cheat-sheet/airflow.md

github.com/cherkavi/cheat-sheet/blob/master/airflow.md

- Airflow apache
- REST API
- source code
- providers, operators
    - base operator parameters
- how to
- OpenSource wrapper CLI
- podcast
- tutorial
- blog
- examples
- best practices
- task additional button on UI
- components

Key concepts

official documentation of key concepts

- DAG a graph object representing your data pipeline (collection of tasks).
  Should be:
    - idempotent (execution of many times without side effect)
    - can be retried automatically
    - toggle should be "turned on" on UI for execution

- Operator describe a single task in your data pipeline
  - **action** - perform actions (airflow.operators.BashOperator, airflow.operators.PythonOperator, airflow.operators.EmailOperator...)
  - **transfer** - move data from one system to another (SftpOperator, S3FileTransformOperator, MySqlOperator, SqliteOperator, PostgresOperator, MsSqlOperator, OracleOperator, JdbcOperator, airflow.operators.HiveOperator....) (don't use it for BigData - source->executor machine->destination)
  - **sensor** - waiting for arriving data to predefined location (airflow.contrib.sensors.file_sensor.FileSensor) has a method #poke that is calling repeatedly until it returns True
- Task An instance of an operator
- Task Instance Represents a specific run of a task = DAG + Task + Point of time
- Workflow Combination of Dags, Operators, Tasks, TaskInstances

configuration, settings

- executor/airflow.cfg
  - remove examples from UI (restart) load_examples = False
  - how much time a new DAGs should be picked up from the filesystem min_file_process_interval = 0 dag_dir_list_interval = 60
- variables

```
from airflow.models import Variable
my_var = Variable.set("my_key", "my_value")
```

- connections as variables

```
from airflow.hooks.base_hook import BaseHook
my_connection = BaseHook.get_connection("name_of_connection")
login = my_connection.login
pass = my_connection.password
```

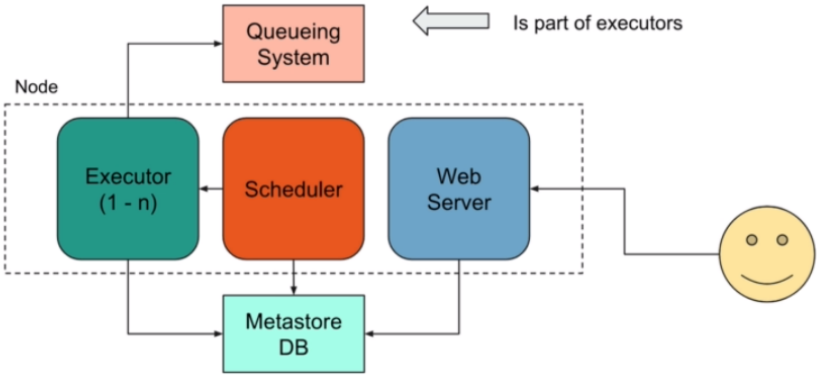
- templating

```
{{ var.value.<variable_key> }}
```

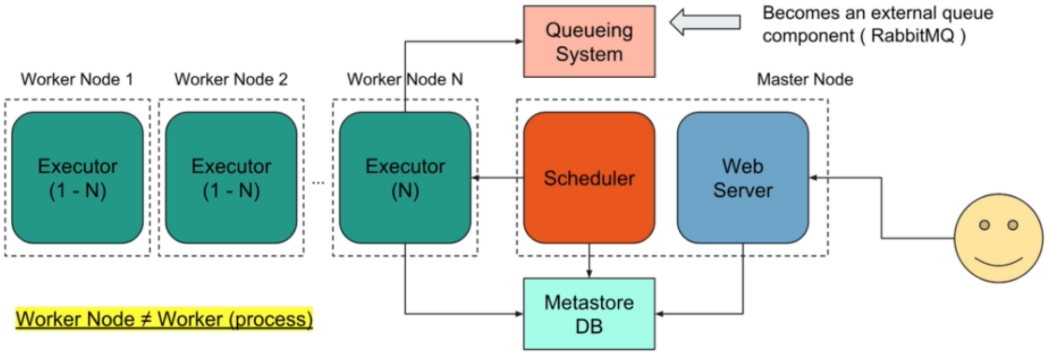
Remember, don’t put any get/set of variables outside of tasks.

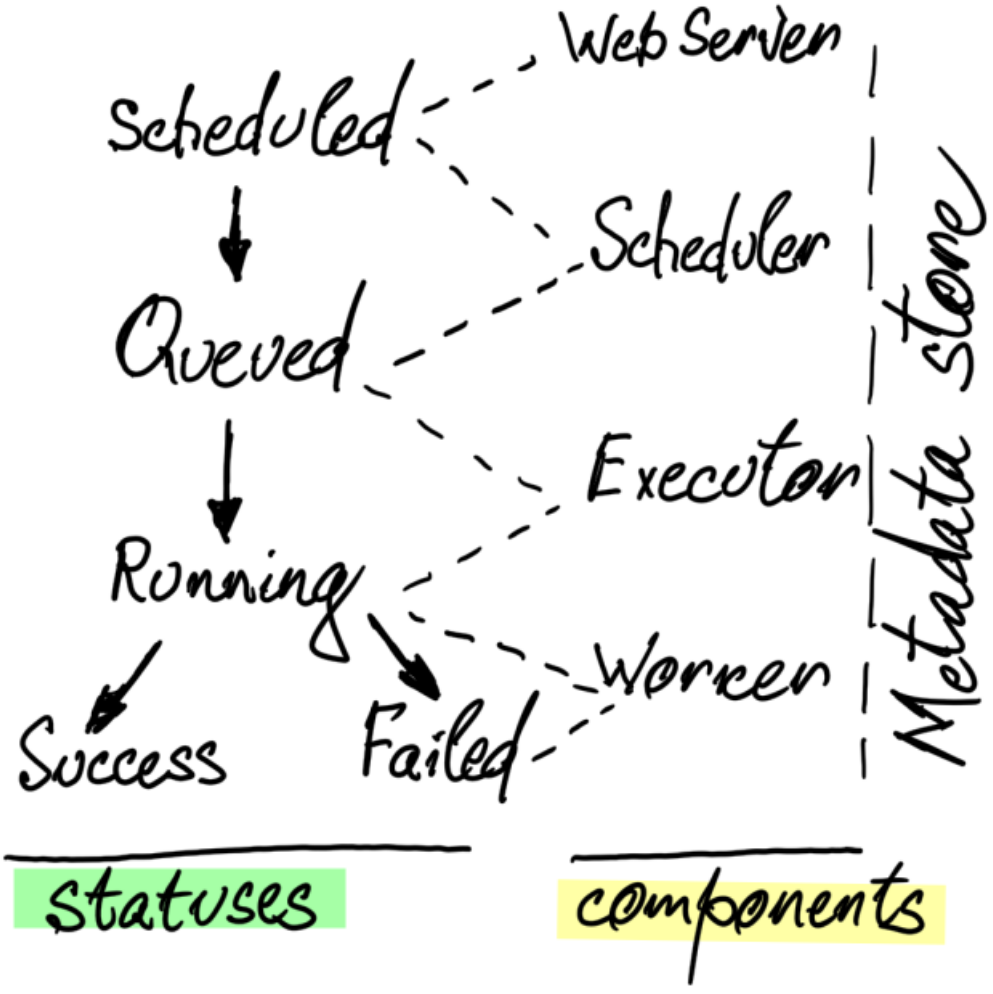
Architecture overview

Architecture Overview (Single Node)

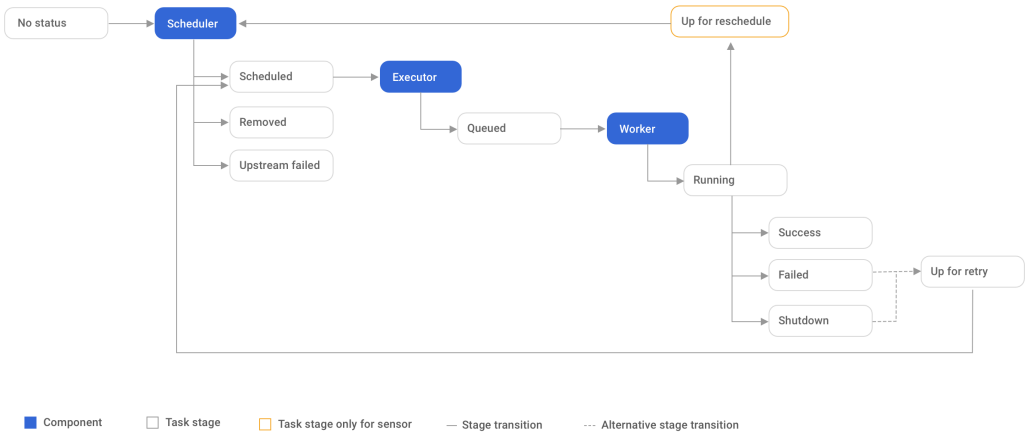


Architecture Overview (Multi Nodes)





to scheduled:
https://github.com/apache/airflow/blob/866a601b76e219b3c043e1dbbc8fb22300866351/airflow/jobs/scheduler_job.py#L810
from scheduled:
https://github.com/apache/airflow/blob/866a601b76e219b3c043e1dbbc8fb22300866351/airflow/jobs/scheduler_job.py#L329
to
queued:https://github.com/apache/airflow/blob/866a601b76e219b3c043e1dbbc8fb22300866351/airflow/jobs/scheduler_job.py#L483



components

- WebServer
  - read user request
  - UI

- Scheduler
  - scan folder "%AIRFLOW%/dags" (config:dag_folder) and with timeout (config:dag_dir_list_interval)
  - monitor execution "start_date" (+ "schedule_interval", first run with start_date), write "execution_date" (last time executed)
  - create DagRun (instance of DAG) and fill DagBag (with interval config:worker_refresh_interval)
    - start_date (start_date must be in past, start_date+schedule_interval must be in future)
    - end_date
    - retries
    - retry_delay
    - schedule_interval (cron:str / datetime.timedelta) (cron presets: @once, @hourly, @daily, @weekly, @monthly, @yearly)
    - catchup (config:catchup_by_default) or "BackFill" (fill previous executions from start_date) actual for scheduler only (backfill is possible via command line)

```
airflow dags backfill -s 2021-04-01 -e 2021-04-05 --
reset_dagruns my_dag_name
```

print snapshot of task state tracked by executor

```
pkill -f -USR2 "airflow scheduler"
```

- Executor (**How** task will be executed, how it will be queued)
  type: LocalExecutor(multiple task in parallel), SequentialExecutor, CeleryExecutor, DaskExecutor
- Worker (**Where** task will be executed)
- Metadatabase (task status)
  - types
  - configuration:
    - sql_alchemy_conn
    - sql_alchemy_pool_enabled

installation

[Airflow install on python virtualenv]

```
# create python virtual env
python3 -m venv airflow-env
source airflow-env/bin/activate

# create folder
mkdir airflow
export AIRFLOW_HOME=`pwd`/airflow

# install workflow
AIRFLOW_VERSION=2.0.2
PYTHON_VERSION=3.8

pip install apache-airflow==$AIRFLOW_VERSION \
 --constraint
"https://raw.githubusercontent.com/apache/airflow/constraints-$AIRFLOW

 # necessary !!!
 exit
```

generate configuration file

```
airflow
```

change configuration file

```
dags_folder = /home/ubuntu/airflow/dags
sql_alchemy_conn = postgresql+psycopg2://airflow:airflow@airflow-
db.cpw.us-east-1.rds.amazonaws.com:5432/airflow
load_examples=False
dag_dir_list_interval = 30
catchup_by_default = False
auth_backend = airflow.api.auth.backend.basic_auth
expose_config = True
dag_run_conf_overrides_params=True

# hide all Rendered Templates
show_templated_fields=none

[webserver]
instance_name = "title name of web ui"
```

[Airflow start on python, nacked start, start components, start separate components, start locally]

```
# installed package
/home/ubuntu/.local/lib/python3.8/site-packages
# full path to airflow
/home/ubuntu/.local/bin/airflow

# init workflow
airflow initdb
# create user first login
airflow users  create --role Admin --username vitalii --email
vcherkashyn@gmail.com --firstname Vitalii --lastname Cherkashyn --
password my_secure_password

# airflow resetdb - for reseting all data
airflow scheduler &
airflow webserver -p 8080 &
echo "localhost:8080"

# check logs
airflow serve_logs

# sudo apt install sqllite3
# sqllite3 $AIRFLOW_HOME/airflow.db
```

Airflow docker

astro cli

```
astro dev init
astro dev start
astro dev ps
astro dev stop

# * copy your dags to ``` .dags```
docker-compose -f docker-compose-LocalExecutor.yml up -d
```

Airflow virtual environment

```
python env create -f environment.yml
source activate airflow-tutorial
```

Airflow Virtual machine

credentials

```
ssh -p 2200 airflow@localhost
# passw: airflow
```

activate workspace

```
source .sandbox/bin/activate
```

update

1. backup DB
2. check you DAG for deprecations

3. upgrade airflow

```
pip install "apache-airflow==2.0.1" --constraint constraint-file
```

4. upgrade DB

```
airflow db upgrade
```

5. restart all

commands

check workspace

```
airflow --help
```

operator types (BaseOperator)

- action
- transfer (data)
- sensor (waiting for some event)
    - long running task
    - BaseSensorOperator
    - poke method is responsible for waiting

Access to DB

!!! create env variables for securing connection

```
Admin -> Connections -> postgres_default
# adjust login, password
Data Profiling->Ad Hoc Query-> postgres_default

select * from dag_run;
```

via PostgreConnection

```
    clear_xcom = PostgresOperator(
        task_id='clear_xcom',
        provide_context=True,
        postgres_conn_id='airflow-postgres',
        trigger_rule="all_done",
        sql="delete from xcom where dag_id LIKE 'my_dag%'",
        dag=dag)
```

REST API

trigger DAG - python

```
import urllib2
import json

AIRFLOW_URL="https://airflow.local/api/experimental/dags/name_of_my_da

payload_dict = {"conf": {"dag_param_1": "test value"}}

req = urllib2.Request(AIRFLOW_URL, data=json.dumps(payload_dict))
req.add_header('Content-Type', 'application/json')
req.add_header('Cache-Control', 'no-cache')
req.get_method = lambda: "POST"
f = urllib2.urlopen(req)
print(f.read())
```

curl request

```
AIRFLOW_ENDPOINT="https://airflow.local/api/experimental"
AIRFLOW_USER=my_user
AIRFLOW_PASSWORD=my_passw

function airflow-event-delete(){
        if [ -z "$1" ]
        then
           echo "first argument should have filename"
           exit 1
        fi

        DAG_NAME="shopify_product_delete"
        DAG_RUN_ID="manual_shopify_product_delete_"`date +%Y-%m-%d-
%H:%M:%S:%s`
        ENDPOINT="$AIRFLOW_URL/api/v1/dags/$DAG_NAME/dagRuns"
        BODY="{\"conf\":
{\"account_id\":\"$ACCOUNT_ID\",\"filename\":\"$1\"},\"dag_run_id\":\"$

        echo $BODY
        curl -H "Content-Type: application/json" --data-binary $BODY -
u $AIRFLOW_USER:$AIRFLOW_PASSWORD -X POST $ENDPOINT
}
```

airflow test connection

```
curl -u $AIRFLOW_USER:$AIRFLOW_PASSWORD -X GET "$ENDPOINT/test"
```

airflow cli commandline console command

https://airflow.apache.org/docs/apache-airflow/stable/usage-cli.html

```
# activation
register-python-argcomplete airflow >> ~/.bashrc
```

```
# dag list
airflow list_dags
airflow list_tasks dag_id
airflow trigger_dag my-dag
# triggering
# https://airflow.apache.org/docs/apache-airflow/1.10.2/cli.html
airflow trigger_dag -c ""  dag_id
```

airflow create dag start dag run dag

doc run in case of removing dag (delete dag) - all metadata will me removed from database

```
# !!! no spaces in request body !!!
REQUEST_BODY='{"conf":{"session_id":"bff2-08275862a9b0"}}'

# ec2-5-221-68-13.compute-
1.amazonaws.com:8080/api/v1/dags/test_dag/dagRuns
curl --data-binary $REQUEST_BODY -H "Content-Type: application/json" -
u $AIRFLOW_USER:$AIRFLOW_PASSWORD -X POST
$AIRFLOW_URL"/api/v1/dags/$DAG_ID/dagRuns"

# run dag from command line

REQUEST_BODY='{"conf":{"sku":"bff2-
08275862a9b0","pool_for_execution":"test_pool2"}}'
DAG_ID="test_dag2"

airflow dags trigger -c $REQUEST_BODY  $DAG_ID
```

airlfow check dag execution

```
curl -X GET -u $AIRFLOW_USER:$AIRFLOW_PASSWORD
"$AIRFLOW_ENDPOINT/dags/$DAG_ID/dagRuns" | jq '.[] | if
.state=="running" then . else empty end'
```

airflow get dag task

```
curl -u $AIRFLOW_USER:$AIRFLOW_PASSWORD -X GET
$AIRFLOW_ENDPOINT"/dags/$DAG_ID/dag_runs/$DATE_DAG_EXEC/tasks/$TASK_ID
```

airflow get task url

```
curl -u $AIRFLOW_USER:$AIRFLOW_PASSWORD -X GET
"$AIRFLOW_ENDPOINT/task?
dag_id=$DAG_ID&task_id=$TASK_ID&execution_date=$DATE_DAG_EXEC"
```

airflow get all dag-runs

```
BODY='{"dag_ids":["shopify_product_create"],"page_limit":30000}'
curl -X POST "$AIRFLOW_URL/api/v1/dags/~/dagRuns/list" -H "Content-
Type: application/json" --data-binary $BODY --user
"$AIRFLOW_USER:$AIRFLOW_PASSWORD" > dag-runs.json
```

get list of dag-runs

```
curl -X GET "$AIRFLOW_URL/api/v1/dags/shopify_product_create/dagRuns"
-H "Content-Type: application/json" --data-binary $BODY --user
"$AIRFLOW_USER:$AIRFLOW_PASSWORD"
```

batch retrieve

```
BODY='{"dag_ids":["shopify_product_create"]}'
curl -X POST "$AIRFLOW_URL/api/v1/dags/~/dagRuns/list" -H "Content-
Type: application/json" --data-binary $BODY --user
"$AIRFLOW_USER:$AIRFLOW_PASSWORD"

DAG_ID=shopify_product_create
TASK_ID=product_create
DAG_RUN_ID=shopify_product_create_2021-06-
15T18:59:35.1623783575Z_6062835
alias get_airflow_log='curl -X GET --user
"$AIRFLOW_USER:$AIRFLOW_PASSWORD"
$AIRFLOW_URL/api/v1/dags/$DAG_ID/dagRuns/$DAG_RUN_ID/taskInstances/$TA
```

get list of tasks

```
BODY='{"dag_ids":["shopify_product_create"],"state":["failed"]}'
curl -X POST "$AIRFLOW_URL/api/v1/dags/~/dagRuns/~/taskInstances/list"
-H "Content-Type: application/json" --data-binary $BODY --user
"$AIRFLOW_USER:$AIRFLOW_PASSWORD"
```

create variable

```
BODY="
{\"key\":\"AWS_ACCESS_KEY_ID\",\"value\":\"${AWS_ACCESS_KEY_ID}\"}"
curl --data-binary $BODY -H  "Content-Type: application/json" --user
"$AIRFLOW_USER:$AIRFLOW_PASSWORD" -X POST $CREATE_VAR_ENDPOINT
```

create pool

```
curl -X POST "$AIRFLOW_URL/api/v1/pools" -H "Content-Type:
application/json" --data '{"name":"product","slots":18}' --user
"$AIRFLOW_USER:$AIRFLOW_PASSWORD"
```

configuration

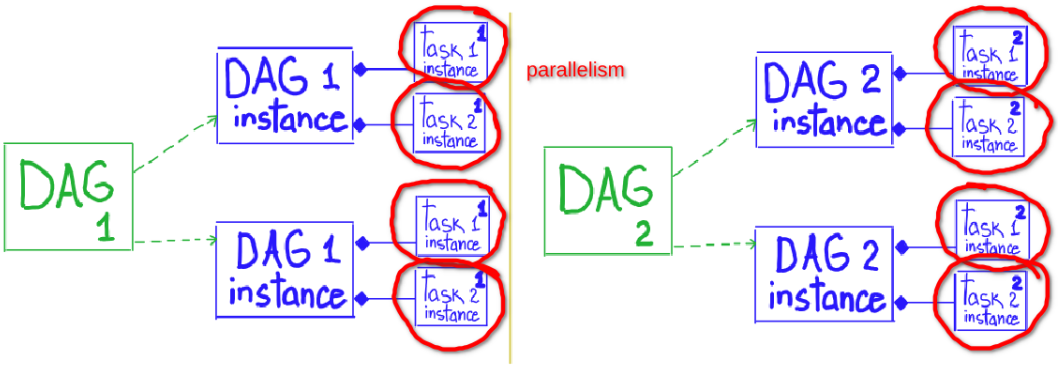
rewrite configuration with environment variables

example of overwriting configuration from config file by env-variables

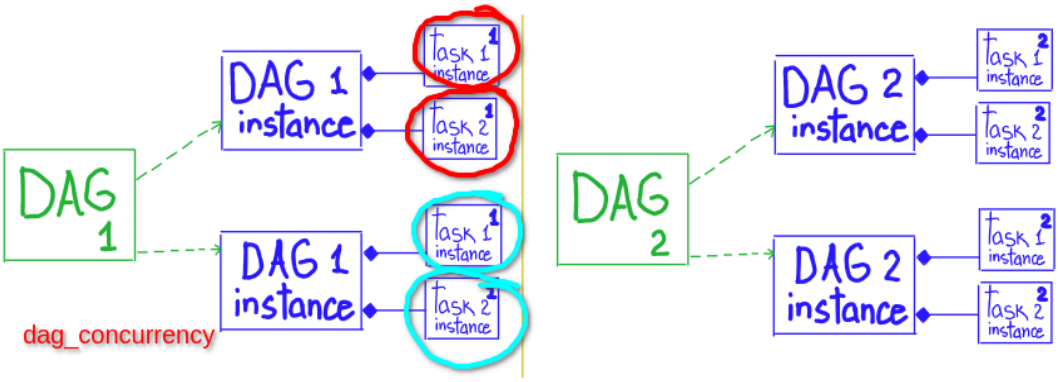
```
[core]
airflow_home='/path/to/airflow'
dags_folder='/path/to/dags'

AIRFLOW__CORE__DAGS_FOLDER='/path/to/new-dags-folder'
AIRFLOW__CORE__AIRFLOW_HOME='/path/to/new-version-of-airflow'
```

multi-tasks

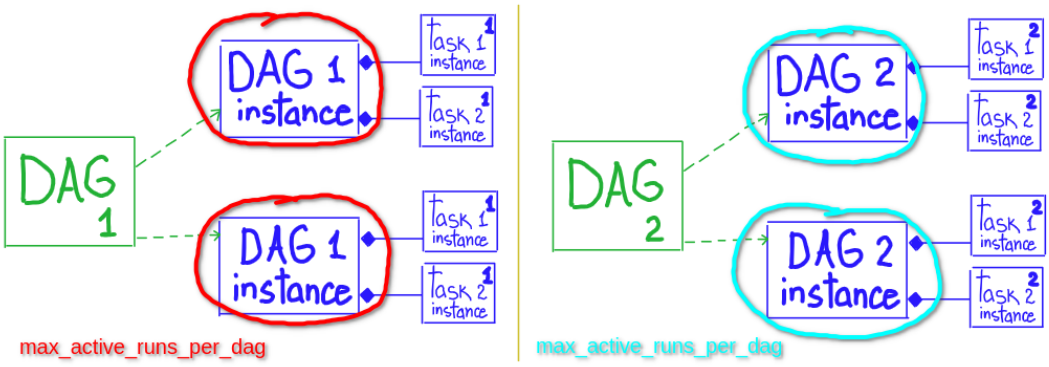
how to speedup airflow




```
# * maximum number of tasks running across an entire Airflow
installation
# * number of physical python processes the scheduler can run, task
(processes) that running in parallel
# scope: Airflow
core.parallelism
```

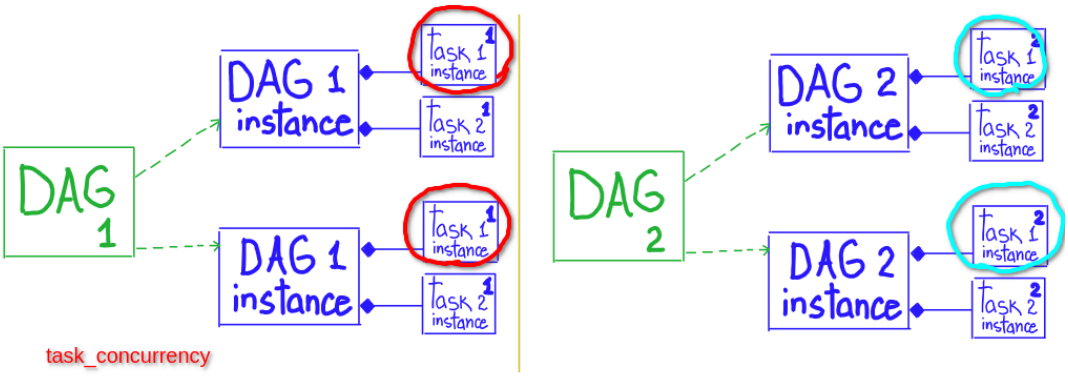



```
# * max number of tasks that can be running per DAG (across multiple
DAG runs)
# * number of tast instances that are running simultaneously per
DagRun ( amount of TaskInstances inside one DagRun )
# scope: DAG.task
core.dag_concurrency
```




```
# * maximum number of active DAG runs, per DAG
# * number of DagRuns - will be concurrency in dag execution, don't
use in case of dependencies of dag-runs
# scope: DAG.instance
core.max_active_runs_per_dag
```

```
# Only allow one run of this DAG to be running at any given time,
default value = core.max_active_runs_per_dag
dag = DAG('my_dag_id', max_active_runs=1)
```




```
# Allow a maximum of 10 tasks to be running across a max of 2 active
DAG runs
dag = DAG('example2', concurrency=10, max_active_runs=2)
# !!! pool: the pool to execute the task in. Pools can be used to
limit parallelism for only a subset of tasks

core.non_pooled_task_slot_count: number of task slots allocated to
tasks not running in a pool
scheduler.max_threads: how many threads the scheduler process should
use to use to schedule DAGs
celery.worker_concurrency: max number of task instances that a worker
will process at a time if using CeleryExecutor
celery.sync_parallelism: number of processes CeleryExecutor should use
to sync task state
```

different configuration of executor

LocalExecutor with PostgreSQL

```
executor = LocalExecutor
sql_alchemy_conn =
postgresql+psycopg2://airflow@localhost:5432/airflow_metadata
```

CeleryExecutor with PostgreSQL and RabbitMQ (recommended for prod)

settings

```
executor = CeleryExecutor
sql_alchemy_conn =
postgresql+psycopg2://airflow@localhost:5432/airflow_metadata
# RabbitMQ UI: localhost:15672
broker_url = pyamqp://admin:rabbitmq@localhost/
result_backend =
db+postgresql://airflow@localhost:5432/airflow_metadata
worker_log_server_port = 8899
```

start Celery worker node

```
# just a start worker process
airflow worker
# start with two child worker process - the same as
'worker_concurrency" in airflow.cfg
airflow worker -c 2
# default pool name: default_pool, default queue name: default
airflow celery worker --queues default
```

normal celery worker output log

```
[2021-07-11 08:23:46,260: INFO/MainProcess] Connected to
amqp://dskcfg:**@toad.rmq.cloudamqp.com:5672/dskcf
[2021-07-11 08:23:46,272: INFO/MainProcess] mingle: searching for
neighbors
[2021-07-11 08:23:47,304: INFO/MainProcess] mingle: sync with 1 nodes
[2021-07-11 08:23:47,305: INFO/MainProcess] mingle: sync complete
[2021-07-11 08:23:47,344: INFO/MainProcess] celery@airflow-01-worker-
01 ready.
```

** in case of adding/removing Celery Workers - restart Airflow Flower **

DAG

task dependencies in DAG

```
# Task1 -> Task2 -> Task3
t1.set_downstream(t2);t2.set_downstream(t3)
t1 >> t2 >> t3

t3.set_upstream(t2);t2.set_upstream(t1)
t3 << t2 << t1

from airflow.models.baseoperator import chain, cross_downstream
chain(t1,t2,t3)
cross_downstream([t1,t2], [t3,t4])

# or set multiply dependency
upstream_tasks = t3.upstream_list
upstream_tasks.append(t2)
upstream_tasks.append(tt1)
upstream_tasks >> t3
```

task information, task metainformation, task context, exchange

```
def python_operator_core_func(**context):
   print(context['task_instance'])
   context["dag_run"].conf['dag_run_argument']
   # the same as previous
   # manipulate with task-instance inside custom function, context
inside custom function
   //  context['ti'].xcom_push(key="k1", value="v1")
   context.get("ti").xcom_push(key="k1", value="v1")

   // and after that pull it and read first value
   // context.get("ti").xcom_pull(task_ids="name_of_task_with_push")
[0]
   // context.get("ti").xcom_pull(task_ids=["name_of_task_with_push",
"name_another_task_to_push"])[0]
   return "value for saving in xcom" # key - return_value
...
PythonOperator(task_id="python_example",
python_callable=python_operator_core_func, provide_context=True,
do_xcom_push=True )
```

task context without context, task jinja template, jinja macros

magic numbers for jinja template

```
def out_of_context_function():
   return_value = ("{{ ti.xcom_pull(task_ids='name_of_task_with_push')
[0] }}")
```

retrieve all values from XCOM

```
from datetime import datetime
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.utils.timezone import make_aware
from airflow.models import XCom

def pull_xcom_call(**kwargs):
    # if you need only TaskInstance: pull_xcom_call(ti)
    # !!! hard-coded value
    execution_date = make_aware(datetime(2020, 7, 24, 23, 45, 17, 00))
    xcom_values = XCom.get_many(dag_ids=["data_pipeline"],
include_prior_dates=True, execution_date=execution_date)
    print('XCom.get_many >>>', xcom_values)

    get_xcom_with_ti = kwargs['ti'].xcom_pull(dag_id="data_pipeline",
include_prior_dates=True)
    print('ti.xcom_pull with include_prior_dates >>>',
get_xcom_with_ti)


xcom_pull_task = PythonOperator(
    task_id='xcom_pull_task',
    dag=dag, # here need to set DAG
    python_callable=pull_xcom_call,
    provide_context=True
)
```

sub-dags

```
from airflow.operators.subdag_operator import SubDagOperator
...
subdag_task =
SubDagOperator(subdag=DAG(SUBDAG_PARENT_NAME+"."+SUBDAG_NAME,schedule_
 start_date=parent_dag.start_date,catchup=False))
...
```

test task

```
airflow tasks test my_dag_name my_task_name 2021-04-01
```

hooks

collaboration with external sources via "connections"
Hooks act as an interface to communicate with the external shared resources in a DAG.

- official airflow hooks
- SparkSubmitHook, FtpHook, JenkinsHook....

XCOM, Cross-communication

GUI: Admin -> Xcoms
Should be manually cleaned up
Exchange information between multiply tasks - "cross communication".

Object must be serializable
Some operators (BashOperator, SimpleHttpOperator, ...) have parameter xcom_push=True - last std.output/http.response will be pushed
Some operators (PythonOperator) has ability to "return" value from function (defined in operator) - will be automatically pushed to XCOM
Saved in Metadabase, also additional data: "execution_date", "task_id", "dag_id"
"execution_date" means hide(skip) everything(same task_id, dag_id...) before this date

```
xcom_push(key="name_of_value", value="some value")
xcom_pull(task_ids="name_of_task_with_push")
```

task state

```
if ti.state not in ["success", "failed", "running"]:
    return None
```

branching, select next step, evaluate next task, condition

!!! don't use "depends_on_past"

```
def check_for_activated_source():
  # return name ( str ) of the task
  return "mysql_task"

branch_task = BranchPythonOperator(task_id='branch_task',
python_callable=check_for_activated_source)
mysql_task      = BashOperator(task_id='mysql_task',
bash_command='echo "MYSQL is activated"')
postgresql_task = BashOperator(task_id='postgresql_task',
bash_command='echo "PostgreSQL is activated"')
mongo_task      = BashOperator(task_id='mongo_task',
bash_command='echo "Mongo is activated"')

branch_task >> mysql_task
branch_task >> postgresql_task
branch_task >> mongo_task
# branch_task >> [mongo_task, mysql_task, postgresql_task]
```

branching with avoiding unexpected run, fix branching

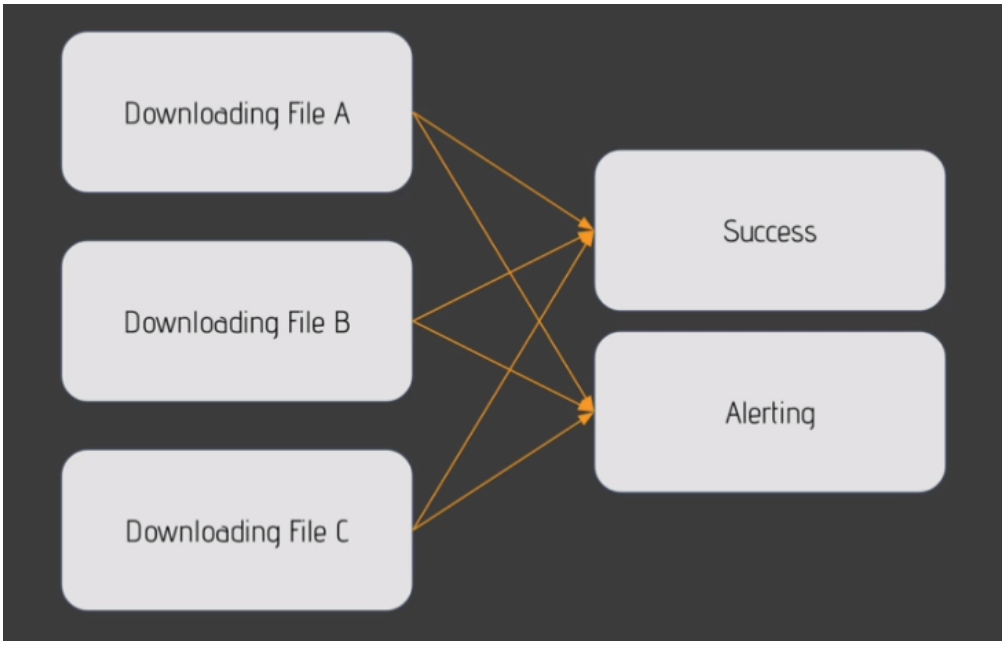
```
from airflow.operators.python_operator import PythonOperator
from airflow.models.skipmixin import SkipMixin

 def fork_label_determinator(**context):
            decision = context['dag_run'].conf.get('branch',
'default')
            return "run_task_1"

        all_tasks = set([task1, task2, task3])
        class SelectOperator(PythonOperator, SkipMixin):
            def execute(self, context):
                condition = super().execute(context)
                self.log.info(">>> Condition %s", condition)
                if condition=="run_task_1":
                    self.skip(context['dag_run'],
context['ti'].execution_date, list(all_tasks-set([task1,])) )
                    return

        # not working properly - applied workaround
        # fork_label = BranchPythonOperator(
        fork_label = SelectOperator(
            task_id=FORK_LABEL_TASK_ID,
            provide_context=True,
            python_callable=fork_label_determinator,
            dag=dag_subdag
        )
```

Trigger rules

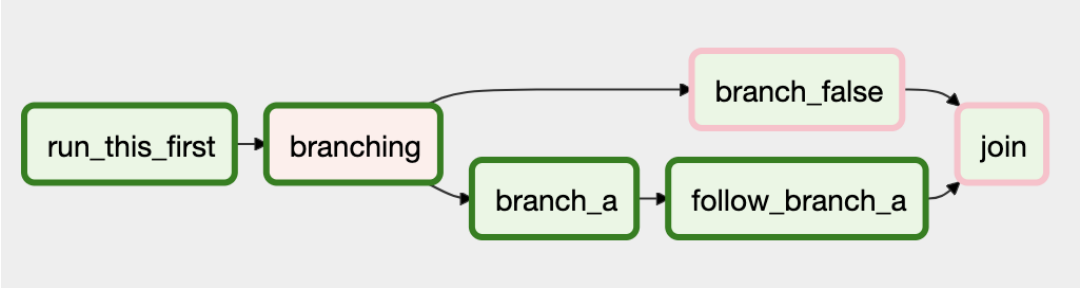
Task States:

- succeed
- skipped

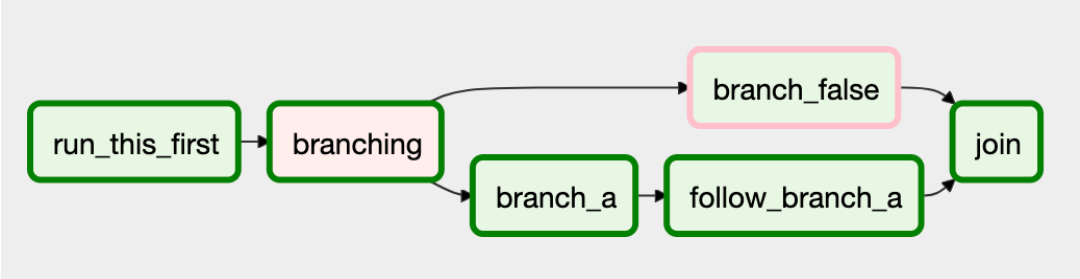
- failed




```
run_this_first >> branching
branching >> branch_a >> follow_branch_a >> join
branching >> branch_false >> join
```

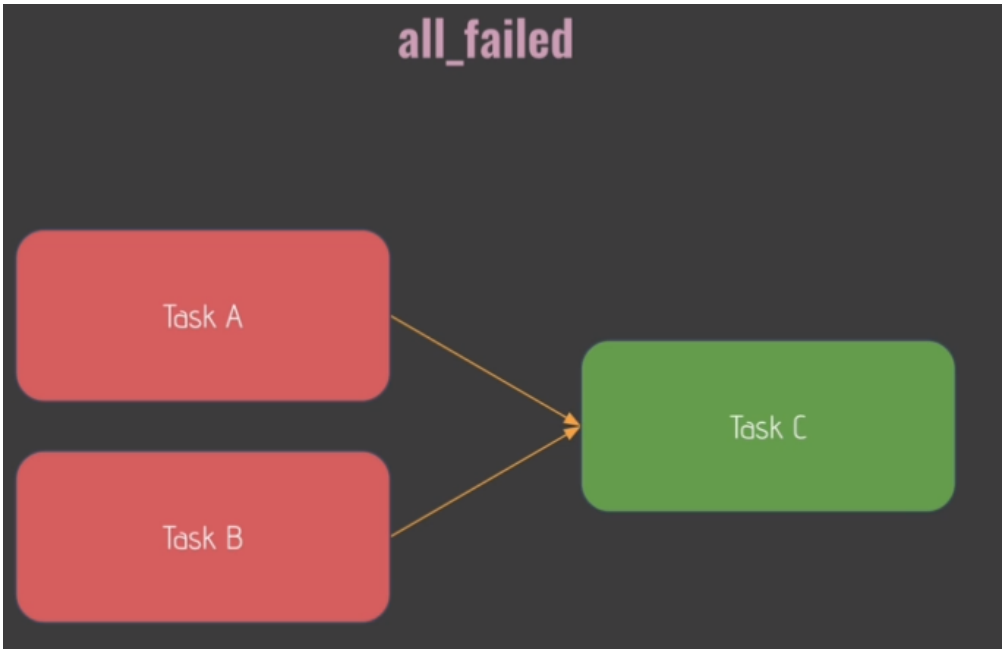


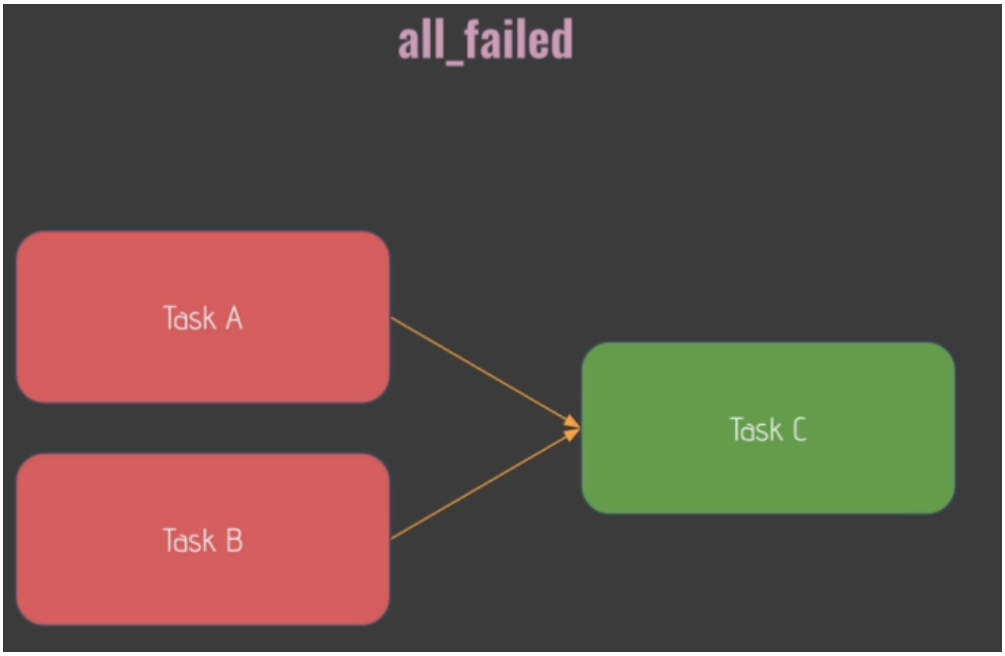

```
join = DummyOperator(task_id='join', dag=dag,
trigger_rule='none_failed_or_skipped')
```

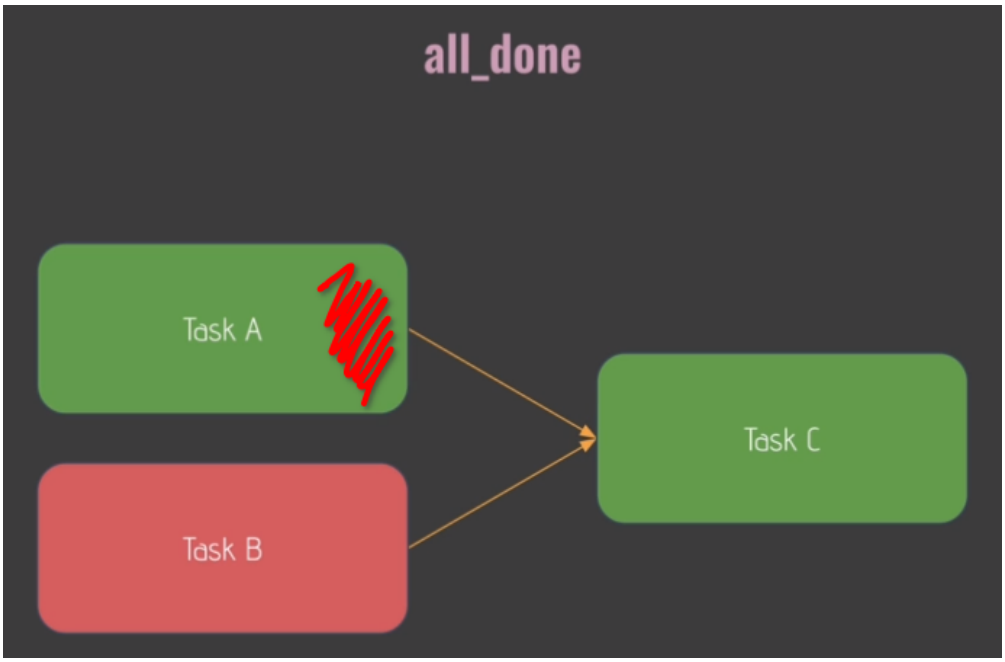
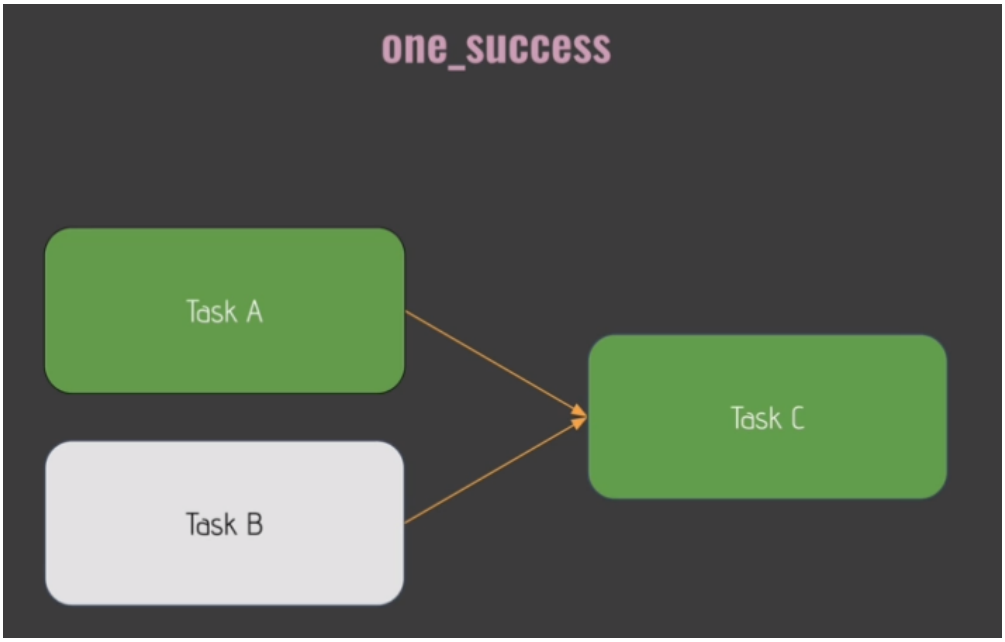
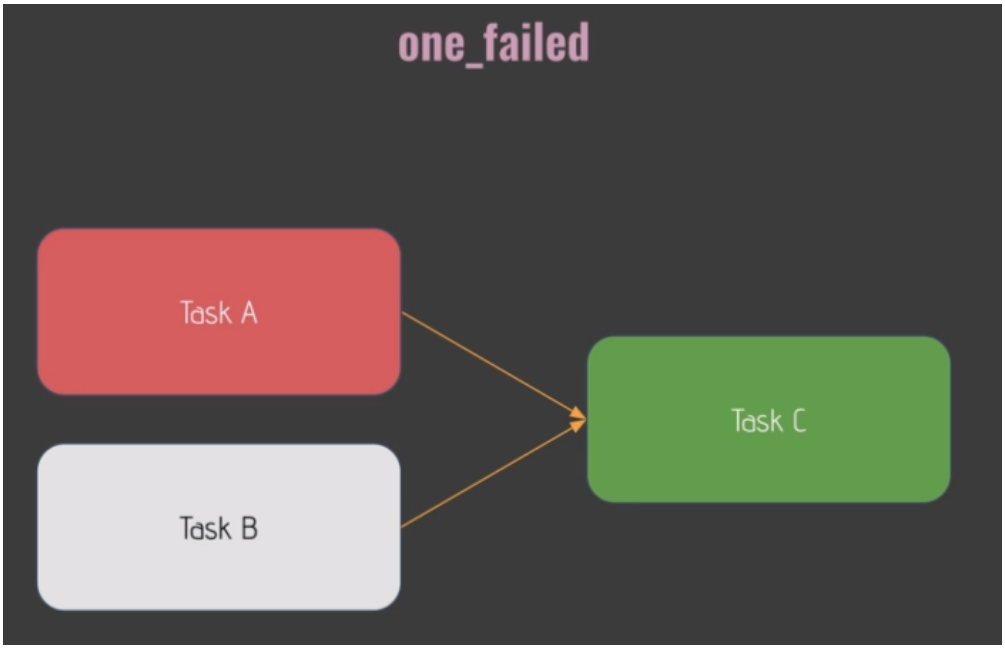


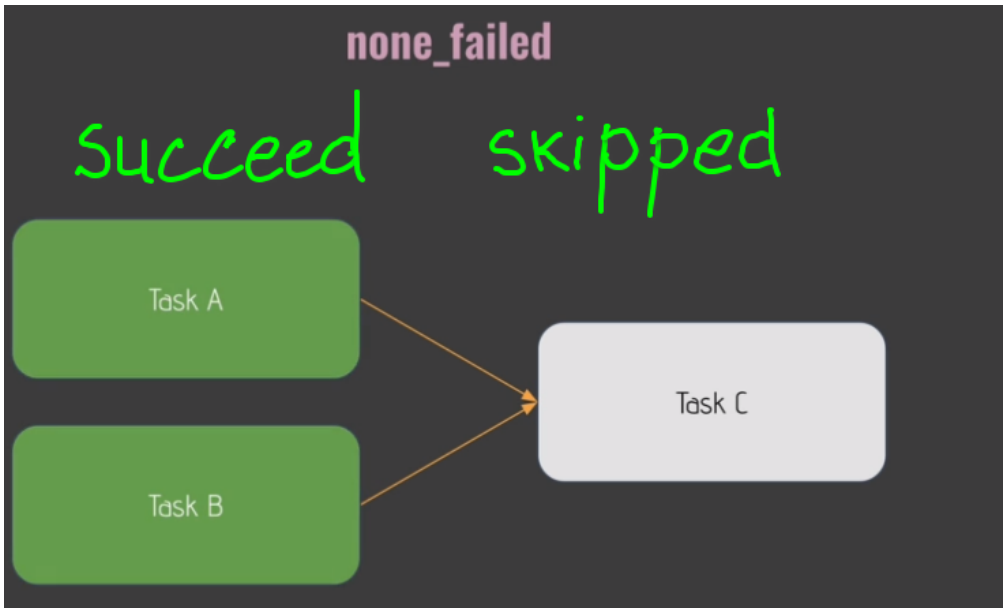
Trigger Rules:

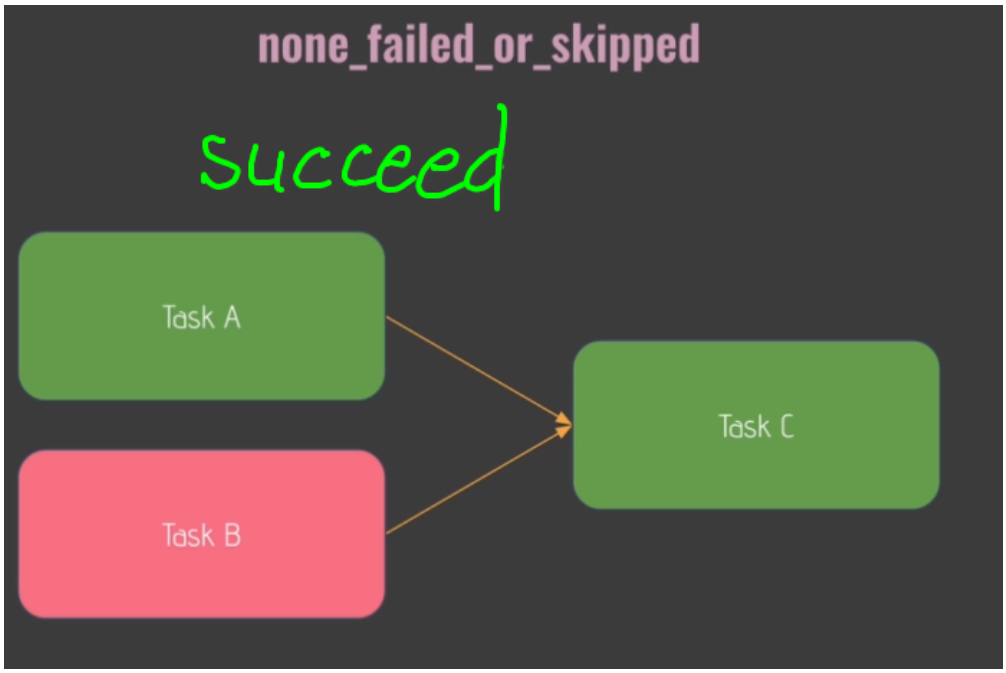
- default:



- 

- 
all_done
Task A
Task B
Task C
- 
one_success
Task A
Task B
Task C
- 
one_failed
Task A
Task B
Task C

- 

- 'none_skipped'
- 

Service Level Agreement, SLA

GUI: Browse->SLA Misses

```
def log_sla_miss(dag, task_list, blocking_task_list, slas,
blocking_tis):
    print("SLA was missed on DAG {0}s by task id {1}s with task list:
{2} which are " \
        "blocking task id {3}s with task list: {4}".format(dag.dag_id,
slas, task_list, blocking_tis, blocking_task_list))

...

# call back function for missed SLA
with DAG('sla_dag', default_args=default_args,
sla_miss_callback=log_sla_miss, schedule_interval="*/1 * * * *",
catchup=False) as dag:
  t0 = DummyOperator(task_id='t0')
        t1 = BashOperator(task_id='t1', bash_command='sleep 15',
sla=timedelta(seconds=5), retries=0)
        t0 >> t1
```

DAG examples

should be placed into "dag" folder (default: %AIRFLOW%/dag)

minimal dag

```
from airflow import DAG
from datetime import datetime, timedelta

with DAG('airflow_tutorial_v01',
         start_date=datetime(2015, 12, 1),
         catchup=False
         ) as dag:
    print(dag)
    # next string will not work !!! only for Task/Operators values
!!!!
    print("{{ dag_run.conf.get('sku', 'default_value_for_sku') }}" )
```

```
from airflow import DAG
from datetime import datetime, timedelta
from airflow.operators.python import PythonOperator
from airflow.utils.dates import days_ago

def print_echo(**context):
    print(context)
    # next string will not work !!! only for Task/Operators values
!!!!
    print("{{ dag_run.conf.get('sku', 'default_value_for_sku') }}" )

with DAG('test_dag',
         start_date=days_ago(100),
         catchup=False,
         schedule_interval=None,
         ) as dag:
    PythonOperator(task_id="print_echo",
                   python_callable=print_echo,
                   provide_context=True,
                   retries=3,
                   retry_delay=timedelta(seconds=30),
                   priority_weight=4,
                   weight_rule=WeightRule.ABSOLUTE, # mandatory for
exected priority behavior
                   # dag_run.conf is not working for pool !!!
                   pool="{{ dag_run.conf.get('pool_for_execution',
'default_pool') }}",
                   # retries=3,
                   # retry_delay=timedelta(seconds=30),
                   doc_md="this is doc for task")

# still not working !!!! impossible to select pool via parameters
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.utils.dates import days_ago

dag = DAG("test_dag2", schedule_interval=None, start_date=days_ago(2))
dag_pool="{{ dag_run.conf['pool_for_execution'] }}"
print(dag_pool)
parameterized_task = BashOperator(
    task_id='parameterized_task',
    queue='collections',
    pool=f"{dag_pool}",
    bash_command=f"echo  {dag_pool}",
    dag=dag,
)
print(f">>> {parameterized_task}")
```

```
        DEFAULT_ARGS = {
    'owner': 'airflow',
    'depends_on_past': True,
    'start_date': datetime(2015, 12, 1),
    'email_on_failure': False,
    'email_on_retry': False,
    # 'retries': 3,
    # 'retry_delay': timedelta(seconds=30),
}

with DAG(DAG_NAME,
         start_date=datetime(2015, 12, 1),
         catchup=False,
         catchup=True,
         schedule_interval=None,
         max_active_runs=1,
         concurrency=1,
         default_args=DEFAULT_ARGS
         ) as dag:
    PythonOperator(task_id="image_set_variant",
                   python_callable=image_set_variant,
                   provide_context=True,
                   retries=3,
                   retry_delay=timedelta(seconds=30),
                   # retries=3,
                   # retry_delay=timedelta(seconds=30),
                   #
https://github.com/apache/airflow/blob/866a601b76e219b3c043e1dbbc8fb22

                   # priority_weight=1 default is 1, more high will be
executed earlier
                   doc_md="this is doc for task")

# task concurrency
t1 = BaseOperator(pool='my_custom_pool', task_concurrency=12)
```

simple DAG

```
from airflow import DAG
from datetime import date, timedelta, datetime
from airflow.models import BaseOperator
from airflow.operators.bash_operator import BashOperator
# airflow predefined intervals
from airflow.utils.dates import days_ago

def _hook_failure(error_contect):
  print(error_context)


# default argument for each task in DAG
default_arguments = {
    'owner': 'airflow'
    ,'retries': 1
    ,'retry_delay': timedelta(minutes=5)
    ,'email_on_failure':True
    ,'email_on_retry':True
    ,'email': "my@one.com" # smtp server must be set up
    ,'on_failure_callback': _hook_failure
}

# when schedule_interval=None, then execution of DAG possible only
with direct triggering
with DAG(dag_id='dummy_echo_dag_10'
          ,default_args=default_arguments
          ,start_date=datetime(2016,1,1) # do not do that:
datetime.now() # days_ago(3)
          ,schedule_interval="*/5 * * * *"
          ,catchup=False # - will be re-writed from ConfigFile !!!
          ,depends_on_past=False
         ) as dag:
    # not necessary to specify dag=dag, source code inside
BaseOperator:
    # self.dag = dag or DagContext.get_current_dag()
    BashOperator(task_id='bash_example', bash_command="date", dag=dag)
```

reading data from api call https://airflow.apache.org/docs/apache-airflow/2.0.1/dag-run.html#external-triggers

```
value_from_rest_api_call='{{ dag_run.conf["session_id"] }}'
# or
kwargs['dag_run'].conf.get('session_id',
'default_value_for_session_id')
```

reading settings files (dirty way)

```
# settings.json should be placed in the same folder as dag description
# configuration shoulhttps://github.com/cherkavi/cheat-
sheet/blob/master/development-process.md#concurrency-vs-parallelismd
contains: dags_folder = /usr/local/airflow/dags
def get_request_body():
    with open(f"
{str(Path(__file__).parent.parent)}/dags/settings.json", "r") as f:
        request_body = json.load(f)
        return json.dumps(request_body)
```

collaboration between tasks, custom functions

```
COLLABORATION_TASK_ID="mydag_first_call"

def status_checker(resp):
    job_status = resp.json()["status"]
    return job_status in ["SUCCESS", "FAILURE"]

def cleanup_response(response):
    return response.strip()

def create_http_operator(connection_id=MYDAG_CONNECTION_ID):
    return SimpleHttpOperator(
        task_id=COLLABORATION_TASK_ID,
        http_conn_id=connection_id,
        method="POST",
        endpoint="v2/endpoint",
        data="{\"id\":111333222}",
        headers={"Content-Type": "application/json"},
        # response will be pushed to xcom with COLLABORATION_TASK_ID
        xcom_push=True,
        log_response=True,
    )


def second_http_call(connection_id=MYDAG_CONNECTION_ID):
    return HttpSensor(
        task_id="mydag_second_task",
        http_conn_id=connection_id,
        method="GET",
        endpoint="v2/jobs/{{ parse_response(ti.xcom_pull(task_ids='" +
COLLABORATION_TASK_ID + "' )) }}",
        response_check=status_checker,
        poke_interval=15,
        depends_on_past=True,
        wait_for_downstream=True,
    )


with DAG(
    default_args=default_args,
    dag_id="dag_name",
    max_active_runs=1,
    default_view="graph",
    concurrency=1,
    schedule_interval=None,
    catchup=False,
    # custom function definition
    user_defined_macros={"parse_response": cleanup_response},
) as dag:
    first_operator = first_http_call()
    second_operator = second_http_call()
    first_operator >> second_operator
```

avoid declaration of Jinja inside parameters

```
    # api_endpoint = "{{ dag_run.conf['session_id'] }}"
    maprdb_read_session_metadata = SimpleHttpOperator(
        task_id=MAPRDB_REST_API_TASK_ID,
        method="GET",
        http_conn_id="{{ dag_run.conf['session_id'] }}",
        # sometimes not working and need to create external variable
like api_endpoint !!!!
        endpoint="{{ dag_run.conf['session_id'] }}",
        data={"fields": [JOB_CONF["field_name"], ]},
        log_response=True,
        xcom_push=True
```

logging, log output, print log

```
import logging
logging.info("some logs")
```

logging for task, task log

```
 task_instance = context['ti']
 task_instance.log.info("some logs for task")
```

execute list of tasks from external source, subdag, task loop

```
def trigger_export_task(session, uuid, config):
    def trigger_dag(context: Dict, dag_run: DagRunOrder) ->
DagRunOrder:
        dag_run.payload = config
        return dag_run

    return AwaitableTriggerDagRunOperator(
        trigger_dag_id=DAG_ID_ROSBAG_EXPORT_CACHE,
        task_id=f"{session}_{uuid}",
        python_callable=trigger_dag,
        trigger_rule=TriggerRule.ALL_DONE,
    )

# DAG Definition
with DAG(
    dag_id=DAG_NAME_WARMUP_ROSBAG_EXPORT_CACHE,
    default_args={"start_date": datetime(2020, 4, 20),
**DEFAULT_DAG_PARAMS},
    default_view="graph",
    orientation="LR",
    doc_md=__doc__,
    schedule_interval=SCHEDULE_DAILY_AFTERNOON,
    catchup=False,
) as dag:
    # generate export configs
    dag_modules = _get_dag_modules_containing_sessions()
    export_configs = _get_configs(dag_modules, NIGHTLY_SESSION_CONFIG)

    # generate task queues/branches
    NUM_TASK_QUEUES = 30
    task_queues = [[] for i in range(NUM_TASK_QUEUES)]

    # generate tasks (one task per export config) and assign them to
queues/branches (rotative)
    for i, ((session, uuid), conf) in
enumerate(export_configs.items()):
        queue = task_queues[i % NUM_TASK_QUEUES]
        queue.append(trigger_export_task(session, uuid, conf))

        # set dependency to previous task
        if len(queue) > 1:
            queue[-2] >> queue[-1]
```

task branching, task logic of moving, tasks order execution depends on parameters https://www.astronomer.io/guides/airflow-branch-operator/

```
with DAG(default_args=DAG_DEFAULT_ARGS,
         dag_id=DAG_CONFIG['dag_id'],
         schedule_interval=DAG_CONFIG.get('schedule_interval', None))
as dag:

    def return_branch(**kwargs):
        """
        start point (start task) of the execution
        ( everything else after start point will be executed )
        """
        decision = kwargs['dag_run'].conf.get('branch',
'run_markerers')
        if decision == 'run_markerers':
            return 'run_markerers'
        if decision == 'merge_markers':
            return 'merge_markers'
        if decision == 'index_merged_markers':
            return 'index_merged_markers'
        if decision == 'index_single_markers':
            return 'index_single_markers'
        if decision == 'index_markers':
            return ['index_single_markers', 'index_merged_markers']
        else:
            return 'run_markerers'


    fork_op = BranchPythonOperator(
        task_id='fork_marker_jobs',
        provide_context=True,
        python_callable=return_branch,
    )

    run_markerers_op = SparkSubmitOperator(
        task_id='run_markerers',
        trigger_rule='none_failed',
    )

    merge_markers_op = SparkSubmitOperator(
        task_id='merge_markers',
        trigger_rule='none_failed',
    )

    index_merged_markers_op = SparkSubmitOperator(
        task_id='index_merged_markers',
        trigger_rule='none_failed',
    )

    index_single_markers_op = SparkSubmitOperator(
        task_id='index_single_markers',
        trigger_rule='none_failed',
    )

    fork_op >> run_markerers_op >> merge_markers_op >>
index_merged_markers_op
    run_markerers_op >> index_single_markers_op
```

access to dag runs, access to dag instances, set dags state

```
from airflow.models import DagRun
from airflow.operators.python_operator import PythonOperator
from airflow.utils.db import provide_session
from airflow.utils.state import State
from airflow.utils.trigger_rule import TriggerRule

@provide_session
# custom parameter for operator
def stop_unfinished_dag_runs(trigger_task_id, session=None,
**context):
    print(context['my_custom_param'])
    dros = context["ti"].xcom_pull(task_ids=trigger_task_id)
    run_ids = list(map(lambda dro: dro.run_id, dros))

    # identify unfinished DAG runs of rosbag_export
    dr = DagRun
    running_dags = session.query(dr).filter(dr.run_id.in_(run_ids),
dr.state.in_(State.unfinished())).all()

    if running_dags and len(running_dags)>0:
        # set status failed
        for dag_run in running_dags:
            dag_run.set_state(State.FAILED)
        print("set unfinished DAG runs to FAILED")


def dag_run_cleaner_task(trigger_task_id):
    return PythonOperator(
        task_id=dag_config.DAG_RUN_CLEAN_UP_TASK_ID,
        python_callable=stop_unfinished_dag_runs,
        provide_context=True,
        op_args=[trigger_task_id], # custom parameter for operator
        op_kwargs={"my_custom_param": 5}
    )
```

python operator new style

```
from airflow.operators.python import get_current_context

@task
def image_set_variant():
    context = get_current_context()
    task_instance = context["ti"]


with DAG(DAG_NAME,
         start_date=datetime(2015, 12, 1),
         catchup=False,
         schedule_interval=None
         ) as dag:
    image_set_variant()
```

trig and wait, run another dag and wait

```
from airflow.models import BaseOperator
from airflow.operators.dagrun_operator import DagRunOrder

from airflow_common.operators.awaitable_trigger_dag_run_operator
import \
    AwaitableTriggerDagRunOperator
from airflow_dags_manual_labeling_export.ad_labeling_export.config
import \
    dag_config


def _run_another(context, dag_run_obj: DagRunOrder):
    # config from parent dag run
    config = context["dag_run"].conf.copy()
    config["context"] = dag_config.DAG_CONTEXT
    dag_run_obj.payload = config

    dag_run_obj.run_id = f"
{dag_config.DAG_ID}_triggered_{context['execution_date']}"
    return dag_run_obj


def trig_another_dag() -> BaseOperator:
    """
    trig another dag
    :return: initialized TriggerDagRunOperator
    """
    return AwaitableTriggerDagRunOperator(
        task_id="task_id",
        trigger_dag_id="dag_id",
        python_callable=_run_another,
        do_xcom_push=True,
    )
```

read input parameters from REST API call

```
DAG_NAME="my_dag"
PARAM_1="my_own_param1"
PARAM_2="my_own_param2"
ENDPOINT="https://prod.airflow.vantage.zur/api/experimental/dags/$DAG_

BODY='{"configuration_of_call":
{"parameter1":"'$PARAM_1'","parameters2":"'$PARAM_2'"}}'
curl --data-binary $BODY -u $AIRFLOW_USER:$AIRFLOW_PASSWORD -X POST
$ENDPOINT

decision = context['dag_run'].configuration_of_call.get('parameter1',
'default_value')
```

read system configuration

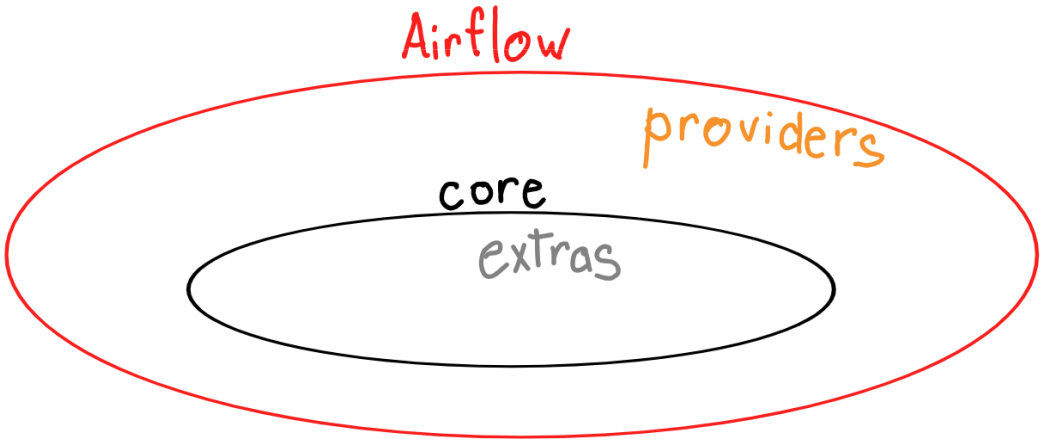
```
from airflow.configuration import conf
# Secondly, get the value somewhere
conf.get("core", "my_key")

# Possible, set a value with
conf.set("core", "my_key", "my_val")
```

sensor example

```
SensorFile(
  task_id="sensor_file",
  fs_conn_id="filesystem_connection_id_1", # Extras should have:
{"path":"/path/to/folder/where/file/is/"}
  file_path="my_file_name.txt"
)
```

smart skip, skip task

```
from airflow.models import DAG
from airflow.operators.python_operator import BranchPythonOperator
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import PythonOperator
from airflow.models.skipmixin import SkipMixin

class SelectOperator(PythonOperator, SkipMixin):

    def _substract_by_taskid(self, task_list, filtered_ids):
        return filter( lambda task_instance: task_instance.task_id not
in filtered_ids, task_list);

    def execute(self, context):
        condition = super().execute(context)
        # self.skip(context['dag_run'], context['ti'].execution_date,
downstream_tasks)

        self.log.info(">>>  SelectOperator")
        self.log.info(">>> Condition %s", condition)
        downstream_tasks =
context['task'].get_flat_relatives(upstream=False)

        # self.log.info(">>> Downstream task_ids %s",
downstream_tasks)
        # filtered_tasks =
list(self._substract_by_taskid(downstream_tasks, condition))
        # self.log.info(">>> Filtered task_ids %s", filtered_tasks)
        # self.skip(context['dag_run'], context['ti'].execution_date,
filtered_tasks)

        self.skip_all_except(context['ti'], condition)
        self.log.info(">>>>>>>>>>>>>>>>>>>>")

with DAG('autolabelling_example', description='First DAG',
schedule_interval=None, start_date=datetime(2018, 11, 1),
catchup=False) as dag:
    def fork_label_job_branch(**context):
        return ['index_single_labels']

    fork_operator = SelectOperator(task_id=FORK_LABEL_TASK_ID,
provide_context=True, python_callable=fork_label_job_branch)
```



Providers

```
pip install apache-airflow-providers-presto
```

Plugins

official documentation
examples of airflow plugins

- Operators: They describe a single task in a workflow. Derived from BaseOperator.
- Sensors: They are a particular subtype of Operators used to wait for an event to happen. Derived from BaseSensorOperator
- Hooks: They are used as interfaces between Apache Airflow and external systems. Derived from BaseHook
- Executors: They are used to actually execute the tasks. Derived from BaseExecutor
- Admin Views: Represent base administrative view from Flask-Admin allowing to create web interfaces. Derived from flask_admin.BaseView (new page = Admin Views + Blueprint)
- Blueprints: Represent a way to organize flask application into smaller and re-usable application. A blueprint defines a collection of views, static assets and templates. Derived from flask.Blueprint (new page = Admin Views + Blueprint)
- Menu Link: Allow to add custom links to the navigation menu in Apache Airflow. Derived from flask_admin.base.MenuLink
- Macros: way to pass dynamic information into task instances at runtime. They are tightly coupled with Jinja Template.

plugin template

```
# init.py

from airflow.plugins_manager import AirflowPlugin
from elasticsearch_plugin.hooks.elasticsearch_hook import
ElasticsearchHook

# Views / Blueprints / MenuLinks are instantied objects
class MyPlugin(AirflowPlugin):
        name                    = "my_plugin"
        operators               = [MyOperator]
        sensors                 = []
        hooks                   = [MyHook]
        executors               = []
        admin_views             = []
        flask_blueprints        = []
        menu_links              = []

my_plugin/
├── __init__.py
├── hooks
│   ├── my_hook.py
│   └── __init__.py
├── menu_links
│   ├── my_link.py
│   └── __init__.py
├── operators
    ├── my_operator.py
    └── __init__.py
```

Maintenance

Metedata cleanup

```
-- https://github.com/teamclairvoyant/airflow-maintenance-
dags/blob/master/db-cleanup/airflow-db-cleanup.py

-- "airflow_db_model": BaseJob.latest_heartbeat,
select count(*) from job where latest_heartbeat < (CURRENT_DATE -
INTERVAL '5 DAY')::DATE;

-- "airflow_db_model": DagRun.execution_date,
select count(*) from dag_run where execution_date < (CURRENT_DATE -
INTERVAL '5 DAY')::DATE;

-- "airflow_db_model": TaskInstance.execution_date,
select count(*) from task_instance where execution_date <
(CURRENT_DATE - INTERVAL '5 DAY')::DATE;

-- "airflow_db_model": Log.dttm,
select count(*) from log where dttm < (CURRENT_DATE - INTERVAL '5
DAY')::DATE;

-- "age_check_column": XCom.execution_date,
select count(*) from xcom where execution_date < (CURRENT_DATE -
INTERVAL '5 DAY')::DATE;

-- "age_check_column": SlaMiss.execution_date,
select count(*) from sla_miss where execution_date < (CURRENT_DATE -
INTERVAL '5 DAY')::DATE;

-- "age_check_column": TaskReschedule.execution_date,
select count(*) from task_reschedule where execution_date <
(CURRENT_DATE - INTERVAL '5 DAY')::DATE;

-- "age_check_column": TaskFail.execution_date,
select count(*) from task_fail where execution_date < (CURRENT_DATE -
INTERVAL '5 DAY')::DATE;

-- "age_check_column": RenderedTaskInstanceFields.execution_date,
select count(*) from rendered_task_instance_fields where
execution_date < (CURRENT_DATE - INTERVAL '5 DAY')::DATE;


------------------------------------------------------------------------
-------------------------------------
delete from job where latest_heartbeat < (CURRENT_DATE - INTERVAL '5
DAY')::DATE;y
delete from dag_run where execution_date < (CURRENT_DATE - INTERVAL '5
DAY')::DATE;y
delete from task_instance where execution_date < (CURRENT_DATE -
INTERVAL '5 DAY')::DATE;y
delete from log where dttm < (CURRENT_DATE - INTERVAL '5 DAY')::DATE;y
delete from xcom where execution_date < (CURRENT_DATE - INTERVAL '5
DAY')::DATE;y
delete from sla_miss where execution_date < (CURRENT_DATE - INTERVAL
'5 DAY')::DATE;y
delete from task_reschedule where execution_date < (CURRENT_DATE -
INTERVAL '5 DAY')::DATE;y
delete from task_fail where execution_date < (CURRENT_DATE - INTERVAL
'5 DAY')::DATE;y
delete from rendered_task_instance_fields where execution_date <
(CURRENT_DATE - INTERVAL '5 DAY')::DATE;y
```