COUTURE AI WORKFLOW ORCHESTRATOR

Tech Design and Architecture

Table of Contents

Introduction

Workflow Orchestrator is a platform to programmatically author, schedule, and monitor workflows.

When workflows are defined as code, they become more maintainable, version-able, testable, and collaborative.
Use this orchestrator to author workflows as directed acyclic graphs (DAGs) of tasks. The orchestrator scheduler executes your tasks on an array of workers while following the specified dependencies. Basically, it helps to automate scripts in order to perform tasks.

The rich user interface makes it easy to visualize pipelines running in production, monitor progress, and troubleshoot issues when needed.
This will be a running document to be refined over the time, as new use cases are defined and added to scope.

Installation

Installing workflow orchestrator (with Internet)

- Prerequisites:
  Docker and Docker-compose should be installed

- Fetching the dependencies:
  Get the docker-compose.yml file from shared artifacts

- Getting orchestrator:
  Go to the directory where docker-compose.yml file is located
  Run the following command

```
sudo COUTURE_WORKFLOW_USER=<your name> docker-compose up worker
```

In case, internet access is not available, please follow below steps to pull the docker images:

- Configuring to access private docker registry:
  - Add the following entry to /etc/hosts
    10.144.97.22 CR1
  - Change /etc/docker/daemon.json file to add the following properties
    {"insecure-registries" : ["CR1:5005"] }
    If the file is not present create and add the configuration assuming there are no other settings.
  - Restart the docker daemon after updating the configurations
    sudo systemctl restart docker

- Fetching the dependencies:
  Pull the images from private registry
  docker pull CR1:5005/rabbitmq:3.7-management
  docker pull CR1:5005/mysql:5.7
  docker pull CR1:5005/couture-workflow:1.0
  Tag the images
  docker tag CR1:5005/rabbitmq:3.7-management rabbitmq:3.7-management
  docker tag CR1:5005/mysql:5.7 mysql:5.7
  docker tag CR1:5005/couture-workflow:1.0 couture-workflow
  Get the docker-compose.yml file from shared artifacts

- Getting orchestrator:
  Go to the directory where docker-compose.yml file is located
  Run the following command

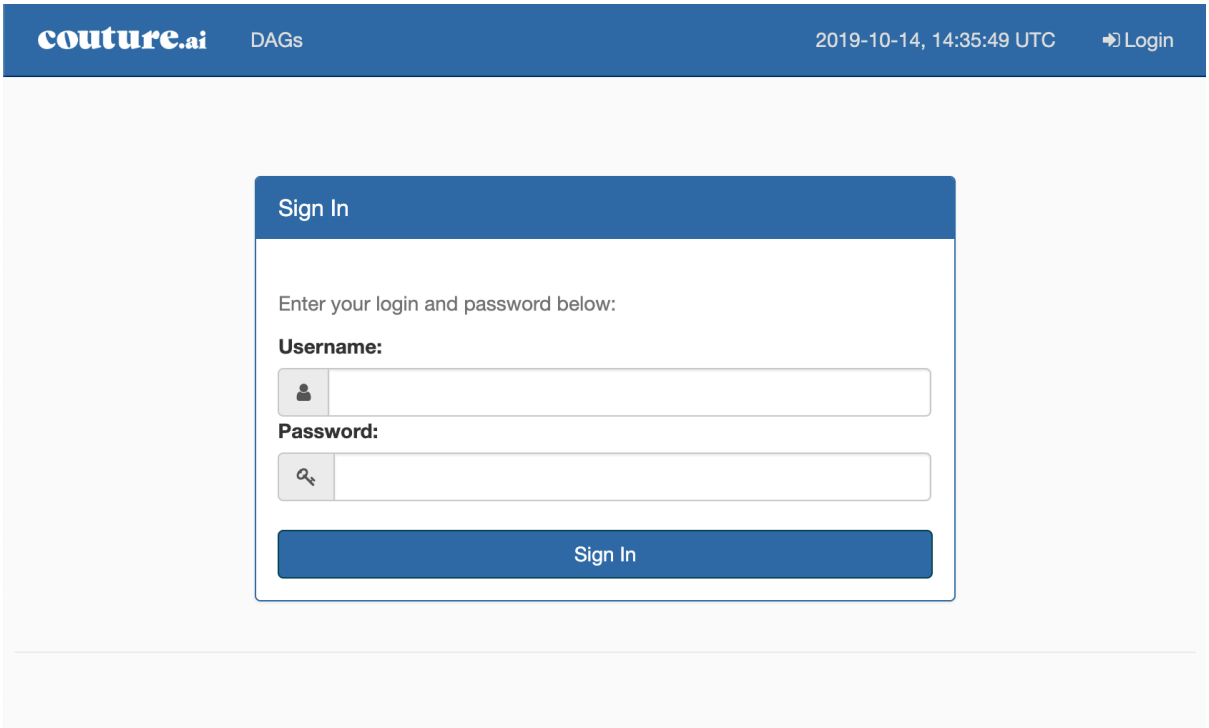
```
sudo COUTURE_WORKFLOW_USER=<your name> docker-compose up worker
```

Note that COUTURE_WORKFLOW_USER is optional and can be used to have personalized tags view.
Go to following URL to start the orchestrator

URL:

http://{HOST}:8080
Where {HOST} refers to the hostname of the server
ex: http://localhost:8080

Login:



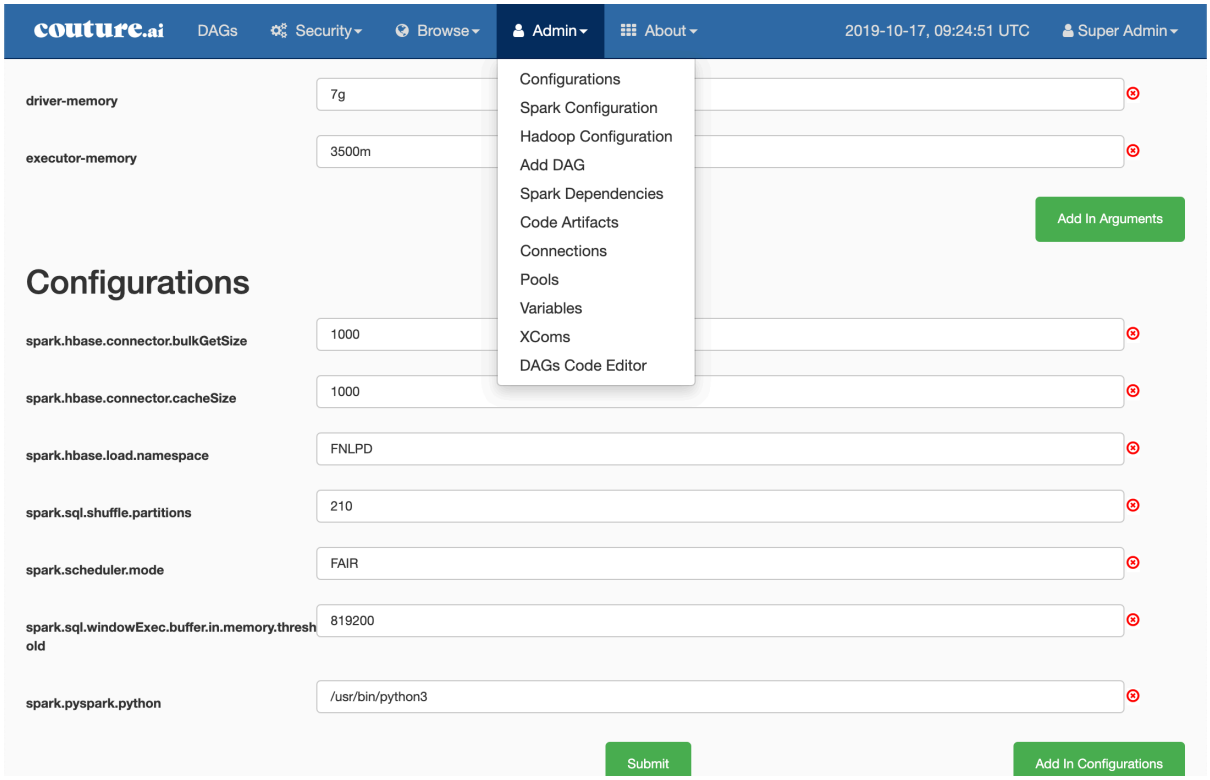
RBAC is used to provide security. The product is shipped with user ‘superadmin’ and password as ‘couture@123’
One can change the password by: Super Admin (Right top) -> Reset my password -> Save

Getting Started

Login as an admin user and configure spark and hadoop configurations as follows.

Spark Configurations and Dependencies:

Easily configure spark jobs by providing options through orchestrator using below steps:
Visit, Admin -> Spark Configuration



Update existing arguments/configurations or add new ones by clicking on 'Add on' buttons. Existing options can be deleted by clicking on delete option, next to the option.

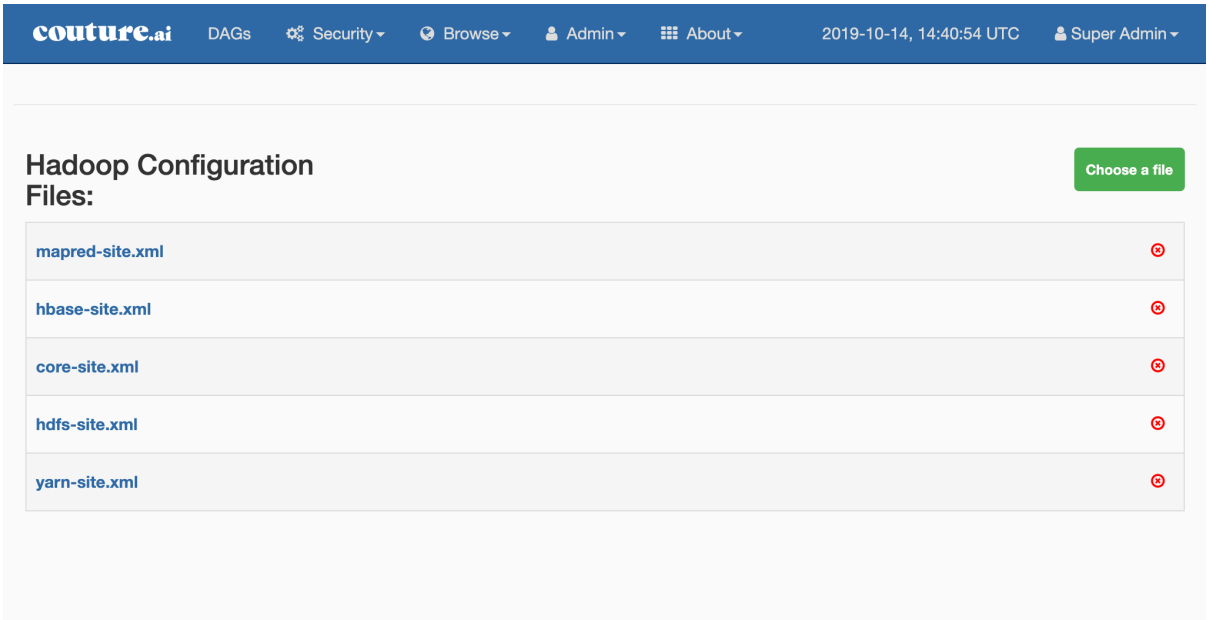
Available jars to include on the driver and executor classpaths are listed under jars option. Similarly, available list of .zip, .egg, or .py files to place on the PYTHONPATH for Python apps are listed under py-files. To upload/delete these jars or python files, visit, Admin -> Spark Dependencies. Upon uploading new jars/python files under 'Spark Dependencies', the same will be added to respective dropdown here.

Note: New changes gets automatically picked up for all the NEWLY triggered spark jobs.

Hadoop Configurations:

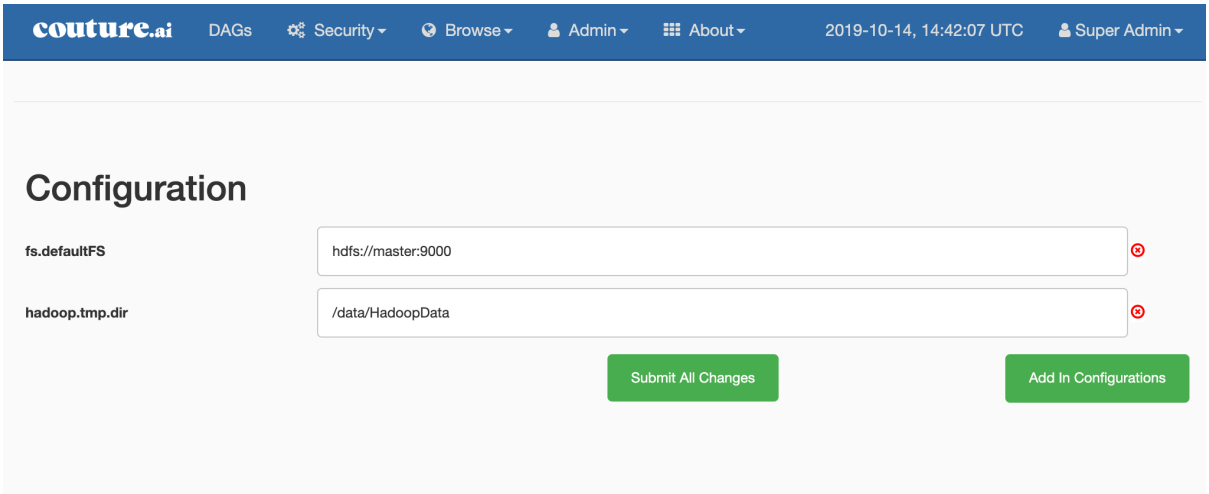
Configurations files required for runtime environment settings of a hadoop cluster can be easily configured through orchestrator using below steps:

Visit, Admin -> Hadoop Configuration



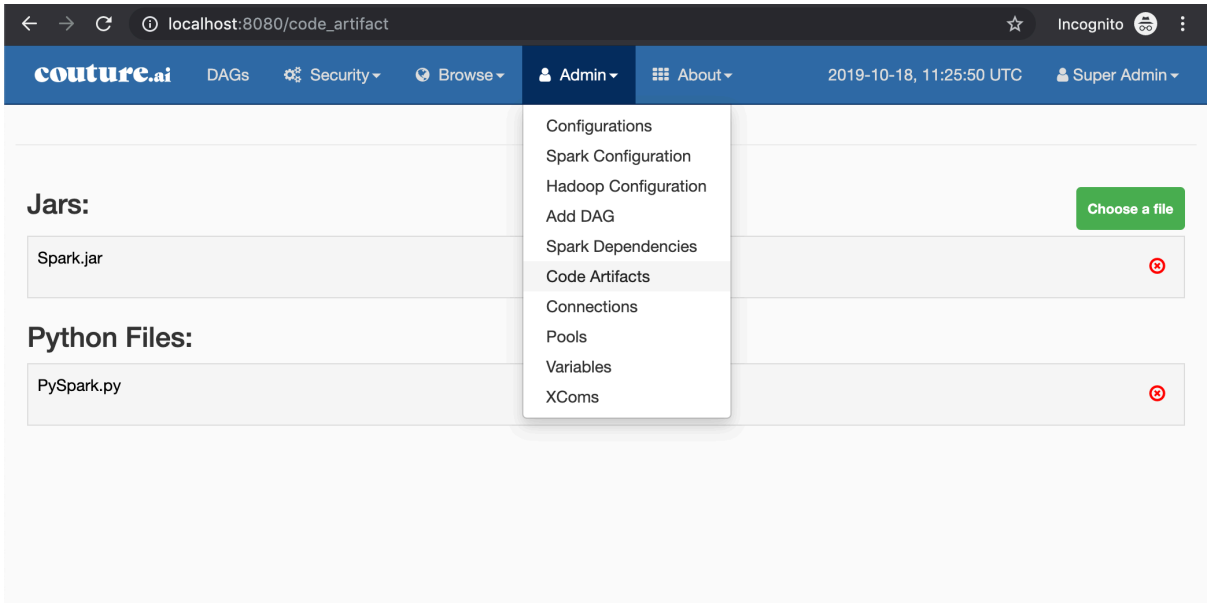
New hadoop conf file can be added using ‘Choose a file’ option. Please note that only XML files are allowed.

The configurations within a file can be updated by clicking on the respective file name.



Code Artifacts:

To upload artifacts i.e. either your jar code for spark jobs or python files for py-spark jobs, visit Admin->Code Artifact.



The artifact can be referred in your dag while creating tasks under 'code_artifact' attribute as shown below:

```
Get_Ratings_History = CouturePySparkOperator(
    task_id='Get_Ratings_History',
    app_name=appName,
    code_artifact='PySpark.py',
    application_arguments=[],
    dag=dag,
    description='Get ratings history'
)
```

What is DAG?:

In mathematics and computer science, a directed acyclic graph, is a finite directed graph with no directed cycles. That is, it consists of finitely many vertices and edges, with each edge directed from one vertex to another, such that there is no way to start at any vertex v and follow a consistently-directed sequence of edges that eventually loops back to v again. Equivalently, a DAG is a directed graph that has a topological ordering, a sequence of the vertices such that every edge is directed from earlier to later in the sequence.

DAGs are composed of tasks which are created by instantiating an operator class.
There are different types of operators available.

Example:

```
from datetime import datetime, timedelta

from airflow import DAG
from airflow.operators import CouturePySparkOperator

appName = 'FunWithPySpark'

default_args = {
   'owner': 'couture',
   'depends_on_past': False,
   'start_date': datetime(2018, 10, 8),
   'retries': 1,
   'retry_delay': timedelta(minutes=5),
```

```
 }

schedule = None
dag = DAG('PySpark', default_args=default_args, catchup=False,
schedule_interval=schedule)

Get_Ratings_Data = CouturePySparkOperator(
    task_id='Get_Ratings_Data',
    app_name=appName,
    code_artifact='Ratings.py',
    application_arguments=[],
    dag=dag,
    description='Dump ratings data in hdfs'
)

Get_Ratings_History = CouturePySparkOperator(
    task_id='Get_Ratings_History',
    app_name=appName,
    code_artifact='History.py',
    application_arguments=[],
    dag=dag,
    description='Get ratings history'
)

Get_Ratings_Data >> Get_Ratings_History
```

Creating a DAG:

- Importing Modules:

An orchestrator pipeline is just a Python script that happens to define a DAG object. Let's start by importing the libraries we will need.

```
# The DAG object; we'll need this to instantiate a DAG

from airflow import DAG

# Operators; we need this to operate!

from airflow.operators.bash_operator import BashOperator
```

To create spark and pyspark jobs, import operators CoutureSparkOperator and CouturePySparkOperator respectively.

```
from airflow.operators import CoutureSparkOperator
```

- Default Arguments:

We're about to create a DAG and some tasks, and we have the choice to explicitly pass a set of arguments to each task's constructor (which would become redundant), or (better!) we can define a dictionary of default parameters that we can use when creating tasks.

```
from datetime import datetime, timedelta

default_args = {

    'owner': 'couture',

    'depends_on_past': False,

    'start_date': datetime(2015, 6, 1),
```

```
    'email': ['couture@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1
}
```

start_date tells since when this DAG should start executing the workflow.
This start_date could belong to the past.

The retries parameter retries to run the DAG X number of times in case of not executing successfully.

Note that you could easily define different sets of arguments that would serve different purposes. An example of that would be to have different settings between a production and development environment.

- Instantiate a DAG

We'll need a DAG object to nest our tasks into. Here we pass a string that defines the dag_id, which serves as a unique identifier for your DAG. We also pass the default argument dictionary that we just defined and define a schedule_interval of 1 day for the DAG.

```
dag = DAG('dag',
       default_args=default_args,
       schedule_interval=timedelta(days=1))
```

Workflow consists of a scheduler which is monitoring process that runs all the time and triggers task execution based on schedule_interval and execution_date.

- Tasks

Tasks are generated when instantiating operator objects. An object instantiated from an operator is called a constructor. The first argument task_id acts as a unique identifier for the task.

Task description describing its functionality can be added using 'description' attribute.

```
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag,
    description='Print Date')
```

```
operator_task = CoutureSparkOperator(
       task_id='operator_task',
```

```
      dag=dag,
      app_name='Couture',
      class_path='ai.couture.MainClass',
      code_artifact= ‘couture.jar’,
      application_arguments=['inputData', ‘/outputData‘],
      description=''
)
```

- Setting up Dependencies

We have tasks *t1*, *t2* and *t3* that do not depend on each other. Here’s a few ways you can define dependencies between them:

```
t1.set_downstream(t2)
```

```
# This means that t2 will depend on t1
# running successfully to run.
# It is equivalent to:
t2.set_upstream(t1)
# The bit shift operator can also be
# used to chain operations:
t1 >> t2
```

```
# And the upstream dependency with the
# bit shift operator:
t2 << t1
```

```
# Chaining multiple dependencies becomes
# concise with the bit shift operator:
t1 >> t2 >> t3
```

```
# A list of tasks can also be set as
# dependencies. These operations
# all have the same effect:
t1.set_downstream([t2, t3])
t1 >> [t2, t3]
[t2, t3] << t1
```

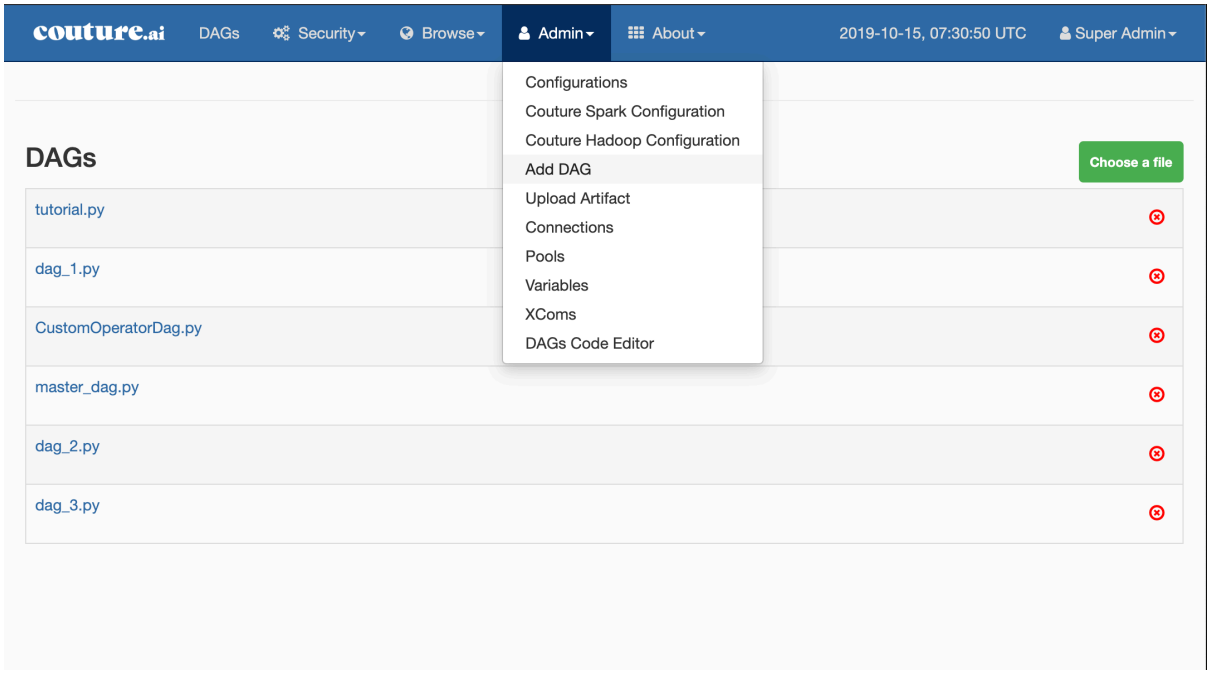
Note that when executing your script, orchestrator will raise exceptions when it finds cycles in your DAG or when a dependency is referenced more than once.

Adding a new DAG:

After creating a <dag>.py file, add a new dag using below steps:

- Visit, Admin -> Add DAG
- 'Choose a file' and select *.py file.

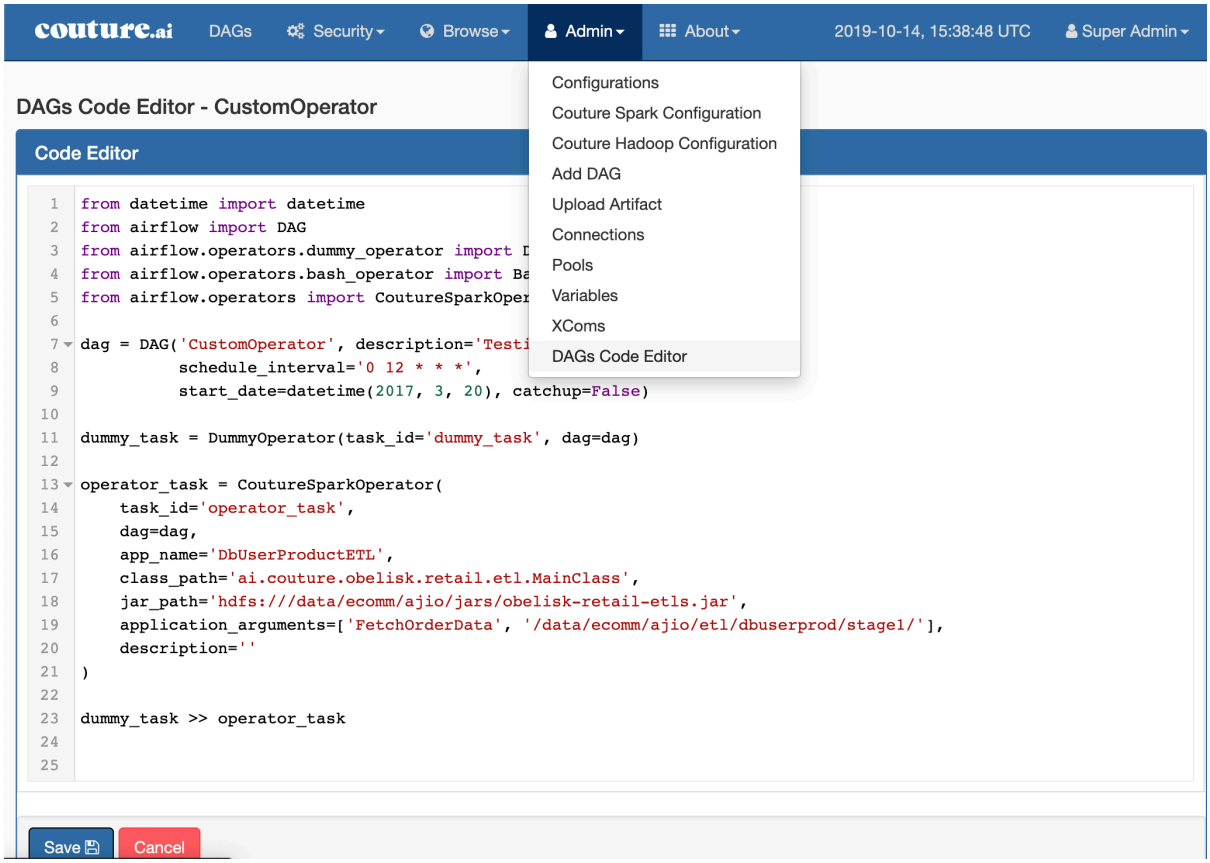
Note that only *.py format is supported.



Edit DAG code:

If you have an existing dag added to the orchestrator, you can edit the same using below steps:

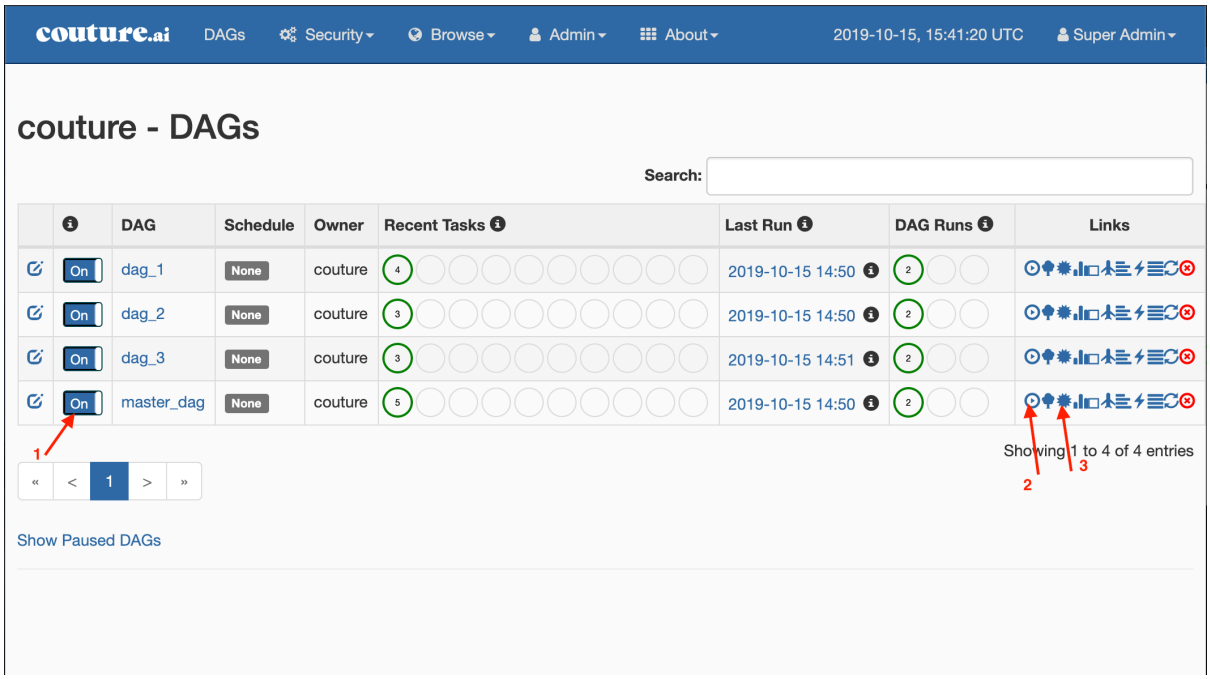
- Visit, Admin -> DAGs Code Editor
- Make the required changes and Save



Run DAG:

The **Scheduler** is responsible at what time DAG should be triggered. By default all the dags are paused to be scheduled.

Also, please note that all the paused dags are hidden by default. To un-pause the dag, click on the 'Show Paused DAGs' and 'ON' the required dag.

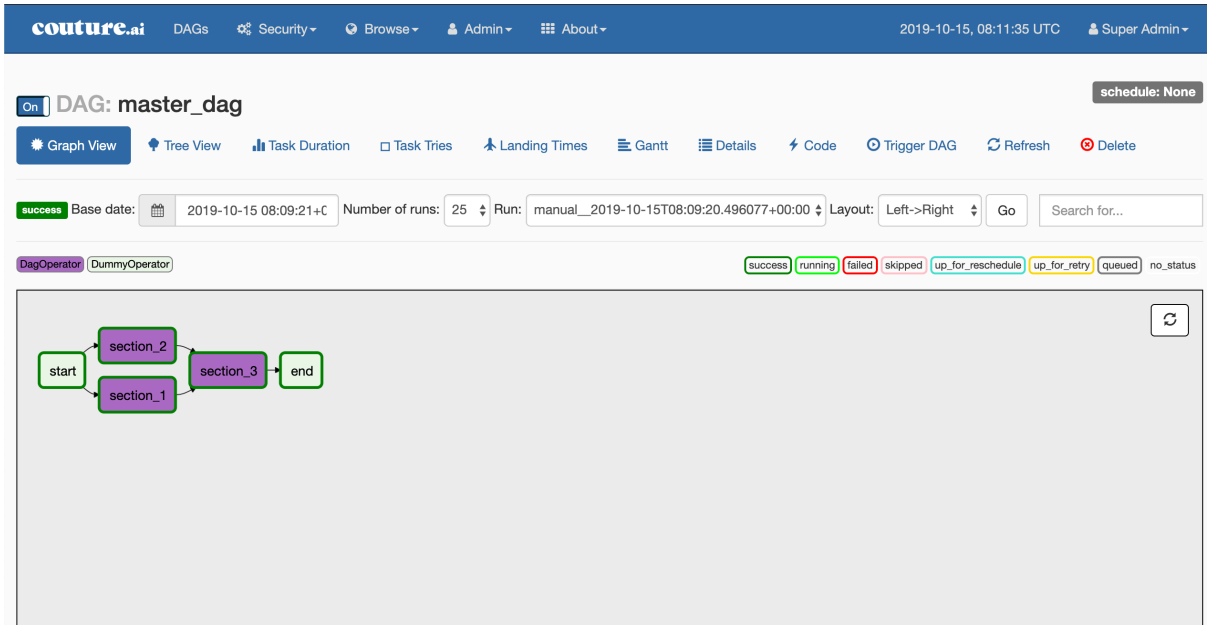


In order to start a DAG Run, first turn the workflow on (arrow **1**), then click the **Trigger Dag** button (arrow **2**) and finally, click on the **Graph View** (arrow **3**) to see the progress of the run.

The graph view can be reloaded until all the tasks reach the status **Success**. You can also click on a task and then click **View Log** to see the log of task instance run.

Master DAG:

An end-to-end pipeline can be designed by creating master dag, i.e. DAG of dags



- Importing Modules:

Let's start by importing the libraries we will need.

```
from airflow.operators.dag_operator import DagOperator
```

- Linking DAGs

An object should be instantiated from DagOperator. The first argument task_id acts as a unique identifier for the task.

```
section_1 = DagOperator(
    task_id='section_1',
    run_dag_id="dag_1",
    python_callable=conditionally_trigger,
    params={'condition_param': True, 'message': 'Hii there!!'},
    dag=dag,
)
```

- Conditionally trigger DAGs

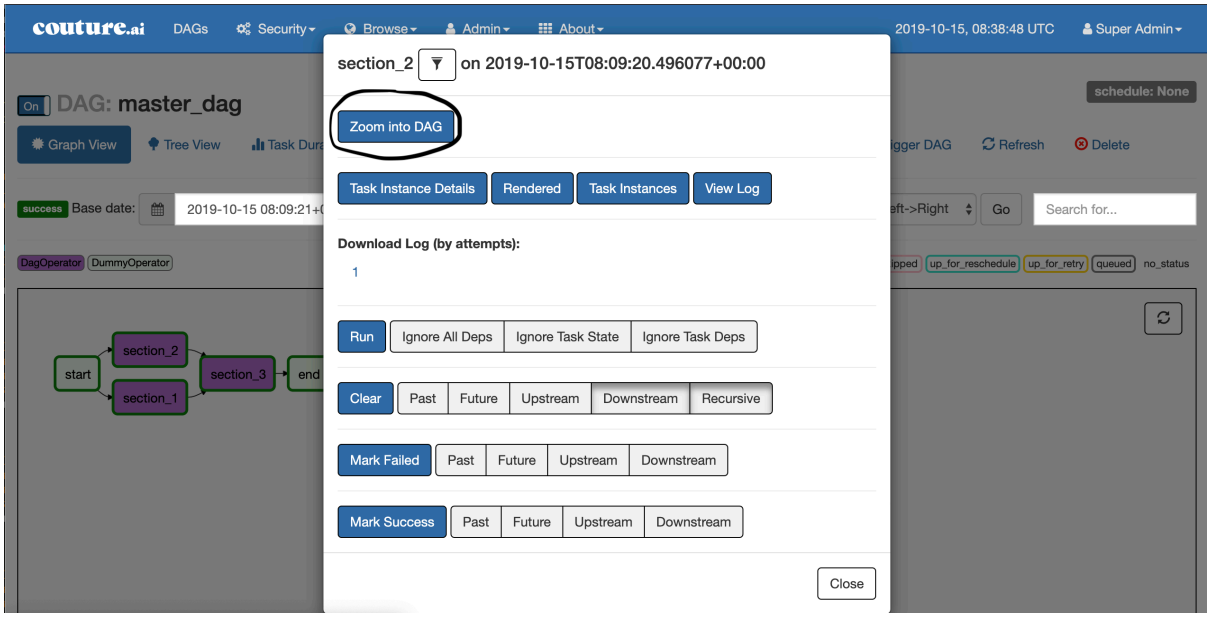
A condition can be set as to whether or not to trigger the remote DAG, by defining a python function as below

```
def conditionally_trigger(context, dag_run_obj):
    """This function decides whether or not to Trigger the remote DAG"""
    if context['params']['condition_param']:
        dag_run_obj.payload = {'message': context['params']['message']}
        return dag_run_obj
```

Note: All the dags which are part of master dag, should be 'ON'

- Zoom into DAG

The dags which are part of master dag, can be visited by clicking on them and then click Zoom into DAG.

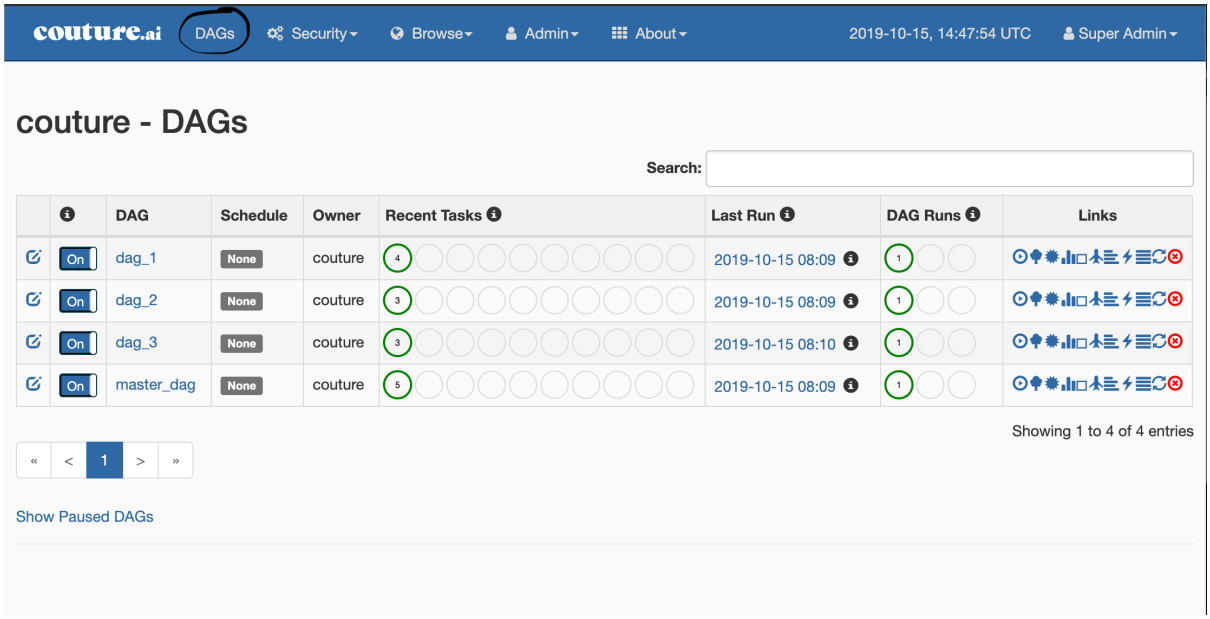


User Interface

The Workflow UI makes it easy to monitor and troubleshoot your data pipelines. Here's a quick overview of some of the features and visualizations you can find in the Workflow UI.

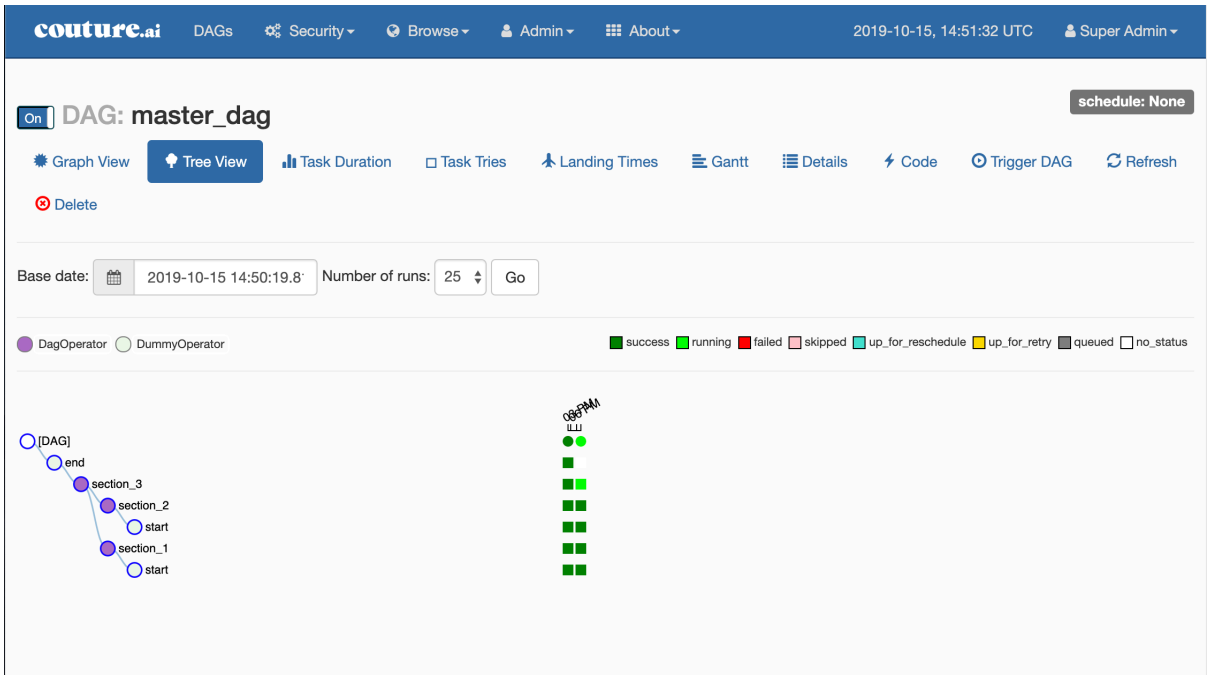
DAGs View

List of the DAGs in your environment, and a set of shortcuts to useful pages. You can see exactly how many tasks succeeded, failed, or are currently running at a glance.



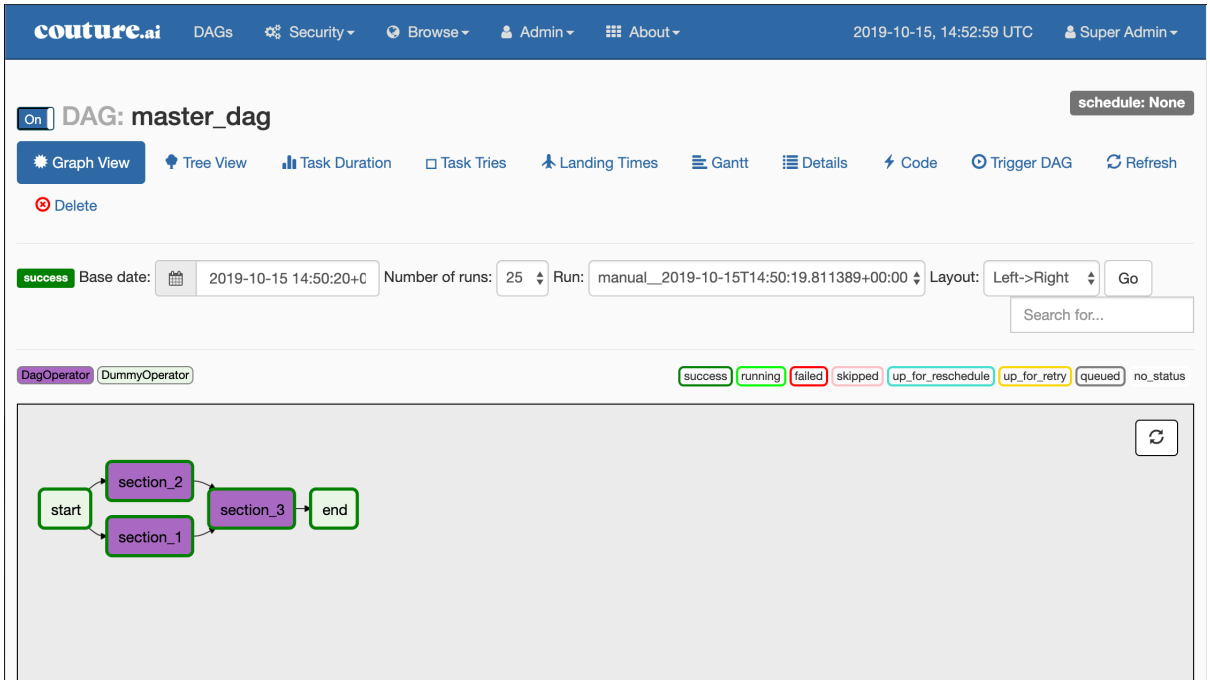
Tree View

A tree representation of the DAG that spans across time. If a pipeline is late, you can quickly see where the different steps are and identify the blocking ones.



Graph View

The graph view is perhaps the most comprehensive. Visualize your DAG's dependencies and their current status for a specific run.



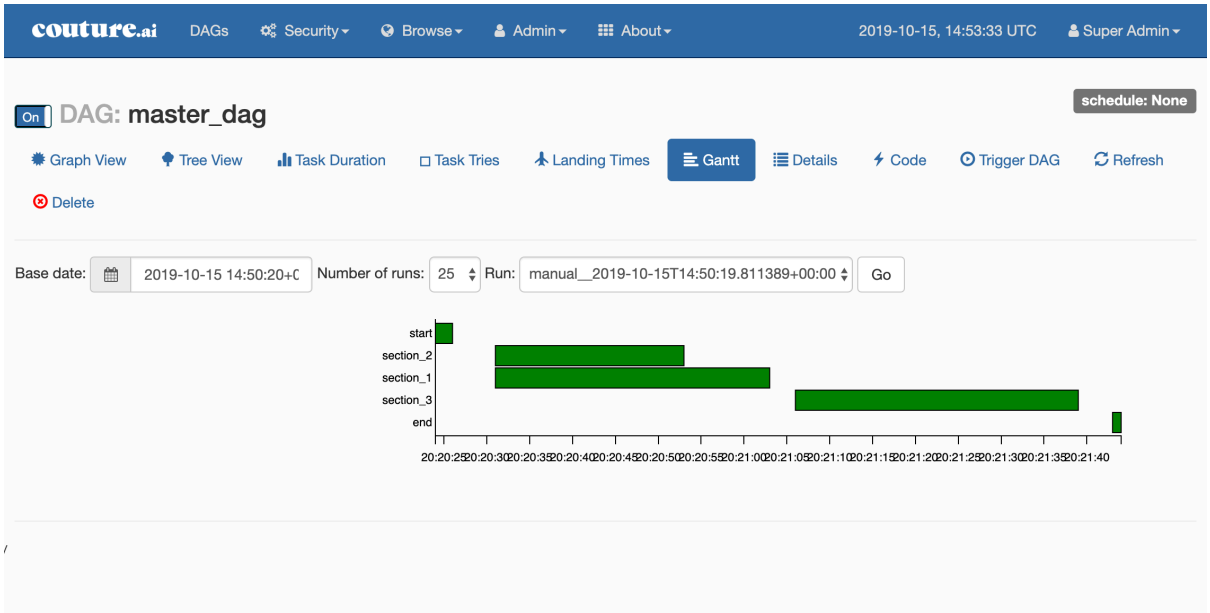
Here is the screenshot of the DAG executed. You can see rectangular boxes representing a task. You can also see different color boxes on the top right of the greyed box, named: success, running, failed etc, representing status of the task.

Variable View

The variable view allows you to list, create, edit or delete the key-value pair of a variable used during jobs. Value of a variable will be hidden if the key contains any words in ('password', 'secret', 'passwd', 'authorization', 'api_key', 'apikey', 'access_token') by default, but can be configured to show in clear-text.

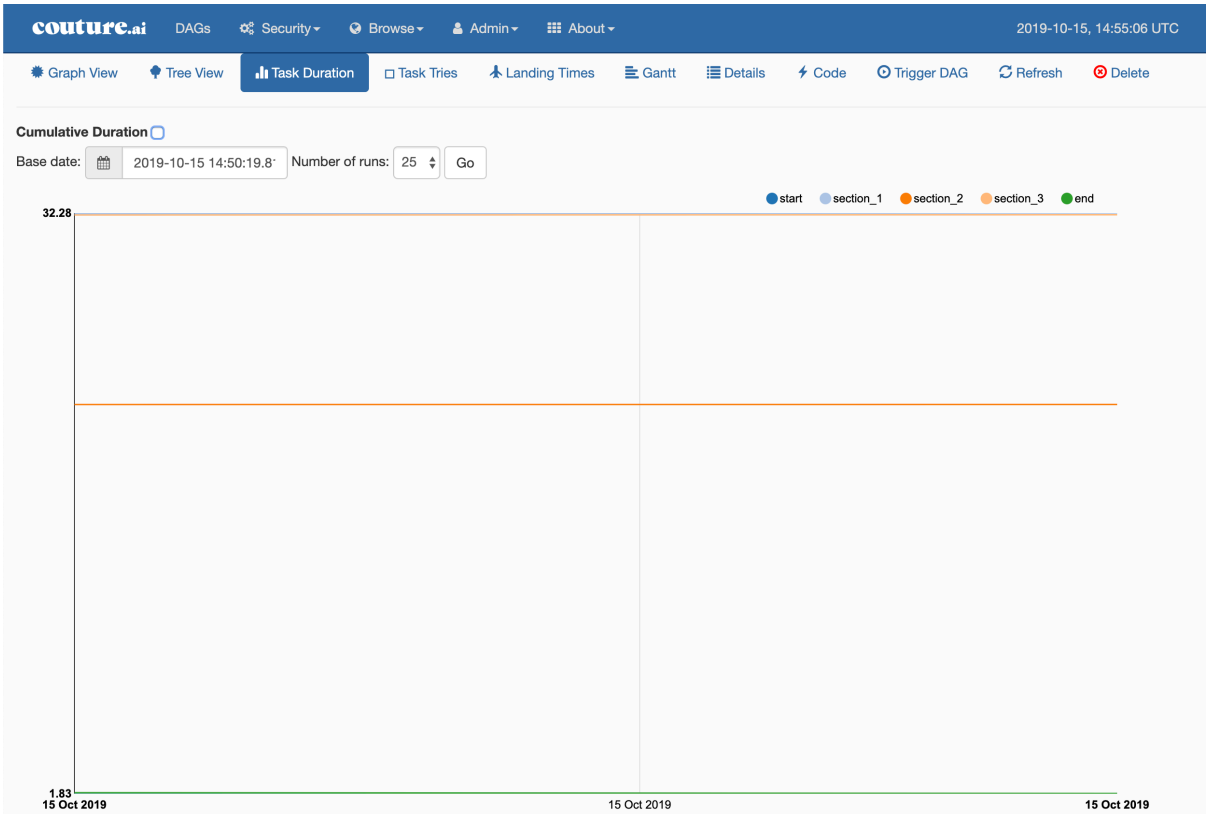
Gantt Chart

The Gantt chart lets you analyse task duration and overlap. You can quickly identify bottlenecks and where the bulk of the time is spent for specific DAG runs.



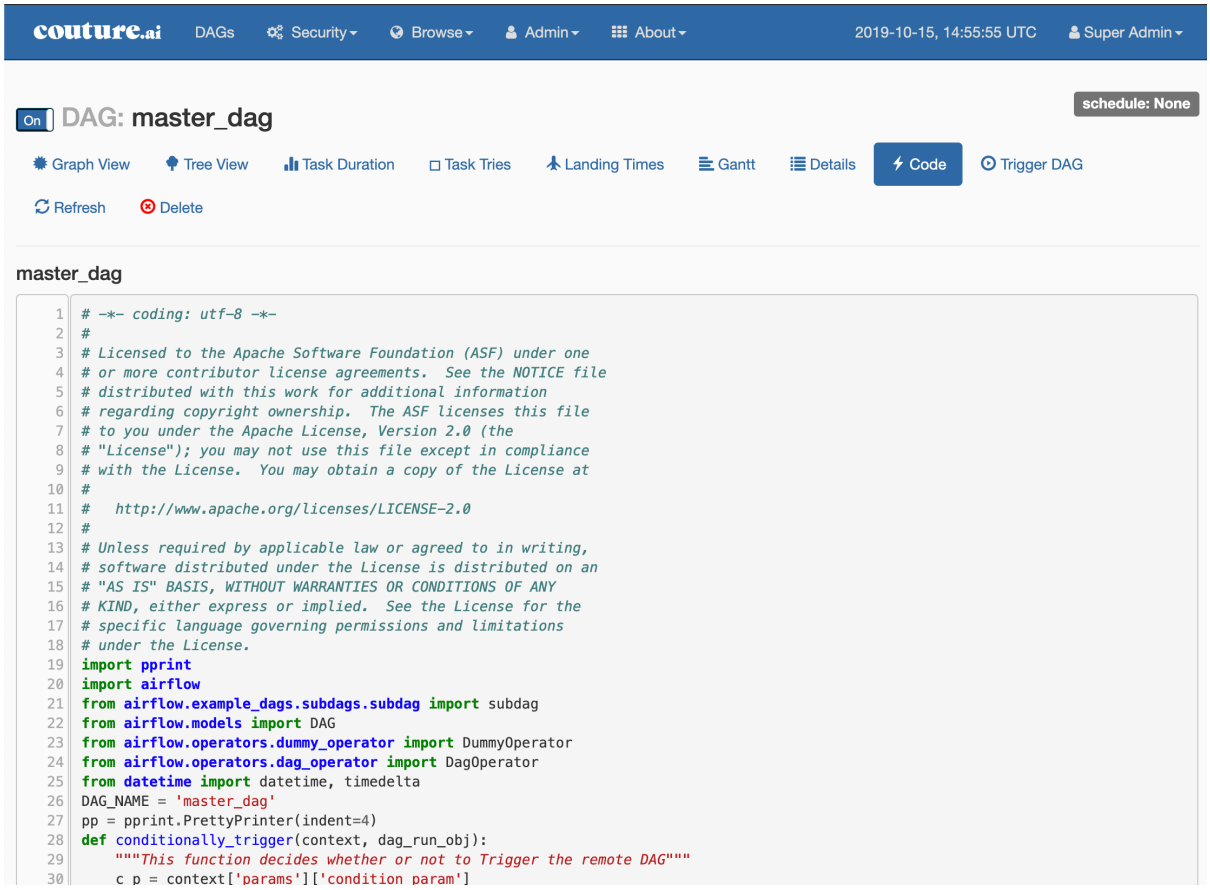
Task Duration

The duration of your different tasks over the past N runs. This view lets you find outliers and quickly understand where the time is spent in your DAG over many runs.



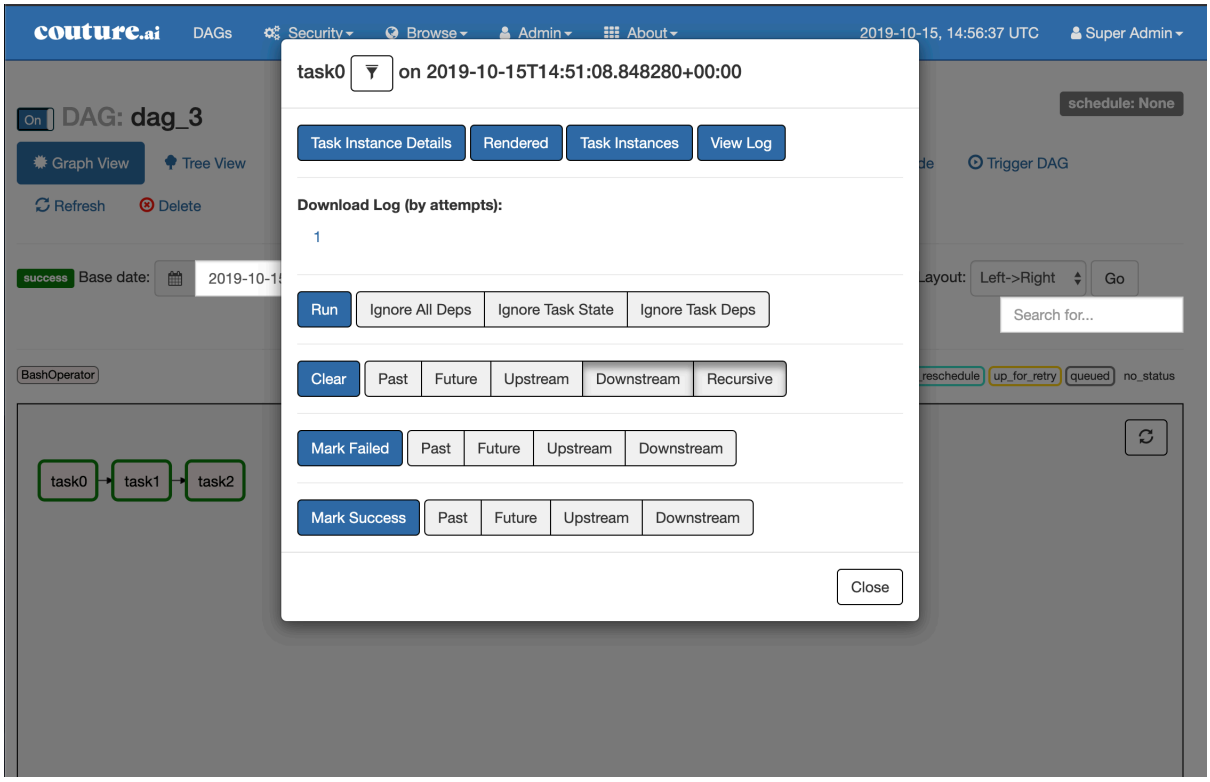
Code View

Transparency is everything. While the code for your pipeline is in source control, this is a quick way to get to the code that generates the DAG and provide yet more context.



Task Instance Context Menu

From the pages seen above (tree view, graph view, gantt, …), it is always possible to click on a task instance, and get to this rich context menu that can take you to more detailed metadata, and perform some actions.



Users

Role Based Access Control

Workflow provides role-based access control (RBAC), allowing you to configure varying levels of access across all Users within your Workspace.

There are five roles created for Workflow by default: Admin, User, Op, Viewer, and Public.

- Admin

Admin users have all possible permissions, including granting or revoking permissions from other users.

- Public

Public users (anonymous) don't have any permissions.

- Viewer

Viewer users have limited viewer permissions

```
VIEWER_PERMS = {
    'menu_access',
    'can_index',
    'can_list',
    'can_show',
    'can_chart',
    'can_dag_stats',
    'can_dag_details',
    'can_task_stats',
    'can_code',
    'can_log',
    'can_get_logs_with_metadata',
    'can_tries',
    'can_graph',
    'can_tree',
    'can_task',
    'can_task_instances',
    'can_xcom',
    'can_gantt',
    'can_landing_times',
    'can_duration',
    'can_blocked',
    'can_rendered',
    'can_pickle_info',
    'can_version',
}
```

on limited web views

```
VIEWER_VMS = {
    'Airflow',
    'DagModelView',
    'Browse',
    'DAG Runs',
    'DagRunModelView',
    'Task Instances',
```

```
    'TaskInstanceModelView',
    'SLA Misses',
    'SlaMissModelView',
    'Jobs',
    'JobModelView',
    'Logs',
    'LogModelView',
    'Docs',
    'Documentation',
    'GitHub',
    'About',
    'Version',
    'VersionView',
}
```

- User

User users have Viewer permissions plus additional user permissions

```
USER_PERMS = {
    'can_dagrun_clear',
    'can_run',
    'can_trigger',
    'can_add',
    'can_edit',
    'can_delete',
    'can_paused',
    'can_refresh',
    'can_success',
    'muldelete',
    'set_failed',
    'set_running',
    'set_success',
    'clear',
    'can_clear',
}
```

on User web views which is the same as Viewer web views.

- Op

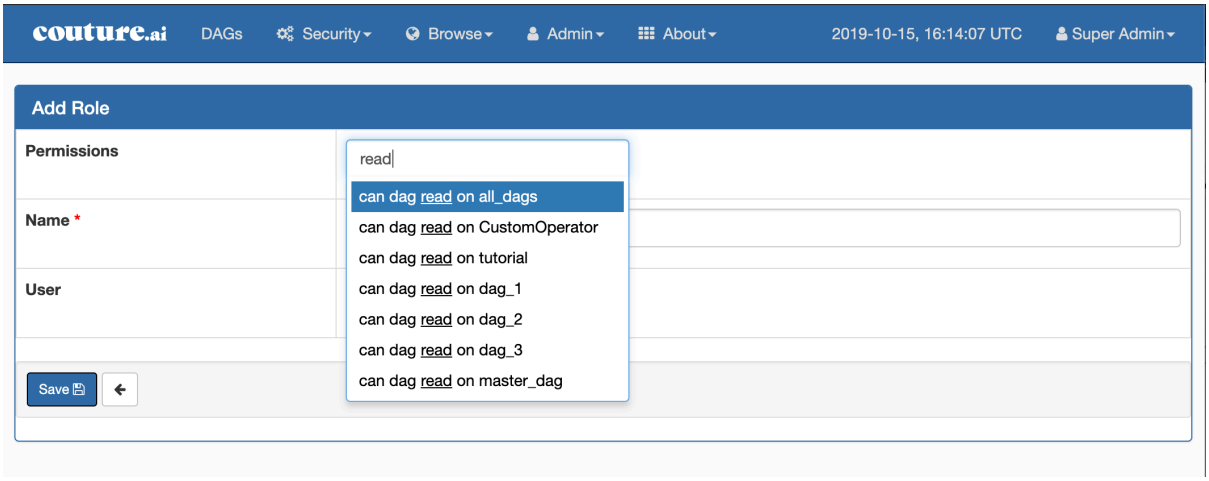
Op users have User permissions plus additional op permissions

```
OP_PERMS = {
    'can_conf',
    'can_varimport',
}
```

on User web views plus these additional op web views

```
OP_VMS = {
    'Admin',
    'Configurations',
    'ConfigurationView',
    'Connections',
    'ConnectionModelView',
    'Pools',
    'PoolModelView',
    'Variables',
    'VariableModelView',
    'XComs',
    'XComModelView',
}
```

The Admin could create a specific role which is only allowed to read/write certain DAGs. To configure a new role, go to Security tab and click List Roles in the new UI.

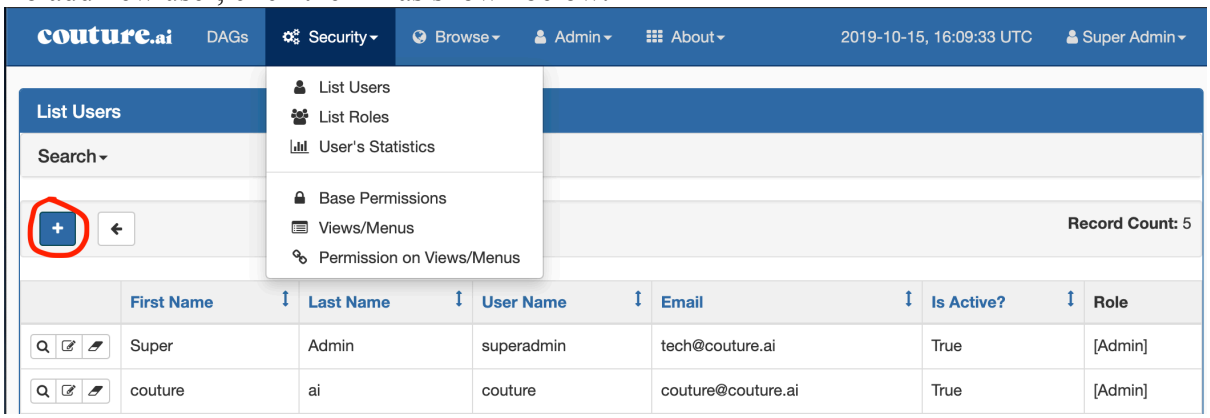


The image shows the creation of a role which can only read all dags.

User creation

To add new user, go to Security tab and click List Users in the UI. This can also be integrated with Kerberos or LDAP.

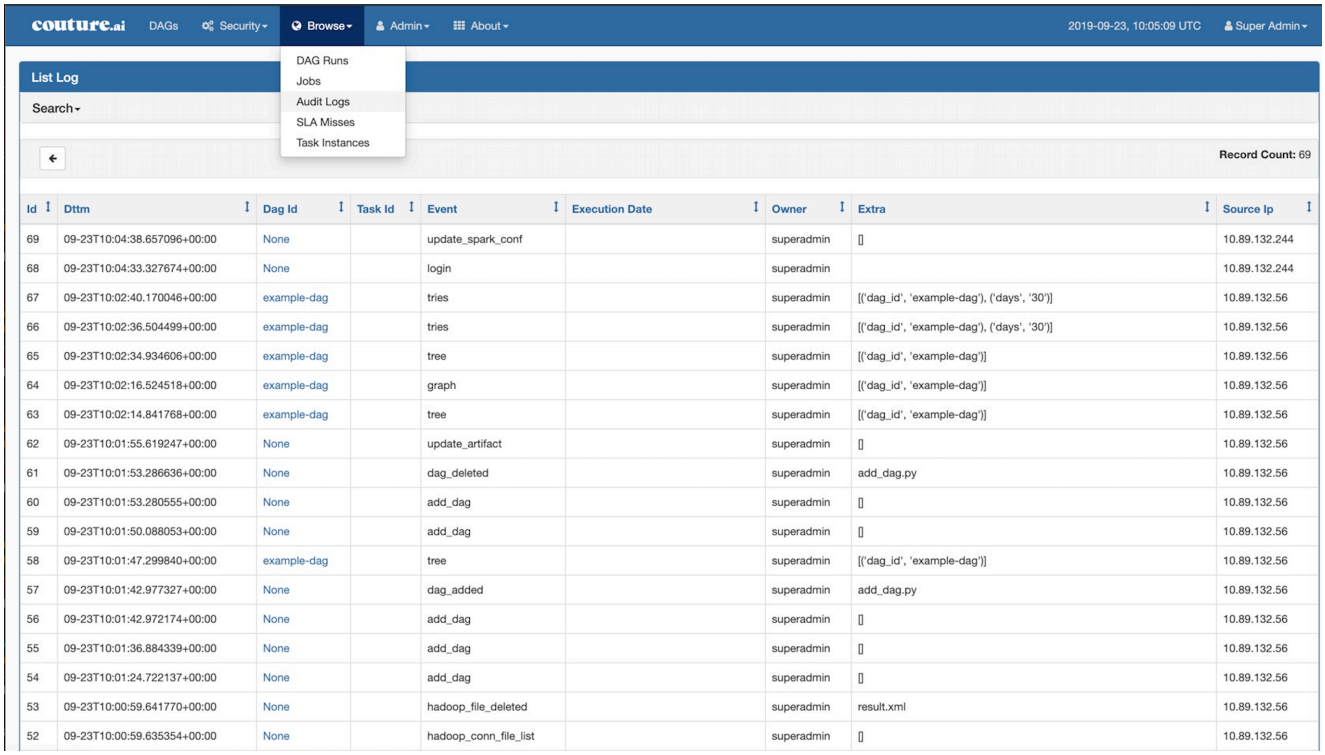
To add new user, click the '+' as shown below.



Note only admin can create new users.

Access Audit Logging

User journey can be easily tracked by audit logs. Audit logs can be viewed by going to Browse and clicking on Audit Logs. All the activities/events are captured. Also, the source IP address from where the event occurred is captured as shown below.

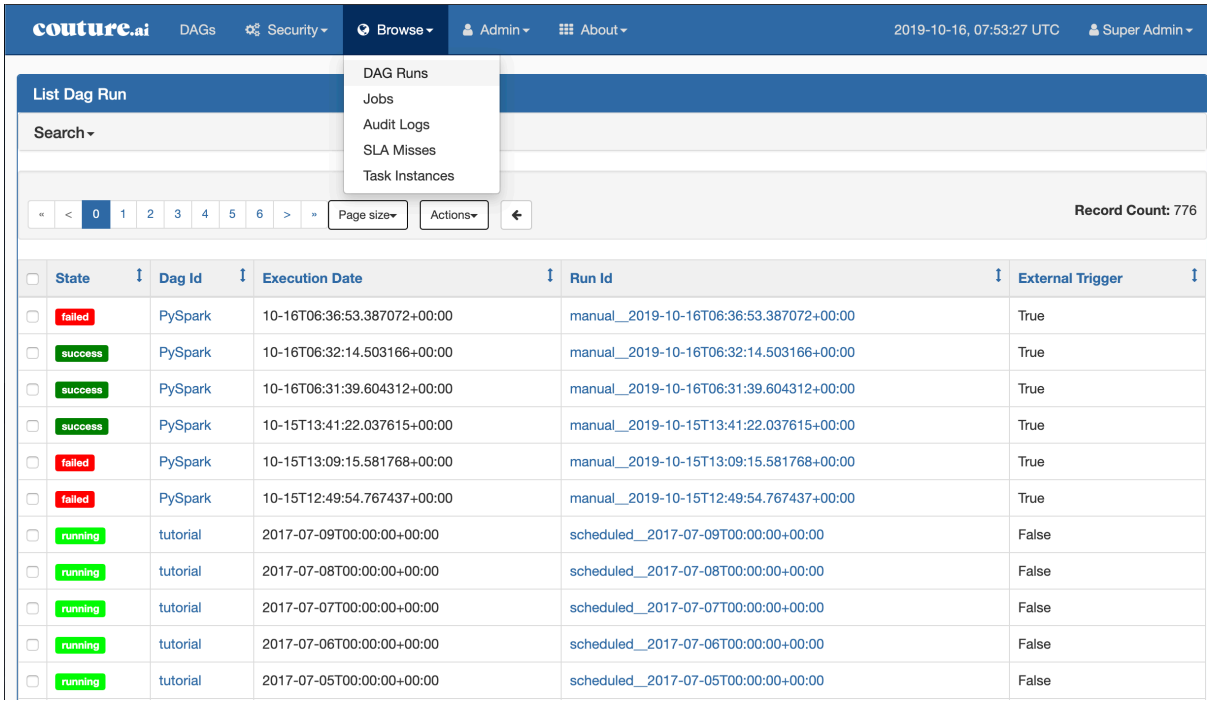


Id	Dttm	Dag Id	Task Id	Event	Execution Date	Owner	Extra	Source Ip
69	09-23T10:04:38.657096+00:00	None		update_spark_conf		superadmin	[]	10.89.132.244
68	09-23T10:04:33.327674+00:00	None		login		superadmin		10.89.132.244
67	09-23T10:02:40.170046+00:00	example-dag		tries		superadmin	[('dag_id', 'example-dag'), ('days', '30')]	10.89.132.56
66	09-23T10:02:36.504499+00:00	example-dag		tries		superadmin	[('dag_id', 'example-dag'), ('days', '30')]	10.89.132.56
65	09-23T10:02:34.934606+00:00	example-dag		tree		superadmin	[('dag_id', 'example-dag')]	10.89.132.56
64	09-23T10:02:16.524518+00:00	example-dag		graph		superadmin	[('dag_id', 'example-dag')]	10.89.132.56
63	09-23T10:02:14.841768+00:00	example-dag		tree		superadmin	[('dag_id', 'example-dag')]	10.89.132.56
62	09-23T10:01:55.619247+00:00	None		update_artifact		superadmin	[]	10.89.132.56
61	09-23T10:01:53.286636+00:00	None		dag_deleted		superadmin	add_dag.py	10.89.132.56
60	09-23T10:01:53.280555+00:00	None		add_dag		superadmin	[]	10.89.132.56
59	09-23T10:01:50.088053+00:00	None		add_dag		superadmin	[]	10.89.132.56
58	09-23T10:01:47.299840+00:00	example-dag		tree		superadmin	[('dag_id', 'example-dag')]	10.89.132.56
57	09-23T10:01:42.977327+00:00	None		dag_added		superadmin	add_dag.py	10.89.132.56
56	09-23T10:01:42.972174+00:00	None		add_dag		superadmin	[]	10.89.132.56
55	09-23T10:01:36.884339+00:00	None		add_dag		superadmin	[]	10.89.132.56
54	09-23T10:01:24.722137+00:00	None		add_dag		superadmin	[]	10.89.132.56
53	09-23T10:00:59.641770+00:00	None		hadoop_file_deleted		superadmin	result.xml	10.89.132.56
52	09-23T10:00:59.635354+00:00	None		hadoop_conn_file_list		superadmin	[]	10.89.132.56

DAG Runs

A DagRun is the instance of a DAG that will run at a time. When it runs, all task inside it will be executed.

DAG Runs tell how many times a certain DAG has been executed. **Recent Tasks** tells which task out of many tasks within a DAG currently running and what's the status of it.



State	Dag Id	Execution Date	Run Id	External Trigger
failed	PySpark	10-16T06:36:53.387072+00:00	manual__2019-10-16T06:36:53.387072+00:00	True
success	PySpark	10-16T06:32:14.503166+00:00	manual__2019-10-16T06:32:14.503166+00:00	True
success	PySpark	10-16T06:31:39.604312+00:00	manual__2019-10-16T06:31:39.604312+00:00	True
success	PySpark	10-15T13:41:22.037615+00:00	manual__2019-10-15T13:41:22.037615+00:00	True
failed	PySpark	10-15T13:09:15.581768+00:00	manual__2019-10-15T13:09:15.581768+00:00	True
failed	PySpark	10-15T12:49:54.767437+00:00	manual__2019-10-15T12:49:54.767437+00:00	True
running	tutorial	2017-07-09T00:00:00+00:00	scheduled__2017-07-09T00:00:00+00:00	False
running	tutorial	2017-07-08T00:00:00+00:00	scheduled__2017-07-08T00:00:00+00:00	False
running	tutorial	2017-07-07T00:00:00+00:00	scheduled__2017-07-07T00:00:00+00:00	False
running	tutorial	2017-07-06T00:00:00+00:00	scheduled__2017-07-06T00:00:00+00:00	False
running	tutorial	2017-07-05T00:00:00+00:00	scheduled__2017-07-05T00:00:00+00:00	False

Connections

The connection information to external systems is stored in the workflow metadata database and managed in the UI (Menu -> Admin -> Connections). A conn_id is defined there and hostname / login / password / schema information attached to it. Pipelines can simply refer to the centrally managed conn_id without having to hard code any of this information anywhere.

Many connections with the same conn_id can be defined, workflow will choose one connection randomly, allowing for some basic load balancing and fault tolerance when used in conjunction with retries.

Variables and XComs

XComs

XComs let tasks exchange messages, allowing more nuanced forms of control and shared state. The name is an abbreviation of "cross-communication". Any object that can be pickled can be used as an XCom value, so users should make sure to use objects of appropriate size.

XComs can be "pushed" (sent) or "pulled" (received). When a task pushes an XCom, it makes it generally available to other tasks. Tasks can push XComs at any time by calling the xcom_push() method. In addition, if a task returns a value (either from its Operator's execute() method, or from a PythonOperator's python_callable function), then an XCom containing that value is automatically pushed.

Tasks call xcom_pull() to retrieve XComs, optionally applying filters based on criteria like key, source task_ids, and source dag_id. By default, xcom_pull() filters for the keys that are automatically given to XComs when they are pushed by being returned from execute functions (as opposed to XComs that are pushed manually).

If xcom_pull is passed a single string for task_ids, then the most recent XCom value from that task is returned; if a list of task_ids is passed, then a corresponding list of XCom values is returned. If you set provide_context=True, the returned value of the function is pushed itself into XCOM which itself is nothing but a Db table.

```
# inside a PythonOperator called 'pushing_task'
def push_function():
    return value

# inside another PythonOperator
```

```
def pull_function(task_instance):
    value = task_instance.xcom_pull(task_ids='pushing_task')
```

Note that XComs are similar to Variables, but are specifically designed for inter-task communication rather than global settings.

Example:

```
def parse_recipes(**kwargs):
  return 'RETURNS parse_recipes'def download_image(**kwargs):
    ti = kwargs['ti']
    v1 = ti.xcom_pull(key=None, task_ids='parse_recipes')print('Printing Task 1
values in Download_image')
    print(v1)
    return 'download_image'
```

The first task has no such changes other than providing **kwargs which let share key/value pairs. The other is setting provide_context=True in each operator to make it *XCom compatible*. For instance:

```
opr_parse_recipes = PythonOperator(task_id='parse_recipes',
python_callable=parse_recipes, provide_context=True)
```

The download_image will have the following changes:

```
def download_image(**kwargs):
    ti = kwargs['ti']
    v1 = ti.xcom_pull(key=None, task_ids='parse_recipes')print('Printing Task 1
values in Download_image')
    print(v1)
    return 'download_image'
```

The first line is ti=kwargs['t1']` get the instances details by access ti key. In case you wonder why this has been done. If you print kwargs it prints something like below in which you can find keys like *t1*, *task_instance* etc to get a task's pushed value.

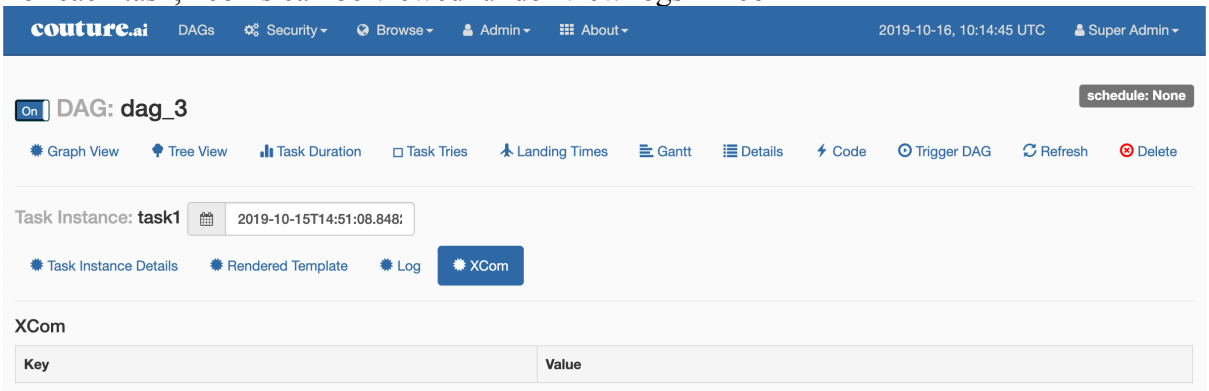
```
{
  'dag': <DAG: parsing_recipes>,
  'ds': '2018-10-02',
  'next_ds': '2018-10-02',
  'prev_ds': '2018-10-02',
  'ds_nodash': '20181002',
  'ts': '2018-10-02T09:56:05.289457+00:00',
  'ts_nodash': '20181002T095605.289457+0000',
  'yesterday_ds': '2018-10-01',
  'yesterday_ds_nodash': '20181001',
  'tomorrow_ds': '2018-10-03',
  'tomorrow_ds_nodash': '20181003',
  'END_DATE': '2018-10-02',
```

```
  'end_date': '2018-10-02',
  'dag_run': <DagRunparsing_recipes@2018-10-0209: 56: 05.289457+00: 00:
manual__2018-10-02T09: 56: 05.289457+00: 00,
  externallytriggered: True>,
  'run_id': 'manual__2018-10-02T09:56:05.289457+00:00',
  'execution_date': <Pendulum[
    2018-10-02T09: 56: 05.289457+00: 00
  ]>,
  'prev_execution_date': datetime.datetime(2018,
  10,
  2,
  9,
  56,
  tzinfo=<TimezoneInfo[
    UTC,
    GMT,
    +00: 00: 00,
    STD
  ]>),
  'next_execution_date': datetime.datetime(2018,
  10,
  2,
  9,
  58,
  tzinfo=<TimezoneInfo[
    UTC,
    GMT,
    +00: 00: 00,
    STD
  ]>),
  'latest_date': '2018-10-02',
  'params': {

  },
  'tables': None,
  'task': <Task(PythonOperator): download_image>,
  'task_instance': <TaskInstance: parsing_recipes.download_image2018-10-02T09: 56:
05.289457+00: 00[
    running
  ]>,
  'ti': <TaskInstance: parsing_recipes.download_image2018-10-02T09: 56:
05.289457+00: 00[
    running
  ]>,
  'task_instance_key_str': 'parsing_recipes__download_image__20181002',
  'test_mode': False,
  'var': {
    'value': None,
    'json': None
  },
  'inlets': [

  ],
  'outlets': [

  ],
  'templates_dict': None
}
```

Next, xcom_pull can be called to put the certain task's returned value. In my the task id is parse_recipes:

v1 = ti.xcom_pull(key=None, task_ids='parse_recipes')

For each task, xcoms can be viewed under view logs-> xcom



Variables

Variables are a generic way to store and retrieve arbitrary content or settings as a simple key value store within workflow. Variables can be listed, created, updated and deleted from the UI (Admin -> Variables. In addition, json settings files can be bulk uploaded through the UI. While your pipeline code definition and most of your constants and variables should be defined in code and stored in source control, it can be useful to have some variables or configuration items accessible and modifiable through the UI.

```
from airflow.models import Variable
foo = Variable.get("foo")
bar = Variable.get("bar", deserialize_json=True)
baz = Variable.get("baz", default_var=None)
```

The second call assumes json content and will be deserialized into bar. Note that Variable is a sqlalchemy model and can be used as such. The third call uses the default_var parameter with the value None, which either returns an existing value or None if the variable isn't defined. The get function will throw a KeyError if the variable doesn't exist and no default is provided.

Variables can be pushed and pulled in a similar fashion to XComs:

```
config = Variable.get("db_config")
set_confg = Variable.set(db_config)
```

Note: Although variables are Fernet key encrypted in the database, they are accessible in the UI and therefore should not be used to store passwords or other sensitive data.

When to use each?

In general - since XComs are meant to be used to communicate between tasks and store the "conditions" that led to that value being created, they should be used for values that are going to be changing each time a workflow runs.

Variables on the other hand are much more natural places for constants like a list of tables that need to be synced, a configuration file that needs to be pulled from, or a list of IDs to dynamically generate tasks from.

Both can be very powerful where appropriate, but can also be dangerous if misused.

SLAs

Service Level Agreements, or time by which a task or DAG should have succeeded, can be set at a task level as a timedelta. If one or many instances have not succeeded by that time, an alert email is sent detailing the list of tasks that missed their SLA. The event is recorded in the database and made available in the web UI under Browse->SLA Misses where events can be analyzed and documented.

SLAs can be configured for scheduled tasks by using the sla parameter. In addition to sending alerts to the addresses specified in a task's email parameter,
the sla_miss_callback specifies an additional Callable object to be invoked when the SLA is not met.