

Cosmos Documentation

Release 1.0

Laboratory for Personalized Medicine

CONTENTS

1.1 Features	5
1.2 Multi-platform Support	5
CHAPTER TWO: INSTALL.....	6
2.1 Requirements	6
2.2 Quick Install.....	6
2.3 Highly Recommended Install Method	6
2.4 Experimental Features	7
CHAPTER THREE: CONFIGURATION	8
3.1 1. Install Configuration File	8
3.1.1 Local Development and Testing	8
3.2 2. Create SQL Tables and Load Static Files	8
CHAPTER FOUR: GETTING STARTED.....	9
4.1 Execute an Example Workflow.....	9
4.2 Launch the Web Interface	9
4.3 Terminating a Workflow	10
4.4 Resuming a workflow	10
CHAPTER FIVE: WRITING WORKFLOWS WITH <i>EZFLOW</i>	12
5.1 Defining Tools.....	12
5.1.1 Overriding map_inputs	12
5.2 Defining Input Files	12
5.3 Designing Workflows	13
5.4 API.....	15
CHAPTER SIX: WORKFLOW CODE EXAMPLES	23
6.1 Basic Workflows	23

6.1.1 Hello World	23
6.1.2 Reload a Workflow	23
6.2 Advanced Workflows.....	24
6.2.1 Branching Workflows	24
6.2.2 Command Line Interface and Signals	26
CHAPTER SEVEN: COMMAND LINE INTERFACE	30
7.1 Examples	30
7.1.1 Get Usage Help:.....	30
7.1.2 Reset the SQL database.....	30
7.1.3 List workflows.....	30
CHAPTER EIGHT: COSMOS SHELL.....	31
8.1 Launch the IPython shell.....	31
8.2 Interactive Workflow	31
8.3 Filtering	31
8.4 Deleting.....	32
CHAPTER NINE : ADVANCED USAGE	33
9.1 Setting Custom Job Submission Flags	33
9.1.1 API	34
CHAPTER TEN: COSMOS FAQ.....	35
CHAPTER ELEVEN: API	36
11.1 Workflow.....	36
11.1.1 Primary Workflow Objects	36
11.1.2 API	36
11.2 JobManager.....	43
MISCELLANEOUS: TO BE ADDED SOMEWHERE.....	ERROR! BOOKMARK NOT DEFINED.
GLOSSARY	45
INDICES AND TABLES	ERROR! BOOKMARK NOT DEFINED.
PYTHON MODULE INDEX.....	ERROR! BOOKMARK NOT DEFINED.

CHAPTER ONE: Introduction

Cosmos is a workflow management system for Python. It allows you to efficiently program complex workflows of command line tools that automatically take advantage of a compute cluster, and provides a web interface to monitor, debug, and analyze your jobs.

1.1 Features

- Written in python which is easy to learn, powerful, and popular. A programmer with limited experience can begin writing Cosmos workflows right away.
- Powerful syntax and system for the creation of complex workflows.
- Keeps track of workflows, job information, and resource utilization and provenance in an SQL database.
- The ability to visualize all jobs and job dependencies as a convenient image.
- Monitor and debug running workflows, and a history of all workflows via a web interface.

1.2 Multi-platform Support

- Support for DRMS such as SGE, LSF, PBS/Torque, and Condor by utilizing DRMAA
- Supports for MySQL, PostgreSQL, Oracle, SQLite by using the Django ORM
- Extremely well suited for cloud computing

CHAPTER TWO: Install

2.1 Requirements

- The only other requirement is that DRMAA is installed on the system if you want Cosmos to submit jobs to a DRMS like LSF or Grid Engine.
- Cosmos requires python2.6 or python2.7 and is completely untested on python3.
- For Local Development and Testing, it is highly recommended that you install Ubuntu inside VirtualBox.
- Many python libraries won't be able to install unless their dependent software is already installed on the system. For example, pygraphviz requires graphviz-dev and python-mysql require python-dev libmysqlclient-dev. If pip install is failing, try running:

```
sudo apt-get update -y
sudo apt-get install python-dev libmysqlclient-dev mysql-server graphviz graphviz-dev
```

2.2 Quick Install

This is generally for advanced users who have worked with python packages before.

```
cd /dir/to/install/Cosmos/to

git clone git@github.com:ComputationalBiomedicine/Cosmos.git --depth=1

pip install distribute --upgrade

cd Cosmos

pip install .
```

Hint: You need root access to install python packages to the system directories. You may have to run pip install with 'sudo pip install ...', or to install to the user level use 'pip install --user'. 99% of users should just use the Highly Recommended Install Method described next.

2.3 Highly Recommended Install Method

Install Cosmos in a virtual environment using virtualenvwrapper. This will make sure all python libraries and files related to Cosmos are installed to a sandboxed location in `$HOME/.virtualenvs/cosmos`

```
pip install virtualenvwrapper --user

source $HOME/.local/bin/virtualenvwrapper.sh

echo "\nsource $HOME/.local/bin/virtualenvwrapper.sh" >> ~/.bash.rc

echo "PATH=$HOME/.local/bin:$PATH" >> ~/.bash.rc
```

```
mkvirtualenv cosmos --no-site-packages

cd /dir/to/install/Cosmos/to

pip install distribute --upgrade

git clone git@github.com:ComputationalBiomedicine/Cosmos.git --depth=1

cd Cosmos

pip install .
```

Cosmos will be installed to its own python virtual environment, which you can activate by executing the following virtualenvwrapper command:

```
$ workon cosmos
```

Make sure you execute workon cosmos anytime you want to interact with Cosmos, or run a script that uses Cosmos. Deactivate the virtual environment by executing:

```
$ deactivate
```

2.4 Experimental Features

Optionally, if you want the experimental graphing capabilities to automatically summarize computational resource usage, R and the R package ggplot2 are required.

```
sudo apt-get install r graphviz-dev # or whatever works on your OS

sudo R

> install.packages("ggplot2")
```

CHAPTER THREE: Configuration

3.1 1. Install Configuration File

Type `cosmos` at the command line, and generate a default configuration file in `~/ .cosmos/config.ini`. Edit `~/ .cosmos/config.ini`, and configure it to your liking.

3.1.1 Local Development and Testing

Setting `DRM = local` in your config file will cause jobs to be submitted as background processes on the local machine using `subprocess.Popen`. The `DRM = local` setting's primary purpose is for testing and developing workflows, not computing on large datasets.

Warning: Be careful how many resource intensive jobs your workflow submits at once when using `DRM = local`. Currently there's no way to set a ceiling on the number of processes being executed simultaneously, since this is supposed to be handled by a DRM. A feature to set a ceiling on concurrent processes may be added in the future.

Hint: If you do not have linux installed and want to use this feature, consider installing Ubuntu inside VirtualBox. Cosmos does not support Windows.

3.2 2. Create SQL Tables and Load Static Files

Once you've configured Cosmos, setting up the SQL database tables is easy. The web interface is a Django app, which requires you to run the `collectstatic` command. This moves all the necessary image, css, and javascript files to `~/ .cosmos/static/` directory. Run these two commands after you've configured the database in the cosmos configuration file.

```
$ cosmos syncdb --noinput
```

```
$ cosmos collectstatic
```

If you ever switch to a different database in your `~/ .cosmos/config.ini`, be sure to run `cosmos syncdb` to recreate your tables.

CHAPTER FOUR: Getting Started

We'll start by running a simple test workflow, exploring it via the web interface, and terminating it. Then you'll be ready to start learning how to write your own.

4.1 Execute an Example Workflow

The console will generate a lot of output as the workflow runs. This workflow tests out various features of Cosmos. The number beside each object inside brackets, `[#]`, is the ID of that object.

```
$ cd Cosmos
$ python example_workflows/ex1.py

wrote to /tmp/ex1.svg
INFO: 2012-12-06 16:45:51: Created Workflow Workflow[2] Example 1.
INFO: 2012-12-06 16:45:51: Adding tasks to workflow.
INFO: 2012-12-06 16:45:51: Creating Stage[16] ECHO from scratch.
INFO: 2012-12-06 16:45:51: Creating Stage[17] CAT from scratch.
INFO: 2012-12-06 16:45:51: Total tasks: 6, New tasks being added: 6
INFO: 2012-12-06 16:45:51: Bulk adding 6 TaskFiles...
INFO: 2012-12-06 16:45:51: Bulk adding 6 Tasks...
INFO: 2012-12-06 16:45:51: Bulk adding 10 TaskTags...
INFO: 2012-12-06 16:45:51: Bulk adding 4 task edges...
INFO: 2012-12-06 16:45:51: Generating DAG...
INFO: 2012-12-06 16:45:51: Running DAG.
INFO: 2012-12-06 16:45:51: Running Task[91] ECHO {'word': 'hello'}
INFO: 2012-12-06 16:45:51: Submitted jobAttempt with drmaa jobid 4039.
INFO: 2012-12-06 16:45:52: Running Task[94] ECHO {'word': 'world'}
Job <4040> is submitted to default queue <medium_priority>.
INFO: 2012-12-06 16:45:52: Submitted jobAttempt with drmaa jobid 4040.
INFO: 2012-12-06 16:46:00: Task[91] ECHO {'word': 'hello'} Successful!
INFO: 2012-12-06 16:46:00: Running Task[92] CAT {'i': 1, 'word': 'hello'}
INFO: 2012-12-06 16:46:00: Submitted jobAttempt with drmaa jobid 4041.
INFO: 2012-12-06 16:46:00: Running Task[93] CAT {'i': 2, 'word': 'hello'}
INFO: 2012-12-06 16:46:00: Submitted jobAttempt with drmaa jobid 4042.
INFO: 2012-12-06 16:46:01: Task[94] ECHO {'word': 'world'} Successful!
INFO: 2012-12-06 16:46:01: Stage Stage[16] ECHO successful!
INFO: 2012-12-06 16:46:01: Running Task[95] CAT {'i': 1, 'word': 'world'}
INFO: 2012-12-06 16:46:01: Submitted jobAttempt with drmaa jobid 4043.
INFO: 2012-12-06 16:46:01: Running Task[96] CAT {'i': 2, 'word': 'world'}
INFO: 2012-12-06 16:46:01: Submitted jobAttempt with drmaa jobid 4044.
INFO: 2012-12-06 16:46:12: Task[93] CAT {'i': 2, 'word': 'hello'} Successful!
INFO: 2012-12-06 16:46:12: Task[92] CAT {'i': 1, 'word': 'hello'} Successful!
INFO: 2012-12-06 16:46:13: Task[96] CAT {'i': 2, 'word': 'world'} Successful!
INFO: 2012-12-06 16:46:14: Task[95] CAT {'i': 1, 'word': 'world'} Successful!
INFO: 2012-12-06 16:46:14: Stage Stage[17] CAT successful!
INFO: 2012-12-06 16:46:14: Finished.
```

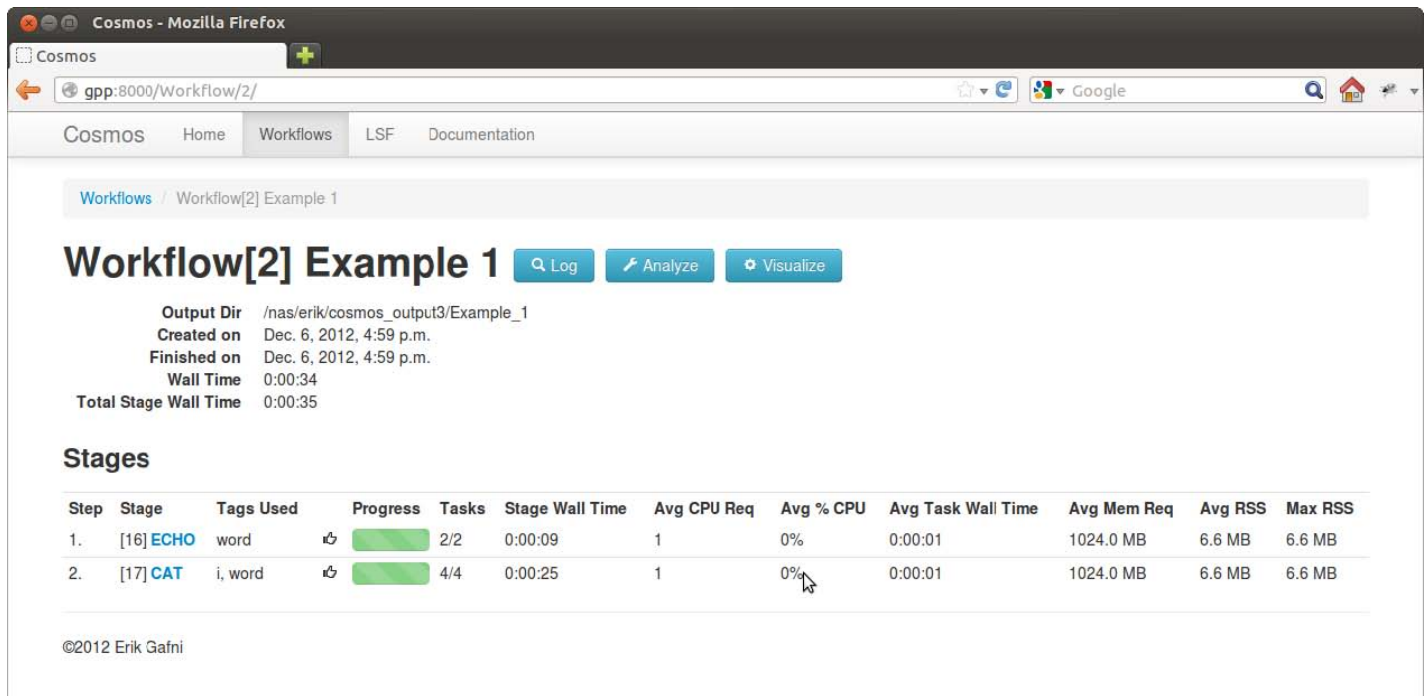
4.2 Launch the Web Interface

You can use the web interface to explore the history and debug all workflows. To start it, run:

```
cosmos runweb -p 8080
```

Note: Currently the system you're running the web interface on must be the same (or have DRMAA access to) as the system you're running the workflow on.

Visit <http://servername:8080> to access it (or <http://localhost:8080>) if you're running cosmos locally.



Workflow[2] Example 1 [Log] [Analyze] [Visualize]

Output Dir: /nas/erik/cosmos_output3/Example_1
Created on: Dec. 6, 2012, 4:59 p.m.
Finished on: Dec. 6, 2012, 4:59 p.m.
Wall Time: 0:00:34
Total Stage Wall Time: 0:00:35

Stages

Step	Stage	Tags Used	Progress	Tasks	Stage Wall Time	Avg CPU Req	Avg % CPU	Avg Task Wall Time	Avg Mem Req	Avg RSS	Max RSS
1.	[16] ECHO	word	<div><div></div></div>	2/2	0:00:09	1	0%	0:00:01	1024.0 MB	6.6 MB	6.6 MB
2.	[17] CAT	i, word	<div><div></div></div>	4/4	0:00:25	1	0%	0:00:01	1024.0 MB	6.6 MB	6.6 MB

©2012 Erik Gafni

Hint: If the cosmos webserver is running, but you can't connect, it is likely because there is a firewall in front of the server. You can get around it by using ssh port forwarding, for example" `$ ssh -L 8080:servername:8080 user@server`

Warning: The webserver is NOT secure. If you need it secured, you'll have to set it up in a production Django web server environment (for example, using mod_wsgi with Apache2).

4.3 Terminating a Workflow

To terminate a workflow, simply press ctrl+c (or send the process a SIGINT signal) in the terminal. Cosmos will terminate running jobs and mark them as failed. You can resume from the point in the workflow you left off later.

4.4 Resuming a workflow

A workflow can be resumed by re-running a script that originally. The algorithm for resuming is as follows:

1. Delete any failed tasks
2. Add any tasks that do not exist in the database (Keyed by the task's stage name and tags)
3. Run the workflow

Warning: If a task in a stage with the same tags has already been executed successfully, it will not be re-executed or altered, even if the actual command has already changed because you modified the script. In the future Cosmos may

emit a warning when this occurs. This can be especially tricky when you try to change task that has no tags (when it's the only task in its stage), and has executed successfully.

CHAPTER FIVE: Writing Workflows with *EZFlow*

The easiest way to write a workflow is to use the *EZFlow* package.

See Also:

There are a lot of useful examples. Some of the more advanced examples even demonstrate useful advanced features that are not described here yet. *Workflow Code Examples*

EZFlow is a package that contains powerful modules for describing workflows. It allows you to define classes that represent a command line tool and various functions to make creating a complex workflow of jobs represented by a *DAG* simple.

A *DAG* consists of Stages, Tools and Tool dependencies.

5.1 Defining Tools

A tool represents an executable (like echo, cat, or paste, or script) that is run from the command line. A tool is a class that overrides `Tool`, and defines `cmd()`, (unless the tool doesn't actually perform an operation, i.e., `tool.Tool.NOOP == True`).

```
from cosmos.contrib.ezflow.tool import Tool

class WordCount(Tool):
    name = "Word Count"
    inputs = ['txt']
    outputs = ['txt']
    mem_req = 1*1024
    cpu_req = 1

    def cmd(self,i,s,p):
        return r"""
            wc {i[txt][0]} > $OUT.txt
            """
```

This tool will read a txt file, count the number of words, and write it to another text file. See the Tool API for more properties that can be overridden to obtain various behaviors.

5.1.1 Overriding map_inputs

To do

5.2 Defining Input Files

An Input file is an instantiation of `tool.INPUT`, which is just a Tool with `tool.INPUT.NOOP` set to `True`, and a way to initialize it with a single output file from an existing path on the filesystem.

An INPUT has one outputfile, which is an instance of `Workflow.models.TaskFile`. It has 3 important attributes:

- **name**: This is the name of the file, and is used as the key for obtaining it. No Tool can have multiple Task-Files with the same name. Defaults to `fmt`.
- **fmt**: The format of the file. Defaults to the extension of `path`.
- **path**: The path to the file. Required.

Here's an example of how to create an instance of an `tool.INPUT` file:

```
from cosmos.contrib.ezflow import INPUT

input_file = INPUT('/path/to/file.txt', tags={'i':1})
```

`input_file` will now be a tool instance with an output file called 'txt' that points to `/path/to/file.txt`.

A more fine grained approach to defining input files:

```
from cosmos.contrib.ezflow import INPUT
INPUT(name='favorite_txt', path='/path/to/favorite_txt.txt.gz', fmt='txt.gz', tags={'color':
'red'})
```

5.3 Designing Workflows

All jobs and job dependencies are represented by the `dag.DAG` class.

There are 5 infix operators you can use to generate a DAG. They each take an instance of a DAG on the left, and apply the `tool.Tool` class on the right to the last `cosmos.Workflow.models.Stage` added to the DAG.

Hint: You can always visualize the DAG that you've built using `dag.DAG.create_dag_img()`. (see Workflow Code Examples for details)

The 5 infix operators are:

|Add|

Always the first operator of a workflow. Simply adds the list of tool instances in `tools` to the dag, without adding any dependencies. This behavior is significantly different from the other infix operators in that the main parameter is a list of instantiated tools, rather than a `ezflow.tool.Tool` class.

Note: This might be renamed to Root in a future release

Parameters

- `tools` – (list of tools) A list of tool instantiated tool instances to add.
- `stage_name` – (str) The name of the stage. Defaults to the `tool_class.name`.

Returns: The modified dag.

```
>>> dag() |Add| [tool1,tool2,tool3,tool4]

>>> dag() |Add| ([tool1,tool2,tool3,tool4], 'My Stage Name')
```

|Map|

Creates a one2one relationships for each tool in the stage last added to the dag, with a new tool of type `tool_class`.

Parameters

- `tool_class` – (subclass of Tool)
- `stage_name` – (str) The name of the stage. Defaults to the `tool_class.name`.

Returns: The modified dag.

```
>>> dag() |Map| Tool_Class
```

|Split|

Creates one2many relationships for each tool in the stage last added to the dag, with every possible combination of keywords in `split_by`. New tools will be of class `tool_class` and tagged with one of the possible keyword combinations.

Parameters

- `tool_class` – (subclass of Tool)
- `split_by` – (list of (str,list)) Tags to split by.
- `stage_name` – (str) The name of the stage. Defaults to the `tool_class.name`.

Returns: The modified dag.

```
>>> dag() |Split|
([('shape', ['square', 'circle']), ('color', ['red', 'blue'])], Tool_Class)

>>> dag() |Split|
([('shape', ['square', 'circle']), ('color', ['red', 'blue'])], Tool_Class, 'My Stage')
```

The above will generate 4 new tools dependent on each tool in the most recent stage. The new tools will be tagged with:

```
{'shape': 'square', 'color': 'red'}, {'shape': 'square', 'color': 'blue'},
{'shape': 'circle', 'color': 'red'}, {'shape': 'circle', 'color': 'blue'}
```

|Reduce|

Creates many2one relationships for each tool in the stage last added to the dag grouped by **keywords**, with a new tool of type `tool_class`.

Parameters

- **keywords** – (list of str) Tags to reduce to. All keywords not listed will not be passed on to the tasks generated.
- **tool_class** – (subclass of Tool)
- **stage_name** – (str) The name of the stage. Defaults to the `tool_class.name`.

Returns: The modified dag.

```
>>> dag() |Reduce| ([ 'shape' , 'color' ], Tool_Class)
```

In the above example, the most recent stage will be grouped into tools with the same shape and color, and a dependent tool of type `tool_class` will be created tagged with the shape and color of their parent group.

|ReduceSplit|

Creates many2one relationships for each tool in the stage last added to the dag grouped by **keywords** and split by the product of **split_by**, with a new tool of type `tool_class`.

Parameters

- **keywords** – (list of str) Tags to reduce to. All keywords not listed will not be passed on to the tasks generated.
- **split_by** – (list of (str,list)) Tags to split by.
- **tool_class** – (subclass of Tool)
- **stage_name** – (str) The name of the stage. Defaults to the `tool_class.name`.

Returns: The modified dag.

```
>>> dag() |ReduceSplit| ([ 'color' , 'shape' ], [ ('size' , ['small' , 'large' ]) ], Tool_Class)
```

The above example will group the last stage into tools with the same color and shape, and create two new dependent tools with tags `{ 'size' : 'large' }` and `{ 'size' : 'small' }`, plus the `color` and `shape` of their parents.

5.4 API

```
class cosmos.contrib.ezflow.tool.Tool(stage_name=None, tags={}, dag=None)
```

A Tool is a class who's instances represent a command that gets executed. It also contains properties which define the resources that are required

Property inputs (list of str) a list of input names. Defaults to []. Can be overridden.

Property outputs (list of str) a list of output names. Defaults to []. Can be overridden.

Property mem_req (int) Number of megabytes of memory to request. Defaults to 1024. Can be overridden.

Property `cpu_req` (int) Number of cores to request. Defaults to 1. Can be overridden.

Property `NOOP` (bool) If True, these tasks do not contain commands that are executed. Used for INPUT. Default is False. Can be overridden.

Property `forward_input` (bool) If True, the input files of this tool will also be input files of children of this tool. Default is False. Can be overridden.

Property `succeed_on_failure` (bool) If True, if this tool's tasks' job attempts fail, the task will still be considered successful. Default is False. Can be overridden.

Property `default_params` (dict) A dictionary of default parameters. Defaults to {}. Can be over-ridden.

Property `stage_name` (str) The name of this Tool's stage. Defaults to the name of the class.

Property `dag` (DAG) The dag that is keeping track of this Tool.

Property `id` (int) A unique identifier. Useful for debugging.

Property `input_files` (list) This Tool's input TaskFiles.

Property `output_files` (list) This Tool's output TaskFiles. A tool's output taskfile names should be unique.

Property `tags` (dict) This Tool's tags.

`get_output`(name, error_if_missing=True)

Returns the output TaskFiles who's name == name. This should always be one element.

Parameters

- name – the name of the output file.
- error_if_missing – (bool) Raises a `GetOutputError` if the output cannot be found

`add_output`(taskfile)

Adds an taskfile to self.output_files.

Parameters

- taskfile – an instance of a `TaskFile`

`input_files`

An alias to :py:meth:map_inputs

`label`

Label used for the DAG image

`map_inputs`()

Default method to map inputs. Can be overridden if a different behavior is desired :returns: (dict) A dictionary of taskfiles which are inputs to this tool. Keys are names of the taskfiles, values are a list of taskfiles.

process_cmd()

Stuff that happens in between map_inputs() and cmd()

cmd(i, s, p)

Constructs the preformatted command string. The string will be .format()ed with the i,s,p dictionaries, and later, `$OUT.outname` will be replaced with a TaskFile associated with the output name outname

Parameters

- i – (dict) Input TaskFiles.
- s – (dict) Settings. The settings dictionary, set by using `contrib.ezflow.dag.configure()`
- p – (dict) Parameters.

Returns: (str|tuple(str,dict)) A preformatted command string, or a tuple of the former and a dict with extra values to use for formatting.

```
class cosmos.contrib.ezflow.tool.INPUT(path, name=None, fmt=None, *args, **kwargs)
```

An Input File. Does not actually execute anything, but provides a way to load an input file.

```
>>> INPUT('/path/to/file.ext', tags={'key': 'val'})
```

```
>>> INPUT(path='/path/to/file.ext.gz', name='ext', fmt='ext.gz', tags={'key': 'val'})
```

```
exception cosmos.contrib.ezflow.dag.DAGError
```

```
exception cosmos.contrib.ezflow.dag.StageNameCollision
```

```
exception cosmos.contrib.ezflow.dag.FlowFxnValidationError
```

```
cosmos.contrib.ezflow.dag.flowfxn(func)
```

- The decorated function should return a generator, so evaluate it
- Set the dag.active_tools to the decorated function's return value, if the function was not sequence_, which handles this automatically
- Return the dag

```
class cosmos.contrib.ezflow.dag.DAG(cpu_req_override=False, ignore_stage_name_collisions=False, mem_req_factor=1)
```

A Representation of a workflow as a DAG of jobs.

```
get_tools_by(stage_names=[], tags={})
```

Parameters

- `stage_names` – (str) Only returns tasks belonging to stages in `stage_names`
- `tags` – (dict) The criteria used to decide which tasks to return. Returns (list) A list of tasks

`add_edge(parent, child)`

Adds a dependency :param parent: (Tool) a parent :param child: (Tool) a child :return: (DAG) self

`branch_from_tools(tools)`

Branches from a list of tools :param tools: (list) a list of tools :return: (DAG) self

`branch_(stage_names=[], tags={})`

Updates `active_tools` to be the tools in the stages with name `stage_name`. The next infix operation will thus be applied to **`stage_name`**. This way the infix operations can be applied to multiple stages if the workflow isn't "linear".

Parameters

- `stage_names` – (str) Only returns tasks belonging to stages in `stage_names`
- `tags` – (dict) The criteria used to decide which tasks to return.

Returns: (list) A list of tasks

`create_dag_img(path)`

Writes the DAG as an image. `path` :param path: the path to write to.

`Configure(settings={}, parameters={})`

Sets the parameters and settings of every tool in the dag.

Parameters

- `parameters` – (dict) { 'stage_name': { 'name': 'value', ... }, { 'stage_name2': { 'key': 'value', ... } } }
- `settings` – (dict) { 'key': 'val' }

`add_to_workflow(workflow)`

Add this dag to a workflow. Only adds tasks to stages that are new, that is, another tag in the same stage with the same tags does not already exist.

Parameters

- `workflow` – the workflow to add

`add_run(workflow, finish=True)`

Shortcut to add to workflow and then run the workflow :param workflow: the workflow this dag will be added to :param finish: pass to workflow.run()

```
add_(tools, stage_name=None, tag={})
```

Always the first operator of a workflow. Simply adds a list of tool instances to the dag, without adding any dependencies.

Warning: This operator is different than the others in that its input is a list of instantiated instances of Tools.

Parameters

- tools – (list) Tool instances.
- stage_name – (str) The name of the stage to add to. Defaults to the name of the tool class.
- tag – (dict) A dictionary of tags to add to the tools produced by this flowfxn Returns (dag) self

```
>>> dag() |Add| [tool1,tool2,tool3,tool4]
```

```
map_(tool_class, stage_name=None, tag={})
```

Creates a one2one relationships for each tool in the stage last added to the dag, with a new tool of type `tool_class`.

Parameters

- tool_class – (subclass of Tool)
- stage_name – (str) The name of the stage to add to. Defaults to the name of the tool class.
- tag – (dict) A dictionary of tags to add to the tools produced by this flowfxn Returns (DAG) self

```
>>> dag.map_(Tool_Class)
```

```
split_(split_by, tool_class, stage_name=None, tag={})
```

Creates one2many relationships for each tool in the stage last added to the dag, with every possible combination of keywords in `split_by`. New tools will be of class `tool_class` and tagged with one of the possible keyword combinations.

Parameters

- split_by – (list of (str,list)) Tags to split by.
- tool_class – (list) Tool instances.
- stage_name – (str) The name of the stage to add to. Defaults to the name of the tool class.
- tag – (dict) A dictionary of tags to add to the tools produced by this flowfxn Returns (DAG) self.

```
>>> dag() |Split|
([('shape', ['square', 'circle']), ('color', ['red', 'blue'])], Tool_Class)
```

```
reduce_(keywords, tool_class, stage_name=None, tag={})
```

Create new tools with a many2one to parent_tools.

Parameters

- keywords – (list of str) Tags to reduce to. All keywords not listed will not be passed on to the tasks generated.
- tool_class – (list) Tool instances.
- stage_name – (str) The name of the stage to add to. Defaults to the name of the tool class.
- tag – (dict) A dictionary of tags to add to the tools produced by this flowfxn Returns (DAG) self

```
>>> dag() |Reduce| (['shape'], Tool_Class)
```

```
reduce_split_(keywords, split_by, tool_class, stage_name=None, tag={})
```

Create new tools by first reducing then splitting.

Parameters

- keywords – (list of str) Tags to reduce to. All keywords not listed will not be passed on to the tasks generated.
- split_by – (list of (str,list)) Tags to split by. Creates every possible product of the tags.
- tool_class – (list) Tool instances.
- stage_name – (str) The name of the stage to add to. Defaults to the name of the tool class.
- tag – (dict) A dictionary of tags to add to the tools produced by this flowfxn Returns (DAG) self

```
>>> dag() |ReduceSplit| (['color', 'shape'], [(size, ['small', 'large'])], Tool_Class)
```

```
apply_(*flowlist, **kwargs)
```

Applies each flowfxn in *flowlist to current dag.active_tools. This is different from **sequence_()**, because **sequence_** applies the flowfns in flowlist to each other sequentially. With **apply_**, all functions in *flowlist are applied onto the current active_tools.

After the **apply_**, dag.active_tools will be the tools added by the last flowfxn in *flowlist.

Parameters

- *flowlist – A sequence of flowfxns
- combine – Combines all tools produced by flowlist and sets the **self.active_tools** to the union of them.

Returns: (DAG) this dag

```
>>> dag.apply_(map_(ToolX), seq_([ reduce_(['a'],ToolA), map_(,ToolB)],
split_(['b', ['2']],ToolC))
```

In the above example, ToolA, ToolB and ToolC will all be applied to the instances generated by ToolX. The next flowfxn will only apply to the tools generated by ToolC.

`sequence_(*flowlist, **kwargs)`

Applies each flowfxn in *flowlist sequentially to each other. Very similar to python's `reduce()` function (not to be confused with `reduce_()`), initialized with the current `active_nodes`.

Parameters

- *flowlist – A sequence of flowfxns
- combine – Combines all tools produced by flowlist and sets the `self.active_tools` to the union of them.

Returns: (DAG) this dag

```
>>> dag.sequence_(map_(ToolX), seq_([ reduce_(['a'],ToolA), map_(,ToolB)],
split_(['b', ['2']],ToolC))
```

In the above example, ToolA will be applied to Toolx, ToolB to ToolA, and ToolC applied to ToolB. ToolC will be set as `dag.active_tools`

`_DAG__new_task(stage, tool)`

Instantiates a task from a tool.

Parameters

- stage – The stage the task should belong to.
- tool – The Tool.

```
class cosmos.contrib.ezflow.dag.MethodStore(*args, **kwargs)
```

```
class cosmos.contrib.ezflow.dag.add_(*args, **kwargs)
```

```
class cosmos.contrib.ezflow.dag.map_(*args, **kwargs)
```

```
class cosmos.contrib.ezflow.dag.split_(*args, **kwargs)
```

```
class cosmos.contrib.ezflow.dag.reduce_(*args, **kwargs)
```

```
class cosmos.contrib.ezflow.dag.reduce_split_(*args, **kwargs)
```

```
class cosmos.contrib.ezflow.dag.branch_(*args, **kwargs)
```

```
class cosmos.contrib.ezflow.dag.sequence_(*args, **kwargs)
```

```
class cosmos.contrib.ezflow.dag.apply_(*args, **kwargs)

class cosmos.contrib.ezflow.dag.configure(*args, **kwargs)

class cosmos.contrib.ezflow.dag.add_run(*args, **kwargs)
```

CHAPTER SIX: Workflow Code Examples

The easiest way to learn is often by example. When you're learning to write your own workflows, make liberal use of the `create_dag_img()` and the `visualize` button in the web interface.

6.1 Basic Workflows

6.1.1 Hello World

Here is the source code of the `example_workflows/ex1.py` you ran in Getting Started.

```
from cosmos.Workflow.models import Workflow from cosmos.contrib.ezflow.dag import
DAG, add_,split_,sequence_ from tools import ECHO, CAT, PASTE, WC

##### # Workflow #####

dag = DAG().sequence_(

add_([ ECHO(tags={'word':'hello'}), ECHO(tags={'word':'world'}) ]),

split_([(('i',[1,2])),CAT) ] dag.create_dag_img('/tmp/ex.svg')

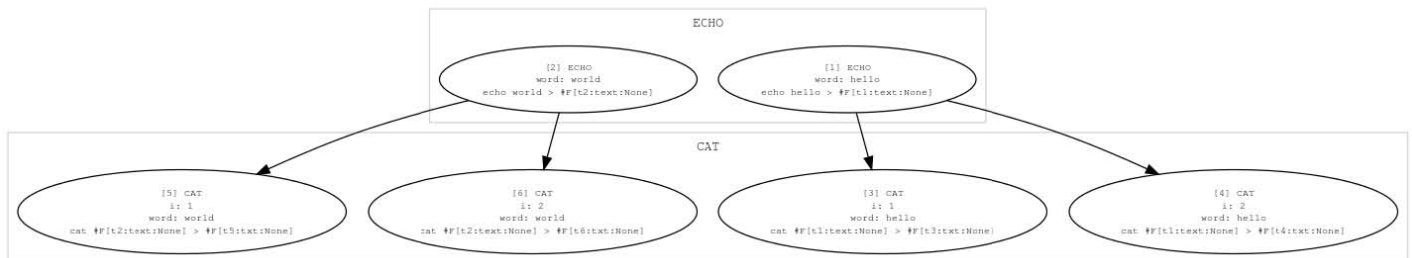
##### # Run Workflow #####

WF = Workflow.start('Example Fail',restart=True) dag.add_to_workflow(WF) WF.run()
```

Here's the job dependency graph that was created:

6.1.2 Reload a Workflow

You can add more stages to the workflow, without re-running tasks that were already successful. An example is in `example_workflows/ex2.py`.



```
from cosmos.Workflow.models import Workflow

from cosmos.contrib.ezflow.dag import DAG, split_,add_,map_,reduce_

from tools import ECHO, CAT, WC, PASTE
```

```
#####

# Workflow

#####

dag = DAG().sequence_(
    add_([ ECHO(tags={'word':'hello'}), ECHO(tags={'word':'world'}) ]),
    split_([('i',[1,2])], CAT),
    reduce_([], PASTE),
    map_(WC)
)

dag.create_dag_img('/tmp/ex.svg')

#####

# Run Workflow

#####

WF = Workflow.start('Example 2',restart=True,delete_intermediates=True)

dag.add_to_workflow(WF)

WF.run()
```

Run it with the command:

```
$ python ex1_b.py
```

6.2 Advanced Workflows

6.2.1 Branching Workflows

```
example_workflows/ex_branch.py

"""
```


This workflow demonstrates branching for when you need something more complicated than a linear step-by-step series of stages.

`cosmos.contrib.ezflow.dag.DAG.branch()` is the key to branching.

```
"""
```

```
from cosmos.Workflow.models import Workflow
```

```
from cosmos.contrib.ezflow.dag import DAG, Map, Split, Add
```

```
import tools
```

```
#####
```

```
# Workflow
```

```
#####
```

```
dag = ( DAG()
```

```
    .add([ tools.ECHO(tags={'word':'hello'}),
tools.ECHO(tags={'word':'world'}) ])
```

```
    .split([( 'i' , [1,2]) ],tools.CAT)
```

```
    .map(tools.WC)
```

```
    .branch('ECHO')
```

```
    .map(tools.WC,'Extra Independent Word Count')
```

```
)
```

```
# Generate image
```

```
dag.create_dag_img('/tmp/ex_branch.svg')
```

```
#####
```

```
# Run Workflow

#####

WF = Workflow.start('Example Branch',restart=True)

dag.add_to_workflow(WF)

WF.run()
```

6.2.2 Command Line Interface and Signals

```
example_workflows/ex_branch.py
```

```
"""
```

```
This shows how to use signals to run arbitrary
code when a stage fails. One could easily
modify this to send SMS texts instead of e-mails.
```

```
.. note::
```

```
    Signals for status changes are not
    triggered when they occur due to a
    workflow starting or terminating.
```

```
"""
```

```
#####
```

```
# CLI
```

```
#####
```

```
from cosmos.Workflow.cli import CLI
```

```
cli = CLI()
```

```

cli.parser.add_argument('-e', '--email', type=str, help='Email address to report
messages to.',

                        required=True)

WF = cli.parse_args() # parses command line arguments

email = cli.parsed_kwargs['email']

#####

# Signals

#####

from cosmos.Workflow import signals

from django.dispatch import receiver

import smtplib

# Send an e-mail each time the workflow fails

# For more details on using signals,

# See the `Django Signals Documentation
<https://docs.djangoproject.com/en/dev/topics/signals/>`_

#

# .. note:: I have not tested the actual e-mailing code, but the signal does work.

#

@receiver(signals.stage_status_change)

def email_on_fail(sender, status, **kwargs):

    stage = sender

    WF.log.warning('Notifying {0} of stage failure.'.format(email))

    if status == 'failed':

        SUBJECT = "{0} Failed!".format(stage)

        TO = email

        FROM = "me@me.com"

```

```

HOST = "smtp.server"

text = '{0} has failed at stage {1}'.format(stage.workflow, stage)

BODY = "\r\n".join(
    "From: %s" % FROM,
    "To: %s" % TO,
    "Subject: %s" % SUBJECT,
    "",
    text
)

server = smtplib.SMTP(HOST)

server.sendmail(FROM, [TO], BODY)

server.quit()

#####

# Workflow

#####

from cosmos.contrib.ezflow.dag import DAG, Map, Split, Add

import tools

dag = ( DAG()

    |Add| [tools.ECHO(tags={'word': 'hello'}), tools.ECHO(tags={'word':
'world'})]

    |Map| tools.FAIL # Automatically fail

)

#####

# Run Workflow

```

```
#####
```

```
dag.add_to_workflow(WF)
```

```
WF.run()
```

CHAPTER SEVEN: Command Line Interface

Make sure your environment variables are properly set (see Configuration). Use the shell command `env` if you're not sure what's in your environment.

```
$ cosmos -h
usage: cosmos [-h] <command> ...
```

Cosmos CLI

optional arguments:
-h, --help show this help message and exit

Commands:

<command>	
resetdb	DELETE ALL DATA in the database and then run a syncdb
shell	Open up an ipython shell with Cosmos objects preloaded
syncdb	Sets up the SQL database
list	List all workflows
runweb	Start the webserver

Explore the available commands, using `-h` if you wish. Or see the Command Line Interface for more info. Note that when listing workflows, the number beside each Workflow inside brackets, `[#]`, is the ID of that object.

7.1 Examples

7.1.1 Get Usage Help:

```
$ cosmos -h
```

7.1.2 Reset the SQL database

Warning: This will not delete the files associated with workflow output.

```
$ cosmos resetdb
```

7.1.3 List workflows

```
$ cosmos ls
```

CHAPTER EIGHT: Cosmos Shell

Using various Django features and apps, you can quickly enter an ipython shell with access to all your workflow objects.

Note: This is an advanced feature, and relies on Django Queries

8.1 Launch the IPython shell

```
$ cosmos shell
```

You can then interactively investigate and perform all sorts of operations. This is really powerful when interacting with the Django API, since most of the Cosmos objects are Django models. Most Cosmos classes are automatically imported for you.

```
all_workflows = Workflow.objects.all()

workflow = all_workflows[2]

stage = workflow.stages[3]

stage.file_size

stage.tasks[3].get_successful_jobAttempt().queue_status
```

8.2 Interactive Workflow

You can even run a workflow:

```
wf = Workflow.start('Interactive')

stage = wf.add_stage('My Stage')

task = stage.add_task('echo "hi"')

wf.run()

wf.finished()
```

8.3 Filtering

pk is a shortcut for primary key, so you can use this to quickly get objects you’re seeing in the web interface, which is always displayed. For example, if you see “Stage[200] Stage Name”, you can query for it like so

```
Workflow.objects.get(pk=200)
```

Or for multiple objects:

```
Workflow.objects.filter(pk__in=[200,201,300])
```

You can also query on the many fields available for each object:

```
Task.objects.filter(status='in_progress',memory_requirement=1024)
```

For filtering by tags, use the special method `cosmos.Workflow.models.Workflow.get_task_by()`.

```
wf = Workflow.objects.get(name="My Workflow")
```

```
wf.get_task_by(tags={'color':'orange','shape':'circle'})
```

For more advanced queries, see Django Queries.

8.4 Deleting

You can delete records by simply calling `object.delete()`

Warning: Do not call `.delete()` on a queryset, as it will not run a lot of important cleanup code. i.e. don't do this:

```
>>> Task.objects.get(success=False).delete()
```

or

```
>>> Stage.objects.get(name="My Stage").delete()
```

Instead, do the following, which will perform a lot of extra important code for each task:

```
>>> for t in Task.objects.get(success=False): t.delete()
```

or

```
>>> for s in Stage.objects.get(name="My Stage"): s.delete()
```


CHAPTER NINE : Advanced Usage

9.1 Setting Custom Job Submission Flags

If you want to specify custom DRMS specific flags, all you have to do is set `cosmos.session.get_drmaa_native_specification` in your workflow script.

Hint: By default, cosmos uses `cosmos.session.default_get_drmaa_native_specification()` and you'll probably want to take a look at its source code.

For example, to submit to a queue depending on the task's time_requirement:

```
from cosmos import session

def my_get_drmaa_native_specification(jobAttempt):

    task = jobAttempt.task

    DRM = settings['DRM']

    cpu_req = task.cpu_requirement

    mem_req = task.memory_requirement

    time_req = task.time_requirement

    queue = task.workflow.default_queue

    if time_req < 10:

        queue = 'mini'

    if time_req < 12*60:

        queue = 'short'

    else:

        queue = 'i2b2_unlimited'

    if DRM == 'LSF':

        s = '-R "rusage[mem={0}] span[hosts=1]" -n {1}'.format(mem_req,cpu_req)
```

```

    if time_req:

        s += ' -W 0:{0}'.format(time_req)

    if queue:

        s += ' -q {0}'.format(queue)

    return s

else:

    raise Exception('DRM not supported')

session.get_drmaa_native_specification = get_drmaa_native_specification

```

9.1.1 API

Session

A Cosmos session. Must be the first import of any cosmos script.

```
cosmos.session.default_get_drmaa_native_specification(jobAttempt)
```

Default method for the DRM specific resource usage flags passed in the jobtemplate's dr-maa_native_specification. Can be overridden by the user when starting a workflow.

Parameters

- jobAttempt – the jobAttempt being submitted

```
cosmos.session.get_drmaa_native_specification(jobAttempt)
```

The method to produce a cosmos.Job.models.JobAttempt's extra submission flags. Can be overridden by user if special behavior is desired.

CHAPTER TEN: Cosmos FAQ

This is a list of Frequently Asked Questions about Cosmos. Feel free to suggest new entries!

- Why is my job failing with Execution Halted?

You probably didn't request enough resources. If Drmaa returns **WasAborted=True**, then that's generally the case.

CHAPTER ELEVEN: API

11.1 Workflow

11.1.1 Primary Workflow Objects

11.1.2 API

Workflow models

```
class cosmos.Workflow.models.TaskFile(*args, **kwargs)
```

Task File

task

The task this TaskFile is an output for .

file_size

Size of the taskfile's output_dir

delete_because_intermediate()

Deletes this file and marks it as deleted because it is an intermediate file.

```
class cosmos.Workflow.models.Workflow(*args, **kwargs)
```

This is the master object. It contains a list of Stage which represent a pool of jobs that have no dependencies on each other and can be executed at the same time.

tasks

Tasks in this Workflow.

task_edges

Edges in this Workflow.

task_tags

TaskTags in this Workflow.

task_files

TaskFiles in this Stage.

wall_time

Time between this workflow's creation and finished datetimes. Note, this is a `timedelta` instance, not seconds

stages

Stages in this Workflow.

file_size

Size of the output directory.

log_txt

Path to the logfile.

static start(name, **kwargs)

Starts a workflow. If a workflow with this name already exists, return the workflow.

Parameters

- **name** – (str) A unique name for this workflow. All spaces are converted to underscores. Required.
- **restart** – (bool) Complete restart the workflow by deleting it and creating a new one. Optional.
- **dry_run** – (bool) Don't actually execute jobs. Optional.
- **root_output_dir** – (bool) Replaces the directory used in settings as the workflow output directory. If None, will use `default_root_output_dir` in the config file. Optional.
- **default_queue** – (str) Name of the default queue to submit jobs to. Optional.
- **delete_intermediates** – (str) Deletes intermediate files to save scratch space.

add_stage(name)

Adds a stage to this workflow. If a stage with this name (in this Workflow) already exists, and it hasn't been added in this session yet, return the existing one.

Parameters

- **name** – (str) The name of the stage, must be unique within this Workflow. Required.

terminate()

Terminates this workflow and Exits.

get_all_tag_keys_used()

Returns a set of all the keyword tags used on any task in this workflow.

save_resource_usage_as_csv(filename)

Save resource usage to filename.

yield_stage_resource_usage()

Yields - A dict of all resource usage, tags, and the name of the stage of every task.

bulk_save_tasks(*args, **kwargs)

Does a bulk insert of tasks. Identical tasks should not be in the database.

Parameters

- **tasks** – (list) a list of tasks

Note: this does not save task->taskfile relationships

```
>>> tasks =  
[stage.new_task(pcmd='cmd1', save=False, {'i':1}), stage.new_task(pcmd='cmd2',  
save=False, {'i':2})]  
>>> stage.bulk_save_tasks(tasks)
```

bulk_save_taskfiles(*args, **kwargs)

Parameters

- **taskfiles** – (list) A list of taskfiles.

bulk_save_task_edges(*args, **kwargs)

Parameters

- **edges** – [(parent, child),...] A list of tuples of parent -> child relationships.

bulk_delete_tasks(*args, **kwargs)

Bulk deletes tasks and their related objects.

delete(*args, **kwargs)

Deletes this workflow.

run(terminate_on_fail=True, finish=True)

Runs a workflow using the DAG of jobs.

Parameters

terminate_on_fail – (bool) If True, the workflow will self terminate if any of the tasks of this stage fail max_job_attempts times.

finished()

Call at the end of every workflow.

get_tasks_by(stage=None, tags={}, op='and')

Returns the list of tasks that are tagged by the keys and vals in tags dictionary.

Parameters

- **op** – (str) either 'and' or 'or' as the logic to filter tags with
- **tags** – (dict) tags to filter for

Returns (queryset) a queryset of the filtered tasks

```
>>> task.get_tools_by(op='or', tags={'color': 'grey', 'color': 'orange'})
>>> task.get_tools_by(op='and', tags={'color': 'grey', 'shape': 'square'})
```

get_task_by(tags={}, stage=None, op='and')

Returns the list of tasks that are tagged by the keys and vals in tags dictionary.

Raises Exception if more or less than one task is returned

Parameters

- **op** – (str) Choose either 'and' or 'or' as the logic to filter tags with
- **tags** – (dict) A dictionary of tags you'd like to filter for Returns (queryset) a queryset of the filtered tasks

```
>>> task.get_task_by(op='or', tags={'color': 'grey', 'color': 'orange'})
>>> task.get_task_by(op='and', tags={'color': 'grey', 'color': 'orange'})
```

class cosmos.Workflow.models.**Stage**(*args, **kwargs)

A group of jobs that can be run independently. See Embarrassingly Parallel.

Note: A Stage should not be directly instantiated, use Workflow.models.Workflow.add_stage() to create a new stage.

set_status(new_status, save=True)

Set Stage status

percent_done

Percent of tasks that have completed

get_sjob_stat(field, statistic)

Aggregates a task successful job's field using a statistic. :param field: (str) name of a tasks's field. ex: wall_time or avg_rss_mem :param statistic: (str) choose from ['Avg', 'Sum', 'Max', 'Min', 'Count']

```
>>> stage.get_stat('wall_time','Avg')
120
```

get_task_stat(field, statistic)

Aggregates a task's field using a statistic :param field: (str) name of a tasks's field. ex: cpu_req, mem_req
:param statistic: (str) choose from ['Avg','Sum','Max','Min','Count']

```
>>> stage.get_stat('cpu_requirement','Avg')
120
```

file_size

Size of the stage's output_dir

wall_time

Time between this stage's creation and finished datetimes. Note, this is a timedelta instance, not seconds

output_dir

Absolute path to this stage's output_dir

tasks

Queryset of this stage's tasks

task_edges

Edges in this Stage

task_tags

TaskTags in this Stage

task_files

TaskFiles in this Stage

num_tasks

The number of tasks in this stage

num_tasks_successful

Number of successful tasks in this stage

get_all_tag_keys_used()

Returns a set of all the keyword tags used on any task in this stage

yield_task_resource_usage()

Yields (list of tuples) tuples contain resource usage and tags of all tasks. The first element is the name, the second is the value.

is_done()

Returns True if this stage is finished successfully or failed, else False

get_tasks_by(tags={}, op='and')

An alias for `Workflow.get_tasks_by()` with `stage=self`

Returns a queryset of filtered tasks

get_task_by(tags={}, op='and')

An alias for `Workflow.get_task_by()` with `stage=self`

Returns a queryset of filtered tasks

group_tasks_by(keys=[])

Yields tasks, grouped by tags in keys. Groups will be every unique set of possible values of tags. For example, if you had tasks tagged by color, and shape, and you ran `func:stage.group_tasks_by(['color','shape'])`, this function would yield the group of tasks that exist in the various combinations of 'colors and shapes. So for example one of the yields might be `(({'color':'orange','shape':'circle'}), [orange_circular_tasks])`

Parameters

- keys – The keys of the tags you want to group by.

Yields (a dictionary of this group's unique tags, tasks in this group).

Note: a missing tag is considered as None and thus placed into a 'None' group with other untagged tasks. You should generally try to avoid this scenario and have all tasks tagged by the keywords you're grouping by.

delete(*args, **kwargs)

Bulk deletes this stage and all files associated with it.

url(*args, **kwargs)

The URL of this stage

class `cosmos.Workflow.models.TaskTag`(*args, **kwargs)

A SQL row that duplicates the information of `Task.tags` that can be used for filtering, etc.

class `cosmos.Workflow.models.TaskEdge`(*args, **kwargs)

`TaskEdge(id, parent_id, child_id)`

child

The keys associated with the relationship. ex, the group_by parameter of a many2one

class cosmos.Workflow.models.Task(*args, **kwargs)

The object that represents the command line that gets executed.

tags must be unique for all tasks in the same stage

static create(stage, pcmd, **kwargs)

Creates a task.

workflow

This task's workflow

parents

This task's parents

log

This task's workflow's log

file_size

Task filesize

output_file_size

Task filesize

output_dir

Task output dir

job_output_dir

Where the job output goes

jobAttempts

Queryset of this task's jobAttempts.

wall_time

Task's wall_time

numAttempts()

This task's number of job attempts.

get_successful_jobAttempt()

Get this task's successful job attempt.

Returns this task's successful job attempt. If there were no successful job attempts, returns None

set_status(new_status, save=True)

Set Task's status

tag(kwargs)**

Tag this task with key value pairs. If the key already exists, its value will be overwritten.

```
>>> task.tag(color="blue", shape="circle")
```

clear_job_output_dir()

Removes all files in this task's output directory

url(*args, **kwargs)

This task's url.

tags_as_query_string()

Returns a string of tag keys and values as a url query string

delete(*args, **kwargs)

Deletes this task and all files associated with it

11.2 JobManager

class cosmos.Job.models.JobAttempt(*args, **kwargs)

An attempt at running a task.

STDERR_filepath

Returns the path to the STDERR file

STDERR_txt

The contents of the STDERR file, or the string 'File does not exist.'

STDOUT_filepath

Returns the path to the STDOUT file

STDOUT_txt

The contents of the STDOUT file, or the string 'File does not exist.'

get_command_shell_script_text()

Return the contents of the command.sh file

static profile_fields_as_list()

Returns [profile_fields], a simple list of profile_field names, without their type information

profile_output_path

Path to store a job's profile.py output

resource_usage

Returns (name,value,help,type)

resource_usage_short

Returns (name,value)

update_from_profile_output()

Updates the resource usage from profile output

workflow

This jobattempt's workflow

Glossary

LSF - Platform LSF is a commercial DRMS.

SGE - Sun Grid Engine is a commercial DRMS.

GE - Grid Engine is an open source version of SGE.

DRMS - Distributed Resource Management System. This is the underlying queuing software that manages jobs on a cluster. Examples include LSF, and SGE.

DRMAA - Distributed Resource Management Application API. A standard library that is an abstraction built on top of DRMS so that the same application code can seamlessly run on any DRMS that supports DRMAA.

DAG - Directed Acyclic Graph. A DAG is used by Cosmos to describe a workflow's jobs and their dependencies on each other.

Django - A Python web framework that much of Cosmos is built on.