Apache Airflow

Aa Name	Tags
Untitled	
Untitled	
Untitled	

▼ Опыт

SLA (Service Level Agreement) — это соглашение об уровне обслуживания, заключённое между поставщиком услуги (например, IT-компания, дата-центр или отдел ИТ) и пользователем этой услуги (компания, команда или клиент).

Проще говоря, SLA — это договор, где описано, какого качества и уровня обслуживания можно ожидать от услуги. В нём прописаны ключевые параметры, такие как:

1. **Время отклика**: Как быстро должна быть оказана помощь или выполнена работа (например, время ответа на запрос или запрос на техническую поддержку).
2. **Время доступности**: На какой процент времени услуга должна быть доступна (например, 99.9% времени в месяц).
3. **Показатели производительности**: Качество и скорость работы услуги, такие как скорость загрузки сайта или время выполнения операций.
4. **Штрафы за невыполнение условий**: Условия компенсации или штрафов, если поставщик не выполняет условия SLA.

SLA помогает пользователю понимать, на какую надёжность и скорость он может рассчитывать, а поставщику — видеть чёткие критерии для поддержания качества.

Примеры:

- **Критические запросы** могут иметь SLA на отклик в 10 минут. Например, если система выходит из строя, то поддержка обязана ответить и начать устранять проблему в течение 10 минут.

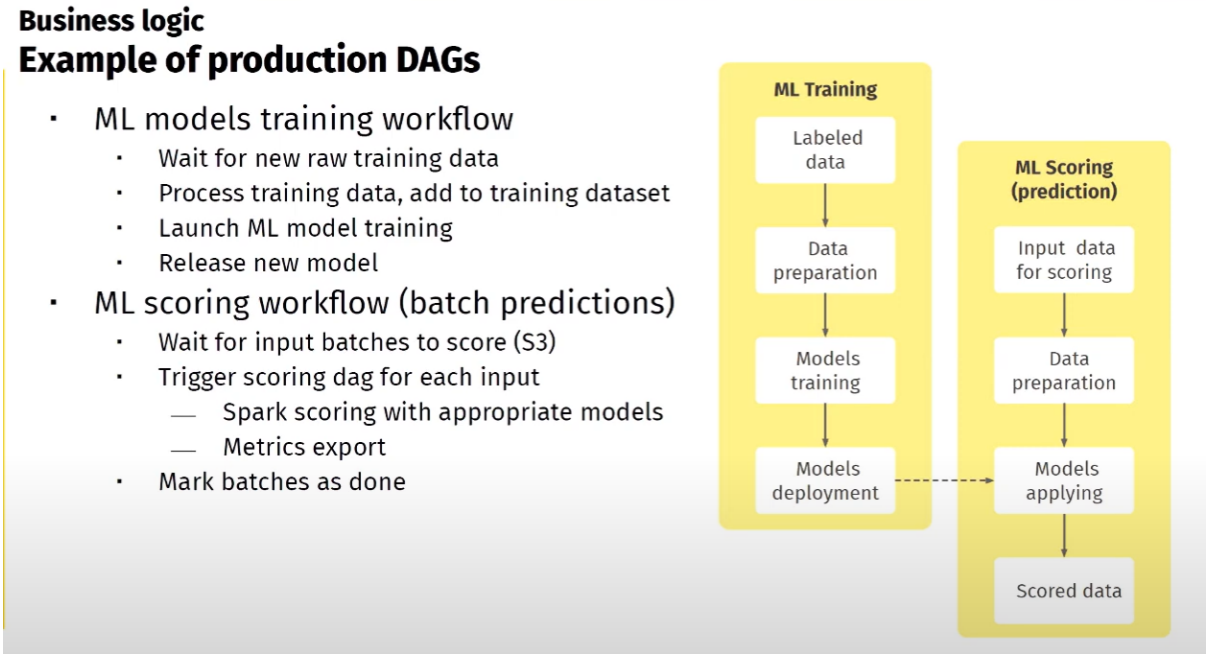
- **Менее приоритетные запросы** могут иметь SLA на несколько часов. Например, для незначительных запросов или запросов на изменение, которые не влияют на работу бизнеса, отклик может быть гарантирован в течение нескольких часов.

Один из проектов в сбере: анализ рекламы, сбор анализа по просмотру рекламы, анализируем, собираем модельки и генерируем репорты для клиентов, которые должны улучшать показатели рекламных компаний. Объем данны: 10тки терабайтов в сутки, SLA (время реагирования) от 10 минут до нескольких часов, батч процессинг с фокусом на ML.

Бизнес-модель

Организовывала Apache Airlow для ML пайплайна. Первый пайплайн обучал модельку. Изначально загружали датасет, размеченный аналитиками, потом был препроцессинг, обучение, а на выходе получали артефакт модели, который деплоили в обжект сторедж.

Второй пайплайн — получали батч, готовили данные, преминяли модель (выбирали изходя из метрик или свежие, а на выходе получали готовый батч)

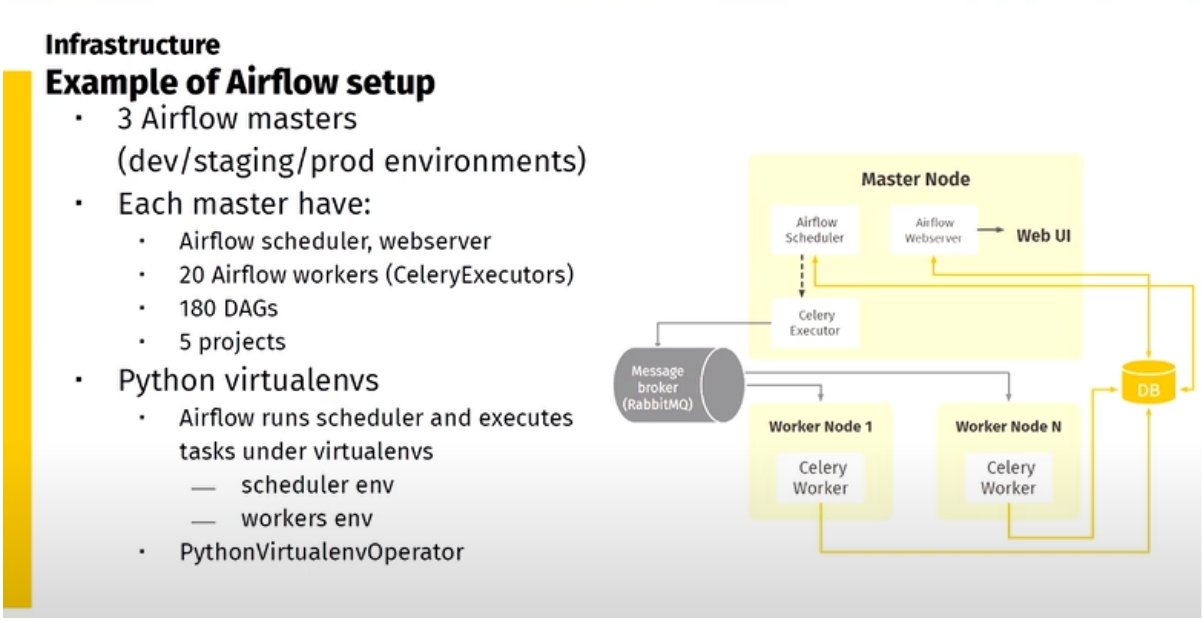


Инфраструктура

Есть мастер нода, есть нескольсколько воркеров. На мастер ноде хостится шедулер и веб-сервер. На воркерах случают очередь, забирают таску на исполнение и возвращают результат на мастер.

Необходим брокер сообщений (RabbitMQ) + есть БД Airflow. Под каждый слой (дев, тестовая среда и staging) было по кластеру Aiflow. На каждом мастере было около 200 дагов, и 20 Aiflow воркеров. Всего было несколько проектов, поэтому тут присуствуют даги с соседних проектов, которые могут друг друга аффектить.

Процессы на мастере и на воркере стартовали под своим python virtualenv. Поэтому если всякие зависимости были внутри них.



Как было внутри: был пайплайн, который анализирует рекламный компании, и зависимости от id рекламной компании мы с разными конфигурациями, динамически под клиентов генерим пайпланы, в случае падения они не аффектили другие пайпайны → прозрачно виден анализ рекламной компании.

Делали тестирование: импорт дага без ошибок (все зависимости, что даг существует, что у него есть таски в нужном количестве). QA через рест тригеррили даги и смотрели логи, интегрировали в CI/CD loop.

Настраивали всякие мелочи по типу Airflow ignore (скан папок сократился с 10с до 1)

`airflow.cfg` → `dag_ignore_file_syntax = .` `\.jar$|.` `\.pyc$|.` `__pycache__` `.` `|.*\.tmp$`

Какие операторы использовала: PythonOperator, Spark_SQL operator, Hive operator, самописный оператор к API мониторингу CTL.

Apache AirFlow — это open-source инструмент, который позволяет **разрабатывать, планировать и осуществлять мониторинг сложных рабочих процессов**. Главной особенностью является то, что для описания процессов используется язык программирования **Python**. Airflow используется как планировщик ETL/ELT-процессов.

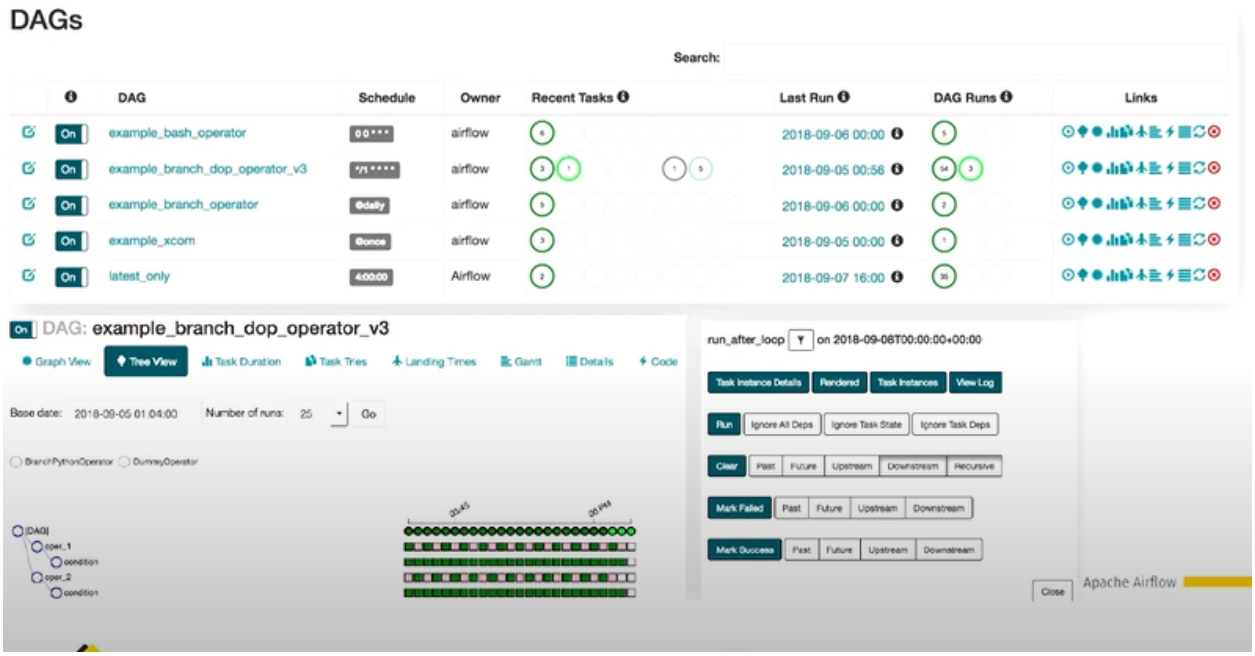
Изначально ориентируется на Batch, а не NRT операции.

✅ Когда выбираем

- **Сложные ETL-пайплайны с зависимостями**: Airflow отлично подходит для управления зависимостями между задачами и сложными процессами обработки данных.
- **Регулярные процессы с расписанием**: Airflow создан для автоматизации периодических задач, таких как ежедневная загрузка данных и обновление аналитики.
- **Интеграция с большим количеством источников**: Подходит, если нужно собирать, обрабатывать и объединять данные из разных систем (SQL, облачные хранилища, API).
- **Масштабируемая обработка данных**: Airflow можно масштабировать, используя Celery или Kubernetes для распределённой обработки задач.

⛔ Когда не подойдем

- **Потоковая обработка данных**: Если данные нужно обрабатывать в реальном времени (streaming), лучше выбрать специализированные инструменты, такие как Apache Kafka, Apache Flink или Spark Streaming.
- **Маленькие, простые задачи без сложных зависимостей**: Если требуется выполнение простых скриптов без сложного планирования и зависимостей, Airflow может быть излишне сложным. В таких случаях подойдёт cron или простые планировщики задач.
- **Высокочастотные задачи с минимальной задержкой**: Airflow не оптимизирован для задач, запускаемых каждую минуту или чаще, из-за ограничений планировщика и задержек.
- **Интерфейсные или пользовательские процессы**: Airflow предназначен для бэкэнд-обработки и не подходит для выполнения задач, требующих взаимодействия с пользователем в реальном времени.



Пример того, как выглядит WEB UI с разными дагами и параметрами запуска

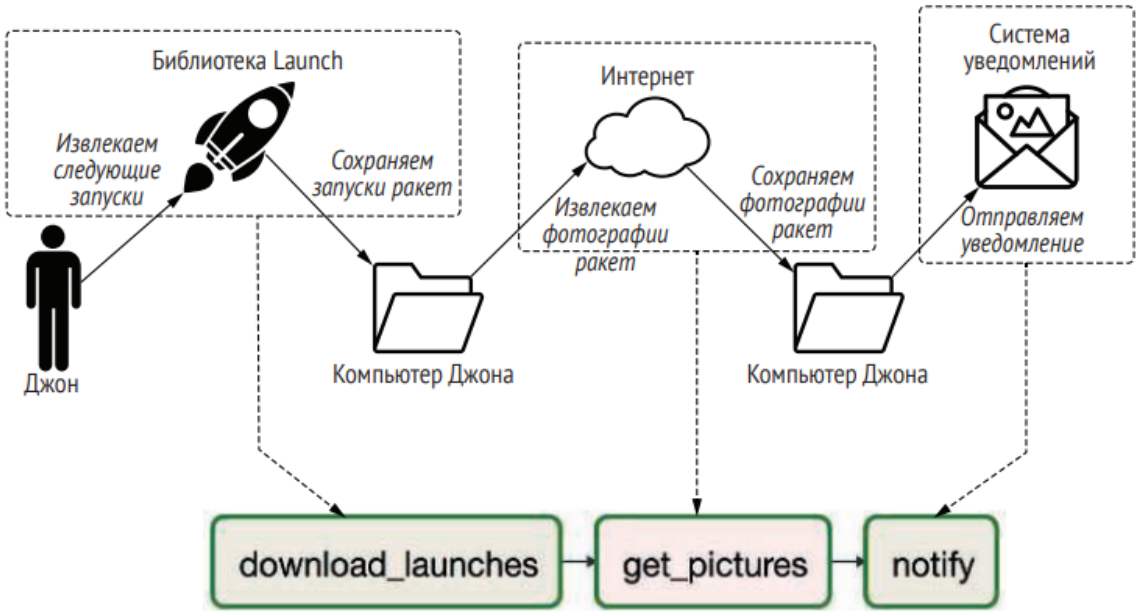


Рис. 2.3 Ментальная модель Джона, сопоставленная с задачами в Airflow

▼ Основные компоненты

▼ DAG

DAG (Directed acyclic graph) — это ориентированный ациклический граф. В Apache Aiflow сложный конвеер преобразования данных и их последовательного запуска мы можем представить в виде графа. **Основная задача DAG** — организовать выполнение набора операторов.

Обрати внимание!

Чтобы создать DAG, необходимо три основных параметра: id, start_day, schedul
Они обязательны.

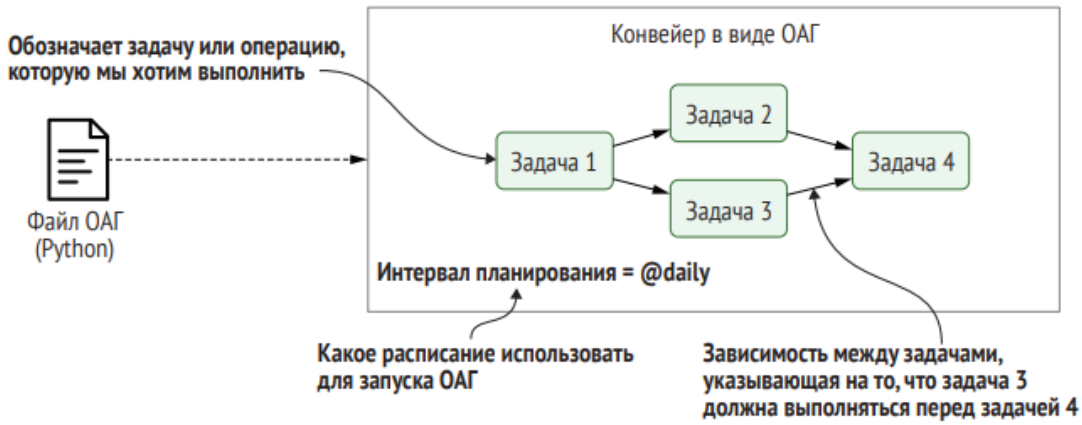
```
with DAG(
     dag_id="sample_dag",
    start_date=datetime(year=2024, month=1, day=1, hour
    schedule_interval="@daily",
) as dag:
    …
```

▼ Запуск по интервала

DAG мы описываем на языке **Python** в отдельном файле.Каждый файл DAG обычно описывает набор задач для данного графа и **зависимости между задачами**, которые затем анализируются Airflow для определения структуры графа. Помимо этого, эти файлы обычно содержат некоторые дополнительные метаданные о графе, сообщающие Airflow, как и когда он должен выполняться, и так далее.

Почему Python

- Гибкость при создании DAG’ов (например, кодом сделать динамическую генерацию доп. задачек в зависимости от условий)
- Задачи могут выполнять любую операцию, которую можно реализовать на Python



▼ *Пример кода Dag'ax*

Импорты:

```
import json
import pathlib
import airflow
import requests
import requests.exceptions as requests_exceptions
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
```

Создаем и инициализируем DAG

```
dag = DAG( //dag - имя Dag'a
 dag_id="download_rocket_launches",
 start_date=airflow.utils.dates.days_ago(14), //когда его
 schedule_interval=None, //интервал запуска
)
```

Пример добавления BashOperator'a

```
download_launches = BashOperator(
 task_id="download_launches",
 bash_command="curl -o /tmp/launches.json -L
```

```
'https://ll.thespacedevs.com/2.0.0/launch/upcoming'",
 dag=dag, //не забыть указать название дага
)
```

Как показываем очередность в даге:

```
download_launches >> get_pictures >> notify
```

В Airflow можно использовать бинарный оператор сдвига вправо (например, «rshift» [>>]) для определения зависимостей между задачами. Это гарантирует, что **задача 1** запустится только после успешного завершения **задачи 2**.

▼ Task/Operator

Operator — это шаблон или класс, который описывает тип действия, которое нужно выполнить. Он определяет, какой тип операции будет выполняться, но ещё не запускает саму задачу.

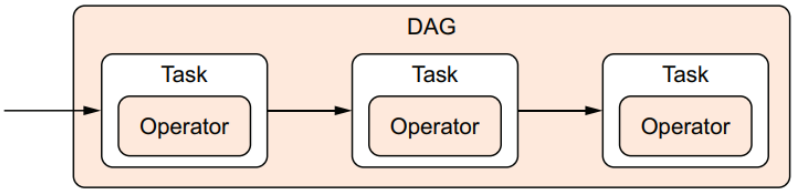
Типы операторов:

- **PythonOperator**: выполняет Python-функцию.
- **BashOperator**: выполняет команду в Bash.
- **SQL оператор**: для выполнения SQL-запросов (например, PostgresOperator, MySqlOperator).
- **SensorOperator**: ожидает выполнения события, например, появления файла в S3.

***Примеры использования**: Операторы — это "шаблоны" действий, такие как "выполнить SQL-запрос" или "выполнить Python-скрипт".*

Task — это конкретный экземпляр оператора в DAG. Это единичное выполнение операции, созданное на основе оператора и привязанное к расписанию и зависимостям в рамках DAG. Главное отличие от оператора то, что задача — это оболочка, менеджер вокруг оператора, которая шаблон реализует.

***Пример**: Если вы создаёте Task `task1` с использованием `PythonOperator`, это значит, что `task1` будет выполнять определённую Python-функцию в соответствии с заданным расписанием и зависимостями.*



Зачастую понятие task и operator взаимозаменяемое: по сути, это узел нашего DAG’a, который мы применяем к нашим источникам. Например, оператор BashOperator используется для написания скриптика или PythonOperator (для написания функции питона). В любом случае, это кусочек работы (piece of work).

▼ Variables

Variables в Apache Airflow — это механизм хранения и управления значениями, которые могут быть использованы в DAG и задачах для настройки или передачи информации. Переменные полезны для хранения значений, которые часто меняются, но должны оставаться доступными в разных задачах или DAG'ах, таких как пути к файлам, параметры подключения, секретные ключи и другие параметры конфигурации.

Глобальная доступность: Переменные создаются на уровне всей системы Airflow и доступны для всех DAG'ов и задач, что позволяет переиспользовать значения в нескольких рабочих процессах.

Хранение и управление через UI, CLI и API: Переменные можно управлять через **веб-интерфейс**, **CLI** и **REST API**:

- **Веб-интерфейс**: Раздел **Admin > Variables**, где можно создать, изменить или удалить переменные.
- **CLI**: Например, команда `airflow variables set key value` для создания переменной.
- **API**: Через вызов API для автоматизированного управления переменными.

Использование в DAG и задачах: Переменные можно использовать прямо в коде DAG и задач с помощью `Variable.get("key")`.

```
from airflow.models import Variable

# Получение значения переменной
my_value = Variable.get("my_variable_key")
```

Шифрование конфиденциальных данных. Переменные, содержащие конфиденциальные данные (например, пароли или API-ключи), могут быть зашифрованы с помощью ключей шифрования в Airflow. Это обеспечивает дополнительную безопасность для хранения чувствительной информации.

Переменные по умолчанию: При вызове `Variable.get()` можно задать значение по умолчанию, которое будет возвращено, если переменная не найдена.

```
my_value = Variable.get("non_existing_key", default_var="de
fault_value")
```

Поддержка JSON Переменные могут хранить значения в формате JSON, что удобно для хранения сложных структур данных, таких как списки или словари.

```
my_json_data = Variable.get("my_json_key", deserialize_json
=True)
```

Примеры использования Variables в Airflow

- **Хранение путей к файлам и параметров конфигурации**: Например, переменные можно использовать для хранения путей к данным, чтобы менять их по мере необходимости, не изменяя код DAG'a.
- **Хранение ключей и токенов**: Переменные удобны для хранения API-ключей и токенов, необходимых для подключения к внешним системам.
- **Параметры для динамических DAG'ов**: Переменные могут быть полезны для параметризации DAG'ов. Например, можно использовать переменную для хранения значения даты или диапазона, которые будут использоваться для фильтрации данных при запуске DAG'a.

▼ Executor

SequentialExecutor

- **Описание**: Выполняет задачи последовательно, одна за другой.
- **Преимущества**: Легковесный и простой, так как работает без параллелизма.
- **Когда использовать**: Подходит для тестирования или разработки, особенно если требуется запустить Airflow на одной машине с минимальными ресурсами и без сложной установки.
- **Недостаток**: Не поддерживает параллельное выполнение задач, что делает его непрактичным для production.

LocalExecutor

- **Описание**: Выполняет задачи параллельно на одной машине, используя потоки или процессы.
- **Преимущества**: Поддерживает параллелизм, более производителен, чем SequentialExecutor, так как использует ресурсы одного узла для одновременного выполнения задач.
- **Когда использовать**: Хороший выбор для небольших и средних проектов, где задачи могут быть выполнены на одной машине без распределённой архитектуры. Подходит для разработки и небольших production-систем.
- **Недостаток**: Ограничен мощностью одной машины, поэтому не подходит для высоконагруженных систем.

1. **CeleryExecutor**
    - **Описание**: Работает с распределённой архитектурой, используя Celery для управления задачами, которые обрабатываются на нескольких воркерах.
    - **Преимущества**: Масштабируем и может обрабатывать множество задач параллельно, распределяя их по нескольким серверам. Использует брокер сообщений (например, Redis или RabbitMQ) для координации между воркерами.
    - **Когда использовать**: Подходит для production-систем с высокой нагрузкой, требующих параллельной и распределённой обработки задач. Особенно эффективен в средах, где есть несколько серверов или виртуальных машин для воркеров.
    - **Недостаток**: Требует настройки и управления брокером сообщений и воркерами, что усложняет установку и обслуживание.

2. **KubernetesExecutor**
    - **Описание**: Запускает каждую задачу в отдельном поде Kubernetes, что позволяет динамически масштабировать задачи и ресурсы.
    - **Преимущества**: Высокая гибкость и масштабируемость, так как Kubernetes автоматически выделяет ресурсы на основе требований задачи. Поддерживает изоляцию задач и позволяет точно настроить ресурсы (например, память, CPU) для каждой задачи.
    - **Когда использовать**: Рекомендуется для облачных решений и гибридных архитектур, а также для production-систем с высокой нагрузкой, где требуется гибкость и точный контроль над ресурсами.
    - **Недостаток**: Сложность в установке и настройке, требует использования Kubernetes-кластера.

Сравнение CeleryExecutor и LocalExecutor

- **Архитектура**:
    - **LocalExecutor**: Выполняет задачи локально на одной машине, используя потоки или процессы.
    - **CeleryExecutor**: Распределённая архитектура с поддержкой нескольких воркеров, которые могут выполняться на разных серверах.
- **Параллелизм и масштабируемость**:
    - **LocalExecutor**: Параллелизм ограничен ресурсами одного узла (CPU и память).
    - **CeleryExecutor**: Масштабируем за счёт распределённых воркеров, можно добавить больше машин для увеличения параллельности.
- **Использование ресурсов**:
    - **LocalExecutor**: Легче установить и поддерживать, но требует мощного узла для обработки больших объёмов данных.
    - **CeleryExecutor**: Требует настройки брокера сообщений (например, Redis, RabbitMQ) и может потребовать больше администрирования.

Когда использовать каждый из экзекьюторов:

- **SequentialExecutor**: Для разработки и тестирования без параллелизма.

- **LocalExecutor**: Для небольших или средних проектов, когда всё выполняется на одной машине и требуется ограниченный параллелизм.
- **CeleryExecutor**: Для высоконагруженных production-систем, требующих распределённого выполнения задач на нескольких воркерах.
- **KubernetesExecutor**: Для облачных сред, где необходимы гибкость, высокая масштабируемость и изоляция задач. Подходит для проектов с высокими требованиями к производительности и для сред с динамически меняющимися нагрузками.

▼ XCom

XCom (Cross-communication) в Apache Airflow — это механизм для обмена данными между задачами внутри одного DAG. XCom позволяет передавать небольшие объёмы данных из одной задачи в другую, что полезно для динамического использования результатов, полученных на предыдущих этапах. M*ожно им пользоваться, но лимит до 2Гб. Желательно сгружать данные в БД и не передавать их по XCOM*

XCom лучше не использовать для передачи больших объёмов данных, так как они хранятся в базе метаданных Airflow и могут перегружать её.

Основные особенности XCom

1. **Передача данных между задачами**:
   - XCom позволяет одной задаче записывать данные, которые другая задача может затем извлечь и использовать.
   - Например, задача A может получить данные из API и передать их задаче B для дальнейшей обработки.
2. **Механизм «ключ-значение»**:
   - XCom использует пары ключ-значение для хранения данных. Когда задача записывает данные в XCom, она может указать ключ, чтобы другая задача могла найти и извлечь это значение.
3. **Поддержка произвольных данных**:

- XCom поддерживает передачу различных типов данных, включая строки, числа, списки и даже более сложные объекты Python. Однако рекомендуется использовать XCom только для передачи небольших объёмов данных , поскольку все данные хранятся в базе данных метаданных Airflow, и большие объёмы могут перегружать её.

4. **Методы работы с XCom**:
   - **push**: Метод, который передаёт данные в XCom. Обычно вызывается в `return` функции оператора, если передача значений задана в PythonOperator.
   - **pull**: Метод, который извлекает данные из XCom. Вызывается в другой задаче, которая получает результат от предыдущей.

XCom vs. Variable: Сравнение

Критерий	XCom	Variable
Основная цель	Передача данных между задачами	Хранение глобальных конфигураций и данных
Срок хранения	Только в рамках выполнения DAG	Постоянное (сохраняется между DAG-запусками)
Область видимости	Только внутри одного DAG	Глобально для всех DAG
Способ работы	`push` и `pull` между задачами	Доступ через `Variable.get()` и `Variable.set()`
Размер данных	Небольшие данные	Может использоваться для любых данных (с осторожностью для больших)
Хранение данных	В базе данных Airflow	В базе данных Airflow
Типы данных	Поддерживает разные типы данных	Поддерживает строки и JSON

Пример использования XCom для передачи данных

Допустим, у нас есть DAG, где одна задача извлекает данные из API, а другая задача должна использовать их для обработки.

```
from airflow import DAG
from airflow.operators.python import PythonOperator
```

```
from datetime import datetime

def fetch_data(**kwargs):
    data = {"key": "value"}
    # Сохраняем данные в XCom
    kwargs['ti'].xcom_push(key='api_data', value=data)

def process_data(**kwargs):
    # Извлекаем данные из XCom
    data = kwargs['ti'].xcom_pull(key='api_data')
    print(f"Processing data: {data}")

with DAG(dag_id="xcom_example_dag", start_date=datetime(202
3, 1, 1), schedule_interval="@daily") as dag:
    fetch_task = PythonOperator(
        task_id="fetch_data",
        python_callable=fetch_data,
        provide_context=True
    )

    process_task = PythonOperator(
        task_id="process_data",
        python_callable=process_data,
        provide_context=True
    )

    fetch_task >> process_task
```

Когда использовать XCom вместо Variable:

- Используйте **XCom**, если данные нужны только на этапе выполнения DAG и не будут использоваться в других DAG или при следующих запусках.
- Используйте **Variable**, если вам нужно хранить настройки, параметры, данные между запусками DAG или делиться ими между различными DAG.

Таким образом, XCom отлично подходит для временного обмена данными между задачами внутри одного DAG, в то время как Variables лучше использовать для

хранения постоянных параметров и конфигураций, доступных глобально.

Обрати внимание!

Данные в XCom не удаляются автоматически, поэтому они будут накапливаться базе данных метаданных. Это требует регулярной очистки, чтобы не перегружа базу данных и избежать увеличения объёма хранимых данных. **Вместо передач данных через XCom, используйте базу данных для хранения данных и передавайте ссылки на них через XCom.**

```
def save_data_to_db(**kwargs):
    data = {"key": "value"}
    # Подключение и запись в базу данных
    conn = sqlite3.connect('/path/to/my_database.db')
    cursor = conn.cursor()
    cursor.execute("INSERT INTO my_table (data) VALUES
    conn.commit()
    conn.close()
    # Передача ссылки на данные через XCom
    kwargs['ti'].xcom_push(key='db_reference', value='/
```

▼ Sensor

Sensor — это специальный оператор, который "ожидает" наступления определённого события или выполнения условия, прежде чем продолжить выполнение DAG. Sensor полезен для проверки, готово ли условие для выполнения следующей задачи, например, для ожидания появления файла или завершения внешнего процесса.

Что такое Backfill и Catchup, и в чём между ними разница?

Ответ:

- **Backfill** — это выполнение DAG задним числом за указанный диапазон дат, что позволяет выполнить DAG за пропущенные периоды.
- **Catchup** — параметр в настройках DAG, который при значении `True` автоматически запускает DAG для всех пропущенных периодов. Если `Catchup=False`, Airflow будет выполнять DAG только для текущей даты без выполнения за пропущенные интервалы.

Чем отличается ETL от ELT, и как Airflow поддерживает оба подхода?

Ответ: **ETL** — это процесс, при котором данные **Extract** (извлекаются) из источника, **Transform** (преобразуются) и затем **Load** (загружаются) в целевую систему. В **ELT** данные сначала **Extract** (извлекаются), затем **Load** (загружаются в хранилище данных), и уже там выполняются **Transform** (преобразования). Airflow поддерживает оба подхода за счёт гибкой структуры DAG и операторов, позволяя настраивать этапы в любом порядке.

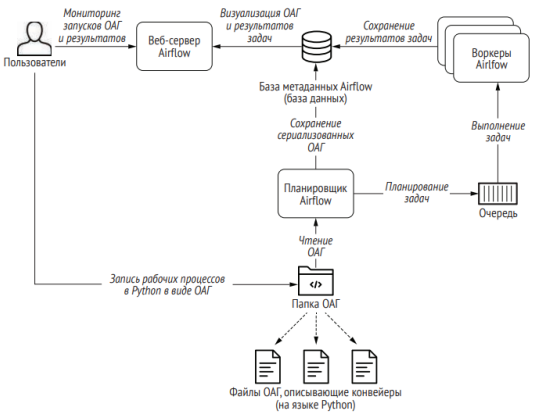
Каковы лучшие практики для использования XCom и Variables в Airflow?

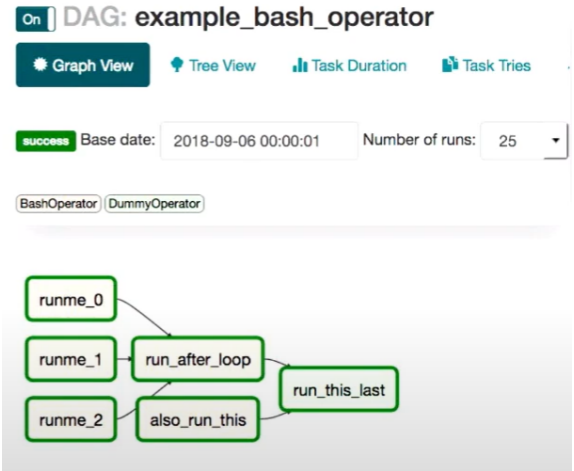
Ответ:

- **XCom**: Использовать только для небольших объёмов данных и кратковременной передачи данных между задачами. Для больших объёмов данных лучше использовать хранилище и передавать ссылки через XCom.
- **Variables**: Использовать для хранения глобальных конфигураций и параметров, доступных в нескольких DAG. Также следует избегать использования Variables для больших объёмов данных и чувствительной информации, если не включено шифрование.

▼ Архитектура

Airflow позволяет вам определить параметр `schedule_interval` для каждого графа, который точно решает, **когда ваш конвейер будет запущен Airflow**. Таким образом, вы можете дать указание Airflow выполнять ваш граф каждый час, ежедневно, каждую неделю и т. д. Или даже использовать более сложные интервалы, основанные на выражениях





В Airflow используется master/worker архитектура для узлов. На высоком уровне Airflow состоит из трех основных компонентов:

- **Воркеры1 (workers) Airflow** – выбирают задачи, которые запланированы для выполнения, и выполняют их. Таким образом, они несут ответственность за фактическое «выполнение работы»;
- **Metadata Database (Метаданные) —** Это база данных, где Airflow хранит информацию о DAG'ax, расписаниях, истории выполнения, статусах задач и логах. Обычно используется PostgreSQL или MySQL, которые поддерживают высокую производительность и надёжность.
- **Message Broker (Брокер сообщений)** — для CeleryExecutor: Используется только если выбран `CeleryExecutor`. Брокер сообщений (например, Redis или RabbitMQ) отвечает за распределение задач между воркерами и координацию выполнения задач.

▼ Scheduler (Планировщик)

Планировщик Airflow (Shedulder) – сердце приложения, анализирует DAG, проверяет параметр `schedule_interval` и (если все в порядке) начинает планировать задачи DAG для выполнения, передавая их воркерам Airflow

План работы:

1. После того как пользователи написали свои рабочие процессы в виде DAG, файлы, содержащие эти графы, считываются планировщиком для извлечения соответствующих задач, зависимостей и интервалов каждого DAG.
2. После этого для каждого графа планировщик проверяет, все ли в порядке с интервалом с момента последнего чтения. Если да, то задачи в графе планируются к выполнению.
3. Для каждой задачи, запускаемой по расписанию, планировщик затем проверяет, были ли выполнены зависимости (= вышестоящие задачи) задачи. Если да, то задача добавляется в очередь выполнения.
4. Планировщик ждет несколько секунд, прежде чем начать новый цикл, перескакивая обратно к шагу 1

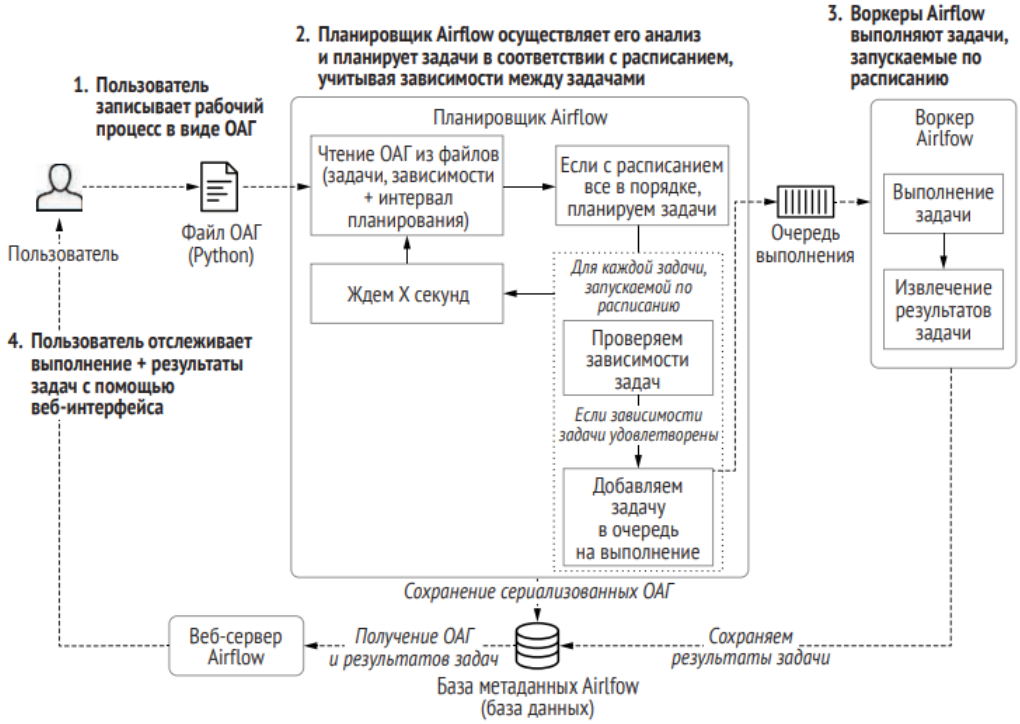
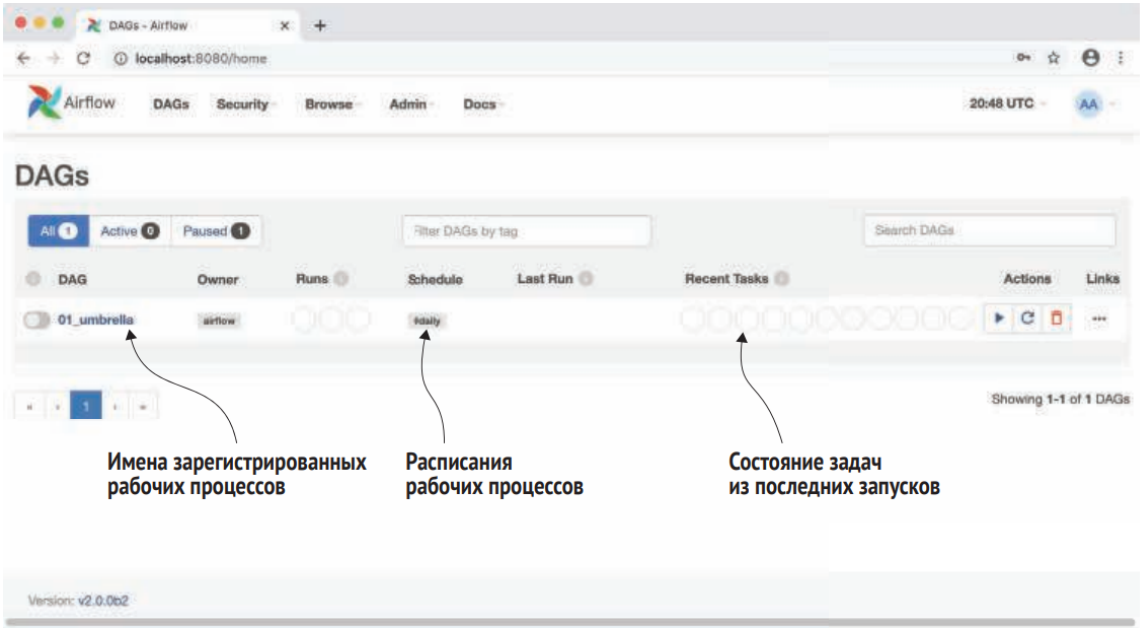


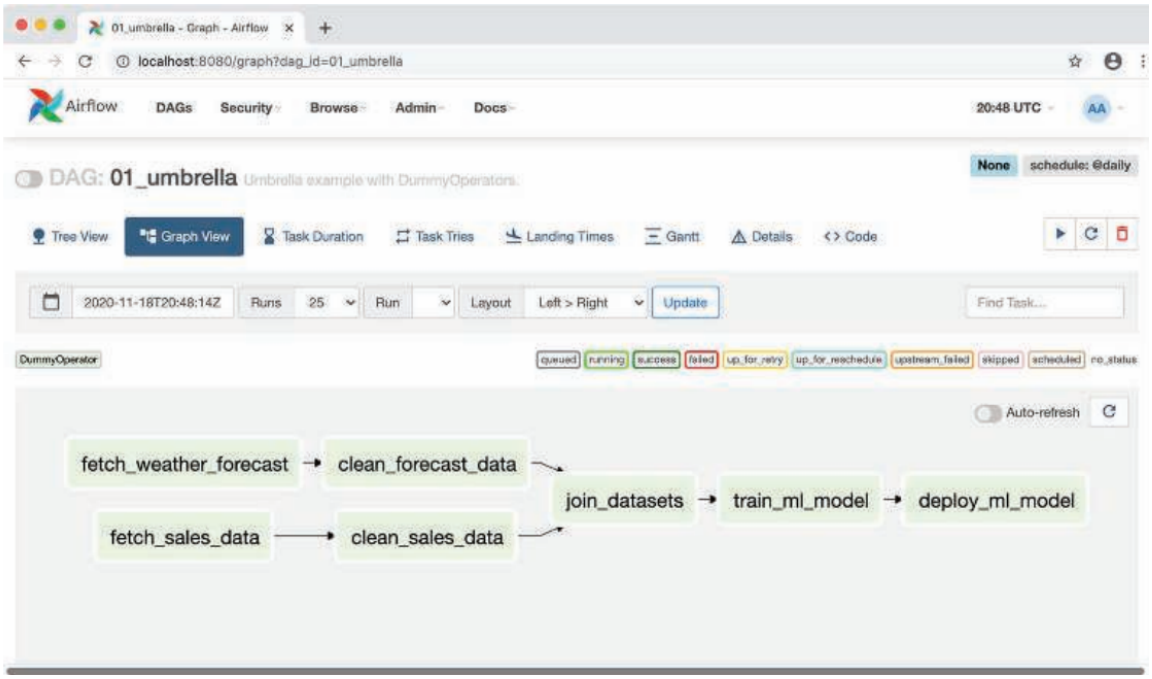
Рис. 1.9 Схематический обзор процесса, участвующего в разработке и выполнении конвейеров в виде ОАГ с использованием Airflow

▼ Веб-сервер Airflow

Веб-сервер Airflow – визуализирует DAG, анализируемые планировщиком, и предоставляет пользователям основной интерфейс для отслеживания выполнения графов и их результатов **(мониторинг)**.



Можно посмотреть список дагов и их состояние



Внутри каждого дага есть дерево из задачек, где подробно планировщик подсказывает, на каком этапе мы находимся

▼ Основные задачи

▼ Запуск по расписанию

`schedule_interval`

Airflow запускает задачи в конце интервала. Если разработка ОАГ ведется 1 января 2019 года в 13:00, с start_date – 01-01-2019 и интервалом @daily, то это означает, что **сначала он запускается в полночь**. Поначалу ничего не произойдет, если вы запустите ОАГ 1 января в 13:00 до полуночи.

- Основной параметр для задания расписания DAG.
- Может принимать значения в формате cron, специальных пресетов или объекта `timedelta` для настройки интервала между запусками.
  - `"@daily"` — запуск каждый день в полночь.
  - `"@hourly"` — запуск каждый час.
  - `"@weekly"` — запуск раз в неделю в полночь по понедельникам.
  - `"@monthly"` — запуск раз в месяц в полночь первого числа.
  - `"@once"` — DAG запускается один раз при первом включении.
  - `timedelta(hours=6)` — запуск каждые 6 часов, `dt.timedelta(days=3)` — запуск каждые три дня.
  - `"0 9 * * *"` — запуск каждый день в 9:00 утра (формат cron).
    - 0 * * * * = ежечасно (запуск по часам);
    - 0 0 * * * = ежедневно (запуск в полночь);
    - 0 0 * * 0 = еженедельно (запуск в полночь в воскресенье).
    - 0 0 1 * * = полночь первого числа каждого месяца;
    - 45 23 * * SAT = 23:45 каждую субботу.

Рис. 3.1 Интервалы для ОАГ, запускаемого по расписанию ежедневно, с заданной датой запуска (2019-01-01). Стрелки указывают момент времени, в который выполняется ОАГ. Если дата окончания не указана, ОАГ будет выполняться каждый день до тех пор, пока не будет отключен

```
dag = DAG(
 dag_id="download_rocket_launches",
 start_date=airflow.utils.dates.days_ago(14),
 schedule_interval="@daily",
)
```

`start_date`

- Устанавливает дату и время, с которых начинается расписание DAG.
- Airflow будет начинать выполнение DAG с этой даты, учитывая заданный `schedule_interval`.
- Важно: `start_date` не должна быть установлена на текущий момент или будущее время, так как Airflow начнёт выполнение DAG только после наступления `start_date`.

`end_date`

- Опциональный параметр, который задаёт конечную дату для запуска DAG.
- После достижения `end_date` DAG перестаёт выполняться по расписанию.

`catchup`

- Булевый параметр (`True` или `False`), указывающий, должен ли DAG "догонять" пропущенные запуски.
- По умолчанию `catchup=True`, то есть, если DAG не выполнялся за прошлые даты, Airflow создаст задачи для всех пропущенных запусков с момента `start_date`.

- Полезен для отключения догоняющего поведения (например, для ежедневных DAG с длинной историей), чтобы запускаться только для текущей даты.

`timezone`

- Устанавливает часовой пояс для запуска DAG.
- По умолчанию используется часовой пояс UTC, но можно указать другие зоны, например, `"Europe/Moscow"`.
- Пример: `timezone="Europe/Moscow"`, чтобы DAG запускался по московскому времени.

Пример настройки DAG с расписанием:

```
from datetime import datetime, timedelta
from airflow import DAG

with DAG(
    dag_id="example_dag",
    start_date=datetime(2023, 1, 1),
    end_date=datetime(2023, 12, 31),
    schedule_interval="@daily",
    catchup=False,
    timezone="Europe/Moscow"
) as dag:
    ..
```