# Airflow Use Case for PDILL

Airflow is an orchestration tool that can communicate with multiple services. With multithreaded deployments of the Airflow Environment you can horizontally scale your application dynamically.

In PDILL, airflow can run workflows on a schedule or on demand with using the endpoints. Alternatively, Airflow can also SSH into the machine and trigger scripts as per our requirements eliminating the need for us to manage API end points.

PDILL Architecture with Airflow-Celery

* User Interacts with DAG through the Airflow Webserver
* Airflow metadata database is hosted on a separate server
* Scheduler connects with the metadata database through a SQL client
* Airflow uses Celery to distribute data across the various nodes
* Each node has a Celery worker installed on the node
* Celery depends on the message broker to distribute tasks to the nodes
* The Celery worker then executes the tasks
* The edge nodes communicate with API layer to trigger processes and orchestrate
* The edge nodes also perform audit tasks and send out notifications

A picture containing graphical user interface

Description automatically generated

Code Management

The code for creating the jobs will be written in Python and managed on a repository either on github or Azure Dev Ops

The Airflow webserver will be hosted on Sandbox, Dev, Test, Stage and Prod server, the code for deployment on each branch will be deployed on the Master and Main branch.

For making enhancements to the code or adding new jobs the latest code deployed on each environment will be pulled from master/main branch and committed to a side branch(Developer Branch) for development purposes and then undergo a PR process involving code review and code approvals after which the new code will be deployed on the servers.

Diagram

Description automatically generated

Code Promotion

The code for Airflow will be initially hosted on sandbox server where all the POC and enhancements are worked on.

After which the code will be pushed to dev server after going a PR process involving code review and code approval.

Once the code is ready to be pushed to Test it will again go through the same PR process and then to stage and prod.

If there are any issues observed in the code on test the code will be debugged and changed on dev and then again promoted to test, stage and prod.

\*\* Note: The Dev, Test, Stage and Prod servers share the same configuration with respect to access and network settings.

Graphical user interface

Description automatically generated

# Airflow Code Demo

Basic Idea : The dag is a sequence of tasks that have dependencies between on another. In the pdill dag we have http operators that trigger the jobs and then we have sensors to track the progress of the job. These sensors halt the execution of the dag until the underlying job has completed and only proceeds if the job results in a success

from datetime import datetime

from airflow import DAG

import json

from airflow.operators.http\_operator import SimpleHttpOperator

from airflow.operators.python\_operator import PythonOperator

from airflow.sensors.http\_sensor import HttpSensor

Firstly, We import the required libraries such as the DAG class and other operators which are required for the development.

We can also import custom libraries from custom packages here.

Summary - Importing libraries and operators and custom functions

default\_args = {

'description':'TDMF data masking',

'start\_date': datetime(2020,9,25),

'email': ['prashanth.desani@wellsfargo.com'],

'retries': 3,

'email\_on\_failure': True,

'email\_on\_retry': False,

}

Summary: Default arguments are required and are sent to all tasks. We can set options that are redundant in all tasks here

The default arguments are the arguments or options that will be passed to all the tasks in the Dag. This will help us avoid redundantly providing these options to each and every task separately. It is also useful if we need to make small changes in these options from time to time. We will have to make it in just one place and not having to make changes in all the tasks.

Description: This parameter holds a one-line description that will show up on the UI beside your dag name (which is defined later in the code).

Start\_date: start date is defined for the scheduler to understand from which day it is supposed to start executing this particular dag. It is helpful when you want to start the dag on a certain day post-deployment or otherwise you can set any past date. However, this is an important option. There is another option catchup (defined later in the DAG object) which goes in conjunction with this start date. The catchup option tells the scheduler if it is supposed to ‘catch-up’ on executing the historical dag runs or not. So if you set a historical date say 5 days ago from the date of deployment. The scheduler will create 5 dag runs and the next required run when the scheduled interval arrives

Email: The email option defines a list of emails that the task is supposed to send the emails to.

Retries: The number of times the task should retry before giving out a failure status

Email\_on\_failure: This option tells the scheduler if it is supposed to email on the failure of a task

Email\_on\_retry: each task also needs retry options. The task then retries for the specified number of retries and this option just tells the scheduler if it is supposed to send an email to the email list or not.

configuration\_id = ""

pdill\_datamasking\_bkp='pdill\_datamasking\_bkp'

These are initialization variables just created for passing them to the tasks.

def resolve\_config\_id(\*\*kwargs):

task\_instance = kwargs['task\_instance']

configuration\_dict = json.loads(task\_instance.xcom\_pull(task\_ids='data\_extraction', key='return\_value'))

configuration\_id = configuration\_dict['configurationId']

return configuration\_id

Summary - This is a Custom Function that helps us resolve the configuration ID. You can define any number of functions and call them through the Python Operator

The resolve\_config\_id function resolves the configuration\_id for the consequent steps in the dag. It pulls the return value from the starting task and then pushes it as a singleton value in the Xcom to make it easily accessible to the consequent tasks.

Xcom is a common Dictionary available through all the tasks in the dag. This helps the tasks to talk to each other and pass variables and make decisions on the outcome of the previous

Note: the tasks of a Dag are rendered every time a dag run is created. Hence some parameters which are dynamic need to be passed in the ‘Jinja Templating’ syntax. Jinja Template Engine is a web page rendering engine that resolves native python variables to our tasks. You can find them enclosed in ‘{{ … }}’ these braces. It is also important when you want to use variables that are passed on runtime. These runtime variables are attached to the dag\_run object which is only available during runtime.

def monitor\_job(response):

print('###################################monitor process response ##################')

print(response.text)

responseJson=json.loads(response.text)

jobStatus=responseJson["status"]

if jobStatus == "FAILED":

raise AirflowException("Job FAILED.")

else:

return jobStatus=="SUCCESS"

Summary : This is also a custom function that is invoked by the HttpSensor. It processes the response received and gives either True or False based on condition.

This function is defined to be called in the response check part of the HttpSensors (defined later). It takes in a response dictionary and then prints out the entire message in the log giving us a better idea of all the things the response contained.

dag = DAG('pdill\_datamasking', schedule\_interval=None, catchup=False, default\_args=default\_args)

Summary : The dag object is created here with the required parameters.

This line creates the dag object to which all the tasks will be attached and execute.

The first argument is the name of the dag (which will show up on the UI),

next is the scheduled interval which defines the schedule of the dag in the ‘cron’ format

Next is the catchup option to tell the scheduler if historical dag runs are to be created or not

Lastly, we have the default\_arguments dictionary wherein we pass the default arguments that are needed in all the tasks and we do not want to keep passing them in each one separately

post\_data = {

"app": "CNAPP",

"project": "SS",

"sor": "msp",

"datasetName": "CHARMING\_17",

"date": "2021-05-30"

}

This is a data dictionary sent to the POST endpoint at the start of the execution. It is parameterized.

datamasking\_pipeline\_execution=SimpleHttpOperator(

task\_id='datamasking\_pipeline\_execution',

http\_conn\_id=pdill\_datamasking\_bkp,

endpoint='/datamasking/execution',

headers={"Content-Type": "application/json"},

data=json.dumps(post\_data),

response\_check=lambda response: response.json()['configurationId'] != None,

xcom\_push=True,

log\_response=True,

dag=dag

)

Summary: This the the SimpleHttpOperator, that helps us send Post and get requests to the api

Now creating a task in Airlfow means just invoking the right operator object you want with the right options. Here we invoke the SimpleHttpOperator which we imported at the beginning and provide the right options to it.

The task\_id defines the name of the task. It should be unique throughout the DAG.

The next is the http\_conn\_id. This is the connection id that we define in the ‘Connections’ section of the DAG UI. This helps us decouple secret information from the DAG’s codebase.

The next is ‘endpoint’ which is the endpoint we’re trying to hit. Note that it does not have the hostname attached to it.

The headers are the headers to be passed with your request.

Data indicates any data you want to pass with the request. In our case, it is the post\_data dictionary that we have defined earlier.

Moving on we have the response\_check option it checks for the required success criteria and only gives a success output if it is met otherwise the task returns a failed state.

Xcom\_push value indicates if the return\_value is supposed to be pushed to the xcom or not.

Log\_response value indicates if the task is supposed to log the entire response or not.

Lastly, the dag option indicates which dag this task belongs to

get\_configurationId = PythonOperator(

task\_id='get\_configurationId',

dag=dag,

python\_callable=resolve\_config\_id,

provide\_context=True

)

Summary: this is a python operator that resolves the configuration ID and pushes then to Xcom.

This is a python operator that helps us execute custom python functions. Here we are using it to invoke resolve\_config\_id function that we have defined.

datamasking\_extraction = SimpleHttpOperator(

task\_id='datamasking\_extraction',

http\_conn\_id=pdill\_datamasking\_bkp,

endpoint="/datamasking/extraction/{{ task\_instance.xcom\_pull(task\_ids='get\_configurationId', key='return\_value') }}",

method='GET',

response\_check=lambda response: response.json()['status']=="SUCCESS",

log\_response=True,

dag=dag

)

Summary: this is again a Simple Http Operator that fires a get request to the datamasking/extraction endpoint.

Note : Response check in Simple Http Operator pertains to that request only. This means the operator only triggers the job and then exits with success when the response\_check condition is satisfied. This is why we use a sensor to track progress

This is similar to the previous SimpleHttpOperator but with a few differences.

The first one is that we attach a configuration\_id dynamically based on what we receive from the spark UI.

The next is the ‘method’, we specify it as ‘GET’. If we do not specify this value it defaults to ‘POST’.

datamasking\_extraction\_jobstatus = HttpSensor(

task\_id='datamasking\_extraction\_jobstatus',

http\_conn\_id=pdill\_datamasking\_bkp,

endpoint="/datamasking/jobstatus/{{ task\_instance.xcom\_pull(task\_ids='get\_configurationId', key='return\_value') }}",

method='GET',

request\_params={},

response\_check=lambda body:monitor\_job(body),

#response\_check=lambda response: response.json()['status']=="SUCCESS",

poke\_interval=5,

dag=dag

)

Summary : the http sensor pokes the tracking endpoint after the set interval of 5 seconds. This interval can be changed based on our requirements.

This is a HttpSensor. We have imported this at the start and it is used to keep poking any endpoint until we get the required response from the server. So here we have an endpoint that pokes with the interval of 5 seconds and it will keep poking it until we get the ‘status’ key in the response as ‘SUCCESS’

.

.

.

.

.

.

.

datamasking\_pipeline\_execution >> get\_configurationId >> datamasking\_extraction >> datamasking\_extraction\_jobstatus >> datamasking\_execution >> datamasking\_execution\_jobstatus >> datamasking\_validation >> datamasking\_validation\_jobstatus >> datamasking\_ndm >> datamasking\_ndm\_jobstatus >> datamasking\_ingestion >> datamasking\_ingestion\_jobstatus >> datamasking\_complete

Summary : The above is the hierarchy that we have defined. The hierarchy can be any way we want. It can have branches and also conjunctions. In our case, we did not need any of them so we have a pretty simple hierarchy. The ‘>>’ operator defines a dependency between the tasks.

# Email Code Explanation

The email code is encapsulated in a function called as a notify email. This function is called in the on\_failure\_callback and on\_success\_callback in the default arguments of the dag (This means that these options are going to be passed to all the tasks in the dag). Thus all the tasks will call the ‘notify\_email’ function when they succeed or when they fail.

def notify\_email(kwargs):  
 *"""Send custom email alerts."""*

ti = kwargs['ti']  
 dag\_run = kwargs['dag\_run']  
 var = kwargs['var']['json']  
 params = kwargs['params']

The above variables are available in the kwargs which is like a global dictionary available for us to get metadata about the dag and the tasks.

Here in the above snippet we have used the following.

“ti” : ti stands for task\_instance that is the instance of the task from where the function is called.

“dag\_run” : dag\_run is a global object that holds information about the execution\_date the run\_id and other dag related information.

“var” : This variable is the configuration that we pass to the dag. Like the SOR name and everything. Basically everything that we give while triggering the dag. Or the parameters of the

dag basically.

recipient\_emails = ['prashanth.desani@wellsfargo.com']

The above is the variable where we store the recipient emails. The people airflow needs to send emails to whether on success or failure.  
  
 logs\_link = '{}/log?task\_id={}&dag\_id={}&execution\_date={}'.format(dag\_run.conf.get("webserver", "base\_url"),  
 ti.task\_id, ti.dag\_id, ti.execution\_date)

The above creates a log link. When anyone receives this link and clicks on it. They will be redirected to the logs page for that task on the UI directly from the email.

title = ''  
 body = ''  
  
 # email title.  
 if dag\_run.\_state == "success":  
 title = f"[ Airflow Success Alert ]: {dag\_run.dag\_id} Execution Completed"  
 body = """  
 Hi Everyone, <br>  
 <br>  
 Job {dag\_id} has executed successfully.<br>  
 <br>  
 Thank you,<br>  
 Airflow Bot <br>  
 """.format(dag\_id=dag\_run.dag\_id)  
 else:  
 if ti.state == "failed":  
 title = f"[ Airflow Failure Alert ]: {ti.task\_id} Failed for {dag\_run.dag\_id} failed"  
 body = """  
 Hi Everyone, <br>  
 <br>  
 Task {task\_id} failed.<br>  
 Please check the log at : {log\_link}<br>  
 <br>  
 Thank you,<br>  
 Airflow bot <br>  
 """.format(task\_id=ti.task\_id, log\_link=ti.log\_url)  
 elif ti.state == "success":  
 title = f"[ Airflow Success Alert ]: {ti.task\_id} Succeeded for {dag\_run.dag\_id}"  
 body = """  
 Hi Everyone, <br>  
 <br>  
 Task {task\_id} has completed successfully<br>  
 <br>  
 Thank you,<br>  
 Airflow bot <br>  
 """.format(task\_id=ti.task\_id, log\_link=ti.log\_url)  
 else:  
 raise AirflowException('{} task state is not supported in email notifications'.format(ti.state))

The above is a simple snippet that handles the the creation of the email template using multiple variables. Each object has a different metadata about the dag and the tasks.

send\_email(recipient\_emails, title, body)

Once the above template is created the email is simply sent using the send\_email function.