iDDS

Release 0.0.4

Wen Guan (Wisconsin, IRIS-HEP, ATLAS@CERN)

Oct 06, 2023

CONTENTS

1 General Information 3

2 Use Cases 13

3 User Documentation 25

4 DOMA User Documentation 49

5 Source Codes 53

6 Indices and tables 67

iDDS(intelligent Data Delivery Service) is an intelligent Data Delivery Service. It's designed to intelligently transform and deliver the needed data to the processing workflow in a fine-grained approach for High Energy Physics workloads.

iDDS is developed with modular and is highly scalable with a serializable Work/Workflow structure.

This documentation is generated by the Sphinx toolkit. and lives in the source tree.

CHAPTER

ONE

GENERAL INFORMATION

This section contains the general information related to iDDS which is common to all developers, users, and operators. For documentation specific to any of these three, please see the subsequent sections.

1.1 Functions and Example Workflows

iDDS(intelligent Data Delivery Service) is an intelligent Data Delivery Service. It's designed to intelligently transform and deliver the needed data to the processing workflow in a fine-grained approach for High Energy Physics workloads. It will not only reduce the need for replicas, but also enable benefits such as decreasing the time period of caching transient data, transforming expensive replicas to cheaper format data at remote sites and only cache cheaper new data for processing. iDDS will also work to orchestrate the WorkFlow Management System (WFMS) and the Distributed Data Management (DDM) systems to trigger them to process the data as soon as possible. In addition, iDDS will have intelligent algorithms to adjust the lifetime of cache, the format transformation and the delivery destination. The iDDS will increase the efficiency of data usage, reduce storage usage for processing and speed up the processing workflow.

1.1.1 Functions

iDDS is proposed to intelligently transform and deliver the needed data to a processing workflow in a high granularity. Here are(or will be) the main functions of iDDS:

1. Transformation on demand: Transform expensive data on demand to the format needed for processing on a remote site and only deliver the needed data to the following processing steps. At first, transformation on demand will avoid producing unused data. Secondly the storage-side transformation will minimize the network load. Thirdly, instead of delivering expensive complete replicas, only cheaper transformed data will be delivered and cached, which will reduce local replicas or cache usage. Last but not least, we can apply data locality knowledge and intelligence in the caching process to promote the cache reuse.

2. Fine-grained delivery: Coordinate with the following processing steps to process data and to remove data in a fine-grained way, without waiting for all data to be cached. It will reduce the replica usage or cache usage and speed up the processing workflow.

3. Orchestration: Orchestration between WFMS and DDM for optimal usage of limited resources for workflows that intersects the boundaries of data management and workflow management.

4. Intelligent: To develop intelligent algorithms as a brain to improve the scheduling in iDDS, which will apply data locality knowledge and processing requests to trigger on-demand transformations, fine-grained delivery and cache management to optimize the processing workflow and promote the cache reuse.

1.1.2 Example Workflows

Several workflows are proposed. Here are some examples:

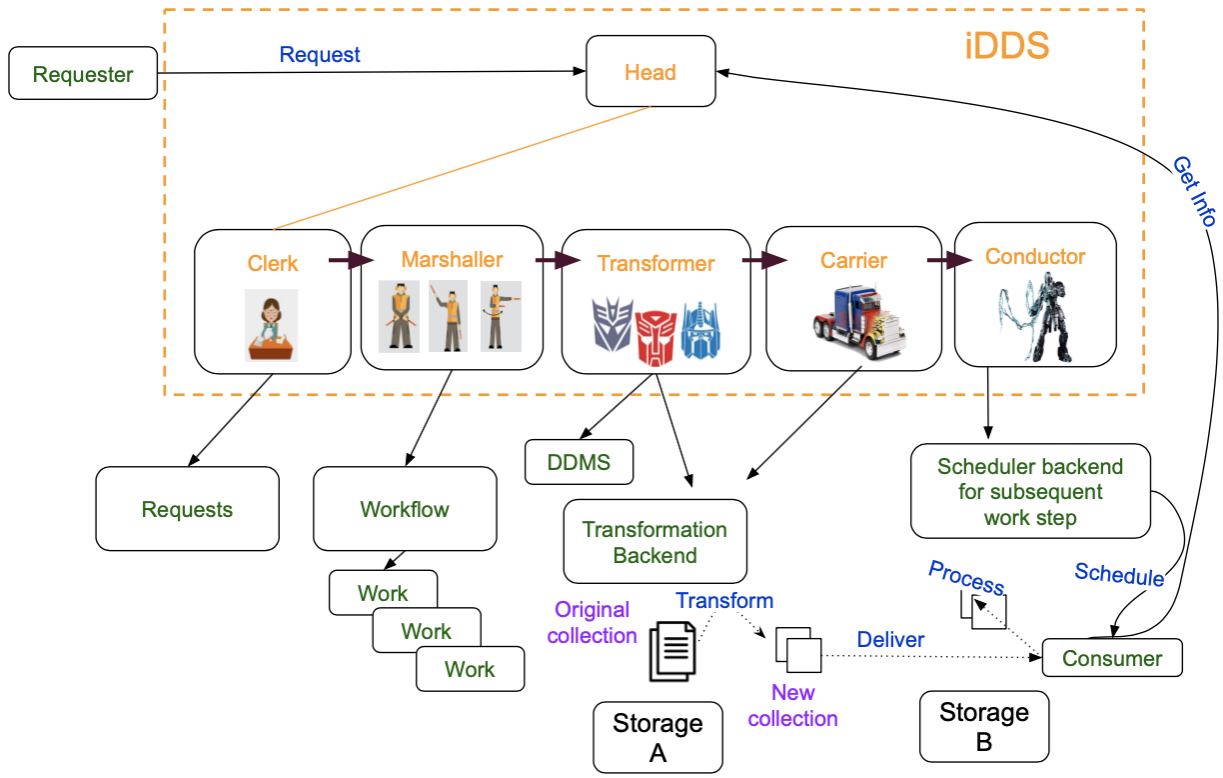
1. For data carousel, instead of waiting to release jobs until all files of a dataset have been staged-in, we can process the file that is already staged-in and remove it after it's processe . In this fine-grained way, we can speed up the file processing and reduce the stage-in pool usage.

2. For some analysis format data, such as DAOD, in the current computing model they are centrally produced and stored for a long time. Some of the produced data may never be used and some of them are used just a few times in a short period and are never touched after that. It occupies a lot of storage space. If we can produce these analysis format data on demand with adjustable lifetime based on the data usage frequency, we may be able to reduce the storage used by these data.

3. For the HL-LHC, we will produce a lot more data and it will be difficult for users to download all data to local storage. However, some analysis methods such as machine learning require to load all events and then evaluate them in many iterations. As a result, a preparation step to skim/slim data and only download required event information to local storage is needed. If we can standardize this step and make it reusable, we will not only make the life of physicists easier, but may also reduce the CPU time used today to skim/slim the data many times for every analysis.

4. Some existing services developed with accumulated requirements are housed in inappropriate components, for example, dynamic data placement service is mixed in the job management system PanDA and the data management system Rucio, which increases the difficulty to maintain them and to optimize the workflow.

1.2 Schematic View

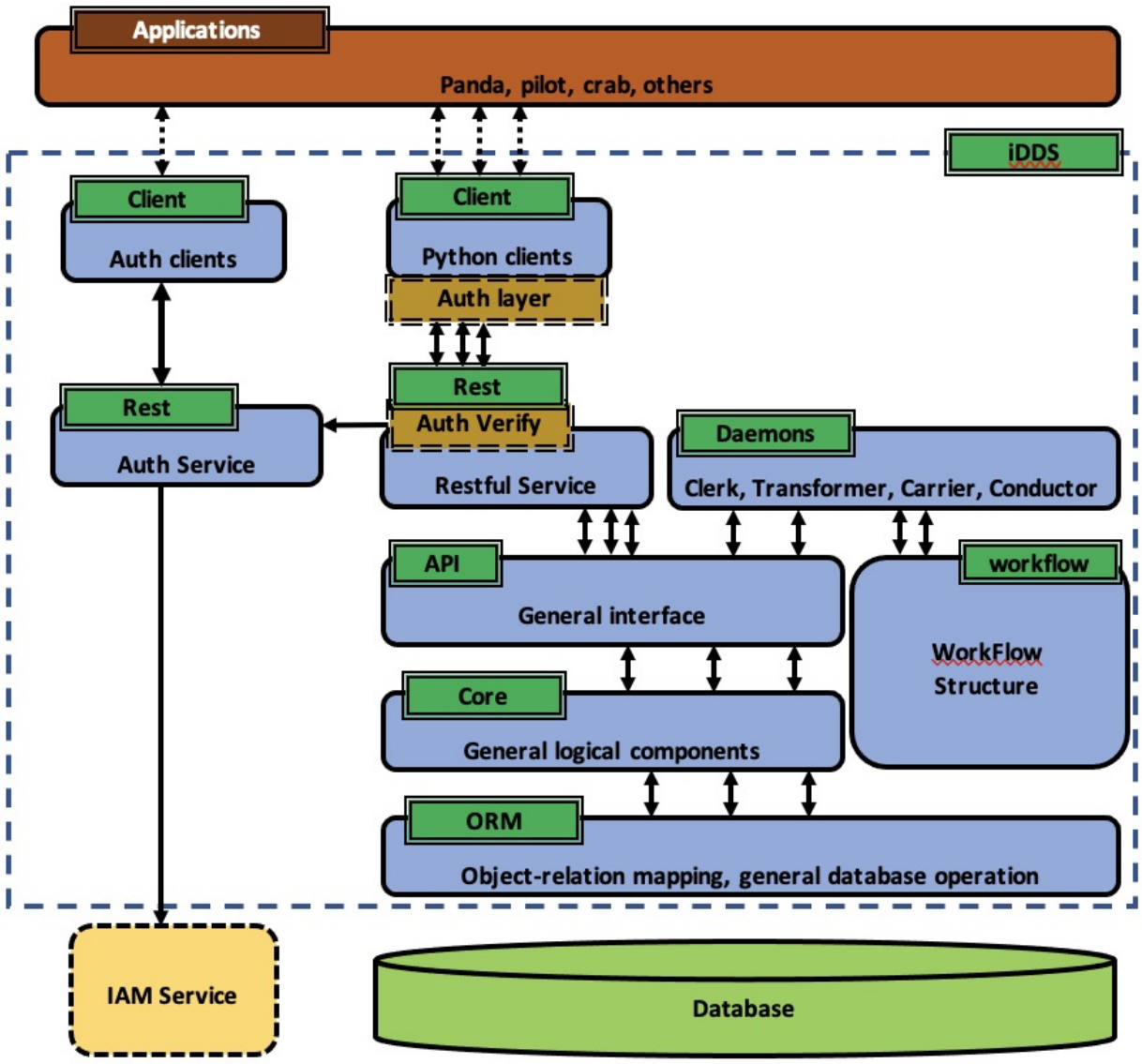


The iDDS is designed as a standalone experiment agnostic service. It consists of a general Restful service to receive requests from WFMS and several running agents in a daemon mode to process the requests.

In this model, the Restful service is used to register and query requests. It also provides a catalog service for users to retrieve required collections or contents. In the daemon mode, an agent Clerk works to manage requests and covert requests to workflow. The agent Marshaller works to manage workflow and split workflow to works, where one work is one transformation. The workflow management includes DAG management. The transformer works to query input replicas from DDMs, maps inputs to different transform outputs and create processings to process the transformation. The Carrier agent works to submit the processing and poll the processing. When the output data is available, the agent Conductor works in a fine-grained approach to notify consumers to process the new transformed data.

1.3 Architecture

The iDDS is implemented in a distributed architecture. It composed of Daemons Agents, RESTful serivces, User Interface and External Plugins.



1.3.1 Layers

The iDDS is designed with abstract layers to hide the complexity of different logics and every layer concentrates on one type of operations. It simplifies the logic of every layer and smooths the development and maintenance. The iDDS layers are composed of:

- ORM (Oject-Relational Mapping)
- Core layer
- API
- Restful services

1.3.2 Workflow

The iDDS workflow architecture is another experiment agnostic design to support new emerging workflows. In iDDS, a workflow consists of multiple works, where a work is a transformation. The iDDS processing is designed based on the workflow(work) structure. For a new emerging use case, an inherited Work class can be developed with implemented hook functions. Then the new use case can be imported to iDDS.

1.3.3 Daemons

The iDDS daemons are active agents that orchestrate the collaborative work of the whole system. They are for example:

- Request Daemon (Clerk) - in charge of handling requests
- Workflow Daemon (Marshaller) – in charge of handling workflows.
- Transform daemon (Transformer) - in charge of handling transforms
- Processing daemon (Carrier) - in charge of handling processings which do the real data transformation.
- Message daemon (Conductor) - in charge of delivering messages to ActiveMQ, which will be consumed by following workflow.

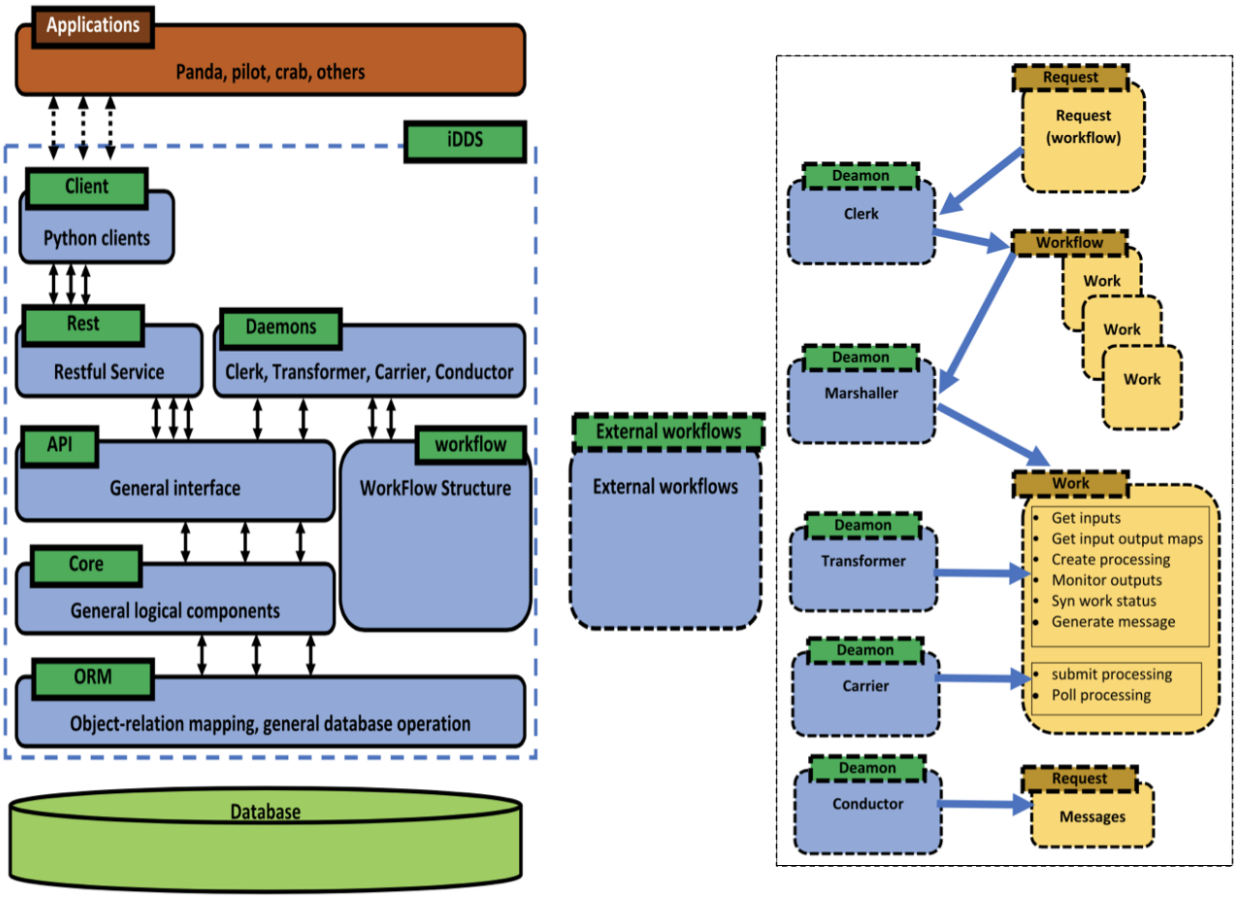
1.3.4 Client

The client is the user interface for users to communicate the RESTful service.

- Request client
- Catalog client
- HPO(HyperParameterOptimization) client

1.4 Workflow

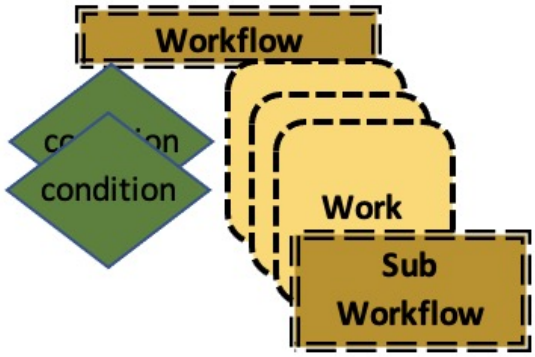
The iDDS has implemented a workflow architecture to support new use cases. The Workflow consists of Work(Transformation) and Condition, where Condition can be used to implement DAG support. New use cases can be implemented with inherited Work class with developed hook functions.

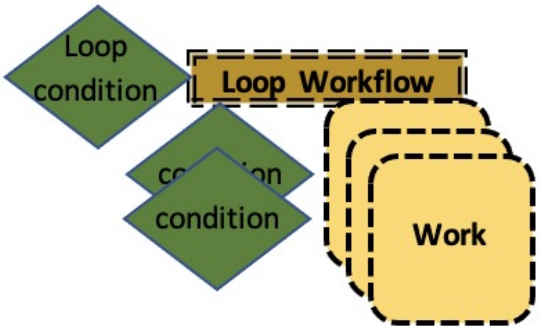


1.4.1 DictClass

DictClass is designed to automatically convert all attributes of the class to a json dictionary and convert it back from the json dictionary to the class. The Workflow and Work classes are inherited from this class. In this way, the Workflow and Work can be transmitted from client to Rest service and be saved to database, in a json format. When reading, this json format can be converted back to Workflow or Work instance, which simplies the handings in the following step.

1.4.2 Workflow





A Workflow is designed to manage multiple Works(Transformations). It's a sub-class of DictClass. With Condition supports, the Workflow can support DAG management.

1.4.3 SubWorkflow

A workflow can be added as a subworkflow of another workflow. In this case, it can be regarded as a Work. However, this work will generate new Works.

1.4.4 LoopWorkflow

When adding a loop condition to a workflow, the workflow can be looped.

1.4.5 Work

A Work, a sub-class of DictClass, is a transformation. New use cases can be implemented by inherited Work class.

1. Functions need to be overwritten for Transformer:
    a. get_input_collections: poll DDM to get the status and metadata of the collections.
    b. get_new_input_output_maps(registered_input_output_maps): registered_input_output_maps is provided by iDDS with contents registered in iDDS db. This function should return maps between new inputs to outputs.
    c. create_processing(input_output_maps): Creating a processing with maps between inputs and outputs.
    d. syn_work_status(registered_input_output_maps): registered_input_output_maps is provided by iDDS with contents registered in iDDS db. It works to update the Work.status to be Transforming, Finished, SubFinished or Failed based on all outputs' status in registered_input_output_maps.
2. Functions need to be overwritten for Carrier:
    a. submit_processing: Implement functions how to submit the processing.
    b. poll_processing_updates: Implement functions how to poll the processing status.

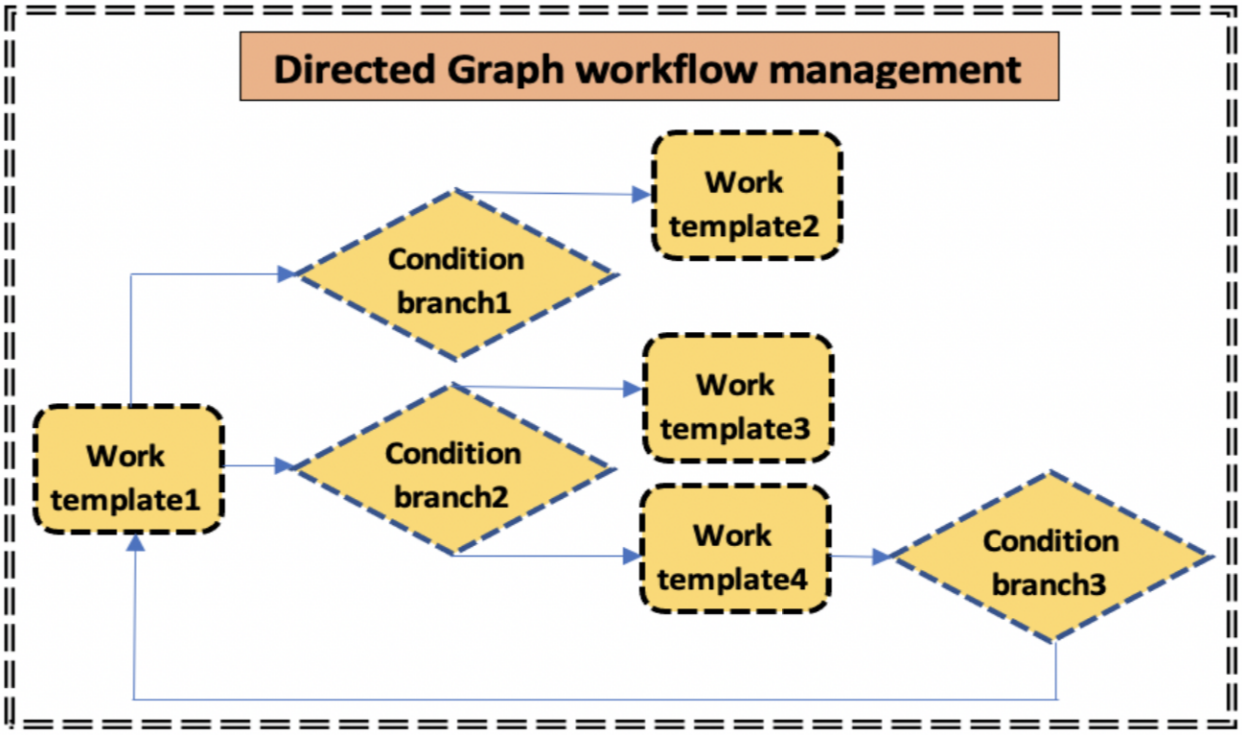
1.4.6 WorkflowManager

It works to automatically convert a workflow to an iDDS request(The workflow will be converted to a json dictionary by DictClass) and send the request to iDDS Restful service.

1.5 Directed Graph(DG) and Directed Acyclic Graph(DAG)

The DG (Directed Graph) workflow management in iDDS not only supports DAG (Directed Acyclic Graph), but also supports graphs with cycles. iDDS supports two different types of DAG.

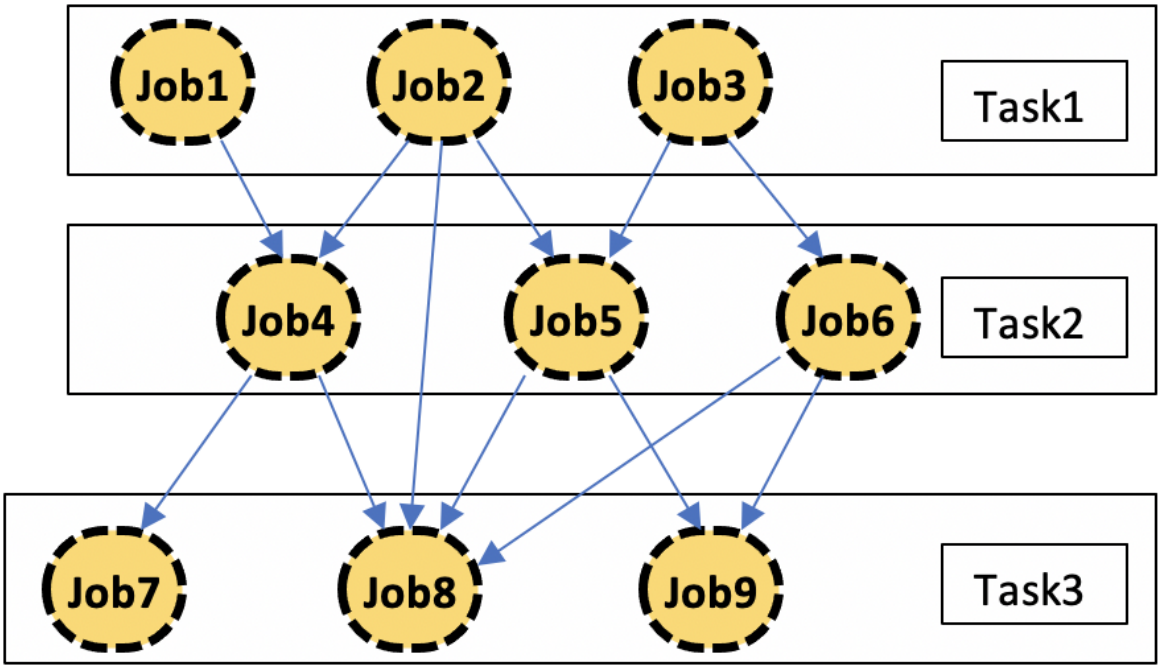
1.5.1 Task Level DAG



A Task level DAG is represented as a Workflow object which is composed of multiple Work template objects and their relationship with condition branches, as shown below. A Work template is a placeholder to generate new Work objects by assigning values for pre-defined parameters. The condition will be evaluated when a work's status has changed, the following work will be triggerred based on the conditions.

1. When a work generates some new outputs, trigger to generate new jobs for the following work.
2. When a work is terminated, trigger to create new work or terminate the workflow based on condition evaluation.

1.5.2 Job Level DAG



For Job level DAG, it's different that at the beginning all jobs and their relations are defined. Based on their relations, the jobs can be grouped to tasks to adapt workflow management system (For example, PanDA job management is based on tasks). iDDS Job DAG manages the dependencies and triggers to release jobs when all dependencies are ready.

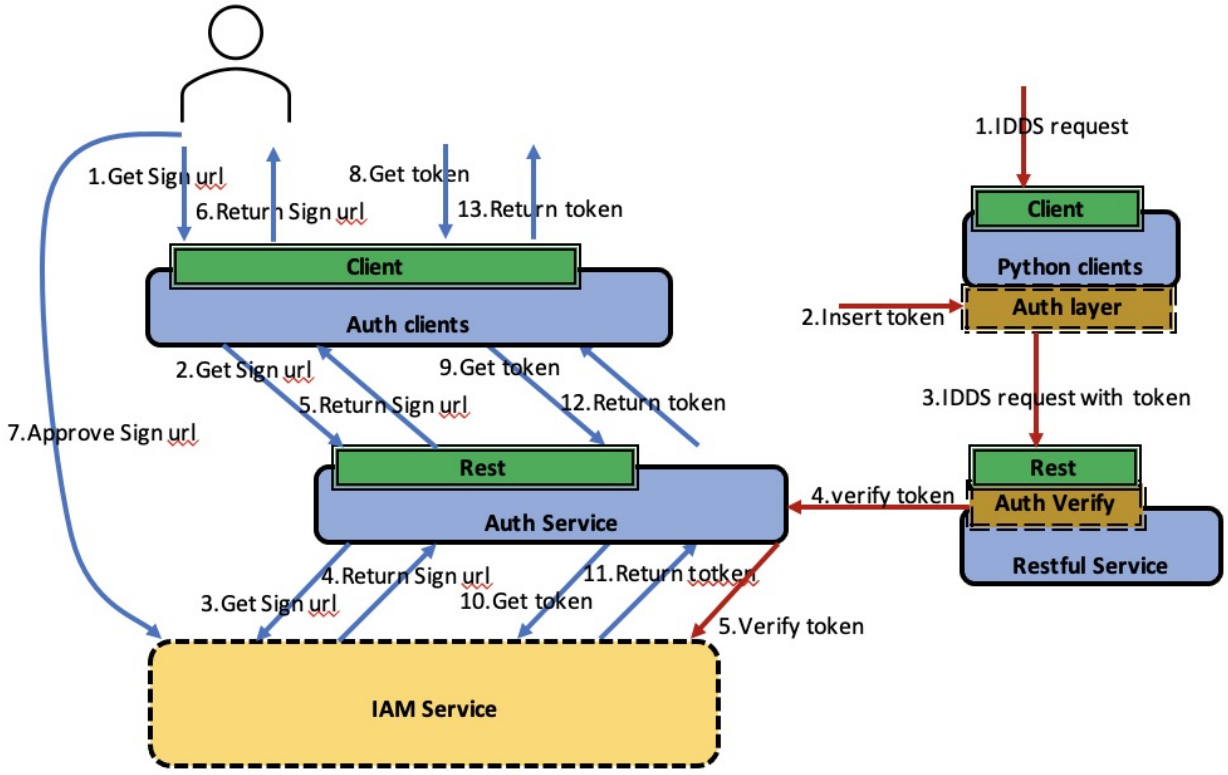
1.6 Authorization

The iDDS currently supports both x509_proxy and oidc based authroization.

1.6.1 509_proxy

x509_proxy based authorization is the default authorization method for iDDS. It's implemented mainly based mod_ssl and mod_gridsite.

1.6.2 oidc



The iDDS OIDC authorization is based on the IAM service. Here are the steps for token initialization:

1. Get sign url: Get a sign url with a device code for users to approve.
2. User goes to the IAM service and approves the token request with the sign url.
3. Get the token with the device code.
4. The iDDS OIDC authorization service also includes services such as token refresh, token clean, token information checks and so on.

For normal iDDS requests, here are steps how iDDS authorize a users.

1. User initializes a normal request.
2. iDDS automatically finds the token and loads the token to headers of the http request.
3. Send the request to iDDS REST server.
4. iDDS server parse the token and verify the token against the IAM server. Verified users will be authorized.

CHAPTER

TWO

USE CASES

Here it includes use cases that iDDS supports or will supports.

2.1 Data Carousel

Data Carousel is a use-case of iDDS. The purpose of Data Carousel with iDDS is to release tasks quickly enough, avoid redundant job attempts, and improve error handling.

2.1.1 Workflow

1. Prodsys2 submits tasks with *inputPreStaging* and *toStaging* parameters to JEDI.
2. Tasks go to *staging* state in JEDI.
3. Prodsys2 creates rules in Rucio to stage-in data from tape to disk by taking into account global share, the number of requests in each FTS channel, and so on, and immediately sends notifications to JEDI before rules are completed.
4. For each task, JEDI finds input dataset scope/name and the corresponding rule and sends a request to iDDS.
5. iDDS creates a transform object for each request to monitor the rule. Note that in this use-case, tape and disk file replicas are regarded as input and output collections, respectively, and thus data are not really transformed.
6. iDDS periodically checks the rule and notifies JEDI via ActiveMQ when files become available on disk (or disk buffer of the tape system).
7. JEDI generates jobs using only input files which have disk replicas, assigns the jobs, and submits them to PanDA.
8. PanDA creates rules in Rucio to transfer input files if jobs are assigned to satellites where input data are unavailable.
9. Jobs get started when input files are or become available.

2.1.2 Advantages

Without iDDS, JEDI is not aware of which files are on disk while input data are being staged-in, and generates jobs using input files even if some of them have not had disk replicas yet. There are some issues.

- Jobs create other redundant rules in Rucio and requests in FTS to stage-in input files from tape to disk if those files are available only on tape.
- Those jobs tend to silently sit in assigned state and occupy queues until they eventually time-out. They may prevent new jobs from being assigned to the queues, leading to unbalanced job distribution.

A naive solution is to not generate jobs until 90% of the input files become available on disk, i.e., not to release tasks very quickly. iDDS has solved those issues by letting JEDI timely use only the files with disk replicas.

2.1.3 Future improvements to shorten the tails on completing tasks

iDDS gives JEDI knowledge of problematic files, which means that iDDS and/or JEDI can take actions if necessary. For example, when files are stuck for long time, iDDS would make new rules to transfer them from other sites. Possible actions and conditions to trigger them to be defined.

2.1.4 iDDS ATLAS data carousel status monitor

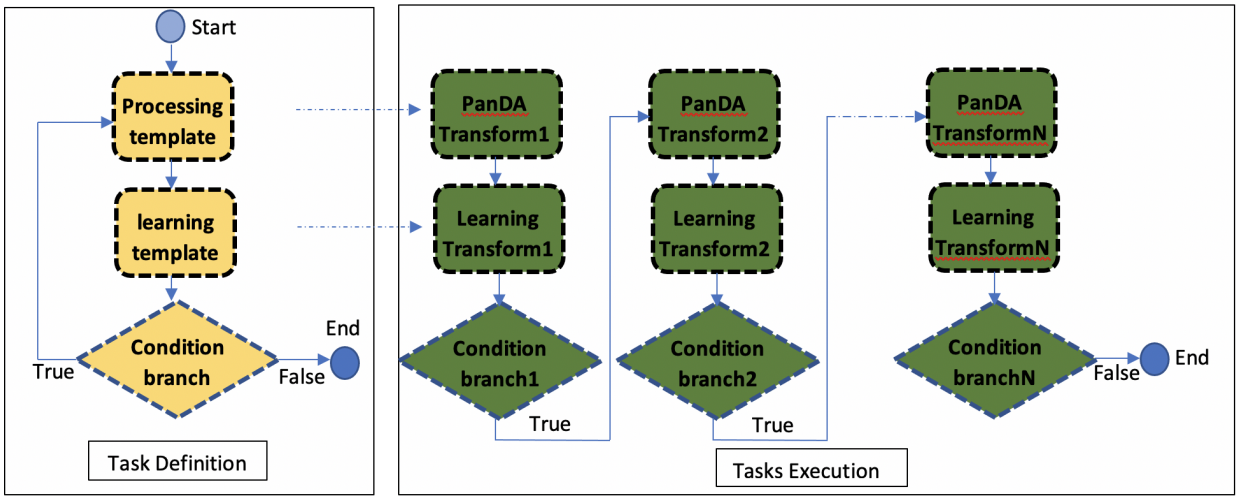
Monitor: https://bigpanda.cern.ch/idds/

1. transform_status:
    a. Finished: All files are processed.
    b. Transforming: Some files are still under processing.
2. in_status: Input dataset status. If the input dataset is not closed, it means it's still possible that some other system will add more files to the input dataset. So iDDS will monitor whehter there are new files added to this input dataset.
3. in_total_files: Total number of files in the input dataset.
4. in_processed_files: Total number of files handled by iDDS(Files that are used as input in an iDDS transformation).
5. out_status: The status of the output dataset. It will be closed(the transform will be finished) when:
    a. All input files are processed and the input dataset is closed.
    b. All output files are processed.
6. out_total_files: Total number of files in the output dataset.
7. output_processed_files: Total number of processed files in the output dataset.

2.2 Active Learning(AL)

Active Learning is an usecase of iDDS. The purpose of iDDS AL is to use iDDS to run some 'active learning' process to tell production system whether to continue some process.

2.2.1 iDDS AL workflow



ActiveLearning employs iDDS DAG workflow management to define tasks.

1. It uses processing template and learning template to define the processing workflow.
2. It uses a Condition branch to control the workflow.
3. When executing, the processing template will generate a PanDA task.
4. When the PanDA task finishes, the learning template will generate a learning task which will run in iDDS internally condor cluster, to analyse the outputs of the PanDA task. The result of the learning task will decide whether to generate new PanDA tasks or to terminate.

processing task (ATLASPandaWork)

1. upload inputs to Panda cache server and define the task parameter map.

```
import json
import re
import time
# import traceback

try:
    from urllib import quote
except ImportError:
    from urllib.parse import quote

from pandatools import Client

from idds.client.clientmanager import ClientManager
```

(continues on next page)

(continued from previous page)

```
from idds.common.utils import get_rest_host, run_command

from idds.workflow.workflow import Condition, Workflow
from idds.atlas.workflow.atlaspandawork import ATLASPandaWork
from idds.atlas.workflow.atlasactuatorwork import ATLASActuatorWork


# Here a fake method is used.
def get_task_id(output, error):
    m = re.search('jediTaskID=(\d+)', output + error)  # noqa W605
    task_id = int(m.group(1))
    return task_id


def submit_processing_task():
    outDS = "user.wguan.altest%s" % str(int(time.time()))
    cmd = "cd /afs/cern.ch/user/w/wguan/workdisk/iDDS/main/lib/idds/tests/activelearning_
↪test_codes; prun --exec 'python simplescript.py 0.5 0.5 200 output.json' --outDS %s  --
↪outputs output.json --nJobs=10" % outDS
    status, output, error = run_command(cmd)
    """
    status:
    0
    output:

    error:
    INFO : gathering files under /afs/cern.ch/user/w/wguan/workdisk/iDDS/main/lib/idds/
↪tests/activelearning_test_codes
    INFO : upload source files
    INFO : submit user.wguan.altest1234/
    INFO : succeeded. new jediTaskID=23752996
    """
    if status == 0:
        task_id = get_task_id(output, error)
        return task_id
    else:
        raise Exception(output + error)


def get_panda_task_paramsmap(panda_task_id):
    status, task_param_map = Client.getTaskParamsMap(panda_task_id)
    if status == 0:
        task_param_map = json.loads(task_param_map)
        return task_param_map
    return None


def define_panda_task_paramsmap():
    # here is using a fake method by submitting a panda task.
    # Users should define the task params map by themselves.

    # (0, '{"buildSpec": {"jobParameters": "-i ${IN} -o ${OUT} --sourceURL ${SURL} -r . ",
↪ "archiveName": "sources.0ca6a2fb-4ad0-42d0-979d-aa7c284f1ff7.tar.gz", "prodSourceLabel
↪": "panda"}, "sourceURL": "https://aipanda048.cern.ch:25443", "cliParams": "prun --
↪exec \\"python simplescript.py 0.5 0.5 200 output.json\\" --outDS user.wguan.
↪altest1234 --outputs output.json --nJobs=10", "site": null, "vo": "atlas",
↪"respectSplitRule": true, "osInfo": "Linux-3.10.0-1127.19.1.el7.x86_64-x86_64-with-
↪centos-7.9.2009-Core", "log": {"type": "template", "param_type": "log", "container":
↪"user.wguan.altest1234.log/", "value": "user.wguan.altest1234.log.$JEDITASKID.${SN}.
```

(continues on next page)

(continued from previous page)

```
    task_id = submit_processing_task()
    task_param_map = get_panda_task_paramsmap(task_id)
    cmd_to_arguments = {'arguments': 'python simplescript.py 0.5 0.5 200',
                        'parameters': 'python simplescript.py {m1} {m2} {nevents}'}

    # update the cliParams to have undefined parameters, these parameters {m1}, {m2},
↪{nevents} will be the outputs of learning script.
    task_param_map['cliParams'] = task_param_map['cliParams'].replace(cmd_to_arguments[
↪'arguments'], cmd_to_arguments['parameters'])
    jobParameters = task_param_map['jobParameters']
    for p in jobParameters:
        if 'value' in p:
            p['value'] = p['value'].replace(quote(cmd_to_arguments['arguments']),␣
↪quote(cmd_to_arguments['parameters']))
    return task_param_map
```

2. define the panda work.

```
task_param_map = define_panda_task_paramsmap()
panda_work = ATLASPandaWork(panda_task_paramsmap=task_param_map)

# it's needed to parse the panda task parameter information, for example output dataset␣
↪name, for the next task.
# if the information is not needed, you don't need to run it manually. iDDS will call it␣
↪interally to parse the information.
panda_work.initialize_work()
```

learning task (ATLASActuatorWork)

1. **define the learning task.**

   (a) The input collection of the learning task is the output of the panda task. iDDS will download all files of this dataset to local storage and process them.

   (b) The sandbox is using the panda task's sandbox. You can also use iDDS cache server for it.

```
work_output_coll = panda_work.get_output_collections()[0]

input_coll = {'scope': work_output_coll['scope'],
              'name': work_output_coll['name'],
              'coll_metadata': {'force_close': True}}  # by default the panda collection␣
↪is not closed. If it's not closed, iDDS will poll again and again without stop.
output_coll = {'scope': work_output_coll['scope'],
               'name': work_output_coll['name'] + "." + str(int(time.time()))}

# acutator = ATLASActuatorWork(executable='python', arguments='merge.py {output_json}
↪{events} {dataset}/{filename}',
acutator = ATLASActuatorWork(executable='python', arguments='merge.py {output_json}
↪{events} {dataset}',
                             parameters={'output_json': 'merge.json',
                                         'events': 200,
```

(continues on next page)

(continued from previous page)

```
                                        'dataset': '{scope}:{name}'.format(**input_
↪coll),
                                        'filename': 'output*.json'},
                                sandbox=panda_work.sandbox, primary_input_collection=input_
↪coll,
                                output_collections=output_coll, output_json='merge.json')
```

Define workflow

```
wf = Workflow()
# because the two tasks are in a loop. It's good to set which one to start.
wf.add_work(panda_work)
wf.add_work(acutator)
cond = Condition(panda_work.is_finished, current_work=panda_work, true_work=acutator,␣
↪false_work=None)
wf.add_condition(cond)
cond1 = Condition(acutator.generate_new_task, current_work=acutator, true_work=panda_
↪work, false_work=None)
wf.add_condition(cond1)

# because the two works are in a loop, they are not independent. This call is needed to␣
↪tell which one to start.
# otherwise idds will use the first one to start.
wf.add_initial_works(work)

# work.set_workflow(wf)
return wf
```

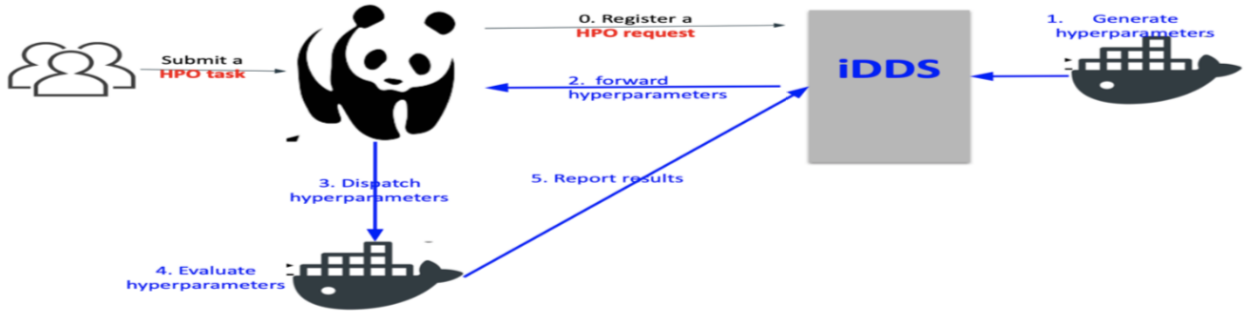
The AL example

See examples in https://github.com/wguanicedew/iDDS/blob/dev/main/lib/idds/tests/test_activelearning.py

2.3 HyperParameterOptimization(HPO)

HPO is a use-case of iDDS. The purpose of iDDS HPO is to use iDDS to generate hyperparameters for machine learning and trigger production system to automatically process the machine learning training with provided hyperparameters.

2.3.1 iDDS HPO workflow



0. The user prepares two container images. One for optimization and sampling (Steering container), and the other for training (Evaluation container). He/she can use pre-defined methods if they meet his/her requirements.
1. The user submit a HPO task to JEDI.
2. JEDI submits a HPO request to iDDS.
3. iDDS generates multiple sets of hyperparameters (hyperparameter points) in the search space using the Steering container or pre-defined methods, and stores them in iDDS database. Each hyperparameter point has a serial number and corresponds to a single machine learning training job.
4. Serial numbers are sent to JEDI through ActiveMQ.
5. JEDI generates PanDA jobs and dispatches them to pilots running on compute resources.
6. Each PanDA job fetches a serial number from PanDA and converts it to the corresponding hyperparameter point by checking with iDDS.
7. Once the hyperparameter point is evaluated using the Evaluation container, the loss is registered to iDDS.
8. The PanDA job fetches another serial number and evaluates the corresponding hyperparameter point if wall-time is still available.
9. When the number of unprocessed hyperparameter points goes below a threshold, iDDS automatically generates new hyperparameter points. Steps 3-9 will be iterated until:
    a. The total number of hyperparameter points reaches max_points. This parameter can be defined in the request.
    b. The Steering container / pre-defined method fails or returns [].
10. When all hyperparameter points are evaluated, iDDS sends a message to JEDI to terminate the HPO task.

Optimization and Sampling

Currently iDDS support several different ways to generate hyperparameter points.

> See examples in “User Documents” -> “iDDS RESTful client: Examples”

RESTful Service

1. To retrieve hyperparameters.

   See examples in "User Documents" -> "iDDS RESTful client: Examples" clientmanager.get_hyperparameters(workload_id, request_id, id=None, status=None, limit=None)

   examples: clientmanager.get_hyperparameters(workload_id=123, request_id=None) clientmanager.get_hyperparameters(workload_id=None, request_id=456) clientmanager.get_hyperparameters(workload_id=None, request_id=456, id=0)

2. To register loss of a group of hyperparameters.

   clientmanager.update_hyperparameter(request_id, id, loss)

3. Example code: main/lib/idds/tests/hyperparameteropt_client_test.py

User-defined Steering Container

Users can provide their own container images to generate hyperparameter points using the ask-and-tell pattern. Note that users need to use HPO packages such as skopt and nevergrad which support the ask-and-tell pattern when making Steering containers. Users can also provide execution strings to specify what are executed in containers. Each execution string needs to contain the following placeholders to get some parameters through command-line arguments.

%MAX_POINTS The max number of hyperparameter points to be evaluated in the entire search. iDDS stops generating new hyperparameter points when it receives an empty list []. The container needs to return [] if enough hyperparameter points are generated.

%NUM_POINTS The number of hyperparameter points to be generated in this iteration. The default is 10. It can be changed by setting 'num_points_per_generation' in the request_metadata. The Steering container runs (%MAX_POINTS / %NUM_POINTS) times in the entire search.

%IN The name of input file which iDDS places in the current directory every time it calls the container. The file contains a json-formatted list of all hyperparameter points, which have been generated so far, with corresponding loss (or None if it is not yet evaluated).

%OUT The name of output file which the container creates in the current directory. The file contains a json-formatted list of new hyperparameter points.

When iDDS runs Steering containers, iDDS will replace %XYZ with actual parameters. For example, if an execution string is something like `python opt.py --n=%NUM_POINT ...`, `python opt.py --n=10 ...` will be executed in the container. Input and output are done through json files in the current directly ($PWD) so that the directory needs to be mounted.

Here is one example for the input (main/lib/idds/tests/idds_input.json). It is a json dump of `{"points": [[{hyperparameter_point_1}, loss_or_None], ..., [{hyperparameter_point_N}, loss_or_None]], "opt_space": <opt space>}`. `{hyperparameter_point}` is a dictionary representing a single hyperparameter point. The keys of the dictionary can be arbitrary strings and correspond to the axes of the search space. For example, if there is a two dimensional search space with two hyperparameters, 'epochs' and 'batch_size', the dictionary could be something like `{'epochs': blah_1, 'batch_size': blah_2}`. `points` includes all hyperparameter points, which have been generated so far, whether or not they have been evaluated. If a hyperparameter point is not yet evaluated, the `loss_or_None` will be None. `opt_space` is a copy of the content from your request. If in your request `opt_space` is not defined, `opt_space` will be None.

The output is a json dump of `[{new_hyperparameter_point_1}, , ..., [{new_hyperparameter_point_N}]`. `{new_hyperparameter_point}` is a dictionary representing a new hyperparameter point. The format of the dictionary is the same as the one in the input.

Basically what the Steering container needs to do is as follows:

1. Define an optimizer with a search space.
2. Json-load `%IN` and update the optimizer with all hyperparameter points in `%IN` using the tell method.
3. Generate new hyperparameter points using the ask method, and json-dump them to `%OUT`. The number of new hyperparameter points is `min(%NUM_POINTS, %MAX_POINTS - NUM_POINTS_SO_FAR)` where `NUM_POINTS_SO_FAR` stands for the total number of hyperparameter points generated so far.

How to test the Steering container

Here is one example (`Steering_local_test https://github.com/HSF/iDDS/blob/master/main/lib/idds/tests/hyperparameteropt_dock
Users can update the request part and test their docker locally.

User-defined Evaluation Container

Users can provide their own container images to evaluate hyperparameter points and can provide execution strings to specify what are executed in their containers. The pilot and user-defined Evaluation container communicate with each other using the following files in the current directory ($PWD), so that the directory needs to be mounted. Their filenames can be defined in HPO task parameters. There are two files for input (one for a hyperparameter point to be evaluated and the other for training data) and three files for output (the first one to report the loss, the second one to report job metadata, and the last one to preserve training metrics). The input file for a hyperparameter point and the output file to report the loss are mandatory, while other files are optional.

Input for Evaluation Container

The pilot places two json files before running the Evaluation container. One file contains a json-formatted list of all filenames in the training dataset, i.e., it is a json-dump of `[training_data_filename_1, training_data_filename_2, ..., training_data_filename_N]`. If training data files need to be directly read from the storage the file contains a json-formatted list of full paths to training data files. The other file contains a single hyperparameter point to be evaluated. A hyperparameter point is represented as a dictionary and the format of the dictionary follows what the Steering container generated. For example, if the Steering container generates a hyperparameter point like `{'epochs': blah_1, 'batch_size': blah_2}`, the file will be a json-dump of `{'epochs': blah_1, 'batch_size': blah_2}`.

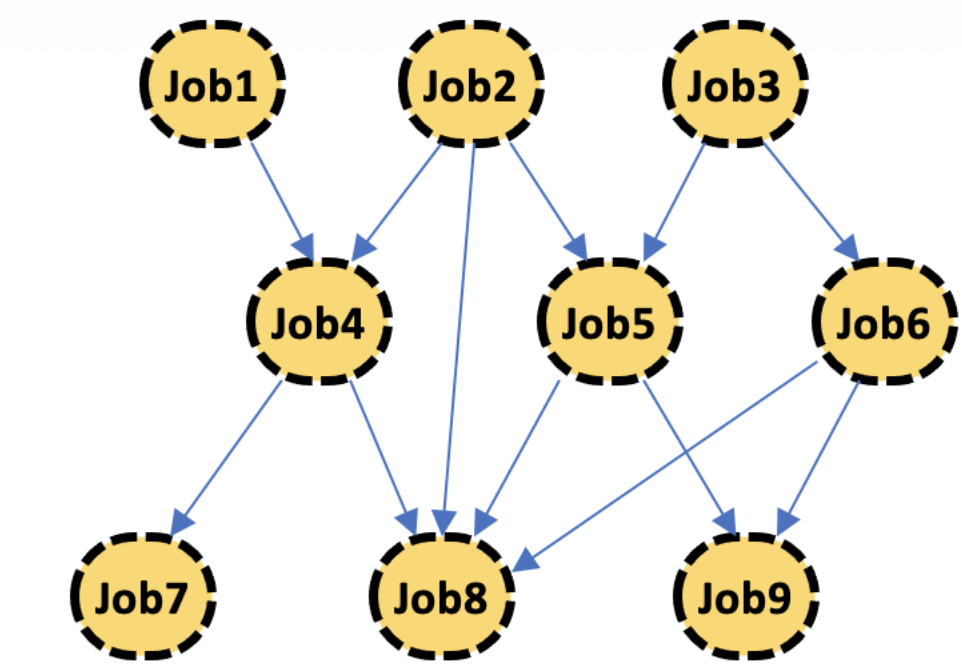
Output from Evaluation Container

The Evaluation container evaluates the hyperparameter point and produces one json file. The file contains a json-formatted dictionary with the following key-values: `status`: `integer` (0: OK, others: Not Good), `loss`: `float`, `message`: `string` (optional). It is possible to produce another json file to report job metadata to PanDA. It is a json-dump of an arbitrary dictionary, but the size must be less than 1MB. It is also possible to produce a tarball to preserve training metrics. The tarball is uploaded to the storage so that the size can be larger. The tarball can be used for post-processing such as visualization of the search results after been downloaded locally.

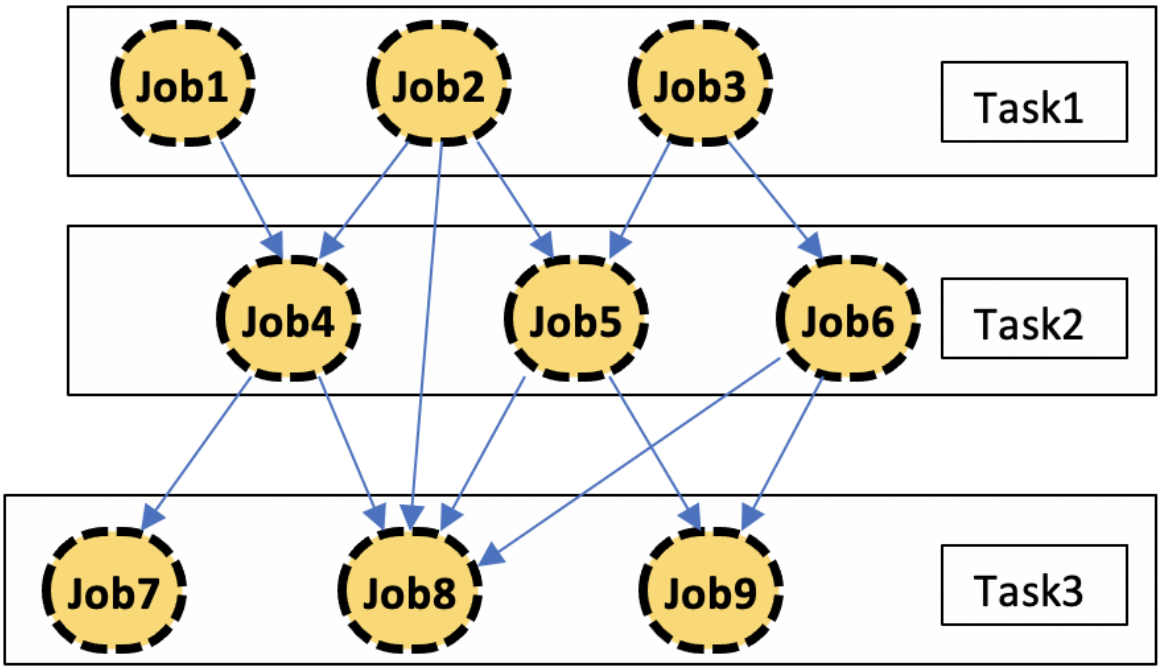
2.4 Doma Rubin (LSST)

The Rubin Observatory (LSST) exercise is an ongoing activity to evaluate PanDA as both a workflow and workload management system. A workflow graph is dynamically generated by Rubin middleware for each payload submission and includes, among others, a set of dependencies for each individual job that must be satisfied before the job could be processed. A single workflow can consist of a hundred thousand jobs forming the vertexes of a DAG. It is the first use case of the DG-based workflow support in iDDS. Every workflow is mapped to sequentially concatenated Work objects in iDDS. iDDS also allows Work objects to be incrementally released based on messaging, in order to avoid long waiting in each Work.

2.4.1 iDDS Rubin flow



Rubin at the beginning is DAG in jobs level. It's different iDDS Task level DAG. In iDDS Task level DAG, when a task generates new output files, it triggers to generate new jobs in the following tasks with these new inputs. iDDS Rubin is job level DAG. At the beginning, all jobs and their relations are defined.



To adapt the PanDA and iDDS task management, the client organizes jobs and splits jobs into different tasks based on their dependencies (https://github.com/lsst/ctrl_bps/blob/master/python/lsst/ctrl/bps/wms/panda/idds_tasks.py). The dependency map is transferred to iDDS, for every task. iDDS maps the dependency to inputs and outputs of jobs. The inputs of a job are its dependencies. So at the beginning, all jobs are defined. iDDS needs to release the jobs when all inputs for a job is ready. To efficiently release jobs, iDDS improves the inputs and outputs management. iDDS adds a trigger system to release jobs, instead of checking whether all inputs are ready for every job.

CHAPTER

THREE

USER DOCUMENTATION

3.1 Installing iDDS Clients

3.1.1 Prerequisites

iDDS clients run on Python 2.7, 3.6 on any Unix-like platform.

3.1.2 Python Dependencies

All Dependencies are automatically installed with pip.

3.1.3 Install via pip

When `pip` is available, the distribution can be downloaded from the iDDS PyPI server and installed in one step:

```
$> pip install idds-common idds-client
```

This command will download the latest version of Rucio and install it to your system.

3.1.4 Upgrade via pip

To upgrade via pip:

```
$> pip install --upgrade idds-common idds-client
```

3.1.5 ATLAS Users

To install via pip:

```
$> pip install --upgrade idds-common idds-client idds-workflow idds-atlas
```

3.1.6 DOMA Users

To install via pip:

```
$> pip install --upgrade idds-common idds-client idds-workflow idds-doma
```

3.1.7 config client

To use iDDS client to access the iDDS server, a config file is needed. Below is an example of the config file:

```
$> idds setup --auth_type oidc --host https://<hostname or ip>:443/idds --vo Rubin
```

3.2 iDDS RESTful client: Examples

iDDS provides RESTful services and the client is used to access the RESTful service.

3.2.1 iDDS OIDC authorization

1. Setup the client. It's for users to setup the client for the first time or update the client configurations. By default it will create a file in ~/.idds/idds_local.cfg to remember these configurations.

```
from idds.client.clientmanager import ClientManager
cm = ClientManager()
cm.setup_local_configuration(local_config_root=<local_config_root>,  # default ~/.idds/
                             host=<host>,                             # default host for␣
↪different authorization methods. https://<hostname or ip>:443/idds
                             auth_type=<auth_type>,                   # authorization␣
↪type: x509_proxy, oidc
                             auth_type_host=<auth_type_host>,         # for different␣
↪authorization methods, users can define different idds servers.
                             x509_proxy=<x509_proxy path>,
                             vo=<vo name>,
```

2. setup oidc token

```
from idds.client.clientmanager import ClientManager
cm = ClientManager()
cm.setup_oidc_token()
```

3. refresh oidc token

```
from idds.client.clientmanager import ClientManager
cm = ClientManager()
cm.refresh_oidc_token()
```

4. get token info

```
from idds.client.clientmanager import ClientManager
cm = ClientManager()
cm.check_oidc_token_status()
```

5. clean oidc token

```
from idds.client.clientmanager import ClientManager
cm = ClientManager()
cm.clean_oidc_token()
```

3.2.2 iDDS OIDC Command Line Interface (CLI)

1. Setup the client. It's for users to setup the client for the first time or update the client configurations. By default it will create a file in ~/.idds/idds_local.cfg to remember these configurations.

```
idds setup --auth_type oidc --host https://<hostname or ip>:443/idds --vo Rubin
```

2. setup oidc token

```
idds setup_oidc_token
```

3. refresh oidc token

```
idds refresh_oidc_token
```

4. get token info

```
idds get_oidc_token_info
```

5. clean oidc token

```
idds clean_oidc_token
```

3.2.3 iDDS workflow manager

1. submit a workflow to the idds server

Below is one example for submitting a workflow.

```
from idds.client.clientmanager import ClientManager
from idds.common.utils import get_rest_host

# get the host from the client cfg
host = get_rest_host()
# or for example, host = https://iddsserver.cern.ch:443/idds

# get a workflow
workflow = get_workflow()

cm = ClientManager(host=host)         # here the host will overwrite the host defined in␣
↪the configurations.
request_id = cm.submit(workflow)
```

Below is an example for data carousel

```
def get_workflow():
    from idds.workflow.workflow import Workflow
    from idds.atlas.workflow.atlasstageinwork import ATLASStageinWork

    scope = 'data16_13TeV'
    name = 'data16_13TeV.00298862.physics_Main.daq.RAW'
    src_rse = 'NDGF-T1_DATATAPE'
    dest_rse = 'NDGF-T1_DATADISK'
    rule_id = '*****'
    workload_id = <panda task id for example>
    work = ATLASStageinWork(primary_input_collection={'scope': scope, 'name': name},
                            output_collections={'scope': scope, 'name': name + '.idds.
↪stagein'},
                            max_waiting_time=max_waiting_time,
                            src_rse=src_rse,
                            dest_rse=dest_rse,
                            rule_id=rule_id)
    wf = Workflow()
    wf.set_workload_id(workload_id)
    wf.add_work(work)
    return wf
```

Below is an example for hyperparameter optimization

```
def get_workflow():
    from idds.workflow.workflow import Workflow
    from idds.atlas.workflow.atlashpowork import ATLASHPOWork

    # request_metadata for predefined method 'nevergrad'
    request_metadata = {'workload_id': '20525135', 'sandbox': None, 'method': 'nevergrad
↪', 'opt_space': {"A": {"type": "Choice", "params": {"choices": [1, 4]}}, "B": {"type":
↪"Scalar", "bounds": [0, 5]}}, 'initial_points': [({'A': 1, 'B': 2}, 0.3), ({'A': 1, 'B
↪': 3}, None)], 'max_points': 20, 'num_points_per_generation': 10}

    # request_metadata for docker method
    request_metadata = {'workload_id': '20525134', 'sandbox': 'wguanicedew/idds_hpo_
↪nevergrad', 'workdir': '/data', 'executable': 'docker', 'arguments': 'python /opt/
↪hyperparameteropt_nevergrad.py --max_points=%MAX_POINTS --num_points=%NUM_POINTS --
↪input=/data/%IN --output=/data/%OUT', 'output_json': 'output.json', 'opt_space': {"A":
↪{"type": "Choice", "params": {"choices": [1, 4]}}, "B": {"type": "Scalar", "bounds":␣
↪[0, 5]}}, 'initial_points': [({'A': 1, 'B': 2}, 0.3), ({'A': 1, 'B': 3}, None)], 'max_
↪points': 20, 'num_points_per_generation': 10}

    work = ATLASHPOWork(executable=request_metadata.get('executable', None),
                        arguments=request_metadata.get('arguments', None),
                        parameters=request_metadata.get('parameters', None),
                        setup=None, exec_type='local',
                        sandbox=request_metadata.get('sandbox', None),
                        method=request_metadata.get('method', None),
                        container_workdir=request_metadata.get('workdir', None),
                        output_json=request_metadata.get('output_json', None),
                        opt_space=request_metadata.get('opt_space', None),
                        initial_points=request_metadata.get('initial_points', None),
```

(continues on next page)

(continued from previous page)

```
                        max_points=request_metadata.get('max_points', None),
                        num_points_per_iteration=request_metadata.get('num_points_per_
↪iteration', 10))
    wf = Workflow()
    wf.set_workload_id(request_metadata.get('workload_id', None))
    wf.add_work(work)
    return wf
```

2. Abort a request

```
# One of workload_id or request_id can be None
clientmanager.abort(request_id=<request_id>, workload_id=<workload_id>)
```

3. Suspend a request

```
# One of workload_id or request_id can be None
clientmanager.suspend(request_id=<request_id>, workload_id=<workload_id>)
```

4. Resume a request

```
# One of workload_id or request_id can be None
clientmanager.resume(request_id=<request_id>, workload_id=<workload_id>)
```

5. Retry a request

```
# One of workload_id or request_id can be None
clientmanager.retry(request_id=<request_id>, workload_id=<workload_id>)
```

6. Finish a request

```
# One of workload_id or request_id can be None
# if set_all_finished is set, all left files will be set finished
clientmanager.finish(request_id=<request_id>, workload_id=<workload_id>, set_all_
↪finished=False)
```

7. Get progress report

```
# One of workload_id or request_id can be None
clientmanager.get_status(request_id=<request_id>, workload_id=<workload_id>, with_
↪detail=False/True)
```

8. Download logs for a request

```
# One of workload_id or request_id can be None
clientmanager.download_logs(request_id=<request_id>, workload_id=<workload_id>, dest_dir=
↪'./', filename=None)
```

9. Upload a file to the iDDS cacher

```
# filename is the source filename or full path of the source file.
# Upload file to iDDS cacher: On the cacher, the filename will be the basename of the␣
↪file.
clientmanager.upload_to_cacher(filename)
```

10. Download a file from the iDDS cacher

```
# filename is the destination filename or full path of the destination file.
# Download file from iDDS cacher: On the cacher, the filename will be the basename of␣
↪the file.
clientmanager.download_from_cacher(filename)
```

11. Get hyperparameters

```
clientmanager.get_hyperparameters(request_id=<request_id>, workload_id=<workload_id>,
                                  id=<id>, status=<status>, limit=<limit>)

from idds.client.clientmanager import ClientManager
clientmanager = ClientManager(host='https://aipanda160.cern.ch:443/idds')
clientmanager.get_hyperparameters(workload_id=123, request_id=None)
clientmanager.get_hyperparameters(workload_id=None, request_id=456)
clientmanager.get_hyperparameters(workload_id=None, request_id=456, id=0)
```

12. Update hyperparameter

```
clientmanager.update_hyperparameter(request_id=<request_id>, workload_id=<workload_id>,
                                    id=<id>, loss=<loss>)
```

13. Get messages

```
clientmanager.get_messages(request_id=<idds_request_id>, workload_id=<workload_id>)

from idds.client.clientmanager import ClientManager
host = 'https://iddsserver.cern.ch:443/idds'
clientmanager = ClientManager(host=host)     # here the host will overwirte the host␣
↪defined in the configurations.

# clientmanager = ClientManager()  #  if idds.cfg is configured with [rest] host.

ret = clientmanager.get_messages(request_id=<idds_request_id>)
ret = clientmanager.get_messages(workload_id=<JEDI_task_id>)
status, msgs = ret
```

3.2.4 iDDS Command Line Interface (CLI)

1. Abort a request

```
# One of workload_id or request_id can be None
idds abort_requests --request_id=<request_id> --workload_id=<workload_id>
```

2. Suspend a request

```
# One of workload_id or request_id can be None
idds suspend_requests --request_id=<request_id> --workload_id=<workload_id>
```

3. Resume a request

```
# One of workload_id or request_id can be None
idds resume_requests --request_id=<request_id> --workload_id=<workload_id>
```

4. Retry a request

```
# One of workload_id or request_id can be None
idds retry_requests --request_id=<request_id> --workload_id=<workload_id>
```

5. Finish a request

```
# One of workload_id or request_id can be None
idds finish_requests --request_id=<request_id> --workload_id=<workload_id> [--set_all_
↪finished]
```

6. Get progress report

```
# One of workload_id or request_id can be None
idds get_requests_status --request_id=<request_id> --workload_id=<workload_id> --with_
↪detail

# idds get_requests_status --request_id 94
request_id    request_workload_id    scope:name                                  status    ␣
↪errors
------------  ---------------------  ------------------------------------------  -------- ␣
↪-----------
          94             1616422511  pseudo_dataset:pseudo_input_collection#1  Finished
↪{'msg': ''}

# idds get_requests_status --request_id 94 --with_detail
request_id    transform_id    request_workload_id    transform_workload_id    scope:name␣
↪                                status[Total/OK/Processing]    errors
------------  --------------  ---------------------  -----------------------  -----------
↪------------------------------  ------------------------------  -----------
          94             151             1616422511                     1003  pseudo_
↪dataset:pseudo_output_collection#1  Finished[6/6/0]                {'msg': ''}
          94             152             1616422511                     1002  pseudo_
↪dataset:pseudo_output_collection#2  Finished[3/3/0]                {'msg': ''}
          94             153             1616422511                     1001  pseudo_
↪dataset:pseudo_output_collection#3  Finished[5/5/0]                {'msg': ''}
```

7. Download logs for a request

```
# One of workload_id or request_id can be None
idds download_logs --request_id=<request_id> --workload_id=<workload_id> --dest_dir='./'␣
↪--filename=<filename>
```

8. Upload a file to the iDDS cacher

```
# filename is the source filename or full path of the source file.
# Upload file to iDDS cacher: On the cacher, the filename will be the basename of the␣
↪file.
idds upload_to_cacher --filename=<filename>
```

9. Download a file from the iDDS cacher

```
# filename is the destination filename or full path of the destination file.
# Download file from iDDS cacher: On the cacher, the filename will be the basename of␣
↪the file.
```

(continues on next page)

(continued from previous page)

```
idds download_from_cacher --filename=<filename>
```

10. Get hyperparameters

```
idds get_hyperparameters --request_id=<request_id> --workload_id=<workload_id>
                         --id=<id> --status=<status> --limit=<limit>)

idds get_hyperparameters --workload_id=123
idds get_hyperparameters --request_id=456
idds get_hyperparameters --request_id=456 --id=0
```

11. Update hyperparameter

```
idds update_hyperparameter --request_id=<request_id> --workload_id=<workload_id>,
                           --id=<id> --loss=<loss>
```

12. Get messages

```
idds get_messsage --request_id=<request_id> --workload_id=<workload_id>

idds get_messages --request_id=75483
idds get_messages --workload_id=25792557
```

3.3 Conditions: Examples

iDDS provides composite conditions to support complicated workflows.

3.3.1 conditions

1. work custom conditions

Here are examples how to define custom conditions with work attributes

```
from idds.workflow.work import Work

work1 = Work(executable='/bin/hostname', arguments=None, sandbox=None, work_id=1)
work1.add_custom_condition('to_exit', True)    # it's equal to: work1.add_custom_
↪condition('to_exit', True, op='and')
assert(work1.get_custom_condition_status() is False)
work1.to_exit = False
assert(work1.get_custom_condition_status() is False)
work1.to_exit = 'True'
assert(work1.get_custom_condition_status() is False)
work1.to_exit = True
assert(work1.get_custom_condition_status() is True)

# or_custom_conditions or (and_custom_conditions)
work1 = Work(executable='/bin/hostname', arguments=None, sandbox=None, work_id=1)
# to_exit or (to_exit1 or to_exit2)
work1.add_custom_condition('to_exit', True, op='and')
```

(continues on next page)

(continued from previous page)

```
work1.add_custom_condition('to_exit1', True, op='or')
work1.add_custom_condition('to_exit2', True, op='or')
assert(work1.get_custom_condition_status() is False)
work1.to_exit1 = False
assert(work1.get_custom_condition_status() is False)
work1.to_exit1 = 'False'
assert(work1.get_custom_condition_status() is False)
work1.to_exit1 = True
assert(work1.get_custom_condition_status() is True)
work1.to_exit1 = False
work1.to_exit2 = 'true'
assert(work1.get_custom_condition_status() is False)
work1.to_exit2 = True
assert(work1.get_custom_condition_status() is True)
work1.to_exit1 = False
work1.to_exit2 = False
work1.to_exit = True
assert(work1.get_custom_condition_status() is True)
assert(work1.get_not_custom_condition_status() is False)
```

2. condtions combination

Here are conditions iDDS supports

```
from idds.workflow.work import Work, WorkStatus
from idds.workflow.workflow import (CompositeCondition, AndCondition, OrCondition,␣
↪Condition,
                                    ConditionOperator, ConditionTrigger, Workflow)

com_cond = CompositeCondition(operator=ConditionOperator.And, conditions=[], true_
↪works=[], false_works=[])

and_cond = AndCondition(conditions=[], true_works=[], false_works=[])
         = CompositeCondition(operator=ConditionOperator.And, conditions=[], true_
↪works=[], false_works=[])

or_cond = OrCondition(conditions=[], true_works=[], false_works=[])
        = CompositeCondition(operator=ConditionOperator.Or, conditions=[], true_works=[],
↪ false_works=[])

# To support old conditions
cond = Condition(cond=<cond>, true_work=<true_work>, false_work=<false_work>)
     = CompositeCondition(operator=ConditionOperator.And, conditions=[cond], true_
↪works=[true_work], false_works=[false_work])


# combine multiple condtions into one condition
and_cond = AndCondition(conditions=[work5.is_terminated], true_works=[work6])
or_cond = OrCondition(conditions=[work7.is_terminated, work8.is_terminated], true_
↪works=[work9])
conds = AndCondition(conditions=[work1.is_finished, work2.is_started],
                     true_works=[work3, or_cond], false_works=[work4, and_cond])
# for combined conditions, only need to add the top(top tree) condition to the workflow.
```

(continues on next page)

(continued from previous page)

```
# Since every add_condition will create an evaluation entrypoint in iDDS.
#  If a branch of a combined condition is added to a workflow with add_condition, this␣
↪branch will be evaluated as a separate condition tree.
workflow.add_condition(conds)
```

3. condition trigger

(This part is for iDDS developers. Users normally should not use this function.) Condition trigger is an option for iDDS to process whether to remember whether a condition work is already triggered. It's used to avoid duplicated triggering some processes(For example, when using work1.is_started to trigger work2. If the condition is not recorded, work2 can be triggered many times every time when the condition is evaluated).

```
class ConditionTrigger(IDDSEnum):
     NotTriggered = 0
     ToTrigger = 1
     Triggered = 2

 # ToTrigger will return untriggered works based on the conditions and mark the work as␣
↪triggered.
 # exception: if the work.is_template is true, even the condition work is marked as␣
↪triggered, the work will still be triggered. So for cases such as work.is_started␣
↪should not be used as a condition for works with is_template=True.
 cond.get_next_works(trigger=ConditionTrigger.ToTrigger)

 # Will only return untriggered works based on conditions. It will not update the␣
↪trigger status.
 cond.get_next_works(trigger=ConditionTrigger.NotTriggered)

 # Will only return triggered works based on conditions. It will not update the trigger␣
↪status.
 cond.get_next_works(trigger=ConditionTrigger.Triggered)
```

3.4 Workflow: Examples

iDDS examples for subworkflow and loopworkflow. (example: main/lib/idds/tests/test_workflow_condition_v2.py)

3.4.1 Loop workflow

Here is a simple example of loop workflow.

```
from idds.workflowv2.work import Work, WorkStatus
from idds.workflowv2.workflow import (CompositeCondition, AndCondition, OrCondition,
                                      Condition, ConditionTrigger, Workflow,␣
↪ParameterLink)

work1 = Work(executable='/bin/hostname', arguments=None, sandbox=None, work_id=1)
work2 = Work(executable='/bin/hostname', arguments=None, sandbox=None, work_id=2)

workflow = Workflow()
workflow.add_work(work1, initial=False)
```

(continues on next page)

(continued from previous page)

```
workflow.add_work(work2, initial=False)

cond = Condition(cond=work2.is_finished)
workflow.add_loop_condition(cond)

# custom_condition
# iDDS will check to_contiue status.
# With ATLASLocalPanDAWork, the output will be parsed. If 'to_continue' is in the output,␣
↪it will be used.
work2.add_custom_condition(key='to_continue', value=True)
cond = Condition(work2.get_custom_condition_status)
workflow.add_loop_condition(cond1)

# multiple custom_condition
# to_continue and to_continue1
work2.add_custom_condition(key='to_continue', value=True, op='and')
work2.add_custom_condition(key='to_continue1', value=True, op='and')
cond = Condition(work2.get_custom_condition_status)
workflow.add_loop_condition(cond1)

# multiple custom_condition
# (to_continue and to_continue1) or to_exit or to_exit1
work2.add_custom_condition(key='to_continue', value=True, op='and')
work2.add_custom_condition(key='to_continue1', value=True, op='and')
work2.add_custom_condition(key='to_exit', value=False, op='or')
work2.add_custom_condition(key='to_exit1', value=False, op='or')
cond = Condition(work2.get_custom_condition_status)
workflow.add_loop_condition(cond1)
```

3.4.2 Sub workflow

Here is a simple example of sub workflow.

```
from idds.workflowv2.work import Work, WorkStatus
from idds.workflowv2.workflow import (CompositeCondition, AndCondition, OrCondition,
                                      Condition, ConditionTrigger, Workflow,␣
↪ParameterLink)

work1 = Work(executable='/bin/hostname', arguments=None, sandbox=None, work_id=1)
work2 = Work(executable='/bin/hostname', arguments=None, sandbox=None, work_id=2)

workflow1 = Workflow()
workflow1.add_work(work1, initial=False)
workflow1.add_work(work2, initial=False)

work3 = Work(executable='/bin/hostname', arguments=None, sandbox=None, work_id=3)

workflow = Workflow()
workflow.add_work(work3, initial=False)
workflow.add_work(workflow1, initial=False)
```

3.4.3 Sub loop workflow with ParameterLinks

Here is a simple example of sub loop workflow with parameter links.

```
from idds.workflowv2.work import Work, WorkStatus
from idds.workflowv2.workflow import (CompositeCondition, AndCondition, OrCondition,
                                      Condition, ConditionTrigger, Workflow,␣
↪ParameterLink)

work1 = Work(executable='/bin/hostname', arguments=None, sandbox=None, work_id=1,
             primary_input_collection={'scope': 'test_scop', 'name': 'input_test_work_1'}
↪,
             primary_output_collection={'scope': 'test_scop', 'name': 'output_test_work_1
↪'})
work2 = Work(executable='/bin/hostname', arguments=None, sandbox=None, work_id=2,
             primary_input_collection={'scope': 'test_scop', 'name': 'input_test_work_2'}
↪,
             primary_output_collection={'scope': 'test_scop', 'name': 'output_test_work_2
↪'})

workflow1 = Workflow()
workflow1.add_work(work1, initial=False)
workflow1.add_work(work2, initial=False)

cond1 = Condition(cond=work1.is_finished, true_work=work2)
workflow1.add_condition(cond1)

p_link = ParameterLink(parameters=[{'source': 'primary_output_collection',
                                    'destination': 'primary_input_collection'}])
workflow1.add_parameter_link(work1, work2, p_link)

cond = Condition(cond=work2.is_finished)
workflow1.add_loop_condition(cond)

work3 = Work(executable='/bin/hostname', arguments=None, sandbox=None, work_id=3,
             primary_input_collection={'scope': 'test_scop', 'name': 'input_test_work_3'}
↪,
             primary_output_collection={'scope': 'test_scop', 'name': 'output_test_work_3
↪'})
cond2 = Condition(cond=work3.is_finished, true_work=workflow1)
p_link1 = ParameterLink(parameters=[{'source': 'primary_output_collection',
                                     'destination': 'primary_input_collection'}])

workflow = Workflow()
workflow.add_work(work3, initial=False)
workflow.add_work(workflow1, initial=False)
workflow.add_condition(cond2)
workflow.add_parameter_link(work3, work1, p_link1)
```

3.4.4 Workflow with global parameters

Here is a simple example of workflow with global parameters. When a work starts, the work will use the global parameters and call 'setattr' to set the attributes for this work. When a work terminates, idds will call getattr to get the values for global parameters and store them in the global parameters. However, to avoid the global parameters overwrite the work's private attributes, currently only parameters start with '**user_**' will be accepted as global parameters. Other parameters will be ignored with a warning logging messages.

```
from idds.workflowv2.work import Work, WorkStatus
from idds.workflowv2.workflow import (CompositeCondition, AndCondition, OrCondition,
                                      Condition, ConditionTrigger, Workflow,␣
↪ParameterLink)

work1 = Work(executable='/bin/hostname', arguments=None, sandbox=None, work_id=1,
             primary_input_collection={'scope': 'test_scop', 'name': 'input_test_work_1'}
↪,
             primary_output_collection={'scope': 'test_scop', 'name': 'output_test_work_1
↪'})
work2 = Work(executable='/bin/hostname', arguments=None, sandbox=None, work_id=2,
             primary_input_collection={'scope': 'test_scop', 'name': 'input_test_work_2'}
↪,
             primary_output_collection={'scope': 'test_scop', 'name': 'output_test_work_2
↪'})

workflow1 = Workflow()
workflow1.add_work(work1, initial=False)
workflow1.add_work(work2, initial=False)

# global paraeters
workflow1.set_global_parameters({'user_attr1': 1, 'user_attr2': 2})

# sliced global parameters
workflow1.set_global_parameters({'user_attr': [1, 2, 3]})
workflow1.set_sliced_global_parameters(source='user_attr', index=0)
workflow1.set_sliced_global_parameters(source='user_attr', index=1, name='user_myattr')
```

3.5 Monitor API: Examples

iDDS provides a group of monitor APIs which returns JSON outputs.

3.5.1 Request information

It returns a summary information monthly(accumulated and not accumulated).

```
curl https://hostname:443/idds/monitor_request/<request_id>/<workload_id>
curl https://hostname:443/idds/monitor_request/null/null

{
"month_acc_status": {
    "Finished": {
```

(continues on next page)

(continued from previous page)


```
        "2021-05": 1,
        "2021-06": 1,
        "2021-07": 1,
        "2021-08": 13,
        "2021-09": 13,
        "2021-10": 28
    },
    "Total": {
        "2021-05": 8,
        "2021-06": 8,
        "2021-07": 8,
        "2021-08": 52,
        "2021-09": 54,
        "2021-10": 80
    },
},
"month_status": {
    "Finished": {
        "2021-05": 1,
        "2021-06": 0,
        "2021-07": 0,
        "2021-08": 12,
        "2021-09": 0,
        "2021-10": 15
    },
    "SubFinished": {
        "2021-05": 0,
        "2021-06": 0,
        "2021-07": 0,
        "2021-08": 15,
        "2021-09": 2,
        "2021-10": 1
    },
    "Total": {
        "2021-05": 8,
        "2021-06": 0,
        "2021-07": 0,
        "2021-08": 44,
        "2021-09": 2,
        "2021-10": 26
    },
},
"total": 80
}
```


Here it returns the detail of requests.

```
curl https://hostname:443/idds/monitor/<request_id>/<workload_id>/true/false/false
curl https://hostname:443/idds/monitor/null/null/true/false/false
curl https://hostname:443/idds/monitor/210/null/true/false/false
[
{
```

(continues on next page)

(continued from previous page)


```
    "created_at": "Fri, 22 Oct 2021 22:17:42 UTC",
    "input_coll_bytes": 3,
    "input_processed_files": 14,
    "input_processing_files": 0,
    "input_total_files": 14,
    "output_coll_bytes": 14,
    "output_processed_files": 14,
    "output_processing_files": 0,
    "output_total_files": 14,
    "request_id": 210,
    "status": "Finished",
    "transforms": {
        "Finished": 3
    },
    "updated_at": "Fri, 22 Oct 2021 23:55:13 UTC",
    "workload_id": 1634941059
}
]
```


3.5.2 Transform information

It returns a summary information monthly(accumulated and not accumulated).


```
curl https://hostname:443/idds/monitor_transform/<request_id>/<workload_id>
curl https://hostname:443/idds/monitor_transform/null/null
curl https://hostname:443/idds/monitor_transform/210/null
{
"month_acc_processed_bytes": {
    "Finished": {
        "2021-10": 3
    },
    "Total": {
        "2021-10": 3
    }
},
"month_acc_processed_bytes_by_type": {
    "Processing": {
        "Finished": {
            "2021-10": 3
        },
        "Total": {
            "2021-10": 3
        }
    }
},
"month_acc_processed_files": {
    "Finished": {
        "2021-10": 14
    },
    "Total": {
        "2021-10": 14
```


(continues on next page)

(continued from previous page)


```
    }
},
"month_acc_processed_files_by_type": {
    "Processing": {
        "Finished": {
            "2021-10": 14
        },
        "Total": {
            "2021-10": 14
        }
    }
},
"month_acc_status": {
    "Finished": {
        "2021-10": 3
    },
    "Total": {
        "2021-10": 3
    }
},
"month_acc_status_dict_by_type": {
    "Processing": {
        "Finished": {
            "2021-10": 3
        },
        "Total": {
            "2021-10": 3
        }
    }
},
"month_processed_bytes": {
    "Finished": {
        "2021-10": 3
    },
    "Total": {
        "2021-10": 3
    }
},
"month_processed_bytes_by_type": {
    "Processing": {
        "Finished": {
            "2021-10": 3
        },
        "Total": {
            "2021-10": 3
        }
    }
},
"month_processed_files": {
    "Finished": {
        "2021-10": 14
    },
```


(continues on next page)

(continued from previous page)


```
    "Total": {
        "2021-10": 14
    }
},
"month_processed_files_by_type": {
    "Processing": {
        "Finished": {
            "2021-10": 14
        },
        "Total": {
            "2021-10": 14
        }
    }
},
"month_status": {
    "Finished": {
        "2021-10": 3
    },
    "Total": {
        "2021-10": 3
    }
},
"month_status_dict_by_type": {
    "Processing": {
        "Finished": {
            "2021-10": 3
        },
        "Total": {
            "2021-10": 3
        }
    }
},
"total": 3,
"total_bytes": 17,
"total_files": 14
}
```


Here it returns the list of detailed transforms.

```
curl https://hostname:443/idds/monitor/<request_id>/<workload_id>/false/true/false
curl https://hostname:443/idds/monitor/null/null/false/true/false
curl https://hostname:443/idds/monitor/210/null/false/true/false
```


```
[
{
    "errors": {
        "msg": ""
    },
    "input_coll_bytes": 1,
    "input_processed_files": 3,
    "input_processing_files": 0,
    "input_total_files": 3,
    "output_coll_bytes": 3,
```


(continues on next page)

(continued from previous page)


```
    "output_coll_name": "pseudo_output_collection#2",
    "output_coll_scope": "pseudo_dataset",
    "output_processed_files": 3,
    "output_processing_files": 0,
    "output_total_files": 3,
    "request_id": 210,
    "transform_created_at": "Fri, 22 Oct 2021 22:50:16 UTC",
    "transform_finished_at": "Fri, 22 Oct 2021 23:20:43 UTC",
    "transform_id": 2445,
    "transform_status": "Finished",
    "transform_type": "Processing",
    "transform_updated_at": "Fri, 22 Oct 2021 23:20:43 UTC",
    "transform_workload_id": 7169,
    "workload_id": 1634941059
},
{
    "errors": {
        "msg": ""
    },
    "input_coll_bytes": 1,
    "input_processed_files": 6,
    "input_processing_files": 0,
    "input_total_files": 6,
    "output_coll_bytes": 6,
    "output_coll_name": "pseudo_output_collection#1",
    "output_coll_scope": "pseudo_dataset",
    "output_processed_files": 6,
    "output_processing_files": 0,
    "output_total_files": 6,
    "request_id": 210,
    "transform_created_at": "Fri, 22 Oct 2021 22:17:45 UTC",
    "transform_finished_at": "Fri, 22 Oct 2021 22:46:15 UTC",
    "transform_id": 2444,
    "transform_status": "Finished",
    "transform_type": "Processing",
    "transform_updated_at": "Fri, 22 Oct 2021 22:46:15 UTC",
    "transform_workload_id": 7168,
    "workload_id": 1634941059
},
{
    "errors": {
        "msg": ""
    },
    "input_coll_bytes": 1,
    "input_processed_files": 5,
    "input_processing_files": 0,
    "input_total_files": 5,
    "output_coll_bytes": 5,
    "output_coll_name": "pseudo_output_collection#3",
    "output_coll_scope": "pseudo_dataset",
    "output_processed_files": 5,
    "output_processing_files": 0,
```


(continues on next page)

(continued from previous page)

```
    "output_total_files": 5,
    "request_id": 210,
    "transform_created_at": "Fri, 22 Oct 2021 23:24:46 UTC",
    "transform_finished_at": "Fri, 22 Oct 2021 23:53:15 UTC",
    "transform_id": 2446,
    "transform_status": "Finished",
    "transform_type": "Processing",
    "transform_updated_at": "Fri, 22 Oct 2021 23:53:15 UTC",
    "transform_workload_id": 7170,
    "workload_id": 1634941059
}
]
```

3.5.3 Processing information

Here it returns a summary information monthly(accumulated and not accumulated).

```
curl https://hostname:443/idds/monitor_processing/<request_id>/<workload_id>
curl https://hostname:443/idds/monitor_processing/null/null
curl https://hostname:443/idds/monitor_processing/210/null
{
"month_acc_status": {
    "Finished": {
        "2021-10": 3
    },
    "Total": {
        "2021-10": 3
    }
},
"month_status": {
    "Finished": {
        "2021-10": 3
    },
    "Total": {
        "2021-10": 3
    }
},
"total": 3
}
```

```
curl https://hostname:443/idds/monitor/<request_id>/<workload_id>/false/false/true
curl https://hostname:443/idds/monitor/null/null/false/false/true
curl https://hostname:443/idds/monitor/210/null/false/false/true
[
{
    "processing_created_at": "Fri, 22 Oct 2021 23:24:50 UTC",
    "processing_finished_at": "Fri, 22 Oct 2021 23:53:10 UTC",
    "processing_id": 1230,
    "processing_status": "Finished",
    "processing_updated_at": "Fri, 22 Oct 2021 23:53:10 UTC",
```

(continues on next page)

(continued from previous page)


```
    "request_id": 210,
    "workload_id": 7170
},
{
    "processing_created_at": "Fri, 22 Oct 2021 22:50:20 UTC",
    "processing_finished_at": "Fri, 22 Oct 2021 23:18:40 UTC",
    "processing_id": 1229,
    "processing_status": "Finished",
    "processing_updated_at": "Fri, 22 Oct 2021 23:18:40 UTC",
    "request_id": 210,
    "workload_id": 7169
},
{
    "processing_created_at": "Fri, 22 Oct 2021 22:17:49 UTC",
    "processing_finished_at": "Fri, 22 Oct 2021 22:46:05 UTC",
    "processing_id": 1228,
    "processing_status": "Finished",
    "processing_updated_at": "Fri, 22 Oct 2021 22:46:05 UTC",
    "request_id": 210,
    "workload_id": 7168
}
]
```


3.5.4 DAG relationships

Here it returns the request information with dag relationships. It returns a list of works. For every work, it returns "work" for work data and "next_works" for its followings(if existing).

```
curl https://hostname:443/idds/monitor_request_relation/<request_id>/<workload_id>
curl https://hostname:443/idds/monitor_request_relation/212/null

[{
   ......
   "relation_map": [
       {
           "next_works": [
               {
                   "work": {
                       "external_id": null,
                       "workload_id": 7175
                   }
               }
           ],
           "work": {
               "external_id": null,
               "workload_id": 7174
           }
       }
   ],
   ......
}]
```

If there is a loop workflow or a sub loop wookflow. The returned format will be:

3.6 Administrator guides

Here is a quick tutorial for setup an iDDS server.

3.6.1 Environment setup on CENTOS 7

1. setup environment for the first time.

```
yum install -y httpd.x86_64 conda gridsite mod_ssl.x86_64 httpd-devel.x86_64 gcc.x86_64␣
↪supervisor.noarch
mkdir /opt/idds
mkdir /opt/idds_source
mkdir /opt/idds
mkdir /var/log/idds
mkdir /var/log/idds/wsgisocks
chown atlpilo1 -R /opt/idds
chown atlpilo1 -R /opt/idds_source
chown atlpilo1 /var/log/idds
chown apache -R /var/log/idds/wsgisocks

source /etc/profile.d/conda.sh
conda create --prefix=/opt/idds python=3.6.2
conda activate /opt/idds
pip install idds-server idds-doma idds-atlas idds-monitor idds-website

pip install rucio-clients-atlas rucio-clients panda-client-light
# # add "auth_type = x509_proxy" to /opt/idds/etc/rucio.cfg
```

2. setup environment after installed.

```
source /etc/profile.d/conda.sh
conda activate /opt/idds

pip install --upgrade idds-server idds-doma idds-atlas idds-monitor idds-website
```

3. Configure REST service

The Rest service is based http server. By default it's using the port 443. If the port 443 is used, you need to comment out the 443 port in ssl.conf.

```
# configure httpd service
cp /opt/idds/etc/idds/rest/httpd-idds-443-py36-cc7.conf.install_template /etc/httpd/conf.
↪d/httpd-idds-443-py36-cc7.conf
# comment /etc/httpd/conf.d/ssl.conf "Listen 443 https"
systemctl restart httpd.service
systemctl enable httpd.service
```

4. Configure iDDS agents

"supervisord" is employed to manage iDDS agents. Here is the configuration.

```
# configure iDDS agents
cp /opt/idds/etc/idds/supervisord.d/idds.ini /etc/supervisord.d/idds.ini
cp /opt/idds_source/main/etc/idds/supervisord.d/idds.ini /etc/supervisord.d/idds.ini
systemctl start supervisord
systemctl status supervisord
systemctl enable supervisord
```

Normally, these agents are required for idds. a) Clerk, to manage requests and workflow. b) Transformer, to manage transforms, collections, contents and create processings. c) Carrier, to submit processings to workload manager and poll processings. d) Conductor, to send messages to ActiveMQ for workload managers to consume.

```
cp /opt/idds/etc/idds/idds.cfg.template /opt/idds/etc/idds/idds.cfg
# configure the database
```

5. Logs locations (httpd REST logs and idds agents logs)

```
ls /var/log/idds
ls /var/log/idds/httpd_error_log
ls /var/log/idds/idds-server-std*
# Normally grep 'Traceback' can find the errors.
```

6. Restart service

```
systemctl stop httpd
systemctl start httpd

# restart agents
supervisorctl stop all
supervisorctl start all
```

3.7 Contributor Guide

- Thank you for participating!
- Below are the guids about how to contribute to it.

The repository consists of different branches:

- the **master** branch includes the main stable developments for the next major version.
- the **dev** branch includes latest developments. A contribution should be based on the dev branch.
- the **<version>** branch includes deployments for different versions.

Generally all pull requests are to be created against the iDDS **dev** branch. Contributions will end up in the upstream **dev** when merged.

3.7.1 Getting started

Step 1: Fork the repository on Github.

Step 2: Clone the repository to your development machine and configure it:

```
$ git clone https://github.com/<YOUR_USER>/iDDS/
$ cd iDDS
$ git remote add upstream https://github.com/HSF/iDDS.git
```

Step 3: Setup local dev environment(The virtual environment is based on conda):

```
$ # If you are not in the idds top directory.
$ # cd iDDS
$ CurrentDir="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
$ CondaDir=${CurrentDir}/.conda/iDDS
$ mkdir -p $CondaDir

$ # For your local development, you may do not need all packages. In this case, you may␣
↪need to comment out some packages, for example cx_Oracle.
$ echo conda env create --prefix=$CondaDir  -f=main/tools/env/environment.yml
$ conda env create --prefix=$CondaDir  -f=main/tools/env/environment.yml
```

Step 4: Configure local environment:

```
$ # If you are not in the idds top directory.
$ # cd iDDS
$ RootDir="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
$ CondaDir=${RootDir}/.conda/iDDS
$ export IDDS_HOME=$RootDir
$ conda activate $CondaDir
```

3.7.2 Contributing

Step 1: Developing based on your personal repository(if you already have some codes and want to keep them, you need to use 'git rebase'. In this case, you may need to fix some conflicts; If you start from scratch, you can directly use 'git reset'. This method will overwrite local changes. Becareful to backup if you are not confident about using it):

```
$ git checkout dev
$ # echo "Updating dev"
$ git pull --all --prune --progress
$ echo "Rebasing dev"
$ # git rebase upstream/dev dev
$ git reset --hard upstream/dev
```

Step 2: Check your codes and fix the codes(flake8 is installed when you configure your virtual environments):

```
$ flake8 yourcodes.py
$ flake8 */lib/idds/
```

Step 3: Commit your change. The commit command must include a specific message format:

```
$ git commit -m "...."
$ git push origin dev
```

Step 4: Create the pull request to the **dev** branch of HSF/iDDS repository.

While using the github interface is the default interface to create pull requests, you could also use GitHub's command-line wrapper hub or the GitHub CLI.

Step 5: Watch the pull request for comments and reviews. If there are some conflicts, you may need to rebase your codes and fix the conflicts. For any pull requests update, please try to squash/amend your commits to avoid "in-between" commits:

```
$ git rebase upstream/dev dev
```

3.7.3 Human Review

Anyone is welcome to review merge requests and make comments!

The development team can approve, request changes, or close pull requests. Merging of approved pull requests is done by the iDDS development lead.

3.7.4 Coding Style

We use flake8 to sanitize our code. Please do the same before submitting a pull request.

CHAPTER

FOUR

DOMA USER DOCUMENTATION

4.1 Installing iDDS Clients

4.1.1 Prerequisites

iDDS clients run on Python 2.7, 3.6 on any Unix-like platform.

4.1.2 Python Dependencies

All Dependencies are automatically installed with pip.

4.1.3 Install via pip

When `pip` is available, the distribution can be downloaded from the iDDS PyPI server and installed in one step:

```
$> pip install idds-common idds-client idds-workflow idds-doma
```

This command will download the latest version of Rucio and install it to your system.

4.1.4 Upgrade via pip

To upgrade via pip:

```
$> pip install --upgrade idds-common idds-client idds-workflow idds-doma
```

4.1.5 config client

To use iDDS client to access the iDDS server, a config file is needed. Below is an example of the config file:

```
$> idds setup --auth_type oidc --host https://<hostname or ip>:443/idds --vo Rubin
```

4.2 iDDS OIDC authorization

Here are the commands how to setup oidc tokens. For other client examples, please check normal user documents.

4.2.1 iDDS OIDC authorization

1. Setup the client. It's for users to setup the client for the first time or update the client configurations. By default it will create a file in ~/.idds/idds_local.cfg to remember these configurations.

```
from idds.client.clientmanager import ClientManager
cm = ClientManager()
cm.setup_local_configuration(local_config_root=<local_config_root>,  # default ~/.idds/
                             host=<host>,                             # default host for␣
↪different authorization methods. https://<hostname or ip>:443/idds
                             auth_type=<auth_type>,                   # authorization␣
↪type: x509_proxy, oidc
                             auth_type_host=<auth_type_host>,         # for different␣
↪authorization methods, users can define different idds servers.
                             x509_proxy=<x509_proxy path>,
                             vo=<vo name>,
```

2. setup oidc token

```
from idds.client.clientmanager import ClientManager
cm = ClientManager()
cm.setup_oidc_token()
```

3. refresh oidc token

```
from idds.client.clientmanager import ClientManager
cm = ClientManager()
cm.refresh_oidc_token()
```

4. get token info

```
from idds.client.clientmanager import ClientManager
cm = ClientManager()
cm.check_oidc_token_status()
```

5. clean oidc token

```
from idds.client.clientmanager import ClientManager
cm = ClientManager()
cm.clean_oidc_token()
```

4.2.2 iDDS OIDC Command Line Interface (CLI)

1. Setup the client. It's for users to setup the client for the first time or update the client configurations. By default it will create a file in ~/.idds/idds_local.cfg to remember these configurations.

```
idds setup --auth_type oidc --host https://<hostname or ip>:443/idds --vo Rubin
```

2. setup oidc token

```
idds setup_oidc_token
```

3. refresh oidc token

```
idds refresh_oidc_token
```

4. get token info

```
idds get_oidc_token_info
```

5. clean oidc token

```
idds clean_oidc_token
```

CHAPTER

FIVE

SOURCE CODES

5.1 iDDS Source Codes

5.1.1 iDDS Codes

iDDS codes documents are automatically generated by sphinx.

Common libraries

Common libaries are the basic libraries which are used by all other libraries.

idds

idds package

Subpackages

idds.common package

Subpackages

idds.common.plugin package

Submodules

idds.common.plugin.plugin_base module

idds.common.plugin.plugin_utils module

Module contents

Submodules

idds.common.config module

idds.common.constants module

idds.common.dict_class module

idds.common.exceptions module

idds.common.status_utils module

idds.common.utils module

idds.common.version module

Module contents

Module contents

Workflow libraries

Workflow libaries are the basic libraries which are used by all other libraries.

idds

idds package

Subpackages

idds.workflow package

Submodules

idds.workflow.base module

idds.workflow.version module

idds.workflow.work module

idds.workflow.workflow module

idds.workflow.workflow1 module

idds.workflow.workflowv2 module

Module contents

idds.workflowv2 package

Submodules

idds.workflowv2.base module

idds.workflowv2.version module

idds.workflowv2.work module

idds.workflowv2.workflow module

Module contents

Module contents

Main libraries

Main libaries include the core functions, RESTful service and daemon agents which run on the servers.

idds

idds package

Subpackages

idds.agents package

Subpackages

idds.agents.carrier package

Submodules

idds.agents.carrier.carrier module

Module contents

idds.agents.clerk package

Submodules

idds.agents.clerk.clerk module

Module contents

idds.agents.common package

Submodules

idds.agents.common.baseagent module

idds.agents.common.timerscheduler module

idds.agents.common.timertask module

Module contents

idds.agents.conductor package

Submodules

idds.agents.conductor.conductor module

idds.agents.conductor.consumer module

Module contents

idds.agents.marshaller package

Submodules

idds.agents.marshaller.marshaller module

Module contents

idds.agents.transformer package

Submodules

idds.agents.transformer.helper module

idds.agents.transformer.transformer module

Module contents

idds.agents.transporter package

Submodules

idds.agents.transporter.transporter module

Module contents

Submodules

idds.agents.main module

Module contents

idds.api package

Submodules

idds.api.catalog module

idds.api.collections module

idds.api.contents module

idds.api.processings module

idds.api.requests module

idds.api.transforms module

Module contents

idds.core package

Submodules

idds.core.catalog module

idds.core.health module

idds.core.messages module

idds.core.processings module

idds.core.requests module

idds.core.transforms module

idds.core.workprogress module

Module contents

idds.orm package

Subpackages

idds.orm.base package

Submodules

idds.orm.base.enum module

idds.orm.base.models module

idds.orm.base.session module

idds.orm.base.types module

idds.orm.base.utils module

Module contents

Submodules

idds.orm.collections module

idds.orm.contents module

idds.orm.health module

idds.orm.messages module

idds.orm.processings module

idds.orm.requests module

idds.orm.transforms module

idds.orm.workprogress module

Module contents

idds.rest package

Subpackages

idds.rest.v1 package

Submodules

idds.rest.v1.app module

idds.rest.v1.cacher module

idds.rest.v1.catalog module

idds.rest.v1.controller module

idds.rest.v1.hyperparameteropt module

idds.rest.v1.logs module

idds.rest.v1.messages module

idds.rest.v1.monitor module

idds.rest.v1.requests module

idds.rest.v1.utils module

Module contents

Module contents

idds.tests package

Submodules

idds.tests.activelearning_test module

idds.tests.cacher_test module

idds.tests.catalog_test module

idds.tests.client_test module

idds.tests.common module

idds.tests.core_tests module

idds.tests.datacarousel_test module

idds.tests.hyperparameteropt_bayesian_test module

idds.tests.hyperparameteropt_client_test module

idds.tests.hyperparameteropt_docker_local_test module

idds.tests.hyperparameteropt_docker_test module

idds.tests.hyperparameteropt_nevergrad_test module

idds.tests.logs_test module

idds.tests.message_test module

idds.tests.message_test1 module

idds.tests.migrating_requests_v1_to_v2 module

idds.tests.panda_iam_test module

idds.tests.panda_test module

idds.tests.performance_test_with_cx_oracle module

idds.tests.performance_test_with_sqlalchemy module

idds.tests.rest_test module

idds.tests.run_sql module

idds.tests.scaling_checks module

idds.tests.test_activelearning module

idds.tests.test_atlaspandawork module

idds.tests.test_catalog module

idds.tests.test_datacarousel module

idds.tests.test_domapanda module

idds.tests.test_domapanda_workflow module

idds.tests.test_hyperparameteropt module

idds.tests.test_migrate_requests module

idds.tests.test_property module

idds.tests.test_request_transform module

idds.tests.test_requests module

idds.tests.test_running_data module

idds.tests.test_scaling module

idds.tests.test_transform_collection_content module

idds.tests.test_transform_processing module

idds.tests.test_workflow module

idds.tests.test_workflow_condition module

idds.tests.test_workflow_condition_v2 module

idds.tests.trigger_release module

Module contents

Submodules

idds.version module

Module contents

Client libraries

Client libraries is used for users to communicate with iDDS service.

idds

idds package

Subpackages

idds.client package

Submodules

idds.client.base module

idds.client.cacherclient module

idds.client.catalogclient module

idds.client.client module

idds.client.clientmanager module

idds.client.hpoclient module

idds.client.logsclient module

idds.client.messageclient module

idds.client.requestclient module

idds.client.version module

Module contents

Module contents

ATLAS libraries

ATLAS libraries include plugins for ATLAS special services.

idds

idds package

Subpackages

idds.atlas package

Subpackages

idds.atlas.notifier package

Submodules

idds.atlas.notifier.messaging module

Module contents

idds.atlas.processing package

Submodules

idds.atlas.processing.activelearning_condor_poller module

idds.atlas.processing.activelearning_condor_submitter module

idds.atlas.processing.base_plugin module

idds.atlas.processing.condor_poller module

idds.atlas.processing.condor_submitter module

idds.atlas.processing.hyperparameteropt_bayesian module

idds.atlas.processing.hyperparameteropt_condor_poller module

idds.atlas.processing.hyperparameteropt_condor_submitter module

idds.atlas.processing.hyperparameteropt_nevergrad module

idds.atlas.processing.stagein_poller module

idds.atlas.processing.stagein_submitter module

Module contents

idds.atlas.rucio package

Submodules

idds.atlas.rucio.base_plugin module

idds.atlas.rucio.collection_lister module

idds.atlas.rucio.collection_metadata_reader module

idds.atlas.rucio.contents_lister module

idds.atlas.rucio.contents_register module

idds.atlas.rucio.rule_creator module

idds.atlas.rucio.rule_poller module

idds.atlas.rucio.rule_submitter module

Module contents

idds.atlas.transformer package

Submodules

idds.atlas.transformer.activelearning_transformer module

idds.atlas.transformer.base_plugin module

idds.atlas.transformer.hyperparameteropt_transformer module

idds.atlas.transformer.stagein_transformer module

Module contents

idds.atlas.workflow package

Submodules

idds.atlas.workflow.atlasactuatorwork module

idds.atlas.workflow.atlascondorwork module

idds.atlas.workflow.atlasdagwork module

idds.atlas.workflow.atlashpowork module

idds.atlas.workflow.atlaspandawork module

idds.atlas.workflow.atlasstageinwork module

Module contents

idds.atlas.workflowv2 package

Submodules

idds.atlas.workflowv2.atlasactuatorwork module

idds.atlas.workflowv2.atlascondorwork module

idds.atlas.workflowv2.atlasdagwork module

idds.atlas.workflowv2.atlashpowork module

idds.atlas.workflowv2.atlaspandawork module

idds.atlas.workflowv2.atlasstageinwork module

Module contents

Submodules

idds.atlas.version module

Module contents

Module contents

DOMA libraries

DOMA libraries include plugins for DOMA special services.

idds

idds package

Subpackages

idds.doma package

Subpackages

idds.doma.workflow package

Submodules

idds.doma.workflow.domapandawork module

Module contents

idds.doma.workflowv2 package

Submodules

idds.doma.workflowv2.domapandawork module

Module contents

Submodules

idds.doma.version module

Module contents

Module contents

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search