Migrating Apache Oozie Workflows to Apache Airflow

Presenters: James Malone, Feng Lu
Overview
The Need for Workflow Solutions

Cron simple tasks

* * * * * command to run
CRON Job

Cost: low
Friction: medium

Cron + scripting

Cost: medium
Friction: high

Custom applications

Cost: high
Friction: high

Google Cloud
The Landscape

OSS
- Apache Oozie
- Luigi (Spotify)
- Apache Airflow
- Azkaban (Linkedin)
- Cadence (Uber)

Managed
- AWS Glue
- Google Cloud Scheduler
- Google Cloud Composer
- AWS DataPipeline
- Azure DataFactory

Google Cloud
Workflow Solution Lock-in

- Workflow structure mismatch (e.g., loop vs DAG)
- Workflow language spec (e.g., code vs config, XML vs YAML)
- No standard set of supported tasks
- Workflow expressiveness (e.g., dependency relationship)
- Coupling between workflow language and its underlying implementation

It's hard to migrate workflows from one system to another.
Oozie to Airflow Converter

- Understand the pain of workflow migration
- Figure out a viable migration path (hopefully it’s generic enough)
- Incorporate lessons learned towards future workflow spec design
- Why Apache Oozie and Apache Airflow?
  - Widely used
  - OSS
  - Sufficiently different (e.g., XML vs Python)
Apache Oozie

- Apache Oozie is a workflow management system to manage Hadoop jobs.
- It is deeply integrated with the rest of Hadoop stack supporting a number of Hadoop jobs out-of-the-box.
- Workflow is expressed as XML and consists of two types of nodes: control and action.
- Scalable, reliable and extensible system.
Hello world

Oozie workflow

This is a very simple Oozie workflow that performs a shell action. Pay attention to the following XML elements:

1. start
2. action
3. kill
4. end

```
<workflow-app xmlns="uri:oozie:workflow:1.0" name="shell-wf">
  <start to="shell-node"/>
  <action name="shell-node">
    <shell xmlns="uri:oozie:shell-action:1.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="uri:oozie:shell-action:1.0">
      <resource-manager>${resourceManager}\</resource-manager>
      <name-node>${nameNode}\</name-node>
      <prepare>
        <delete path="${nameNode}/user/${userName}/${examplesRoot}/apps/shell/test"/>
        <mkdir path="${nameNode}/user/${userName}/${examplesRoot}/apps/shell/test"/>
      </prepare>
      <configuration>
        <property>
          <name>mapred.job.queue.name</name>
          <value>${queueName}</value>
        </property>
      </configuration>
      <exec>java</exec>
      <argument>-version</argument>
      <capture-output/>
    </shell>
    <ok to="end"/>
    <error to="fail"/>
  </action>
  <kill name="fail">
    <message>Shell action failed, error message[${wf:errorMessage(wf:lastErrorNode())}]</message>
  </kill>
  <end name="end"/>
</workflow-app>
```
Apache Airflow

- Apache Airflow is a top-level project at the Apache Software Foundation (ASF).
- Airflow has become very popular within the open-source community.
- It’s designed to enable users to programatically author, schedule and monitor workflows.
- Workflows are authored as directed acyclic graphs (DAGs), and can be configured as code - using Python 2.x or 3.x.
Hello world DAG code

This is a very simple Airflow DAG that does three things in order:

1. Echo “hi”
2. Run the date command
3. Sleep for 5 seconds

As you can see, the workflow is written entirely in Python.

```python
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta

YESTERDAY = datetime.combine(
    datetime.today() - timedelta(days=1), datetime.min.time())

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': YESTERDAY,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

with DAG('hello_world', default_args=default_args) as dag:
    t0 = BashOperator(task_id='p_hi', bash_command='echo "hi"', dag=dag)
    t1 = BashOperator(task_id='p_date', bash_command='date', dag=dag)
    t2 = BashOperator(task_id='sleep', bash_command='sleep 5', dag=dag)
    t0 >> t1 >> t2
```
Design Goals

Correctness
1. Identical actions
2. Respect dependencies
3. Side effects are captured
4. Same workflow outcome

Flexibility
1. Support 1:M action mappings
2. Swappable mapper
3. Allow scheduling info overrides

We focus on workflow-app migration initially.
High-level Design Overview

workflow XML → [nodes] where node: name, attributes, child elements, etc.

[nodes] -> workflow object:
- dependencies
- relationship
- airflow-nodes

workflow object → dag.py
The Workflow Class

- Container object to hold metadata regarding an oozie workflow
- Intermediate representation of Oozie workflows
- Notable properties:
  - nodes: list of control/action nodes
  - relationships: task dependencies (e.g., “ok”, “error”)
  - dependencies: airflow “import” statements
The Mapper Class

- Two types of mapper classes
  - Control mapper: maps a control node in Oozie
  - Action mapper: maps an action node in Oozie
- Control mapper: update task relationship
- Action mapper: in addition to updating task relationship, also transform oozie action properties to Airflow operator arguments
- These arguments are then fed into per-operator Jinja templates
Operator Jinja Templates

Pig action template

```python
{{ task_id }}_prepare = bash_operator.BashOperator(
    task_id='{{ task_id }}_prepare',
    bash_command='{{ prepare_command }}'
)

{{ task_id }} = dataproc_operator.DataProcPigOperator(
    query_uri='{}/{}'.format(PARAMS['gcp_uri_prefix'], '{{ script_file_name }}'),
    task_id='{{ task_id }}',
    trigger_rule='{{ trigger_rule }}',
    variables={{ params_dict }},
    dataproc_pig_properties={{ properties }},
    cluster_name=PARAMS['dataproc_cluster'],
    gcp_conn_id=PARAMS['gcp_conn_id'],
    region=PARAMS['gcp_region'],
    dataproc_job_id='{{ task_id }}'
)

{%- with relation=relations %}
{%- include "relations.tpl" %}
{%- endwith %}
```

Shell action template

```python
{{ task_id }}_prepare = bash_operator.BashOperator(
    task_id='{{ task_id }}_prepare',
    bash_command='{{ prepare_command }}'
)

{{ task_id }} = bash_operator.BashOperator(
    task_id='{{ task_id }}',
    bash_command="gcloud dataproc jobs submit pig --cluster={0} --region={1}
        --execute 'sh {{ bash_command }}'".ormat(PARAMS['dataproc_cluster'], PARAMS['gcp_region'])
)

{{ task_id }}_prepare.set_downstream('{{ task_id }}')
```
## Oozie Control Node Mapping

<table>
<thead>
<tr>
<th>Oozie Node</th>
<th>Airflow Operator/Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>None (Airflow doesn't need an explicit start node)</td>
</tr>
<tr>
<td>DECISION</td>
<td>PythonBranchOperator</td>
</tr>
<tr>
<td>FORK</td>
<td>None (Airflow runs concurrent tasks whenever it can)</td>
</tr>
<tr>
<td>JOIN</td>
<td>None (TriggerRule.ALL)</td>
</tr>
<tr>
<td>END</td>
<td>DummyOperator if DECISION in upstream.nodes None otherwise</td>
</tr>
<tr>
<td>KILL</td>
<td>None (Task failure leads to DAG failure by default)</td>
</tr>
<tr>
<td>Oozie Node</td>
<td>Airflow Operator/Representation</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>PIG</td>
<td>DataProcPigOperator</td>
</tr>
<tr>
<td>MapReduce</td>
<td>DataprocHadoopOperator</td>
</tr>
<tr>
<td>Shell</td>
<td>BashOperator where a pig job is submitted to run a shell script</td>
</tr>
<tr>
<td>SubWorkflow</td>
<td>SubDagOperator</td>
</tr>
<tr>
<td>HDFS</td>
<td>BashOperator where a pig job is submitted to run a shell script</td>
</tr>
<tr>
<td>SPARK</td>
<td>DataprocSparkOperator</td>
</tr>
<tr>
<td>SSH</td>
<td>SSHOperator</td>
</tr>
</tbody>
</table>

Prepare statement in an Oozie action is mapped to a BashOperator.
Load and parse the workflow XML file with the Python XML ElementTree API → workflow

Output: workflow

Syntax clean-up and map Oozie node to Airflow node

Output: workflow + airflow nodes

Set up task relationship by following the “to” links and create trigger rules for each Airflow node

Output: workflow + airflow nodes + relationship + trigger rules

Convert transformed workflow to Airflow Dags with Jinja template rendering

Output: raw DAG file

Prettify the DAG to improve readability and facilitate future changes

Output: formatted DAG file
Demo Recap

- Oozie to Airflow converter repo: [https://github.com/GoogleCloudPlatform/cloud-composer](https://github.com/GoogleCloudPlatform/cloud-composer)
- Successfully converted a representative Oozie workflow to Airflow DAG
  - Includes all control nodes
  - Embeds a sub-workflow
    - Contains common actions such as MapReduce, Shell, Pig
- No post-conversion modification and runs well out of the box
Roadmap
Next Step

- Implement the rest of Oozie actions
- Support coordinator app for scheduling and pre-condition check
- Complete the development of EL functions
- Improve user experience of the converter tool (e.g., better error messaging, debugging support, etc)
- How to solve the general workflow migration problem?
  - A config-based workflow language spec (e.g., YAML spec)
  - Opinionated on control and data flow
  - Open to any task
Call for Contribution

- We collaborated with Polidea to design and implement the conversation tool.
- To scale the development and make it useful for the community, we welcome all contributions:
  - Try out the tool
  - Share your Oozie workflow
  - Help with the tool development
  - Improve documentation, testing coverage, etc

[GitHub Link](https://github.com/GoogleCloudPlatform/cloud-composer/tree/master/oozie-to-airflow)
Thank you!