AGLOW Documentation

Release 0.0.9

Alexandar Mechev

Apr 06, 2020

Contents:

This package is built by Alexandar Mechev and the LOFAR e-infra group at Leiden University with the support of SURFsara. The goals of this package is to enable High Throughput processing of LOFAR data on the Dutch Grid infrastructure. We do this by making a set of tools designed to wrap around several different LOFAR processing strategies. These tools are responsible for staging data at the LOFAR Long Term Archives, creating and launching Grid jobs on the Gina cluster, as well as managing intermediate data on the SURFsara dCache storage.

This software combines Apache Ariflow.. and the Grid LOFAR Tools..

CHAPTER 1

Installation

Note:

Some AGLOW operators use GRID_LRT and
so you need the correct GRID_LRT version to use them

1.1 Via Python Package Index

Install the package (or add it to your `requirements.txt` file):

```
pip install AGLOW
```

1.2 Via Git or Download

Download the latest version from `https://www.github.com/apmechev/AGLOW`. To install, use

```
python setup.py build
python setup.py install
```

In the case that you do not have access to the python system libraries, you can use `--prefix=` to specify install folder. For example if you want to install it into a folder you own (say /home/apmechev/software/python) use the following command:

```
python setup.py build
python setup.py install --prefix=${HOME}/software/python
```

Note: NOTE: you need to have your pythonpath containing

"${HOME}/software/python/lib/python[2.6|2.7|3.4]/site_packages"
and that folder needs to exist beforehand or setuptools will complain

CHAPTER 2

AGLOW utils

The AGLOW utils module includes small functions that can be used with a PythonOperator to interface with LOFAR data, LOFAR fields processing, GRID storage and others.

AGLOW.airflow.utils.AGLOW_utils.**archive_all_tokens**(*token_type*, *archive_location*, *delete=False*)

AGLOW.airflow.utils.AGLOW_utils.**archive_tokens_from_task**(*token_task*, *delete=False*, ***context*)

Determines whic tokens to archive and saves them. delete if necessary

AGLOW.airflow.utils.AGLOW_utils.**check_folder_for_files**(*folder*, *number=1*)

Raises an exception (IE FAILS) if the (gsiftp) folder has less than 'number' files

By default fails if folder is empty

AGLOW.airflow.utils.AGLOW_utils.**check_folder_for_files_from_task**(*taskid*, *xcom_key*, *number*, ***context*)

Either uses number to see how many files should be there or checks the number of tokens in the view TODO:make this the right way

AGLOW.airflow.utils.AGLOW_utils.**check_folder_for_files_from_tokens**(*task_id*, *dummy*, *number*, ***context*)

AGLOW.airflow.utils.AGLOW_utils.**check_if_running_field**(*fields_file*)

Checks if there are fields that are running (ie not completed, or Error) Returns True if running, False if not

AGLOW.airflow.utils.AGLOW_utils.**copy_gsifile**(*src*, *dest*)

AGLOW.airflow.utils.AGLOW_utils.**copy_to_archive**(*src_dir=u'gsiftp://gridftp.grid.sara.nl:2811/pnfs/grid.sara.nl/d*
dest_dir=u'gsiftp://gridftp.grid.sara.nl:2811/pnfs/grid.sara.nl/
***context*)

AGLOW.airflow.utils.AGLOW_utils.**count_files_uberftp**(*directory*)

AGLOW.airflow.utils.AGLOW_utils.**count_from_task**(*srmlist_task*, *srmfile_name*, *task_if_less*, *task_if_more*, *pipeline=u'SKSP'*, *step=u'pref_cal2'*, *min_num_files=1*, *parent_dag=False*, ***context*)

AGLOW.airflow.utils.AGLOW_utils.**count_grid_files**(*srmlist_file*, *task_if_less*, *task_if_more*, *pipeline=u'SKSP'*, *step=u'pref_cal1'*, *min_num_files=1*)

An airflow function that branches depending on whether there are calibrator or target solutions matching the files in the srmlist.

AGLOW.airflow.utils.AGLOW_utils.**create_gsiftp_directory**(*gsiftp_directory*)

AGLOW.airflow.utils.AGLOW_utils.**delete_gsiftp_files**(*gsiftp_directory*)

AGLOW.airflow.utils.AGLOW_utils.**delete_gsiftp_from_task**(*root_dir*, *OBSID_task*, ***context*)

A task that returns OBSID, and a root directory come together in this function to gracefully delete all files in this here directory

AGLOW.airflow.utils.AGLOW_utils.**get_cal_from_dir**(*base_dir*, *return_key=None*, ***context*)

AGLOW.airflow.utils.AGLOW_utils.**get_field_location_from_srmlist**(*srmlist_task*, *srmfile_key=u'targ_srmfile'*, ***context*)

Gets the srmlist from a task and returns the location of the field IE the LTA location where the raw data is stored

AGLOW.airflow.utils.AGLOW_utils.**get_list_from_dir**(*base_dir*, ***context*)

AGLOW.airflow.utils.AGLOW_utils.**get_next_field**(*fields_file*, *indicator=u'SND'*, ***context*)

Determines the next field to be processed, uses the set_field_status function to set its status to started(). By default it locks SARA tokens (marked SND in the first column of the fields_file), however using the indicator variable, other files can be locked

AGLOW.airflow.utils.AGLOW_utils.**get_result_files_from_tokenlist**(*token_type*, *token_ids*, *key=u'Results_location'*, ***kwargs*)

AGLOW.airflow.utils.AGLOW_utils.**get_results_from_subdag**(*subdag_id*, *task=u'tokens'*, *key=u'Results_location'*, *return_key=None*, ***context*)

AGLOW.airflow.utils.AGLOW_utils.**get_srmfile_from_dir**(*srmdir*, *field_task*, *var_calib=u'SKSP_Prod_Calibrator_srm_file'*, *var_targ=u'SKSP_Prod_Target_srm_file'*, ***context*)

AGLOW.airflow.utils.AGLOW_utils.**get_task_instance**(*context*, *key*, *parent_dag=False*)

AGLOW.airflow.utils.AGLOW_utils.**get_user_proxy**(*username*)

Gets the X509 file by the user in order to perform authorized operations by them as opposed as by the DAG Executor

AGLOW.airflow.utils.AGLOW_utils.**get_var_from_task_decorator**(*Cls*, *upstream_task_id=u"*, *upstream_return_key=u"*, *u_task=None*)

wrapper for functions that require an fixed input, which is here provided by a previous task

AGLOW.airflow.utils.AGLOW_utils.**launch_processing_subdag**(*prev_task*, ***context*)

AGLOW.airflow.utils.AGLOW_utils.**make_srmfile_from_step_results**(*prev_step_token_task*, *parent_dag=None*)

Makes a list of srms using the results of all the tokens in a previous task

AGLOW.airflow.utils.AGLOW_utils.**modify_parset**(*parset_path*, *freq_res*, *time_res*, *OBSID*, *flags*, *demix_sources*)

Takes in a base_parset path and changes the time and frequency resolution parameters of this parset. Saves it into a tempfile. Returns the tempfile_path

AGLOW.airflow.utils.AGLOW_utils.**modify_parset_from_fields_task**(*parsets_dict={}*, *fields_task=None*, *time_avg=8*, *freq_avg=2*, *flags=None*, ***context*)

Takes a dict of 'original' parsets and the task with the fields information which will be used to update the averaging paremetes. Returns a dict of the 'name':'location' of the modified parsets, so they can be used by the upload tokens task

AGLOW.airflow.utils.AGLOW_utils.**set_field_status**(*fields_file*, *cal_OBSID*, *targ_OBSID*, *field_name*, *status*)

AGLOW.airflow.utils.AGLOW_utils.**set_field_status_from_task_return**(*fields_file*, *task_id*, *status_task*, ***context*)

Sets the field status based on the (String) reuturned by the status_task variable

AGLOW.airflow.utils.AGLOW_utils.**set_field_status_from_taskid**(*fields_file*, *task_id*, *status*, ***context*)

sets the field status as the status input variable

AGLOW.airflow.utils.AGLOW_utils.**set_user_proxy_var**(*proxy_location*)

AGLOW.airflow.utils.AGLOW_utils.**stage_if_needed**(*stage_task*, *run_if_staged*, *run_if_not_staged*, ***context*)

This function takes the check_if_staged task and stages the files if None is returned. Otherwise it passes the srmlist to the 'join' task and the processing continues

CHAPTER 3

AGLOW sensors

Derived from the airflow sensor class, the AGLOW sensors can track long-running events using GRID and LOFAR tools

3.1 gliteSensor

The purpose of the gliteSensor is to monitor a job submitted with glite-wms-job-submit. Given a URL pointing to the job, this sensor polls the jobs status at a regular interval using the airflow sensor's poke feature. It parses the output and exits when all of the jobs have exited (whether completed successfully or not)

class AGLOW.airflow.sensors.glite_wms_sensor.**gliteSensor**(***kwargs*)

Bases: airflow.operators.sensors.BaseSensorOperator

An sensor initialized with the glite-wms job ID. It tracks the status of the job and returns only when all the jobs have exited (finished OK or not)

Parameters

- **submit_task** (*string*) – The task which submitted the jobs (should return a glite-wms job ID)
- **success_threshold** – Currently a dummy

__init__(***kwargs*)

x.__init__(...) initializes x; see help(type(x)) for signature

add_only_new(*item_set*, *item*)

Adds only new items to item set

clear(***kwargs*)

Clears the state of task instances associated with the task, following the parameters specified.

count_successes(*jobs*)

Counts the number of Completed jobs in the results of the glite-wms-job-status output. Returns all the job statuses and sets self.job_status if it's Done

Parameters **jobs** (*str*) – A string containing the full output of glite-wms-job-status

dag
Returns the Operator's DAG if set, otherwise raises an error

dag_id
Returns dag id if it has one or an adhoc + owner

deps
Adds one additional dependency for all sensor operators that checks if a sensor task instance can be rescheduled.

downstream_list
@property: list of tasks directly downstream

downstream_task_ids
@property: list of ids of tasks directly downstream

dry_run()
Performs dry run for the operator - just render template fields.

execute(*context*)
This is the main method to derive when creating an operator. Context is the same dictionary used as when rendering jinja templates.

Refer to get_template_context for more context.

extra_links
@property: extra links for the task.

get_direct_relative_ids(*upstream=False*)
Get the direct relative ids to the current task, upstream or downstream.

get_direct_relatives(*upstream=False*)
Get the direct relatives to the current task, upstream or downstream.

get_extra_links(*dttm*, *link_name*)
For an operator, gets the URL that the external links specified in *extra_links* should point to.

Raises **ValueError** – The error message of a ValueError will be passed on through to the fronted to show up as a tooltip on the disabled link

Parameters

- **dttm** – The datetime parsed execution date for the URL being searched for
- **link_name** – The name of the link we're looking for the URL for. Should be one of the options specified in *extra_links*

Returns A URL

get_flat_relative_ids(*upstream=False*, *found_descendants=None*)
Get a flat list of relatives' ids, either upstream or downstream.

get_flat_relatives(*upstream=False*)
Get a flat list of relatives, either upstream or downstream.

classmethod get_serialized_fields()
Stringified DAGs and operators contain exactly these fields.

get_task_instances(***kwargs*)
Get a set of task instance related to this task for a specific date range.

get_template_env()
Fetch a Jinja template environment from the DAG or instantiate empty environment if no DAG.

global_operator_extra_link_dict
Returns dictionary of all global extra links

has_dag()
Returns True if the Operator has been assigned to a DAG.

log

logger

next()

on_kill()
Override this method to cleanup subprocesses when a task instance gets killed. Any use of the threading, subprocess or multiprocessing module within an operator needs to be cleaned up or it will leave ghost processes behind.

operator_extra_link_dict
Returns dictionary of all extra links for the operator

operator_extra_links = ()

parse_glite_jobs(*jobs*)

poke(*context*)
Function called every (by default 2) minutes. It calls glite-wms-job-status on the jobID and exits if all the jobs have finished/crashed.

pool = ''

post_execute(*context*, **args*, ***kwargs*)
This hook is triggered right after self.execute() is called. It is passed the execution context and any results returned by the operator.

pre_execute(*context*, **args*, ***kwargs*)
This hook is triggered right before self.execute() is called.

prepare_template()
Hook that is triggered after the templated fields get replaced by their content. If you need your operator to alter the content of the file before the template is rendered, it should override this method to do so.

priority_weight_total
Total priority weight for the task. It might include all upstream or downstream tasks. depending on the weight rule.

- WeightRule.ABSOLUTE - only own weight
- WeightRule.DOWNSTREAM - adds priority weight of all downstream tasks
- WeightRule.UPSTREAM - adds priority weight of all upstream tasks

render_template(*content*, *context*, *jinja_env=None*, *seen_oids=None*)
Render a templated string. The content can be a collection holding multiple templated strings and will be templated recursively.

Parameters

- **content** (*Any*) – Content to template. Only strings can be templated (may be inside collection).
- **context** (*dict*) – Dict with values to apply on templated content
- **jinja_env** (*jinja2.Environment*) – Jinja environment. Can be provided to avoid re-creating Jinja environments during recursion.

- **seen_oids** (*set*) – template fields already rendered (to avoid RecursionError on circular dependencies)

Returns Templated content

render_template_fields(*context*, *jinja_env=None*)
Template all attributes listed in template_fields. Note this operation is irreversible.

Parameters

- **context** (*dict*) – Dict with values to apply on content
- **jinja_env** (*jinja2.Environment*) – Jinja environment

reschedule

resolve_template_files()

run(*start_date=None*, *end_date=None*, *ignore_first_depends_on_past=False*, *ignore_ti_state=False*, *mark_success=False*)
Run a set of task instances for a date range.

schedule_interval
The schedule interval of the DAG always wins over individual tasks so that tasks within a DAG always line up. The task still needs a schedule_interval as it may not be attached to a DAG.

set_downstream(*task_or_task_list*)
Set a task or a task list to be directly downstream from the current task.

set_upstream(*task_or_task_list*)
Set a task or a task list to be directly upstream from the current task.

shallow_copy_attrs = ()

skip(***kwargs*)
Sets tasks instances to skipped from the same dag run.

Parameters

- **dag_run** – the DagRun for which to set the tasks to skipped
- **execution_date** – execution_date
- **tasks** – tasks to skip (not task_ids)
- **session** – db session to use

skip_all_except(*ti*, *branch_task_ids*)
This method implements the logic for a branching operator; given a single task ID or list of task IDs to follow, this skips all other tasks immediately downstream of this operator.

task_type
@property: type of the task

template_ext = ()

template_fields = ()

ui_color = '#7c7287'

ui_fgcolor = '#000'

upstream_list
@property: list of tasks directly upstream

upstream_task_ids
@property: list of ids of tasks directly upstream

valid_modes = ['poke', 'reschedule']

xcom_pull(*context*, *task_ids=None*, *dag_id=None*, *key='return_value'*, *include_prior_dates=None*)
See TaskInstance.xcom_pull()

xcom_push(*context*, *key*, *value*, *execution_date=None*)
See TaskInstance.xcom_push()

CHAPTER 4

AGLOW Operators

Derived from the airflow base operator class, the AGLOW operators integrate with staging services and the GRID_LRT package. These sensors are used to define, package, build and launch LOFAR jobs.

4.1 LRT_Sandbox

This operator builds and uploads a sandbox, given a sandbox definition file. As an input, it takes a configuration file and returns the sandbox location in a dictionary with a key “SBX_location”.

CHAPTER 5

AGLOW Subdags

To make it easier to build complex workflows; subdags can be incorporated in the parent workflow.

5.1 SKSP_Calibrator

The calibrator workflow for the SKSP data processing pipeline.

AGLOW.airflow.subdags.SKSP_calibrator.**archive_all_tokens**(*token_type*, *archive_location*, *delete=False*)

AGLOW.airflow.subdags.SKSP_calibrator.**archive_tokens_from_task**(*token_task*, *delete=False*, ***context*)

Determines whic tokens to archive and saves them. delete if necessary

AGLOW.airflow.subdags.SKSP_calibrator.**calibrator_subdag**(*parent_dag_name*, *subdagname*, *dag_args*, *args_dict=None*)

AGLOW.airflow.subdags.SKSP_calibrator.**test_check_staged**(*srm_variable*, ***context*)

CHAPTER 6

AGLOW dags

To make it easier to build complex workflows; subdags can be incorporated in the parent workflow.

6.1 SKSP_Launcher

This Launcher DAG queries a remote database for the next SKSP Field and launches processing either at SURFsara or at Juelich.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

C

D

E

G

H

L

M

N

O

P

R

S

T

U

V

X