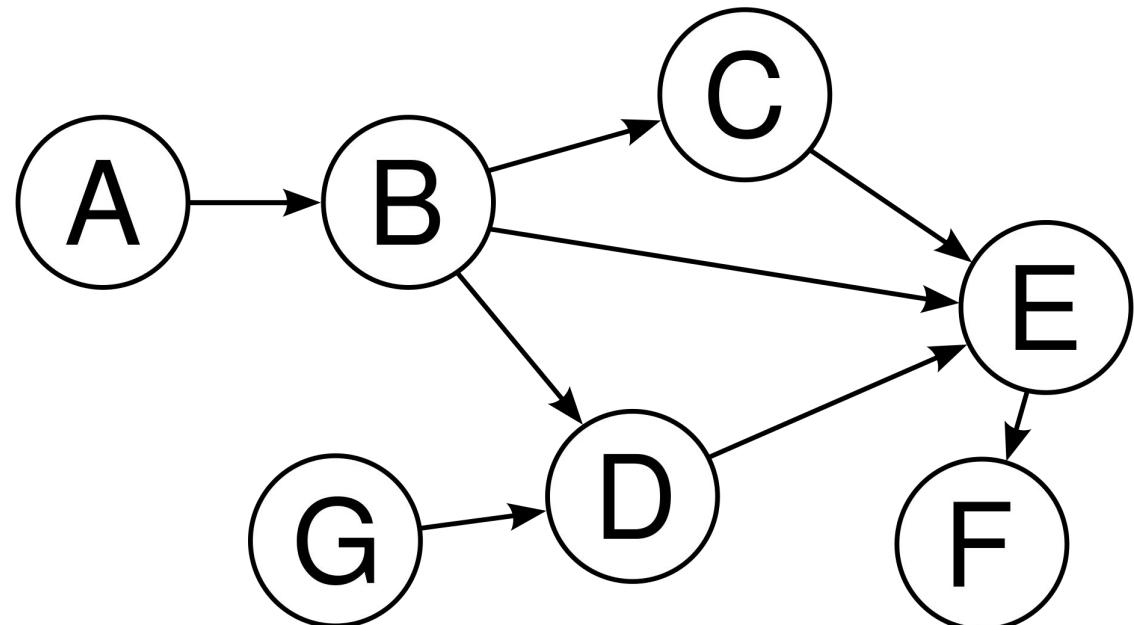
---

- 3) Для каждой задачи(task), которая находится в статусе scheduled проверяет, что все его зависимости(upstream) были завершены успешно, если да, то задача добавляется в очередь на исполнение.
- 4) Ждет какое-то время — и возвращается к шагу 1

# Как работает scheduler

---

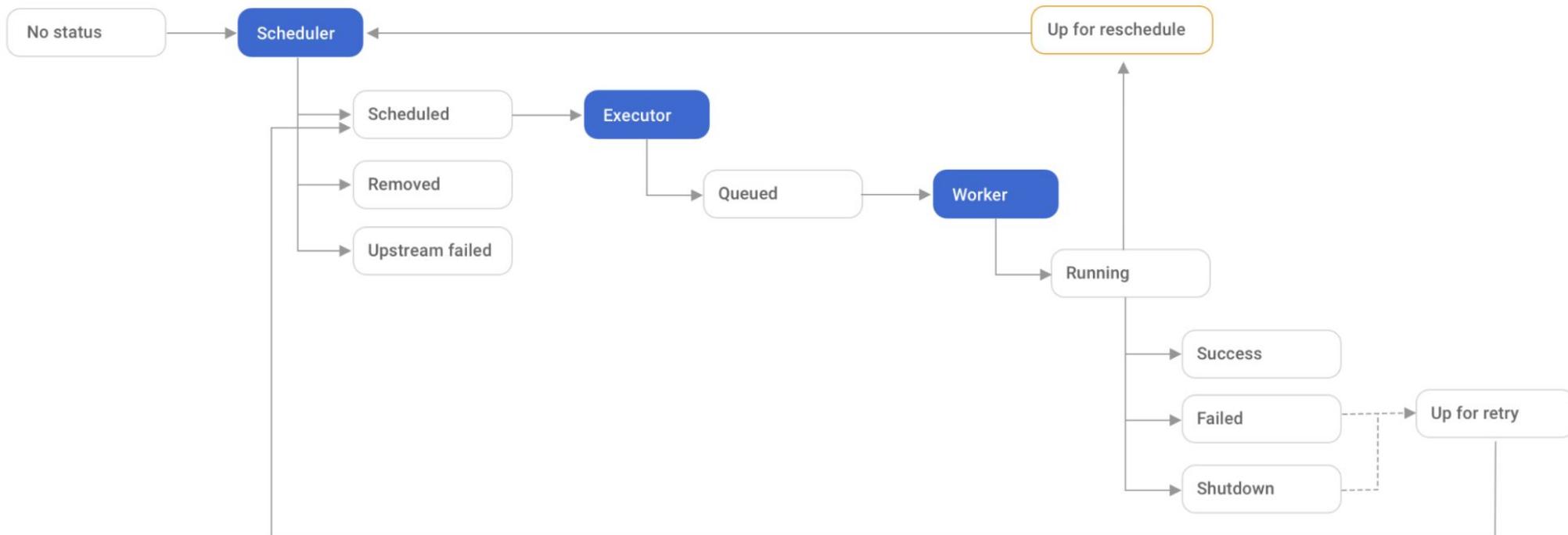
- 1) спарсить даги
- 2) Зашедулиить даги
- 3) Проверить зависимости тасок
- 4) Вернуться к 1



# Жизненный цикл задачи



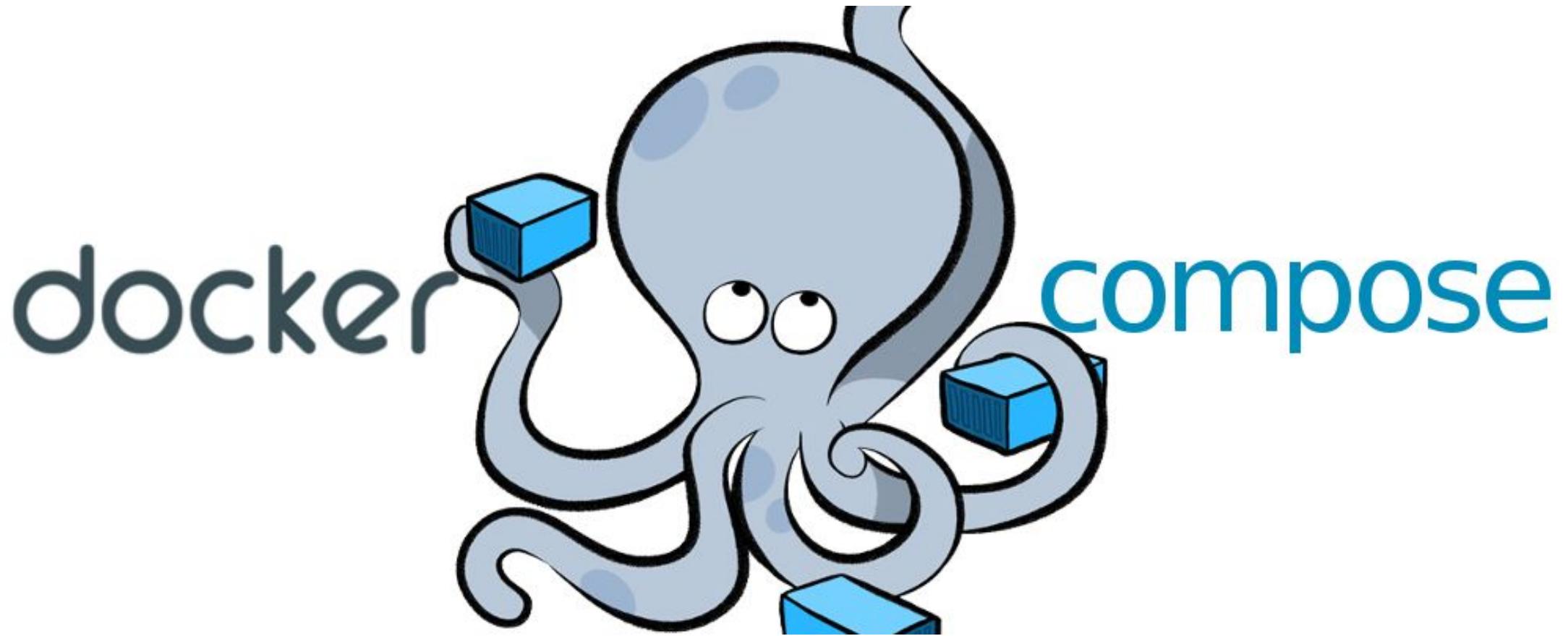
The complete lifecycle of the task looks like this:



**DEMO:** разворачиваем  
Airflow

# Docker Compose

---



<https://docs.docker.com/compose/gettingstarted/>

# Docker Compose

```
version: '3'
services:
  app:
    build:
      context: ./app
      dockerfile: Dockerfile
    volumes:
      - /datastore/app:/app
    ports:
      - "5000:5000"
      - "9001:9001"
      - "80:80"
    depends_on:
      - influxdb
  influxdb:
    image: influxdb
    volumes:
      - /datastore/influx:/var/lib/influxdb
    ports:
      - "8086:8086"
  grafana:
    build:
      context: ./grafana
      dockerfile: Dockerfile
    volumes:
      - /datastore/grafana:/var/lib/grafana
    ports:
      - "3000:3000"
```

docker compose up  
docker compose down  
docker compose build

<https://docs.docker.com/compose/gettingstarted/>

# Возможности для написания DAG

# DAG

Сущность, агрегирующая задачи(task).

В airflow задается классом DAG, в который можно передать всякую метаинформацию и расписание.

```
default_args = {  
    "owner": "airflow",  
    "email": ["airflow@example.com"],  
    "retries": 1,  
    "retry_delay": timedelta(minutes=5),  
}  
  
dag = DAG(  
    "docker_sample",  
    default_args=default_args,  
    schedule_interval=timedelta(minutes=10),  
    start_date=days_ago(2),  
)
```

# DAG – способы задания

```
default_args = {  
    'start_date': datetime(2016, 1, 1),  
    'owner': 'airflow'  
}  
  
dag = DAG('my_dag', default_args=default_args)  
op = DummyOperator(task_id='dummy', dag=dag)  
print(op.owner) # airflow
```

```
with DAG('my_dag', start_date=datetime(2016, 1, 1)) as dag:  
    op = DummyOperator('op')  
  
op.dag is dag # True
```

# ЧТО МЫ МОЖЕМ ЗАПУСКАТЬ?

За то что мы можем запускать в Airflow отвечает сущность

## Operator

### Примеры:

- BashOperator
- SparkSubmitOperator
- PythonOperator
- SimpleHttpOperator
- PostgresOperator
- HiveOperator
- DockerOperator
- SSHOperator
- SlackAPIOperator

```
from airflow.models.baseoperator import BaseOperator
from airflow.utils.decorators import apply_defaults

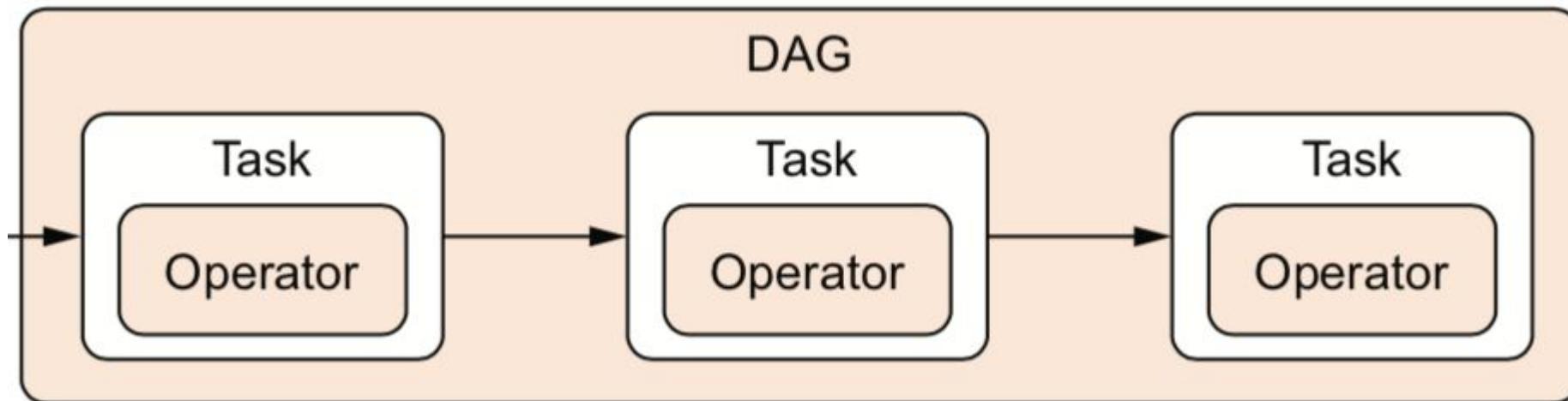
class HelloOperator(BaseOperator):
    @apply_defaults
    def __init__(self, name: str, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.name = name

    def execute(self, context):
        message = "Hello {}".format(self.name)
        print(message)
        return message
```

# Operator VS Task

---

Task это внутренняя сущность, которая следит за состоянием оператора и предоставляет пользователю статусы **started/failed/finished/etc**



# BashOperator

---

Позволяет исполнить любую команду bash

```
t2 = BashOperator(  
    task_id="sleep", bash_command="sleep 1", dag=dag,  
)
```



# Как выстраивать зависимости

---

```
download >> preprocess >> predict
```

```
download << preprocess << predict
```

```
download.set_downstream(preprocess)  
preprocess.set_downstream(predict)
```

```
background-color: #fff;
text-shadow: 0px -1px 0px #000;
filter: dropshadow(color:#000);
color:#777;

}

header #main-navigation ul li span:hover,
header #main-navigation ul li span:active {
  border: 1px solid #000;
  background-color: #F9F9F9;
  box-shadow: 0px 0px 1px #000;
  -webkit-box-shadow: 0px 0px 2px #000;
  -moz-box-shadow: 0px 0px 1px #000;
}
```

DEMO: пишем и запускаем собственный  
простой dag с bash operator, работаем с  
заданием зависимостей

# PythonOperator

---

Позволяет выполнять произвольный Python код

```
def print_context(ds, **kwargs):
    """Print the Airflow context and ds variable from the context."""
    pprint(kwargs)
    print(ds)
    return 'Whatever you return gets printed in the logs'

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

DEMO: dag c python  
operator про самолетики

# BranchPythonOperator

---

Позволяет выбирать ветки исполнения в зависимости от кода на python



```
{ if  
  else }
```

# Trigger Rule

---

## Параметр BaseOperator

`all_success` : (default) all parents have succeeded

`all_failed` : all parents are in a `failed` or `upstream_failed` state

`all_done` : all parents are done with their execution

`one_failed` : fires as soon as at least one parent has failed, it does not wait for all parents to be done

`one_success` : fires as soon as at least one parent succeeds, it does not wait for all parents to be done

`none_failed` : all parents have not failed ( `failed` or `upstream_failed` ) i.e. all parents have succeeded or been skipped

`none_failed_or_skipped` : all parents have not failed ( `failed` or `upstream_failed` ) and at least one parent has succeeded.

`none_skipped` : no parent is in a `skipped` state, i.e. all parents are in a `success` , `failed` , or `upstream_failed` state

`dummy` : dependencies are just for show, trigger at will

## DEMO: branching operator

# XCOM

Позволяет сохранять и передавать информацию между тасками

The screenshot shows the Airflow interface with a teal header bar. The header includes the Airflow logo, navigation links for DAGs, Data Profiling, Browse, Admin, Docs, and About, and a timestamp on the right: 2019-11-07 05:18:27 UTC.

The main content area is titled "Xcoms". It features a table with the following columns: Key, Value, Timestamp, Execution Date, Task ID, and Dag ID. The table lists seven entries, all associated with the DAG "test\_dag" and the task "xcom\_push\_task", under the "write\_to\_xcom" DAG ID. Each entry has a timestamp between November 6 and 7, 2019, and a corresponding execution date.

	Key	Value	Timestamp	Execution Date	Task ID	Dag ID	
<input type="checkbox"/>		test_dag	{"key1": "value1"}	2019-11-06 07:04:42.534514+00:00	2019-11-05 00:48:00+00:00	xcom_push_task	write_to_xcom
<input type="checkbox"/>		test_dag	{"key1": "value1"}	2019-11-06 07:04:47.838219+00:00	2019-11-05 00:51:00+00:00	xcom_push_task	write_to_xcom
<input type="checkbox"/>		test_dag	{"key1": "value1"}	2019-11-06 07:04:51.009360+00:00	2019-11-05 00:39:00+00:00	xcom_push_task	write_to_xcom
<input type="checkbox"/>		test_dag	{"key1": "value1"}	2019-11-06 07:04:56.234171+00:00	2019-11-05 00:54:00+00:00	xcom_push_task	write_to_xcom
<input type="checkbox"/>		test_dag	{"key1": "value1"}	2019-11-06 07:05:07.818095+00:00	2019-11-05 01:00:00+00:00	xcom_push_task	write_to_xcom
<input type="checkbox"/>		test_dag	{"key1": "value1"}	2019-11-06 07:05:21.093908+00:00	2019-11-05 01:03:00+00:00	xcom_push_task	write_to_xcom
<input type="checkbox"/>		test_dag	{"key1": "value1"}	2019-11-06 07:05:28.601755+00:00	2019-11-05 01:06:00+00:00	xcom_push_task	write_to_xcom

# ХСОМ – ограничения в использовании

---

Для их хранения используется metastorage от airflow,  
который обычная DB

*SQLite*—Stored as BLOB type, 2GB limit

*PostgreSQL*—Stored as BYTEA type, 1 GB limit

*MySQL*—Stored as BLOB type, 64 KB limit



# Custom XCOM

---

```
from typing import Any
from airflow.models.xcom import BaseXCom

class CustomXComBackend(BaseXCom):

    @staticmethod
    def serialize_value(value: Any):
        ...

    @staticmethod
    def deserialize_value(result) -> Any:
        ...
```

# DEMO: XCOM

# Расписание

Задается параметрами в DAG

`start_date` — начиная с какого числа

выполнять dag

`end_date` — когда закончить

`schedule_interval` — с какой периодичностью

## Cron Presets

preset	meaning	cron
<code>None</code>	Don't schedule, use for exclusively "externally triggered" DAGs	
<code>@once</code>	Schedule once and only once	
<code>@hourly</code>	Run once an hour at the beginning of the hour	<code>0 * * * *</code>
<code>@daily</code>	Run once a day at midnight	<code>0 0 * * *</code>
<code>@weekly</code>	Run once a week at midnight on Sunday morning	<code>0 0 * * 0</code>
<code>@monthly</code>	Run once a month at midnight of the first day of the month	<code>0 0 1 * *</code>
<code>@quarterly</code>	Run once a quarter at midnight on the first day	<code>0 0 1 */3 *</code>
<code>@yearly</code>	Run once a year at midnight of January 1	<code>0 0 1 1 *</code>

# Как даги можно запускать?

- **по расписанию** (весь крон к нашим услугам)
- **через cli**
- **через UI** (обычно используется, когда что-то упало)
- **Через API** (когда интегрируете airflow с внешними системами) - <http://airflow.apache.org/docs/stable/rest-api-ref.html>

Cron Presets

preset	meaning	cron
<code>None</code>	Don't schedule, use for exclusively "externally triggered" DAGs	
<code>@once</code>	Schedule once and only once	
<code>@hourly</code>	Run once an hour at the beginning of the hour	<code>0 * * * *</code>
<code>@daily</code>	Run once a day at midnight	<code>0 0 * * *</code>
<code>@weekly</code>	Run once a week at midnight on Sunday morning	<code>0 0 * * 0</code>
<code>@monthly</code>	Run once a month at midnight of the first day of the month	<code>0 0 1 * *</code>
<code>@quarterly</code>	Run once a quarter at midnight on the first day	<code>0 0 1 */3 *</code>
<code>@yearly</code>	Run once a year at midnight of January 1	<code>0 0 1 1 *</code>



# Зачем запускать по расписанию?

---

Чтобы каждый день/etc обрабатывать  
новую порцию поступающих данных

27 июня

28 июня

29 июня

30 июня

```
python process_pipeline.py s3://input/22_06_2020/ s3://output/22_06_2020/
```

# Templating

```
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'},
)
```

# Templating

Variable	Description
<code>{{ ds }}</code>	the execution date as <code>YYYY-MM-DD</code>
<code>{{ ds_nodash }}</code>	the execution date as <code>YYYYMMDD</code>
<code>{{ prev_ds }}</code>	the previous execution date as <code>YYYY-MM-DD</code> if <code>{{ ds }}</code> is <code>2018-01-08</code> and <code>schedule_interval</code> is <code>@weekly</code> , <code>{{ prev_ds }}</code> will be <code>2018-01-01</code>
<code>{{ prev_ds_nodash }}</code>	the previous execution date as <code>YYYYMMDD</code> if exists, else <code>None</code>
<code>{{ next_ds }}</code>	the next execution date as <code>YYYY-MM-DD</code> if <code>{{ ds }}</code> is <code>2018-01-01</code> and <code>schedule_interval</code> is <code>@weekly</code> , <code>{{ next_ds }}</code> will be <code>2018-01-08</code>
<code>{{ next_ds_nodash }}</code>	the next execution date as <code>YYYYMMDD</code> if exists, else <code>None</code>
<code>{{ yesterday_ds }}</code>	the day before the execution date as <code>YYYY-MM-DD</code>
<code>{{ yesterday_ds_nodash }}</code>	the day before the execution date as <code>YYYYMMDD</code>
<code>{{ tomorrow_ds }}</code>	the day after the execution date as <code>YYYY-MM-DD</code>
<code>{{ tomorrow_ds_nodash }}</code>	the day after the execution date as <code>YYYYMMDD</code>
<code>{{ ts }}</code>	same as <code>execution_date.isoformat()</code> . Example: <code>2018-01-01T00:00:00+00:00</code>
<code>{{ ts_nodash }}</code>	same as <code>ts</code> without <code>-</code> , <code>:</code> and TimeZone info. Example: <code>20180101T000000</code>

```
background-color: #F5F5F5;
text-shadow: 0px -1px 0px #EAEAEA;
filter: dropshadow(color:#777;
color:#777;

}
header #main-navigation ul li span:hover,
box-shadow: 0px 0px 1px #EAEAEA;
-webkit-box-shadow: 0px 0px 2px #EAEAEA;
moz-box-shadow: 0px 0px 1px #EAEAEA;
background-color:#F9F9F9;
```

DEMO: Используем jinja  
template для работы с  
датами

# Sensors

Sensor - это оператор, который “ждет”(пуллит) наступления какого-то события

## Примеры:

- Появление файла на FTP
- Появление файла на HDFS
- 200 от как http запроса с какому-нибудь сервису
- Выполнение SQL-запроса

## `airflow.sensors`

### Submodules

- `airflow.sensors.base_sensor_operator`
- `airflow.sensors.date_time_sensor`
- `airflow.sensors.external_task_sensor`
- `airflow.sensors.hdfs_sensor`
- `airflow.sensors.hive_partition_sensor`
- `airflow.sensors.http_sensor`
- `airflow.sensors.metastore_partition_sensor`
- `airflow.sensors.named_hive_partition_sensor`
- `airflow.sensors.s3_key_sensor`
- `airflow.sensors.s3_prefix_sensor`
- `airflow.sensors.sql_sensor`
- `airflow.sensors.time_delta_sensor`
- `airflow.sensors.time_sensor`
- `airflow.sensors.web_hdfs_sensor`

DEMO: Используем  
sensor

# Variables

На уровне сервиса есть возможность задавать “глобальные” переменные (Admin -> Variables)

The best way of using variables is via a Jinja template, which will delay reading the value until the task execution.

```
 {{ var.value.<variable_name> }}
```

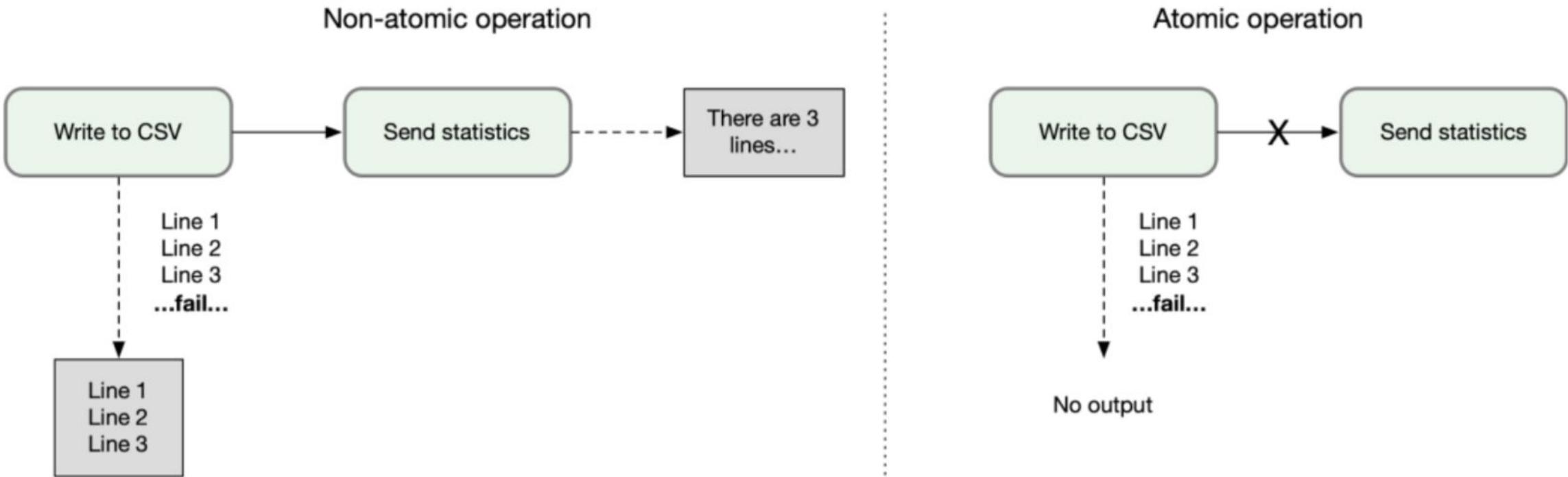
or if you need to deserialize a json object from the variable :

```
 {{ var.json.<variable_name> }}
```

# Best Practice по написанию

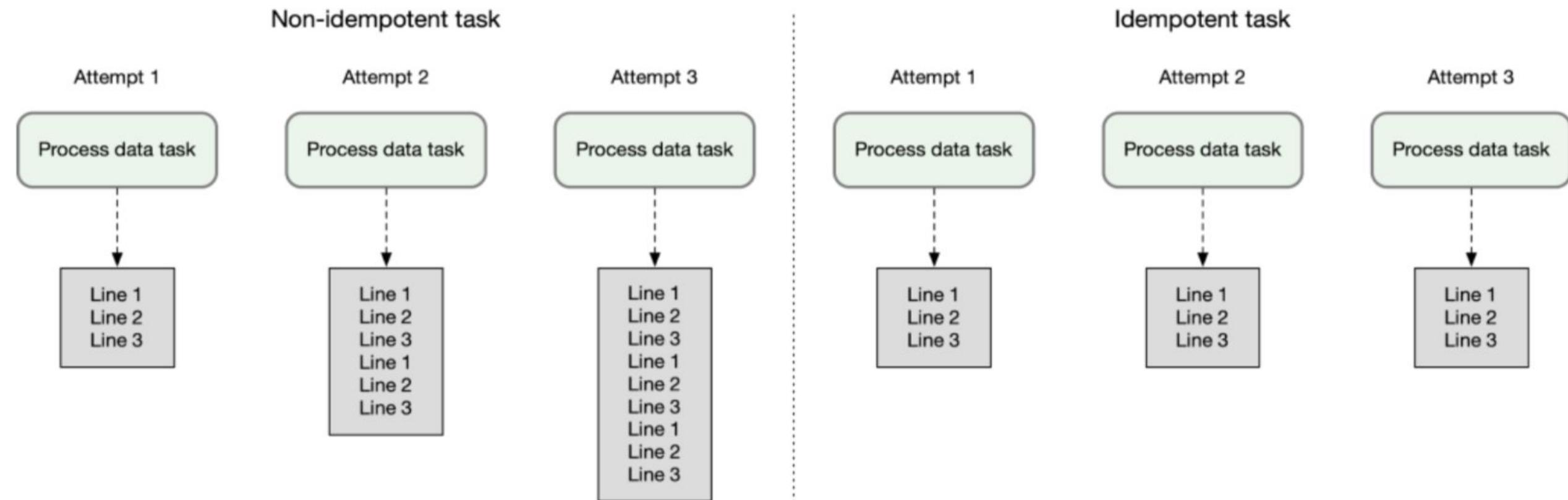
# Атомарность

Таски должны быть атомарными и независимыми друг от друга



# Идемпотентность

Задача обладает свойство идемпотентности, если ее вызов несколько раз с одними и теми же аргументами не приводит к дополнительному эффекту





# Детерминированность

---

Задача возвращает один и тот же выход, на один и тот же вход

# Тестирование dag

- 1) Тестируем все сущности, которые написали самостоятельно (Operators, sensors, hooks, etc)
- 2) Тестируем корректность дага
- 3) Прогоняем даг целиком на подмножестве реальных данных

```
from airflow.models import DagBag
import unittest

class TestHelloWorldDAG(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.dagbag = DagBag()

    def test_dag_loaded(self):
        dag = self.dagbag.get_dag(dag_id='hello_world')
        assert self.dagbag.import_errors == {}
        assert dag is not None
        assert len(dag.tasks) == 1
```

# Не делать вычислений в коде dag

```
wait = PythonSensor(  
    task_id="wait_for_file",  
    python_callable=_wait_for_file,  
    timeout=6000,  
    poke_interval=10,  
    retries=100,  
    mode="poke",  
)  
  
do_train_neural_network()  
  
t3 = BashOperator(  
    task_id="touch_file_3",  
    depends_on_past=True,  
    bash_command="touch /opt/airflow/data/2.txt",  
)
```



# Соответствие code-style

---

- Все что проходили ранее про pep8, etc
- Объявление дагов в одном стиле(context managers or not)
- Задание зависимостей в одном стиле

# Использовать фабрики для генерации dag

---

```
def generate_set_of_tasks(dataset_name):
    download_task = DummyOperator(task_id=f"download_{dataset_name}")
    preprocess_task = DummyOperator(task_id=f"preprocess_{dataset_name}")
    download_task >> preprocess_task
```

## DEMO: генерация тасок-дагов

# Использовать TaskGroup для группировки связанных тасок

---

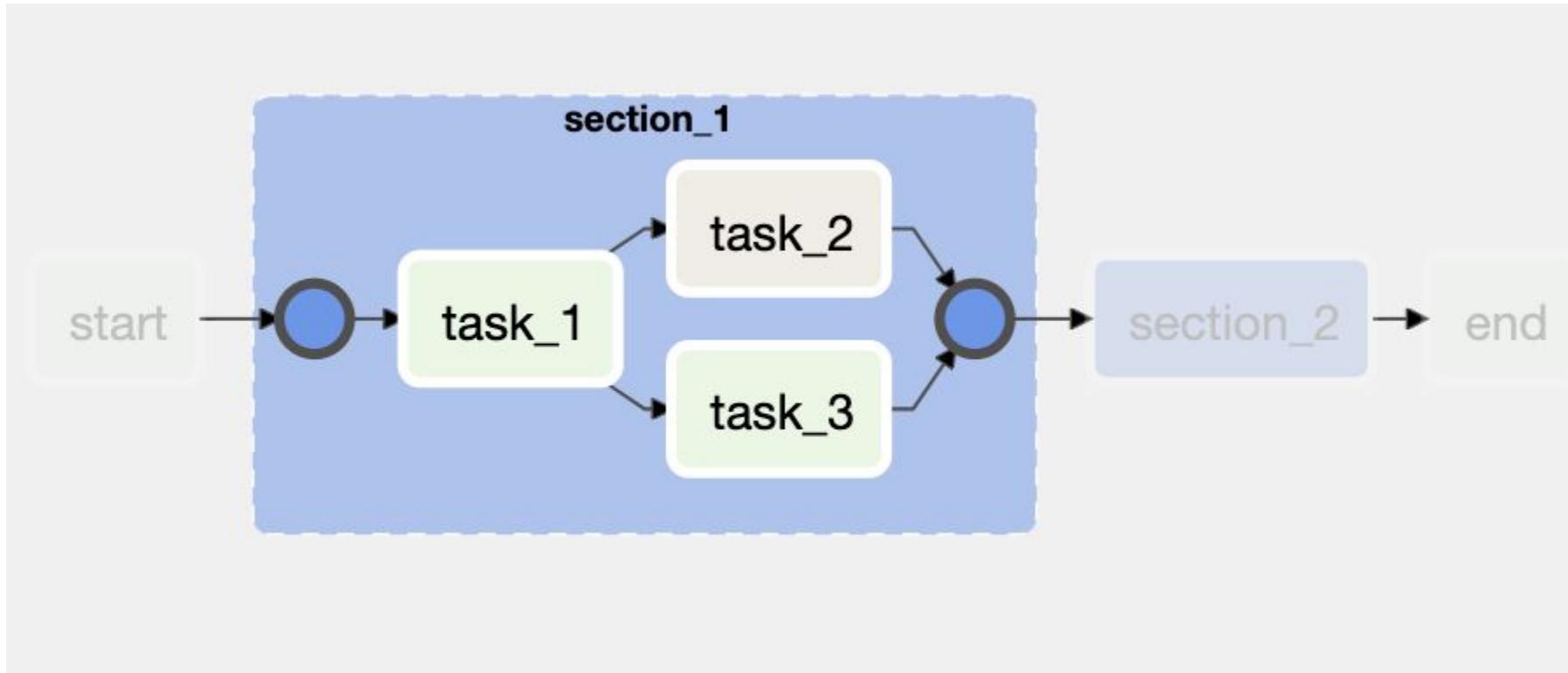
```
# [START howto_task_group]
with DAG(dag_id="example_task_group", start_date=days_ago(2), tags=["example"]) as dag:
    start = DummyOperator(task_id="start")

    # [START howto_task_group_section_1]
    with TaskGroup("section_1", tooltip="Tasks for section_1") as section_1:
        task_1 = DummyOperator(task_id="task_1")
        task_2 = BashOperator(task_id="task_2", bash_command='echo 1')
        task_3 = DummyOperator(task_id="task_3")

        task_1 >> [task_2, task_3]
    # [END howto_task_group_section_1]
```

# Использовать TaskGroup для группировки связанных тасок

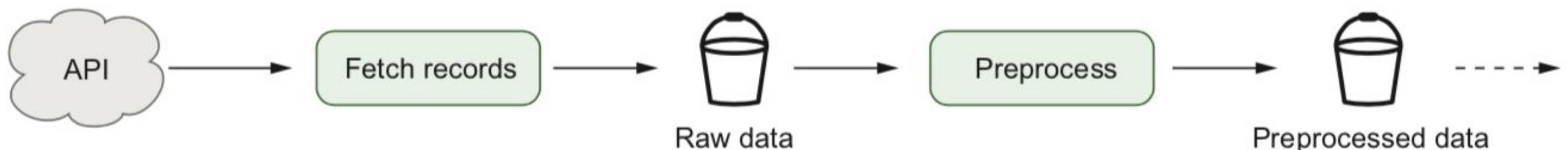
---



**DEMO:** тыкаем в  
TaskGroup

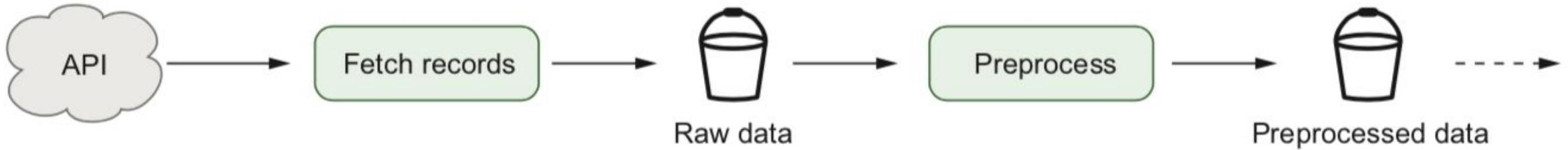
# Сохранять промежуточные данные

---



Уходим в мир множества  
воркеров

# Сохранять промежуточные данные



Не сохранять их на локальной файловой системе(а  
сохранять на s3, hdfs, etc)

# Executors

Для исполнения своих тасок Airflow поддерживает несколько “бэкендов”

- Sequential Executor
- Debug Executor
- Local Executor
- Dask Executor
- Celery Executor
- Kubernetes Executor
- Scaling Out with Mesos (community contributed)

# Зависимости

## PythonVirtualenvOperator

```
def callable_virtualenv():
    """
    Example function that will be performed in a virtual environment.

    Importing at the module level ensures that it will not attempt to import the
    library before it is installed.
    """
    from time import sleep

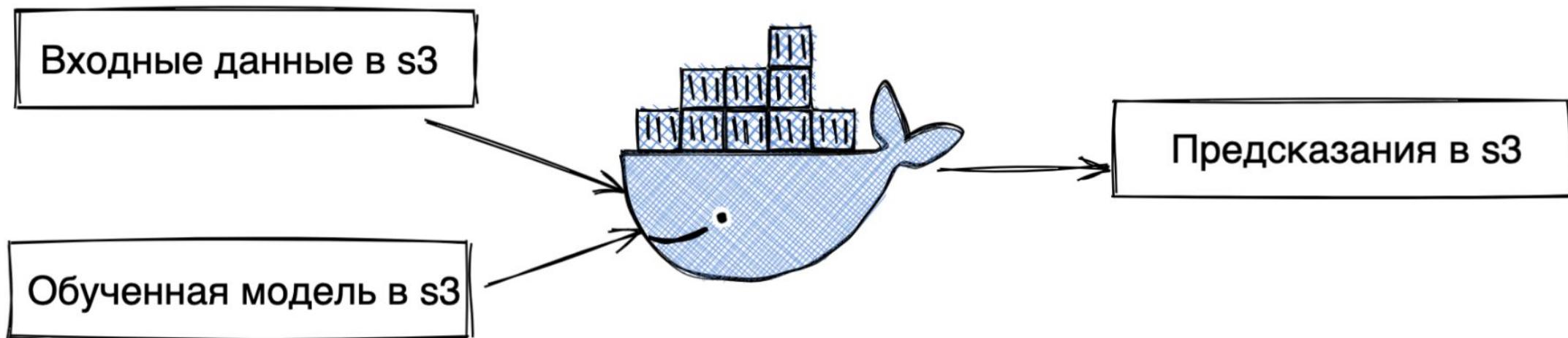
    from colorama import Back, Fore, Style

    print(Fore.RED + 'some red text')
    print(Back.GREEN + 'and with a green background')
    print(Style.DIM + 'and in dim text')
    print(Style.RESET_ALL)
    for _ in range(10):
        print(Style.DIM + 'Please wait...', flush=True)
        sleep(10)
    print('Finished')

virtualenv_task = PythonVirtualenvOperator(
    task_id="virtualenv_python",
    python_callable=callable_virtualenv,
    requirements=["colorama==0.4.0"],
    system_site_packages=False,
)
```

# Batch инференс модели в docker

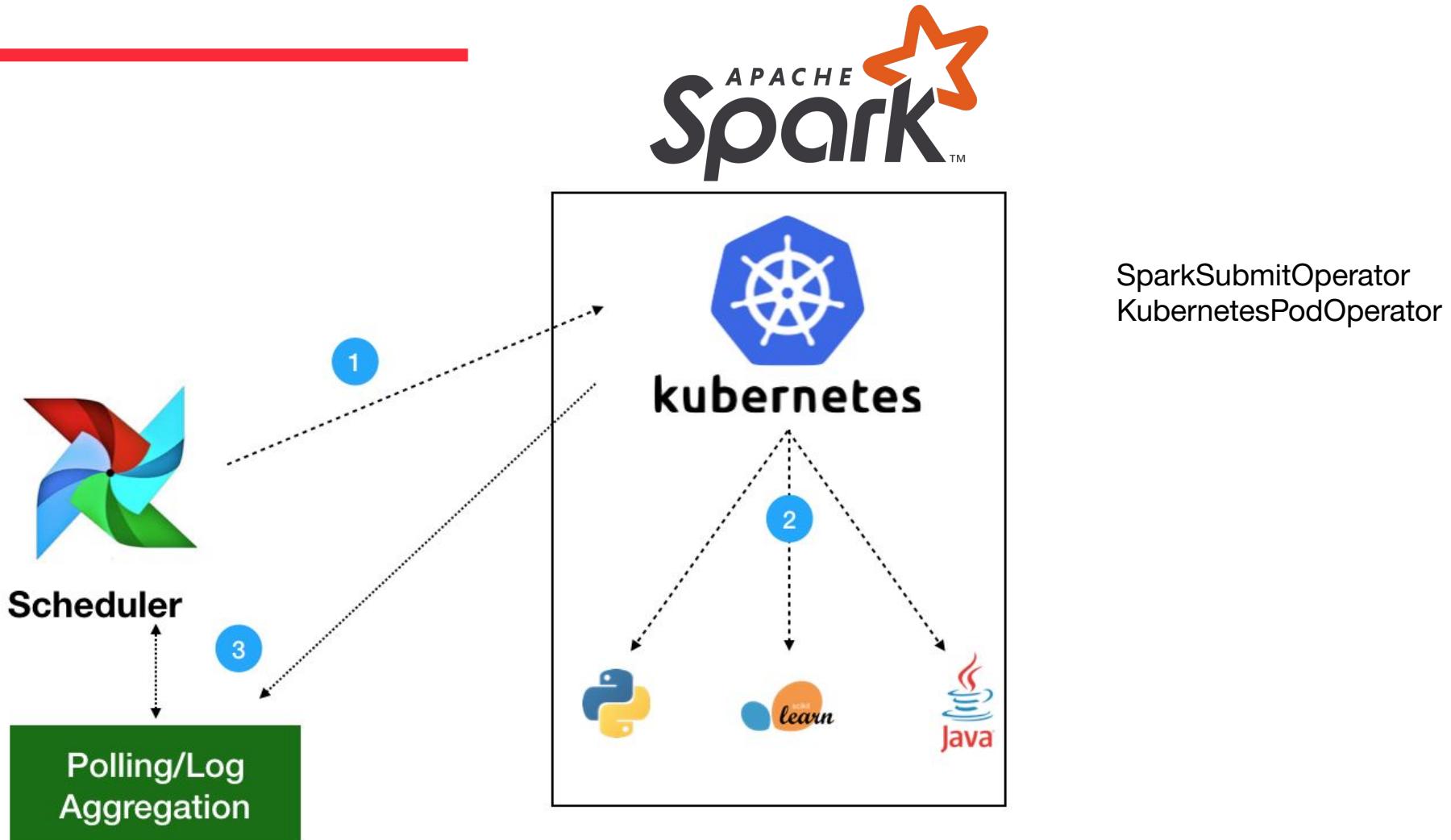
---



DockerOperator

DEMO: пара демок с  
DockerOperator

# Сбрасывать тяжелые вычисления в другие системы



# Pools

---

Позволяют ограничивать количество задач одного типа. Задается в UI.

Полезно для:

- ограничения кол-ва одновременно запущенных задач
- использования ресурса(знаем, что некоторые таски отправляют запросы на определенный сервис и делаем, чтобы их было не более 10 одновременно)
- позволяет задавать приоритеты

```
PythonOperator(  
    task_id="my_task",  
    ...  
    pool="my_resource_pool"  
)
```

## Альтернативные оркестраторы

# Oozie

Был стандартным оркестратором для hadoop мира. Даги задаются в XML

```
<workflow-app xmlns = "uri:oozie:workflow:0.4" name = "test-workflow">  
    <start to = "run_hive_script" />  
    <action name = "run_hive_script">  
        <hive xmlns = "uri:oozie:hive-action:0.4">  
            <job-tracker>xyz.com:8088</job-tracker>  
            <name-node>hdfs://rootname</name-node>  
            <script>hdfs_path_of_script/external.hive</script>  
        </hive>  
        <ok to = "end" />  
        <error to = "kill_job" />  
    </action>  
    <kill name = "kill_job">  
        <message>Job failed</message>  
    </kill>  
    <end name = "end" />  
</workflow-app>
```

# Luigi

Довольно старый оркестратор, очень похожий на airflow  
**не содержит функционал шедулинга**

```
class MyTask1(luigi.Task):
    x = luigi.IntParameter()
    y = luigi.IntParameter(default=0)

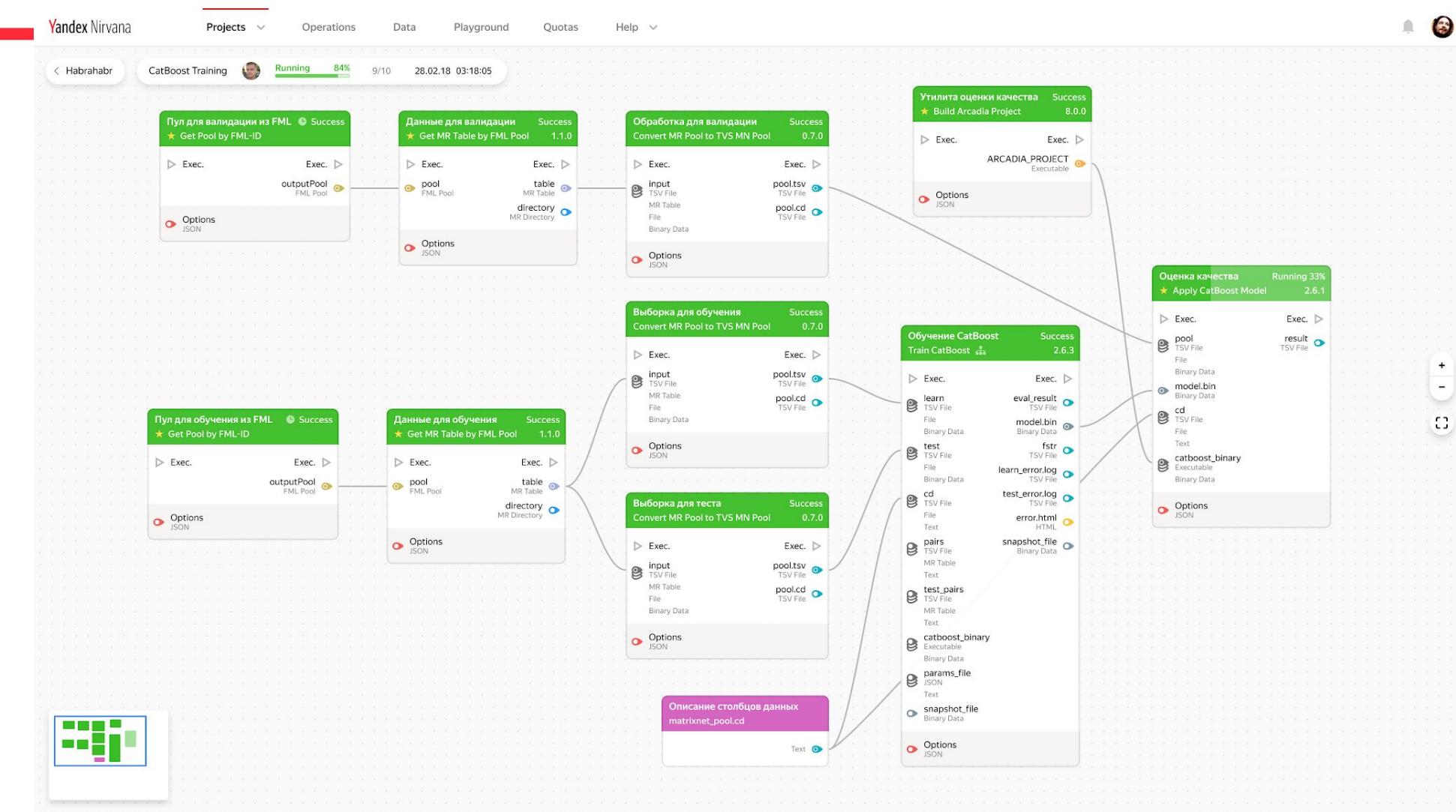
    def run(self):
        print(self.x + self.y)

class MyTask2(luigi.Task):
    x = luigi.IntParameter()
    y = luigi.IntParameter(default=1)
    z = luigi.IntParameter(default=2)

    def run(self):
        print(self.x * self.y * self.z)

if __name__ == '__main__':
    luigi.build([MyTask1(x=10), MyTask2(x=15, z=3)])
```

# Яндекс Нирвана



# Dagster

Hello dagster 🙌

`hello_dagster.py`

```
from dagster import execute_pipeline, pipeline, solid

@solid
def get_name(_):
    return 'dagster'

@solid
def hello(context, name: str):
    context.log.info('Hello, {name}!'.format(name=name))

@pipeline
def hello_pipeline():
    hello(get_name())
```

# Dagster

The screenshot shows the Dagster playground interface for an `airline_demo_ingest_pipeline`. The left sidebar includes navigation links for Pipelines, Solids, Runs, Playground, Schedules, and a version indicator (0.76). The main area displays the pipeline configuration code:

```
10 target_folder: /tmp/dagster/airline_data/file_cache
11 pyspark:
12 config:
13 spark_conf:
14 spark:
15 jars:
16 packages: "com.databricks:spark-avro_2.11:3.0.0,com.databricks:spark-redshift_2.11:config:1.0.0,com.databricks:spark-sql-redshift_2.11:3.0.0"
17
18 solids:
19 process_q2_coupon_data:
20 inputs:
21 s3_coordinate:
22 bucket: dagster-airline-demo-source-data
23 key: test/Origin and Destination Survey DB1BCoupon 2018 2.zip
```

A code completion tooltip is visible on the right side of the configuration text.

Below the configuration, there are tabs for RUNTIME (execution, loggers, storage, db\_info, file\_cache, pyspark, s3) and SOLIDS (april\_on\_time\_s3\_to\_df, download\_q2\_sfo\_weather, ingest\_q2\_sfo\_weather, join\_q2\_data, june\_on\_time\_s3\_to\_df, load\_q2\_on\_time\_data, load\_q2\_sfo\_weather, master\_cord\_s3\_to\_df, may\_on\_time\_s3\_to\_df, process\_q2\_coupon\_data, process\_q2\_market\_data, process\_q2\_ticket\_data, process\_sfo\_weather\_data).

At the bottom right is a blue button labeled **▶ Start Execution**.



# Dagster

---

- **Strengths/Weaknesses:** I think the answer here really depends on the use case; if you're already a company that's heavily invested in Airflow, we see our integration as a way to gain access to the strengths of Dagster—a sane local development story, great tools (Dagit), testability, our GraphQL API, etc.—while continuing to use all of your existing infrastructure for existing DAGs and for ops. For teams that haven't yet adopted a workflow system, Dagster + Dask (or even just execution with vanilla Dagster) can likely work well without the heavyweight introduction of Airflow.

<https://github.com/dagster-io/dagster/issues/1593>

# AIRFLOW TASKAPI

```
with DAG(
    'send_server_ip', default_args=default_args, schedule_interval=None
) as dag:

    # Using default connection as it's set to httpbin.org by default
    get_ip = SimpleHttpOperator(
        task_id='get_ip', endpoint='get', method='GET', xcom_push=True
    )

    @dag.task(multiple_outputs=True)
    def prepare_email(raw_json: str) -> Dict[str, str]:
        external_ip = json.loads(raw_json)['origin']
        return {
            'subject':f'Server connected from {external_ip}',
            'body': f'Seems like today your server executing Airflow is connected from the external IP {external_ip}<br>'
        }

    email_info = prepare_email(get_ip.output)

    send_email = EmailOperator(
        task_id='send_email',
        to='example@example.com',
        subject=email_info['subject'],
        html_content=email_info['body']
    )
```



# Содержание занятия

---

- Зачем нужны и что такое оркестраторы данных +
- Что такое Airflow +
- Разворачиваем Airflow +
- Пишем даги на airflow +
- Best practice написание дагов на airflow +

академия  
больших  
данных



mail.ru  
group

## Data Pipelines

Михаил Марюфич, MLE

