



July 10, 2020

AIP-31: Airflow functional DAG

Airflow Summit 2020



- 1 Introduction**
- 2 Why functional DAG?**
- 3 Explicit XCom: XComArg**
- 4 @task decorator**
- 5 Future work**



Intro





Gerard Casas Saez

Software Engineer

ML Platform - Cortex @ Twitter

Follow me @casassaez



Why functional DAG?



Example ETL pipeline

Extract

GET request to HttpBin
/get endpoint

Data out: HttpBin JSON
string

Transform

Parse JSON
Extract origin parameter
Format email subject and
content

Data out: Email subject +
content strings

Load

Send email to myself to get
current IP



Passing data between operators

- XCom value vs Execution date based file paths
- **Preferred: XCom. Why?**
 - Sometimes **data fits in DB!** Ex: model training metrics.
 - **More flexible paths**, not only date needed, custom config (HDFS cluster, GCS vs HDFS...)
 - XCom are **visible from Web UI**, easier to debug
 - Better **reusability** of operators
 - Already used by a lot of **OSS Airflow operators!**



Example DAG

```
with DAG(...) as dag:
    extract = SimpleHttpOperator(
        task_id='extract', endpoint='get', method='GET', xcom_push=True
    )

    def transform(ti):
        content = ti.xcom_pull('extract', key='return_value')
        external_ip = json.loads(content)['origin']
        ti.xcom_push('subject', f'Server connected from {external_ip}')
        ti.xcom_push('body', f'Seems like today your Airflow is connected from {external_ip}')

    transform = PythonOperator('transform', python_callable=transform)

    load = EmailOperator(
        task_id='load', to='test@example.com',
        subject='{{context["ti"].xcom_pull("transform", "subject")}}',
        html_content='{{context["ti"].xcom_pull("transform", "body")}}'
    )
```




Example DAG

```
with DAG(...) as dag:
    extract = SimpleHttpOperator(
        task_id='extract', endpoint='get', method='GET', xcom_push=True
    )

    def transform(ti):
        content = ti.xcom_pull('extract', key='return_value')
        external_ip = json.loads(content)['origin']
        ti.xcom_push('subject', f'Server connected from {external_ip}')
        ti.xcom_push('body', f'Seems like today your Airflow is connected from {external_ip}')

    transform = PythonOperator('transform', python_callable=transform)

    load = EmailOperator(
        task_id='load', to='test@example.com',
        subject='{{context["ti"].xcom_pull("transform", "subject")}}',
        html_content='{{context["ti"].xcom_pull("transform", "body")}}'
    )

    extract >> transform >> load
```



AIP-31: Motivation

- ETL workflow resemble functions: [Functional Data Engineering](#)
 - Variable == data artifact \approx xcom metadata
 - Function == operator
- Data artifacts are **implicit** in Airflow (XCom table for metadata)
- Needs **explicit task dependency declaration**
- Custom **function to operator** is hard-ish (PythonOperator)



Prior art/Inspiration

- [Streamlined \(Functional\) Airflow roadmap](#)
- [TypedXComArg in ML Workflows](#) (internal Twitter Airflow fork)
- ML pipelines investigation
 - Prefect Functional DAG
 - Dagster pipelines and solids
 - Tensorflow Extended pipelines
 - Square's Bionic pipelines
 - Netflix Metaflow pipelines



Explicit XCom: XComArg class



XComArg: Reference to future XCom value

- Resolved on operator execution for templated fields
 - `XComArg(op, 'subject') == “{{context['ti'].xcom_pull('op_id', 'subject')}}”`
 - `XComArg(op, 'subject').resolve() == ti.xcom_pull(op, 'subject')`
- Used in **DAG definition**
- Change XComArg key using `__getitem__: val['body']`
- BaseOperator property to generate default XComArg: `.output`
- **Implicit task dependency** based on XComArg dependency



Example DAG

```
with DAG(...) as dag:
    extract = SimpleHttpOperator(
        task_id='extract', endpoint='get', method='GET', xcom_push=True
    )

    def transform(ti):
        content = ti.xcom_pull('extract', key='return_value')
        external_ip = json.loads(content)['origin']
        ti.xcom_push('subject', f'Server connected from {external_ip}')
        ti.xcom_push('body', f'Seems like today your Airflow is connected from {external_ip}')

    transform = PythonOperator('transform', python_callable=transform)

    load = EmailOperator(
        task_id='load', to='test@example.com',
        subject='{{context["ti"].xcom_pull("transform", "subject")}}',
        html_content='{{context["ti"].xcom_pull("transform", "body")}}'
    )

    extract >> transform >> load
```




Example DAG

```
with DAG(...) as dag:
    extract = SimpleHttpOperator(
        task_id='extract', endpoint='get', method='GET', xcom_push=True
    )

    def transform(content, ti):
        external_ip = json.loads(content)['origin']
        ti.xcom_push('subject', f'Server connected from {external_ip}')
        ti.xcom_push('body', f'Seems like today your Airflow is connected from {external_ip}')

    transform = PythonOperator('transform', python_callable=transform, op_args=[extract.output])

    load = EmailOperator(
        task_id='load', to='test@example.com', subject=transform.output['subject'],
        html_content=transform.output['body']
    )
```



@task decorator



Python function to Airflow operator

```
def transform(content, ti):  
    external_ip = json.loads(content)['origin']  
    ti.xcom_push('subject', f'Server connected from {external_ip}')  
    ti.xcom_push('body', f'Seems like today your Airflow is connected from {external_ip}')  
  
transform = PythonOperator('transform', python_callable=transform, op_args=[extract.output])
```




@task decorator

- Usage:
 - `@airflow.decorators.task`
 - `@dag.task`
- Calling decorated function **generates PythonOperator**
- **Set op_args and op_kwargs**
- **Multiple outputs support**, return dictionary with string keys.
- **Generate Task ids** automatically
- **Return default XComArg** when called
- [UPCOMING] No context kwarg support, instead `get_current_context()`



Example DAG

```
with DAG(...) as dag:
    extract = SimpleHttpOperator(
        task_id='extract', endpoint='get', method='GET', xcom_push=True
    )

    def transform(content, ti):
        external_ip = json.loads(content)['origin']
        ti.xcom_push('subject', f'Server connected from {external_ip}')
        ti.xcom_push('body', f'Seems like today your Airflow is connected from {external_ip}')

    transform = PythonOperator('transform', python_callable=transform, op_args=[extract.output])

    load = EmailOperator(
        task_id='load', to='test@example.com', subject=transform.output['subject'],
        html_content=transform.output['body']
    )
```




Example DAG

```
with DAG(...) as dag:
    extract = SimpleHttpOperator(
        task_id='extract', endpoint='get', method='GET', xcom_push=True
    )

    @dag.task(multiple_outputs=True)
    def transform(content):
        external_ip = json.loads(content)['origin']
        return {
            'subject': f'Server connected from {external_ip}',
            'body': f'Seems like today your Airflow is connected from {external_ip}'
        }
    transformed = transform(extract.output)

    load = EmailOperator(
        task_id='load', to='test@example.com', subject=transformed['subject'],
        html_content=transformed['body']
    )
```




Future work!





Future work + Contributions

- **@dag decorator:** Same concept as @task but to create DAG
 - Function kwargs == DAG parameters
- **Type hints** support for multiple outputs
 - Automatically detect if output must be splitted into different XCom values.
- Custom XCom backends
 - Handle **serialization for specific Python classes**
 - Handle I/O for different centralized local file systems: HDFS, GCS, S3...
 - **Ex:** Serialize/Deserialize pandas from/into CSV in HDFS when used for XCom values



Custom XCom backend

```

@dag.task
def transform(df: pd.DataFrame):
    df['new_col'] = df['x'] * df['y']
    return df

@dag.task
def other_transform(df: pd.DataFrame):
    df['other_col'] = df['new_col'] * df['y']
    return df

other_transform(transform(df))
```




@dag decorator

```
@dag(owner='airflow')
def my_pipeline(origin_data_path: str):
    @task
    def feature_eng(data_path):
        ...
    transformed = feature_eng(origin_data_path)

    @task(multiple_outputs=True)
    def train_model(data, hyper_params):
        ...

    train_model(transformed, {'lr': 0.1})

my_pipeline('/some/path/with/data')
```



Last but not least.
Not working alone:
Functional Ops SIG



Kudos to..

- Contributors for AIP-31
 - Tomek Urbaszek
 - Evgeny Shulman
 - Jonathan Shir
- + Airflow reviewers and committers (Kaxil, Ash, Jarek, Dan...)



Questions?





Thank you.

