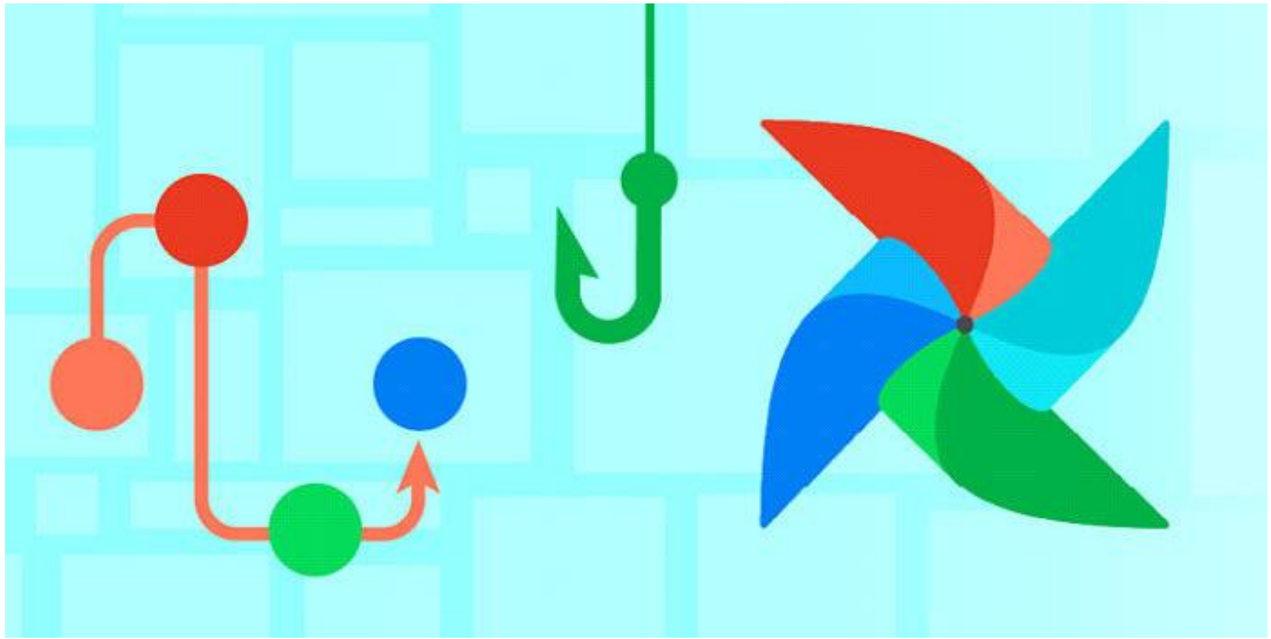
Apache Airflow

Назначение Apache Airflow

Apache Airflow – платформа для **создания, оркестрации, управления** расписанием и **мониторингом** Workflow-процессов загрузок данных.

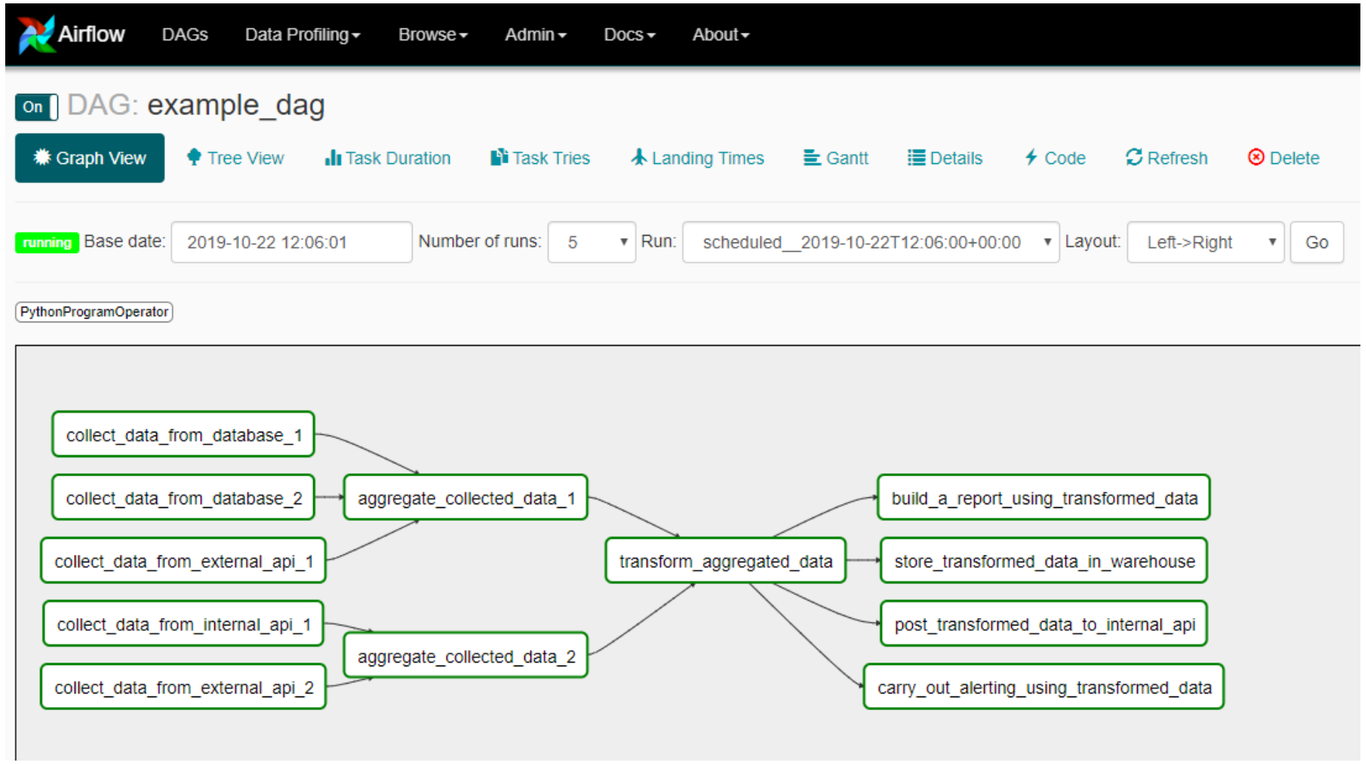
Основные сущности рабочего процесса на Apache Airflow:

- **Направленные ациклические графы (DAG)**
- **Планировщик (Scheduler)**
- **Операторы (Operators)**
- **Задачи (Tasks)**



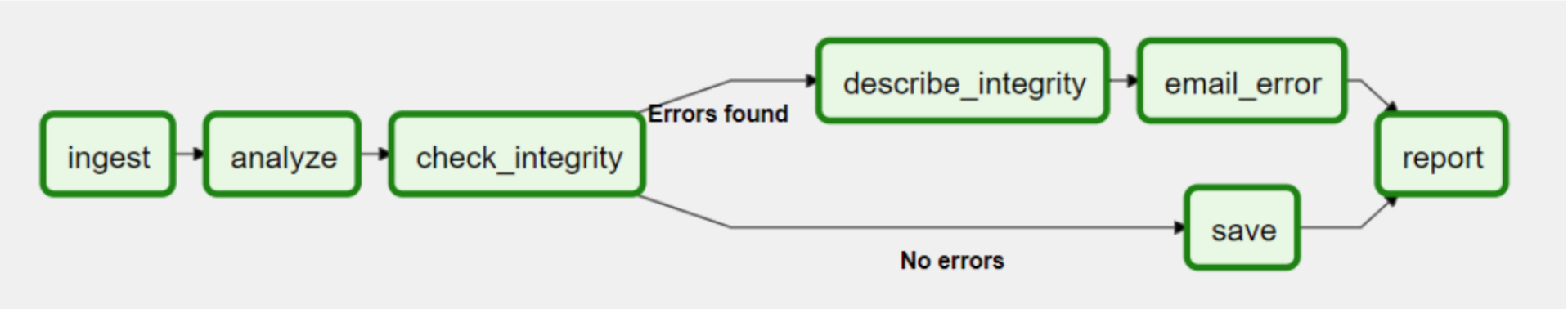
Основные особенности

- Airflow используется для создания рабочих процессов в виде направленных ациклических графов (DAG) задач.
- Планировщик Airflow выполняет задачи в виде множества рабочих процессов, следуя указанным зависимостям.
- Пользовательский интерфейс позволяет легко визуализировать конвейеры, отслеживать ход выполнения и при необходимости устранять неполадки.
- Когда рабочие процессы определяются как код, они становятся более удобными в сопровождении, версиях, тестировании и совместной работе.



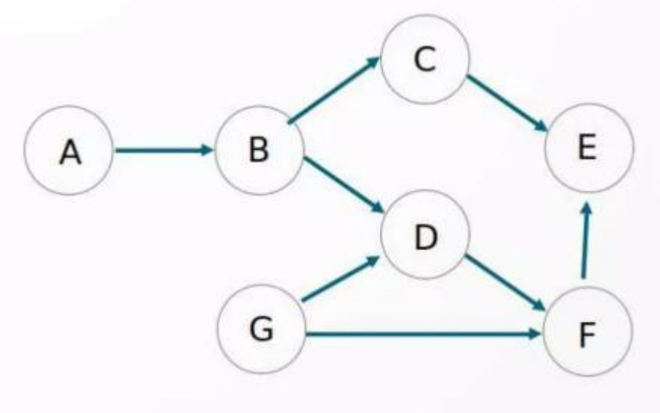
Основная идея рабочих процессов в управлении данными

Airflow — это платформа, которая позволяет создавать и запускать рабочие процессы. Рабочий процесс представлен как **DAG** (ориентированный ациклический граф) и содержит отдельные части работы, называемые задачами, организованные с учетом зависимостей и потоков данных.



DAG определяет зависимости между задачами и порядок их выполнения и выполнения повторных попыток

Сами задачи описывают, что нужно делать, будь то получение данных, запуск анализа, запуск других систем, скриптов ML, проверки качества данных, e-mail рассылка и т. д.

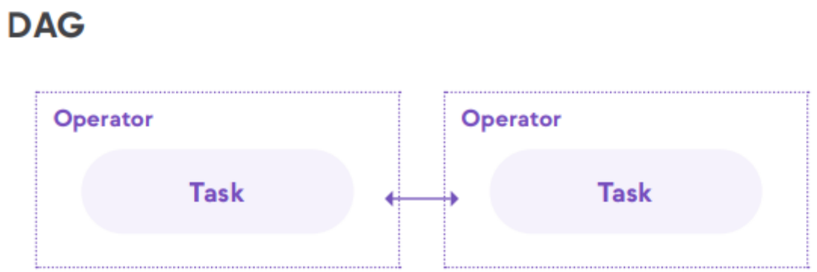


DAG

•**Зависимости в DAG** гарантируют, что ваши задачи обработки данных каждый раз выполняются в одном и том же порядке, это создает надежный процесс для вашей повседневной инфраструктуры обработки данных.

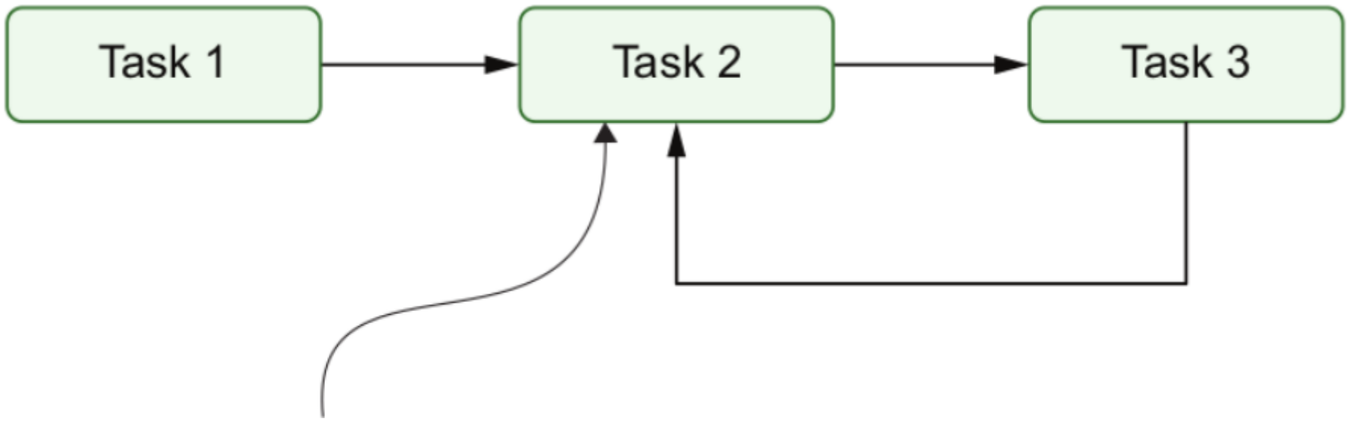
•**Графический компонент DAG** позволяет визуализировать зависимости в пользовательском интерфейсе Airflow.

•**Поскольку каждый путь в DAG является линейным**, легко разрабатывать и тестировать конвейеры данных на соответствие ожидаемым результатами.



А почему не циклические графы?

Ориентированный циклический граф привел бы к дэдлоку



Видим, что таска 2 элементарно не может выполниться, т.к. зависит от результата таски 3, а та, в свою очередь, зависит от таска 2

Поток управления

- DAGs предназначены для многократного запуска, и несколько их запусков могут выполняться параллельно.
- DAGs параметризуются, всегда включая интервал, для которого они «выполняются» (интервал данных), но также и с другими необязательными параметрами.
- Задачи имеют зависимости друг от друга. Вы увидите это в DAG либо с помощью операторов >> и <<:

```
first_task >> [second_task, third_task]
fourth_task << third_task
```

- Или с помощью методов set_upstream и set_downstream:

```
first_task.set_downstream([second_task, third_task])
fourth_task.set_upstream(third_task)
```

- Эти зависимости составляют «ребра» графа и то, как Airflow определяет, в каком порядке выполнять задачи. По умолчанию задача будет ждать, пока все ее вышестоящие задачи не будут выполнены успешно, прежде чем она запустится, но это может быть настроено также с помощью функций

Передача данных между задачами

XComs («кросс-коммуникации»)
система, в которой вы можете задавать задачи для передачи и извлечения небольших данных

Загрузка и скачивание больших файлов из общей системы хранения
(либо запущенной вами, либо части общедоступного облака)

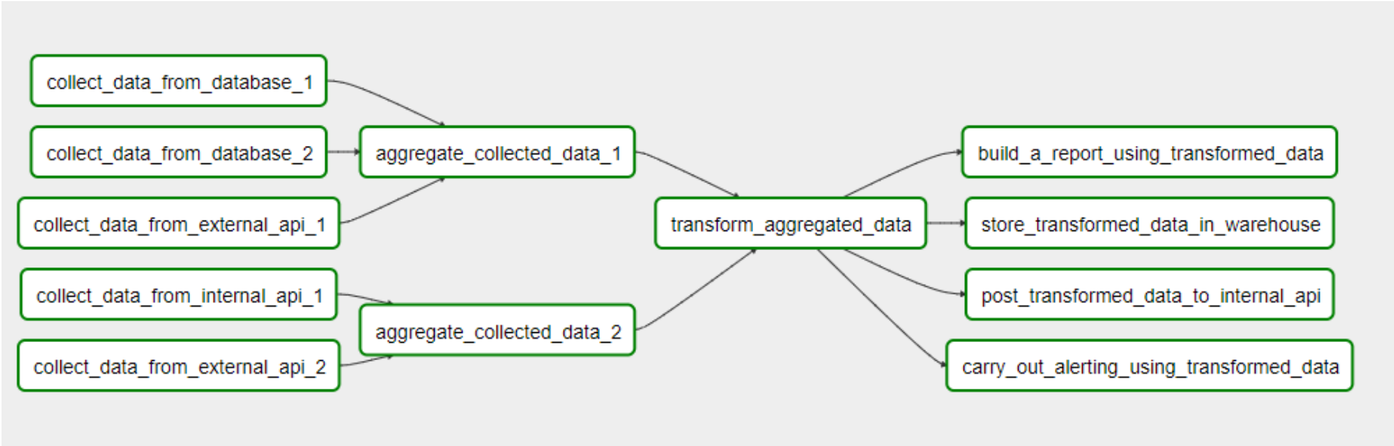
Airflow отправляет задачи для выполнения на рабочих процессах по мере освобождения места (пула ресурсов), поэтому нет гарантии, что все задачи в вашем DAGe будут выполняться на одном Worker или на одной и той же машине.

Передача данных между задачами

По мере того, как вы создаете свои DAG, они становятся очень сложными и большими, поэтому Airflow предоставляет несколько механизмов для того, чтобы сделать это более устойчивым и удобным для восприятия:

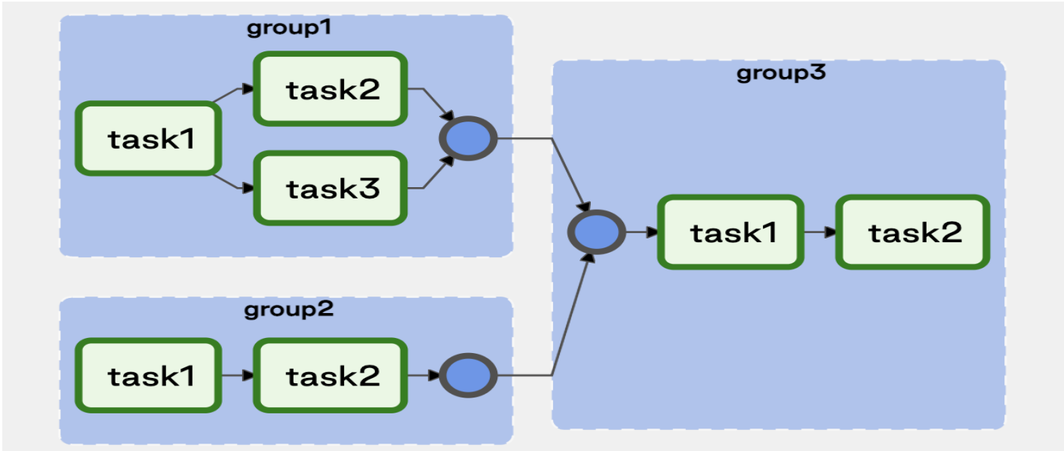
SubDAG

они позволяют вам создавать «повторно используемые» DAGs, которые вы можете встраивать в другие



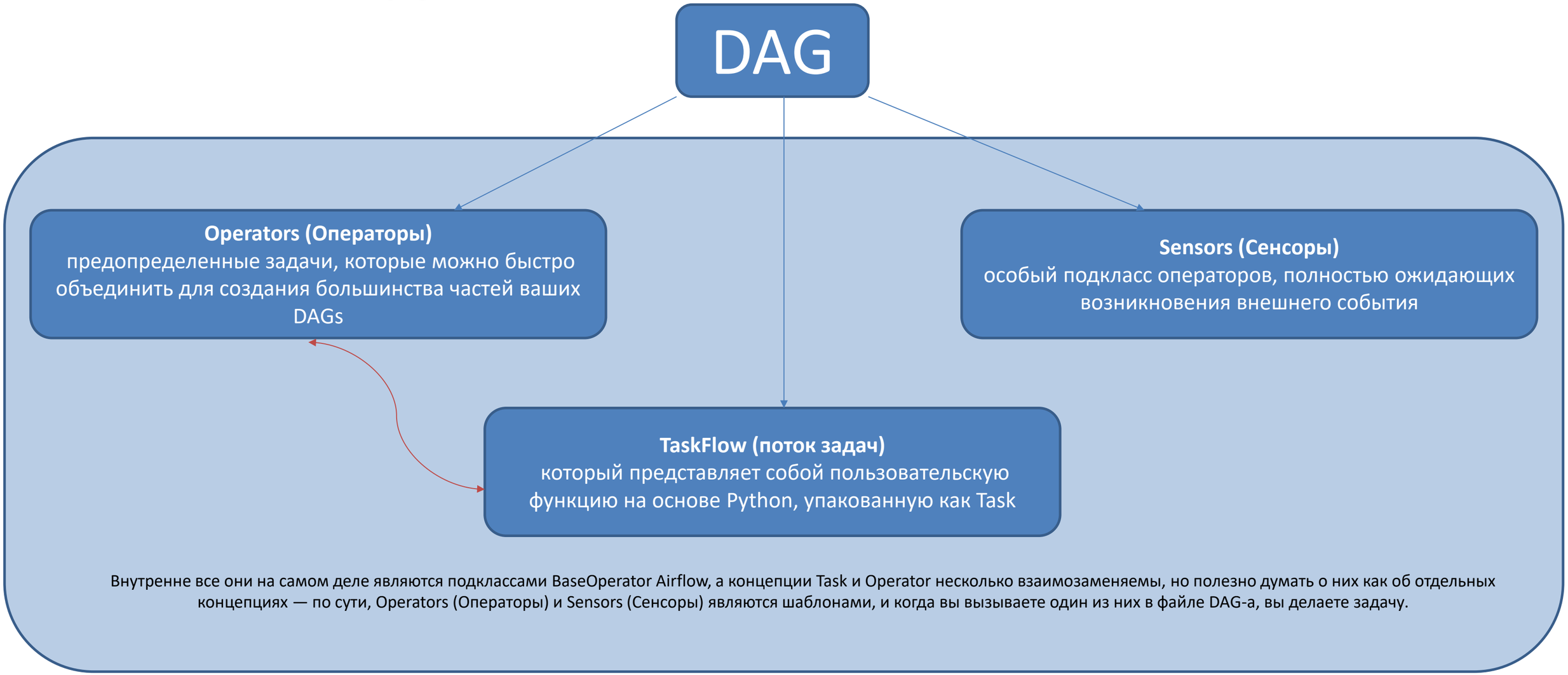
TaskGroups

позволяют визуально группировать задачи в пользовательский интерфейс



Существуют также функции, позволяющие легко предварительно настроить доступ к центральному ресурсу, такому как хранилище данных, в форме Connections & Hooks, а также для ограничения параллелизма через Pools (пулы)

Рабочие нагрузки

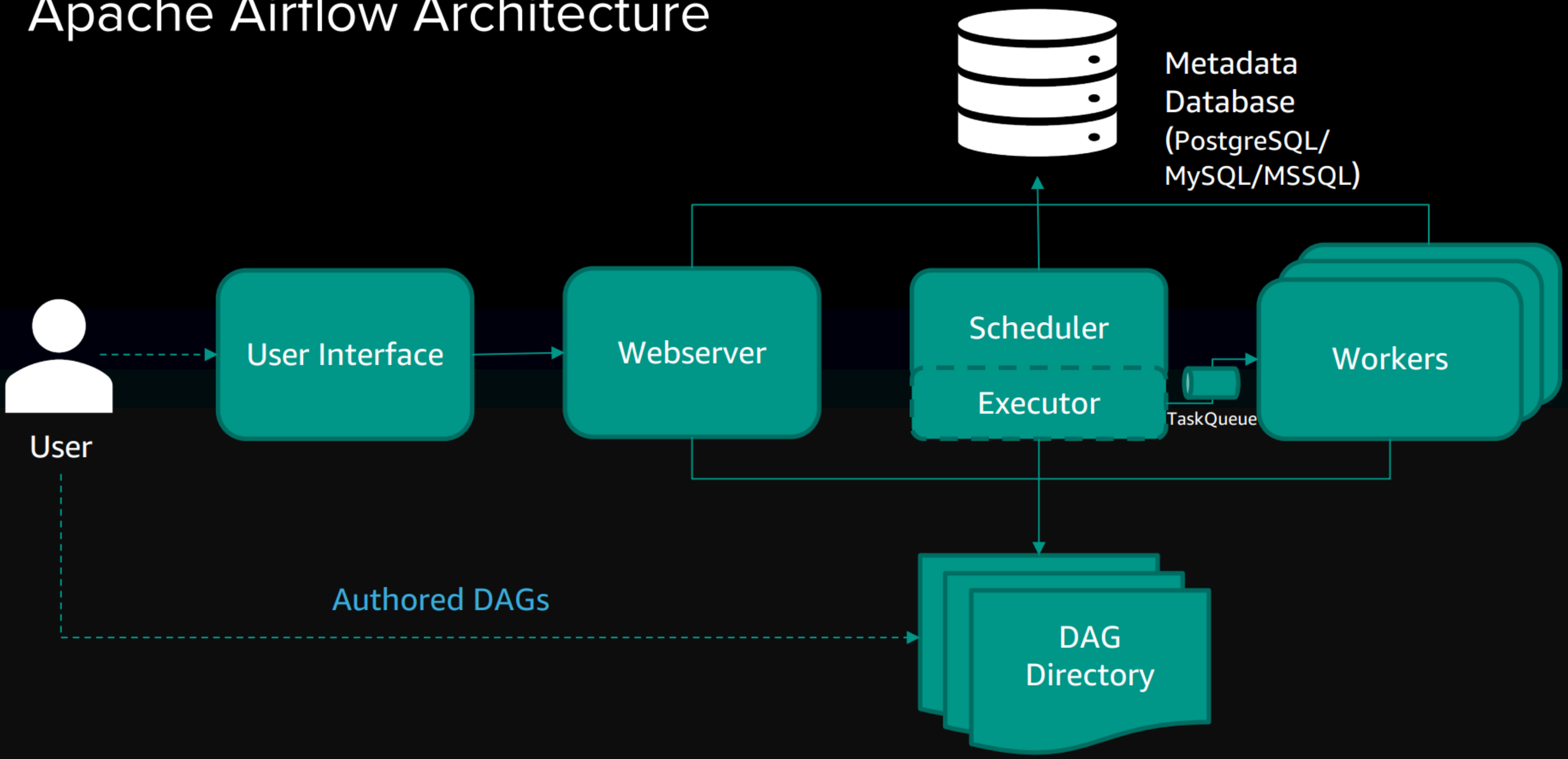


Внутренне все они на самом деле являются подклассами BaseOperator Airflow, а концепции Task и Operator несколько взаимозаменяемы, но полезно думать о них как об отдельных концепциях — по сути, Operators (Операторы) и Sensors (Сенсоры) являются шаблонами, и когда вы вызываете один из них в файле DAG-а, вы делаете задачу.

Операторы

- BashOperator
- PythonOperator
- PostgresOperator
- DockerOperator
- SparkSubmitOperator
- etc (>100)

Apache Airflow Architecture



Типы Executors

LocalExecutor

Локальное исполнение
DAG`ов внутри scheduler

CeleryExecutor

Распределённое исполнение с
помощью механизмов celery.
Легко масштабируется

KubernetesExecutor

Интеграция с Kubernetes
позволяет использовать
для каждой Task`и отдельный POD

LocalKubernetesExecutor
CeleryKubernetesExecutor

Гибридные экзекьюторы, совмещающие
в себе разные технологии

DAGs

All 26 | Active 10 | Paused 16

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 * * *	2020-10-26, 21:08:11	6		
example_branch_dop_operator_v3 example	airflow		*/1 * * * *				
example_branch_operator example example2	airflow	1	@daily	2020-10-23, 14:09:17	11		
example_complex example example2 example3	airflow	1 1	None	2020-10-26, 21:08:04	37 37		
example_external_task_marker_child	airflow	1	None	2020-10-26, 21:07:33	2		
example_external_task_marker_parent	airflow	1	None	2020-10-26, 21:08:34	1		
example_kubernetes_executor example example2	airflow		None				
example_kubernetes_executor_config example3	airflow	1	None	2020-10-26, 21:07:40	5		
example_nested_branch_dag example	airflow	1	@daily	2020-10-26, 21:07:37	9		
example_passing_params_via_test_command example	airflow		*/1 * * * *				

01.07.2025 22:11:07 | 25 | All Run Types | All Run States | Clear Filters | Auto-refresh

Press shift + / for Shortcuts

deferred | failed | queued | removed | restarting | running | scheduled | skipped | success | up_for_reschedule | up_for_retry | upstream_failed | no_status

Duration
Jun 29, 13:00
03:02:16
01:31:08
00:00:00

stage_sources
dbt_run
source_access_groups
source_access_groups_postgres
source_accounts
source_accounts_postgres
source_calculator_intervalpayments
source_calculator_intervalpayments_postgres
source_calculator_orderprices_fixed
source_calculator_orderprices_fixed_postgres
source_calculator_orders
source_calculator_orders_postgres
source_calculator_resources
source_calculator_resources_orderstatus
source_calculator_resources_postgres
source_data_centers
source_data_centers_postgres
source_environment_prefixes
source_environment_prefixes_postgres
source_events
source_events_lastitem
source_events_lastitem_postgres
source_flavors
source_flavors_postgres
source_folder_account_link
source_folder_account_link_postgres
source_folders
source_folders_postgres
source_information_systems
source_information_systems_postgres
source_order_actions
source_order_actions_postgres
source_orders
source_orders_postgres
source_organizations
source_organizations_postgres
source_portal_options
source_portal_options_postgres
source_product_graphs
source_product_graphs_postgres
source_product_graphversions
source_product_graphversions_postgres
source_product_versions

DAG dbt_workflow / Run 2025-07-01, 07:00:00 MSK

Clear | Mark state as...

Details | Graph | Gantt | Code

Layout: Left -> Right

React Flow

Apache Airflow Code Example

```python
from airflow.contrib.sensors.file_sensor import FileSensor
from airflow.operators.dummy_operator import DummyOperator

import datetime
from datetime import date, timedelta
import airflow

default_args = {
    "depends_on_past": False,
    "start_date": airflow.utils.dates.days_ago(1),
    "retries": 1,
    "retry_delay": datetime.timedelta(hours=5),
}
today = datetime.datetime.today()
yesterday = date.today() - timedelta(days=1)

with airflow.DAG("file_sensor_example", default_args=default_args,
                 schedule_interval= "*/5 * * * *") as dag:

    start_task = DummyOperator(task_id="start")
    stop_task = DummyOperator(task_id="stop")
    sensor_task = FileSensor(task_id="file_sensor_task",
                             poke_interval=30,
                             fs_conn_id= "<path>",
                            filepath= "<file or directory name>")

start_task >> sensor_task >> stop_task
```

Imported Libraries

Default Arguments/dictionary of default parameters

Unique identifier + schedule interval

Task definitions

Dependency

Demo

https://tinyurl.com/itmo-airflow