Paper SM04

Data Pipelines in Clinical Trials: Leveraging Apache Airflow ® for Automated Task Management

Kostiantyn Zvieriev, Veramed, Kharkiv, Ukraine

ABSTRACT

Currently, there exists a wide array of tools available for data processing and analysis, including programming languages with data analysis packages, big data processing solutions distributed across clusters, and various data storage systems from local files to cloud databases. The main challenge lies in effectively integrating these interconnected systems to function as a cohesive data processing pipeline.
Apache Airflow is an open-source platform designed for orchestrating complex workflows and data pipelines. With an intuitive interface, it simplifies task management and scheduling, automating workflows efficiently. Airflow represents workflows as Directed Acyclic Graphs (DAGs), aiding visualization of dependencies and task monitoring. Airflow provides extensive support for various data sources and integrations, facilitating interaction with popular tools and technologies in the data ecosystem.
This paper examines the pros and cons of Apache Airflow, explores its popularity in the field of data engineering, and discusses its applications and implementation in clinical trials.

AIRFLOW INTRODUCTION

Apache Airflow is an open-source platform designed to programmatically author, schedule, and monitor workflows. At its core, Airflow utilizes Directed Acyclic Graphs (DAGs) to represent workflows, where nodes represent tasks and edges denote dependencies. This modular and extensible architecture provides users with flexibility and scalability, making it an ideal choice for managing complex processes in clinical trials.
Airflow was developed by Airbnb to manage their complex data pipelines. Creating Airflow allowed Airbnb to programmatically author and schedule their workflows and monitor them via the built-in Airflow user interface. From the beginning, the project was made open source, becoming an Apache Incubator project in March 2016 and a top-level Apache Software Foundation project in January 2019.
Airflow is written in Python, and workflows are created via Python scripts. Airflow is designed under the principle of "configuration as code". While other "configuration as code" workflow platforms exist using markup languages like XML, using Python allows developers to import libraries and classes to help them create their workflows.

AIRFLOW ARCHITECTURE

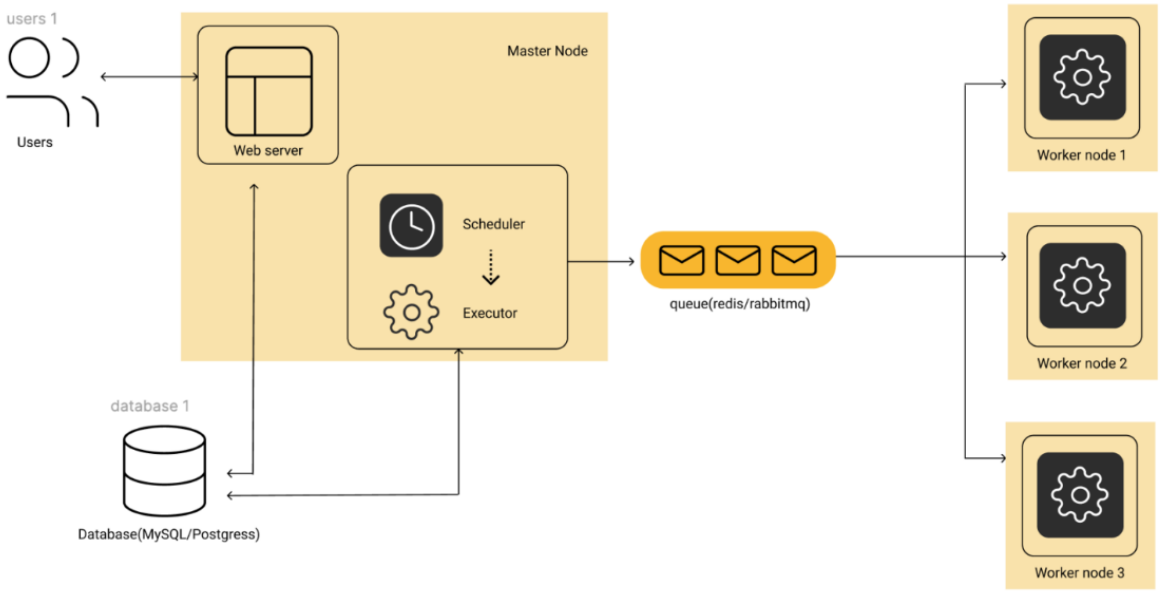
Prior to delving into the intricacies of task management within Apache Airflow, it is essential to familiarize oneself with its architecture. Understanding the architecture provides valuable insights into how the various components of Airflow interact and function cohesively. This foundational knowledge is crucial for effectively utilizing Airflow's capabilities in orchestrating complex workflows and managing data pipelines.
1) **Webserver** – servers as Web UI which allows users to monitor and manage workflows efficiently. Through the Web UI, users can monitor the status of Directed Acyclic Graphs (DAGs), access logs, and track the progress of individual tasks.
Additionally, the Webserver facilitates user management by allowing administrators to set permissions and control access to different sections of the interface and specific DAGs. It also plays a vital role in establishing connections with external services, such as AWS S3, local SQL databases, and email servers.
2) **Scheduler** – a critical component which is responsible for scheduling and determining DAGs timing of execution. It scans the definitions of DAGs and issues execution requests to the Executor.
3) **Executor** – a component which is responsible for defining the execution environment for tasks, determining which worker will carry out each task. There are several types of executors:

Local Executor - Airflow tasks are run locally within the scheduler process.

Celery Executor - distributes the workload from the main application onto multiple celery workers with the help of a message broker such as RabbitMQ or Redis. The executor publishes a request to execute a task in the queue, and one of several worker nodes picks up the request and runs as per the directions provided.

Kubernetes Executor - uses the Kubernetes API for resource optimization. It runs as a fixed-single Pod in the scheduler that only requires access to the Kubernetes API.

4) **Metadata Database** – Airflow uses Metadata DB to store information about DAGs, DAG runs, logs, statuses, scheduling.
5) **Workers** – execute tasks queried by Executor. For Celery and Kubernetes executors, workers can be distributed across the cluster of computers.
6) **Trigerrer** - The Triggerer allows tasks to be paused and resumed, making it ideal for long-running tasks that would otherwise tie up resources. Triggerer can handle thousands of deferrable tasks.



A minimal Airflow installation consists of the following components: a scheduler, a webserver, a metadata database. Notice that LocalExecutor is a part of scheduler process.

DAGS ANATOMY

DAGS

In Airflow, a **DAG** – or a Directed Acyclic Graph – is a collection of all the tasks you want to run, organized in a way that reflects their relationships and dependencies.
A DAG is defined in a Python script, which represents the DAGs structure (tasks and their dependencies) as code. For example, a simple DAG could consist of three tasks: A, B, and C. It could say that A has to run successfully before B can run, but C can run anytime. It could say that task A times out after 5 minutes, and B can be restarted up to 5 times in case it fails. It might also say that the workflow will run every night at 10pm, but shouldn't start until a certain date.
In this way, a DAG describes how you want to carry out your workflow; but notice that we haven't said anything about what we actually want to do! A, B, and C could be anything. Maybe A prepares data for B to analyze while C sends an email. The important thing is that the DAG isn't concerned with what its constituent tasks do; its job is to make sure that whatever they do happens at the right time, or in the right order, or with the right handling of any unexpected issues.

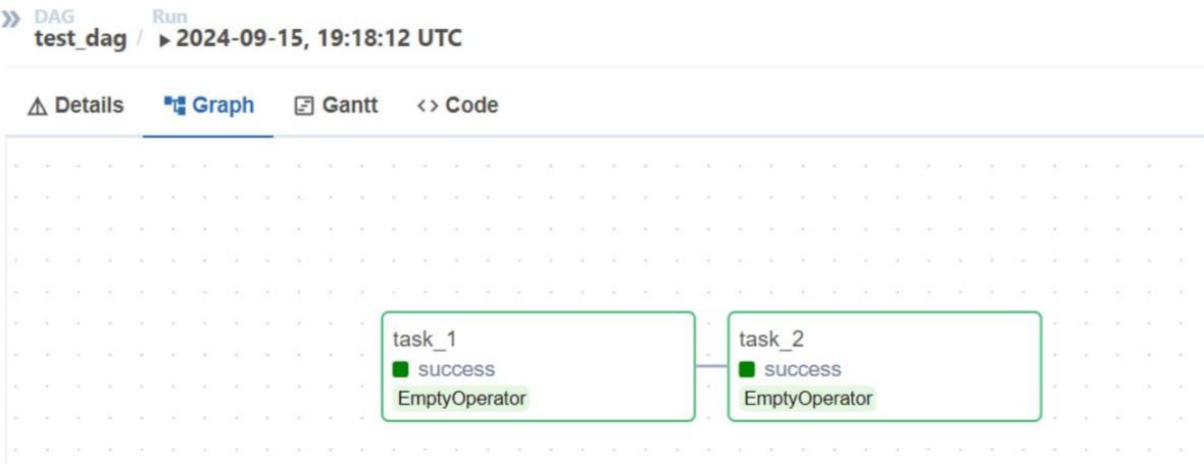
OPERATORS AND TASKS

An **Operator** defines a unit of work for Airflow to complete. Using operators is the classic approach to defining work in Airflow.
A **Task** defines a unit of work within a DAG; it is represented as a node in the DAG graph, and it is written in Python. Each task is an implementation of an Operator, for example a PythonOperator to execute some Python code, or a BashOperator to run a Bash command.

Consider the following DAG with two tasks. Each task is a node in our DAG, and there is a dependency from task_1 to task_2:

```
1. with DAG('my_dag', start_date=datetime(2016, 1, 1)) as dag:
2.     task_1 = DummyOperator('task_1')
3.     task_2 = DummyOperator('task_2')
4.     task_1 >> task_2 # Define dependencies
```

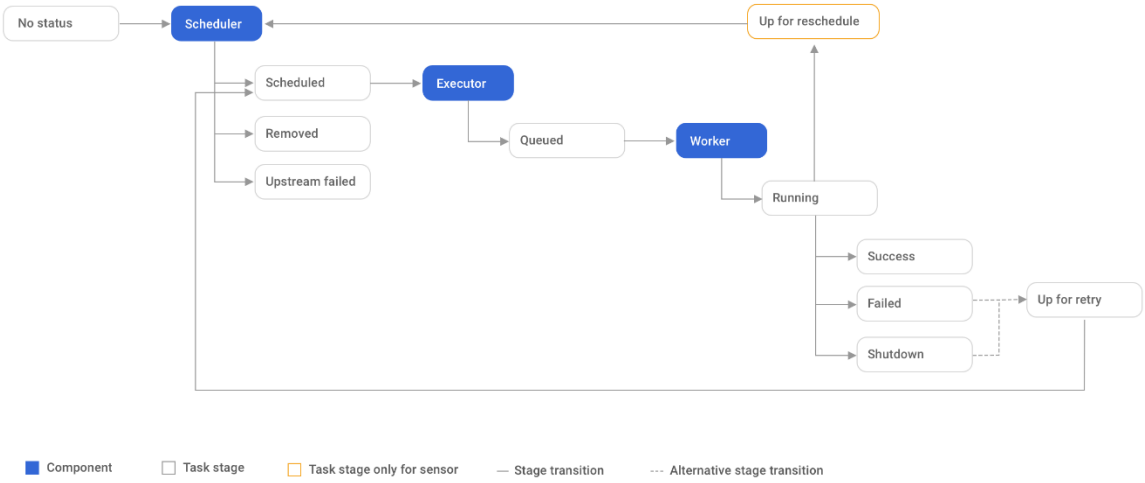
We can say that task_1 is *upstream* of task_2, and conversely task_2 is *downstream* of task_1. When a DAG Run is created, task_1 will start running and task_2 waits for task_1 to complete successfully before it may start.
DAG run can be visualized via Web UI:



TASK LIFECYCLE

A task goes through various stages from start to completion. In the Airflow UI (graph and tree views), these stages are displayed by a color representing each stage:

The complete lifecycle of the task looks like this:



OPERATORS

While DAGs describe *how* to run a workflow, Operators determine what actually gets done by a task.

An operator describes a single task in a workflow. Operators are usually (but not always) atomic, meaning they can stand on their own and don't need to share resources with any other operators. The DAG will make sure that operators run in the correct order; other than those dependencies, operators generally run independently. In fact, they may run on two completely different machines.
Airflow provides operators for many common tasks, including:

- BashOperator - executes a bash command
- PythonOperator - calls an arbitrary Python function
- EmailOperator - sends an email
- SimpleHttpOperator - sends an HTTP request
- MySqlOperator, SqliteOperator, PostgresOperator, MsSqlOperator, OracleOperator, JdbcOperator, etc. - executes a SQL command
- Sensor - an Operator that waits (polls) for a certain time, file, database row, S3 key, etc…

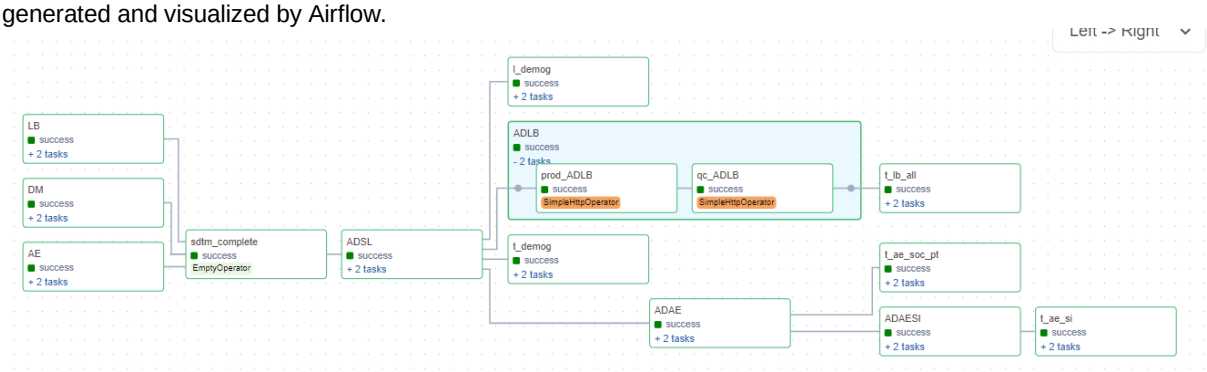
In Airflow user-defined operators can also be created.

THE CHALLENGE OF TASK DEPENDENCIES IN CLINICAL TRIALS

Running tasks in clinical trials presents a major challenge due to the network of dependencies between raw data, SDTMs, ADaMs and TFLs. Dependencies between tasks and human factors can cause delays and bottlenecks, creating a cascade effect. The traditional approach of manually managing tasks and taking into account their dependencies is not the most effective and can be quite time-consuming.

HOW AIRFLOW CAN SOLVE IT

Airflow addresses the curse of task dependencies through its DAG-based architecture. By representing workflows as DAGs, users have the ability to explicitly define task dependencies, enabling Airflow to intelligently orchestrate their execution. DAGs facilitate parallelism, retries, and backfilling, ensuring tasks are executed in the proper sequence while optimizing efficiency. Below is a diagram showcasing an example DAG, which was automatically generated and visualized by Airflow.

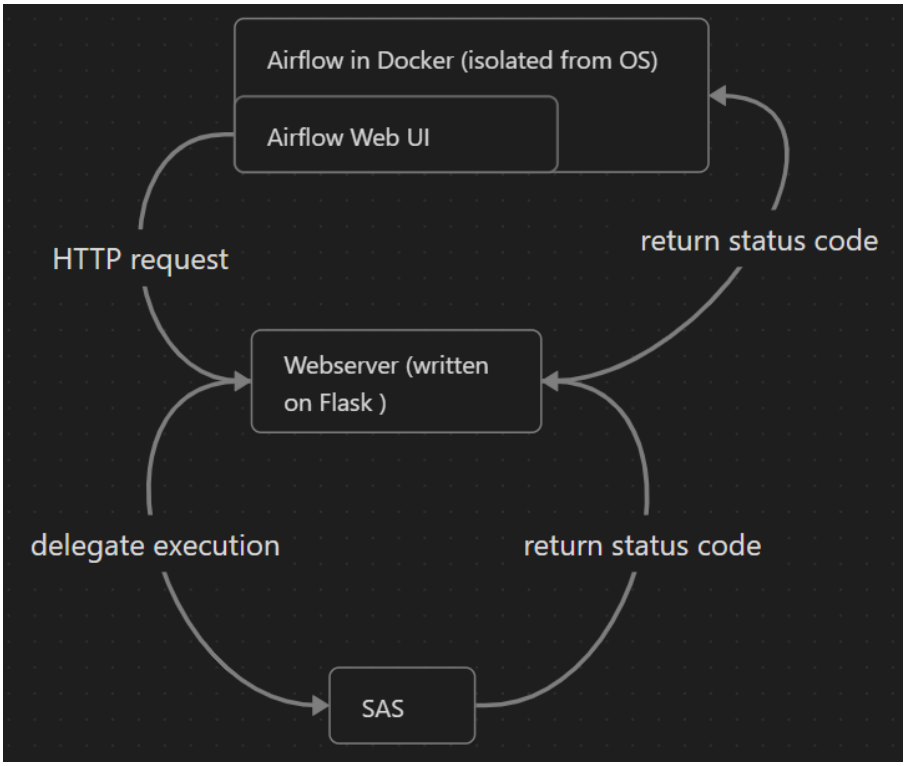


SETUP AND IMPLEMENTATION FOR CLINICAL TRIALS

Implementing Airflow for clinical trials begins with configuring the environment and defining DAGs to model trial workflows. Key components include setting up the Airflow scheduler, executor, metadata database, and web interface. DAGs are constructed using Python scripts, specifying tasks, dependencies, and execution logic.
We will be using Airflow for a range of important tasks:

- managing workflow on extensive runs of study,
- adding extra steps in the default data processing pipeline;
- sending out notifications to the users in case something goes wrong,
- keeping users away from other data pipelines to ensure no tampering with data and security.

For that, we will execute Airflow within a Docker ® environment that brings significant benefits in terms of portability, consistency, and non-complex scaling across different deployment scenarios. Besides being lightweight, Docker containers are isolated; therefore, it is much easier to deploy and manage one such container compared to a full-fledged virtual machine. While implementing the use case for Airflow in clinical studies, we have few important assumptions. Because Airflow is designed to process data on a cluster of computers communicating on the network, we believe using network interactions will be better suited for a more accurate result than relying on pure directory operations. We will create a simple web server using Flask, which acts like a pass-through that dispatches external systems to execute a task, thus enabling it to receive web requests from Airflow easily.
Following diagram illustrates our setup:



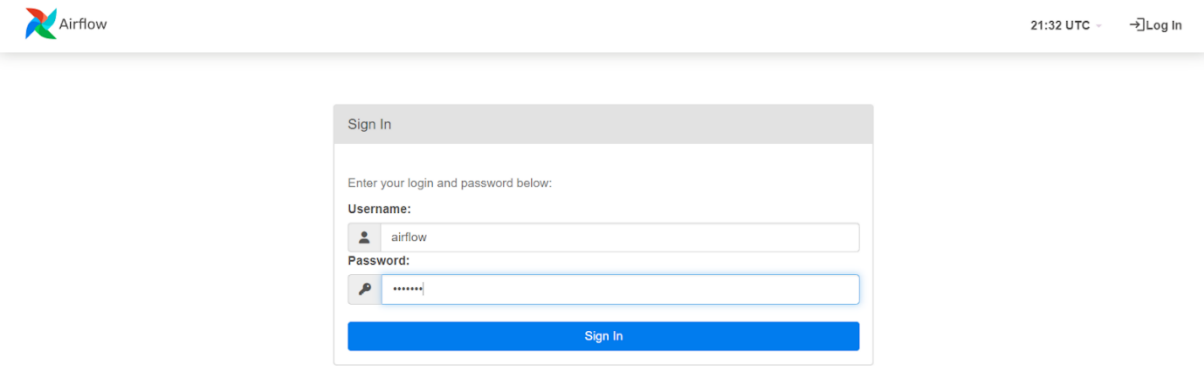
AIRFLOW INSTALLATION

Prior to diving into the world of Airflow pipelines, it's essential to set the stage for a smooth experience:

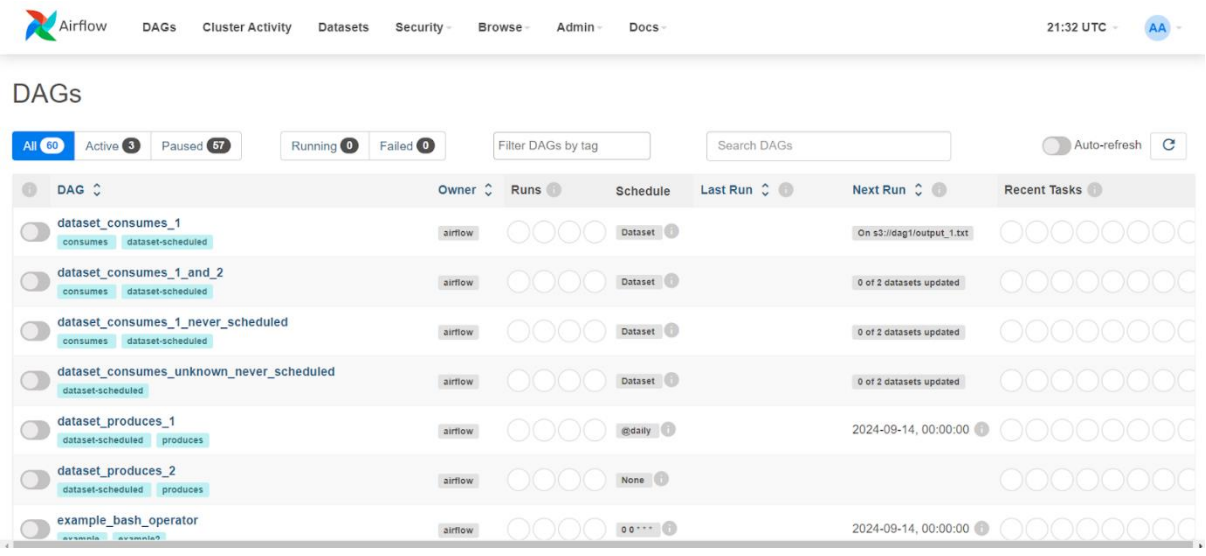
- Start by installing Docker on your machine, as it serves as the foundation for running Airflow.
- Next, head over to the official Apache Airflow website to download the docker-compose.yaml file, which acts as a crucial configuration blueprint for managing Airflow within Docker.
- This file is not set in stone; feel free to modify it to suit the specific requirements of your tasks.
- Once you have the docker-compose file in project directory, navigate to it and create the necessary folders: dags/, logs/, config/, and plugins/.
- Finally, kick off your setup by executing the docker-compose up command, which will bring your Airflow environment to life. Make sure Docker is up and running.

Detailed documentation is available on the official website.

Once Airflow is up and running within your Docker environment, accessing the platform requires you to input your credentials. The default login credentials for Airflow are both set to "airflow," which means you will need to enter this username and password combination to enter the system. Make sure to change password for Admin account in production environment.



Once you have logged into the Airflow Web UI, the following interface will be displayed:



CREATING FIRST DAG

It's time to create our first Directed Acyclic Graph (DAG) for SDTM DM and DM QC. We will establish two tasks, t1 for DM and t2 for DM QC, and set up their dependencies accordingly. Since the QC process must follow the Production program, we will arrange it as t1 >> t2. By utilizing the SimpleHttpOperator, we can send requests to run our SAS ® code on machines that may not have Airflow installed.

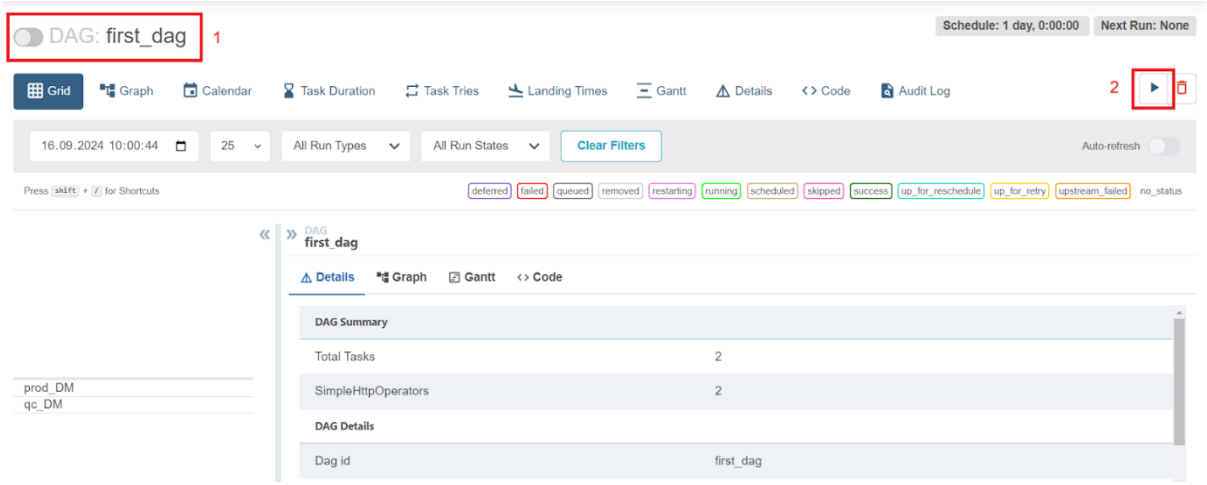
Create myfirstdag.py in /dags folder:

```
1. from airflow import DAG
2. from airflow.operators.http_operator import SimpleHttpOperator
3. from datetime import datetime
4.
5. with DAG(dag_id='first_dag') as dag:
6.     t1 = SimpleHttpOperator(
7.         task_id=f'prod_DM',
```

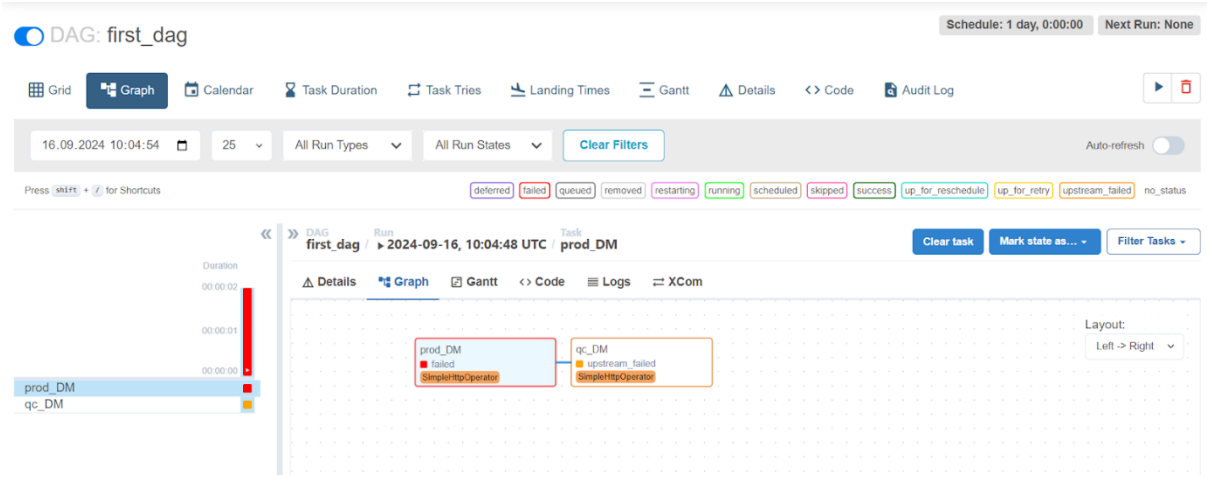
```
 8.          http_conn_id='http_flask_server',
 9.          method='GET',
10.          endpoint=f'/exec?dsname=DM'
11.      )
12.      t2 = SimpleHttpOperator(
13.          task_id=f'qc_DM',
14.          http_conn_id='http_flask_server',
15.          method='GET',
16.          endpoint=f'/exec?dsname=qc_DM'
17.      )
18.      t1>>t2
```

When Airflow is up within a container, a new DAG named first_dag will be visible in your Web UI. This occurs as the scheduler scans the dags directory and updates the Metadata Database with details about the new DAG, which Webserver subsequently checks.

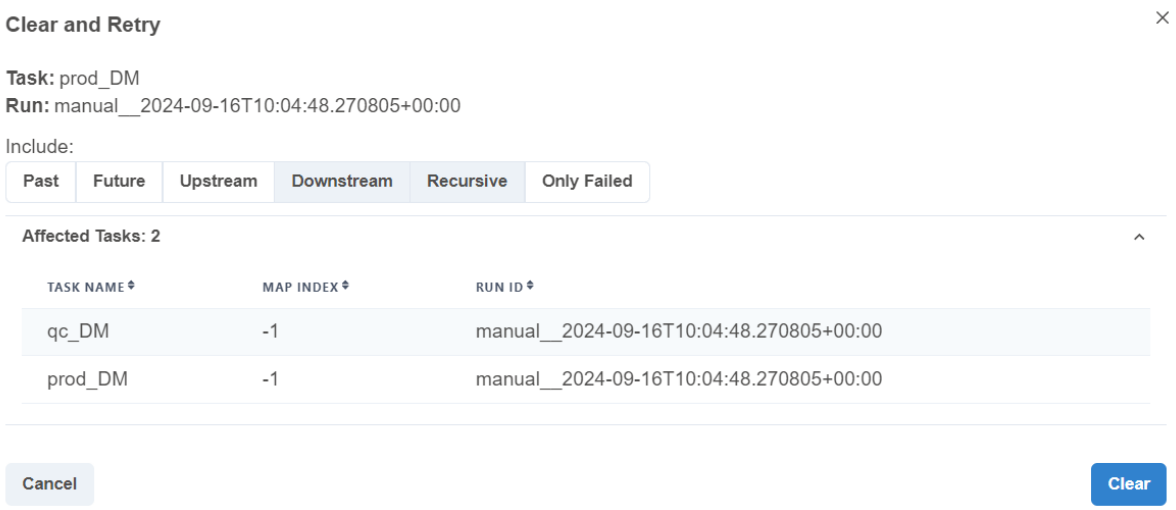
Unpause (1) and start (2) your DAG:



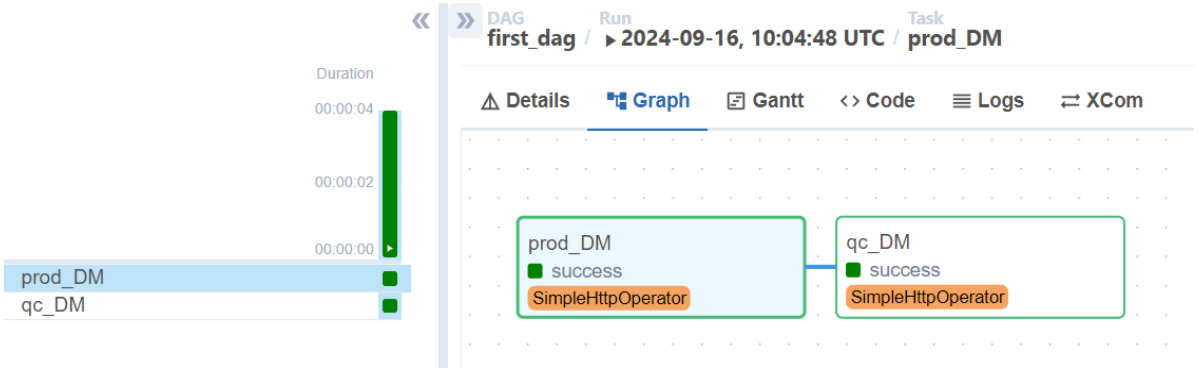
The initial attempt failed due to the Flask web server was not running:



By clicking the Clear button, our tasks will be restarted. Airflow indicates which tasks will be impacted by this restart.

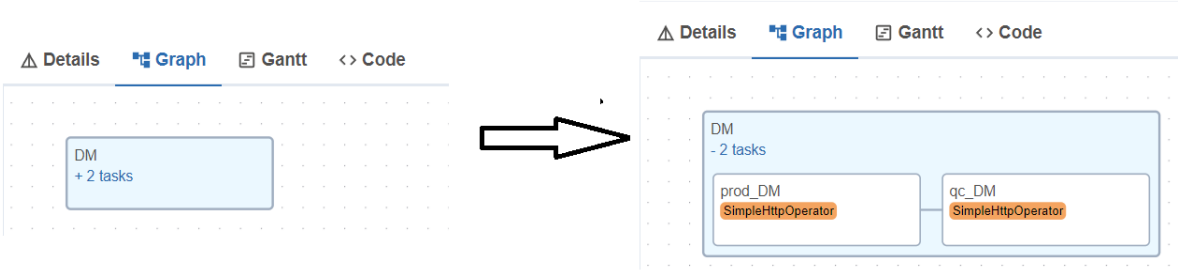


After restarting, the tasks executes successfully:



To organize our DAG with all its instantly increasing dependencies, we utilize TaskGroup, which helps us visually cluster related tasks in the Airflow UI grid, making complex DAGs more manageable while allowing us to apply default_args to groups of tasks.

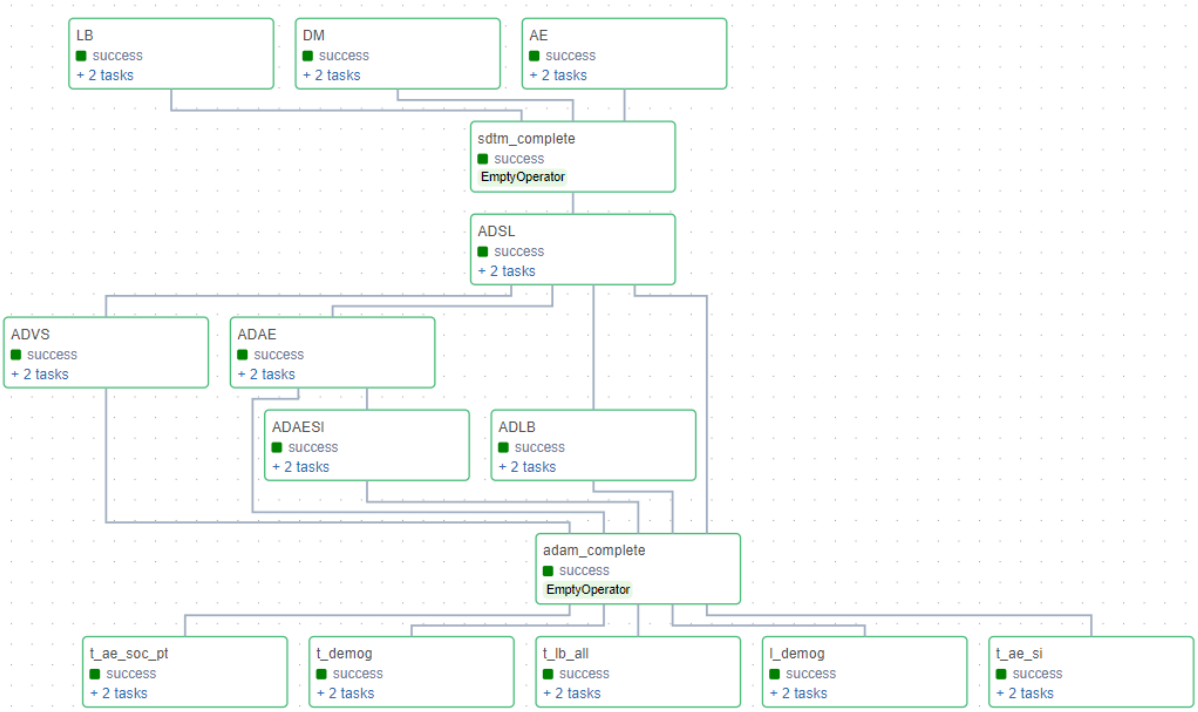
```
from airflow import DAG
from airflow.operators.http_operator import SimpleHttpOperator
from airflow.utils.task_group import TaskGroup
from datetime import datetime

with DAG(dag_id='first_dag') as dag:
    with TaskGroup(f"DM", tooltip = f"Prod and QC for DM") as tgr:
        t1 = SimpleHttpOperator(
            task_id=f'prod_DM',
            http_conn_id='http_flask_server',
            method='GET',
            endpoint=f'/exec?dsname=DM'
        )
        t2 = SimpleHttpOperator(
            task_id=f'qc_DM',
            http_conn_id='http_flask_server',
            method='GET',
            endpoint=f'/exec?dsname=qc_DM'
        )
        t1>>t2
```



Airflow empowers the creation of incredibly flexible data pipelines. This method of organizing task dependencies serves as an example:

1. Each task is followed by a corresponding QC task, and both are organized within a TaskGroup.
2. Initially, SDTM tasks are executed, followed by the addition of an EmptyOperator SDTM_complete to enhance dependency visualization in the Web UI.
3. Once all SDTM programs are completed, the ADaM programs are initiated, respecting their dependencies, with ADSL being prioritized as the first dataset. Additional dependencies can be incorporated as needed, such as ADAE-ADAESI, and an EmptyOperator named ADAM_complete can also be utilized for clearer visualization.
4. The next phase involves launching TFL tasks.
5. Optional: the creation of xpt files and validation through the Pinnacle21 API.
6. Optional: Additional data checks, user-defined data validation tools, integration with third-party systems.
7. Optional: Inclusion of PK-related datasets (PopPK) into pipeline.
8. Optional: Notification about successful completion of data pipeline using EmailOperator.

```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.http_operator import SimpleHttpOperator
from airflow.operators.dummy import DummyOperator
from airflow.utils.task_group import TaskGroup

default_args = {
    'owner' : "Airflow",
}

with DAG(
    dag_id="my5dag",
    schedule_interval=None, #add how frequently pipeline run. None = only manual run
    default_args=default_args
) as dag1:
    task_names = ["DM", "LB", "AE", "ADSL", "ADAE", "ADLB", "ADVS", "ADAESI",
                  "t_demog", "l_demog", "t_ae_soc_pt", "t_ae_si", "t_lb_all"]

    #dummy tasks
    SDTM_complete = DummyOperator(task_id="sdtm_complete")
    ADAM_complete = DummyOperator(task_id="adam_complete")

    #td - task dictionary
    td = {}
    #create groupings for PROD and QC
    for name in task_names :
        with TaskGroup(f"{name}", tooltip = f"Prod and QC for {name}") as tgr:
            t1 = SimpleHttpOperator(
                task_id=f'prod_{name}',
                http_conn_id='http_flask_server',
                method='GET',
                endpoint=f'/exec?dsname={name}'
            )
            t2 = SimpleHttpOperator(
                task_id=f'qc_{name}',
                http_conn_id='http_flask_server',
                method='GET',
                endpoint=f'/exec?dsname=qc_{name}'
            )
            t1>>t2
            td[name] = (tgr,t1,t2)

    #STDM complete dependency
    for k, v in td.items():
        if k in ["AE", "LB", "DM"]:
            v[0] >> SDTM_complete
    SDTM_complete >> td["ADSL"][0]

    td["ADSL"][0] >> [td["ADLB"][0],td["ADAE"][0],td["ADVS"][0]]
    td["ADAE"][0] >> td["ADAESI"][0]


    for k, v in td.items():
        if k in ["ADSL", "ADLB",  "ADVS","ADAE", "ADAESI"]:
            v[0] >> ADAM_complete

    for k, v in td.items():
        if k in ["t_demog", "l_demog", "t_ae_soc_pt", "t_ae_si", "t_lb_all"]:
            ADAM_complete >> v[0]
```



This data pipeline can be supplemented and expanded depending on the needs and desires of the user.
One of the significant benefits of utilizing Airflow is the elimination of human factor. The study lead programmer is not required to monitor the study run, as tasks are carried out sequentially. In the event of a failure, the data pipeline can resume from the point of interruption rather than restarting from the beginning. The dependencies are executed subsequently, in case of a failure, the data pipeline can be resumed from the failed task rather than restarting from the beginning.
Warning: Adding a new ADaM or SDTM during the study run can create bottlenecks with the ADaM_complete and SDTM_complete tasks, necessitating a restart of all downstream dependencies. For SDTM_complete, this includes all ADaM and TFL programs. You might want to organize tasks by domains and further categorize them into SDTM, ADaM, and TFL, for this purpose Airflow offers support for nested TaskGroups to facilitate this structure.

EMAIL ALERTS ON FAILURE

Airflow offers a variety of useful operators, such as EmailOperator, but it is not suitable for notifying about failed tasks. Fortunately, Airflow includes native support for failure notifications. To activate these notifications, simply incorporate the necessary configuration settings for your Gmail account into the docker-compose.yaml file.

```
1. AIRFLOW__SMTP__SMTP_HOST: smtp.gmail.com
2. AIRFLOW__SMTP__SMTP_STARTTLS : True
3. AIRFLOW__SMTP__SMTP_SSL : False
4. AIRFLOW__SMTP__SMTP_USER : #sender email address
5. AIRFLOW__SMTP__SMTP_PASSWORD: #password (do not save password in config in production env)
6. AIRFLOW__SMTP__SMTP_PORT: 587
7. AIRFLOW__SMTP__SMTP_MAIL_FROM: #sender email address
```

Do not forget to add receivers in DAG's default_args:

```
1. default_args = {
2.     'email': ['receiver1@gmail.com', 'receiver2@yourdomain.com'],
3.     'email_on_failure': True
4. }
```

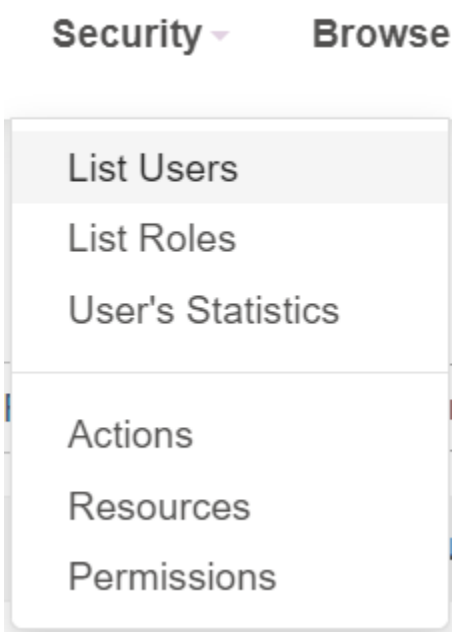
For guidance on setting up Airflow with a different mail host, please refer to the documentation.

SECURITY IN AIRFLOW

This section focuses on the security aspects of Airflow. As an orchestrator, Airflow interfaces with various systems, making it an attractive target for potential intruders. We will ensure that unauthorized individuals cannot access your WebUI and will implement measures to isolate users, granting them access solely to the DAGs they are permitted to view. Additionally, we will explore the encryption of the Metadata Database.

ROLE-BASED ACCESS CONTROL

Airflow provides an RBAC (Role-based access control) interface out of the box. The RBAC interface provides a mechanism that restricts access by defining roles with corresponding permissions and assigning these roles to users. In Airflow, we have the ability to create users and create roles for them.



Airflow comes with a predefined set of five roles that are readily available for use. The default permissions associated with these roles can be found in the table below.

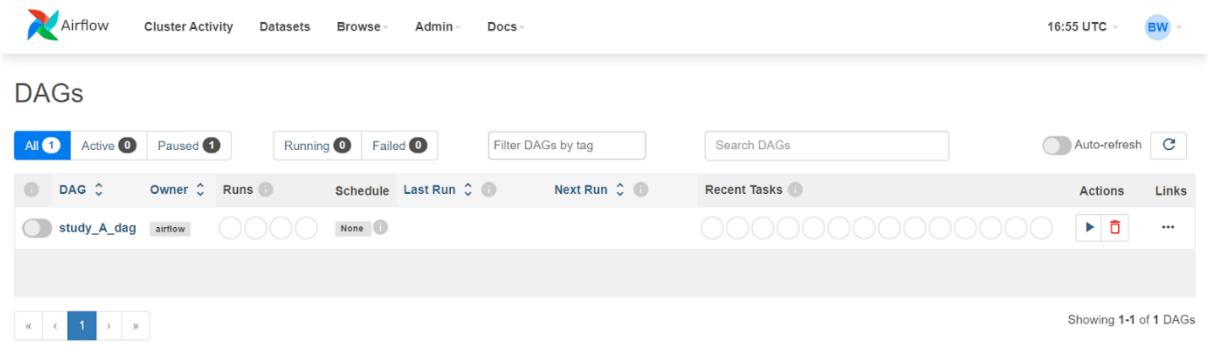
Role name	Intended users/usage	Default permissions
Admin	Only necessary when managing security permissions	All permissions
Public	Unauthenticated users	No permissions
Viewer	Read-only view of Airflow	Read access to DAGs
User	Useful if you want strict separation in your team between developers who can and cannot edit secrets (connections, variables, etc.). This role only grants permissions to create DAGs, not secrets.	Same as viewer but with edit permissions (clear, trigger, pause, etc.) on DAGs
Op	All permissions required for developing Airflow DAGs	Same as user but with additional permissions to view and edit connections, pools, variables, XComs, and configuration

Permissions are quite granular; access to each menu and menu item is controlled by permissions. For example, to make the Documents menu visible, you would add the "access the Documents menu" permission. And to make the Documentation menu item visible in the Documents menu, you would add the "access the Documentation menu" permission. Finding the right permissions can sometimes be cumbersome. The easiest way is to check other roles to see what permissions are available. Permissions are displayed as a string, which in most cases is self-explanatory regarding the access granted. The ability to assign such granular permissions allows us to restrict users from seeing DAGs that are not intended for them. This is especially useful in clinical trials, where we need to isolate different study teams from each other. This approach eliminates the need to spin up a separate Airflow instance for each study.

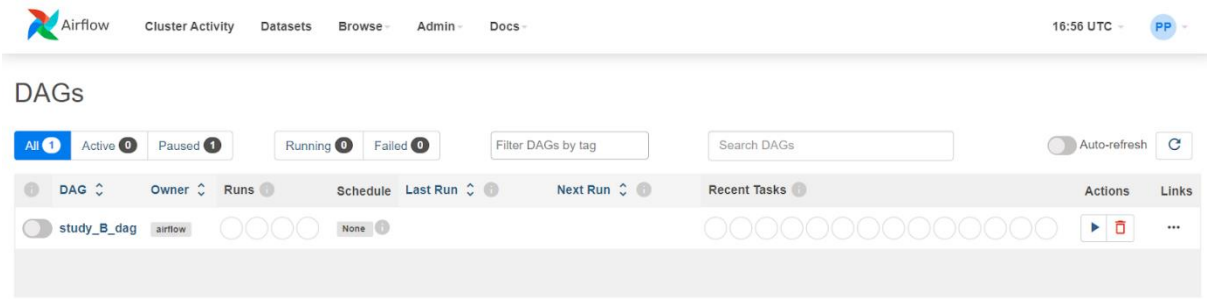
This approach can be implemented by establishing a series of roles, including:
BaseRole – which includes all permissions related to DAGs for a standard user but excludes access to any DAG;
StudyARole – which grants permissions specific to the Study A DAG;
StudyBRole – which grants permissions specific to the Study B DAG.
We will set up two users: StudyAProgrammer and StudyBProgrammer.

	First Name	Last Name	User Name	Email	Is Active?	Role
	Peter	Parker	StudyBProgrammer	petersemail@email.com	True	[BaseUser, StudyBRole]
	Bruce	Wayne	StudyAProgrammer	brucesemail@email.com	True	[BaseUser, StudyARole]
	Airflow	Admin	airflow	airflowadmin@example.com	True	[Admin]

StudyAProgrammer sees Airflow's Web UI as follows:



StudyBProgrammer sees Airflow's Web UI as follows:

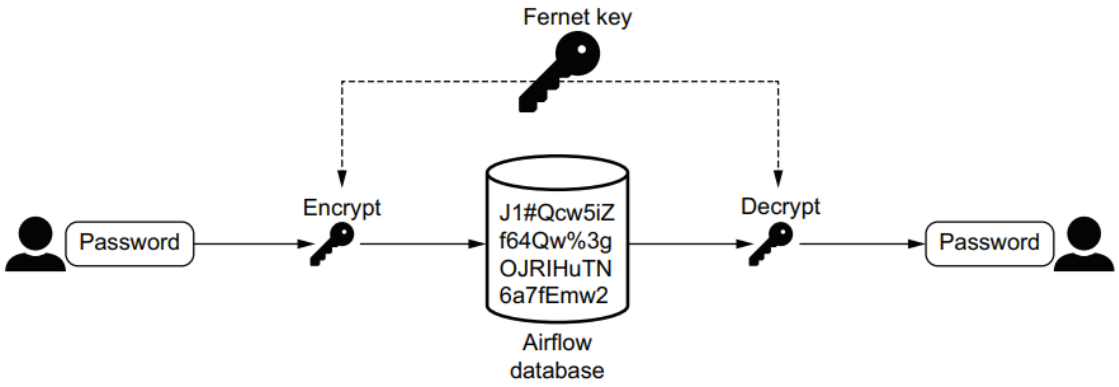


ENCRYPTING DATA

The RBAC interface requires users to be in the database with a username and password. This prevents random strangers from accessing Airflow "just looking around" but it's far from ideal.

Before we dive into encryption, let's go back to the basic Airflow architecture. Airflow is made up of several components. Each of these is a potential threat because it serves as a pathway through which uninvited guests can gain access to your systems. Reducing the number of open entry points (i.e., reducing the attack surface) is always a good idea. If you must expose a service for practical reasons, such as the Airflow web server, you want to make sure it's not publicly accessible. You also want to make sure your data is safe once an attacker has gained access to it. Before you create users and passwords, make sure Airflow has encryption enabled. Without encryption, passwords (and other secrets, like connections) are stored in the database unencrypted. Anyone with access to the database can also read the passwords. When encrypted, they are stored as a sequence of seemingly random characters, which is essentially useless.

Airflow can encrypt and decrypt secrets using what is known as a Fernet key.



The Fernet key serves as a crucial string for both encryption and decryption processes. Losing this key means that any encrypted messages cannot be decrypted.

To provide Airflow with a Fernet key, you can generate one.

You can set this key in the configuration to ensure Airflow can utilize it effectively.

```
1. AIRFLOW__CORE__FERNET_KEY = #your fernet key
```

WHAT IS NEXT?

This article only scratches the surface of the Airflow iceberg, but it should be enough to get you started using Airflow in real projects to speed up your delivery process.

The following aspects of Airflow are beyond the scope of this article but may be useful:

- Scheduling - The ability to schedule data pipeline execution is a key feature of Airflow. In the article, DAGs were started manually, but Airflow allows you to choose the frequency with which the DAG should be executed, for example, once a day or once a month.
- Branching Operators - operators which change flow of execution depending on a condition (e.g. PythonBranchOperator)
- Hooks - A Hook is a high-level interface to an external platform that lets you quickly and easily talk to them without having to write low-level code that hits their API or uses special libraries. They're also often the building blocks that Operators are built out of.
- XComs -XCom (short for "cross-communication") is a mechanism that let Tasks talk to each other, as by default Tasks are entirely isolated and may be running on entirely different machines.
- Executors in depth - for single machine you would probably use LocalExecutor, because it is lightweight and conceptually easier to understand, but for a higher load you might deploy Airflow on cluster using more advanced executors like CeleryExecutor or KubernetesExecutor.

CONCLUSION

Airflow has long been the industry standard in data engineering for batch data processing due to its robust, scalable, and highly customizable workflow management framework. Widely used across a variety of sectors, Airflow excels at orchestrating complex, multi-step processes. Clinical trials are no exception, as Airflow automates routine tasks and reduces delivery time. By automating workflow orchestration using Directed Acyclic Graphs (DAGs), Airflow efficiently manages dependencies, monitors task execution, and ensures data processing flows smoothly.
The use case focused on clinical trials demonstrates how Airflow simplifies handling tasks in extensive data workflows, allowing automation of key processes like data cleaning, analysis, and report generation. Airflow's flexible, Python-based structure enables the construction of highly customized and scalable workflows.

RECOMMENDED READING

- Apache Airflow Official Documentation https://airflow.apache.org/docs/apache-airflow/stable/index.html
- Data Pipelines with Apache Airflow by Bas P. Harenslak, Julian Rutger de Ruiter https://www.amazon.com/Data-Pipelines-Apache-Airflow-Harenslak/dp/1617296902

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Kostiantyn Zvieriev
Veramed
kostiantyn.zvieriev@veramed.co.uk

ATTACHMENT

Minimal docker-compose.yaml for Airflow:

```
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements.  See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership.  The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License.  You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied.  See the License for the
# specific language governing permissions and limitations
# under the License.
#
#
# WARNING: This configuration is for local development. Do not use it in a production deployment.
#
# This configuration supports basic configuration using environment variables or an .env file
# The following variables are supported:
#
# AIRFLOW_IMAGE_NAME           - Docker image name used to run Airflow.
#                                Default: apache/airflow:2.8.1
# AIRFLOW_UID                  - User ID in Airflow containers
```

```
#                                Default: 50000
# AIRFLOW_PROJ_DIR             - Base path to which all the files will be volumed.
#                                Default: .
# Those configurations are useful mostly in case of standalone testing/running Airflow in test/try-out mode
#
# _AIRFLOW_WWW_USER_USERNAME   - Username for the administrator account (if requested).
#                                Default: airflow
# _AIRFLOW_WWW_USER_PASSWORD   - Password for the administrator account (if requested).
#                                Default: airflow
# _PIP_ADDITIONAL_REQUIREMENTS - Additional PIP requirements to add when starting all containers.
#                                Use this option ONLY for quick checks. Installing requirements at container
#                                startup is done EVERY TIME the service is started.
#                                A better way is to build a custom image or extend the official image
#                                as described in https://airflow.apache.org/docs/docker-stack/build.html.
#                                Default: ''
#
# Feel free to modify this file to suit your needs.
---
x-airflow-common:
  &airflow-common
  # In order to add custom dependencies or upgrade provider packages you can use your extended image.
  # Comment the image line, place your Dockerfile in the directory where you placed the docker-compose.yaml
  # and uncomment the "build" line below, Then run `docker-compose build` to build the images.
  image: ${AIRFLOW_IMAGE_NAME:-apache/airflow:2.8.1}
  # build: .
  environment:
    &airflow-common-env
    AIRFLOW__CORE__EXECUTOR: LocalExecutor
    AIRFLOW__DATABASE__SQL_ALCHEMY_CONN: postgresql+psycopg2://airflow:airflow@postgres/airflow
    AIRFLOW__CORE__FERNET_KEY: 'yourfernetkey'
    AIRFLOW__CORE__DAGS_ARE_PAUSED_AT_CREATION: 'true'
    AIRFLOW__CORE__LOAD_EXAMPLES: 'true'
    AIRFLOW__API__AUTH_BACKENDS: 'airflow.api.auth.backend.basic_auth,airflow.api.auth.backend.session'
    # yamllint disable rule:line-length
    # Use simple http server on scheduler for health checks
    # See https://airflow.apache.org/docs/apache-airflow/stable/administration-and-deployment/logging-monitoring/check-health.html#scheduler-health-check-server
    # yamllint enable rule:line-length
    AIRFLOW__SCHEDULER__ENABLE_HEALTH_CHECK: 'true'
    #email notifications config
    AIRFLOW__SMTP__SMTP_HOST: smtp.gmail.com
    AIRFLOW__SMTP__SMTP_STARTTLS : True
    AIRFLOW__SMTP__SMTP_SSL : False
    AIRFLOW__SMTP__SMTP_USER : #sender email
    AIRFLOW__SMTP__SMTP_PASSWORD: #password
    AIRFLOW__SMTP__SMTP_PORT: 587
    AIRFLOW__SMTP__SMTP_MAIL_FROM: #sender email
    # WARNING: Use _PIP_ADDITIONAL_REQUIREMENTS option ONLY for a quick checks
    # for other purpose (development, test and especially production usage) build/extend Airflow image.
    _PIP_ADDITIONAL_REQUIREMENTS: ${_PIP_ADDITIONAL_REQUIREMENTS:-}
  volumes:
    - ${AIRFLOW_PROJ_DIR:-.}/dags:/opt/airflow/dags
    - ${AIRFLOW_PROJ_DIR:-.}/logs:/opt/airflow/logs
    - ${AIRFLOW_PROJ_DIR:-.}/config:/opt/airflow/config
    - ${AIRFLOW_PROJ_DIR:-.}/plugins:/opt/airflow/plugins
  user: "${AIRFLOW_UID:-50000}:0"
  depends_on:
    &airflow-common-depends-on
    postgres:
      condition: service_healthy

services:
  postgres:
    image: postgres:13
```

```
    environment:
      POSTGRES_USER: airflow
      POSTGRES_PASSWORD: airflow
      POSTGRES_DB: airflow
    volumes:
      - postgres-db-volume:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "airflow"]
      interval: 10s
      retries: 5
      start_period: 5s
    restart: always

  airflow-webserver:
    <<: *airflow-common
    command: webserver
    ports:
      - "8080:8080"
    healthcheck:
      test: ["CMD", "curl", "--fail", "http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 5
      start_period: 30s
    restart: always
    depends_on:
      <<: *airflow-common-depends-on
      airflow-init:
        condition: service_completed_successfully

  airflow-scheduler:
    <<: *airflow-common
    command: scheduler
    healthcheck:
      test: ["CMD", "curl", "--fail", "http://localhost:8974/health"]
      interval: 30s
      timeout: 10s
      retries: 5
      start_period: 30s
    restart: always
    depends_on:
      <<: *airflow-common-depends-on
      airflow-init:
        condition: service_completed_successfully

  airflow-init:
    <<: *airflow-common
    entrypoint: /bin/bash
    # yamllint disable rule:line-length
    command:
      - -c
      - |
        if [[ -z "${AIRFLOW_UID}" ]]; then
          echo
          echo -e "\033[1;33mWARNING!!!: AIRFLOW_UID not set!\e[0m"
          echo "If you are on Linux, you SHOULD follow the instructions below to set "
          echo "AIRFLOW_UID environment variable, otherwise files will be owned by root."
          echo "For other operating systems you can get rid of the warning with manually
created .env file:"
          echo "    See: https://airflow.apache.org/docs/apache-
airflow/stable/howto/docker-compose/index.html#setting-the-right-airflow-user"
          echo
        fi
        one_meg=1048576
        mem_available=$$(($$(getconf _PHYS_PAGES) * $$(getconf PAGE_SIZE) / one_meg))
        cpus_available=$$(grep -cE 'cpu[0-9]+' /proc/stat)
        disk_available=$$(df / | tail -1 | awk '{print $$4}')
        warning_resources="false"
        if (( mem_available < 4000 )) ; then
          echo
          echo -e "\033[1;33mWARNING!!!: Not enough memory available for Docker.\e[0m"
          echo "At least 4GB of memory required. You have $$(numfmt --to iec
$$((mem_available * one_meg)))"
          echo
          warning_resources="true"
```

```
        fi
        if (( cpus_available < 2 )); then
          echo
          echo -e "\033[1;33mWARNING!!!: Not enough CPUS available for Docker.\e[0m"
          echo "At least 2 CPUs recommended. You have $${cpus_available}"
          echo
          warning_resources="true"
        fi
        if (( disk_available < one_meg * 10 )); then
          echo
          echo -e "\033[1;33mWARNING!!!: Not enough Disk space available for Docker.\e[0m"
          echo "At least 10 GBs recommended. You have $$(numfmt --to iec $$((disk_available * 1024 )))"
          echo
          warning_resources="true"
        fi
        if [[ $${warning_resources} == "true" ]]; then
          echo
          echo -e "\033[1;33mWARNING!!!: You have not enough resources to run Airflow (see above)!\e[0m"
          echo "Please follow the instructions to increase amount of resources available:"
          echo "   https://airflow.apache.org/docs/apache-airflow/stable/howto/docker-compose/index.html#before-you-begin"
          echo
        fi
        mkdir -p /sources/logs /sources/dags /sources/plugins
        chown -R "${AIRFLOW_UID}:0" /sources/{logs,dags,plugins}
        exec /entrypoint airflow version
    # yamllint enable rule:line-length
    environment:
      <<: *airflow-common-env
      _AIRFLOW_DB_MIGRATE: 'true'
      _AIRFLOW_WWW_USER_CREATE: 'true'
      _AIRFLOW_WWW_USER_USERNAME: ${_AIRFLOW_WWW_USER_USERNAME:-airflow}
      _AIRFLOW_WWW_USER_PASSWORD: ${_AIRFLOW_WWW_USER_PASSWORD:-airflow}
      _PIP_ADDITIONAL_REQUIREMENTS: ''
    user: "0:0"
    volumes:
      - ${AIRFLOW_PROJ_DIR:-.}:/sources

  airflow-cli:
    <<: *airflow-common
    profiles:
      - debug
    environment:
      <<: *airflow-common-env
      CONNECTION_CHECK_MAX_COUNT: "0"
    # Workaround for entrypoint issue. See: https://github.com/apache/airflow/issues/16252
    command:
      - bash
      - -c
      - airflow

volumes:
  postgres-db-volume:
```

Flask server:

```
from flask import Flask, request, make_response
import subprocess
import sys

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"

@app.route("/exec")
def execute_script():
    dsname = request.args.get("dsname")
    p = subprocess.Popen(["Path/to/Sas",
            "-sysin", f"pass/to/program/{dsname}.sas"],
            stdout=subprocess.PIPE,
```

```
            stderr=subprocess.PIPE)
    res, err = p.communicate()
    code = p.returncode
    if code != 0:
        return make_response(f"Following error occured {err}", 504)

    return make_response(f"Script {dsname} is called" , 201)
```