ML

Airflow XCOM : The Ultimate Guide

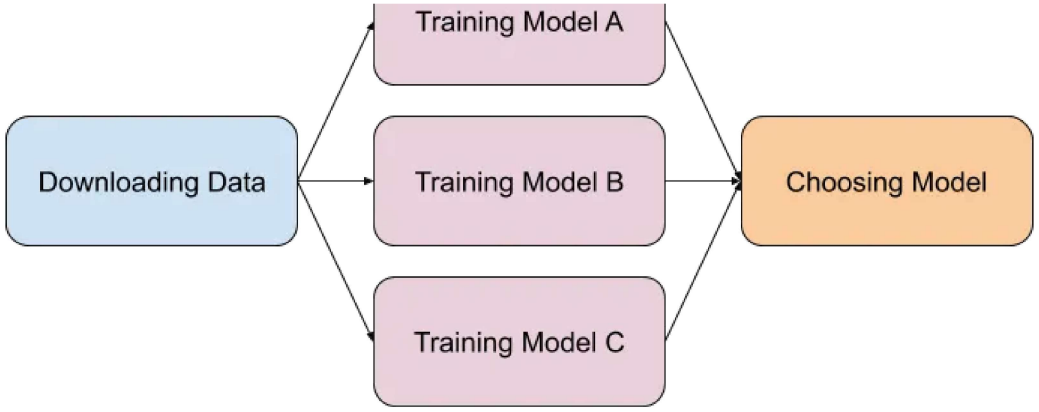
6 Comments / Apache Airflow, xcoms / By marclamberti

Wondering how to share data between tasks? What are XCOMs in Apache Airflow? Well you are at the right place. In this tutorial, you are going to learn everything you need about XComs in Airflow. What are they, how they work, how can you define them, how to get them and more. If you followed my course "Apache Airflow: The Hands-On Guide", Aiflow XCom should not sound unfamiliar to you. At the end of this tutorial, you will have a solid knowledge of XComs and you will be ready to use them in your DAGs. Let's get started!

Table of Contents

Use case

As usual, to better explain why you need a functionality, it's always good to start with a use case. The Airflow XCom is not an easy concept, so let me illustrate why it might be useful for you. Let's imagine you have the following data pipeline:



In a nutshell, this data pipeline trains different machine learning models based on a dataset and the last task selects the model having the highest accuracy. The question is,

How can we get the accuracy of each model in the task *Choosing Model* to choose the best one?

One solution could be to store the accuracies in a database and fetch them back in the task *Choosing Model* with a SQL request. That's perfectly viable. But, it's there any native easier mechanism in Airflow allowing you to do that?

Yes! XComs!

What is an Airflow XCom ?

XCom stands for "cross-communication" and allows to exchange messages or **small** amount of data between tasks. You can think of an XCom as a little object with the following fields:

- The **value** is … the value of your XCom. What you want to share. Keep in mind that **your value must be serializable in JSON or pickable**. Notice that serializing with pickle is disabled by default to avoid RCE exploits/security issues. If you want to learn more about the differences between JSON/Pickle click here.
- The **timestamp** is the data at which the XCom was created.
- The **execution date**! This is important! That execution date corresponds to the execution date of the DagRun having generated the XCom. That's how Airflow avoid fetching an XCom coming from another DAGRun. You don't know what I'm talking about? Check my video about how scheduling works in Airflow.
- The **task id** of the task where the XCom was created.
- The **dag id** of the dag where the XCom was created.

To access your XComs in Airflow, go to Admin -> XComs.

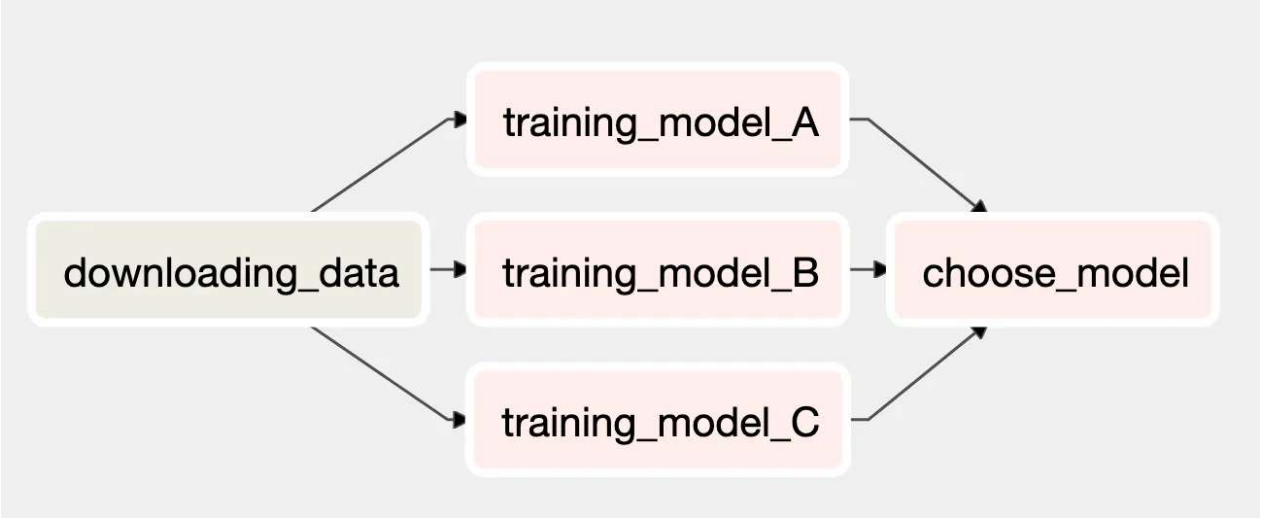
Great! Now you know what a XCom is, let's create your first Airflow XCom

How to use XCom in Airflow

Time to practice! To let you follow the tutorial, here is the data pipeline we use:

```
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
from random import uniform
from datetime import datetime
default_args = {
'start_date': datetime(2020, 1, 1)
}
def _training_model():
accuracy = uniform(0.1, 10.0)
print(f'model\'s accuracy: {accuracy}')
def _choose_best_model():
print('choose best model')
with DAG('xcom_dag', schedule_interval='@daily', default_args=default_args, catchup=False) as
downloading_data = BashOperator(
task_id='downloading_data',
bash_command='sleep 3'
)
training_model_task = [
PythonOperator(
task_id=f'training_model_{task}',
python_callable=_training_model
) for task in ['A', 'B', 'C']]
```

Add this code into a file xcom_dag.py in dags/ and you should obtain the following DAG:



The data pipeline is pretty simple. We have 5 tasks. downloading_data is a BashOperator executing the bash command which waits for 3 seconds. Then, we have 3 tasks, training_model_[A,B,C] dynamically generated in a list comprehension. Each task implements the PythonOperator to execute the function _training_model. That functions generates randomly an accuracy for each models A, B, C. Finally, we want to choose the best model based on the generated accuracies in the task choose_model.

Our goal is to create one XCom for each model and fetch back the XComs from the task choose_model to choose the best.

How? 2 steps:

1. Create an XCom for each training_model task
2. Pull the XComs from choose_model

Let's do it!

How to push an Airflow XCom

In this Airflow XCom example, we are going to discover how to push an XCom containing the accuracy of each model A, B and C.

There are multiple ways of creating a XCom but let's begin the most basic one. Whenever you want to create a XCom from a task, the easiest way to do it is by **returning** a value. In the case of the PythonOperator, use the **return keyword along with the value** in the python callable function in order to create automatically a XCom.

```
def _training_model(ti):
accuracy = uniform(0.1, 10.0)
print(f'model\'s accuracy: {accuracy}')
return accuracy
```

By adding **return accuracy**, if you execute the DAG, you will obtain the following XComs:

Key	Value	Timestamp	Execution Date	Task Id	Dag Id
return_value		2021-01-04, 12:52:01	2021-01-03, 00:00:00	downloading_data	xcom_dag
return_value	7.320465876852752	2021-01-04, 12:52:02	2021-01-03, 00:00:00	training_model_C	xcom_dag
return_value	8.823582660326675	2021-01-04, 12:52:02	2021-01-03, 00:00:00	training_model_A	xcom_dag
return_value	5.600056149882824	2021-01-04, 12:52:02	2021-01-03, 00:00:00	training_model_B	xcom_dag

Well done! With just one line of code, you've already pushed your first XCom!

What's important here is the key, **return_value**. By default, when a XCom is automatically created by returning a value, Airflow assigns the key **return_value**. In addition, you can see that each XCom was well created from different tasks (based on the task ids) but got something weird here. We don't return any value from the task downloading_data but we an associated XCom.

Where does it come from?

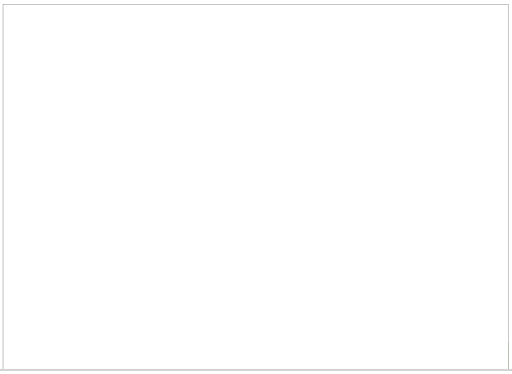
The do_xcom_push argument

By default, **all operators returning a value, create a XCom**. **There is one argument that ALL OPERATORS SHARE** (BashOperator, PythonOperator etc.) **which is do_xcom_push** set to True. Let's change that argument for the BashOperator to False.

```
downloading_data = BashOperator(
task_id='downloading_data',
bash_command='sleep 3',
do_xcom_push=False
)
```

Turn off the toggle of the DAG. Clear the task instances (In Browse -> Task Instances). Delete all DAGRuns (Browse -> DagRuns) as well as the XComs (Browse -> XComs). Now, if you turn on the toggle of your data pipeline again, you obtain the following XComs:

As you can see, this time, we don't get the extra XCom that was generated by downloading_data. As an exercise, try to avoid generating XComs from the PythonOperator with the same argument. At the end, you should have no XComs at all.



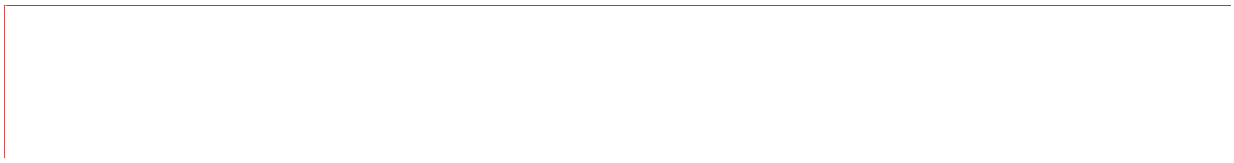
The simplest way to create a XCom is by returning a value from an operator. We know that, and we know that we can change that behaviour with do_xcom_push. By the way, keep in mind that all operators do not return XComs. Therefore. it depends of the implementation of the operator you use.

Ok, is there another way to create a XCom? A way that allows more flexibility? Yes there is! With the method xcom_push. Let's use it!

First thing first, the method **xcom_push is only accessible from a task instance object**. With the PythonOperator we can access it by passing the parameter **ti** to the python callable function. In Airflow 1.10.x, we had to set the argument provide_context but in Airflow 2.0, that's not the case anymore. Now, you just have to specify the keyword argument as a parameter for the python callable function.

```
def _training_model(ti):
accuracy = uniform(0.1, 10.0)
print(f'model\'s accuracy: {accuracy}')
eturn accuracy
```

Notice the argument ti. Once we can access the task instance object, we can call xcom_push.



xcom_push expects two parameters:

1. A key to identify the XCom
2. A value to the XCom that is serializable in JSON or picklable, stored in the metadata database of Airflow.

At the end, to push the accuracy with xcom_push you do,

```
def _training_model(ti):
accuracy = uniform(0.1, 10.0)
print(f'model\'s accuracy: {accuracy}')
ti.xcom_push(key='model_accuracy', value=accuracy)
```

If you trigger the DAG again, you obtain 3 XComs. However, they all have the same key, *model_accuracy* as specified in xcom_push and not return_value as before. By the way, when you execute twice your DAG on the **same execution date,** the XComs created during the first DAGRun are overwritten by the ones created in the second DAGRun.

That's it! That's all you need to know about xcom_push.

The useless argument

Actually, there is one additional parameter I didn't talk about which is *execution_date*. By specifying a date in the future, that XCom won't be visible until the corresponding DAGRun is triggered. To be honnest, I never found any solid use case for this.

Pulling a XCom with xcom_pull

Alright, now we know how to push an XCom from a task, what about pulling it from another task? We are trying to exchange data between tasks, are we? Let's go!

In order to pull a XCom from a task, you have to use the xcom_pull method. Like xcom_push, this method is available through a task instance object. xcom_pull expects 2 arguments:

1. **task_ids**, only XComs from tasks matching ids will be pulled
2. **key**, only XComs with matching key will be returned

Two things to keep in mind here. First, it looks like we can specify multiple task ids, therefore we can pull XComs from

```
3 | print(f'choose best model: {fetched_accuracy}')
```

In the code above, we pull the XCom with the key *model_accuracy* that was created from the task *training_model_A*. Trigger your DAG, click on the task *choose_model* and *log*. You obtain the output:

We have successfully pulled the accuracy stored in a XCom that was created by the task *training_model_A* from the task *choosing_model*! (Notice that the value will be different for you).

We know how to push and pull a XCom between two tasks. At this point, we are able to share data between tasks in Airflow! Great! But that's not all. Indeed, we are able to pull only one XCom from choose_model, whereas we want to pull all XComs from training_model_A, B and C to choose which one is the best.

How can we do this?

Simple! You just need to specify the task ids in xcom_pull.

```
def _choose_best_model(ti):
fetched_accuracies = ti.xcom_pull(key='model_accuracy', task_ids=['training_model_A', 'traini
print(f'choose best model: {fetched_accuracies}')
```

If you trigger you DAG, you obtain the 3 different accuracies and now you are able to choose which model is performing the best.

Congratulations! Now you are able to exchange data between tasks in your data pipelines!

Wait...

You want to learn more? 😱

Ok then!

The bashoperator with XComs

I know, I know. So far, in the Airflow XCom example, we've seen how to share data between tasks using the PythonOperator, which is the most popular operator in Airflow. Great, but. There is another very popular operator which is, the BashOperator.

The BashOperator xcom_push

You already know that by default, an XCom is pushed when you use the BashOperator. We've seen that with the task downloading_data. This controlled by the parameter do_xcom_push which is common to all operators. Nonetheless, there was one issue. The XCom was empty. So, how can we create an XCom having a value with the BashOperator?

By using templating! Wait, what? You don't know what templating is? Well, check my other tutorial right there before moving on. THIS IS SUPER IMPORTANT!

Now you know, what templating is, let's move on! Here is what you should do to push a XCom from the BashOperator:

```
downloading_data = BashOperator(
task_id='downloading_data',
bash_command='echo "Hello, I am a value!"',
do_xcom_push=True
)
```

And you obtain

Keep in mind that, **only the last line written to stdout by your command, will be pushed as a XCom**. By the way, you don't have to specify do_xcom_push here, as it is set to True by default.

Pushing a XCom with the BashOperator done, what about pulling a XCOM?

The bashoperator xcom_pull

Pulling a XCom from the BashOperator is a little bit more complex. This time, as you can't execute a python function to access the task instance object, you are going to use the Jinja Template Engine. Indeed, since the argument bash_command is templated, you can render values at runtime in it. Let's leverage this to pull a XCom.

```
fetching_data = BashOperator(
task_id='fetching_data',
bash_command="echo 'XCom fetched: {{ ti.xcom_pull(task_ids=[\'downloading_data\']) }}'",
do_xcom_push=False
)
```

Here, the magic happens with the two pairs of curly brackets {{}}. That's how we indicate to the Jinja Template Engine that a value here should be evaluated at runtime and in that case, xcom_pull will be replaced by the XCom pushed by the task downloading_data. Notice that I didn't specify a key here. Why? Because the key of the XCom retuned by downloading_data is *return_value*. This is the default behaviour. Same for xcom_pull. By default, the key of the XCom pulled is return_value. That's why, I didn't specify it here.

Add this task just after downloading_data and set the dependency accordingly (downloading_data >> fetching_data) and you should obtain:

Keep in mind that you might not be able to do that with all operators. At the end, you have to understand how your operator works, to know if you can use XComs with it and if so, how. For that, the code/documentation is your friend 😍

XCom limitations

DO NOT SHARE PANDA DATAFRAMES THROUGH XCOMS OR ANY DATA THAT CAN BE BIG!

I insist, do NOT do that! Why?

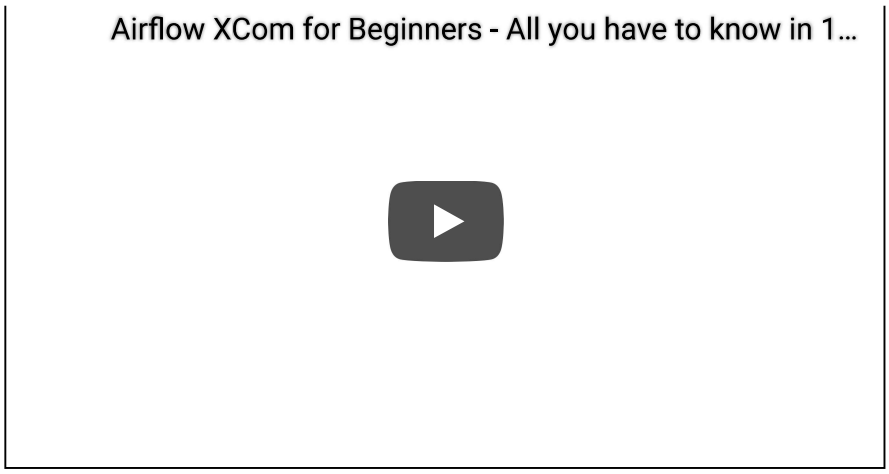
Airflow is NOT a processing framework. It is not Spark, neither Flink. Airflow is an orchestrator, and it the best orchestrator. There is no optimisations to process big data in Airflow neither a way to distribute it (maybe with one executor, but this is another topic). If you try to exchange big data between your tasks, you will end up with a memory overflow error! Oh, and do you know the xcom limit size in Airflow?

Guess what, it depends on the database you use!

- SQLite: 2 Go
- Postgres: 1 Go
- MySQL: 64 KB

Yes, 64 Kilobytes for MySQL! Again, use XComs only for sharing small amount of data.

One last point, don't forget that XComs create implicit dependencies between your tasks that are not visible from the UI.



Conclusion

That's it about Airflow XCom. I hope you really enjoyed what you've learned. There are other topics about XComs that are coming soon (I know, I didn't talk about XCom backends and XComArgs 😉) . If you want to learn more about Airflow, go check my course The Complete Hands-On Introduction to Apache Airflow right here. Or if you already know Airflow and want to go way much further, enrol in my 12 hours course here.

Have a great day! 🙂

6 thoughts on “Airflow XCOM : The Ultimate Guide”

SHUBROOKS8847
2021-01-10 AT 15:31

Thank you!!1

Reply

Reply

MARCLAMBERTI
2021-03-25 AT 08:55

Gigabytes 🙂

Reply

SMITA
2021-03-26 AT 18:03

You are brilliant Marc! can't stop myself from appreciating your great efforts in explaining the concept so well. It's so easy to understand. Keep up the good work!

I am not sure if you would have already made videos or would have written blogs too on airflow variables.It would be great if you can record/write one if that's not already available from you

Thanks a lot.

Reply

SHIVARP@GMAIL.COM
2021-04-06 AT 22:07

Hi Marc,

Did you get a chance to try out the XCOM with KubernetesPodOperator in Airflow 2.0?
I guess the addition of side-car for XCOM adds more complexity there

Thanks

Reply

THIB
2021-06-12 AT 20:41

Great tuto !

Reply

Leave a Comment

Your email address will not be published. Required fields are marked *

Name*

Email*

Website

Post Comment »

Search ...

Recent Posts

api

dag

kubernetes

Non classé

sensors

xcoms