Airflow

План

01. Введение

02. **Core Concepts**

03. Архитектура и компоненты

04. Киллерфичи**!**

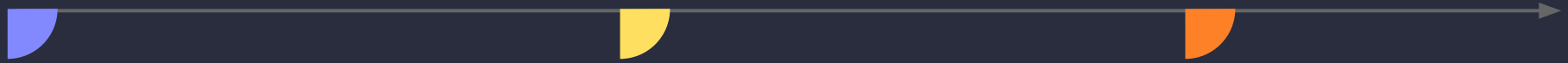
05. **Airflow 3**

Введение

ТЫК

Apache Airflow is an open-source platform for developing, scheduling, and monitoring batch-oriented workflows.

Чуть-чуть истории



Airbnb | 2014

Open Source | 2015

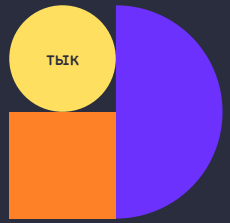
Apache TLP | 2019

```
30M+ скачиваний в месяц (30x с 2020)
80K+ организаций (было 25K в 2020)
```

Основные преимущества

Open Source & Community

Огромное коммьюнити, которое развивает и поддерживает инструмент. Открытый код и подробная документация



Масштабируемость и расширяемость

Гибкая модульная архитектура, 300+ готовых операторов и поддержка кастомных решений для любых масштабов

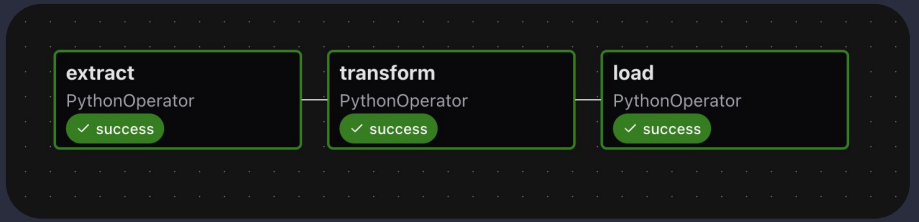
Code-first подход

Workflow описываются в Python-коде (DAG-файлы), что даёт полный контроль над логикой

Core Concepts

DAG

Directed acyclic graph



DAG – граф зависимостей, а не исполнитель кода. Включает в себя:

- Schedule
- Tasks
- Task Dependencies
- Callback
- Additional Parameters

DAG Run – экземпляр дага во времени

- Типы: scheduled, manual, backfill
- Хранит состояние всех тасок в рамках этого запуска
- Имеет уникальный run_id и метку времени (execution_date / logical_date)

Data Interval – временной интервал, за который DAG обрабатывает данные (ex. @daily, @monthly, etc.)

Следует различать

- *logical date (**раньше** execution date)*: указывает, за какой период отвечает запуск
- *data interval start / end*: начало / конец интервала, за который обрабатываются данные
- *start / end date*: начало / конец фактического запуска дага

Чтобы data interval не совпадал с logical date:
CREATE_CRON_DATA_INTERVALS=True (начиная с 3 версии)

Tasks & Dependencies

Минимальная единицы и их взаимосвязь

Тип задачи	Описание	Когда использовать?	Примеры
Операторы	Предопределенные действия	Стандартные операции	`BashOperator`, `PythonOperator`, `PostgresOperator`
Сенсоры	Ожидание внешнего события	Проверка готовности данных	`FileSensor`, `HttpSensor`
Декоратор @task	Кастомная логика на Python	Гибкие преобразования	Любая Python-функция

Зависимости определяют порядок выполнения задач в DAG

- Задача запускается только после выполнения всех своих родителей
- По умолчанию используется правило all_success

Можно задавать множественные зависимости

<parallel_tasks.py> <union_chains.py> <multiple_dependence.py>

```python
#1: через операторы >> и <<
extract_task >> transform_task >> load_task

#2: через set_upstream/set_downstream
extract_task.set_downstream(transform_task)

#3: в TaskFlow API
load(transform(extract()))
```

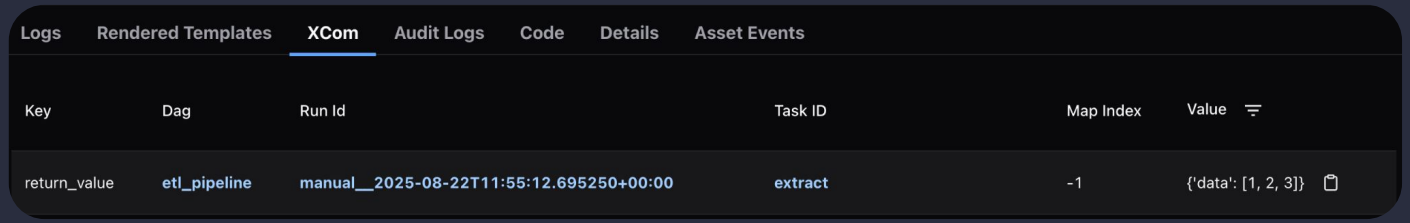
XComs

Cross-Communication

XComs – ключевой механизм Airflow для передачи данных между задачами

- Данные сохраняются в таблице xcom базы данных Airflow
- Ограничение ~48KB
- Данные сериализуются в json

dag_run_id	task_id	map_index	key	dag_id	run_id	value	timestamp
3	task1	-1	return_value	union_chains	manual__2025-07-01T15:45:38.042292+00:00	"task1"	2025-07-01 15:45:49.578752+00
3	task3	-1	return_value	union_chains	manual__2025-07-01T15:45:38.042292+00:00	"task3"	2025-07-01 15:45:49.580835+00
3	task2	-1	return_value	union_chains	manual__2025-07-01T15:45:38.042292+00:00	"task2"	2025-07-01 15:46:00.868167+00
3	task4	-1	return_value	union_chains	manual__2025-07-01T15:45:38.042292+00:00	"task4"	2025-07-01 15:46:11.977818+00
4	task1	-1	return_value	multiple_dependence	manual__2025-07-01T15:46:14.678168+00:00	"task1"	2025-07-01 15:46:25.062418+00
4	task2	-1	return_value	multiple_dependence	manual__2025-07-01T15:46:14.678168+00:00	"task2"	2025-07-01 15:46:36.182113+00
4	task3	-1	return_value	multiple_dependence	manual__2025-07-01T15:46:14.678168+00:00	"task3"	2025-07-01 15:46:36.236168+00



<push_to_xcoms.py>

Архитектура и компоненты

ТЫК

Минимальная конфигурация Airflow состоит из следующих компонентов:

- ⇨ Scheduler
- ⇨ Dag processor
- ⇨ Webserver
- ⇨ Папка с файлами DAG
- ⇨ Metadata database

Scheduler & Executor

Главные компоненты **Airflow**

Scheduler – это компонент Airflow, который отслеживает DAG-и и решает, когда запускать задачи на основе расписания (schedule_interval) и зависимостей

Компоненты:

- SchedulerJob: основной процесс
- Executor: выполняет таски
- DagFileProcessor: парсит даги в таблицу serialized_dag в Metadata DB

Executor – механизм, благодаря которому запускаются Task Instances

Executor определяется в airflow.cfg: executor = LocalExecutor. Можно задавать как встроенный executor, так и кастомный
Основные виды: LocalExecutor, SequentialExecutor, CeleryExecutor, KubernetesExecutor, EdgeExecutor

Можно одновременно использовать несколько executors
executor = LocalExecutor,CeleryExecutor

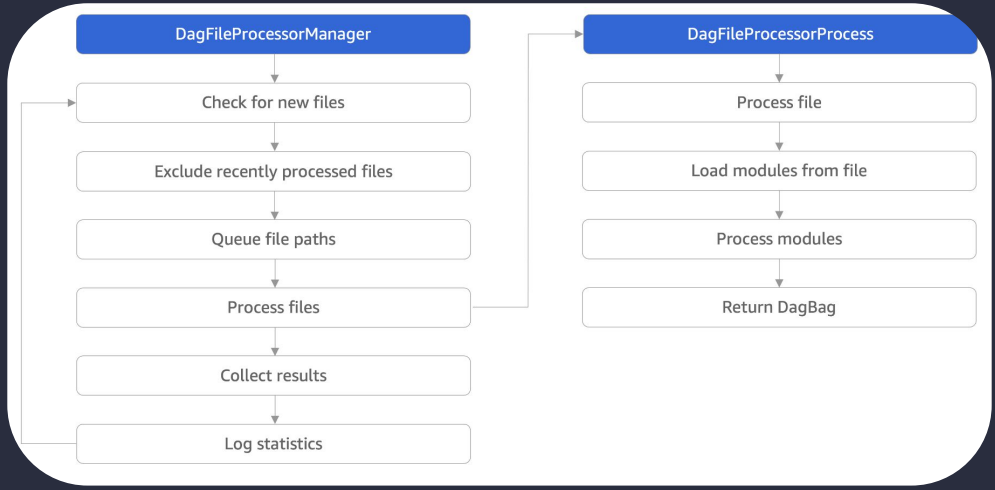
DAG File Processor

Парсинг и загрузка **DAG** файлов

DAG File Processing – это процесс чтения Python-файлов, содержащих DAG, и их сохранения для последующего использования шедулером

Система состоит из двух основных компонентов:

1. ***DagFileProcessorManager***
   a. Работает в бесконечном цикле
   b. Определяет, какие файлы требуют обработки
2. ***DagFileProcessorProcess***
   a. Запускается как отдельный процесс
   b. Преобразует каждый файл в один или несколько DAG-объектов



Код DAG'ов конвертируется в формат json

Metadata Database

Мозг **Airflow**

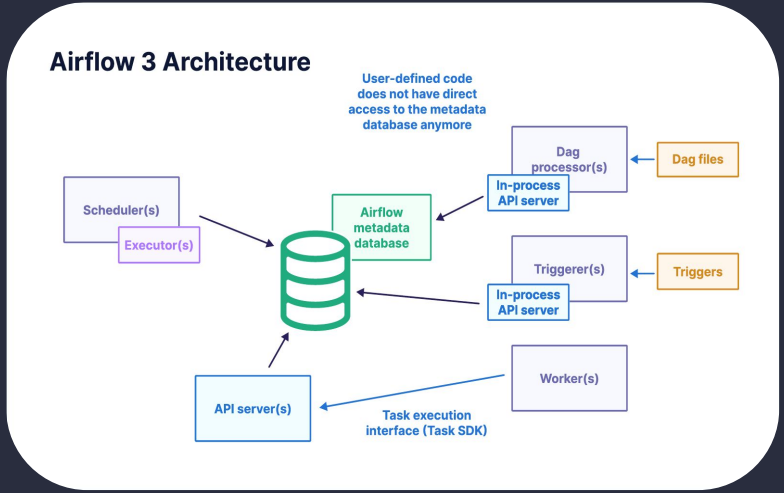
Metadata Database – центральная база данных Airflow, которая хранит:

- Состояние всех DAG, тасок и их зависимости
- Историю запусков, логи, конфигурации и переменные
- Данные о пользователях, ролях и подключениях

Чаще всего в качестве БД используется *Postgres*.

Best Practices:

- **Не использовать прямые SQL-запросы** к БД – только через Airflow API/UI
- Не хранить большие данные в XCom – используйте внешние хранилища (S3/GCS/БД) и регулярно очищайте / архивируйте метаданные, чтобы избежать перегрузки БД Airflow
- Лучше использовать управляемые БД для продакшена



Webserver & Folder of DAG files

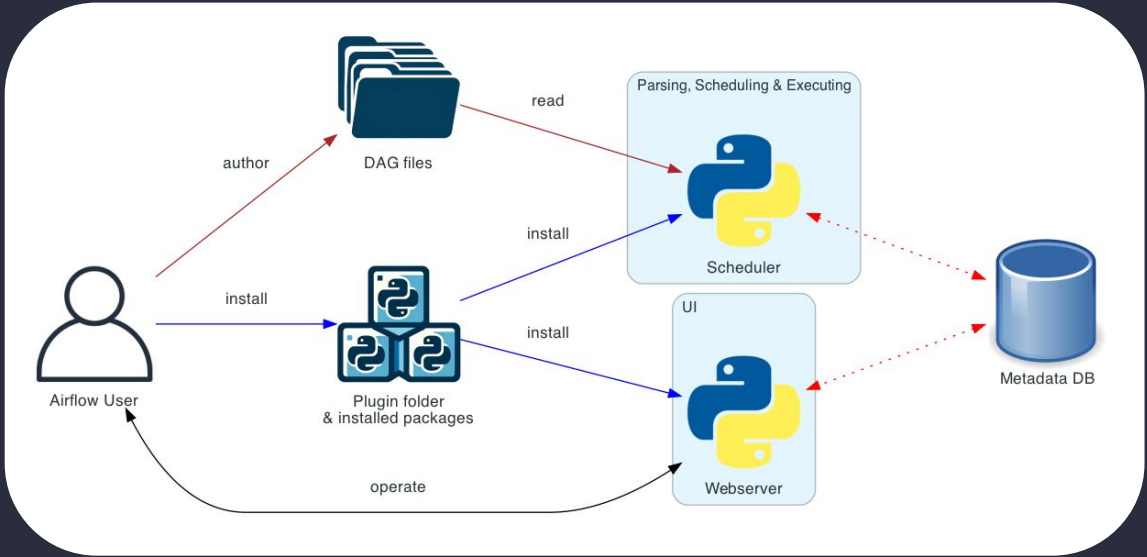
Webserver – это веб-интерфейс Airflow, который позволяет:

- Мониторить и управлять DAG'ами и тасками
- Администрировать
- Управлять подключениями и конфигурациями

Папка с дагами – основное хранилище для DAG-файлов, автоматически сканируется шедулером

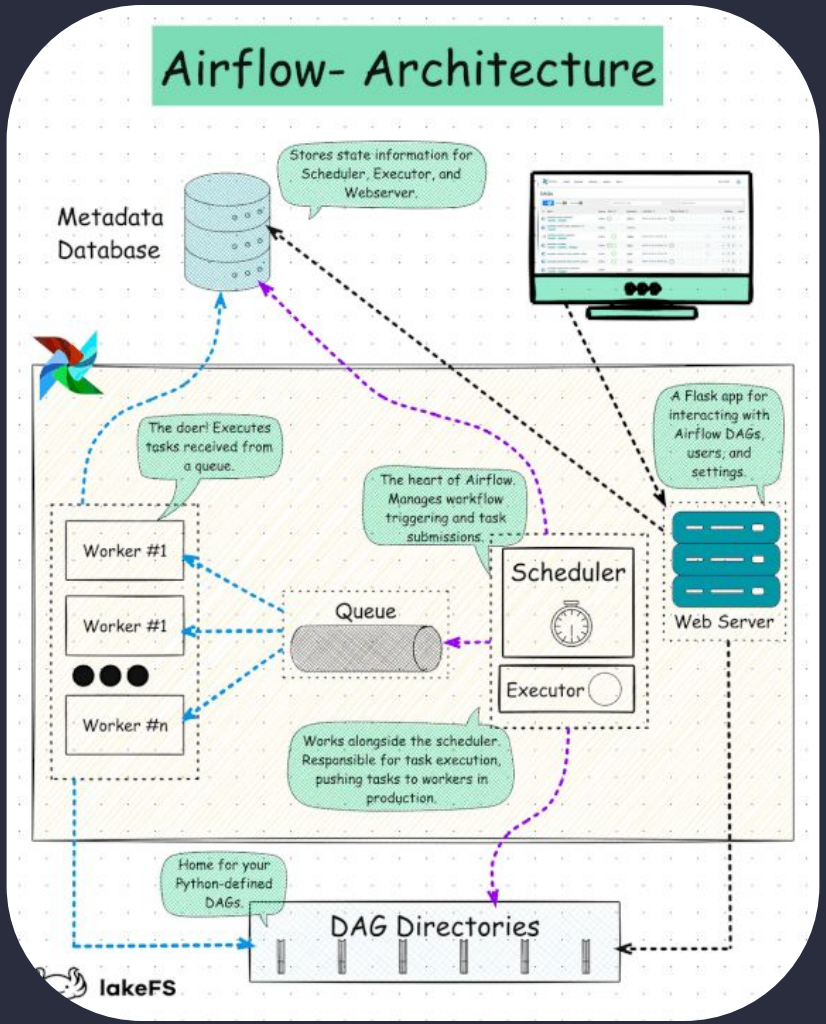
- Пусть настраивается в airflow.cfg: ключ dags_folder
- Файлы должны иметь разрешение .py
- Подпапки тоже сканируются, но можно отключить через load_examples=False
- Обязательно наличие переменной dag в глобальной области видимости

Простейшая архитектурная схема



- коричневые сплошные: отправка и синхронизацию DAG-файлов
- синие сплошные: развертывание и доступ к установленным пакетам и плагинам
- черные сплошные: доступ к интерфейсу для управления выполнением workflow
- красные пунктирные: доступ всех компонентов к базе данных метаданных.

Airflow- Architecture



Киллерфичи!

ТЫК

Продвинутые таски

Deferrable Operators & Triggers

<deferrable_sql_check_dag.py>

Компоненты системы

Deferrable Operator
Оператор, который умеет временно останавливать выполнение и освобождать воркер

Trigger
Асинхронный процесс, который ожидает событие

Triggerer
Отдельный процесс Airflow. Использует asyncio для эффективного ожидания
1 triggerer может обслуживать 100+ отложенных задач

Как это работает:

01
Оператор достигает точки, где ей нужно ждать и откладывает себя, освобождая рабочий слот

02
Триггер регистрируется в Airflow и выполняется в процессе triggerer

03
Когда триггер срабатывает, задача перепланируется в scheduler

04
Scheduler ставит задачу в очередь на возобновление выполнения на рабочем узле

ТЫК

Продвинутые зависимости

Control Flow

01 Branching

В таске ветвления выбирается следующая таска в зависимости от условий

<easy_branching.py>

Trigger Rules 02

Можно контролировать поведение, при котором будет выполняться следующая таска, используя trigger_rule

<trigger_rules.py>

03 Setup and Teardown

Предварительная настройка перед основными задачами DAG, и гарантия очистки ресурсов после

<setup_teardown.py>

Latest Only 04

Пропускает выполнение downstream-задач, если текущий запуск DAG не является последним

<latest_only.py>

05 Depends on Past

Таска запускается, если предыдущий запуск этой таски завершился успешно

<latest_only.py>

TaskFlow API & Timetables

тык

тык

Даги как **Python**-код

TaskFlow API – это синтаксис, который позволяет:

- Писать задачи как обычные Python-функции, используя декораторы
- Автоматически передавать данные между тасками через XCom
- Делать код DAG более читабельным и проще в тестировании
- Можно использовать с классическими операторами
- Возвращаемые данные из таски должны быть сериализуемы

<taskflow_api.py>

Timetable – это объект, который определяет когда запускать DAG и с каким logical date. Используется, когда нужна более сложная логика расписания дага

Примеры

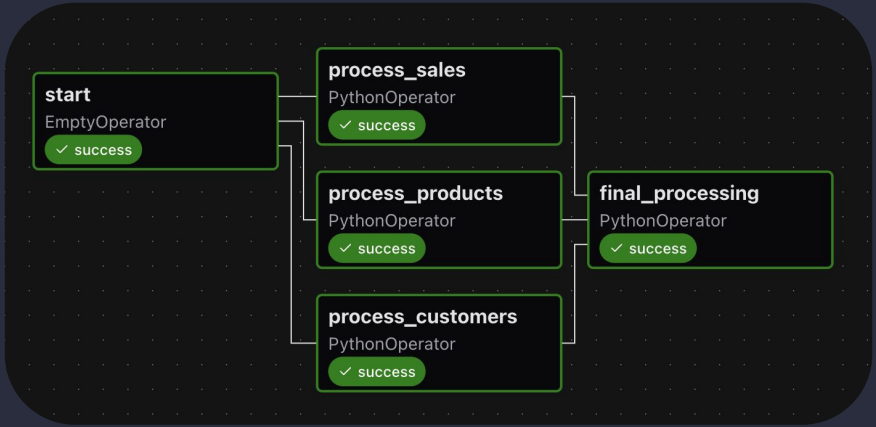
```
DeltaTriggerTimetable(timedelta(days=5)) # раз в 5 дней

CronTriggerTimetable("0 18 * * 5", interval=timedelta(days=4, hours=9)) # каждую пятницу в 18:00, чтобы покрыть
9:00 Monday to 18:00 Friday
```

Dynamic DAGs

Генерация дагов

Dynamic DAG – DAG, который генерируется программно во время загрузки Airflow, а не хранится как статичный файл

- Позволяет создавать шаблонные пайплайны с разными параметрами
- Избегает дублирования кода



<dynamic_dag.py>

Connections & Hooks

Подключение к внешним системам

Connection – это набор параметров (логин, пароль, хост, порт и др.), которые определяют, как подключиться к внешней системе

Ключевые особенности:

- Каждое подключение имеет уникальный conn_id
- Параметры подключения можно задавать
- Подключения можно использовать:
  - Напрямую в коде
  - Через Hooks
  - В шаблонах: {{ conn.postgres_default.host }}

Хранятся:

- По умолчанию в базе данных Airflow
- Можно настроить внешнее хранилище

Hooks – интерфейс для работы с внешними системами. Используют Connection для авторизации

Принцип работы:

1. Берет credentials из Connection
2. Управляет подключениями
3. Предоставляет удобные методы

```
hook = PostgresHook(postgres_conn_id="my_postgres_conn")
df = hook.get_pandas_df("SELECT * FROM my_table")
```

Backfill & Catchup

Догонка пропущенных **DAG**

Catchup – автоматическое создание пропущенных DagRun при включении DAG или изменении start_date

Если catchup=True (по умолчанию), Airflow создаст все DagRun от start_date до текущей даты. Если catchup=False, запустится только один DagRun для текущего интервала.

Backfill – ручной запуск DagRun для конкретного периода вне расписания

API + CLI + UI & RBAC

Администрирование и взаимодействие

RBAC – система ролевой модели доступа с тонкой настройкой прав
Ключевые роли:

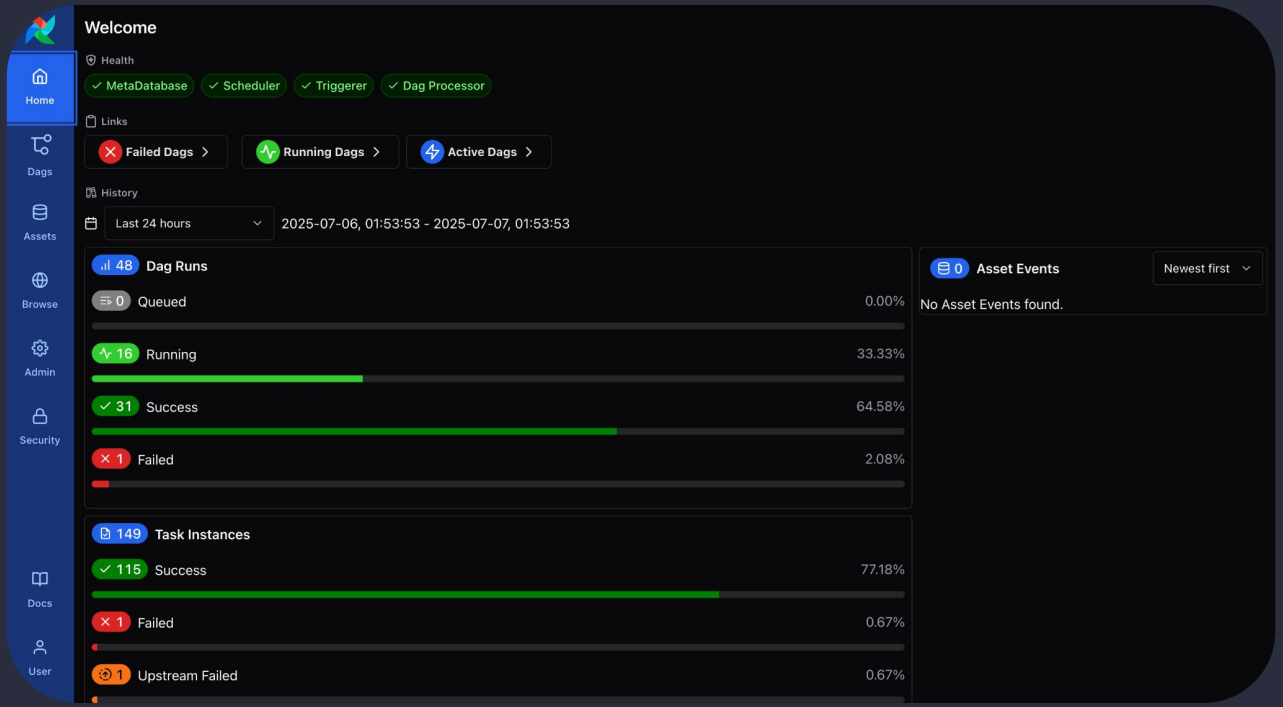
- ***Admin***: Полный доступ
- ***User***: Запуск/остановка своих DAG
- ***Viewer***: Только просмотр
- ***Op***: Редактирование задач + доступ к Connections

API + CLI + UI
С Airflow можно взаимодействовать с помощью REST API, CLI, UI
Почти любую команду можно выполнить, используя подходящий способ, чтобы значительно повышает гибкость для управления

Airflow 3

Черная тема 😏

Ура

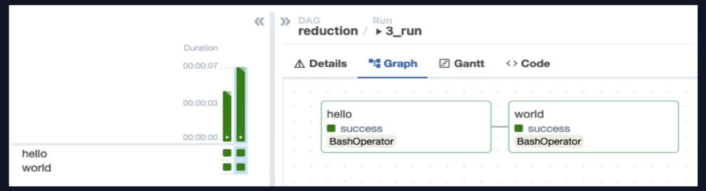


DAG Bundle & DAG versioning

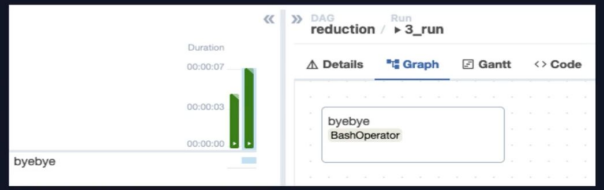
Управление версиями и организация **DAG**-файлов

Went from 2 to 3 tasks

Reduce back to 2 tasks



Completely different tasks



Было:

- Последний код DAG применяет ко всем DAG runs
- Изменения DAG меняют историю, усложняя отслеживание истории изменения
- Если код DAG был изменен в процессе работы DAG, то будет применен последний код, что может сломать всю логику

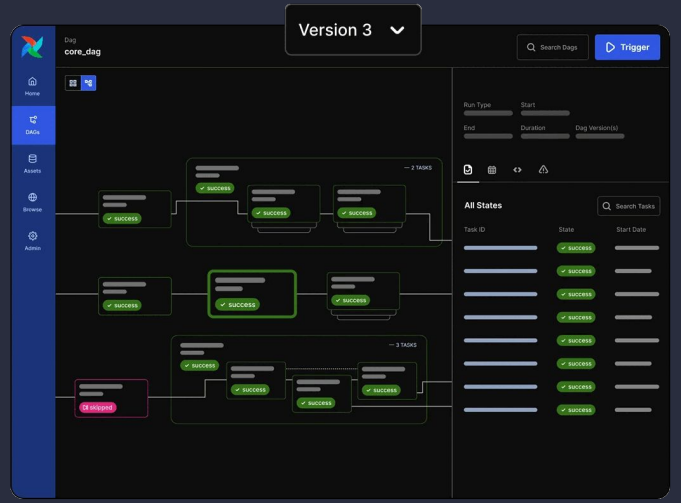
DAG Bundle & DAG versioning

DAG Bundle – это запакованный набор DAG-файлов и их зависимостей, включая:

- Python-скрипты
- Конфигурационные файлы
- Ресурсы

Ключевые преимущества

- ***Версионирование***
  - Фиксация версии DAG и зависимостей
  - Гарантия, что DagRun завершится с тем же кодом, с которого начался
- ***Гибкость***
  - Поддержка Git, S3, локальных файлов и кастомных хранилищ
  - Несколько бандлов в одном Airflow-окружении



<test_dag_bundle_git.py>

Assets

Event-driven DAGs

Asset – логическая единица данных, представленная уникальным URI (s3://bucket/data.csv)

```
asset = Asset(
    uri="s3://etl-bucket/sales_2024.csv",
    name="monthly_sales"
)
```

Используется для:

- Отслеживания, кто создал или использует эти данные
- Автоматического запуска DAG-ов при обновлении данных
- Построения зависимостей между DAG-ами

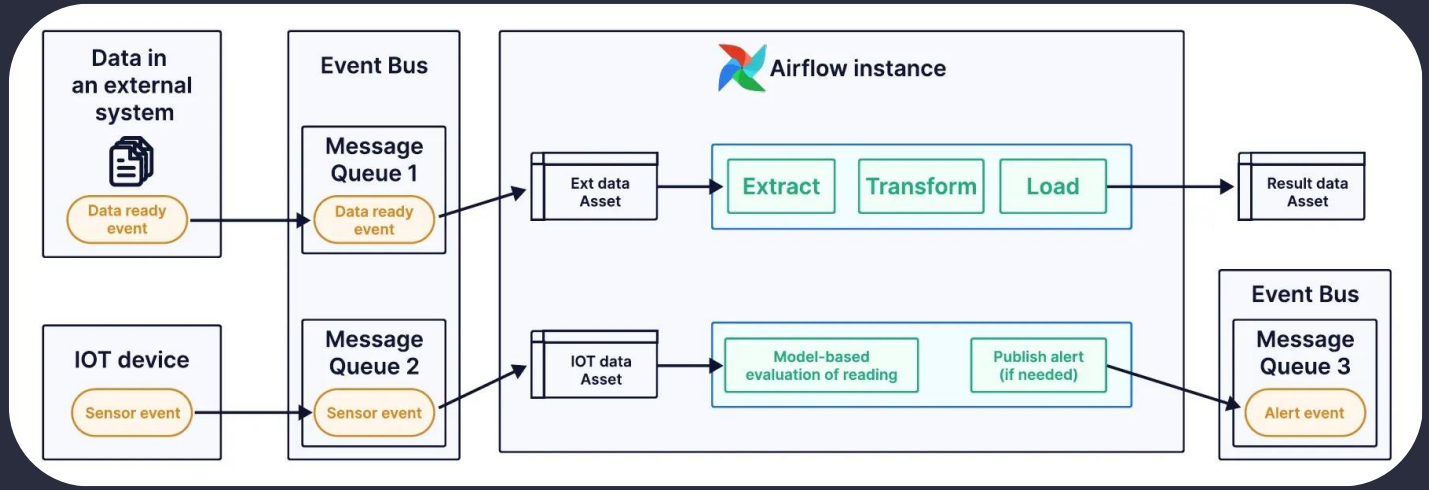
тэг: assets

Assets

Event-driven DAGs

Assets vs Datasets:

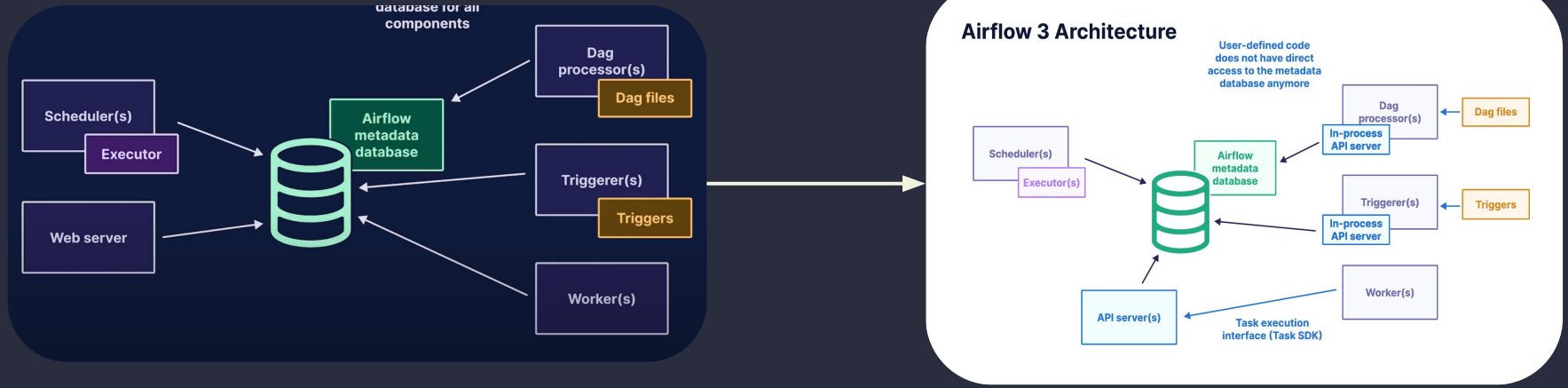
- Watchers: отслеживаем события вне Airflow
- Asset-центральный синтаксис



Остальные важные изменения

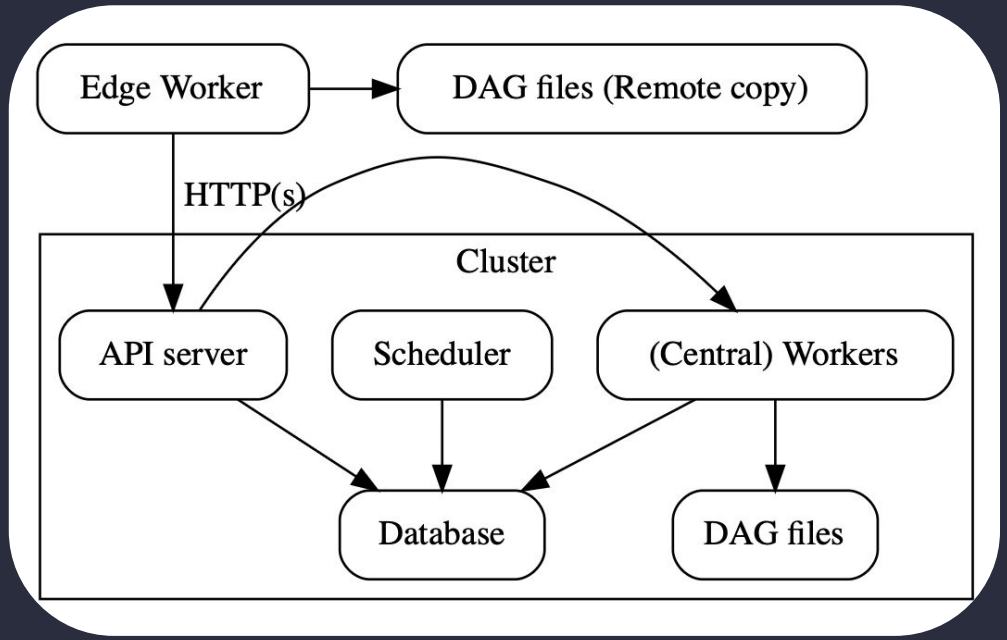
Backfill теперь можно воспользоваться через UI

Обновленная **архитектура**: повышение безопасности, поддержка разных языков, удаленный запуск



Остальные важные изменения

EdgeExecutor: можно запускать даг на любой машине



Когда не **Airflow?**

- ⇨ Запуск одного скрипта по простому расписанию – cron справится лучше
- ⇨ Streaming загрузки – попробуйте Flink + Kafka
- ⇨ Когда нужно быстро собрать простую логику – можно использовать NiFi

Вопросы?

GitHub с материалами

Registry Astronomer

Миграция с **Airflow 2.x** на **Airflow 3.x**

Спасибо за внимание!