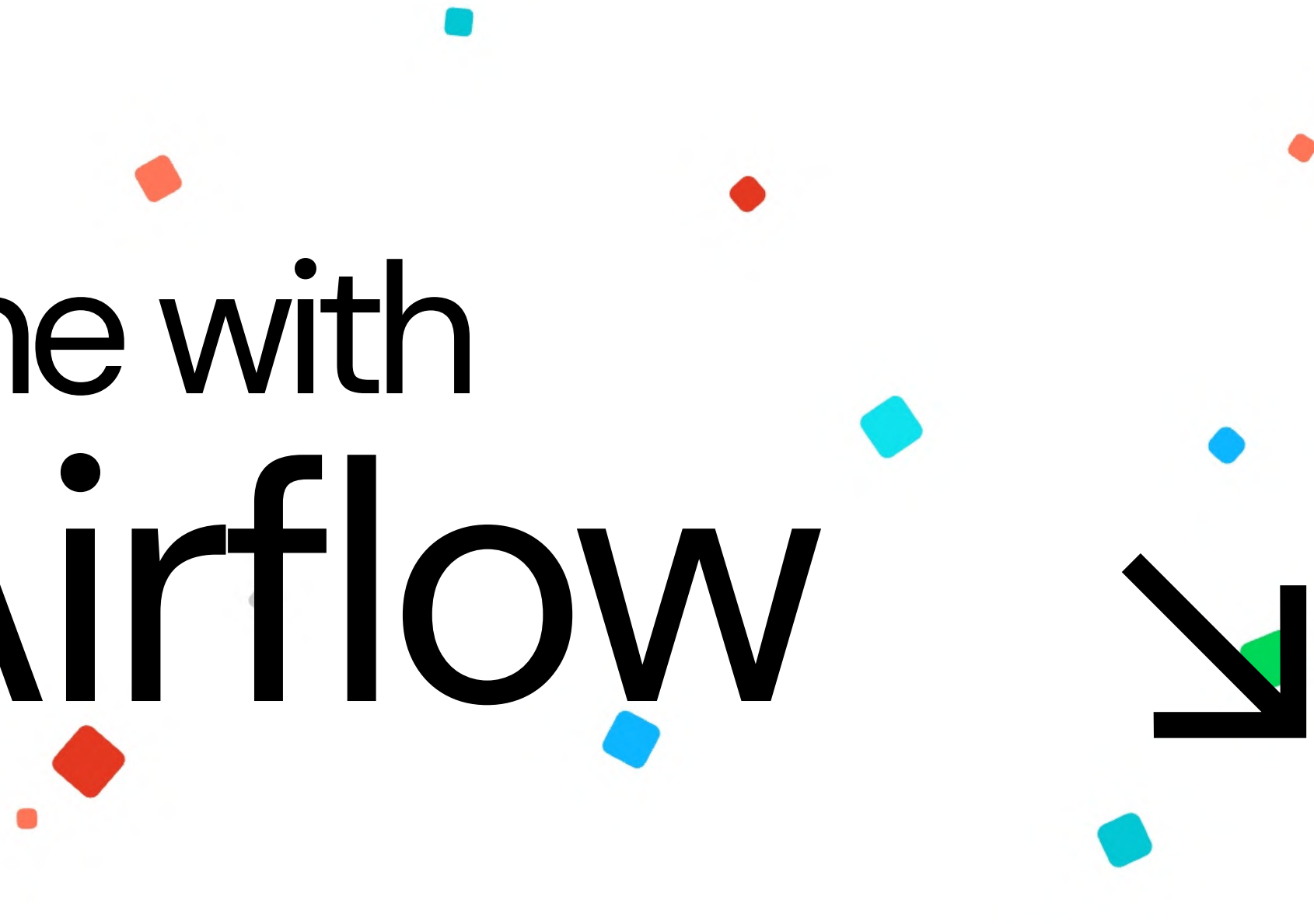
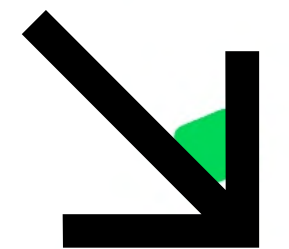


Manik Rana
@ManaMkr

PyDelhi
Jan 2024

Build an ETL pipeline with Apache Airflow

A collection of small, semi-transparent diamond-shaped confetti in red, orange, and cyan colors is scattered across the right side of the slide. A large, bold black arrow points downwards and to the right, with a small green diamond at its tip.

Agenda



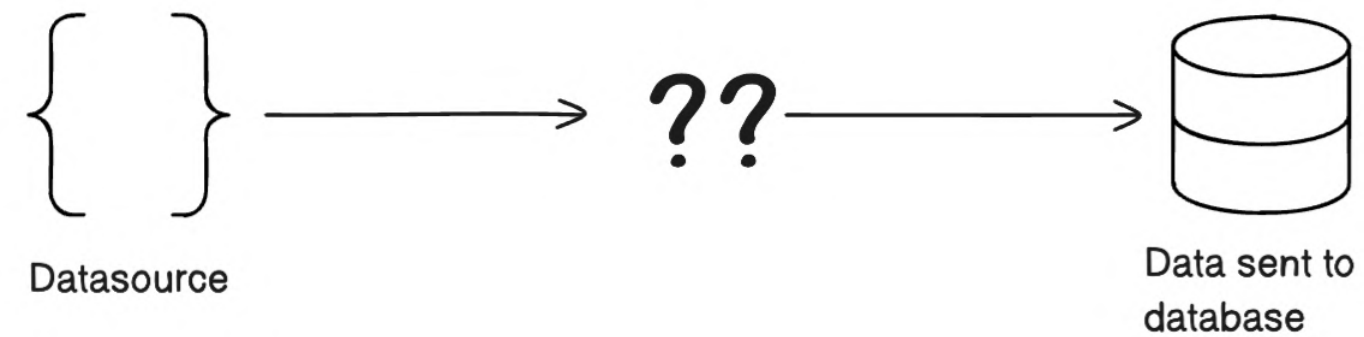
Data Engineering

1. What is Data Engineering
2. Key Concepts

Airflow

1. A quick tour of Airflow
2. DAGs
3. More Airflow concepts
4. Building an ETL pipeline

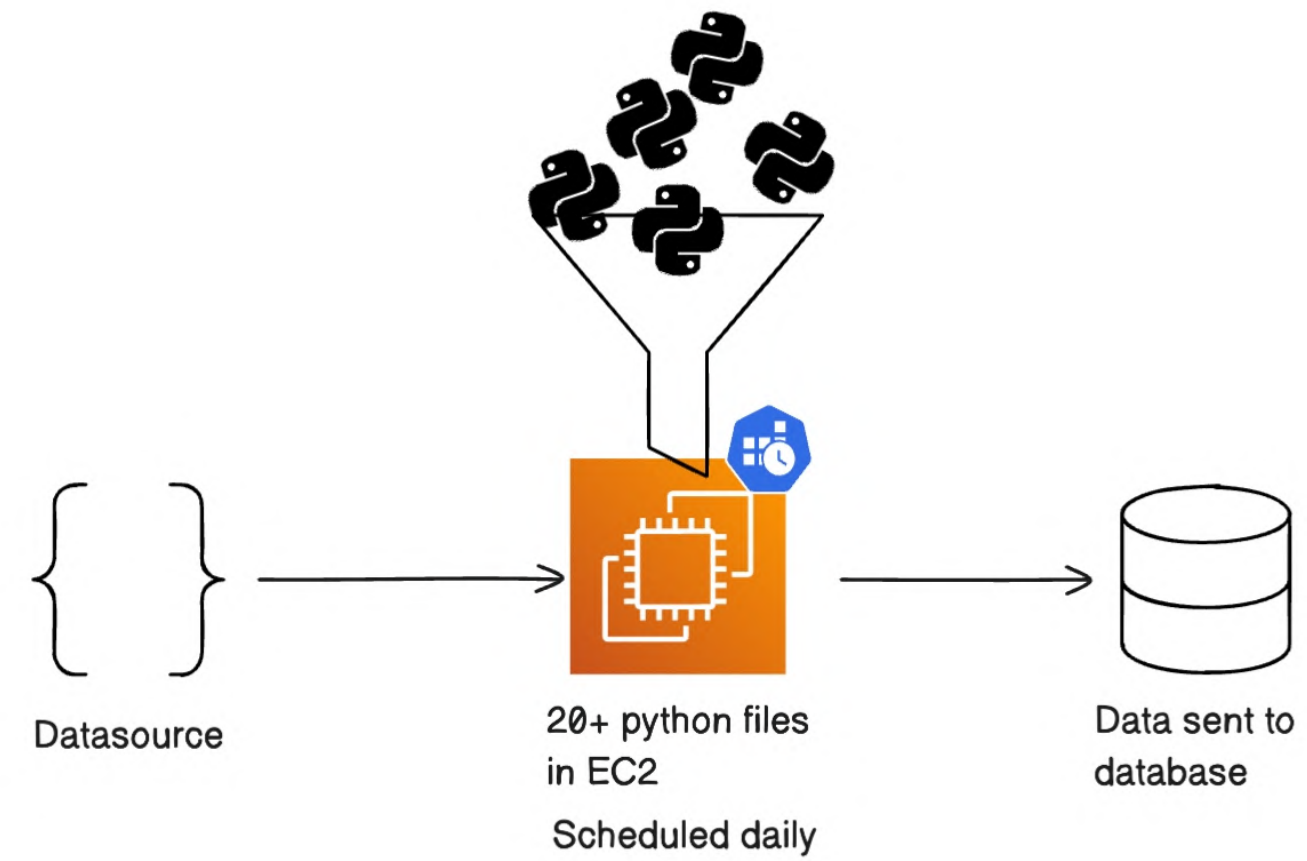
A data collection service at Intern co.



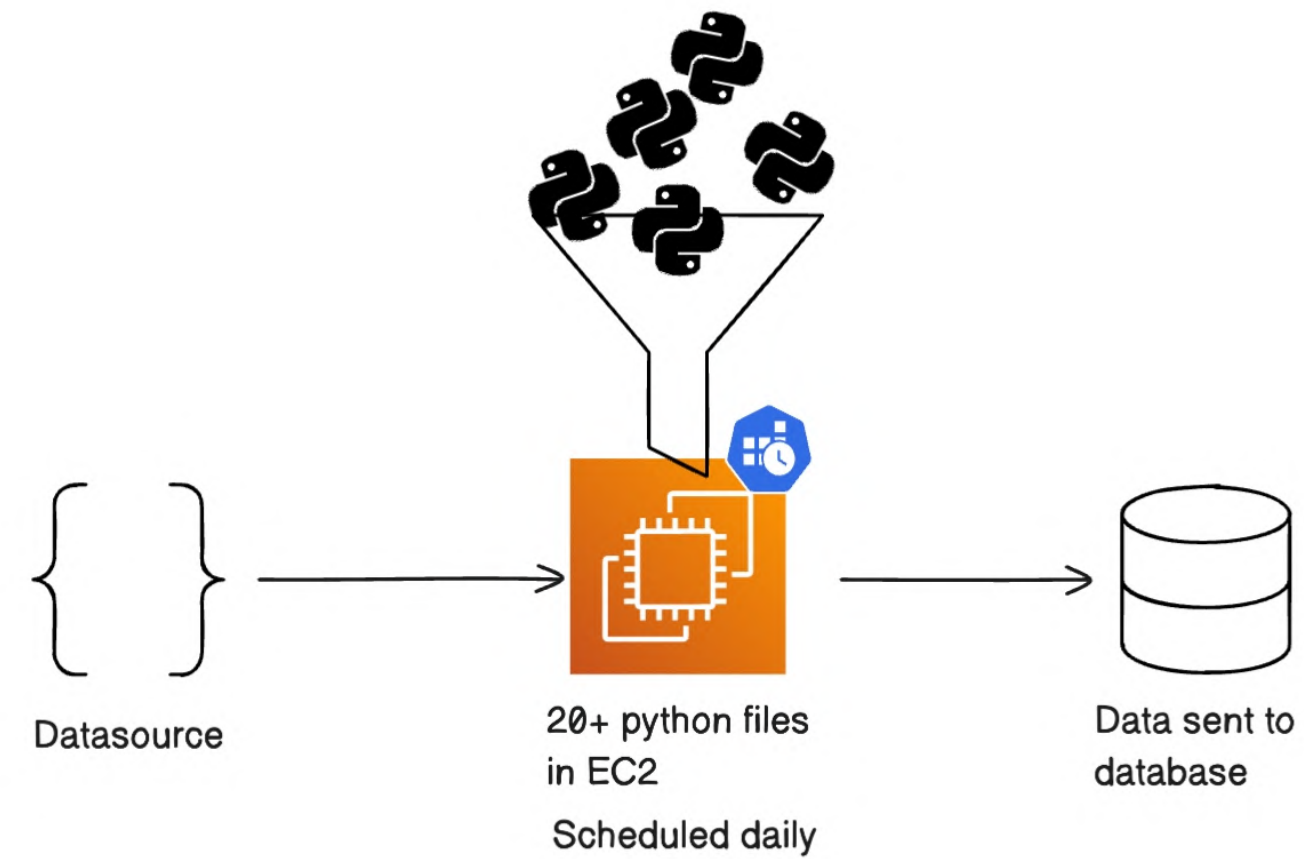
Challenge:

- Need to setup a system to collect, clean, normalize and store data from multiple datasources

A data collection service at Intern co.



A data collection service at Intern co.



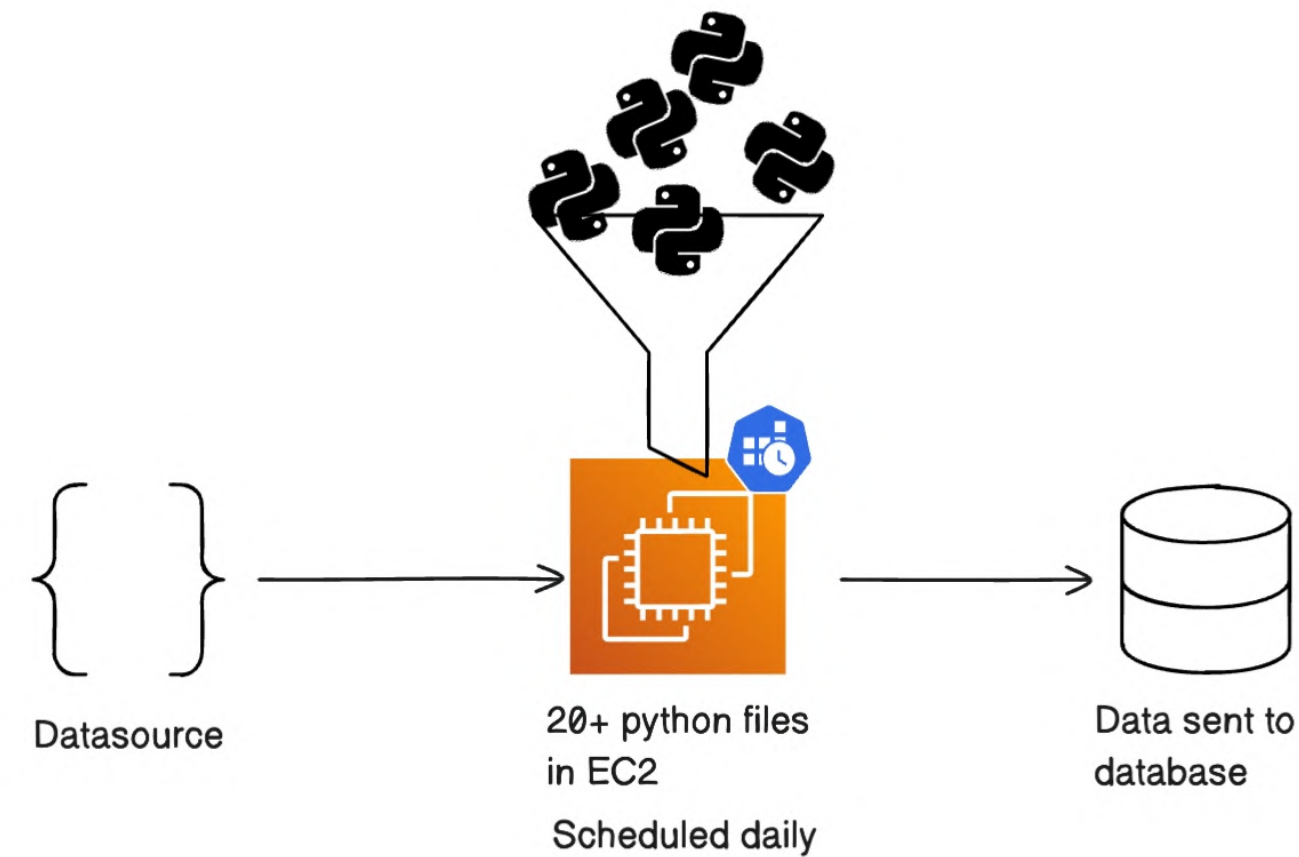
Pros:

- Simple setup

Cons:

- Need to add logging
- Don't know when or where failure happens
- Gets complex fast

A data collection service at Intern co.



Pros:

- Simple setup

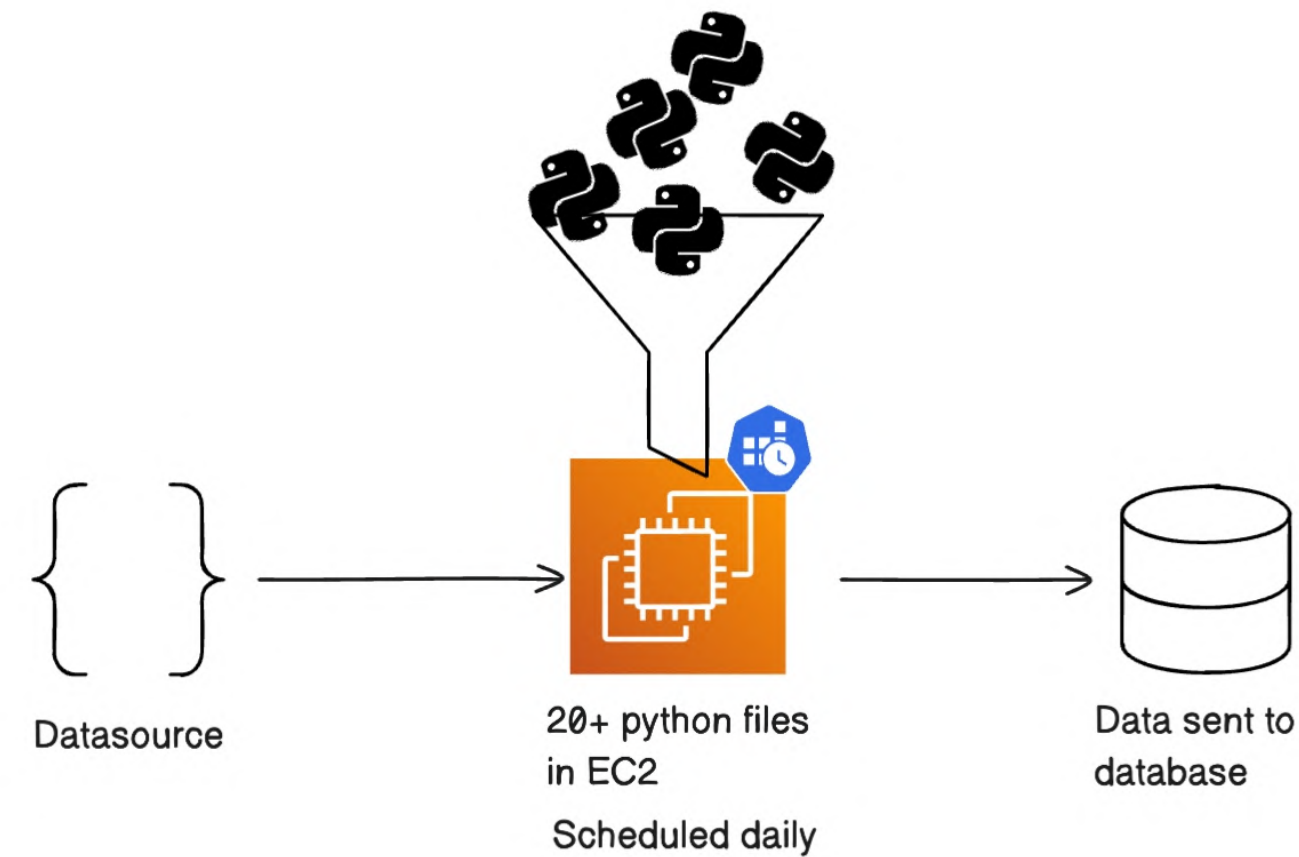
Cons:

- Need to add logging
- Don't know when or where failure happens
- Gets complex fast

Where we failed:

- Terrible data quality in DB and we didn't know why
- ML and frontend team couldn't realistically use collected data
- DB wasn't getting data for 3 days and nobody noticed

A data collection service at Intern co.



Pros:

- Simple setup

Cons:

- Need to add logging
- Don't know when or where failure happens
- Gets complex fast

Where we failed:

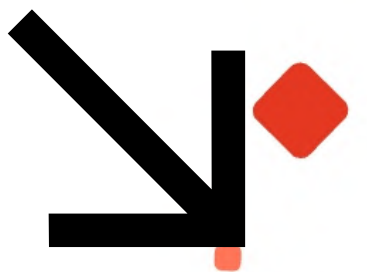
- Terrible data quality in DB and we didn't know why
- ML and frontend team couldn't realistically use collected data
- DB wasn't getting data for 3 days and nobody noticed

We needed a better way to collect and store data

Manik Rana
@ManaMkr

PyDelhi
Jan 2024

Data Engineering

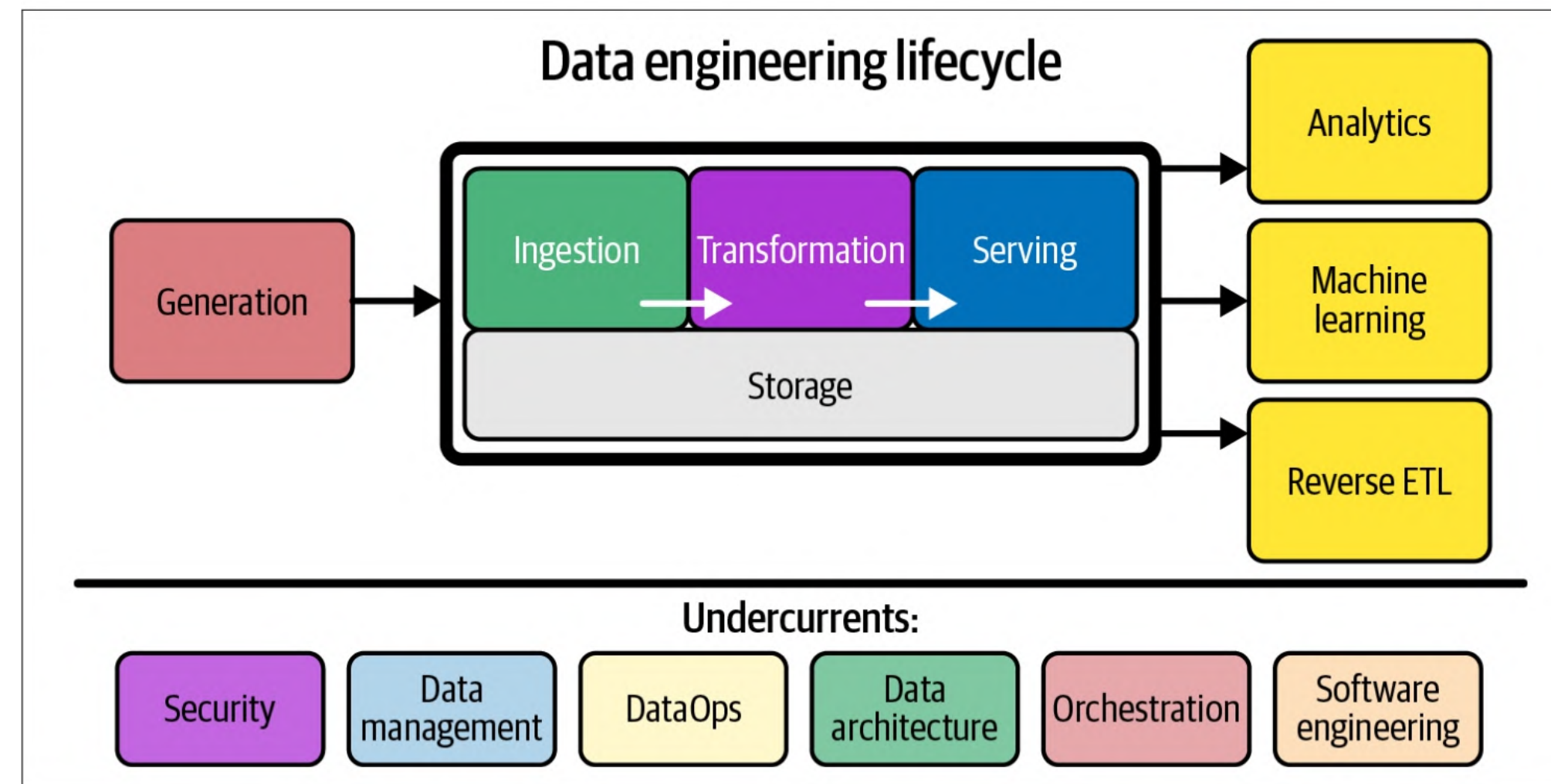


Data Engineering

Data engineering is the **development, implementation, and maintenance** of systems and processes that **take in raw data** and **produce high-quality, consistent information** that **supports downstream use cases**, such as analysis and machine learning.

Data engineering is the intersection of security, data management, DataOps, data architecture, orchestration, and software engineering.

A *data engineer* manages the data engineering lifecycle, **beginning with getting data from source** systems and **ending with serving data** for use cases, such as analysis or machine learning.



<https://www.oreilly.com/library/view/fundamentals-of-data/9781098108298/ch02.html>

Although many data scientists are eager to build and tune ML models, the reality is an estimated 70% to 80% of their time is spent toiling in the bottom three parts of the hierarchy—**gathering data, cleaning data, processing data**—and only a tiny slice of their time on analysis and ML.

THE DATA SCIENCE HIERARCHY OF NEEDS

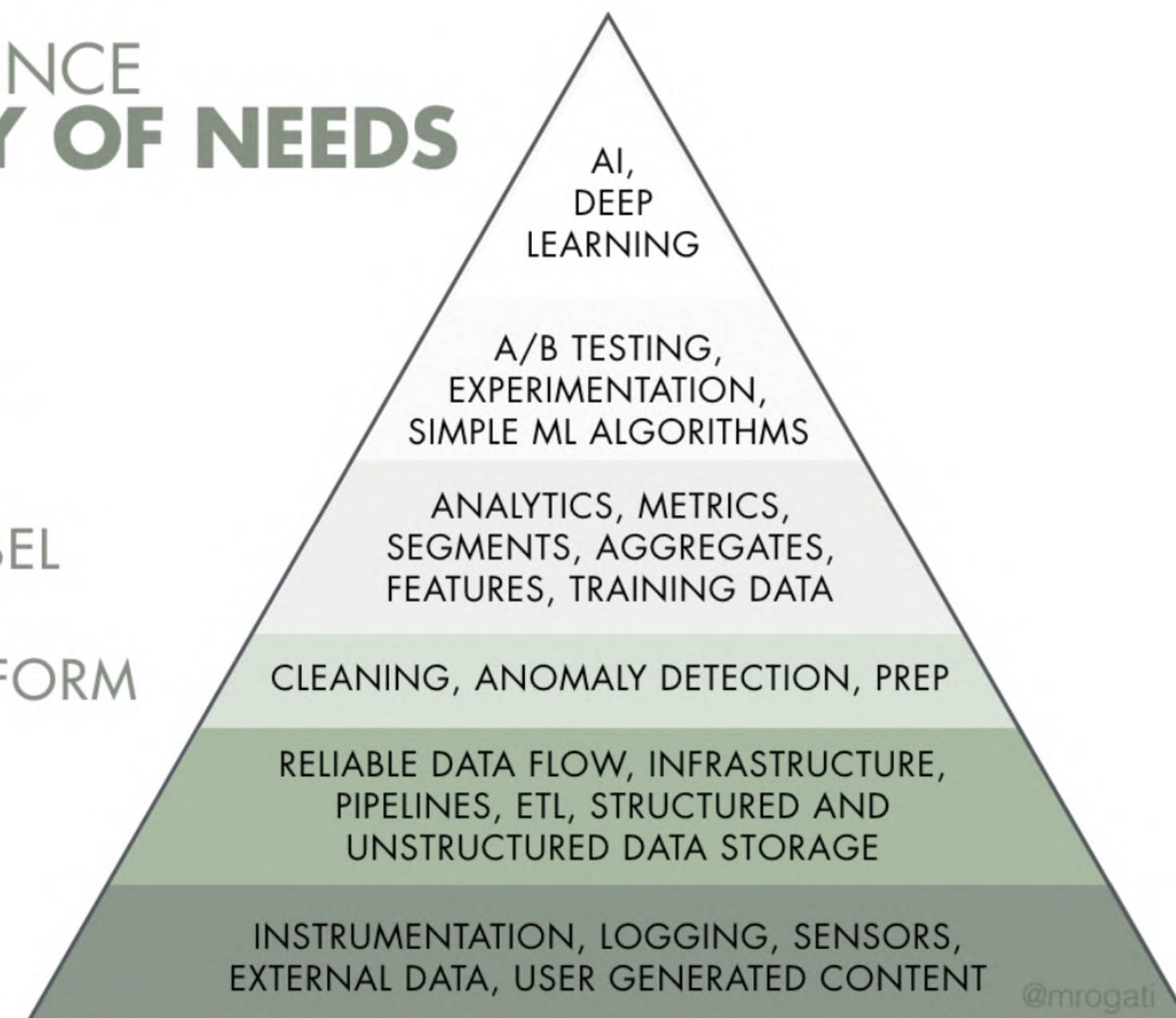
LEARN/OPTIMIZE

AGGREGATE/LABEL

EXPLORE/TRANSFORM

MOVE/STORE

COLLECT



<https://oreil.ly/pGg9U>

In an ideal world, data scientists spend more than 90% of their time focused on the top layers of the pyramid

When data engineers focus on these bottom parts of the hierarchy, they build a solid foundation for data scientists to succeed.

THE DATA SCIENCE HIERARCHY OF NEEDS

LEARN/OPTIMIZE

AGGREGATE/LABEL

EXPLORE/TRANSFORM

MOVE/STORE

COLLECT

AI,
DEEP
LEARNING

A/B TESTING,
EXPERIMENTATION,
SIMPLE ML ALGORITHMS

ANALYTICS, METRICS,
SEGMENTS, AGGREGATES,
FEATURES, TRAINING DATA

CLEANING, ANOMALY DETECTION, PREP

RELIABLE DATA FLOW, INFRASTRUCTURE,
PIPELINES, ETL, STRUCTURED AND
UNSTRUCTURED DATA STORAGE

INSTRUMENTATION, LOGGING, SENSORS,
EXTERNAL DATA, USER GENERATED CONTENT

@mrogati

<https://oreil.ly/pGg9U>

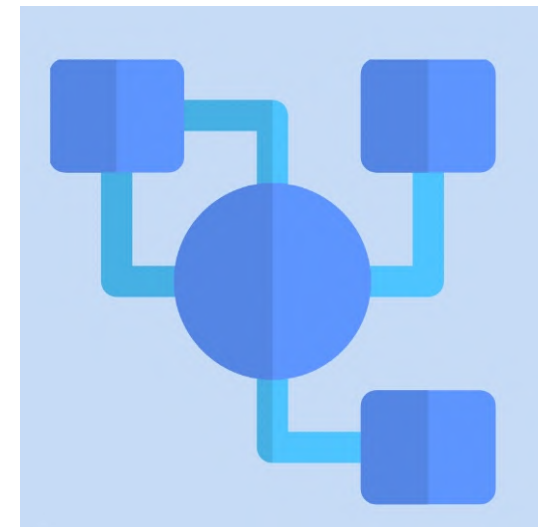
Data Engineering:

Key Concepts

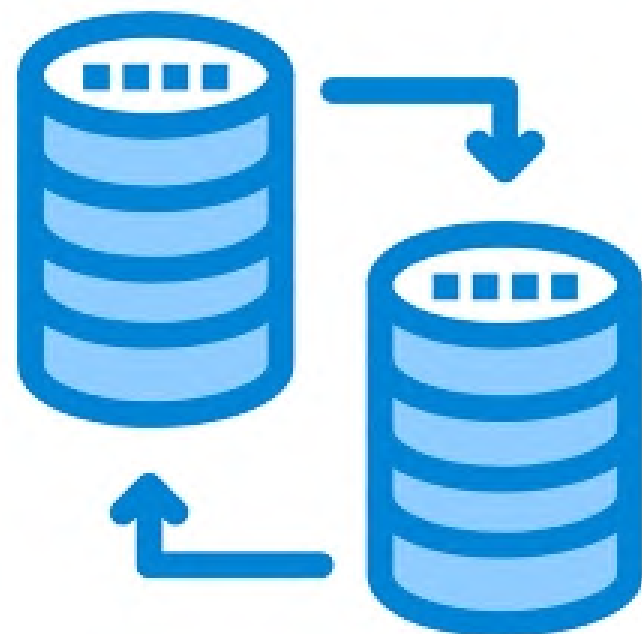
Ingestion



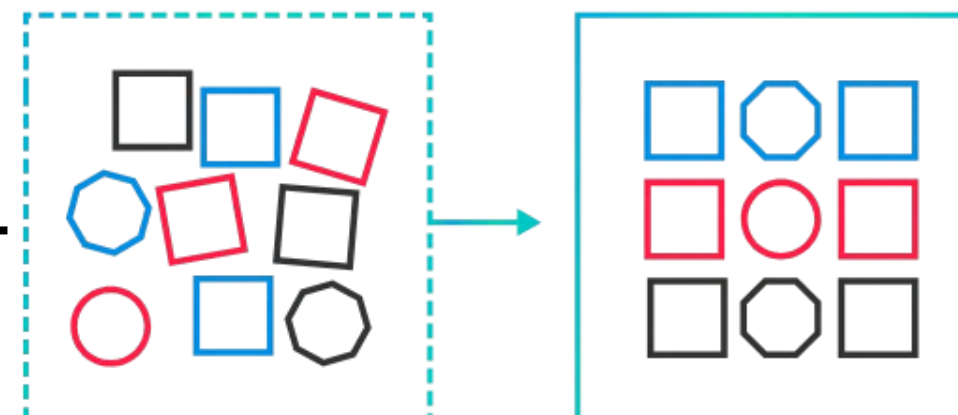
Processing



Storage



Transformation



Data Engineering:

Key Concepts

Ingestion

Collect and bring in raw data from various sources.

It occurs either in real-time or batches.

Processing

Perform initial operations on the raw data to make it more manageable.

Tools: Apache Spark, Apache Flink, or Hadoop MapReduce

Transformation

Convert, cleanse, and structure the processed data for analysis.

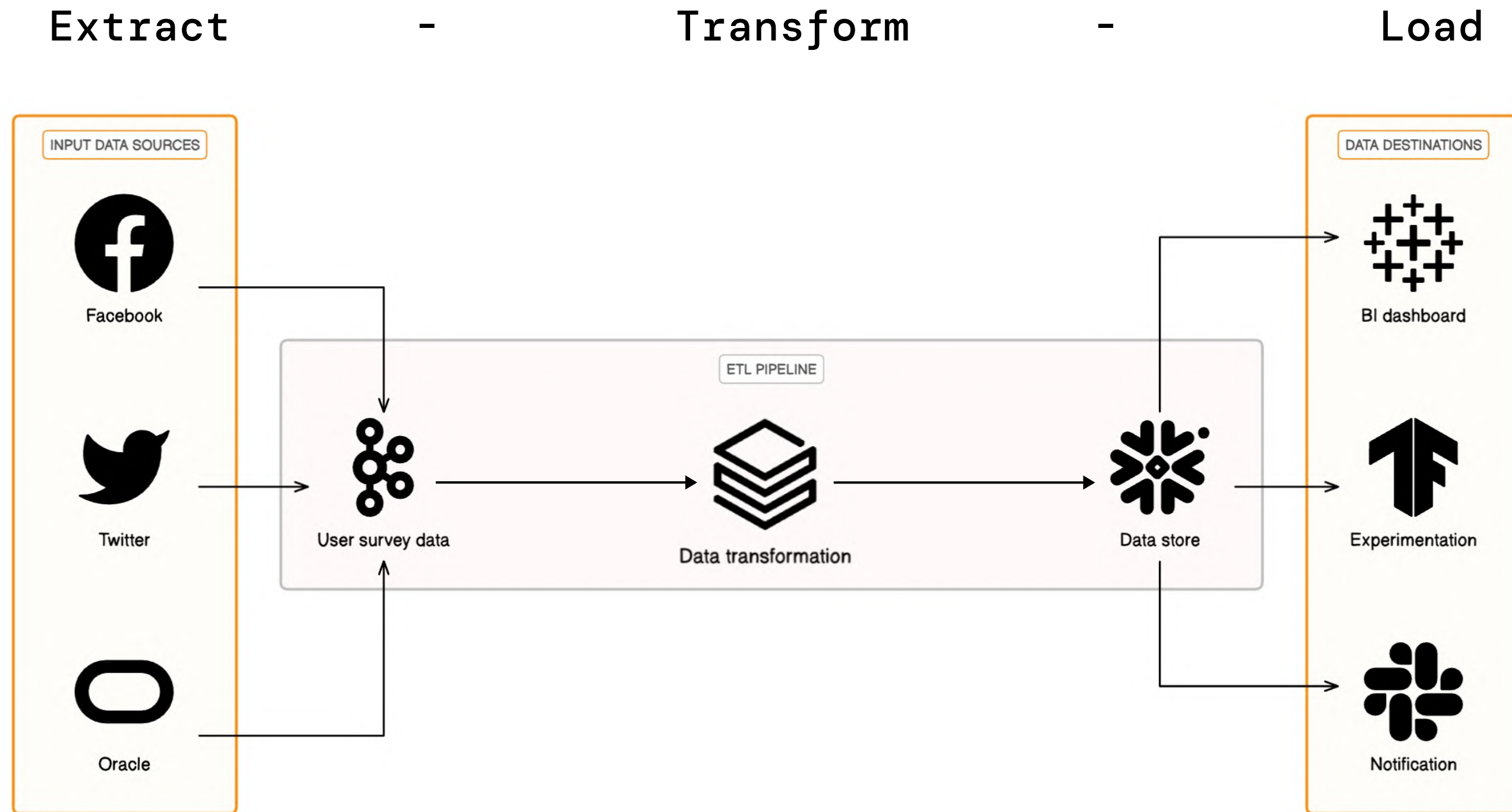
Storage

The recording of information in a storage medium.

Ex. databases, warehouses, cloud storage, and distributed systems

ETL/ELT Pipelines

These involve moving data from source systems to a target system for analysis, reporting, and decision-making.

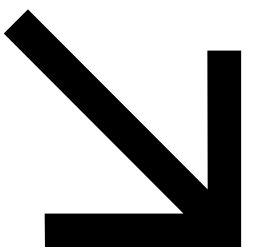


In ETL, the transformation occurs in a separate staging area before loading data into the target system. In ELT, the transformation happens directly within the target data warehouse.

Manik Rana
@ManaMkr

PyDelhi
Jan 2024

Apache Airflow



Airflow:

What is it?

Apache Airflow is an open-source platform to control, monitor, and schedule data engineering pipelines.

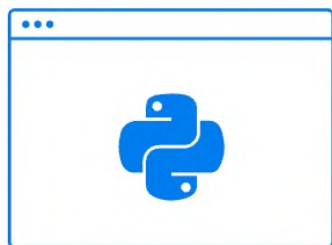
Pipelines are configured as code, allowing for dynamic pipeline generation.

Created at Airbnb as an open-source project in 2014, Airflow was brought into the Apache Software Incubator Program in 2016 and announced as a Top-Level Apache Project in 2019.



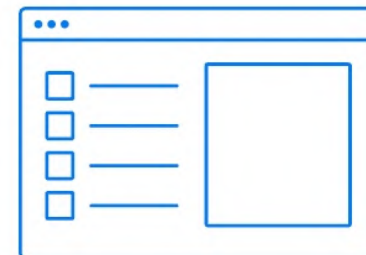
Airflow:

Why Airflow?



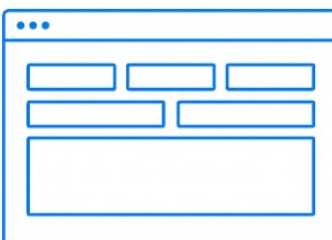
Pure Python

No more command-line or XML black-magic! Use standard Python features to create your workflows, including date time formats for scheduling and loops to dynamically generate tasks. This allows you to maintain full flexibility when building your workflows.



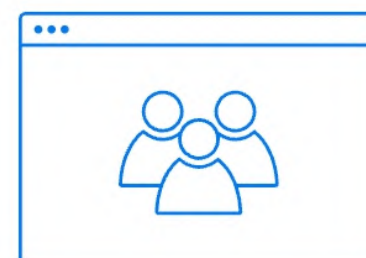
Easy to Use

Anyone with Python knowledge can deploy a workflow. Apache Airflow™ does not limit the scope of your pipelines; you can use it to build ML models, transfer data, manage your infrastructure, and more.



Useful UI

Monitor, schedule and manage your workflows via a robust and modern web application. No need to learn old, cron-like interfaces. You always have full insight into the status and logs of completed and ongoing tasks.



Open Source

Wherever you want to share your improvement you can do this by opening a PR. It's simple as that, no barriers, no prolonged procedures. Airflow has many active users who willingly share their experiences. Have any questions? Check out our buzzing slack.

Airflow:

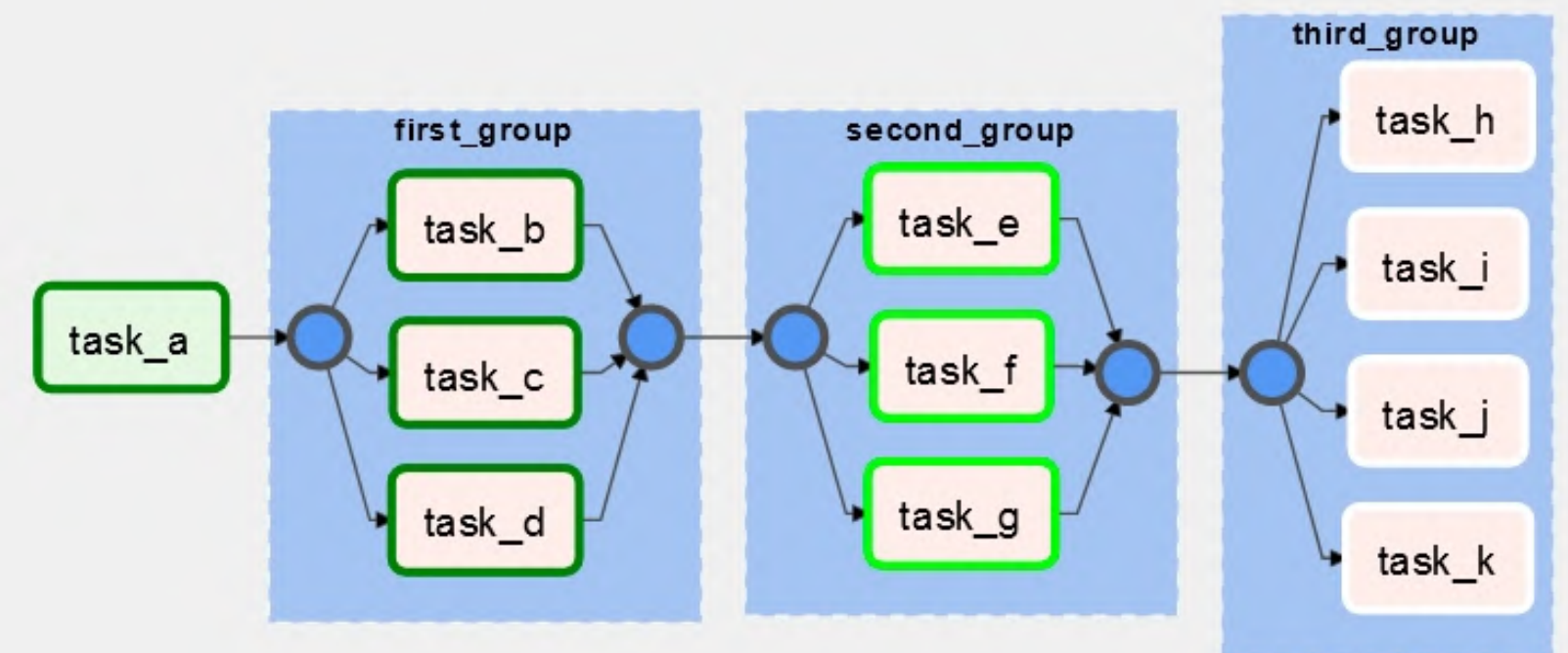
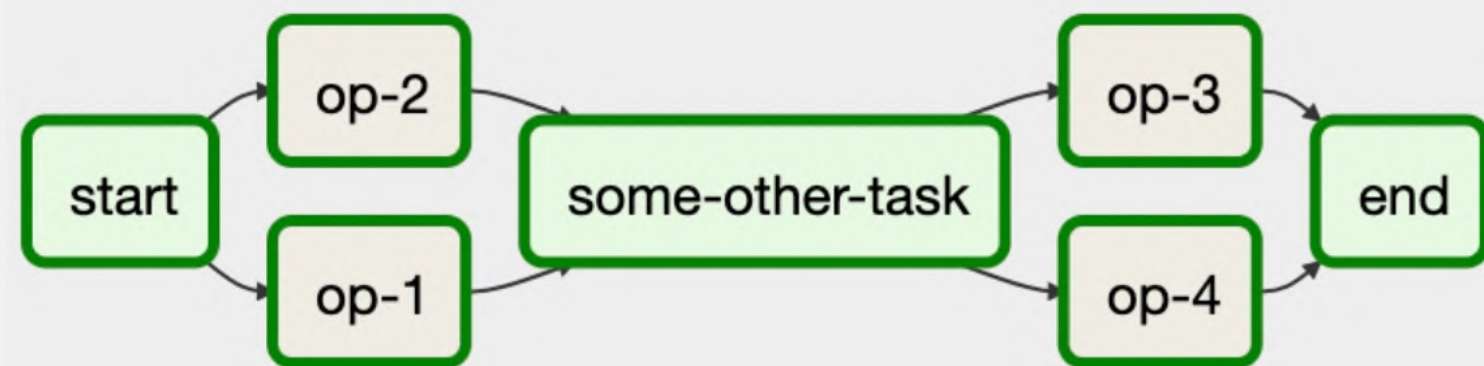
Directed Acyclic Graphs (DAGs)

DAGs are a fundamental concept in Airflow.

They allow users to express the sequence of tasks and the dependencies between them in a structured way.

Each node in the DAG represents a task—a unit of work that needs to be executed.

Dependencies define the order in which tasks should be executed.



Airflow:

Some more key concepts

Operators

Operators define the atomic, indivisible units of work within a task. Each task in a DAG is associated with an operator that specifies what work should be done.

Ex: BashOperator, PythonOperator

Sensors

Sensors are a type of operator that waits for a certain condition to be met before allowing the workflow to proceed.

Good for when waiting on an external event or condition (like checking for changes in a DB)

Hooks

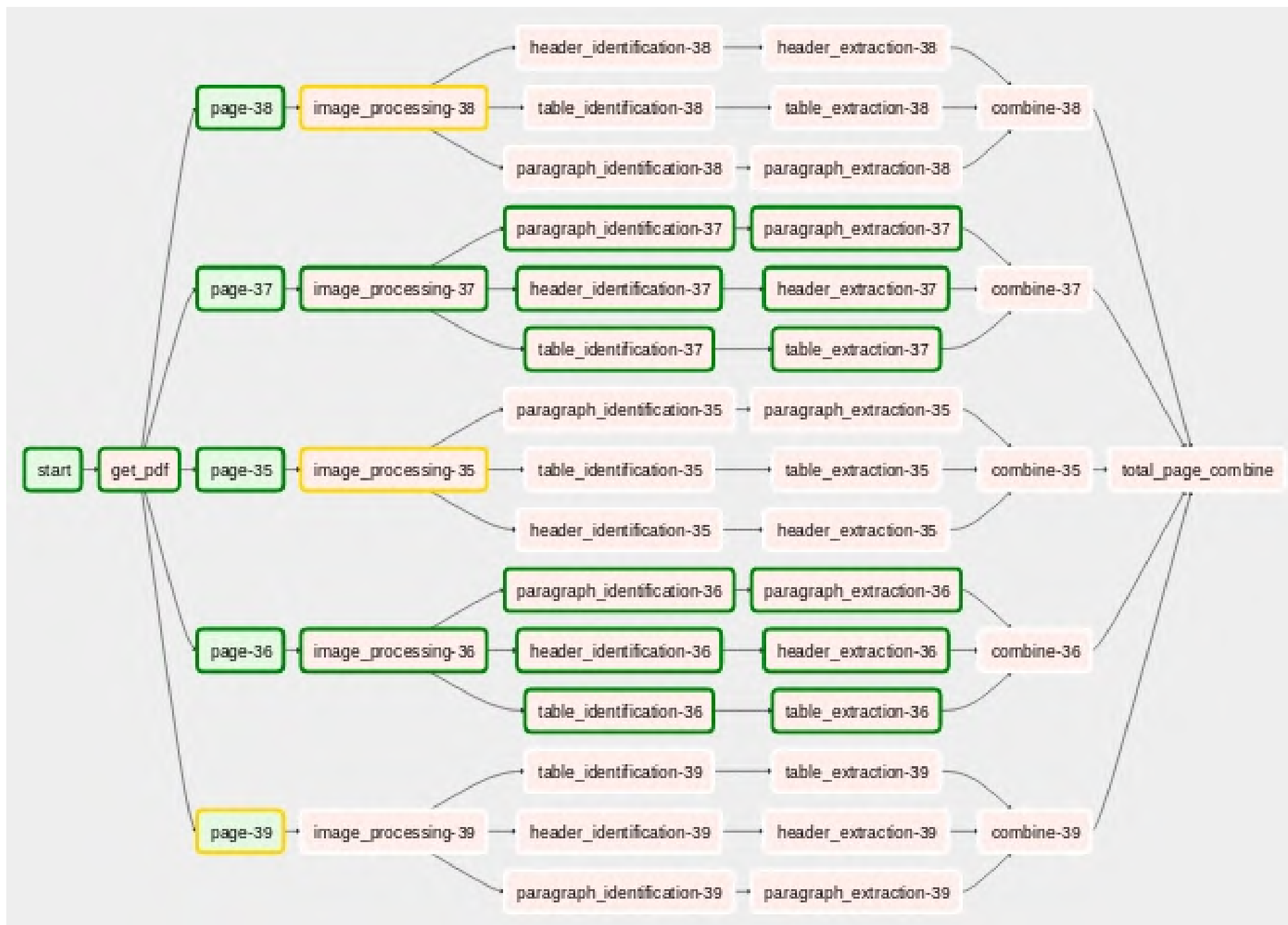
Hooks are a way to interact with external systems or services from within tasks. They provide a Pythonic interface to connect and communicate with databases, APIs, and other external resources.

XCom (Cross-Communication)

Com is a mechanism for tasks to exchange small amounts of data between them during runtime. It allows tasks to communicate and share information within the context of a single DAG run.

Airflow:

DAGs can get crazy





Sign In

Enter your login and password below:

Username:

👤

admin

Password:

🔑

.....|

Sign In

DAGs

All 26
Active 10
Paused 16

Filter DAGs by tag

Search DAGs

i DAG	Owner	Runs i	Schedule	Last Run i	Recent Tasks i	Actions	Links
☒ example_bash_operator example example2	airflow	2	00***	2020-10-26, 21:08:11 i	6	▶ ↺ 🗑	...
☒ example_branch_dop_operator_v3 example	airflow		*/1****			▶ ↺ 🗑	...
☐ example_branch_operator example example2	airflow	1	@daily	2020-10-23, 14:09:17 i	11	▶ ↺ 🗑	...
☒ example_complex example example2 example3	airflow	1 1	None	2020-10-26, 21:08:04 i	37 37	▶ ↺ 🗑	...
☒ example_external_task_marker_child	airflow	1	None	2020-10-26, 21:07:33 i	2	▶ ↺ 🗑	...
☒ example_external_task_marker_parent	airflow	1	None	2020-10-26, 21:08:34 i	1	▶ ↺ 🗑	...
☒ example_kubernetes_executor example example2	airflow		None			▶ ↺ 🗑	...
☒ example_kubernetes_executor_config example3	airflow	1	None	2020-10-26, 21:07:40 i	5	▶ ↺ 🗑	...
☒ example_nested_branch_dag example	airflow	1	@daily	2020-10-26, 21:07:37 i	9	▶ ↺ 🗑	...
☐ example_passing_params_via_test_command example	airflow		*/1****			▶ ↺ 🗑	...

 DAG: example_bash_operator

failed

schedule: 0 0 * * *

 Tree View

 Graph View

 Task Duration

 Task Tries

 Landing Times

 Gantt

 Details

 Code







 2021-03-30T19:27:23Z

Runs 25

Run manual__2021-03-30T19:27:22.155000+00:00

Layout Left > Right

Update

Find Task...

BashOperator

DummyOperator

queued

running

success

failed

up_for_retry

up_for_reschedule

upstream_failed

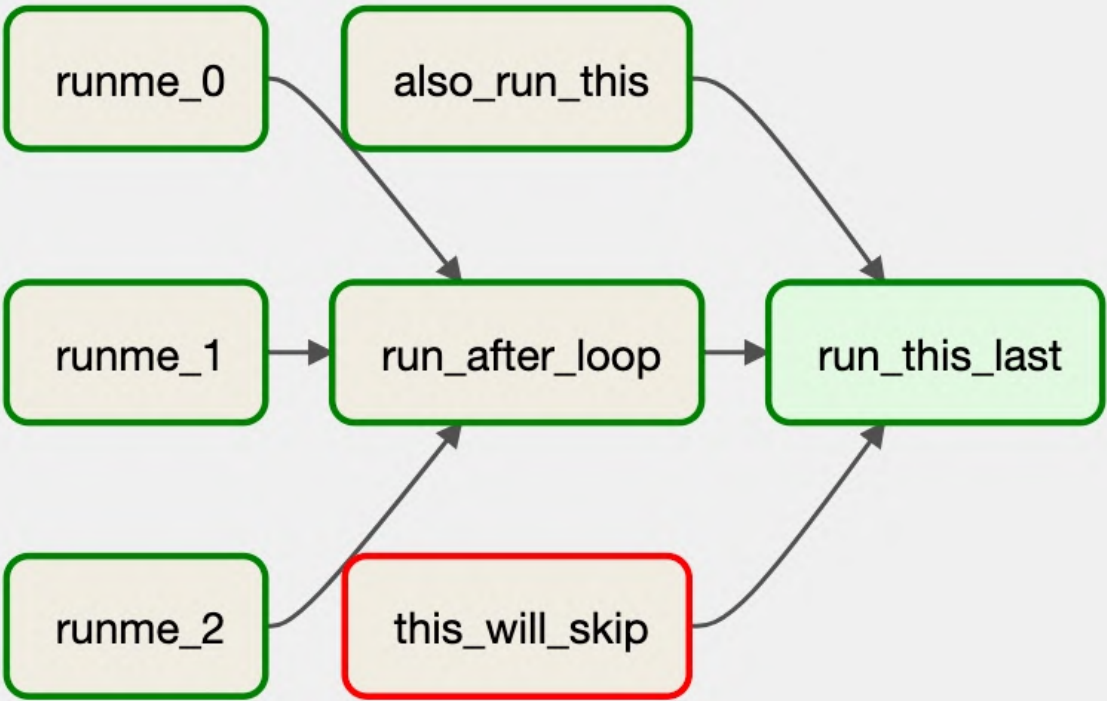
skipped

scheduled

no_status

☐ Auto-refresh





 DAG: example_bash_operator

Schedule: 0 0 * * * ⓘ

Next Run: 2023-05-27, 00:00:00

Grid

Graph

Calendar

Task Duration

Task Tries

Landing Times

Gantt

Details

<> Code

Audit Log





27-05-2023 08:13:44 ⓘ

25 ▾

All Run Types ▾

All Run States ▾

Clear Filters

Auto-refresh ☐

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

« » DAG example_bash_operator

Duration

00:00:06

00:00:03

00:00:00

runme_0

runme_1

runme_2

also_run_this

this_will_skip

run_after_loop

run_this_last

Details

Graph

<> Code

More Details

DAG Runs Summary

Total Runs Displayed

3

Total success

3

First Run Start

2023-05-27, 07:53:57 UTC

Last Run Start

2023-05-27, 07:54:11 UTC

Max Run Duration

00:00:06

Mean Run Duration

00:00:05

Min Run Duration

00:00:05

DAG Summary

Total Tasks

7

BashOperators

6

EmptyOperator

1

A bar chart and grid representation of the DAG that spans across time. The top row is a chart of DAG Runs by duration, and below, task instances. If a pipeline is late, you can quickly see where the different steps are and identify the blocking ones.

DAG: example_bash_operator

Schedule: 0 0 * * *

Next Run: 2023-07-28, 00:00:00

Grid

Graph

Calendar

Task Duration

Task Tries

Landing Times

Gantt

Details

<> Code

Audit Log

07/28/2023, 07:15:58 AM

25

All Run Types

All Run States

Clear Filters

Auto-refresh

Press **shift** + **/** for Shortcuts

deferred

failed

queued

removed

restarting

running

scheduled

shutdown

skipped

success

up_for_reschedule

up_for_retry

upstream_failed

no_status



DAG

example_bash_operator / ▶ 2023-07-28, 00:00:00 UTC

Clear

Mark state as...

Details

Graph

Gantt

<> Code

Layout:

Left -> Right

runme_0

runme_1

runme_2

also_run_this

this_will_skip

run_after_loop

run_this_last

runme_2

success

BashOperator

this_will_skip

skipped

BashOperator

runme_1

success

BashOperator

run_after_loop

success

BashOperator

runme_0

success

BashOperator

also_run_this

success

BashOperator

run_this_last

skipped

EmptyOperator

 DAG: example_task_group

Schedule: 1 day, 0:00:00

Next Run: 2023-07-27, 06:36:59

 Grid

 Graph

 Calendar

 Task Duration

 Task Tries

 Landing Times

 Gantt

 Details

 Code

 Audit Log



07/28/2023, 06:09:10 AM

25

All Run Types

All Run States

Clear Filters

Auto-refresh

Press **shift** + **/** for Shortcuts

deferred

failed

queued

removed

restarting

running

scheduled

shutdown

skipped

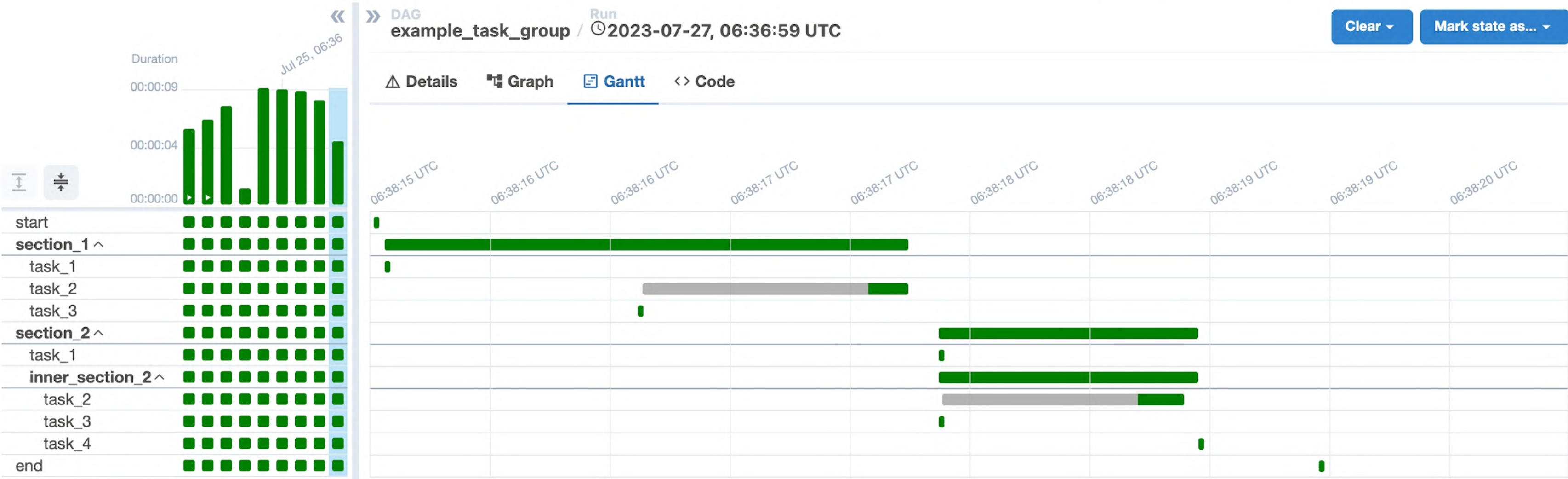
success

up_for_reschedule

up_for_retry

upstream_failed

no_status



DAG: example_bash_operator

Schedule: 0 0 * * *

Next Run: 2023-08-02, 00:00:00

Grid

Graph

Calendar

Task Duration

Task Tries

Landing Times

Gantt

Details

<> Code

Audit Log

08/03/2023, 04:50:30 AM

25

All Run Types

All Run States

Clear Filters

Auto-refresh

Press **shift** + **/** for Shortcuts

deferred

failed

queued

removed

restarting

running

scheduled

shutdown

skipped

success

up_for_reschedule

up_for_retry

upstream_failed

no_status

<< >> DAG

example_bash_operator

Details

Graph

Gantt

<> Code

Parsed at: 2023-08-03, 04:50:00 UTC

```
18 """Example DAG demonstrating the usage of the BashOperator."""
19 from __future__ import annotations
20
21 import datetime
22
23 import pendulum
24
25 from airflow import DAG
26 from airflow.operators.bash import BashOperator
27 from airflow.operators.empty import EmptyOperator
28
29 with DAG(
30     dag_id="example_bash_operator",
31     schedule="0 0 * * *",
32     start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
33     catchup=False,
34     dagrun_timeout=datetime.timedelta(minutes=60),
35     tags=["example", "example2"],
36     params={"example_key": "example_value"},
37 ) as dag:
38     run_this_last = EmptyOperator(
39         task_id="run_this_last",
40     )
```

Toggle Wrap

Logs View

Airflow generates task-specific logs for each execution, documenting key information such as start times, durations, and encountered errors.

 Airflow

DAGs

Browse ▾

Docs ▾

About ▾

2020-03-08 05:39:33 UTC



On

DAG: sync_github_dags

schedule: * * * * *

 Graph View

 Tree View

 Task Duration

 Task Tries

 Landing Times

 Gantt

 Details

 Code

 Trigger DAG

 Refresh

 Delete

Task Instance: git_pull

2020-03-08 05:35:00

 Task Instance Details

 Rendered Template

 Log

 XCom

Log by attempts

1

*** Reading local file: 2020-03-08T05:35:00+00:00/1.log

[2020-03-08 12:36:04,391] {taskinstance.py:615} INFO - Dependencies all met for <TaskInstance: sync_github_dags.git_pull 2020-03-08T05:35:00+00:00 [queued]>

[2020-03-08 12:36:04,398] {taskinstance.py:615} INFO - Dependencies all met for <TaskInstance: sync_github_dags.git_pull 2020-03-08T05:35:00+00:00 [queued]>

[2020-03-08 12:36:04,398] {taskinstance.py:833} INFO -

[2020-03-08 12:36:04,398] {taskinstance.py:834} INFO - Starting attempt 1 of 1

[2020-03-08 12:36:04,398] {taskinstance.py:835} INFO -

[2020-03-08 12:36:04,411] {taskinstance.py:854} INFO - Executing <Task(BashOperator): git_pull> on 2020-03-08T05:35:00+00:00

[2020-03-08 12:36:04,411] {base_task_runner.py:133} INFO - Running: ['airflow', 'run', 'sync_github_dags', 'git_pull', '2020-03-08T05:35:00+00:00', '--job_id', '1', '--raw', '-sd', 'DAG_FOLDER/sync_github_dags']

[2020-03-08 12:36:05,765] {base_task_runner.py:115} INFO - Job 53135: Subtask git_pull [2020-03-08 12:36:05,764] {settings.py:213} INFO - settings.configure_orm(): Using pool settings. pool_size=5, max_overflow=5

[2020-03-08 12:36:05,808] {base_task_runner.py:115} INFO - Job 53135: Subtask git_pull [2020-03-08 12:36:05,807] {__init__.py:51} INFO - Using executor LocalExecutor

Connections Screen

used for configuring and managing external connections to various data sources, APIs, databases, and services, allowing tasks within Airflow DAGs to interact with these external systems.



Airflow

DAGs

Security

Browse

Admin

Docs

List Connection

Search



Actions



Conn Id

Conn Type

Host



airflow_db

mysql

mysql



aws_default

aws

XComs Screen

Data Exchange: The XCom screen in Airflow allows tasks within a DAG to share small amounts of data, enabling communication between different parts of a workflow.

 Airflow

DAGs

Security

Browse

Admin

Docs

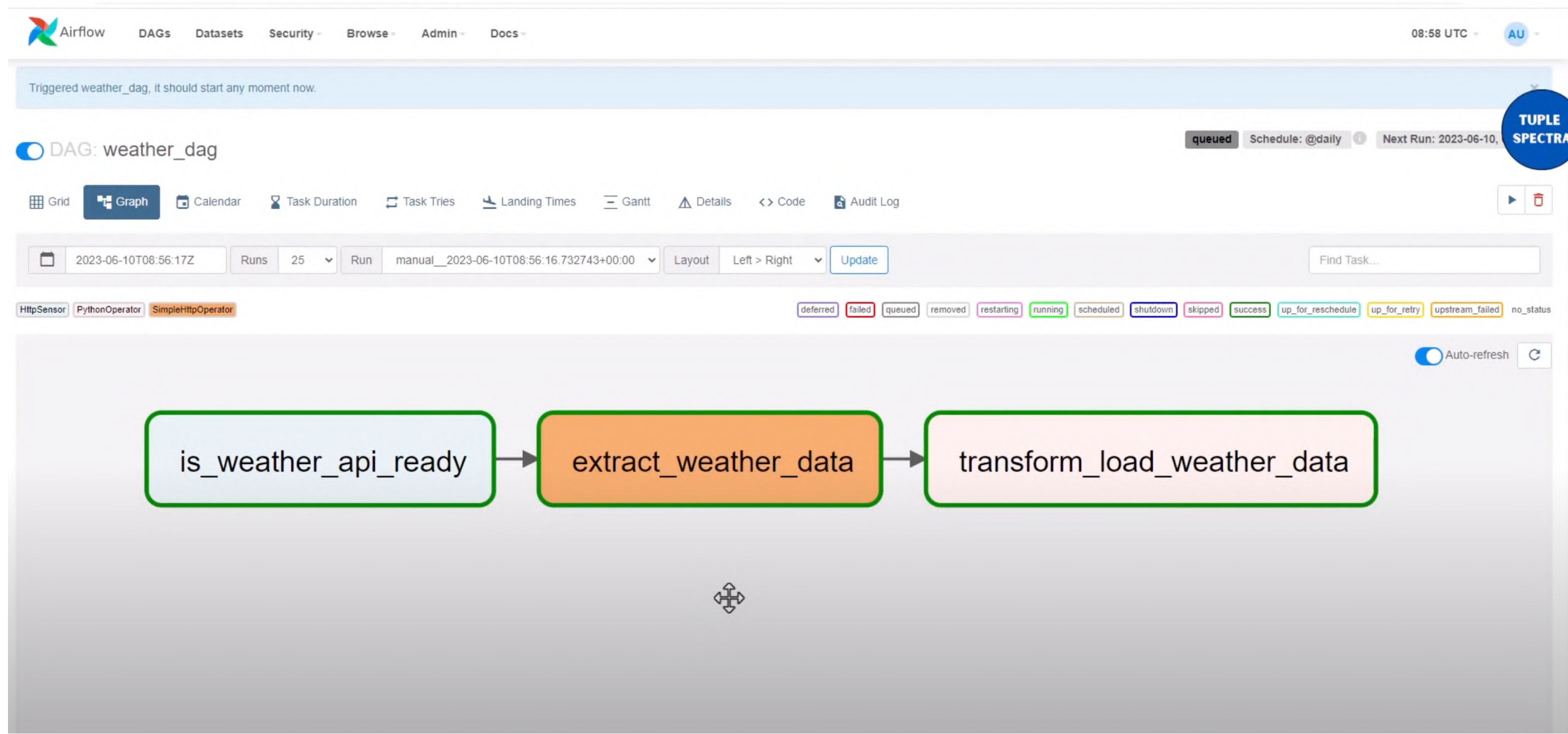
List XComs

Search

Actions

	Key	Value	Timestamp	Execution Date
☐	return_value		2021-01-05, 16:13:48	2021-01-04, 00:00:00
☐	return_value	7.955157501967618	2021-01-05, 16:13:50	2021-01-04, 00:00:00
☐	return_value	3.3474303828590024	2021-01-05, 16:13:50	2021-01-04, 00:00:00
☐	return_value	8.223962759587677	2021-01-05, 16:13:50	2021-01-04, 00:00:00

Building a basic DAG



Initializing the DAG

This Airflow code defines default parameters for a DAG, including the owner, start date, email notifications, and retry behavior. The code then creates a DAG named 'weather_dag' with the specified default parameters, scheduling it to run daily and disabling catch-up for missed runs.

```
default_args = {  
    'owner': 'airflow',  
    'depends_on_past': False,  
    'start_date': datetime(2023, 1, 8),  
    'email': ['myemail@domain.com'],  
    'email_on_failure': False,  
    'email_on_retry': False,  
    'retries': 2,  
    'retry_delay': timedelta(minutes=2)  
}
```

```
with DAG('weather_dag',  
        default_args=default_args,  
        schedule_interval = '@daily',  
        catchup=False) as dag:
```

```
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2023, 1, 8),
    'email': ['myemail@domain.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 2,
    'retry_delay': timedelta(minutes=2)
}
```

Alerts on failure

Retry policies builtin!

```
with DAG('weather_dag',
        default_args=default_args,
        schedule_interval = '@daily',
        catchup=False) as dag:
```

DAGs can be scheduled - No cron jobs needed!

Defining Tasks

Define three tasks within a DAG:
checking the readiness of a weather API, extracting weather data via an HTTP request, and executing a Python function for transforming and loading the data.

Ordering Tasks

(defining dependencies)

```
is_weather_api_ready = HttpSensor(  
    task_id='is_weather_api_ready',  
    http_conn_id='weathermap_api',  
    endpoint='/data/2.5/weather?q=Portland&APPID=5031cde3d1a8b9469fd47e998d7aef79'  
)  
  
extract_weather_data = SimpleHttpOperator(  
    task_id='extract_weather_data',  
    http_conn_id='weathermap_api',  
    endpoint='/data/2.5/weather?q=Portland&APPID=5031cde3d1a8b9469fd47e998d7aef79',  
    method='GET',  
    response_filter=lambda r: json.loads(r.text),  
    log_response=True  
)  
  
transform_load_weather_data = PythonOperator(  
    task_id='transform_load_weather_data',  
    python_callable=transform_load_data  
)  
  
is_weather_api_ready >> extract_weather_data >> transform_load_weather_data
```

Whats this?

```
is_weather_api_ready = HttpSensor(  
    task_id='is_weather_api_ready',  
    http_conn_id='weathermap_api',  
    endpoint='/data/2.5/weather?q=Portland&APPID=5031cde3d1a8b9469fd47e998d7aef79'  
)
```

```
extract_weather_data = SimpleHttpOperator(  
    task_id='extract_weather_data',  
    http_conn_id='weathermap_api',  
    endpoint='/data/2.5/weather?q=Portland&APPID=5031cde3d1a8b9469fd47e998d7aef79',  
    method='GET',  
    response_filter=lambda r: json.loads(r.text),  
    log_response=True  
)
```

```
transform_load_weather_data = PythonOperator(  
    task_id='transform_load_weather_data',  
    python_callable=transform_load_data  
)
```

```
is_weather_api_ready >> extract_weather_data >> transform_load_weather_data
```

- **task_id**: Specifies the unique ID of a task.
- **http_conn_id**: Associates the task with the Airflow connection named 'weathermap_api'.
- **endpoint**: Defines the API endpoint for the HTTP request, targeting weather data for Portland with a specific API key.
- **response_filter**: Utilizes a lambda function as a response filter to parse the HTTP response text as JSON, converting it into a Python dictionary.
- **Python callable**: A custom Python function called transform_load_data

Manik Rana
@ManaMkr

PyDelhi
Jan 2024

Thanks

