Flow is in the Air: Best Practices of Building Analytical Data Pipelines with Apache Airflow

Dr. Dominik Benz, inovex GmbH

PyConDe Karlsruhe, 27.10.2017
Diving deep in the analytical data lake?

- Dependencies between jobs?
- Overview of failed jobs?
- Clean metadata on job runs?
- Avoid boilerplate data transformation code?
- Move analytical pipelines to production?
- Easily test transformations?
The Flow in Airflow

- Airflow Basics
- Comparison with other workflow engines
- Wrap up & Summary
- Lessons Learned & best practices
A typical data lake

- Data of various types
- Dependencies among tables
- Scheduling, orchestration, metadata
- Stored in original format
- Export to external targets

- Ingress
- Raw
- Processed
- Datahub
- Analysis
- Egress

user access, system integration, development
Airflow: let’s DAG!

- Workflow is (python) code
- Specify tasks & dependencies programmatically
- Manages workflow metadata
- Nice GUI 😊
Brief History

› developed at AirBnB by Maxime Beauchemin (former Yahoo / Facebook)

› open-sourced June 2015

› ~4200 commits, 81 releases, 332 contributors

› in Apache Incubator starting 2016/05

› used in several projects since 09/2015 😊
Gimme some code ..

```python
from airflow import DAG

default_args = {'owner': 'airflow',
                 'retries': 2,
                 ...
}

dag = DAG('tutorial', default_args=default_args)

t1 = BashOperator(task_id='print_date',
                  bash_command='date',
                  dag=dag)

t2 = HiveOperator(task_id='make_query',
                  sql='select x from y where z group by k',
                  dag=dag)

t2.set_upstream(t1)
```
Airflow: Architectural components
Basic Concepts

› **DAG**: graph of operator usages (=tasks)

› **Operator**: "Transformation" step
  › **Sensor**: Operator which polls with frequency / timeout (e.g. LocalFileSensor)
  › **Executor**: Trigger operation (e.g. HiveOperator, BashOperator, PigOperator, ...)

› **Task**: Usage of Operator in DAG
  › **Task Instance**: run of a Task at a point in time

› **Hook**: Interface to external System (JDBCHook, HTTPHook, ...)
## Most popular airflow CLI commands

<table>
<thead>
<tr>
<th>command</th>
<th>does</th>
</tr>
</thead>
<tbody>
<tr>
<td>airflow initdb</td>
<td>initialize metadata DB schema</td>
</tr>
<tr>
<td>airflow test &lt;dag&gt; &lt;task&gt; &lt;date&gt;</td>
<td>test task of a dag (shows command only)</td>
</tr>
<tr>
<td>airflow run &lt;dag&gt; &lt;task&gt; &lt;date&gt;</td>
<td>run task of a dag</td>
</tr>
<tr>
<td>airflow backfill &lt;dag&gt; -s &lt;start_date&gt; -e &lt;end_date&gt;</td>
<td>reload / backfill dag</td>
</tr>
<tr>
<td>airflow clear &lt;dag&gt; -s &lt;start_date&gt; -e &lt;end_date&gt; -t &lt;task_regex&gt;</td>
<td>clear state of dag / tasks</td>
</tr>
<tr>
<td>airflow backfill &lt;dag&gt; -s &lt;start_date&gt; -e &lt;end_date&gt; -m true</td>
<td>mark dag runs as success without running</td>
</tr>
</tbody>
</table>
Advanced Concepts

› **XCom**: send „messages“ between tasks

› **Trigger Rules**: specify handling for multiple upstream dependencies (e.g. all_success, one_success, ..)

› **Variables**: define key/value mappings in airflow metadata DB (value can be nested JSON as well)

› **Branching**: Define python function to choose which downstream path to follow

› **SubDAGs**: encapsulate repeating functionality
The Flow in Airflow

**Airflow:**
- Workflow is code
- Dynamic DAGs
- Clean metadata DB

**Comparison with other workflow engines**

**Wrap up & Summary**

**Lessons Learned & best practices**
**What else is out there?**

<table>
<thead>
<tr>
<th></th>
<th>Oozie</th>
<th>Azkaban</th>
<th>Luigi</th>
<th>Airflow</th>
<th>Schedoscope</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
<td>Java</td>
<td>Java</td>
<td>Python</td>
<td>Python</td>
<td>Scala</td>
</tr>
<tr>
<td><strong>WF specification</strong></td>
<td>static (XML)</td>
<td>static (.job file)</td>
<td>static (task = extend class)</td>
<td>dynamic (task = instantiate operator)</td>
<td>dynamic</td>
</tr>
<tr>
<td><strong>Schema / Change Management</strong></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td><strong>Test Framework</strong></td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td><strong>WF trigger</strong></td>
<td>data, time</td>
<td>time</td>
<td>data, time</td>
<td>data (sensors), time</td>
<td>goal</td>
</tr>
</tbody>
</table>
Comparison (2016) of Airflow, Luigi, Pinball by Marton Trencseni (data science manager at Facebook) http://bytepawn.com/luigi-airflow-pinball.html:

„If I had to build a new ETL system today from scratch, I would use Airflow“

Databricks Airflow integration
https://databricks.com/blog/2017/07/19/integrating-apache-airflow-with-databricks.html:

„We implemented an Airflow operator called DatabricksSubmitRunOperator, enabling a smoother integration between Airflow and Databricks“
The Flow in Airflow

Airflow:
• Workflow is code
• Dynamic DAGs
• Clean metadata DB

Comparison
• Less static than most
• More active development
  • Nicest GUI

Wrap up & Summary

Lessons Learned & best practices
Structuring / Cutting DAGs

- one DAG per data source
- one DAG per "project"
- one DAG per data sink
Structuring DAG resources

- keep code in template files
- for hive templates: use `hiveconf_jinja_translate`
- use template searchpath (see next slide)
- keep template files „airflow agnostic“ (if possible)
from airflow import DAG

default_args = { 'owner': 'airflow',
                 'retries': 2,
                 ...
}

dag = DAG('src_pos', default_args=default_args,
          template_searchpath=(
            '/base/workflows/src_pos/hive',
            '/base/workflows/src_pos/sqoop'))

insert = HiveOperator( task_id='insert',
                      sql='insert.sql',
                      dag=dag)

...
Configuration Management

› Built-in: definition of
  › Connections (e.g. to Hive, DBs, ..)
  › Variables (key/value)

› Often other (non-python) tools present
  › e.g. Realtime tools, ..

› Goal: single source of configuration
  › inject e.g. via "user_defined_macros"
Configuration Management (ctd.)

**conf.py**
```
ENV_NAME="prod"
PRJ_NAME="myprj"
...
```

**insert.sql**
```
INSERT INTO TABLE ${ENV_NAME}_${PRJ_NAME}_target
SELECT .. FROM ..
```

**dag definition**
```
from airflow import DAG
import conf
...

dag = DAG('src_pos',
default_args=default_args,
user_defined_macros=conf.__dict__)

insert = HiveOperator(
task_id='insert',
sql='insert.sql',
dag=dag)
...
```
Develop & Deploy Workflows

**PERSONAL**
- data engineers or data scientists
- test via "airflow test"

**INTEGRATION**
- perfomance tests, integration tests
- fixes

**PRODUCTIVE**
- monitoring
- backfilling of failed jobs

Git / Jenkins

LocalExecutor / CeleryExecutor

Cylindrical RDBMS

LocalExecutor

LocalExecutor

LocalExecutor
 Integrating with the Enterprise

› Often existing workflow / scheduling tools present (e.g. control M, ...)

› Existing integration in e.g. ticketing systems

› Idea: “Hierarchy“ of engines:
  › Enterprise engine: scheduling, coarse-grained
  › Airflow: workflow, fine-grained
Writing Plugins & Extensions

› Extension points:
  › **operators**
  › **hooks**
  › executors
  › macros
  › UI adaption (views, links)

› Easy – but also needed 😊

› Start from existing classes, adapt

› Developed so far:
  › SSHFileSensor
  › HiveServer2Operator (you have to!)
  › SparkSQLOperator
  › SparkOperator
  › ...

› integrate via airflow_plugin_directory
### Configs, Gotchas, ..

<table>
<thead>
<tr>
<th>config, topic</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>airflow.cfg: <strong>parallelism</strong></td>
<td>max nr. of task instances to run in parallel (per metadata DB / installation)</td>
</tr>
<tr>
<td>airflow.cfg: <strong>dag_concurrency</strong></td>
<td>how many parallel tasks are allowed per dag (attention: further tasks will <strong>not be scheduled</strong>!)</td>
</tr>
<tr>
<td>LDAP integration</td>
<td>works, but problems with LDAPs who implement another „memberOf“ attribute (fixed in 1.9, see AIRFLOW-1095)</td>
</tr>
<tr>
<td>Kerberos</td>
<td>kerberos relogin process („airflow kerberos“) broken; workaround = own BashOperator who does kinit with a keytab</td>
</tr>
<tr>
<td>Hive integration via impyla</td>
<td>Problems with 1.8.2 (thrift-sasl version mismatch); solution = downgrade thrift_sasl to 0.2.1 (instead of 0.3.0)</td>
</tr>
<tr>
<td>...</td>
<td>fore sure more to come 😊</td>
</tr>
</tbody>
</table>
The Flow in Airflow

Airflow:
- Workflow is code
- dynamic DAGs
- Clean metadata DB

Comparison
- Less static than most
- More active development
- Nicest GUI

Wrap up & Summary

Best practices
- structure workflows + resources
- manage config
- integrate...
Summary

What’s the flow ..

› **Airflow = workflow is code**
› Programmatically define DAGs of Operators
› Integrates seamlessly into “pythonic“ data science stack
› Easily extensible (which is needed)
› Lowers the gap between data science + engineering
› Clean management of workflow metadata (state, runs, ...)
› Under active development
› Fun to use, & real-world project proven 😊
Vielen Dank

Dr. Dominik Benz
dominik.benz@inovex.de

inovex GmbH
Park Plaza
Ludwig-Erhard-Allee 6
76131 Karlsruhe