pylighthouse Documentation

Release 0.2.0

the pylighthouse Authors, see the AUTHORS.md file

Dec 27, 2018

Contents:

Star pylighthouse on Github: Fork pylighthouse on Github:

CHAPTER 1

pylighthouse

Helps workloads find safe harbor.

- Free software: Apache Software License 2.0
- Documentation: https://pylighthouse.readthedocs.io.

1.1 Features

- Scheduling-as-a-library, in pure python
- Schedule workloads onto nodes
- Flexible definition of requirements needed by workloads and resources offered by nodes
- Tag nodes simply by adding a zero-quantity resource
- "Taints and Tolerations"-like behavior supported through the use of Wards and Immunitites
- Anti-affinity-group-like behavior supported through the use of Aversion Groups

1.2 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Installation

2.1 Stable release

To install pylighthouse, run this command in your terminal:

```
$ pip install pylighthouse
```

This is the preferred method to install pylighthouse, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for pylighthouse can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/djhaskin987/pylighthouse
```

Or download the tarball:

```
$ curl  -OL https://github.com/djhaskin987/pylighthouse/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use pylighthouse in a project:

```
import pylighthouse.pylighthouse as lighthouse
```

3.1 Basic Scheduling

You can schedule workloads onto nodes like this:

```
import pylighthouse.pylighthouse as lighthouse

distor = lighthouse.PrioritizedDistributor.from_list(
    nodes=lighthouse.Node.from_list([{
        "name": "cluster-member-1",
        "resources": {
            "cpu": 6,
            "mem": 12,
            "disk": 50
        }
    },
    {
        "name": "cluster-member-2",
        "resources": {
            "cpu": 4,
            "mem": 16,
            "disk": 40
        }
    },
    {
      "name": "cluster-member-3",
      "resources": {
          "cpu": 0.7,
          "mem": 1.3,
```

(continues on next page)

(continued from previous page)

```
            "disk": 17
        }
    }
]))
distor.attempt_assign_loads(lighthouse.Workload.from_list([
    {
        "name": "vm-1",
        "requirements": {
            "cpu": 0.2,
            "mem": 0.1
        }
    },
    {
        "name": "vm-2",
        "requirements": {
            "cpu": 0.3,
            "mem": 0.3,
            "disk": 1
        }
    }]))
# =>
#{
#    "vm-1": "cluster-member-1",
#    "vm-2": "cluster-member-1"
#}
```

As you can see, `attempt_assign_loads` takes a list of workloads and attempts to assign workloads to the nodes given to the distributor at construction time. It returns a dictionary with keys being the names of the workloads and values being the names of the nodes to which those loads were assigned. If workload could not be assigned to a node, the value is `None` for that key instead.

Caution: The `name` field for each workload and node must be unique to that node or workload, or bad things will happen to innocent people (you. At least, I *hope* you're innocent :P).

Node resources and Workload requirements are free-form and can be arbitrary.

Note that the requirements in a workload need not include all the types of resources found in nodes. In the above example, each node has `mem`, `cpu` and `disk` attributes, but the requirements need not list all of these as requirements.

Placement of workloads onto nodes is not guaranteed. That is, simply because room exists for all workloads, this does not mean that pylighthouse will be able to figure this out. You can help pylighthouse get better at packing nodes tightly using the *BinPackDistributor* discussed below, and you can also increase the capacity of the nodes.

Distributors and the nodes they contain are stateful. They remember workloads previously given. So after this code:

```
parent = lighthouse.Node.from_list([
    "name": "parent",
    "resources": {
        "patience": 1
    }
])
a = lighthouse.Workload.from_dict({
    "name": "kid-a",
    "requirements": {
        "patience": 1
```

(continues on next page)

(continued from previous page)

```
    }
})
b = lighthouse.Workload.from_dict({
    "name": "kid-b",
    "requirements": {
        "patience": 1
    }
})
pr = lighthouse.PrioritizedDistributor.from_list([parent])
result1 = pr.attempt_assign_loads([a])
# =>
#{
#    "kid-a": "parent"
#}
```

Running this code afterwards:

```
result2 = pr.attempt_assign_loads([b])
```

Would result in this assignment:

```
{
    "kid-b": None
}
```

This reflects that there is no current room for the second workload, as the first has consumed all resources.

3.2 Placement Strategies

pylighthouse comes with several different distributor classes, all of which place workloads onto nodes. `PrioritizedDistributor` is the simplest, but may not offer the best fit of loads onto nodes. `RoundRobinDistributor` is also offered as a simple way to distribute workloads semi-evenly across a cluster of nodes. In general, `BinPackDistributor` will attempt to pack as many workloads as possible onto as few nodes as possible and is, in general, recommended.

The following code will be referred to when discussing each of the placement strategies below:

```
import pylighthouse.pylighthouse as lighthouse

nodes=lighthouse.Node.from_list([
    {
      "name": "node-1",
      "resources": {
        "cpu": 2,
        "mem": 8,
        "disk": 60
      }
    },
    {
      "name": "node-2",
      "resources": {
        "cpu": 6,
        "mem": 6,
        "disk": 20
```

(continues on next page)

(continued from previous page)

```
      }
    },
    {
      "name": "node-3",
      "resources": {
        "cpu": 4,
        "mem": 2,
        "disk": 40
      }
    }
])
workloads = lighthouse.Workload.from_list([
    {
      "name": "req-1",
      "requirements": {
        "cpu": 8,
        "mem": 8,
        "disk": 80
      }
    },
    {
      "name": "req-2",
      "requirements": {
        "cpu": 8,
        "mem": 8,
        "disk": 80
      }
    },
    {
      "name": "req-3",
      "requirements": {
        "cpu": 8,
        "mem": 8,
        "disk": 60
      }
    }
])
```

3.2.1 Prioritized

With a `PrioritizedDistributor`, pylighthouse will attempt to assign workloads to nodes in the order they appear in the given list of nodes, and in the order the workloads appear.

This is the result if the above were run with `PrioritizedDistributor`:

```
distor = lighthouse.PrioritizedDistributor.from_list(nodes)
distor.attempt_assign_loads(workloads)
# =>
#{
#    "req-1": "node-1",
#    "req-3": "node-1",
#    "req-2": "node-1"
#}
```

In this example, all nodes are assigned to `node-1` because they can all fit on `node-1` and it appears first in the list of nodes given, so it is tried first every time when loads are assigned to nodes.

3.2.2 RoundRobin

With a `RoundRobinDistributor`, assignment of workloads is done in the order given in the list, but placement attempts for each successive load starts on the node just after the successful placement of the previous load – in a "round robin" fashion.

This is the result if the above were run with `RoundRobinDistributor`:: `RoundRobin`:

```
distor = lighthouse.RoundRobinDistributor.from_list(nodes)
distor.attempt_assign_loads(workloads)
# =>
#{
#     "req-1": "node-1",
#     "req-3": "node-3",
#     "req-2": "node-2"
#}
```

3.2.3 BinPack

This strategy requires additional information. A *rubric* must be specified. In discussing the example above, we will assume in our discussion that the following code is also part of the script we are building:

```
rubric_dict = {
    "cpu": 1,
    "mem": 0.5,
    "disk": 0.025
}
```

`BinPackDistributor` attempts to pack in as many requirements into as few nodes as possible. In order to do so, the caller must specify a rubric. This gives quantities that will be used to score each workload and node by multiplying each quantity for a given node or workload and summing the results. If a quantity isn't in the rubric but is in a node's resources or a load's requirements, the quantity won't count towards the score. if a quantity is in the rubric but isn't in a node's resources or a load's requirements, the score will be computed as if the quantity was `0`.

The score of any given node or workload semantically corresponds to the node or load's "size". Therefore, as long as the quantities in nodes and loads that are scored via the rubric are positive, it is recommended to always specify positive quantities in the rubric as well.

> **Caution:** Specifying negative quantities in the rubric is possible, but should be rare, and should be intended only to multiply against a requirement or resource which will also *always* be negative, such as those discussed below under *Wards and Immunities*. If this rule is not followed, `BinPackDistributor` may misbehave. As a rule, if the value is expected to be negative, don't include it in the rubric.

If `BinPackDistributor` was used in the above example, the result would look like this:

```
distor = lighthouse.RoundRobinDistributor.from_list(rubric_dict, nodes)
distor.attempt_assign_loads(workloads)
# =>
#{
#     "req-1": "node-3",
#     "req-3": "node-3",
#     "req-2": "node-3"
#}
```

In this example, all workloads were assigned to `node-3`, since `node-3` had the least room in it going into scheduling, since it had the least disk space.

`BinPackDistributor` first attempts to place workloads by score, but if two workloads share the same score, BinPackDistributor will try to place the workload in sorted order ascending by name of the nodes. So a node named "a" will be tried before a node named "b" if both nodes share the same score.

3.3 Placement Enforcement

At the time of placement of a workload onto a node, the requirements are subtracted from the node's resources so as to keep track of what nodes still have room left for more assignments. In particular, all attributes associated with the *node* must register with a quantity at or above zero in order for the assignment to succeed at *assignment time*.

This allows for some interesting possibilities for how to enforce where workloads can be assigned in your cluster of nodes.

3.3.1 Node Tagging

Sometimes it is desirable to mark a particular node as specifically dedicated to a particular type of workload. When this is desired, it is simply a matter of adding a resource to a node with zero as the quantity:

```
nodes = lighthouse.Nodes.from_list([
    {
        "name": "node1",
        "resources": {
           "dedicated": 0.0,
           #...
        }
    }
])
```

Then, simply place a similar attribute in the requirements dictionary of the workloads that should be run on the dedicated nodes:

```
workloads = lighthouse.Workloads.from_list([
    {
        "name": "workload1",
        "requirements": {
            "dedicated": 0.0,
            #...
        }
    }
])
```

This works because all requirements listed for a workload must be present on the node and none may be allowed to be below zero, but zero is okay.

For example:

```
nodes = lighthouse.Node.from_list([
    {
        "name": "phillip",
        "resources": {
            "bravery": 25,
            "kindness": 25
```

(continues on next page)

(continued from previous page)

```
        }
    },
    {
        "name": "charming",
        "resources": {
            "bravery": 25,
            "kindness": 25,
            "nice-castle": 0,
        }
    }
])
workloads = lighthouse.Workload.from_list([
    {
        "name": "snow-white",
        "requirements": {
            "nice-castle": 0,
        }
    }])
```

Any distributor attempting to assign these workloads to the nodes via `attempt_assign_loads` will yield the following assignment:

```
{
    "snow-white": "charming"
}
```

This is because prince `charming` has the `nice-castle` "tag", while `phillip` does not.

Tags also ensure that no assignment will be made if tags are not present:

```
no_room = lighthouse.Node.from_list([
    {
        "name": "phillip",
        "resources": {
            "bravery": 25,
            "kindness": 25
        }
    },
    {
        "name": "charming",
        "resources": {
            "bravery": 25,
            "kindness": 25
        }
    }
])
```

Any distributor attempting to assign these workloads to the nodes via `attempt_assign_loads` will yield the following assignment:

```
{
    "snow-white": None
}
```

This is because none of the princes (nodes) had a `nice-castle` "tag" present in their resources.

3.3.2 Semaphores

Often it is convenient to limit how many of a particular type of workload is allowed to be placed on a node. This is done simply by listing a resource in a node's resource map and in relevant workload's requirements maps. The pattern is to list the number of workloads a node can handle at the same time in the semaphore as the number for the resource in the node, and list `1` as the quantity for the requirement for each workload. For example:

```
nodes = lighthouse.Node.from_list([
    {
        "name": "prince",
        "resources": {
            "bravery": 25,
            "kindness": 25,
            "nice-castle": 0,
            "wife": 1
        }
    }
])
workloads = lighthouse.Workload.from_list([
    {
        "name": "aurora",
        "requirements": {
            "bravery": 12,
            "nice-castle": 0,
            "wife": 1,
        }
    },
    {
        "name": "buttercup",
        "requirements": {
            "bravery": 12,
            "nice-castle": 0,
            "wife": 1,
        }
    },
    {
        "name": "cinderella",
        "requirements": {
            "bravery": 12,
            "nice-castle": 0,
            "wife": 1,
        }
    }
])
```

In this example, the node is a potential suitor for a number of fairy tale princesses. The prince can only have a single wife, and so `wife` is listed as a resource with quantity `1`. This is the semaphore. Any distributor based off of those nodes will yield the same results as assignments if `attempt_assign_loads` is called:

```
{
    "aurora": "prince",
    "buttercup": None,
    "cinderella": None
}
```

The `PrioritizedDistributor` and `RoundRobinDistributor` will both schedule the first given princess in the list, `aurora`, but will not be able to schedule the remaining princesses. `BinPackDistributor` will likewise schedule `aurora` first because the scores of the workloads based on any reasonable (non-negative) rubric will show

that they have the same sizes of requirements, and `aurora` sorts before the other names.

3.3.3 Wards and Immunities

This concept is similar to Kubernetes' Taints and Tolerations idea, but also has nuances to it that make it more flexible.

The idea is to mark a particular set of nodes as unavailable for workloads unless those workloads specifically opt into being run on those nodes.

We do this in pylighthouse using Wards and Immunities.

It is perfectly valid to list negative values for resources at *node construction time*; however, as has been previously explained, if there are any resources in a node with negative quantity at *assignment time of a workload*, the workload will not be able to be attached to the node.

A negative resource with a finite quantity is called a *shortcoming*, while a negative resource of infinite or very large quantity may be termed a *ward*.

Negative resources can be overcome by a resource in one of two ways.

First, for negative resources of *finite* quantity, this can be overcome by simply listing a negative requirement. That way, when one is subtracted from the other, the result will be zero:

```
nodes = lighthouse.Node.from_list([
    {
        "id: "node1",
        "resources": {
           "flies": -5.0,
           #...
        }
    }
])
workloads = lighthouse.Workload.from_list([
    {
        "name": "workload1",
        "requirements": {
            "flies": -5.0,
            #...
        }
    }
])
```

This may be used to list "shortcomings" of a node that precludes it from having workloads scheduled on it unless at least one workload has a sufficient tolerance to the shortcoming.

Second, we list a node up front at construction time with a ward:

```
nodes = lighthouse.Node.from_list([
    {
        "name": "node1",
        "resources": {
           "spiders": -float("inf")
           #...
        }
    }
]
```

In this scenario, workloads will not be able to overcome the ward no matter how finitely resilient the workload is. However, we can list an immunity on the workload.

An *immunity* in a workload tells pylighthouse to ignore whatever value exists for a resource in a node at assignment time of the workload. So, in order to schedule a workload on the node listed above, we can simply add `"spiders"` to the set of immunities for the workload:

```
workloads = lighthouse.Workload.from_list([
    {
        "name": "workload1",
        "requirements": {
            #...
        },
        "immunities": set([
            "spiders",
            #...
        ])
    }
])
```

3.4 Aversion Groups

Aversion Groups correspond to anti-affinity groups in other scheduling schemes.

Put simply, any aversion group listed for a workload causes that workload to "prefer" to be scheduled on a node without any other workloads listed as "belonging" to the same aversion group, like this::

```
# ...
nodes = lighthouse.Node.from_list([
    {
        "name": "node1",
        "resources": {
            # ...
        }
    },
    {
        "name": "node2",
        "resources": {
            # ...
        }
    }

])
workloads = lighthouse.Workload.from_list([
    {
        "name": "workload1",
        "requirements": {
            # ...
        },
        "aversion_groups": set([
            "io-bound",
            # ...
        ])
    },
    {
        "name": "workload2",
        "requirements": {
            # ...
        },
```

(continues on next page)

(continued from previous page)

```
        "aversion_groups": set([
            "io-bound",
            # ...
        ])
    }
])
```

In the above example, both `workload1` and `workload2` will try really hard to be scheduled on different nodes, becuase they both list the `io-bound` aversion group in their aversion groups list.

In this example, we have two houses and two college students. Each student goes to a different local university and is part of the same cross-school rivalry. We may model this scenario like this:

```
nodes = lighthouse.Node.from_list([
    {
        "name": "house-1",
        "resources": {
            "bathroom": 25,
            "bedroom": 10,
            "kitchen": 10
        }
    },
    {
        "name": "house-2",
        "resources": {
            "bathroom": 25,
            "bedroom": 10,
            "kitchen": 15
        }
    }
])
workloads = lighthouse.Workload.from_list([
    {
        "name": "college-student-1",
        "requirements": {
            "bathroom": 5,
            "bedroom": 2,
            "kitchen": 2
            },
        "aversion_groups": [
            "north_south_rivalry"
        ]
    },
    {
        "name": "college-student-2",
        "requirements": {
            "bathroom": 5,
            "bedroom": 2,
            "kitchen": 2
            },
        "aversion_groups": [
            "north_south_rivalry"
        ]
    }
])
```

Note: The above example shows that `aversion_groups` can be specified as a list or set when calling `Workload.from_list`, but they are internally represented as sets.

Although there is plenty of room for both college students to live in the same house, any distributor attempting to assign these workloads to the nodes via `attempt_assign_loads` will yield the following assignment:

```
{
    "college-student-1": "house-1",
    "college-student-2": "house-2"
}
```

As can be seen, even though there is plenty of room for both students to be in the same house, they are put in different houses due to them being in the same rivalry (aversion group).

However, if there is no other house in which they might live, the students will still reluctantly be scheduled together. Using this list of nodes instead of the one above:

```
nodes = lighthouse.Node.from_list([
    {
        "name": "house-1",
        "resources": {
            "bathroom": 25,
            "bedroom": 10,
            "kitchen": 10
        }
    }
])
```

The assignments would look like this instead:

```
{
    "college-student-1": "house-1",
    "college-student-2": "house-1"
}
```

CHAPTER 4

Modules

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

First, please read the pylighthouse *Contributor Covenant Code of Conduct*. This project will not take any contribution coming from those who do not abide by the code of conduct. This means that while a person is currently under disciplinary action via avenues set forth in that document, this project will ignore and not incorporate any contributions they might give at that time, including pull requests, bug reports, feature requests, or any other contribution. We here at pylighthouse are committed to caring about the well-being of every one in our community, and we are prepared to act to protect that well-being if needed.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/djhaskin987/pylighthouse/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

pylighthouse could always use more documentation, whether as part of the official pylighthouse docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/djhaskin987/pylighthouse/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *pylighthouse* for local development.

1. Fork the *pylighthouse* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pylighthouse.git
```

3. Install your local copy into a virtualenv. This is how you set up your fork for local development:

```
$ cd pylighthouse/
$ virtualenv ve
$ . ve/bin/activate
$ pip install -r requirements_dev.txt
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ . ve/bin/activate
$ flake8 pylighthouse tests
$ python setup.py test or py.test
$ tox
```

   flake8 and tox should be installed if you are in the pipenv shell. If not, just pip install them into your virtualenv like this:

```
$ pip install --user flake8
$ pip install --user tox
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

When you go to make the PR, please use the following checklist to test whether or not it is likely to be accepted:

2. **Do you have tests in your PR, and do they pass?** Tests are in two places in pylighthouse: the `tests/` directory, where more or less normal unit tests reside. You must have at least a few simple unit tests as a "spot-check" of your feature if the PR is to be merged. The tests need not be elaborate; simple tests are better than no tests.
3. **Is your PR backwards compatible?** The biggest feature pylighthouse provides is backwards compatibility. If pylighthouse breaks a build, it is a bug. A PR is herein defined to be "backwards incompatible" if 1) it significantly changes the content of any previously merged unit or script test and 2) if it breaks any of them.
4. **Did you add documentation around the feature in your PR?** Generally this means adding something to the *usage* <*usage*> document.
5. **Did you add an entry to the Changelog?** This project keeps a curated *changelog*.

There are some exceptions to the above rules. If your patch is less than two lines' difference from the previous version, your PR may be a "typo" PR, which may qualify to get around some of the above rules. Just ask the team on your GitHub issue.

5.4 Tips

To run a subset of tests:

```
$ py.test tests.test_pylighthouse
```

5.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run this:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 6

Contributor Covenant Code of Conduct

6.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

6.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

6.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

6.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

6.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at "djhaskin987 at gmail.com" . All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately under the *Contributing* document.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

6.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant homepage, version 1.4.

CHAPTER 7

Changelog

All notable changes to this project will be documented here.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

7.1 Unreleased

7.1.1 Added

7.1.2 Changed

7.1.3 Fixed

7.2 v0.1.0

7.2.1 Added

- Node class
- Workload class
- Distributor base class
- PrioritizedDistributor class
- RoundRobinDistributor class
- BinPackDistributor class
- distributor classes have a method called `attempt_attach_workloads`, the primary use case of this library

7.2.2 Changed

7.2.3 Fixed

CHAPTER 8

Credits

8.1 Development Lead

- Daniel Jay Haskin <djhaskin987@gmail.com>

8.2 Contributors

None yet. Why not be the first?

CHAPTER 9

License

9.1 Apache Software License 2.0

CHAPTER 10

Indices and tables

- genindex
- modindex
- search