

---

# **cobald Documentation**

*Release 0.13.0*

**Eileen Kuehn, Max Fischer**

**Aug 29, 2022**



# USER DOCUMENTATION

<b>1</b>	<b>Resource and Control Model</b>	<b>3</b>
<b>2</b>	<b>Daemon Infrastructure and Facilities</b>	<b>7</b>
<b>3</b>	<b>Custom Controllers, Pools and Extensions</b>	<b>13</b>
<b>4</b>	<b>Glossary of Terms</b>	<b>19</b>
<b>5</b>	<b>cobald</b>	<b>21</b>
<b>6</b>	<b>ChangeLog</b>	<b>43</b>
<b>7</b>	<b>Versioning and Releases</b>	<b>45</b>
<b>8</b>	<b>Quick Info</b>	<b>47</b>
<b>9</b>	<b>About</b>	<b>49</b>
<b>10</b>	<b>Indices and tables</b>	<b>51</b>
	<b>Python Module Index</b>	<b>53</b>
	<b>Index</b>	<b>55</b>







## RESOURCE AND CONTROL MODEL

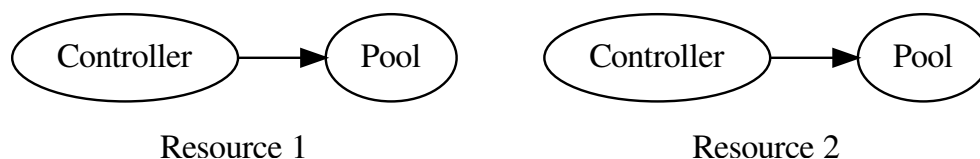
The goal of `cobald` is to simplify the provisioning of *opportunistic resources*. This is achieved with a composable model to define, aggregate, generalise and control resources. The `cobald.interfaces` codify this into a handful of primitive building blocks.

### 1.1 Pool and Control Model

The `cobald` model for controlling resources is built on four simple types of primitives. Two fundamental primitives represent the actual resources and the provisioning strategy:

- The adapter handling concrete resources is a *Pool*. Each Pool merely communicates the total volume of resources and their overall fitness.
- The decision to add or remove resources is made by a *Controller*. Each Controller only inspects the fitness of its Pools and adjusts their desired volume.

These two primitives are sufficient for direct control of simple resources. It is often feasible to control several pools of resources separately.

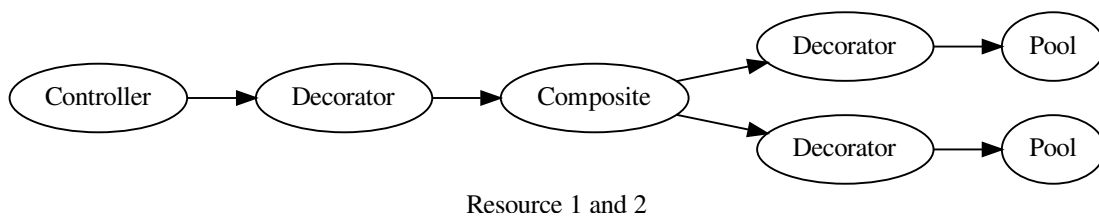


## 1.2 Composition and Decoration

For complex tasks it may be necessary to combine resources or change their interaction and appearance.

- The details of managing resources are encoded by *Decorators*. Each *Decorator* translates between the specific *Pools* and the generic *Controllers*.
- The combination of several resources is made by *CompositePools*. Each *CompositePool* handles several *Pools*, but gives the outward appearance of a single *Pool*.

All four primitives can be combined to express even complex resource and control scenarios. However, there is always a *Controller* on one end and a *Pool* on the other. Since individual primitives can be combined and reused, new use cases require only a minimum of new implementations.



## 1.3 Detail Descriptions

### 1.3.1 Resource Abstraction via Pools

The fundamental abstraction for resources is the *Pool*: a representation for a *number of indistinguishable* resources.

As far as *cobald* is concerned, it is inconsequential which specific resources make up a pool. This allows each *Pool* to implement its own strategy for managing resources. For example, a *Pool* providing virtual machines may silently spawn a new machine to replace another.

The purpose of a *Pool* is just to *provide* resources, not use them for any specific task. For example, the aforementioned VM may integrate into a Batch System which provides the VM with work. What matters to *cobald* is only whether resources match their underlying usage.

#### Supply and Demand

Each *Pool* effectively provides only one type of resources<sup>1</sup>. The only adjustment possible from the outside is how many resources are provided. This is expressed as **supply** and **demand**:

##### **supply** [r/o]

The amount of resources a pool currently provides.

##### **demand** [r/w]

The amount of resources a pool is expected to provide.

<sup>1</sup> What constitutes a single “type” depends on the intended use of the resource. For example, it might be a generic “bytes of storage space” or a specific “consecutive bytes of HDD at 10 ms access time and 2500000 hrs MTBF”.



Note that demand is not derived by a Pool, but should be adjusted from the outside. The task of a Pool is only to adjust its supply to match demand.

### Allocation versus Utilisation

While a Pool does not calculate the demand for its resources, it has to track and expose their usage. This is expressed as two attributes that reflect *how much* and *how well* resources are used:

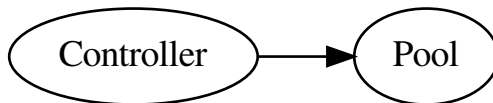
**allocation** [r/o]

Fraction of the supplied resources which are allocated for usage

**utilisation** [r/o]

Fraction of the supplied resources which are actively used

### 1.3.2 Transparent Demand Control



### 1.3.3 Composing Pools of Resources



## DAEMON INFRASTRUCTURE AND FACILITIES

The *cobald.daemon* provides the infrastructure to deploy one or more *resource control pipelines*. Any component integrated into this infrastructure can be configured and controlled in the same fashion.

### 2.1 Component Configuration

Configuration of the *cobald.daemon* is performed at startup via one of two methods: a YAML file or Python code. While the former is more structured and easier to verify, the latter allows for greater freedom.

The configuration file is the only positional argument when launching the *cobald.daemon*. The file extension determines the type of configuration interface to use - `.py` for Python files and `.yaml` for YAML files.

```
$ python3 -m cobald.daemon /etc/cobald/config.yaml
$ python3 -m cobald.daemon /etc/cobald/config.py
```

#### 2.1.1 The YAML Interface

The top level of a YAML configuration file is a mapping with two sections: the `pipeline` section setting up a pool control pipeline, and the `logging` section setting up the logging facilities. The `logging` section is optional and follows the standard *configuration dictionary schema*.<sup>1</sup>

The `pipeline` section must contain a sequence of `Controllers`, `Decorators` and `Pools`. Each `pipeline` is constructed in reverse order: the *last* element should be a `Pool` and is constructed first, then recursively passed to its predecessor for construction.

```
# pool becomes the target of the controller
pipeline:
  - !LinearController
    low_utilisation: 0.9
    high_utilisation: 1.1
  - !CpuPool
    interval: 1
```

---

<sup>1</sup> YAML configurations allow for additional sections to configure plugins. Additional sections are *logged* to the `"cobald.runtime.config"` channel.

## Object References

YAML configurations support `!!` tag and `!` constructor syntax. These allow to use arbitrary Python objects and registered plugins, respectively. Both support keyword and positional arguments.

```
# generic python tag for arbitrary objects
!!python/object:cobald.controller.linear.LinearController {low_utilisation: 0.9}
# constructor tag for registered plugin
!LinearController
low_utilisation: 0.9
```

New in version 0.9.3.

---

**Note:** The YAML configuration is read using `yaml.SafeLoader` to avoid arbitrary code execution. Objects must be marked as safe for loading, either as *COBALD plugins* or using *PyYAML* directly.

---

A legacy format using explicit type references is available, but discouraged. This uses an invocation mechanism that can use arbitrary callables to construct objects: each mapping with a `__type__` key is invoked with its items as keyword arguments, and the optional `__args__` as positional arguments.

```
pipeline:
  # same as `package.module.callable(a, b, keyword1="one", keyword2="two")
  - __type__: package.module.callable
    __args__:
      - a
      - b
  keyword1: one
  keyword2: two
```

Deprecated since version 0.9.3: Use YAML tags and constructors instead.

### 2.1.2 Python Code Inclusion

Python configuration files are loaded like regular modules. This allows to define arbitrary types and functions, and directly chain components or configure logging. At least one pipeline of *Controllers*, *Decorators* and *Pools* should be instantiated.

```
from cobald.controller.linear import LinearController

from cobald_demo.cpu_pool import CpuPool
from cobald_demo.draw_line import DrawLineHook

pipeline = LinearController.s(
    low_utilisation=0.9, high_allocation=1.1
) >> CpuPool()
```

As regular modules, Python configurations must explicitly import the components they use. In addition, everything not bound to a name will be garbage collected. This allows configurations to use temporary objects, e.g. reading from files or sockets, but means persistent objects (such as a pipeline) must be bound to a name.

## 2.2 Standard Logging Facilities

The `cobald.daemon` provides several separate logging channels. Each exposes information from a different view and for a different audience. Both core components and plugins should hook into these channels to supply appropriate information.

### 2.2.1 Logging Channels

Channels are separated by a hierarchical logging name.

#### "cobald.runtime"

Diagnostic information on the health of the daemon and its abstractions. This includes resources initialised (e.g. databases or modules), and any failures that may affect daemon stability (e.g. unavailable resources).

#### "cobald.control"

Information specific to the pool control model. This includes decisions made and statistics used for this purpose.

#### "cobald.monitor"

Monitoring information for automated processing.

Log providers hook into channels by creating a sub-logger. For example, the daemon core uses the `"cobald.runtime.daemon"` logger for diagnostics.

### The Monitor Channel

In contrast to other channels, the `"cobald.monitor"` channel provides structured data. This data is suitable for data transfer formats such as JSON or telegraf. Each entry consists of an identifier and a dictionary of data:

```
# get a separate logger in the 'cobald.monitor' channel
logger = logging.getLogger('cobald.monitor.wheatherapi')
# `message` forms the identifier, `args` contains data
logger.info('forecast', {'temperature': 298, 'humidity': 0.45})
```

Note that the message is *not* formatted with the content of `args``. The specific output format is defined by the `logging.Formatter` used for a `logging.Handler`.

#### *LineProtocolFormatter*

Formatter for the [InfluxDB Line Protocol](#), as used by InfluxDB and Telegraf. This is a structured format, without access to the underlying report metadata. The report message always acts as the measurement key.

Supports adding default data as tags, e.g. as `LineProtocolFormatter({'latitude': 49, 'longitude': 8})`.

```
forecast,latitude=49,longitude=8 humidity=0.45,temperature=298
```

#### *cobald.monitor.format\_json.JsonFormatter*

Formatter for the JSON format. This is an unstructured format, with optional access to the underlying report metadata.

Supports adding default data, e.g. as `JsonFormatter({'latitude': 49, 'longitude': 8})`.

```
{"latitude": 49, "longitude": 8, "temperature": 298, "humidity": 0.45,
 "message": "forecast"}
```

## 2.3 Concurrent Execution

The `cobald.daemon` provides a dedicated concurrent execution environment. This combines several execution mechanisms into a single, consistent runtime. As a result, the daemon can consistently track the lifetime of tasks and react to failures.

The purpose of this is for components to execute concurrently, while ensuring each component is in a valid state. In this regard, the execution environment is similar to an init service such as `systemd`.

### 2.3.1 Registering Background Services

The primary entry point to the runtime is defining services: the main threads of service instances are automatically started, tracked and handled by the `cobald.daemon`. This allows services to update information, manage resources and react to changing conditions.

A service is defined by applying the `service()` decorator to a class. This automatically schedules the `run` method of any instances for execution as a background task.

```
@service(flavour=threading)
class MyService(object):
    # run method of any instances is executed in a thread once the daemon starts
    def run():
        ...
```

### 2.3.2 Task Execution and Abortion

Any background task is adopted by the daemon runtime. Adopted tasks are executed separately for each flavour; this means that `async` code of the same flavour is never run in parallel. However, tasks of non-`async` flavour, such as `threading`, and different flavours can be run in parallel.

Any adopted tasks are considered self-contained by the runtime. Most importantly, they have no parent that can receive return values or exceptions.

**Warning:** Any unhandled return values and exceptions are considered an error. The daemon automatically terminates in this case.

On termination, the daemon aborts all remaining background tasks. Whether this is graceful or not depends on the flavour of each task. In general, coroutines are gracefully terminated whereas subroutines are not.

### 2.3.3 Triggering Background Tasks

The execution environment is exposed as `cobald.daemon.runtime`, an instance of `ServiceRunner`. Via this entry point, new tasks may be launched after the daemon has started.

**runtime.adopt(payload, \*args, flavour, \*\*kwargs)**

Run a payload of the appropriate flavour in the background. The caller is not blocked, but cannot receive any return value or exceptions.

---

**Note:** It is a fatal error if `payload` produces any value or exception.

---

**runtime.execute(payload, \*args, flavour, \*\*kwargs)**

Run a payload of the appropriate flavour until completion. The caller is blocked during execution, and receives any return value or exceptions.

If `*args` or `**kwargs` are provided, the payload is run as `payload(*args, **kwargs)`.

### 2.3.4 Available Flavours

Flavours are identified by the underlying module. The following types are currently supported:

**asnycio**

Coroutines implemented with the `asnycio` library. Payloads are gracefully cancelled.

**trio**

Coroutines implemented with the `trio` library. Payloads are gracefully cancelled.

**threading**

Subroutines implemented with the `threading` library. Payloads run as daemons and ungracefully terminated.

## 2.4 systemd Configs

You can run `cobald` as a system service. We provide `systemd` configs for multiple `cobald` instances run as services. You can manage several instances which are identified with a `systemd` instance name.

Create a file named `cobald@.service` in the `/usr/lib/systemd/system` directory.

An example of a `systemd` config file:

```
[Unit]
Description=COBald - the Opportunistic Balancing Daemon for %I
Documentation=https://cobald.readthedocs.io
After=network.target
Wants=network-online.target
After=network-online.target

[Install]
RequiredBy=multi-user.target

[Service]
Type=simple
ExecStart=/usr/bin/python3 -m cobald.daemon /etc/cobald/%i.py
```

In this example, the configs for the different `COBald` instances are located at `/etc/cobald/instance-name.py`. `cobald` can handle `.py` and `.yaml` configuration files. Please ensure that the chosen python interpreter has `cobald` installed! We recommend to use a `virtualenv`. By using a `virtualenv` you have to set the `ExecStart` to `ExecStart={{ virtualenv }}/bin/python -m cobald.daemon /etc/cobald/%i.yaml`.

After you created or changed the file you need to run:

```
$ systemctl daemon-reload
```

Now you can manage the `cobald` instance which loads the `/etc/cobald/instance-name.py` config file.

- start one instance of cobald

```
$ systemctl start cobald@instance-name
```

- stop the instance of cobald

```
$ systemctl stop cobald@instance-name
```

- report the current status of the cobald instance

```
$ systemctl status cobald@instance-name
```

- enable cobald instance start at boot time

```
$ systemctl enable cobald@instance-name
```



## CUSTOM CONTROLLERS, POOLS AND EXTENSIONS

The *cobald.daemon* is capable of loading any modules and code importable by its Python interpreter. In addition, plugins can be registered for fast access in configuration files. Extensions are integrated as classes that satisfy the *Controller*, *Pool* or *Decorator* interfaces. Internally, extensions can be organized and implemented as required.

### 3.1 Custom Pool Semantics

Adding new types of resources requires writing a new *cobald.interfaces.Pool* implementation. While adherence to the interface ensures compatibility, a custom Pool must also conform to some constraints for consistency.

#### 3.1.1 Behaviour of Pool Implementations

The conventions on Pools are minimal, but their prevalence makes following them critical. Basically, the conventions are implied by the semantics of a Pool's properties.

##### Responsiveness of Properties

The properties *supply*, *demand*, *allocation*, and *utilisation* should respond similar to regular attributes. Getting and setting properties should return quickly - avoid lengthy computations, queries and interactions with external processes. Never use locking for arbitrary times.

If you wish to represent external or complex state, buffer values and react to them or update them at regular intervals.

##### Ordering of Utilisation and Allocation

The model of *allocation* and *utilisation* assumes that only allocated resources can be utilised. As such, *allocation* should generally be greater than *utilisation*. Note that this is a loose assumption that is not enforced. Deviations due to precision or timing should not have a significant impact.

If you have use-cases where this assumption is not applicable, such as overbooking, you may want to write your own *cobald.interfaces.Controller*.

##### Common Utilisation and Allocation scenarios

Depending on the actual resources to manage, it might not be possible to accurately track *allocation* or *utilisation*. Furthermore, at times it is not desirable to use meaningless accuracy. This is why *allocation* and *utilisation* are purposely unrestrictive. The following illustrates several scenarios how to define the two consistently.

Multi-Dimensional Allocations

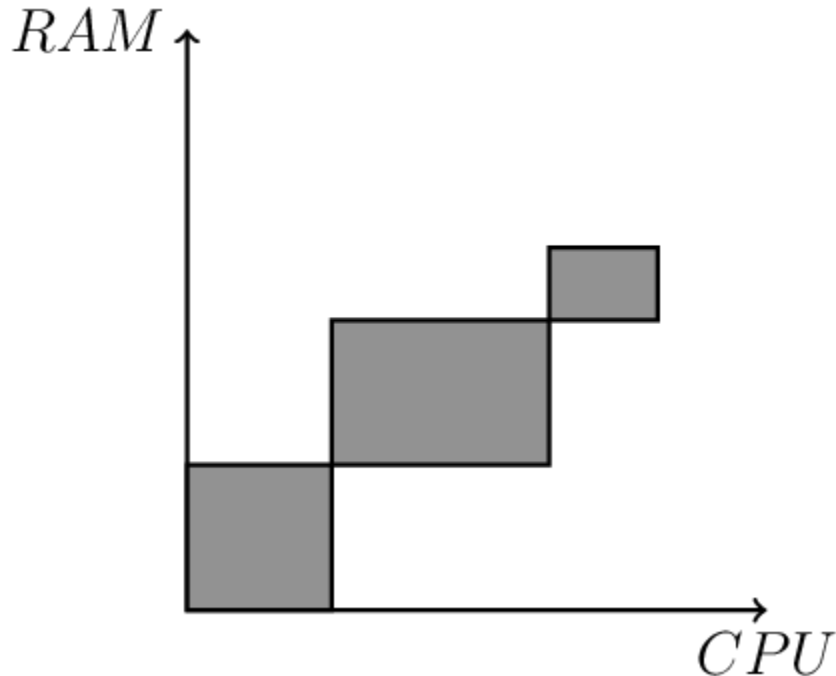


Fig. 1: Allocation of CPU and RAM

## 3.2 Using and Distributing Extensions

Extensions for cobald are regular Python code accessible to the interpreter. For specific problems, extensions can be defined directly in a Python configuration file. General purpose and reusable code should be made available as a Python package. This ensures proper installation and dependency management, and allows quick access from YAML configuration files.

### 3.2.1 Configuration Files

Using Python *configuration files* allows to define arbitrary objects, functions and helpers. This is ideal for minor modifications of existing objects and experimental extensions. Simply add new definitions to the configuration before using them:

```
#!/etc/cobald/my_demo.py
from cobald.interface import Controller

from cobald_demo.cpu_pool import CpuPool
from cobald_demo.draw_line import DrawLineHook

# custom Controller implementation
class StaticController(Controller):
    """Controller that sets demand to a fixed value"""
    def __init__(self, target, demand):
        super().__init__(target)
        self.target.demand = demand
```

(continues on next page)

(continued from previous page)

```
# use custom Controller
pipeline = StaticController.s(demand=50) >> DrawLineHook.s() >> CpuPool(interval=1)
```

Configuration files are easy to use and modify, but impractical for reusable extensions.

### 3.2.2 Python Packages

For generic extensions, Python packages simplify distribution and reuse. Packages are individual `.py` files or folders containing several `.py` files; in addition, packages contain metadata for dependency management and installation.

```
# my_controller.py
from cobald.interfaces import Controller

class StaticController(Controller):
    def __init__(self, target, demand):
        super().__init__(target)
        self.target.demand = demand
```

Packages can be temporarily accessed via `PYTHONPATH` or permanently installed. Once available, packages can be imported and used in any configuration.

```
#!/etc/cobald/my_demo.py
from my_controller import StaticController

from cobald_demo.cpu_pool import CpuPool
from cobald_demo.draw_line import DrawLineHook

# use custom Controller from package
pipeline = StaticController.s(demand=50) >> DrawLineHook.s() >> CpuPool(interval=1)
```

Packages require additional effort to create and use, but are easier to automate and maintain. As with any package, authors should follow the [PyPA recommendations for python packaging](#).

#### The `setup.py` File

The `setup.py` file contains the metadata to install, update and manage a package. For extension packages, it should contain a dependency on `cobald` and the keywords should mention `cobald` for findability.

```
# setup.py
setup(
    # dependency on `cobald` core package
    install_requires=[
        'cobald',
        ...
    ],
    # searchable on pypi index
    keywords='... cobald',
    ...
)
```

## YAML Configuration Plugins

Packages may define two different types of plugins for the *YAML configuration* format: readers for entire configuration sections, and tags for individual configuration elements.

---

**Note:** YAML Plugins only apply to the YAML configuration format. They have no effect if the Python configuration format is used.

---

## YAML Tag Plugins

Tag Plugins allow to execute extensions as configuration elements by using YAML tag syntax, such as `!MyExtension`. Extensions are treated as callables and receive arguments depending on the type of their element: mappings are used as keyword arguments, and sequences are used as positional arguments.

```
# resolves to ExtensionClass(foo=2, bar="Hello World!")
- !MyExtension
  foo: 2
  bar: "Hello World!"
# resolves to ExtensionClass(2, "Hello World!")
- !MyExtension
  - 2
  - "Hello World!"
```

A packages can declare any callable as a Tag Plugin by adding it to the `cobald.config.yaml_constructors` group of `entry_points`; the name of the entry is converted to a Tag when evaluating the configuration. For example, a plugin class `ExtensionClass` defined in `mypackage.mymodule` can be made available as `MyExtension` in this way:

```
setup(
    ...,
    entry_points={
        'cobald.config.yaml_constructors': [
            'MyExtension = mypackage.mymodule:ExtensionClass',
        ],
    },
    ...
)
```

---

**Hint:** Tag Plugins are primarily intended to add custom *Controller*, *Decorator*, and *Pool* types for a COBALD pipeline. If a plugin implements a `s()` method, this is used automatically.

---

---

**Note:** If a plugin requires eager loading of its YAML configuration, decorate it with `cobald.daemon.plugins.yaml_tag()`.

---

New in version 0.12.3: The `cobald.daemon.plugins.yaml_tag()` and eager evaluation.

## Section Plugins

Section Plugins allow to accept and digest new configuration sections. In addition, the cobald daemon verify that there are no unexpected configuration sections to protect against typos and misconfiguration. Extensions are entire top-level sections in the YAML file, which are passed to the plugin after parsing and tag evaluation:

```
# standard cobald pipeline
pipeline:
  - !DummyPool
# passes [{'some_key': 'a', 'more_key': 'b'}, 'foobar', TagPlugin()]
# to the Plugin requesting 'my_plugin'
my_plugin:
  - some_key: a
    more_key: b
  - foobar
  - !TagPlugin
```

A packages can declare any callable as a Section Plugin by adding it to the `cobald.config.sections` group of `entry_points`; the name of the entry is the top-level name of the configuration section. For example, a plugin callable `ConfigReader` defined in `mypackage.mymodule` can request the configuration section `my_plugin` in this way:

```
setup(
    ...,
    entry_points={
        'cobald.config.sections': [
            'my_plugin = mypackage.mymodule:ConfigReader',
        ],
    },
    ...
)
```

---

**Note:** If a plugin must always be covered by configuration, or should run before or after another plugin, decorate it with `cobald.daemon.plugins.constraints()`.

---

New in version 0.12: The `cobald.daemon.plugins.constraints()` and dependency resolution.

## The cobald Namespace

The top-level cobald package itself is a [namespace package](#). This allows the COBaLD developers to add, remove or split sub-packages. In order to not conflict with the core development, do *not* add your own packages to the cobald namespace.



## GLOSSARY OF TERMS

### **Opportunistic Resources**

Any resources available for but not dedicated to a specific task. This includes resources which are acquired temporarily, but not owned permanently. Strongly put, any resource borrowed for usage outside of its dedicated purpose. This includes performing a non-dedicated task instead of idling in the absence of a dedicated task.

### **Indistinguishable Resources**

Any resources that are equally suited to fulfill the same tasks. It is irrelevant which specific resource serves which task. This does not imply strict equality, merely that any differences are inconsequential or negligible.

### **Pool**

A collection of resources which are *indistinguishable*.





## 5.1 cobald namespace

### 5.1.1 Subpackages

**cobald.composite package**

**Submodules**

**cobald.composite.factory module**

**class** `cobald.composite.factory.FactoryPool(*args, **kwargs)`

Bases: *CompositePool*

Composition that adds and removes pools to satisfy demand

#### **Parameters**

- **factory** – a callable that produces a new *Pool*
- **interval** – how often to adjust the number of children

Adjustment uses two extensions that children must respond to adequately:

- When spawned via `factory()`, children shall already be set to their expected demand.
- When disabled via `demand=0`, children shall shut down and free any resources and tasks.

Once spawned, children are free to adjust their demand if required. A child may disable itself permanently by setting its own `demand = 0`. The *FactoryPool* inspects the demand for all its children before spawning or disabling any children.

Any child which satisfies `supply > 0` or `demand > 0` is considered active and contributes to the *FactoryPool* `supply`, `demand`, `allocation`, and `utilisation`. The *FactoryPool* makes no assumption about the validity or fitness of active children. It is the responsibility of children to report their status accordingly. For example, if a child shuts down and does not allocate its `supply` further, it should scale its reported `allocation` accordingly.

#### **property allocation**

Fraction of the provided resources which are assigned for usage

#### **property children**

The individual resource providers making up this pool

**property demand**

The volume of resources to be provided by this pool

**async run()**

Service entry point

**property supply**

The volume of resources that is provided by this pool

**property utilisation**

Fraction of the provided resources which are actively used

**cobald.composite.uniform module**

**class** `cobald.composite.uniform.UniformComposite(*children: Pool)`

Bases: `CompositePool`

Uniform composition of several pools, with each pool weighted the same

**property allocation**

Fraction of the provided resources which are assigned for usage

**children** = []

**property demand**

The volume of resources to be provided by this pool

**property supply**

The volume of resources that is provided by this pool

**property utilisation**

Fraction of the provided resources which are actively used

**cobald.composite.weighted module**

**class** `cobald.composite.weighted.WeightedComposite(*children: Pool, weight: typing_extensions.Literal[supply, utilisation, allocation] = 'supply')`

Bases: `CompositePool`

Composition of pools weighted by their current state

The aggregation of children's *demand*, *utilisation* and *allocation* is weighted by each child's *weight*. Children can be weighted by their *supply*, *utilisation* or *allocation*. Note that weighting the *demand* only applies to *distributing* it to children; the composite's *demand* is always exactly as set by its controller.

If the total weight is 0, the following fallback applies:

- *demand* is applied uniformly, and
- *utilisation* and *allocation* are assumed 1 if there are no children, 0 otherwise.

The latter rule expresses that the total fitness of a Pool is 0 either if the fitness of all its children is 0, or there are no children.

**property allocation**

Fraction of the provided resources which are assigned for usage

**children** = []

**property demand**

The volume of resources to be provided by this pool

**property supply**

The volume of resources that is provided by this pool

**property utilisation**

Fraction of the provided resources which are actively used

## cobald.controller package

### Submodules

#### cobald.controller.linear module

**class** `cobald.controller.linear.LinearController(*args, **kwargs)`

Bases: `Controller`

Controller that linearly increases or decreases demand

#### Parameters

- **target** – the pool to manage
- **low\_utilisation** – pool utilisation below which resources are decreased
- **high\_allocation** – pool allocation above which resources are increased
- **rate** – maximum change of demand in resources per second
- **interval** – interval between adjustments in seconds

**regulate**(*interval*)

**async run**()

Service entry point

#### cobald.controller.relative\_supply module

**class** `cobald.controller.relative_supply.RelativeSupplyController(*args, **kwargs)`

Bases: `Controller`

Controller that adjusts demand relative to supply

#### Parameters

- **target** – the pool to manage
- **low\_utilisation** – pool utilisation below which resources are decreased
- **high\_allocation** – pool allocation above which resources are increased
- **low\_scale** – scale of `target.supply` when decreasing resources
- **high\_scale** – scale of `target.supply` when increasing resources
- **interval** – interval between adjustments in seconds

**regulate**(*interval*)

**async run**()

Service entry point

## cobald.controller.stepwise module

### cobald.controller.stepwise.ControlRule

Individual control rule for a pool on a given interval

When a rule for a *Stepwise* is invoked, it receives the `pool` to manage and the `interval` elapsed since the last modification. It should either *return* the new *demand*, or `None` to indicate no change; the latter can also mean that the function does not hit a return statement.

`\ rule(pool: Pool, interval: float) -> Optional[float]`

Note that a rule should *not* modify the pool directly.

alias of `Callable[[Pool, float], Optional[float]]`

**class** `cobald.controller.stepwise.RangeSelector`(*base: Callable[[Pool, float], Optional[float]]*, *\*rules: Tuple[float, Callable[[Pool, float], Optional[float]]]*)

Bases: `object`

Container that stores rules for the range of their supply bounds

#### Parameters

- **base** – base rule that has no lower bound
- **rules** – lower bound and its control rule

**get\_rule**(*supply: float*)

**class** `cobald.controller.stepwise.Stepwise`(\*args, \*\*kwargs)

Bases: `Controller`

Controller that selects from several strategies based on supply

#### See

*UnboundStepwise* allows creating *Stepwise* instances via decorators.

**async run**()

Service entry point

**class** `cobald.controller.stepwise.UnboundStepwise`(*base: Callable[[Pool, float], Optional[float]]*)

Bases: `object`

Decorator interface for constructing a *Stepwise* controller

Apply this as a decorator to a *ControlRule* callable to create a basic controller skeleton. The initial callable forms the base rule. Additional rules can be added for specific *supply* thresholds using *add()*.

The skeleton can be used like a regular *Controller*: calling it with a *Pool* and update *interval* creates a *Controller* instance with the given rules for the *Pool*.

```
# initial controller skeleton from base case
@stepwise
def control(pool: Pool, interval):
    return 10
```

(continues on next page)

(continued from previous page)

```

# additional rules above specific supply thresholds
@control.add(supply=10)
def quantized(pool: Pool, interval):
    if pool.utilisation < 0.5:
        return pool.demand - 1
    elif pool.allocation > 0.5:
        return pool.demand + 1

@control.add(supply=100)
def continuous(pool: Pool, interval):
    if pool.utilisation < 0.5:
        return pool.demand * 1.1
    elif pool.allocation > 0.5:
        return pool.demand * 0.9

# create controller from skeleton
pipeline = control(pool, interval=10)

```

```

add(rule: Callable[[Pool, float], Optional[float]], *, supply: float) → Callable[[Pool, float], Optional[float]]
add(rule: None, *, supply: float) → Callable[[Callable[[Pool, float], Optional[float]], Callable[[Pool, float],
Optional[float]]]

```

Register a new rule above a given supply threshold

Registration supports a single-argument form for use as a decorator, as well as a two-argument form for direct application. Use the former for def or class definitions, and the later for lambda functions and existing callables.

```

@control.add(supply=10)
def linear(pool, interval):
    if pool.utilisation < 0.75:
        return pool.supply - interval
    elif pool.allocation > 0.95:
        return pool.supply + interval

control.add(
    lambda pool, interval: pool.supply * (
        1.2 if pool.allocation > 0.75 else 0.9
    ),
    supply=100
)

```

```
s(*args, **kwargs) → Partial[Stepwise]
```

Create an unbound prototype of this class, partially applying arguments

```

@stepwise
def control(pool: Pool, interval):
    return 10

pipeline = control.s(interval=20) >> pool

```

#### Note

The partial rules are sealed, and `add()` cannot be called on it.

`cobald.controller.stepwise.stepwise`

alias of *UnboundStepwise*

## cobald.controller.switch module

**class** `cobald.controller.switch.DemandSwitch(*args, **kwargs)`

Bases: *Controller*

Controller that dispatches to slaved controllers based on demand

```
DemandSwitch(pool, linear_control, 10, supply_control)
```

### Parameters

- **target** – the pool on which to regulate demand
- **default** – controller to use by default
- **slaves** – pairs of minimum demand to switch and corresponding controller
- **interval** – interval between adjustments in seconds

**regulate**(*interval*)

**async run**()

Service entry point

## cobald.daemon package

`cobald.daemon.runtime = <cobald.daemon.runners.service.ServiceRunner object>`

The runner invoked on daemon startup

`cobald.daemon.service(flavour)`

Mark a class as implementing a Service

Each Service class must have a `run` method, which does not take any arguments. This method is *adopt*()ed after the daemon starts, unless

- the Service has been garbage collected, or
- the ServiceUnit has been `cancel`()ed.

For each service instance, its *ServiceUnit* is available at `service_instance.__service_unit__`.

## Subpackages

### cobald.daemon.config package

## Submodules

### cobald.daemon.config.mapping module

**exception** `cobald.daemon.config.mapping.ConfigurationError`(*what: Any, where: Optional[str] = None*)

Bases: `Exception`

`cobald.daemon.config.mapping.M`

type of a mapping element, matching JSON/YAML

alias of `TypeVar('M', str, int, float, bool, dict, list)`

**class** `cobald.daemon.config.mapping.SectionPlugin`(*section: str, digest: Callable[[M], Any], requirements: PluginRequirements*)

Bases: `Generic[M]`

Plugin to digest a top-level configuration section

#### Parameters

- **section** – Name of the section to digest
- **digest** – callable that receives the section
- **requirements** – plugin requirements

**property** `after`

**property** `before`

**digest**

**classmethod** `load(entry_point: EntryPoint) → SectionPlugin`

Load a plugin from a pre-parsed entry point

Parses the following options:

**required**

If present implies `required=True`.

**before=other**

This plugin must be processed before `other`.

**after=other**

This plugin must be processed after `other`.

**property** `required`

**requirements**

**section**

**class** `cobald.daemon.config.mapping.Translator`

Bases: `object`

Translator from a mapping to an initialised object hierarchy

**construct**(*mapping: dict, \*\*kwargs*)

Construct an object from a mapping

#### Parameters

- **mapping** – constructor definition, with `__type__` and keyword arguments
- **kwargs** – additional keyword arguments to pass to the constructor

**static load\_name**(*absolute\_name: str*)

Load an object based on an absolute, dotted name

**translate\_hierarchy**(*structure: M, \*, where: str = "", \*\*construct\_kwargs*) → M

`cobald.daemon.config.mapping.configure_logging`(*logging\_mapping: dict*)

`cobald.daemon.config.mapping.load_configuration`(*config\_data: Dict[str, Any], plugins: Tuple[SectionPlugin] = ()*) → Dict[SectionPlugin, Any]

Load the configuration from a mapping, applying plugins to sections

**Parameters**

- **config\_data** – the raw configuration without any plugins applied
- **plugins** – all plugins that *might* apply, in order

**Returns**

the output of all applied plugins

### `cobald.daemon.config.python` module

`cobald.daemon.config.python.load_configuration`(*path*)

Load a configuration from a module stored at path

The path must end in a valid file extension for the appropriate module type, such as .py or .pyc for a plaintext or bytecode python module.

**Raises**

**ValueError** – if the extension does not mark a known module type

### `cobald.daemon.config.yaml` module

`cobald.daemon.config.yaml.load_configuration`(*path: str, loader: ~typing.Type[~yaml.loader.BaseLoader] = <class 'yaml.loader.SafeLoader'>, plugins: ~typing.Tuple[~cobald.daemon.config.mapping.SectionPlugin] = ()*)

`cobald.daemon.config.yaml.yaml_constructor`(*factory: Callable[[...], R], \*, eager: bool = False*) → Callable[[...], R]

Convert a factory function/class to a YAML constructor

**Parameters**

- **factory** – the factory function/class
- **eager** – whether the YAML must be evaluated eagerly

**Returns**

factory constructor

Applying this helper to a factory allows it to be used as a YAML constructor, without it knowing about YAML itself. It properly constructs nodes and converts mapping nodes to `factory(**node)`, sequence nodes to `factory(*node)`, and scalar nodes to `factory()`.



For example, registering the constructor `yaml_constructor(factory)` as `!factory` means the following YAML is converted to `factory(a=0.3, b=0.7)`:

```
- !factory
  a: 0.3
  b: 0.7
```

Since YAML can express recursive data, nested data structures are evaluated lazily by default. Set `eager=True` to enforce eager evaluation before calling the constructor.

## cobald.daemon.core package

### Submodules

#### cobald.daemon.core.cli module

#### cobald.daemon.core.config module

**class** `cobald.daemon.core.config.COBalDLoader`(*stream*)

Bases: `SafeLoader`

Loader with access to COBalD configuration constructors

**class** `cobald.daemon.core.config.PipelineTranslator`

Bases: `Translator`

Translator for cobald pipelines

This allows for YAML configurations to have one or several pipeline elements. Each pipeline is translated as a series of nested elements, the way a `Controller` receives a `Pool`.

```
pipeline:
  # same as `package.module.callable(a, b, keyword1="one", keyword2="two")
  - __type__: package.module.Controller
    interval: 20
  - __type__: package.module.Pool
```

**translate\_hierarchy**(*structure*, \*, *where*="", *\*\*construct\_kwargs*)

`cobald.daemon.core.config.add_constructor_plugins`(*entry\_point\_group*: *str*, *loader*: `Type[BaseLoader]`) → `None`

Add PyYAML constructors from an entry point group to a loader

#### Parameters

- **loader** – the PyYAML loader which uses the plugins
- **entry\_point\_group** – entry point group to search

---

**Note:** This directly modifies the loader by calling `add_constructor()`.

---

`cobald.daemon.core.config.load`(*config\_path*: *str*)

Load a configuration and keep it alive for the given context

**Parameters**

**config\_path** – path to a configuration file

`cobald.daemon.core.config.load_pipeline`(*content: list*)

Load a cobald pipeline of Controller >> ... >> Pool from a configuration section

**Parameters**

**content** – content of the configuration section

**Returns**

`cobald.daemon.core.config.load_section_plugins`(*entry\_point\_group: str*) → `Tuple[SectionPlugin]`

Load configuration plugins from an entry point group

**Parameters**

**entry\_point\_group** – entry point group to search

**Returns**

all loaded plugins

### **cobald.daemon.core.logger module**

`cobald.daemon.core.logger.create_handler`(*target: str*)

Create a handler for logging to target

`cobald.daemon.core.logger.initialise_logging`(*level: str, target: str, short\_format: bool*)

Initialise basic logging facilities

### **cobald.daemon.core.main module**

Daemon core specific to cobald

`cobald.daemon.core.main.cli_run`()

Run the daemon from a command line interface

`cobald.daemon.core.main.run`(*configuration: str, level: str, target: str, short\_format: bool*)

Run the daemon and all its services

### **cobald.daemon.runners package**

#### **Submodules**

#### **cobald.daemon.runners.async\_tools module**

#### **cobald.daemon.runners.asyncio\_runner module**

**class** `cobald.daemon.runners.asyncio_runner.AsyncioRunner`(*asyncio\_loop: AbstractEventLoop*)

Bases: `BaseRunner`

Runner for coroutines with `asyncio`

All active payloads are actively cancelled when the runner is closed.

**async aclose()**

Shut down this runner

```
flavour = <module 'asyncio' from
'/home/docs/.pyenv/versions/3.7.9/lib/python3.7/asyncio/__init__.py'>
```

**async manage\_payloads()**

Implementation of managing payloads when `run()`

This method must continuously execute payloads sent to the runner. It may only return when `stop()` is called or if any orphaned payload return or raise. In the latter case, *OrphanedReturn* or the raised exception must re-raised by this method.

**register\_payload(payload: Callable[[], Awaitable])**

Register payload for background execution in a threadsafe manner

This runs payload as an orphaned background task as soon as possible. It is an error for payload to return or raise anything without handling it.

**run\_payload(payload: Callable[[], Coroutine])**

Execute payload and return its result in a threadsafe manner

This runs payload as soon as possible, blocking until completion. Should payload return or raise anything, it is propagated to the caller.

**cobald.daemon.runners.asyncio\_watcher module****cobald.daemon.runners.base\_runner module**

**class** `cobald.daemon.runners.base_runner.BaseRunner`(*asyncio\_loop: AbstractEventLoop*)

Bases: `object`

Concurrency backend on top of *asyncio*

**abstract async aclose()**

Shut down this runner

**flavour = None**

**abstract async manage\_payloads()**

Implementation of managing payloads when `run()`

This method must continuously execute payloads sent to the runner. It may only return when `stop()` is called or if any orphaned payload return or raise. In the latter case, *OrphanedReturn* or the raised exception must re-raised by this method.

**async ready()**

Wait until the runner is ready to accept payloads

**abstract register\_payload(payload)**

Register payload for background execution in a threadsafe manner

This runs payload as an orphaned background task as soon as possible. It is an error for payload to return or raise anything without handling it.

### **async run()**

Execute all current and future payloads in an *asyncio* coroutine

This method will continuously execute payloads sent to the runner. It only returns when `stop()` is called or if any orphaned payload returns or raises. In the latter case, *OrphanedReturn* or the raised exception is re-raised by this method.

Implementations should override `manage_payloads()` to customize their specific parts.

### **abstract run\_payload(payload)**

Execute `payload` and return its result in a threadsafe manner

This runs `payload` as soon as possible, blocking until completion. Should `payload` return or raise anything, it is propagated to the caller.

### **stop()**

Stop execution of all current and future payloads and block until success

**exception** `cobald.daemon.runners.base_runner.OrphanedReturn(who, value)`

Bases: `Exception`

A runnable returned a value without anyone to receive it

## **cobald.daemon.runners.guard module**

`cobald.daemon.runners.guard.exclusive(via=<built-in function allocate_lock>) → Callable[[C], C]`

Mark a callable as exclusive

#### **Parameters**

**via** – factory for a Lock to guard the callable

Guards the callable against being entered again before completion. Explicitly raises a `RuntimeError` on violation.

#### **Note**

If applied to a method, it is exclusive across all instances.

## **cobald.daemon.runners.meta\_runner module**

**class** `cobald.daemon.runners.meta_runner.MetaRunner`

Bases: `object`

Unified interface to schedule subroutines and coroutines for concurrent execution

**register\_payload(\*payloads, flavour: module)**

Queue one or more payloads for execution after its runner is started

**run()**

Run all runners, blocking until completion or error

**run\_payload(payload, \*, flavour: module)**

Execute one payload and return its output

This method will block until the payload is completed. It is an error to call it during initialisation before the runners are started.

```
runner_types = (<class 'cobald.daemon.runners.trio_runner.TrioRunner'>, <class
'cobald.daemon.runners.asyncio_runner.AsyncioRunner'>, <class
'cobald.daemon.runners.thread_runner.ThreadRunner'>)
```

**property runners**

**stop()**

Stop all runners

## cobald.daemon.runners.service module

**class** cobald.daemon.runners.service.**ServiceRunner**(*accept\_delay: float = 1*)

Bases: `object`

Runner for coroutines, subroutines and services

The service runner prevents silent failures by tracking concurrent tasks and therefore provides safer concurrency. If any task fails with an exception or provides unexpected output values, this is registered as an error; the runner will gracefully shut down all tasks in this case.

To provide `async` concurrency, the runner also manages common `async` event loops and tracks them for failures as well. As a result, `async` code should usually use the “current” event loop directly.

**accept()**

Start accepting synchronous, asynchronous and service payloads

Since services are globally defined, only one `ServiceRunner` may `accept()` payloads at any time.

**adopt**(*payload, \*args, flavour: module, \*\*kwargs*)

Concurrently run `payload` in the background

If `*args*` and/or `**kwargs` are provided, pass them to `payload` upon execution.

**execute**(*payload, \*args, flavour: module, \*\*kwargs*)

Synchronously run `payload` and provide its output

If `*args*` and/or `**kwargs` are provided, pass them to `payload` upon execution.

**shutdown()**

Shutdown the accept loop and stop running payloads

**class** cobald.daemon.runners.service.**ServiceUnit**(*service, flavour*)

Bases: `object`

Definition for running a service

### Parameters

- **service** – the service to run
- **flavour** – runner flavour to use for running the service

**property running**

**start**(*runner: MetaRunner*)

**classmethod units**() → `Set[ServiceUnit]`

Container of all currently defined units

`cobald.daemon.runners.service.Service`(*flavour*)

Mark a class as implementing a Service

Each Service class must have a `run` method, which does not take any arguments. This method is `adopt`()ed after the daemon starts, unless

- the Service has been garbage collected, or
- the ServiceUnit has been `cancel`()ed.

For each service instance, its `ServiceUnit` is available at `service_instance.__service_unit__`.

### `cobald.daemon.runners.thread_runner` module

**class** `cobald.daemon.runners.thread_runner.ThreadRunner`(*asyncio\_loop: AbstractEventLoop*)

Bases: `BaseRunner`

Runner for subroutines with `threading`

Active payloads are *not* cancelled when the runner is closed. Only program termination forcefully cancels leftover payloads.

**async** `aclose`()

Shut down this runner

`flavour = <module 'threading' from  
'/home/docs/.pyenv/versions/3.7.9/lib/python3.7/threading.py'>`

**async** `manage_payloads`()

Implementation of managing payloads when `run`()

This method must continuously execute payloads sent to the runner. It may only return when `stop`() is called or if any orphaned payload return or raise. In the latter case, `OrphanedReturn` or the raised exception must re-raised by this method.

**register\_payload**(*payload*)

Register `payload` for background execution in a threadsafe manner

This runs `payload` as an orphaned background task as soon as possible. It is an error for `payload` to return or raise anything without handling it.

**run\_payload**(*payload*)

Execute `payload` and return its result in a threadsafe manner

This runs `payload` as soon as possible, blocking until completion. Should `payload` return or raise anything, it is propagated to the caller.

### `cobald.daemon.runners.trio_runner` module

**class** `cobald.daemon.runners.trio_runner.TrioRunner`(*asyncio\_loop: AbstractEventLoop*)

Bases: `BaseRunner`

Runner for coroutines with `trio`

All active payloads are actively cancelled when the runner is closed.

**async** `aclose`()

Shut down this runner

```
flavour = <module 'trio' from '/home/docs/checkouts/readthedocs.org/user_builds/cobald/envs/stable/lib/python3.7/site-packages/trio/__init__.py'>
```

#### **async manage\_payloads()**

Implementation of managing payloads when `run()`

This method must continuously execute payloads sent to the runner. It may only return when `stop()` is called or if any orphaned payload return or raise. In the latter case, *OrphanedReturn* or the raised exception must re-raised by this method.

#### **async ready()**

Wait until the runner is ready to accept payloads

#### **register\_payload(payload: Callable[[], Awaitable])**

Register payload for background execution in a threadsafe manner

This runs payload as an orphaned background task as soon as possible. It is an error for payload to return or raise anything without handling it.

#### **run\_payload(payload: Callable[[], Coroutine])**

Execute payload and return its result in a threadsafe manner

This runs payload as soon as possible, blocking until completion. Should payload return or raise anything, it is propagated to the caller.

## Submodules

### **cobald.daemon.debug module**

```
class cobald.daemon.debug.NameRepr(target)
```

Bases: `object`

Lazy pretty formatter for name of objects

```
cobald.daemon.debug.pretty_module(obj: module) → str
```

```
cobald.daemon.debug.pretty_partial(obj: partial) → str
```

```
cobald.daemon.debug.pretty_ref(obj: Any) → str
```

```
cobald.daemon.debug.pretty_ref(obj: partial) → str
```

```
cobald.daemon.debug.pretty_ref(obj: module) → str
```

Pretty object reference using `module.path:qual.name` format

### **cobald.decorator package**

#### Submodules

### **cobald.decorator.buffer module**

```
class cobald.decorator.buffer.Buffer(*args, **kwargs)
```

Bases: `PoolDecorator`

A timed buffer for changes to a pool

#### Parameters

- **target** – the pool to which changes are applied
- **window** – interval after which changes are applied

Any changes made to *demand* are stored internally. Every *window* seconds, the final demand is applied to target.

**demand = 0.0**

**async run()**

Service entry point

### **cobald.decorator.coarser module**

### **cobald.decorator.limiter module**

### **cobald.decorator.logger module**

```
class cobald.decorator.logger.Logger(target: Pool, name: Optional[str] = None, message: str = 'demand = %(value)s [demand=%(demand)s, supply=%(supply)s, utilisation=%(utilisation).2f, allocation=%(allocation).2f]', level: int = 20)
```

Bases: *PoolDecorator*

Log a message on every change of *demand*

#### **Parameters**

- **name** – name of the `logging.Logger` to log to
- **message** – format for message to emit on every change
- **level** – numerical logging level

The message parameter is used as a %-style format string with named fields. Valid named format fields are

#### **value**

for the new demand being set,

#### **demand, supply, utilisation and allocation**

for the current state of *target*, and

#### **target**

for the target pool itself.

For example, a message of "adjust demand from %(demand)s to %(value)s" will log the old and new demand value.

Deprecated since version 0.12.2: The `consumption` format field. Use `allocation` instead.

#### **property demand**

The volume of resources to be provided by this site

**property name:** `str`



## cobald.decorator.standardiser module

**class** `cobald.decorator.standardiser.Standardiser`(*target: Pool, minimum: float = -inf, maximum: float = inf, granularity: int = 1, backlog: float = inf, surplus: float = inf*)

Bases: `PoolDecorator`

Limits for changes to the demand of a pool

### Parameters

- **target** – the pool on which changes are standardised
- **minimum** – minimum `target.demand` allowed
- **maximum** – maximum `target.demand` allowed
- **granularity** – granularity of `target.demand`
- **surplus** – how much `target.demand` may be above `target.supply`
- **backlog** – how much `target.demand` may be below `target.supply`

The supply and backlog clamp the demand such that  $\text{supply} - \text{backlog} \leq \text{demand} \leq \text{supply} + \text{surplus}$  holds.

The default values apply no limits at all so that isolated limits may be used. When several limits are set, `granularity` has the weakest priority, both `surplus` and `backlog` may limit the result of `granularity`, and `minimum` and `maximum` overrule all other limits.

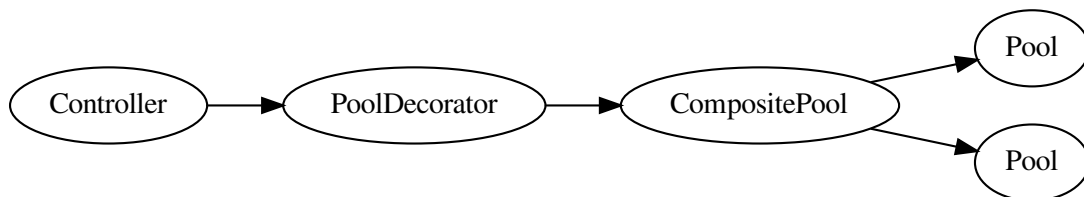
**property demand:** `float`

The volume of resources to be provided by this site

## cobald.interfaces package

Interfaces for primitives of the cobald model

Each `Pool` provides a varying number of resources. A `Controller` adjusts the number of resources that a `Pool` must provide. Several `Pools` can be combined in a single `CompositePool` to appear as one. To modify how a `Pool` presents or digests data, any number of `PoolDecorator` may proceed it.



**class** `cobald.interfaces.CompositePool`

Bases: `Pool`

Concatenation of multiple providers for a number of indistinguishable resources

**abstract property allocation:** `float`

Fraction of the provided resources which are assigned for usage

**abstract property children:** `List[Pool]`

The individual resource providers making up this pool

**abstract property demand**

The volume of resources to be provided by this pool

**abstract property supply**

The volume of resources that is provided by this pool

**abstract property utilisation:** `float`

Fraction of the provided resources which are actively used

**class** `cobald.interfaces.Controller`(*target*: `Pool`)

Bases: `object`

Controller adjusting the demand in a `Pool`

**Parameters**

**target** – the resource pool for which demand is adjusted

**classmethod** `s`(\*args, \*\*kwargs) → `Partial[C]`

Create an unbound prototype of this class, partially applying arguments

```
controller = Controller.s(interval=20)

pipeline = controller(rate=10) >> pool
```

**class** `cobald.interfaces.Partial`(*ctor*: `Type[C_co]`, \*args, *\_\_leaf\_\_*, \*\*kwargs)

Bases: `Generic[C_co]`

Partial application and chaining of Pool `Controllers` and `Decorators`

This class acts similar to `functools.partial`, but allows for repeated application (currying) and explicit binding via the `>>` operator.

```
# incrementally prepare controller parameters
control = Partial(Controller, rate=10, interval=10)
control = control(low_utilisation=0.5, high_allocation=0.9)

# apply target by chaining
pipeline = control >> Decorator() >> Pool()
```

**Note**

The keyword argument `__leaf__` is reserved for internal usage.

**Note**

Binding `Controllers` and `Decorators` creates a temporary `PartialBind`. Only binding to a `Pool` as the last element creates a concrete binding.

**args**

**ctor**

**kwargs**

**leaf**

**class** `cobald.interfaces.Pool`

Bases: `object`

Individual provider for a number of indistinguishable resources

**abstract property allocation:** `float`

Fraction of the provided resources which are assigned for usage

**abstract property demand:** `float`

The volume of resources to be provided by this pool

**classmethod** `s(*args, **kwargs) → Partial[C]`

Create an unbound prototype of this class, partially applying arguments

```
pool = RemotePool.s(port=1337)

pipeline = controller >> pool(host='localhost')
```

**abstract property supply:** `float`

The volume of resources that is provided by this pool

**abstract property utilisation:** `float`

Fraction of the provided resources which are actively used

**class** `cobald.interfaces.PoolDecorator(target: Pool)`

Bases: `Pool`

Decorator modifying how a pool provides resources

**Parameters**

**target** – the resource pool for which demand is adjusted

**property allocation:** `float`

Fraction of the provided resources which is assigned for usage

**property demand**

The volume of resources to be provided by this site

**classmethod** `s(*args, **kwargs) → Partial[C]`

Create an unbound prototype of this class, partially applying arguments

```
decorator = Buffer.s(window=20)

pipeline = controller >> decorator >> pool
```

**property supply**

The volume of resources that is provided by this site

**property utilisation:** `float`

Fraction of the provided resources which is actively used

## cobald.monitor package

### Submodules

#### cobald.monitor.format\_json module

```
class cobald.monitor.format_json.JsonFormatter(fmt: Optional[dict] = None, datefmt: Optional[str] = None)
```

Bases: `Formatter`

Formatter that emits data as JSON

#### Parameters

- **fmt** – default data for all records
- **datefmt** – format for timestamps

The `datefmt` parameter has almost the same meaning as `Formatter`. Setting it to `None` uses the default time format. However, setting it to any other value that is boolean false excludes the timestamp from reports.

**format**(*record*: `LogRecord`)

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using `LogRecord.getMessage()`. If the formatting string uses the time (as determined by a call to `usesTime()`, `formatTime()` is called to format the event time. If there is exception information, it is formatted using `formatException()` and appended to the message.

#### cobald.monitor.format\_line module

```
class cobald.monitor.format_line.LineProtocolFormatter(tags: Optional[Union[Dict[str, Any], Set[str]]] = None, resolution: Optional[float] = None)
```

Bases: `Formatter`

Formatter that emits data as InfluxDB Line Protocol

#### Parameters

- **tags** – record data to use as tags
- **resolution** – resolution of timestamps in seconds

The `tags` act as a whitelist for record keys if they are an iterable. When a dictionary is supplied, its values act as default values if the key is not in a record.

The `resolution` allows summarising data by downsampling the timestamps to the given resolution, e.g. for a resolution of 10 you can expect timestamps 10, 20, 30, ... If `resolution` is `None` the timestamp is omitted from the Line Protocol and Telegraf will take care on setting the current timestamp.

**format**(*record*: `LogRecord`) → `str`

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using `LogRecord.getMessage()`. If the formatting string uses the time

(as determined by a call to `usesTime()`, `formatTime()` is called to format the event time. If there is exception information, it is formatted using `formatException()` and appended to the message.

`cobald.monitor.format_line.escape_field(field: T) → T`

`cobald.monitor.format_line.escape_key(key: str) → str`

`cobald.monitor.format_line.line_protocol(name, tags: Optional[dict] = None, fields: Optional[dict] = None, timestamp: Optional[float] = None) → str`

Format a report as per InfluxDB line protocol

#### Parameters

- **name** – name of the report
- **tags** – tags identifying the specific report
- **fields** – measurements of the report
- **timestamp** – when the measurement was taken, in **seconds** since the epoch

## cobald.utility package

**exception** `cobald.utility.InvariantError`

Bases: `Exception`

An invariant is violated

`cobald.utility.enforce(condition: bool, exception: BaseException = InvariantError())`

Enforce that `condition` is set by raising `exception` otherwise

This is a replacement for `assert` statements as part of validation. It cannot be disabled with `-O` and may raise arbitrary exceptions.

```
def sqrt(value):
    condition(value > 0, ValueError('value must be greater than zero'))
    return math.sqrt(value)
```

`cobald.utility.pairwise(iterable)`

Iterator yielding consecutive pairs from `iterable`

## Submodules

`cobald.utility.primitives module`



## CHANGELOG

### 6.1 0.13 Series

#### 6.1.1 Version [0.13.0] - 2022-08-16

- **[Changed]** Configuration is processed after daemon and *asyncio* initialisation
- **[Changed]** Daemon core implementation is based on *asyncio*

### 6.2 0.12 Series

#### 6.2.1 Version [0.12.3] - 2021-10-29

- **[Added]** YAML ! tags may be eagerly evaluated

#### 6.2.2 Version [0.12.2] - 2021-09-15

- **[Fixed]** pipeline configuration may combine `__type__` and `!yaml` style
- **[Fixed]** pipeline configuration no longer suppresses `TypeError`

#### 6.2.3 Version [0.12.1] - 2020-04-15

- **[Fixed]** fallback for fitness of `WeightedComposite` depends on supply

#### 6.2.4 Version [0.12.0] - 2020-02-26

- **[Changed]** Section Plugin settings are now specified via decorators

## 6.3 0.11 Series

### 6.3.1 Version [0.11.0] - 2020-02-24

- **[Changed]** COBaLD configuration files may include additional sections

## 6.4 0.10 Series

### 6.4.1 Version [0.10.0] - 2019-09-03

- **[Added]** Pools can be templated via `.s` in Python configuration files
- **[Added]** YAML configuration files support plugins via `!MyPlugin` tags
- **[Added]** the `cobald` namespace allows for external plugin packages
- **[Fixed]** fixed Line Protocol sending illegal content
- **[Security]** YAML configuration files no longer allow arbitrary `!!python/object` tags



## VERSIONING AND RELEASES

The COBalD versioning follows [Semantic Versioning](#). Releases are automatically pushed to PyPI from the [GitHub COBalD repository](#).

### 7.1 Versioning and API stability

COBalD is currently published only in the *major version zero* series. The public API is not entirely stable, and may change between releases. However, API changes are already kept to a minimum and significant API changes *SHOULD* relate to an increase of the minor version.

Packages that depend on the COBalD *major version zero* series should accept [compatible release](#) versions for minor versions. For example, a package requiring at least cobald version 0.12.1 should require `cobald ~= 0.12.1` to not accidentally accept `cobald >= 0.13.0`.

### 7.2 Release Process

There is no fixed schedule for releases; a release is manually started whenever significant changes have accumulated or a bugfix requires a prompt publication.

---

**Note:** The following section is only relevant for maintainers of COBalD.

---

Releases are automatically published to PyPI when a GitHub release is created. Each release should be prepared and reviewed via a pull request.

1. **Create a new branch `releases/v<version>` and pull request**
  - Add all to-be-released pull requests to the description
2. **Review all changes added by the new release**
  - Ensure naming, unittests and docs are appropriate
3. **Merge new version metadata (e.g. `v3.9.2`) to repository**
  - Fix change fragment version via `change log ... release 3.9.2`
  - Adjust and commit `__version__ = "3.9.2"` in `cobald.__about__`
  - Create a git tag such as `git tag -a "v3.9.2" -m "important changes"`

Once the pull request has been reviewed and merged, create a new [GitHub release](#).



The `cobald` is a lightweight framework to balance opportunistic resources: cloud bursting, container orchestration, allocation scaling and more. Its lightweight *model* for resources and their composition makes it easy to integrate custom resources and manage them at a large scale. The idea is as simple as it gets:

Start good things.

Stop bad things.

**See also:**

The [cobald demo](#) is a minimal working toy example for using `cobald`.

## QUICK INFO

In the current state, `cobald` is a research and expert tool targeting administrators and developers. You have to manually select your resource backends and compose the strategy. Still, the simplicity of `cobald` should make it accessible for interested users as well.

### *Getting COBalD up and running*

Have a look at the [cobald demo](#). It provides a minimal working example for running COBalD. The demo shows you how to install, configure and run your own COBalD instance.

### *Using COBalD to horizontally scale an HTCondor Pool*

The [TARDIS](#) project provides backends to several cloud providers. This allows you to orchestrate prebuilt VM images.



## ABOUT

The `coba1d` project originates from research on dynamically providing Cloud resources for analysts of the LHC collaborations. It supersedes past work on the [ROCED](#) Cloud resource provider, generalising its goal of provisioning opportunistic resources.

The development of `coba1d` is currently organized by the GridKa and CMS research groups at KIT. We openly encourage adoption and contributions outside of KIT, LHC and our current selection of opportunistic resources. Information on deployment as well as creating and publishing custom plugins will follow.

Please contact us on [github](#) or [gitter](#) if you want to contribute.



## INDICES AND TABLES

- genindex
- modindex
- search





## PYTHON MODULE INDEX

### C

- `cobald.composite`, 21
- `cobald.composite.factory`, 21
- `cobald.composite.uniform`, 22
- `cobald.composite.weighted`, 22
- `cobald.controller`, 23
- `cobald.controller.linear`, 23
- `cobald.controller.relative_supply`, 23
- `cobald.controller.stepwise`, 24
- `cobald.controller.switch`, 26

### d

- `cobald.daemon`, 26
- `cobald.daemon.config`, 26
- `cobald.daemon.config.mapping`, 26
- `cobald.daemon.config.python`, 28
- `cobald.daemon.config.yaml`, 28
- `cobald.daemon.core`, 29
- `cobald.daemon.core.cli`, 29
- `cobald.daemon.core.config`, 29
- `cobald.daemon.core.logger`, 30
- `cobald.daemon.core.main`, 30
- `cobald.daemon.debug`, 35
- `cobald.daemon.runners`, 30
- `cobald.daemon.runners.asyncio_runner`, 30
- `cobald.daemon.runners.base_runner`, 31
- `cobald.daemon.runners.guard`, 32
- `cobald.daemon.runners.meta_runner`, 32
- `cobald.daemon.runners.service`, 33
- `cobald.daemon.runners.thread_runner`, 34
- `cobald.daemon.runners.trio_runner`, 34
- `cobald.decorator`, 35
- `cobald.decorator.buffer`, 35
- `cobald.decorator.coarser`, 36
- `cobald.decorator.limiter`, 36
- `cobald.decorator.logger`, 36
- `cobald.decorator.standardiser`, 37

### i

- `cobald.interfaces`, 37

### m

- `cobald.monitor`, 40
- `cobald.monitor.format_json`, 40
- `cobald.monitor.format_line`, 40

### u

- `cobald.utility`, 41
- `cobald.utility.primitives`, 41



## A

**accept()** (*cobald.daemon.runners.service.ServiceRunner* method), 33  
**aclose()** (*cobald.daemon.runners.asyncio\_runner.AsyncioRunner* method), 30  
**aclose()** (*cobald.daemon.runners.base\_runner.BaseRunner* method), 31  
**aclose()** (*cobald.daemon.runners.thread\_runner.ThreadRunner* method), 34  
**aclose()** (*cobald.daemon.runners.trio\_runner.TrioRunner* method), 34  
**add()** (*cobald.controller.stepwise.UnboundStepwise* method), 25  
**add\_constructor\_plugins()** (in module *cobald.daemon.core.config*), 29  
**adopt()** (*cobald.daemon.runners.service.ServiceRunner* method), 33  
**after** (*cobald.daemon.config.mapping.SectionPlugin* property), 27  
**allocation** (*cobald.composite.factory.FactoryPool* property), 21  
**allocation** (*cobald.composite.uniform.UniformComposite* property), 22  
**allocation** (*cobald.composite.weighted.WeightedComposite* property), 22  
**allocation** (*cobald.interfaces.CompositePool* property), 37  
**allocation** (*cobald.interfaces.Pool* property), 39  
**allocation** (*cobald.interfaces.PoolDecorator* property), 39  
**args** (*cobald.interfaces.Partial* attribute), 38  
**AsyncioRunner** (class in *cobald.daemon.runners.asyncio\_runner*), 30

## B

**BaseRunner** (class in *cobald.daemon.runners.base\_runner*), 31  
**before** (*cobald.daemon.config.mapping.SectionPlugin* property), 27  
**Buffer** (class in *cobald.decorator.buffer*), 35

## C

**children** (*cobald.composite.factory.FactoryPool* property), 21  
**children** (*cobald.composite.uniform.UniformComposite* attribute), 22  
**children** (*cobald.composite.weighted.WeightedComposite* attribute), 22  
**children** (*cobald.interfaces.CompositePool* property), 38  
**cli\_run()** (in module *cobald.daemon.core.main*), 30  
**cobald.composite** module, 21  
**cobald.composite.factory** module, 21  
**cobald.composite.uniform** module, 22  
**cobald.composite.weighted** module, 22  
**cobald.controller** module, 23  
**cobald.controller.linear** module, 23  
**cobald.controller.relative\_supply** module, 23  
**cobald.controller.stepwise** module, 24  
**cobald.controller.switch** module, 26  
**cobald.daemon** module, 26  
**cobald.daemon.config** module, 26  
**cobald.daemon.config.mapping** module, 26  
**cobald.daemon.config.python** module, 28  
**cobald.daemon.config.yaml** module, 28  
**cobald.daemon.core** module, 29  
**cobald.daemon.core.cli** module, 29

- cobald.daemon.core.config
    - module, 29
  - cobald.daemon.core.logger
    - module, 30
  - cobald.daemon.core.main
    - module, 30
  - cobald.daemon.debug
    - module, 35
  - cobald.daemon.runners
    - module, 30
  - cobald.daemon.runners.asyncio\_runner
    - module, 30
  - cobald.daemon.runners.base\_runner
    - module, 31
  - cobald.daemon.runners.guard
    - module, 32
  - cobald.daemon.runners.meta\_runner
    - module, 32
  - cobald.daemon.runners.service
    - module, 33
  - cobald.daemon.runners.thread\_runner
    - module, 34
  - cobald.daemon.runners.trio\_runner
    - module, 34
  - cobald.decorator
    - module, 35
  - cobald.decorator.buffer
    - module, 35
  - cobald.decorator.coarser
    - module, 36
  - cobald.decorator.limiter
    - module, 36
  - cobald.decorator.logger
    - module, 36
  - cobald.decorator.standardiser
    - module, 37
  - cobald.interfaces
    - module, 37
  - cobald.monitor
    - module, 40
  - cobald.monitor.format\_json
    - module, 40
  - cobald.monitor.format\_line
    - module, 40
  - cobald.utility
    - module, 41
  - cobald.utility.primitives
    - module, 41
  - COBaldLoader (class in *cobald.daemon.core.config*), 29
  - CompositePool (class in *cobald.interfaces*), 37
  - ConfigurationError, 26
  - configure\_logging() (in module *cobald.daemon.config.mapping*), 28
  - construct() (*cobald.daemon.config.mapping.Translator* method), 27
  - Controller (class in *cobald.interfaces*), 38
  - ControlRule (in module *cobald.controller.stepwise*), 24
  - create\_handler() (in module *cobald.daemon.core.logger*), 30
  - ctor (*cobald.interfaces.Partial* attribute), 38
- ## D
- demand (*cobald.composite.factory.FactoryPool* property), 21
  - demand (*cobald.composite.uniform.UniformComposite* property), 22
  - demand (*cobald.composite.weighted.WeightedComposite* property), 23
  - demand (*cobald.decorator.buffer.Buffer* attribute), 36
  - demand (*cobald.decorator.logger.Logger* property), 36
  - demand (*cobald.decorator.standardiser.Standardiser* property), 37
  - demand (*cobald.interfaces.CompositePool* property), 38
  - demand (*cobald.interfaces.Pool* property), 39
  - demand (*cobald.interfaces.PoolDecorator* property), 39
  - DemandSwitch (class in *cobald.controller.switch*), 26
  - digest (*cobald.daemon.config.mapping.SectionPlugin* attribute), 27
- ## E
- enforce() (in module *cobald.utility*), 41
  - escape\_field() (in module *cobald.monitor.format\_line*), 41
  - escape\_key() (in module *cobald.monitor.format\_line*), 41
  - exclusive() (in module *cobald.daemon.runners.guard*), 32
  - execute() (*cobald.daemon.runners.service.ServiceRunner* method), 33
- ## F
- FactoryPool (class in *cobald.composite.factory*), 21
  - flavour (*cobald.daemon.runners.asyncio\_runner.AsyncioRunner* attribute), 31
  - flavour (*cobald.daemon.runners.base\_runner.BaseRunner* attribute), 31
  - flavour (*cobald.daemon.runners.thread\_runner.ThreadRunner* attribute), 34
  - flavour (*cobald.daemon.runners.trio\_runner.TrioRunner* attribute), 34
  - format() (*cobald.monitor.format\_json.JsonFormatter* method), 40
  - format() (*cobald.monitor.format\_line.LineProtocolFormatter* method), 40

## G

`get_rule()` (*cobald.controller.stepwise.RangeSelector* method), 24

## I

Indistinguishable Resources, 19

`initialise_logging()` (in module *cobald.daemon.core.logger*), 30

InvariantError, 41

## J

`JsonFormatter` (class in *cobald.monitor.format\_json*), 40

## K

`kwargs` (*cobald.interfaces.Partial* attribute), 38

## L

`leaf` (*cobald.interfaces.Partial* attribute), 38

`line_protocol()` (in module *cobald.monitor.format\_line*), 41

`LinearController` (class in *cobald.controller.linear*), 23

`LineProtocolFormatter` (class in *cobald.monitor.format\_line*), 40

`load()` (*cobald.daemon.config.mapping.SectionPlugin* class method), 27

`load()` (in module *cobald.daemon.core.config*), 29

`load_configuration()` (in module *cobald.daemon.config.mapping*), 28

`load_configuration()` (in module *cobald.daemon.config.python*), 28

`load_configuration()` (in module *cobald.daemon.config.yaml*), 28

`load_name()` (*cobald.daemon.config.mapping.Translator* static method), 27

`load_pipeline()` (in module *cobald.daemon.core.config*), 30

`load_section_plugins()` (in module *cobald.daemon.core.config*), 30

`Logger` (class in *cobald.decorator.logger*), 36

## M

`M` (in module *cobald.daemon.config.mapping*), 27

`manage_payloads()` (*cobald.daemon.runners.asyncio\_runner.AsyncioRunner* method), 31

`manage_payloads()` (*cobald.daemon.runners.base\_runner.BaseRunner* method), 31

`manage_payloads()` (*cobald.daemon.runners.thread\_runner.ThreadRunner* method), 34

`manage_payloads()` (*cobald.daemon.runners.trio\_runner.TrioRunner* method), 35

`MetaRunner` (class in *cobald.daemon.runners.meta\_runner*), 32

module

*cobald.composite*, 21

*cobald.composite.factory*, 21

*cobald.composite.uniform*, 22

*cobald.composite.weighted*, 22

*cobald.controller*, 23

*cobald.controller.linear*, 23

*cobald.controller.relative\_supply*, 23

*cobald.controller.stepwise*, 24

*cobald.controller.switch*, 26

*cobald.daemon*, 26

*cobald.daemon.config*, 26

*cobald.daemon.config.mapping*, 26

*cobald.daemon.config.python*, 28

*cobald.daemon.config.yaml*, 28

*cobald.daemon.core*, 29

*cobald.daemon.core.cli*, 29

*cobald.daemon.core.config*, 29

*cobald.daemon.core.logger*, 30

*cobald.daemon.core.main*, 30

*cobald.daemon.debug*, 35

*cobald.daemon.runners*, 30

*cobald.daemon.runners.asyncio\_runner*, 30

*cobald.daemon.runners.base\_runner*, 31

*cobald.daemon.runners.guard*, 32

*cobald.daemon.runners.meta\_runner*, 32

*cobald.daemon.runners.service*, 33

*cobald.daemon.runners.thread\_runner*, 34

*cobald.daemon.runners.trio\_runner*, 34

*cobald.decorator*, 35

*cobald.decorator.buffer*, 35

*cobald.decorator.coarser*, 36

*cobald.decorator.limiter*, 36

*cobald.decorator.logger*, 36

*cobald.decorator.standardiser*, 37

*cobald.interfaces*, 37

*cobald.monitor*, 40

*cobald.monitor.format\_json*, 40

*cobald.monitor.format\_line*, 40

*cobald.utility*, 41

*cobald.utility.primitives*, 41

## N

`NAME` (*cobald.decorator.logger.Logger* property), 36

`NameRepr` (class in *cobald.daemon.debug*), 35

`BaseRunner`

## O

Opportunistic Resources, 19

OrphanedReturn, 32

`TrioRunner`

## P

`pairwise()` (in module *cobald.utility*), 41

- Partial (class in *cobald.interfaces*), 38
- PipelineTranslator (class in *cobald.daemon.core.config*), 29
- Pool, 19
- Pool (class in *cobald.interfaces*), 39
- PoolDecorator (class in *cobald.interfaces*), 39
- pretty\_module() (in module *cobald.daemon.debug*), 35
- pretty\_partial() (in module *cobald.daemon.debug*), 35
- pretty\_ref() (in module *cobald.daemon.debug*), 35
- ## R
- RangeSelector (class in *cobald.controller.stepwise*), 24
- ready() (cobald.daemon.runners.base\_runner.BaseRunner method), 31
- ready() (cobald.daemon.runners.trio\_runner.TrioRunner method), 35
- register\_payload() (cobald.daemon.runners.asyncio\_runner.AsyncioRunner method), 31
- register\_payload() (cobald.daemon.runners.base\_runner.BaseRunner method), 31
- register\_payload() (cobald.daemon.runners.meta\_runner.MetaRunner method), 32
- register\_payload() (cobald.daemon.runners.thread\_runner.ThreadRunner method), 34
- register\_payload() (cobald.daemon.runners.trio\_runner.TrioRunner method), 35
- regulate() (cobald.controller.linear.LinearController method), 23
- regulate() (cobald.controller.relative\_supply.RelativeSupplyController method), 23
- regulate() (cobald.controller.switch.DemandSwitch method), 26
- RelativeSupplyController (class in *cobald.controller.relative\_supply*), 23
- required (cobald.daemon.config.mapping.SectionPlugin property), 27
- requirements (cobald.daemon.config.mapping.SectionPlugin attribute), 27
- run() (cobald.composite.factory.FactoryPool method), 22
- run() (cobald.controller.linear.LinearController method), 23
- run() (cobald.controller.relative\_supply.RelativeSupplyController method), 24
- run() (cobald.controller.stepwise.Stepwise method), 24
- run() (cobald.controller.switch.DemandSwitch method), 26
- run() (cobald.daemon.runners.base\_runner.BaseRunner method), 31
- run() (cobald.daemon.runners.meta\_runner.MetaRunner method), 32
- run() (cobald.decorator.buffer.Buffer method), 36
- run() (in module *cobald.daemon.core.main*), 30
- run\_payload() (cobald.daemon.runners.asyncio\_runner.AsyncioRunner method), 31
- run\_payload() (cobald.daemon.runners.base\_runner.BaseRunner method), 32
- run\_payload() (cobald.daemon.runners.meta\_runner.MetaRunner method), 32
- run\_payload() (cobald.daemon.runners.thread\_runner.ThreadRunner method), 34
- run\_payload() (cobald.daemon.runners.trio\_runner.TrioRunner method), 35
- runner\_types (cobald.daemon.runners.meta\_runner.MetaRunner attribute), 32
- runners (cobald.daemon.runners.meta\_runner.MetaRunner property), 33
- running (cobald.daemon.runners.service.ServiceUnit property), 33
- runtime (in module *cobald.daemon*), 26
- ## S
- section (cobald.controller.stepwise.UnboundStepwise method), 25
- section (cobald.interfaces.Controller class method), 38
- s() (cobald.interfaces.Pool class method), 39
- s() (cobald.interfaces.PoolDecorator class method), 39
- section (cobald.daemon.config.mapping.SectionPlugin attribute), 27
- SectionPlugin (class in *cobald.daemon.config.mapping*), 27
- service() (in module *cobald.daemon*), 26
- service() (in module *cobald.daemon.runners.service*), 33
- ServiceRunner (class in *cobald.daemon.runners.service*), 33
- ServiceUnit (class in *cobald.daemon.runners.service*), 33
- shutdown() (cobald.daemon.runners.service.ServiceRunner method), 33
- Standardiser (class in *cobald.decorator.standardiser*), 37
- start() (cobald.daemon.runners.service.ServiceUnit method), 33
- Stepwise (class in *cobald.controller.stepwise*), 24
- stepwise (in module *cobald.controller.stepwise*), 26
- stop() (cobald.daemon.runners.base\_runner.BaseRunner method), 32
- stop() (cobald.daemon.runners.meta\_runner.MetaRunner method), 33
- supply (cobald.composite.factory.FactoryPool property), 22
- supply (cobald.composite.uniform.UniformComposite property), 22
- supply (cobald.composite.weighted.WeightedComposite property), 23

supply (*cobald.interfaces.CompositePool property*), 38  
 supply (*cobald.interfaces.Pool property*), 39  
 supply (*cobald.interfaces.PoolDecorator property*), 39

## T

ThreadRunner (class in *cobald.daemon.runners.thread\_runner*),  
 34  
 translate\_hierarchy()  
 (*cobald.daemon.config.mapping.Translator  
 method*), 28  
 translate\_hierarchy()  
 (*cobald.daemon.core.config.PipelineTranslator  
 method*), 29  
 Translator (class in *cobald.daemon.config.mapping*),  
 27  
 TrioRunner (class in *cobald.daemon.runners.trio\_runner*),  
 34

## U

UnboundStepwise (class in *cobald.controller.stepwise*),  
 24  
 UniformComposite (class in  
*cobald.composite.uniform*), 22  
 units() (*cobald.daemon.runners.service.ServiceUnit  
 class method*), 33  
 utilisation (*cobald.composite.factory.FactoryPool  
 property*), 22  
 utilisation (*cobald.composite.uniform.UniformComposite  
 property*), 22  
 utilisation (*cobald.composite.weighted.WeightedComposite  
 property*), 23  
 utilisation (*cobald.interfaces.CompositePool prop-  
 erty*), 38  
 utilisation (*cobald.interfaces.Pool property*), 39  
 utilisation (*cobald.interfaces.PoolDecorator prop-  
 erty*), 39

## W

WeightedComposite (class in  
*cobald.composite.weighted*), 22

## Y

yaml\_constructor() (in  
*cobald.daemon.config.yaml*), 28