Managing DAGs at scale
Deployment, versioning & package management
Agenda

• Keep track of airflow state
• keep track of DAGs states
• Revert to previous state
• Managing DAGs code base
• Releasing next airflow version
• Automate airflow deployment process
• How to manage DAGs package dependencies
Airflow/Dags States

Airflow image
v6

Dag1 v1
Dag2 v4
Dag3 v24
Dag4 v2
Airflow Repository

Contains

- Dags code
- Dockerfile
  - Inherit from airflow base image
  - install custom_dags/ into airflow dags/ directory
  - Install additional py packages
  - generate new custom image version x

Problems:

- All dags code in single repository
- hard to manage
Split Dags repositories

Dag 1

Dag 2

... 

Dag n

Airflow Repository
Split Dags repositories

Airflow Repository
- Dag_1 v?
- Dag_2 v?
- Dag_3 v?

Dag 1

Dag 2

\ldots

Dag n
Dags | Git submodules

Airflow Repository

- Git submodule add https://<dag_1_repo>
- Git submodule add https://<dag_2_repo>
- Git submodule add https://<dag_n_repo>

Diagram:

- Airflow Repository
  - Dag 1
  - Dag 2
  - Dag n

Directory Structure:

```
custom_dags/
|--- dag_1/
|   |--- ...
|--- dag_2/
|   |--- ...
```
Dags | Git submodules

Dag_1

Dag_2

Airflow repo

master v2

master v2

master v3

master v2

master v3

master v2

master v3

from custom_dags/
git submodule update --init

from dag_1/
git checkout v2

airflow v0.1.0
(d1-v3, d2-v2)

airflow v0.1.1
(d1-v2, d2-v2)

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2020, 1, 1),
    'retries': 0,
    'retry_delay': timedelta(minutes=2),
    'tags': [f"dag_version:{DAG_VERSION}"],
    f"operator_version:{OP_VERSION}""
}
CICD Pipeline Steps

Pipeline triggers on Airflow repository master branch

1. Initialize all submodules
2. Dags validation tests
3. Generate version
4. build airflow image
5. push to ECR
6. Update helm chart
7. ArgoCD sync

- Inherit from official airflow base image
- Install custom dags
- Update semantic version
- Identify whether next version is major, feature or patch

from custom_dags/
git submodule update --init
Airflow Deployment

- PROD
- DEV

- release version v0.1.1
- Kubernetes manifests prod-values.yaml
- push to master again
- ArgoCD CD sync v0.1.1
- push to ECR
- commit from cicd pipeline

- scheduler_parsing_processes: 4
- statsd:
  - enabled: true
  - image: prom/statsd-exporter
  - image_pull_policy: IfNotPresent
  - image_tag: v0.21.0
  - ingest_port: 9125
  - prefix: airflow-de
  - scrape_port: 9102
- dags_image:
  - pull_policy: IfNotPresent
  - repository: jagey-ads-airflow-prod-configuration-credentials
  - tag: 0.1.1
- secret: jagey-ads-airflow-prod-configuration-credentials
Package Dependency management

requirements.txt
> pandas 1.3.3
> numpy 1.21.0
...

requirements.txt
> pandas 1.3.1
> numpy 1.20.0
...

Dag 1

Worker - task1
Http Op

Worker - task2
Python Op

Scheduler

Dag 2

Worker - task1
Python Op

Worker - task2
MySql Op
Package Dependency management | Problems

- Airflow workers also use same airflow image (v0.1.1)
- install packages required by all DAGs into airflow image?
  - Hard to track which package is used by which DAG
  - Hard to cleanup packages while DAG removal/upgrade
- Create virtual environment for each DAG?
  - ramp-up and ramp-down venvs while DAG starts/stops
  - Slows down the workflow
  - needs extra work to maintain venvs
Package Dependency management | Solution

- Workflow execution: business logics performed inside DAG’s tasks
- Workflow management: Dags and tasks triggering, alerts, monitoring etc

Need Isolation between these two layers... idea?

1. Build docker container to perform business logics
2. Let airflow run those containers

How?

- Docker Operator
- Kubernetes Pod Operator
Docker Operator

Lets you run docker container within same airflow worker node

Pros:
- not populating airflow image with extra packages
- Tasks packages installed in container image
- Reuse same image with different args
- Not bound to python anymore

Cons:
- resource limited - container running in workers

Scheduler

Dag 1

worker - Task 1
Docker Op
image: abc:latest
args: --fetch-data

worker - Task 2
Docker Op
image: abc:latest
args: --load-data

workflow execution

workflow management
Kubernetes Pod Operator

Lets you run docker container on dedicated worker node (pod) in kubernetes cluster

Workflow management

Pod A
- Scheduler

Pod B
- KubernetesPod Op
- Watcher Pod

Pod C
- KubernetesPod Op
- Watcher Pod

Workflow execution

Pod E
- Worker Pod
  - image: abc:latest
  - args: --fetch-data

Pod F
- Worker Pod
  - image: abc:latest
  - args: --fetch-data
Pros & Cons

Docker Operator

- resource limited - container running inside workers
- not populating airflow image with extra packages
- Tasks packages installed in container image
- Not bound to python anymore
- Reuse same image with different args
- Easy to debug

Kubernetes Pod Operator

- Kubernetes knowledge required
- slightly slow - Each task require pair of nodes