dcos-perf-test-driver Documentation

Release 0.1.0

Ioannis Charalampidis

Aug 10, 2018

Contents:

The *DC/OS Performance Test Driver* is a test harness for running scale and performance tests of any component in DC/OS. Because of it’s modular design it’s simple to configure it to match your needs.

CHAPTER 1

Installation

The `dcos-perf-test-driver` can be installed as a standard python module using `pip`. However since the project is not publicly available in PyPI so you will need to point it to the github repository:

```
pip install git+https://github.com/mesosphere/dcos-perf-test-driver
```

If the installation was successful, the `dcos-perf-test-driver` binary should be now available.

Continue to *Usage* for more information regarding the command-line interface or to the *Example* to get more familiar with the tool.

CHAPTER 2

Concepts

Before you start implementing your own performance test you must first ensure that it's structured in the correct way. To start with, keep in mind the following:

1. The tests do not operate on the time-domain but on a user-defined axis(-es)
2. Metrics are collected for every value on the axis

Effectively this means that the performance driver explores a parameter space, on which every point is a test case. To keep things simple you can think that the perf driver is evaluating this function for every axis and metric:

$$f_{metric}(axis) = value$$

Note: In the perf-driver semantics, each axis is called a *parameter*. That's because the values we are exploring are also the test parameters.

Note: *But why are you not using the time domain?*

If you come from the data analytics world, using a user-defined axis might come as a surprise. However if you consider that the `dcos-perf-test-driver` is designed to accurately and reproducibly measure an observation given a test scenario, you will notice that the time series is an unwanted noise.

For example, if you want to only measure how much time it takes to deploy 1000 applications on marathon you don't really care how much did the individual deployment takes. And if you want to measure the same thing for 100, 200, or 2000 applications, collecting and summarizing the data from a time-series database is more trouble than help.

Therefore by requiring the user to define his own axis, the system can abstract the user input as individual test scenarios, allowing it to easily group the results in the respective bins, even if the tests are running in parallel.

2.1 Summarization

You will also encounter cases where you have collected more than one value for the same axis value and metric. This can either happen because you are sampling multiple times during the test case, or because you have configured the tests to run multiple times for more accurate statistics.

In either way, you will end-up with multiple values for the same function:

$$f_{metric_1}(axis_1) = \{value_1, value_2, value_3 ... \quad \}$$

But this violates our initial assumption that the driver collects a single value for every axis/metric combination.

You can collapse these values down to one using a *Summarizer*. As it's name indicates it is going perform an operation to the values and reduce them to a single scalar.

For example, if you are interested on the worst value of the series, you can use the `max` summarizer. This effectively evaluates to:

$$\sum_{max} f_{metric_1}(axis_1) = max(\{value_1, value_2, value_3 ... \quad \})$$

But usually you are interested on the mean value, so you are going to use the `mean` summarizer, or even the `mean_err` summarizer that will also include the uncertainty values for creating error bars.

Note: The summarizer will automatically reject outliers that are not part of a Gaussian distribution. You can disable this behavior by setting `outliers: no` on the *config.metrics* configuration.

The actual implementation of the function f is not relevant in the conceptual level and therefore is abstracted with the *Black Box Abstraction* as described in the next section

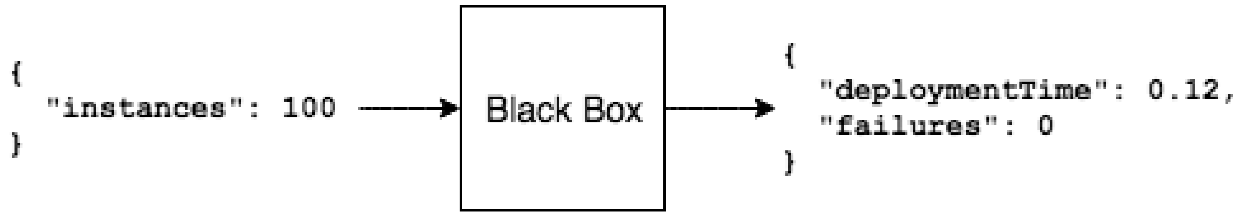
2.2 The Black Box Abstraction

One of the most important terms in the test driver is the *Black Box Abstraction*. This is more like a mental abstraction that helps you design your test in a compatible way, rather than a literal call-out to an unknown tool.

According to this, your test case is always considered a black box with a standardized input and output. A few important things must be noted:

1. Changing something in the input starts a new test, for which some measurements are performed and an output is produced.
2. If the same input is given to the black box, the same output is expected.
3. A black box can take an arbitrary time to complete. In the same time other black boxes could run in parallel.

The **input** and the **output** to the black box is always a dictionary of scalar values:


```
{
  "instances": 100
}
```

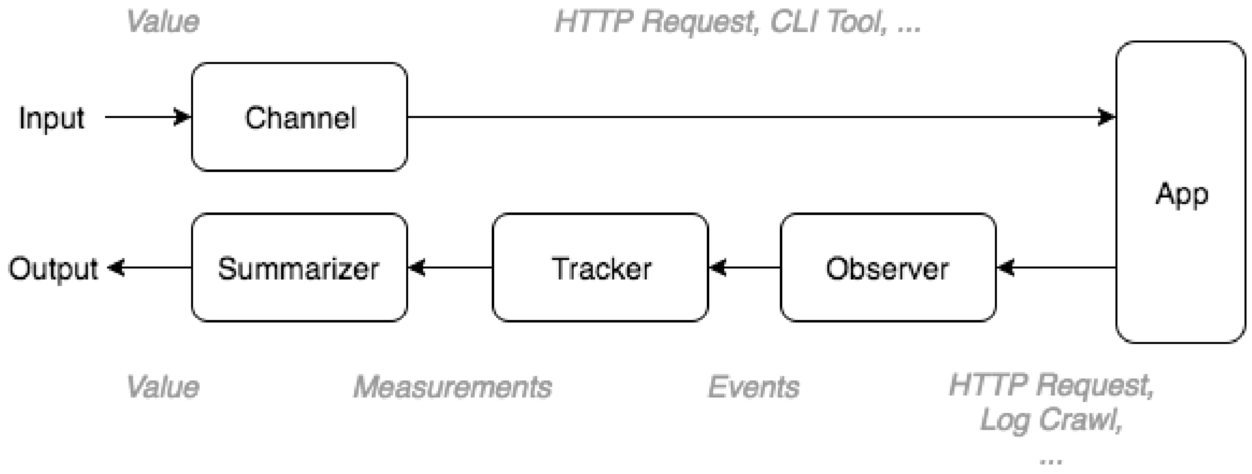
```
{
  "deploymentTime": 0.12,
  "failures": 0
}
```

It's important to note that even though the *Black Box Abstraction* provides a clean mental separation, it's implementation is not that trivial. The next section describes how this is implemented in the driver.

2.2.1 Implementation

The *Black Box Abstraction* is internally implemented as a set of different components. As described in the *Architecture*, these components have no dependencies and are statically configured, therefore it's important to understand what's their functionality in order to be properly configured.

The components taking part in the *Black Box Abstraction* are the following:



1. Every time a value is changed the **Channel** associated with this value is triggered. It's purpose is to apply this change to the application being tested. Values can be applied:
   - By HTTP requests (ex. every time a parameter changes, make an HTTP request)
   - By Calling-out to a command-line tool
   - By re-launching the application being tested with new command-line arguments
2. At the same time an **Observer** starts collecting useful events from the application. Such events could be:
   - Events dispatched by the application over an event bus (ex. WebSocket)
   - Events synthesized via polling (ex. life cycle events of an HTTP request, such as `Started`, `Request Sent`, `First Bytes Arrived` and `Completed`)
   - Events synthesized via log crawling
   - Events that carry updates on application-exposed metrics (ex. by polling a `/metrics` endpoint and publishing an event every time a metric changes it's value)
3. A **Tracker** listens for all relevant events in the bus and calculates the metrics. For example:
   - Number of events observed
   - Duration between events
   - Extract values carried by the events (ex. timestamp)
4. A **Summarizer** collects all the metrics produced by the *Tracker* and calculates a single, "summarized" value. A summarizer could calculate:
   - The minimum value
   - The maximum value
   - The average of the values
   - The average of the values, including uncertainty values (error bars)

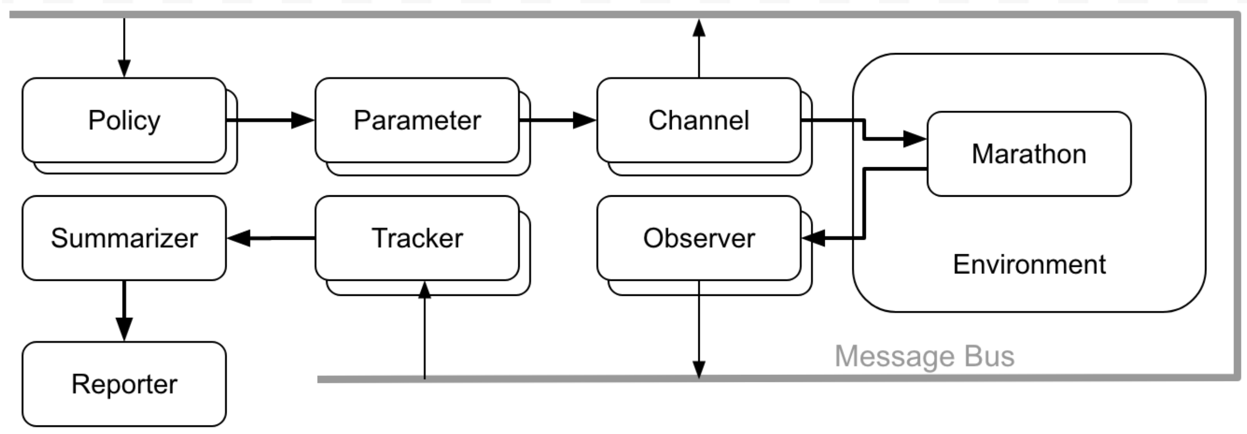
You can refer to the *Example* to see how you can configure these components and see them in action.

CHAPTER 3

Architecture

The *DC/OS Performance Test Driver* is design to be modular and extensible in order to adapt to the needs of every interested party.

It is composed of a set of individual components that all plug on a shared event bus and communicate to each other purely through messages.



According to their main function they are separated in the following types:

- A **Policy** controls the evolution of the test
- A **Channel** applies the changes that occurred to the parameters by the policy to the application being tested
- An **Observer** monitors the application being tested and broadcasts useful events to the event bus
- A **Tracker** is listening for events in the event bus and extracts or calculates useful values from them
- A **Summarizer** collects the values calculated by the tracker, groups them by test case and calculates the min/max/average or other summarization values

You may want to read the *Concepts* section on the reasoning behind this separation or the *Example* section to see how they work in action.

3.1 Instantiating Components

Since there are no cross-component dependencies they can be instantiated and plugged into the bus when needed. Only a static argument configuration would be required that is going to configure it's behavior.

If we were to write it in a python code we would have written something like so:

```
bus.plug(
  HttpChannel(
    url="http://127.0.0.1:8080/some/api",
    method="POST",
    body='{"hello": "rest"}'
  )
)
```

But since all the components are instantiated in the same way we can can completely avoid using code and express the same thing in a YAML block like so:

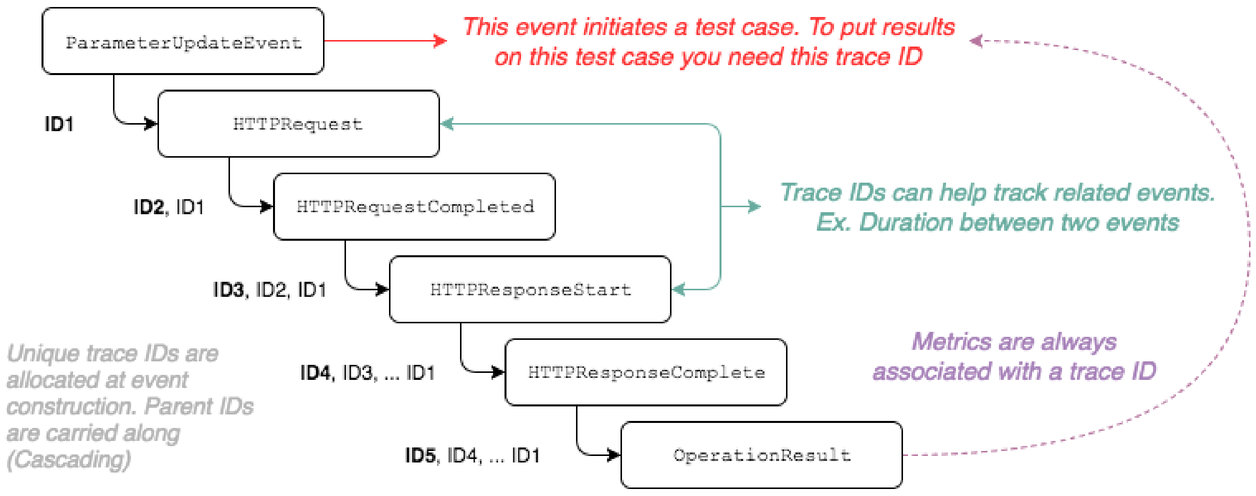
```
- class: HttpChannel
  url: http://127.0.0.1:8080/some/api
  method: POST
  body: |
    {
      "hello": "rest"
    }
```

That's why the *DC/OS Performance Test Driver* is using YAML files for it's configuration.

3.2 Event Cascading

Since everything in the *DC/OS Performance Test Driver* is orchestrated through messages it's important to identify the test case each event belogs into. For this reason the driver is using *Event Cascading* as the means to describe which event was emitted as a response to another.

Take the following diagram for example:


ParameterUpdateEvent
This event initiates a test case. To put results on this test case you need this trace ID
ID1
HTTPRequest
ID2, ID1
HTTPRequestCompleted
Trace IDs can help track related events. Ex. Duration between two events
ID3, ID2, ID1
HTTPResponseStart
Unique trace IDs are allocated at event construction. Parent IDs are carried along (Cascading)
ID4, ID3, ... ID1
HTTPResponseComplete
Metrics are always associated with a trace ID
ID5, ID4, ... ID1
OperationResult

The *Event Cascading* is implemented by assigning unique IDs (called `traceids`) to every event and carrying the related event IDs along when an event is emitted as a response (or in relation) to another.

Usually, the *root* event is the `ParameterUpdateEvent` that is emitted when a test case is initiated and the parameter values are defined. Every other event that takes place in the test is carrying this ID.

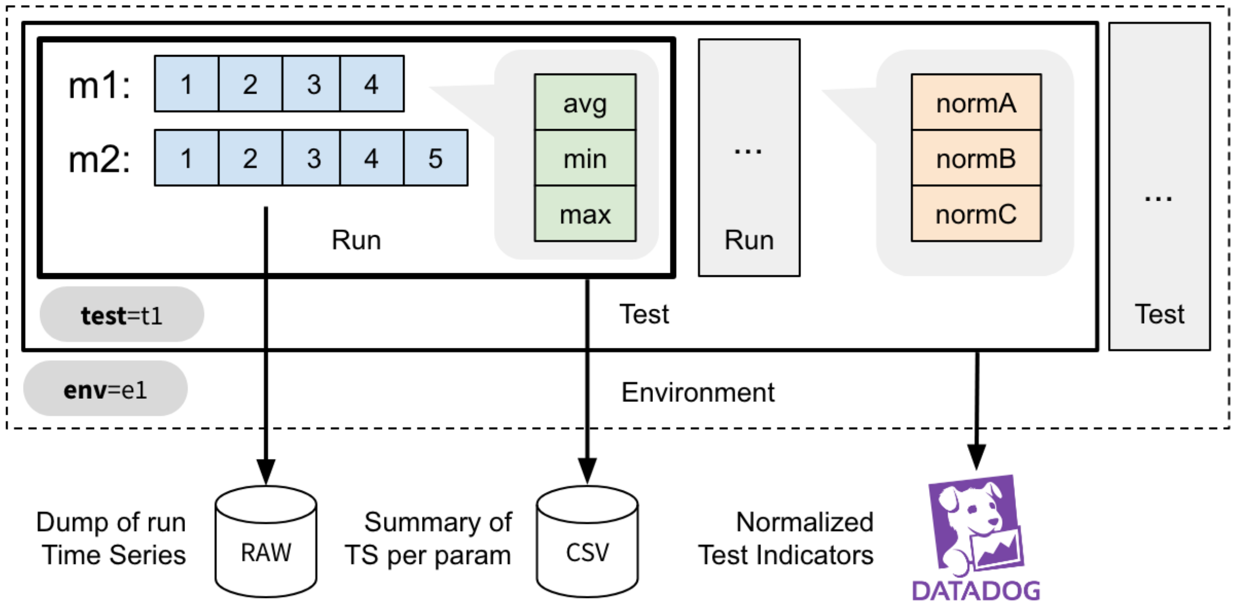
Warning: If you are seeing unexpected results or behaviors it's most probable that you have not taken into account the event tracing.

Important: If you are a component developer you should always carry along the correct traceids when you are publishing events. As a rule of thumb you should take care of the following two cases:

1. Actions that are triggered by an event should always publish events that carry along the traceids from the originating event.
2. Actions that are not part of an event chain should publish events that carry along the traceids from the latest `ParameterUpateEvent` observed.

3.3 Processing the metrics

In `dcos-perf-test-driver` the metric values are produced by the **Trackers** and archived the moment they are emmmited and archived into an array of time series values.



3.3.1 Test Phase

In `dcos-perf-test-driver` the performance tests are executed for each parameter combination in question and are repeated for one or more times in order to increase the statistics.

Each time the actual performance test is executed, a **Test Phase** is initiated.

3.3.2 Timeseries

For every phase one or more metrics are being collected. The moment a metric is sampled it's placed in an in-memory time-series record.

This record is unique for the every parameter combination, effectively creating an *Axis - Values* representation.

These time series are further summarised or post-processed according to the needs of the reporter.

3.3.3 Summarized Values

When a phase is completed, the timeseries values are summarised using one or more summarisers, as defined in the *config.metrics* configuration parameter.

Calculating the summarised values makes the results visualizable. For instance, you can pick a metric and a parameter and easily create an 1D plot with the data.

3.3.4 Indicators

The indicators scalar values that describe the overall outcome of the test as a single number. They are useful to detect deviations from the previous results and to raise alerts.

For instance, you can normalize the value of the time each marathon deployment takes against the number of applications you instructed to scale to, thus creating the `meanDeploymentTimePerApp` indicator.

CHAPTER 4

Example

Since it's difficult to understand how the system works just by looking to the Architecture Diagram, we are going to present a complete example.

1. We are going to start by describing how we created the tests based on the observations we want to make.
2. Then we are describing how our decisions are written down into configuration files.
3. And we are going to end by describing how the test is actually executed within the test driver.

4.1 The testing scenario

We have a marathon service that runs on `127.0.0.1:8080` and we want to see how well the endpoint `/v2/groups` is responding as the load to the `/v2/apps` endpoint increases.

Having read the *Concepts* we decided that our axis is going to be the "deployments per second", so `deploymentRate` and we are going to explore the values from 100 to 1000 with an interval of 50.

To make sure that we are operating on a clean slate every time we also agreed that we should wait for the previous deployments to complete before starting the new ones.

In addition, we decided that we are going to measure how long an HTTP request to the `/v2/groups` endpoint takes, so our metric is the `responseTime` in seconds.

Effectively we want to measure:

$$f_{responseTime}(\{100, 150, ...1000\})$$

Finally, since we will be constantly sampling the groups endpoint we are most certainly going to collect more than one value for the same test case, so we are going to use the `mean_err` summarizer to collect the mean values with uncertainty information.

Having defined the `parameters` and the `metrics` in a conceptual level we can already write them down in the configuration file.

Since they do not target any particular component they are defined in the *global* configuration section like so:

```
config:

  # Test parameters
  parameters:
    - name: deploymentRate
      units: depl/s
      desc: The number of deployments per second

  # Test metrics
  metrics:
    - name: responseTime
      units: sec
      desc: The time for an HTTP request to complete
      summarize: [mean_err]
```

You can refer to the *Global Configuration Statements* for more details on what fields you can use and what's their interpretation.

Note: In the above example, the `summarize` field is using the compact expression for a built-in summarizer. The equivalent full representation would be the following:

```
- name: responseTime
  ..
  summarize:
    - class: "@mean_err"
```

The full representation allows you to customize them even further, providing for example a different name (ex. for the plots) or turning off the automatic outliers rejection.

```
- name: responseTime
  ..
  summarize:
    - class: "@mean_err"
      name: "Mean (With Error)"
      outliers: no
```

4.2 Configuring our black box

According to the *The Black Box Abstraction* we have to configure the components that are going to apply the changes to marathon and collect the measurements.

4.2.1 Input

We are going to start by implementing the `input` direction of our black box, and more specifically we are going to figure out which *Channel* are we going to use for applying the changes to marathon.

As we described above we need to make `deploymentRate`-requests per second. Browsing through the *Channels* reference we notice the *HTTPChannel*. According to it's documentation, it "performs an HTTP request every time a parameter changes".

We also notice that it accepts a `repeat` parameter, that is repeating the same request multiple times.

By copying the fields of interest from the reference and using the correct *Macros* we compose the following configuration fragment:

```
channels:
  - class: channel.HTTPChannel
    url: http://127.0.0.1:8080/v2/apps
    verb: POST
    repeat: "{{deploymentRate}}"
    body: |
      {
        "id": "/scale-instances/{{uuid()}}",
        "cmd": "sleep 1200",
        "cpus": 0.1,
        "mem": 64,
        "disk": 0,
        "instances": 0,
        "backoffFactor": 1.0,
        "backoffSeconds": 0
      }
```

This instantiates a `HTTPChannel` class that is going to perform an HTTP POST to the endpoint `http://127.0.0.1:8080/v2/apps` every time the value of a macro changes. In our case, the `deploymentRate`.

In addition, it is going to repeat this request "deploymentRate" times. This means 100 times on the first run, 150 on the second etc. For the sake of the example let's assume that all 1000 requests will be posted within a second so we don't have to take any other action for satisfying the "per second" part of the test scenario.

Note: The automatic triggering of the channel when a macro changes is a bit of a "magic" behavior only for the channel configuration. It can be configured using the trigger syntax as described in *Channel Triggers*.

4.2.2 Output

We are now going to implement the `output` of our black box. As seen in the *The Black Box Abstraction* diagram we need to define an *Observer*, a *Tracker* and a *Summarizer*. But let's see in detail what they are about.

From our test scenario, we want to measure "how long an HTTP request to the `/v2/groups` endpoint takes". Thus we need to plug an appropriate component to perform this request.

We know from the documentation that the components that makes observations to the application being tested are the **Observers**. By looking on the *Observers* reference page we find out that the *HTTPTimingObserver* is particularly useful in our case.

We start by copying the example from the documentation page, removing the fields we don't need and modifying the values according to our needs

```
observers:
  - class: observer.HTTPTimingObserver
    url: http://127.0.0.1:8080/v2/groups
    interval: 1
```

That's it. Now while our tests are running the `HTTPTimingObserver` is going to poll the `/v2/groups` endpoint every second. Looking into the *Event Reference* we see that this observer broadcasts the `HTTPTimingResultEvent` when a measurement is completed.

Next, we have to define a **Tracker** that is going to convert the observed events into measurements. In our case we just need to extract the fields of interest from the `HTTPTimingResultEvent` event. Again, by looking to the *Trackers* reference we see that *EventAttributeTracker* is what we need.

Again, we copy the example and adjust the values to our needs:

```
trackers:
  - class: tracker.EventAttributeTracker
    event: HTTPTimingResultEvent
    extract:
      - metric: responseTime
        attrib: responseTime
```

Note: This might be a bit difficult to digest at a first glimpse, but it's quite easy after you understand what it does:

1. It waits until a `HTTPTimingResultEvent` is dispatched in the bus
2. It extracts the `responseTime` attribute from the event
3. It stores it as a value for the `responseTime` metric that we defined on the first step.

Note: Not all events have fields. However for the ones that have, the *Event Reference* listing contains everything you will need to know.

Finally, you will notice that we have already defined our **Summarizer** when we defined the metric on the first step. It's configuration belongs on the global section because it's annotating the metric.

Having our black box defined we are going to continue with defining the parameter evolution policy on the next step.

4.3 Defining the axis evolution

As we previously mentioned, we want the `deploymentRate` to increase gradually from 100 to 1000 with an interval of 50. But *when* are we advancing to the next?

Answering this question will help us pick the policy are we going to use. In principle we will need to read the *Policies* class reference and pick the most fitting policy for our case, but briefly we could say:

1. Do we advance to the next value at fixed time intervals (ex. every minute)? Then we are going to use a *TimeEvolutionPolicy*.
2. Do we advance to the next value when a well-described case is met? Then we are going to use the *MultiStepPolicy*.

In our case we don't want to overload the system, so we cannot use fixed timed intervals since an operation might take longer than expected. So we are going to use the *MultiStepPolicy*.

Note: We are choosing *MultiStepPolicy* in favor of *MultivariableExplorerPolicy*, even though they are very close on their features, because the former exposes a more elaborate configuration.

Now let's answer the other question: Which is the "well-described" case that should be met before advancing to the next value?

In our example we are going to wait until all the deployments have completed. To achieve this we are going to wait until the correct number of the appropriate events is received.

Let's start first by copying the example configuration from the `MultiStepPolicy` and let's keep only the `steps` section for now. We are going to keep only one step. Following the examples, we are using the min/max/step configuration for the `deploymentRate`.

```
policies:
  - class: policy.MultiStepPolicy
    steps:

      # Explore deploymentRate from 100 to 1000 with interval 50
      - name: Stress-Testing Marathon
        values:
          - parameter: deploymentRate
            min: 100
            max : 1000
            step: 50
```

Technically, our policy is now syntactically correct. However, if you try to run it you will notice that it will scan full range of options as fast as possible. That's not what we want.

We notice on the `MultiStepPolicy` documentation the `events` section, and in particular the `events.advance` event. That's exactly what we want, but what event are we going to to listen for?

Let's consider what components do we currently have that are broadcasting events:

1. We have an `HTTPChannel` that broadcasts HTTP life cycle events, such as `HTTPRequestStartEvent`, `HTTPRequestEndEvent`, `HTTPResponseStartEvent` and `HTTPResponseEndEvent` – Not interesting.
2. We have an `HTTPTimingObserver` that broadcasts the measurement `HTTPTimingResultEvent` event – Not interesting.
3. We have the `MultiStepPolicy` that broadcasts the `ParameterUpdatedEvent` – Not interesting.

So it looks that we are going to need a new observer. Going back to the *Observers* we notice the *MarathonPollerObserver*. From it's documentation we see that it subscribes to the marathon SSE event stream and brings in the marathon events. More specifically, the `MarathonDeploymentSuccessEvent` that we need. That's perfect!

Again, we copy the example from the documentation and we adjust to our needs

```
observers:
 ...

 - class: observer.MarathonPollerObserver
   url: "http://127.0.0.1:8080"
```

Now that we have our observer in place, let's go back to our policy configuration and let's add an `events` section with an `advance` field, pointing to the `MarathonDeploymentSuccessEvent` event:

```
policies:
  - class: policy.MultiStepPolicy
    steps:

      # Explore deploymentRate from 100 to 1000 with interval 50
      - name: Stress-Testing Marathon
        values:
          - parameter: deploymentRate
            min: 100
            max : 1000
            step: 50
        # Advance when the deployment is successful
        events:
          advance: MarathonDeploymentSuccessEvent:notrace
```

Note the `:notrace` suffix of the event. We are using an *Event Filters* syntax to instruct the policy to ignore tracing due to *Event Cascading*, since the policy does not have enough information to trace the `MarathonDeploymentSuccessEvent` and all these events will be ignored.

Note: You may wonder when you should use `:notrace` and when not. In principle you should always check the component documentation if the events it emits are properly cascaded and which are the event(s) they require in order to properly trace it. If you are properly using them you should never have to use `:notrace`.

However there are also cases where the events you are waiting for do not belong on a trace. For example, the `TickEvent` is sent 30 times per second, but it does not belong on a trace. Therefore if we don't use `:notrace` all of them will be filtered out.

In our particular case, the *MarathonPollerObserver* requires the deployments to be started using a *MarathonDeployChannel* or a *MarathonUpdateChannel*, since it is listening for `MarathonDeploymentRequestedEvent` events in order to extract the ID of the app/pod/group being deployed and link it to the appropriate status update event.

If you test the policy now you will notice that it's indeed waiting for the first deployment success event to arrive, but this is again not what we need.

We should wait until all the requests from the current test cases are handled. Effectively this means waiting for *deploymentRate* number of events. This can be easily defined using the `advance_condition` section and the `events` section:

```
policies:
  - class: policy.MultiStepPolicy
    steps:

      # Explore deploymentRate from 100 to 1000 with interval 50
      - name: Stress-Testing Marathon
        values:
          - parameter: deploymentRate
            min: 100
            max : 1000
            step: 50

        # Advance when the deployment is successful
        events:
          advance: MarathonDeploymentSuccessEvent

        # Advance only when we have received <deploymentRate> events
        advance_condition:
          events: "deploymentRate"
```

Note: You might wonder why we are not using the macro `{deploymentRate}` but we rather used the literal `deploymentRate`?

That's because according to the documentation this value can be any valid python expression where the parameter values and the already existing definitions are available in the globals.

This allows you to have more elaborate advance conditions, such as: `deploymentRate / 3` or `2 *` deploymentRate.

4.4 Ensuring state integrity

If we try to mentally process the series of actions that are going to be taken when the tests are running, you will notice that each test case is deploying some apps but they are never removed.

This means that we do not operate always on a clean marathon state. To mitigate this we need to invoke an *one-time action* in between the tests. These actions are called `tasks` and you can find a list of them in the *Tasks* reference.

We notice that the *marathon.RemoveGroup* task can come in handy, since we are deploying apps inside the same group. We also read on the table on the top of the page that we should trigger this task between the value changes. So we should register the task on the `intertest` trigger.

Again, we copy the example configuration and we modify it to our needs:

```
tasks:
  - class: tasks.marathon.RemoveGroup
    at: intertest
    url: "http://127.0.0.1:8080"
    group: "/scale-instances"
```

Note: Note that with the *MultiStepPolicy* you can also customize further when some triggers are called. For example, if you want the `RemoveGroup` task to be executed *Before* each time the value is changed (the default is *After*), you can use the respective `tasks` section on it's configuration:

```
policies:
  - class: policy.MultiStepPolicy
    steps:

      # Explore deploymentRate from 100 to 1000 with interval 50
      - name: Stress-Testing Marathon
        ...

        # Fire "prevalue" trigger before changing the value
        tasks:
          pre_value: prevalue

tasks:

  # Register the RemoveGroup to be triggered on "prevalue"
  - class: tasks.marathon.RemoveGroup
    at: prevalue
    url: "http://127.0.0.1:8080"
    group: "scale-instances"
```

4.5 Reporting the results

Now that we have completed the test configuration it's time to describe how and where the results will be collected.

The test driver has a variety of reporters that we can choose from. You can see all of them in the *Reporters* reference. However there is a handful that you are going frequently use. These are the reporters that we are going to plug in our example.

4.5.1 Plots

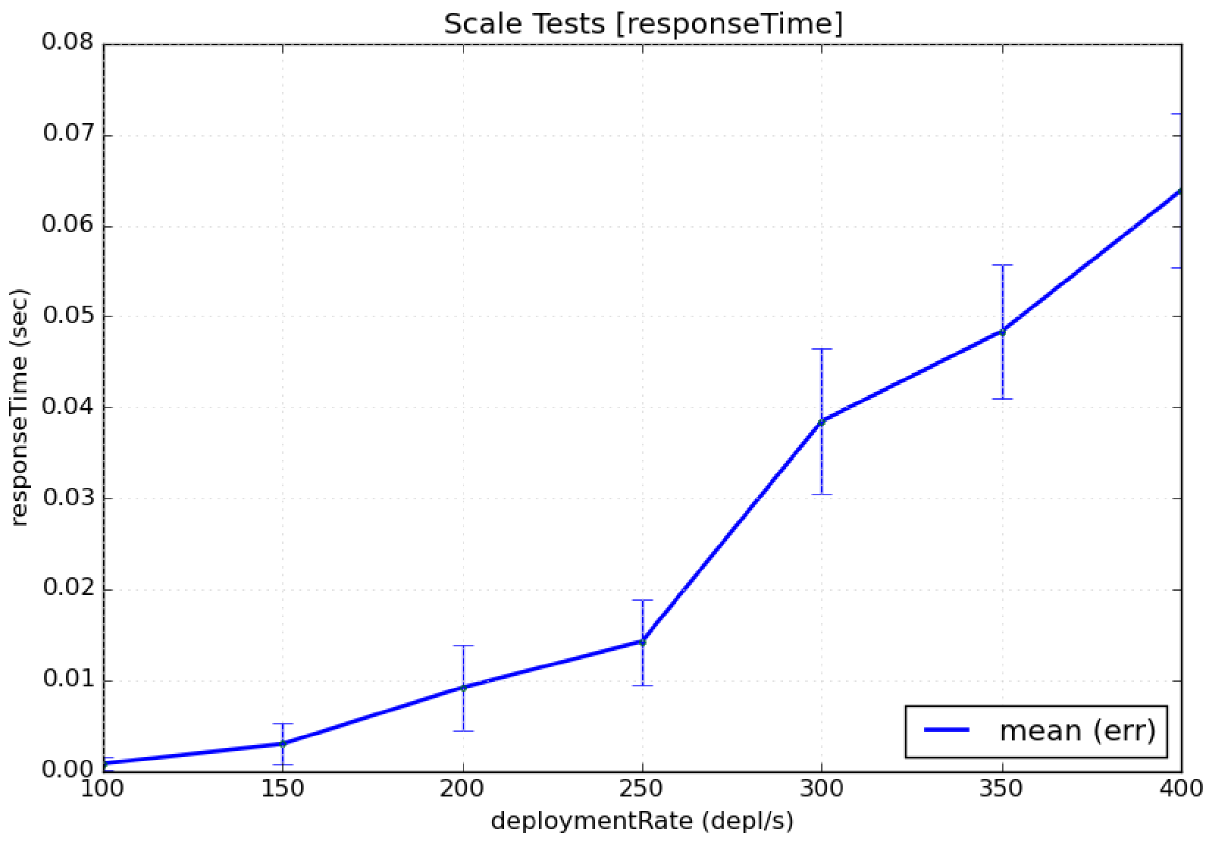
First of all, we are interested into getting some visual feedback with the results. The test driver provides a *PlotReporter* that can be used in this scenario.

This reporter visualizes the *Summarized* results on a plot where the axis is the test parameters and the values are the measured results. An image will be generated for every metric in the configuration.

We noticed that all the parameters of the plot reporter are optional so we are not going to include any. This is as simple as:

```
reporters:
  - class: reporter.PlotReporter
```

This would yield a `plot-responseTime.png` file that looks like this:



Note: The plot reporter can also visualize data in two axes. In this case a 2D plot would be used instead.

Note: The plot reporter will not function with more than two axes. That's because it's not possible to visualize more than two-dimensional data on a static image.

4.5.2 Machine-Readable Data

Of course plots are easy to read, but usually you would need the data to be available in a machine-processable format. You can choose between two options:

- The *CSVReporter* produces a comma-separated-values (CSV) file with the parameter values and the summarized results
- The *RawReporter* produces a detailed JSON dump that includes everything that you would need for processing or reproducing the tests.

Since we want to be verbose, we are going to plug a `RawReporter`:

```
reporters:
  - class: reporter.RawReporter
    filename: "results-raw.json"
```

Note: Having the results collected in a raw dump you can later use the `dcos-compare-tool` to compare runs.

4.6 Indicators

Let's say that you are running this performance test in a CI environment and you want to see the evolution of the measurements over time. What data would you submit to a time-series database?

Submitting the entire plot for every run is rather unhelpful, since you will end up with too many data and you will need to come up with an elaborate data summarization during post-processing.

Instead, you can pre-calculate a summarized value from all the observations of every metric. You can achieve this using the *Indicators*.

An indicator receives both metrics and parameters for every test case and calculates a single scalar value that that carries some meaningful information from the entire run.

A frequently used one is the *NormalizedMeanMetricIndicator*. This indicator normalizes the summarized value of every test case and calculates the mean of all these values.

You could say that for every value of `axis_1` and every respective measurement of `metric_1`, summarized using the `sum_1` summarizer (ex. `mean_err`), the indicator can be expressed as:

$$indicator = \frac{1}{n} \cdot \sum_{axis_1=[...]}^{n} \left(\frac{\sum_{sum_1} f_{metric_1}(axis_1)}{axis_1} \right)$$

In our example, we would like to know what's the average time it took for every instance to be deployed. For this, we are going to calculate:

- The mean value of every deployment measurement (as we already do above)
- Then divide it (aka *normalize it*) by the number of apps being deployed
- Then calculate the mean of all the above measurements

This can be achieved using the *NormalizedMeanMetricIndicator* like so. Note that just like `parameters` and `metrics`, the `indicators` belong on the global configuration:

```
config:
  ...
  indicators:
    # Calculate `meanResponseTime` by calculating the normalizing average
    # of all the `responseTime` mean values, normalized against the current
    # deploymentRate
    - name: meanResponseTime
      class: indicator.NormalizedMeanMetricIndicator
      metric: responseTime.mean_err
      normalizeto: deploymentRate
```

4.7 Increasing our statistics

Finally, like with every statistical problem, you will most probably need to repeat your tests until you have enough statistics.

This can be easily configured with the `repeat` parameter in the global configuration section:

```
config:

  # Repeat this test 5 times
  repeat: 5
```

4.8 Parameterizing your configuration

You might notice that we are frequently repeating the base marathon URL `http://127.0.0.1:8080`. To avoid this repetition we could use *Macros*.

A macro is an expression contained in double brackets, such as `{{marathon_url}}`. At run-time this macro would be replaced with the contents of the *Definition* with the same name. For example we can change our observers like so:

```
observers:

  # Replace http://127.0.0.1:8080 with {{marathon_url}}
  - class: observer.HTTPTimingObserver
    url: {{marathon_url}}/v2/groups
    interval: 1

  # Also replace http://127.0.0.1:8080 with {{marathon_url}}
  - class: observer.MarathonPollerObserver
    url: "{{marathon_url}}"
```

The value for the macro can either be defined using a *define* statement like so:

```
define:
  marathon_url: http://127.0.0.1:8080
```

Or provided by the command-line, like so:

```
~$ dcos-perf-test-driver -Dmarathon_url=http://127.0.0.1:8080
```

Note: Even though it is possible to use the above command-line as-is, it's recommended to use the *config.definitions* section to define which definitions can be provided from the command line.

For example, using:

```
config:
  ...
  definitions:
    - name: marathon_url
      desc: The URL to marathon to use
      required: yes
```

This way, if the user does not provide the `marathon_url` definition, the driver will exit with an error, instructing the user to provide a value instead of silently ignoring it.

4.9 Running the tests

By now your configuration file should look something like the one found in the *Configuration Example*.

Assuming that you have saved it under the name `scale-tests.yml` you can launch it like so:

```
~$ dcos-perf-test-driver ./scale-tests.yml
```

If you observe a behaviour that you don't expect, you can also run the driver in verbose mode. In this mode you will also see the verbose, debug messages that could be helpful to troubleshoot some problems:

```
~$ dcos-perf-test-driver --verbose ./scale-tests.yml
```

Check the *Usage* section for more details on the command-line

CHAPTER 5

Usage

The DC/OS Scale Test Driver is accepting one or more configuration files as it's only positional argument. The *Configuration* files describe which classes to activate and how to run the tests.

In addition, some parameters or metadata can be added as optional arguments via the *–define* and *–meta* flags.

```
usage: dcos-perf-test-driver [-h] [-r RESULTS] [-v] [-D DEFS] [-M META]
                             [config [config ...]]

The DC/OS Performance Tests Driver.

positional arguments:
  config                The configuration script to use.

optional arguments:
  -h, --help            show this help message and exit
  -r RESULTS, --results RESULTS
                        The directory where to collect the results into
                        (default "results")
  -v, --verbose         Show verbose messages for every operation
  -D DEFS, --define DEFS
                        Define one or more macro values for the tests.
  -M META, --meta META  Define one or more metadata value.
```

5.1 –define

```
dcos-perf-test-driver [ --define name1=value1 | -D name2=value2 ] -D ...
```

The `--define` or `-D` argument is defining the value of one or more :ref:

5.2 –meta

```
dcos-perf-test-driver [ --meta name1=value1 | -M name2=value2 ] -D ...
```

The `--meta` or `-D` argument is values for one or more metadata. Such metadata will be part of the final results and can also be defined through the *config.meta* configuration section.

Command-line metadata definition have higher priority than metadata defined in the configuration file.

5.3 –results

```
dcos-perf-test-driver [ --results path/to/results | -r path/to/results ]
```

The `--results` or `-r` argument specifies the location of the results folder to use. If missing the `./results` folder will be used.

5.4 –verbose

```
dcos-perf-test-driver [ --verbose | -v ]
```

The `--verbose` or `-v` argument enables full reporting on the actions being performed by the driver. In addition, this flag will expand all exceptions to the full stack trace instead of only their title.

CHAPTER 6

Cookbook

This cookbook contains a variety of copy-paste-able snippets to help you quickly compose your configuration file.

Compose it picking:

1. *General Section Recipes*
2. *Channel Recipes*
3. *Observer Recipes*
4. *Tracker Recipes*
5. *Policy Recipes*
6. *Tasks Recipes*

6.1 General Section Recipes

How to populate your *config:* section.

6.1.1 General Boilerplate

```
#
# General test configuration
#
config:
  title: "My Test Title"
  repeat: 3

  # Test parameters
  parameters:
    - name: parameter1
      desc: "Description of the parameter"
      units: units
  # Test metrics
  metrics:
    - name: metric1
      desc: "Description of the metric"
      units: units
      summarize: [mean_err]

  # [Optional] Test indicators
  indicators:
    - name: mean_metric1
      class: indicator.NormalizedMeanMetricIndicator
      metric: metric1.mean_err
      normalizeto: "parameter1"
```

6.1.2 Parameter Boilerplate

```
config:
  parameters:
    # ...

    # Test parameter
    - name: parameter1
      desc: "Description of the parameter"
      units: units
```

6.1.3 Metric Boilerplate

```
config:
  metrics:
    # ...

    # Test metric
    - name: metric1
      desc: "Description of the metric"
      units: units
      summarize: [mean_err]
```

6.1.4 Summarizer Boilerplate

Extended format of *config.metrics* with the frequently used `mean_err` summarizer and a custom summarizer name.

```
config:
  metrics:
    - name: metric
    # ...

      # Complete syntax of a metric summarizer
      summarize:
        - class: "@mean_err"
          name: "Mean"
          outliers: yes
```

6.1.5 Indicator Boilerplate

```
config:
  # ...

  # Test indicator
  - name: mean_metric1
    class: indicator.NormalizedMeanMetricIndicator
    metric: metric1.mean_err
    normalizeto: "parameter1"
```

6.1.6 Required command-line definition

The following snippet will require the user to define the specified definition from the command-line:

```
config:
  # ...

  definitions:
    - name: secret
      desc: The secret password to use
      required: yes
```

6.2 Channel Recipes

When a policy changes a parameter a channel takes an action to apply the new value on the application being observed.

The recipes here refer to **when a parameter changes…**

6.2.1 (Re)start an external app with a new command-line

This snippet will call out to the given application when a parameter changes. If the application is still running when a parameter update arrives, the previous instance of the application will be killed:

```
channels:
  # ...

  - class: channel.CmdlineChannel
    restart: no
    shell: no

    # The command-line to execute
    cmdline: "path/to/app --args {{parameter_1}}"

    # [Optional] The standard input to send to the application
    stdin: |
      some arbitrary payload with {{macros}}
      in it's body.

    # [Optional] Environment variables to define
    env:
      variable: value
      other: "value with {{macros}}"
```

6.2.2 Deploy an app on marathon

Deploy a marathon app every time a parameter changes:

```
channels:
  - class: channel.MarathonUpdateChannel
    # The base url to marathon
    url: "{{marathon_url}}"

    # Our one deployment
    deploy:
      - type: app
        spec: |
          {
            "id": "deployment",
            "instances": "{{parameter1}}"
          }
```

6.2.3 Deploy multiple apps on marathon

Deploy a variety of apps every time a parameter changes:

```
channels:
  - class: channel.MarathonUpdateChannel
    # The base url to marathon
    url: "{{marathon_url}}"

    # Our multiple deployments
    deploy:
      - type: app
        spec: |
          {
            "id": "deployment1",
            "instances": "{{parameter1}}"
          }

      - type: app
        spec: |
          {
            "id": "deployment2",
            "instances": "{{parameter1}}"
          }

      - type: app
        spec: |
          {
            "id": "deployment3",
            "instances": "{{parameter1}}"
          }
```

6.2.4 Deploy a group of apps on marathon

Deploy a group of apps every time a parameter changes:

```
channels:
  - class: channel.MarathonUpdateChannel
    # The base url to marathon
    url: "{{marathon_url}}"

    # Our one deployment
    deploy:
      - type: group
        spec: |
          {
            "id": "/apps",
            "apps": [
              {
                "id": "/apps/app1",
                "instances": "{{parameter1}}"
              },
              {
                "id": "/apps/app2",
                "instances": "{{parameter1}}"
              }
            ]
          }
```

6.2.5 Update an app on marathon

Update an existing application on marathon:

```
- class: channel.MarathonUpdateChannel
  url: "{{marathon_url}}"
  update:
    - action: patch_app

      # Update up to 10 instances
      limit: 10

      # Update only apps matching the regex
      filter: "^/groups/variable_"

      # Update the given properties
      patch:
        env:
          PARAMETER_VALUE: "{{parameter1}}"
```

6.2.6 Perform an HTTP request

Perform an arbitrary HTTP request every time a parameter changes:

```
channels:
  - class: channel.HTTPChannel

    # The URL to send the requests at
    url: http://127.0.0.1:8080/v2/apps

    # The body of the HTTP request
    body: |
      {
        "cmd": "sleep 1200",
        "cpus": 0.1,
        "mem": 64,
        "disk": 0,
        "instances": {{instances}},
        "id": "/scale-instances/{{uuid()}}",
        "backoffFactor": 1.0,
        "backoffSeconds": 0
      }

    # [Optional] The HTTP Verb to use (Defaults to 'GET')
    verb: POST

    # [Optional] The HTTP headers to send
    headers:
      Accept: text/plain
```

6.2.7 Perform multiple HTTP requests

You can also repeat the HTTP requests using the *repeat* statement:

TODO: Implement this

6.3 Observer Recipes

TODO: Implement this

6.4 Tracker Recipes

TODO: Implement this

6.5 Policy Recipes

TODO: Implement this

6.6 Tasks Recipes

TODO: Implement this

6.7 Advanced Recipes

This section contains various copy-paste-friendly YAML recipes for addressing frequently-encountered problems.

6.7.1 Launching an app, not part of the test

Some times you might want to launch an application that is going to run for the duration of the test but it's not active part of the test.

To launch such applications you can use a *CmdlineChannel* with the following configuration:

```
channels:
  - class: channel.CmdlineChannel

    # Start this app at launch time and keep it alive
    atstart: yes
    relaunch: yes

    # The command-line to launch.
    cmdline: "path/to/app --args "
```

Note: It's important not to include any `{{macro}}` in the channel. Doing so will link the channel to a parameter and make it part of the test.

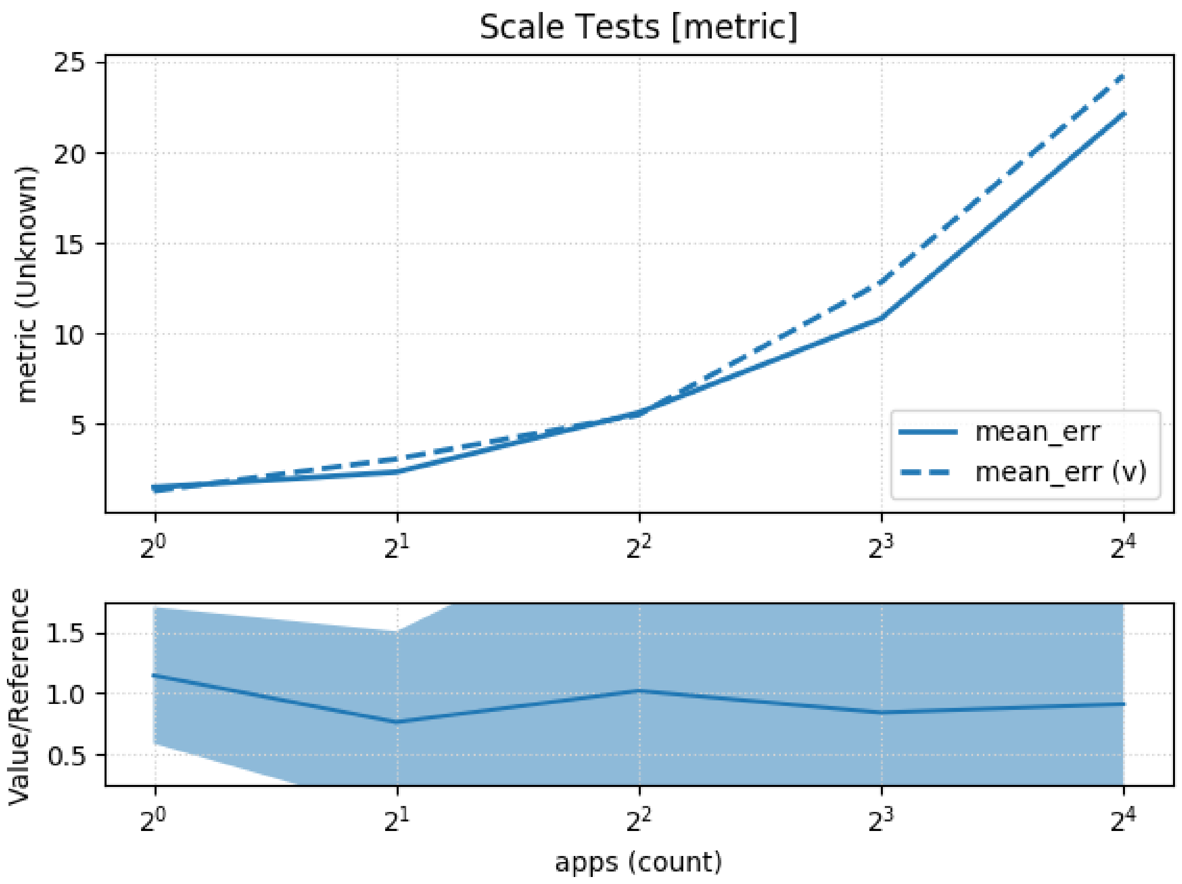
6.7.2 Including reference data in your plots

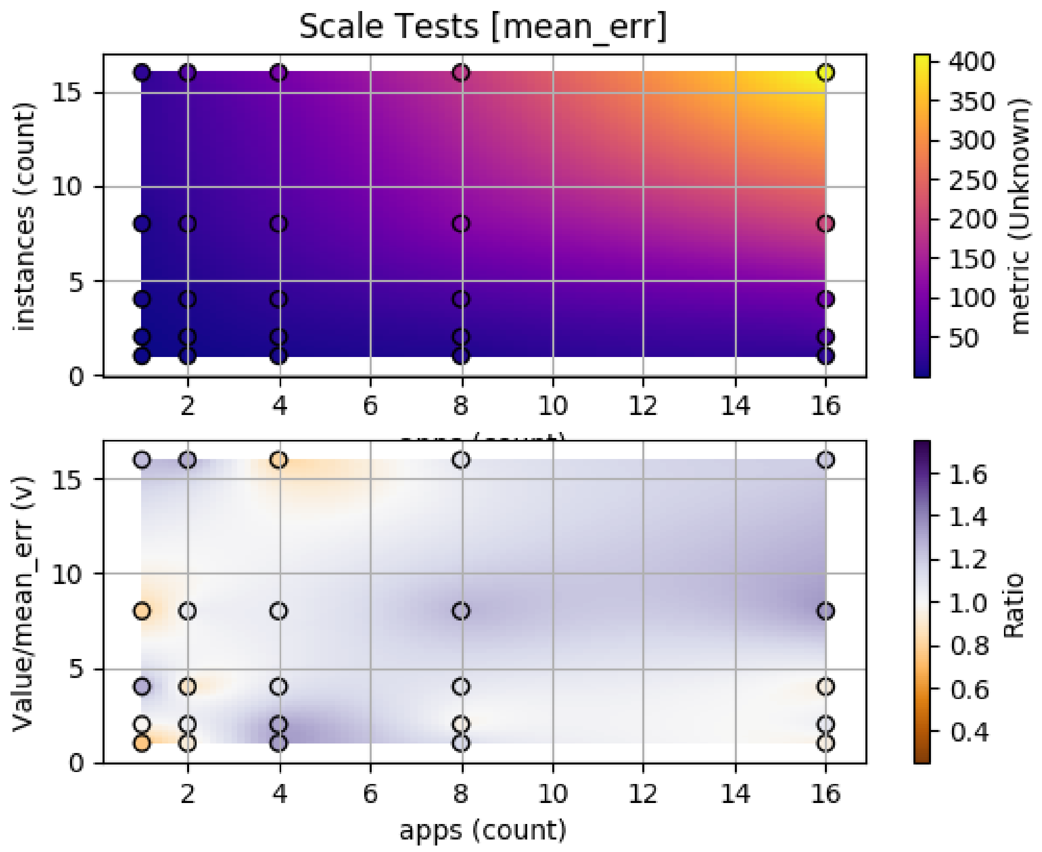
If you are running the tests as part of a CI you migth be interested into comparing the results to a reference run. To do so, use the `reference` parameter in the *PlotReporter*.

The `url` should point to a URL where a raw dump (generated by a *RawReporter*) is available. This raw dump will be used as a reference:

```
reporters:
  - class: reporter.PlotReporter

    # Use the given reference
    reference:
      data: http://path.to/refernce-raw.json
```

The reference can be computed for 1D and 2D plots. For example:


Scale Tests [metric]
metric (Unknown)
25
20
15
10
5
mean_err
mean_err (v)
Value/Reference
1.5
1.0
0.5
2^0
2^1
2^2
2^3
2^4
apps (count)


Scale Tests [mean_err]
instances (count)
apps (count)
metric (Unknown)
400
350
300
250
200
150
100
50
Value/mean_err (v)
apps (count)
Ratio
1.6
1.4
1.2
1.0
0.8
0.6
0.4

CHAPTER 7

Configuration

The `dcos-perf-test-driver` is congured through a human-readable YAML configuration file. This file defines the steering parameters for the test harness and the parameters to passed down to the classes that compose the overall scale test.

Ideally, the task of every class should quite abstract and can be configured into fit any testing scenario. In detail, there are two major configuration groups:

- The *Global Configuration Statements* configuration – that is used by the test harness and the reporting services in order to steer the test process.
- The *Per-Class Configuration Statements* configuration – that provides detailed configuration for every class plugged into the harness.

7.1 Global Configuration Statements

The global configuration statements provide the information required by the test harness to drive the tests. This configuration is shared between every component in the harness and contain the following types of information:

- **Harness Configuration** : That define how many times to repeat the tests, how long to wait before a test is considered stale etc.
- **Parameters, Metrics & Indicators** : What kind of input parameters the tests will operate upon, and what kind of metrics and indicators will be extracted from the test.
- **Metadata** : What kind of arbitrary metadata should be collected along with the results in order to identify this run.
- **Macros** : The values of various macros further used in the configuration.

7.1.1 config

```
config:
  runs: 1
  title: "Test title"
  staleTimeout: 1200
```

The general test configuration section contains global information for every other test section.

config.runs

```
config:
  ...
  runs: 5
```

Defines how many time to repeat the entire test suite in order to increase the quality of the statistics collected. The default value is 1.

config.title

```
config:
  ...
  title: "Some Title"
```

Defines the title of the test. This is mainly used by the reporting services.

config.staleTimeout

```
config:
  ...
  staleTimeout: 1200
```

Defines how long to wait (in seconds) for a policy to change state, before considering it "Stale".

The stale timeout is used as the last resort in order to continue running other test cases when one test case fails.

config.parameters

```
config:
  ...
  parameters:
    - name: parameterName
      desc: A short description of the metric
      units: sec
      uuid: 1234567890
```

Defines the free variables of the test. Each parameter is effectively an *axis* for the test.

It's important to define all the parameters that are going to take part in the test since some components are pre-conditioning their structures based on this configuration.

The `name` and the `summarize` properties are the only ones required. The `desc`, `units` and `uuid` are only used by the reporters.

If you are using the PostgREST reporter, the `uuid` should be a valid GUID for the parameter being tracked.

config.metrics

```
config:
  ...
  metrics:
    - name: parameterName
      desc: A short description of the metric
      summarize: [min, max]
      title: Legend Title
      units: sec
      uuid: 1234567890
```

Defines the measured values that should come as a result of the test.

Like with the parameters, it's important to define all the metrics that take part in the test since some components are pre-conditioning their structures based on this configuration.

The `name` and the `summarize` properties are the only ones required. The `desc`, `title`, `uuid` and `units` are only used by the reporters.

If you are using the PostgREST reporter, the `uuid` should be a valid GUID for the metric being tracked.

The `summarize` array defines one or more summarizer classes to use for calculating a single scalar value from the values of the timeseries. Note that there are two ways to define summarizers:

- The **compact format** accepts no configuration parameters, assumes that the name of the summariser (visible in the reports) is it's type and the functionality will be imported from the *BuiltInSummarizer*.

  ```
  config:
    ...
    metrics:
      - name: parameterName
        ...
        summarize: [min, max]
  ```

- The **extended format** accepts full configuration parameters and you can even specify your own summarizers. You can still use the `@name` class name if you want to refer to a built-in summarizer.

  ```
  config:
    ...
    metrics:
      - name: parameterName
        ...
        summarize:
          - class: summarizers.MyCustomSummarizer.summarisationFunction
            name: "Name in plots"
  ```

config.indicators

```
config:
  ...
  indicators:
    - name: meanDeploymentTime
      class: indicator.NormalizedMeanMetricIndicator
      metric: deploymentTime.mean
      parameter: instances
```

Defines one or more *Indicators* that are going indicate the result of the test as a single scalar value. Usually an indicator normalizes some of the metrics to the axis values and calculates a single number that represents the outcome of the test.

For instance, the example above normalizes the *mean* value of all sampled *deploymentTime* values of each run, to the value of the *instances* parameter. Effectively calculating the *mean deployment time per instance* indicator.

config.definitions

```
config:
  ...
  definitions:
    - name: secret
      desc: The secret password to use
      default: 1234
      required: yes
```

Describes the definitions to require from the user to specify before the tests can be started. This section is only used to provide high-level input-validation.

For instance, invoking the tool without providing the `secret` definition, will yield the following error:

```
ERROR 2017-07-04 15:18:00 Main: Missing required definition `secret` (The secret␣
↪password to use)
```

The values of such definitions are provided via the *–define* command-line argument.

config.meta

```
config:
  ...
  meta:
    test: first-name
```

General purpose metadata that will accompany the test results.

It is also possible to provide metadata via the command-line using the *–meta* argument.

7.1.2 define

```
define:
  parameter1: value
  parameter2: another_value
```

The `define` section assigns values to various macro definitions that can be used later in the configuration file. Refer to *Macros* for more details.

The values of such definitions can be overriden through the *–define* command-line argument.

7.2 Per-Class Configuration Statements

A scale test in `dcos-perf-test-driver` is implemented as an arbitrary number of interconnected classes plugged into a shared event bus.

Each class has it's own configuration format and based on it's task is separated into one of the following categories:

- *policies* : Drive the evolution of the parameters over time.
- *channels* : Define how a parameter change is passed to the app being tested.
- *observers* : Observe the app and extract useful information from it's behaviour.
- *trackers* : Track events and emmit performance measurement metrics.
- *reporters* : Report the test results into a file, network or service.

7.2.1 policies

```
policies:
  - class: policy.SomeClass
    param1: value1
    ...
```

The policies drive the evolution of the performance test. They are receiving synchronisation events from the Event Bus and they are changnging the test parameters.

Every change to the test parameters is triggering a state change to the application being tested. The change is applied to the application through *channels*.

The `class` parameter points to a class from within the `performance.driver.classess` package to load. Every class has it's own configuration parameters check *Class Reference* for more details.

7.2.2 channels

```
channels:
  - class: channel.SomeClass
    param1: value1
    ...
```

Channels apply the changes of the parameters to the application being tested.

The `class` parameter points to a class from within the `performance.driver.classess` package to load. Every class has it's own configuration parameters check *Class Reference* for more details.

Channel Triggers

By default a channel is triggered when any of the macros used on it's expression is modified. For example, the following channel will be triggered when the parameter `param1` changes:

```
channels:
  - class: channel.SomeClass
    param1: "{{param1}}"
    param2: value2
    ...
```

Custom Triggers

There are two properties you can use in order to modify this behaviour:

The **parameters** property override the parameter heuristics and provide an explicit list of the parameters that should be considered. In the following example the channel will be triggered only if `param2` changes:

```
channels:
  - class: channel.SomeClass
    parameters: [param2]
    ...
```

The **trigger** property defines the triggering behavior and it can take the following values:

- `always` : Trigger every time a parameter changes, regardless if it exists in the *parameters* list or in the macros or not
- `matching` (Default): Trigger every time a parameter listed in the *parameters* list or in the macros changes
- `changed`: Trigger every time a parameter listed in the *parameters* list or in the macros changes **and** the new value is different than the previous one. This is particularly useful if you are working with multiple axes.

For example, to trigger the channel on *every* update, use:

```
channels:
  - class: channel.SomeClass
    trigger: always
    ...
```

7.2.3 observers

```
observers:
  - class: observer.SomeClass
    param1: value1
    ...
```

The observers are monitoring the application being tested and they are extracing useful events into the message bus. Such events are usually used by the policy class to steer the evolution of the test and by the tracker classes to extract metric measurements.

The `class` parameter points to a class from within the `performance.driver.classess` package to load. Every class has it's own configuration parameters check *Class Reference* for more details.

7.2.4 trackers

```
trackers:
  - class: tracker.SomeClass
    param1: value1
    ...
```

The trackers are extracting metric values by analysing the events emmited by the observers and other components in the bus.

The `class` parameter points to a class from within the `performance.driver.classess` package to load. Every class has it's own configuration parameters check *Class Reference* for more details.

7.2.5 reporters

```
reporters:
  - class: tracker.SomeClass
    param1: value1
    ...
```

The reporters collecting the test results and createing a report. This could mean either writing some results to the local filesystem, or reporting the data to an online service.

The `class` parameter points to a class from within the `performance.driver.classess` package to load. Every class has it's own configuration parameters check *Class Reference* for more details.

7.2.6 tasks

```
tasks:
  - class: tasks.SomeClass
    at: trigger
    ...
```

The tasks are one-time operations that are executed at some trigger and do not participate in the actual scale test process. Such tasks can be used to log-in into a DC/OS cluster, clean-up some test traces or prepare the environment.

The `class` parameter points to a class from within the `performance.driver.classess` package to load. Every class has it's own configuration parameters check *Class Reference* for more details.

The `at` parameter selects the trigger to use. Supported values for this parameter are:

- `setup` : Called when the sytem is ready and right before the policy is started.
- `pretest` : Called before every run
- `intertest` : Called right after a parameter change has occured
- `posttest` : Called after every run
- `teardown` : Called when the system is tearing down

7.3 Separating Configuration Files

As your configuration increases in size it's some times helpful to separate it into multiple files. The `dcos-perf-test-driver` implements this behaviour using the `import:` array:

```
import:
  - metrics.yml
  - parameters.yml
```

The location of every file is relative to the configuration file they are in.

This instructs `dcos-perf-test-driver` to load the given files and merge all their sections together. Note that array statements and dictionary statements behave differntly when merged.

7.3.1 Merging arrays

Array statements are **concatenated**. This means that the following two files:

```yaml
# First
observers:
  - class: observer.FooObserver
```

```yaml
# Second
observers:
  - class: observer.BarObserver
```

Will result in the following configuration:

```yaml
observers:
  - class: observer.FooObserver
  - class: observer.BarObserver
```

7.3.2 Merging dictionaries

Dictionary statements are **merged**, meaning that same keys are replaced with the values coming from the configuration file that comes last. This means that the following two files:

```yaml
# First
define:
  foo: first
  bar: another
```

```yaml
# Second
define:
  foo: second
  baz: other
```

Will result in the following configuration:

```yaml
define:
  foo: second
  bar: another
  baz: other
```

7.4 Macros

The `dcos-perf-test-driver` implements a minimal template engine that can be used to provide parametric configuration to your test. For example, if you are launching a command-line application and some parameters need to change over time you can use the `{{parameter_name}}` macro expression:

```yaml
channels:
  - class: channel.CmdlineChannel
    cmdline: "benchmark_users --users={{users}}"
```

Such macros can appear at any place in your YAML configuration and they will be evaluated to the definition or parameter with the given name. Refer to *Value sources* for more details.

Note: Be aware that macros can only appear in YAML values and not in the key names. For instance, the following expression is invalid:

```
config:
  "{{prefix}}_run": "always"
```

7.4.1 Value sources

The values of the macros are coming from various sources, in the following order:

- Definitions in the configuration file through the *define* statement.
- Definitions given by the command-line through the *–define* argument.
- The test parameters and their values during the current test phase.

7.4.2 Default values

It is possible to provide a default value to your macros using the `{{macro|default}}` expression. For example:

```
reporters:
  - class: reporter.PostgRESTReporter
    url: "{{reporter_url|http://127.0.0.1:4000}}"
```

7.4.3 Functions

It is possible to call a small set of functions in your macro. The following functions are available:

uuid

Compute a unique GUID ID. For example:

```
channels:
  - class: channel.CmdlineChannel
    cmdline: "ctl useradd --name=user-{{uuid()}}"
```

date(format)

Compose a date expression from the current time and date. For example:

```
reporters:
  - class: reporter.S3Reporter
    path: "metrics/{{date(%Y%m%d)}}-results.json"
```

The `format` argument is exactly what python's `strftime` accepts.

safepath(expression)

Replaces all the 'unsafe' characters for a path expression with '_'

```
reporters:
  - class: reporter.S3Reporter
    path: "metrics/{{safepath(test_name)}}-results.json"
```

The `expression` argument can be any legit macro expression.

eval(expression)

Evaluates the given expression as a python expression.

```
policies:
  - class: policy.SimplePolicy
    value: "{{eval(apps * tasks)}}"
```

7.4.4 Metadata as macros

In some cases (for example in the reporter definitions) it might be needed to evaluate the value of a metadata. You can do so by using the `{{meta:name}}` syntax. For example:

```
reporters:
  - class: reporter.S3Reporter
    bucket: marathon-artifacts
    path: "metrics/{{meta:version}}-results.json"
```

7.5 Configuration Example

The following configuration example accompanies the *Example* test case in the documentation. Refer to it for mroe details.

```
##################################################
# Global test configuration
#################################################
config:

  # Repeat this test 5 times
  repeat: 5

  # The title of the scale test
  title: "Scale Tests"

  # Define the parameters this policy will be updating
  parameters:
    - name: deploymentRate
      units: depl/s
      desc: The number of deployments per second

  # Define the metrics we are measuring
  metrics:
    - name: responseTime
      units: sec
      desc: The time for an HTTP request to complete
      summarize: [mean_err]
  # Introduce some indicators that will be used to extract
  # the outcome of the test as a single scalar value
  indicators:

    # Calculate `meanResponseTime` by calculating the normalizing average
    # of all the `responseTime` mean values, normalized against the current
    # deploymentRate
    - name: meanResponseTime
      class: indicator.NormalizedMeanMetricIndicator
      metric: responseTime.mean_err
      normalizeto: deploymentRate

##################################################
# Macro Values
##################################################
define:

  # Define `marathon_url` that is required by other fragments
  marathon_url: http://127.0.0.1:8080

##################################################
# Test Metadata
##################################################
meta:

  # All these values will be included in the test results but
  # do not participate in the actual test
  test: 1-app-n-instances
  env: local
  config: simulator

##################################################
# Test policy configuration
##################################################
policies:

  # We are using a multi-step policy due to it's configuration
  # flexibility, even though our tests have only one step.
  - class: policy.MultiStepPolicy
    steps:

      # Explore deploymentRate from 100 to 1000 with interval 50
      - name: Stress-Testing Marathon
        values:
          - parameter: deploymentRate
            min: 100
            max : 1000
            step: 50

        # Advance when the deployment is successful
        events:
          advance: MarathonDeploymentSuccessEvent:notrace

        # Advance only when we have received <deploymentRate> events
        advance_condition:
          events: "deploymentRate"
```

```
##################################################
# Channel configuration
##################################################
channels:

  # Perform an HTTP request for every `deploymentRate` parameter change
  - class: channel.HTTPChannel
    url: {{marathon_url}}/v2/apps
    verb: POST
    repeat: "{{deploymentRate}}"
    body: |
      {
        "id": "/scale-instances/{{uuid()}}",
        "cmd": "sleep 1200",
        "cpus": 0.1,
        "mem": 64,
        "disk": 0,
        "instances": 0,
        "backoffFactor": 1.0,
        "backoffSeconds": 0
      }

##################################################
# Observer configuration
##################################################
observers:

  # We are measuring the HTTP response time of the /v2/groups endpoint
  - class: observer.HTTPTimingObserver
    url: {{marathon_url}}/v2/groups
    interval: 1

  # We also need to listen for marathon deployment success events in order
  # to advance to the next test values, so we also need a marathon poller
  - class: observer.MarathonPollerObserver
    url: "{{marathon_url}}"

##################################################
# Tracker configuration
##################################################
trackers:

  # Track the `responseTime`, by extracting the `responseTime` from the
  # HTTP measurement result event
  - class: tracker.EventAttributeTracker
    event: HTTPTimingResultEvent
    extract:
      - metric: responseTime
        attrib: responseTime


##################################################
# Result reporters
##################################################
reporters:

  # Dump raw time series results to results/dump.json
  - class: reporter.RawReporter
```

```
    filename: results/dump.json

  # Dump summarized CSV values to results/results.csv
  - class: reporter.CSVReporter
    filename: results/results.csv

  # Create plots as images to results/plot-*.png
  - class: reporter.PlotReporter
    prefix: results/plot-

##################################################
# One-time tasks
##################################################
tasks:

  # Right after ever test run we should remove all the instances
  - class: tasks.marathon.RemoveGroup
    url: "{{marathon_url}}"
    group: /scale-instances
    at: intertest

  # Also remove the tests if they were abruptly terminated
  - class: tasks.marathon.RemoveGroup
    url: "{{marathon_url}}"
    group: /scale-instances
    at: teardown
```

CHAPTER 8

Class Reference

This section contains the detailed configuration and description of every class available in the DC/OS Performance Test Driver.

8.1 Policies

The `policy` classes are **driving the tests** by controlling the evolution of the test parameters over time. The parameters changed are applied to the test through the *Channels*.

8.1.1 MultivariableExplorerPolicy

class `performance.driver.classes.policy.`**`MultivariableExplorerPolicy`**(*config*, *eventbus*, *parameterBatch*)

The **Multi-Variable Exploration Policy** is running one scale test for every product of the parameters defined in the `matrix`.

```
policies:
  - class: policy.MultivariableExplorerPolicy

    # The following rules describe the permutation matrix
    matrix:
      # The key is the name of the parameter to control
      param:
        ...

      # A "discreet" parameter can take one of the specified values
      apps:
        type: discreet
        values: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
      # A "sample" parameter takes any value within a numerical range
      size:
        type: sample
        min: 0 # Default
        max: 1 # Default
        step: 1 # Default
        samples: 10 # Default

      # A "range" parameter takes all the values within the given range
      instances:
        type: range
        min: 0
        max: 1
        step: 1

    # The event binding configuration
    events:

      # Signals are events that define a terminal condition and it's status
      #
      # For example, in this case when the `MarathonDeploymentSuccessEvent`
      # is received, the test will be completed with status `OK`
      #
      signal:
        OK: MarathonDeploymentSuccessEvent
        FAILED: MarathonDeploymentFailedEvent
        ... : ...

      # [Optional] Wait for the given number of signal events before
      # considering the test complete.
      #
      # This parameter is an expression evaluated at run-time, so you
      # could use anything that can go within a python's `eval` statement
      #
      # For example: "discreet + 2"
      #
      signalEventCount: 1

      # [Optional] Start the tests with this event is received
      start: EventToWaitUntilReady
```

This policy is first computing all possible combinations of the parameter matrix given and is then running the tests for every one.

The policy will start immediately when the test driver is ready unless the `start` event is specified. In that case the policy will wait for this event before starting with the first test.

The policy continues with the next test only when a *signal* event is received. Such events are defined in the `signal` dictionary. Since a test can either complete successfully or fail you are expected to provide the status indication for every signal event.

It is also possible to wait for more than one signal event before considering the test complete. To specify the number of events to expect you can use the `signalEventCount` parameter.

However since the number of events to expect depends on an arbitrary number of factors, it's possible to use an expression instead of a value. For the expression you can use the names of the parameters taking part in the matrix and the special variable `_i` that contains the number of the test, starting from 1.

For example `signalEventCount: "apps + size/2"`

8.1.2 TimeEvolutionPolicy

class `performance.driver.classes.policy.`**`TimeEvolutionPolicy`**(*config*, *eventbus*, *parameterBatch*)

The **Time Evolution Policy** is changing a parameter monotonically as the time evolves.

```
policies:
  - class: policy.TimeEvolutionPolicy

    # Configure which parameters to evolve over time
    evolve:

      # The name of the parameter to change
      - parameter: parameterName

        # [Optional] The interval (in seconds) at which to evolve the
        # parameter (default is 1 second)
        interval: 1

        # [Optional] By how much to increment the parameter (default is 1)
        step: 1

        # [Optional] The initial value of the parameter (default is 0)
        min: 0

        # [Optional] The final value of the parameter, after which the
        # test is completed.
        max: 10

    # The event binding configuration
    events:

      # [Optional] Terminate the tests when this event is received.
      end: EventToWaitForCompletion

      # [Optional] Start the tests with this event is received
      start: EventToWaitUntilReady
```

This policy is first computing all possible combinations of the parameter matrix given and is then running the tests for every one.

8.1.3 MultiStepPolicy

class `performance.driver.classes.policy.`**`MultiStepPolicy`**(*config*, *eventbus*, *parameterBatch*)

The **Step Policy** evolves various parameters through various steps.

```
policies:
  - class: policy.MultiStepPolicy

    # Configure the policy steps
    steps:

      # The name of the step
      - name: First Step

        # The values to explore
        values:

          # 1) A fixed-value parameter
          - parameter: name
            value: 1

          # 2) A fixed-value parameter calculated with an expression
          - parameter: name
            value: "log(parameter1) + parameter2"

          # 3) A fixed list of values
          - parameter: name
            values: [1, 2, 3, 4, 5 ...]

          # 4) A calculated range of values
          - parameter: name
            min: 0
            max : 100
            step: 1

            # Set to `no` if you don't want to include the `max` value
            inclusive: no

          # 5) A sampled subset of a uniformly distributed values
          - parameter: name
            min: 0
            max: 10000
            step: 1
            sample: 100

            # Set to `no` if you don't want to include the `max` value
            inclusive: no

        # [Optional] Trigger the following named tasks
        tasks:

          # [Optional] Run this named task before start
          start: startTask

          # [Optional] Run this named task after step completion
          end: endTask

          # [Optional] Run this named before a value change
          pre_value: advanceTask

          # [Optional] Run this named after a value change
          post_value: advanceTask

        # [Optional] Event configuration
        events:

          # [Optional] Wait for this to start the step
          start: EventName

          # [Optional] Wait for this event to end the step
          end: EventName

          # [Optional] Wait for this event to fail this step
          fail: EventName

          # [Optional] Wait for this event before advancing to the next value
          advance: EventName

        # [Optional] Custom end condition
        end_condition:

          # [Optional] Wait for the specified number of end/fail events
          # before considering the step completed
          events: 1

          # [Optional] Wait for the specified number of seconds after the
          # final step is completed before completing the test
          linger: 0

        # [Optional] Custom advance condition
        advance_condition:

          # [Optional] Wait for the specified number of advance events
          # before considering the value ready to advance. Note that this
          # can be a python expression
          events: ""

          # [Optional] If the step was not advanced by itself in the given
          # time, marked is a timed out and continue with the next
          timeout: 10s

          # [Optional] What flag to set on the run that advanced due to a
          # timeout. Set this to `OK` to make timeout a legit action or
          # `FAILED` to make timeout a critical failure.
          timeout_status: TIMEOUT
```

This policy is first computing all possible combinations of the parameter matrix given and is then running the tests for every one.

Important: The `MultiStepPolicy` is respecting the event tracing principle. This means that all the events in the `events` section will be matched only if they derive from the same step of a policy action.

If the events you are listening for do not belong on a trace initiated by the current step, use the *:notrace* indicator.

For example, let's say that policy sets the number of instances to 3, that triggers a deployment that eventually triggers a `DeploymentCompletedEvent` when completed. In this case you can listen for `advance: DeploymentCompletedEvent` events.

However, if you are advancing at clock ticks, they are not part of a trace initiated by the policy and therefore you must use: ``advance: TickEvent:notrace`

8.1.4 SimplePolicy

class `performance.driver.classes.policy.`**`SimplePolicy`**(*config*, *eventbus*, *parameterBatch*)

The **Simple Policy** submits a single parameter change event and terminates when the designated end condition is met. No repetition or parameter exploration is performed.

```
policies:
  - class: policy.SimplePolicy

    # Which parameters to submit
    parameters:
      apps: 1

    # The event binding configuration
    events:

      # [Optional] Terminate the tests when this event is received.
      end: EventToWaitForCompletion

      # [Optional] Start the tests with this event is received
      start: EventToWaitUntilReady

    # [Optional] Maximum running time for this policy
    timeout: 10s
```

This policy can be used if you are not interested about parameter exploration or any other feature of the driver, but you rather want to observe the system response on a particular condition.

8.2 Channels

The `channel` classes are **applying the test values** to the application being tested. They are responsible for translating the modification of the scalar test parameter to one or more actions that need to be taken in order to implement the test.

After a channel has applied the values, the *Observers* are used to extract useful events from the application for further processing.

For example, the *CmdlineChannel* is re-starting the application with a different command-line, every time the test parameters have changed.

8.2.1 CmdlineChannel

class `performance.driver.classes.channel.`**`CmdlineChannel`**`(*args, **kwargs)`

The *Command-line Channel* launches an application, passes the test parameters through command-line arguments and monitors it's standard output and error.

```
channels:
  - class: channel.CmdlineChannel

    # The command-line to launch.
    cmdline: "path/to/app --args {{macros}}"

    # [Optional] The standard input to send to the application
    stdin: |
      some arbitrary payload with {{macros}}
      in it's body.

    # [Optional] Environment variables to define
    env:
      variable: value
      other: "value with {{macros}}"
    # [Optional] The directory to launch this app in
    cwd: "{{marathon_repo_dir}}"

    # [Optional] The script to use for teardown
    teardown: stop_daemon

    # [Optional] If set to `yes` the app will be launched as soon
    # as the driver is up and running.
    atstart: yes

    # [Optional] If set to `yes` (default) the app will be re-launched
    # if it exits on it's own.
    restart: yes

    # [Optional] If set to `yes` the "cmdline" expression will be evalued
    # in a shell.
    shell: no

    # [Optional] The time between the SIGINT and the SIGKILL signal
    gracePeriod: 10s

    # [Optional] Change the `kind` of log messages emitted by this channel
    # instead of using the default 'stdout' / 'stderr'
    kind:
      stdout: custom_out
      stderr: custom_err
```

When a parameter is changed, the channel will kill the process and re-launch it with the new command-line.

For every line in standard inout or output, a `LogLineEvent` is emitted with the contents of the line.

When the application launched through this channel exits the channel can take two actions depending on it's configuration:

- If `restart: yes` is specitied (default), the channel will re-launch the application in oder to always keep it running.
- If `restart: no` is specified, the channel will give up and publish a `CmdlineExitZeroEvent` or a `CmdlineExitNonzeroEvent` according to the exit code of the application.

Note: Note that if there are no `{{macro}}` defined anywhere in the body of the configuration this channel will not be triggered when a parameter is updated and thus the application will never be launched.

If you still want the application to be launched, use the `atstart: yes` parameter to instruct the channel to launch the application at start.

8.2.2 HTTPChannel

class `performance.driver.classes.channel.`**`HTTPChannel`**(**args*, ***kwargs*)

The *HTTP Channel* performs an HTTP Requests when a parameter changes.

```
channels:
  - class: channel.HTTPChannel

    # The URL to send the requests at
    url: http://127.0.0.1:8080/v2/apps
    # The body of the HTTP request
    body: |
      {
        "cmd": "sleep 1200",
        "cpus": 0.1,
        "mem": 64,
        "disk": 0,
        "instances": {{instances}},
        "id": "/scale-instances/{{uuid()}}",
        "backoffFactor": 1.0,
        "backoffSeconds": 0
      }

    # [Optional] The HTTP Verb to use (Defaults to 'GET')
    verb: POST

    # [Optional] The HTTP headers to send
    headers:
      Accept: text/plain

    # [Optional] How many times to re-send the request (can be
    # a macro value)
    repeat: 1234

    # [Optional] How long to wait between re-sends (in seconds)
    # If missing the next request will be sent as soon as the previous
    # has completed
    repeatInterval: 1234

    # [Optional] For which event to wait before re-sending the request.
    repeatAfter: event
```

When a parameter is changed, a new HTTP request is made. If a `repeat` parameter is specified, the same HTTP request will be sent again, that many times.

Various events are published from this channel, that can be used to synchronise other components or track metrics.

- When an HTTP request is initiated an `HTTPRequestStartEvent` is published.
- When an HTTP request is completed and the response is pending, an `HTTPFirstRequestEndEvent` is published.
- When the HTTP response is starting, an `HTTPFirstResponseStartEvent` is published.
- When the HTTP response is completed, an `HTTPResponseEndEvent` is published.

If you are using the `repeat` configuration parameter you can also use the following events:

- When the first HTTP request is started, the `HTTPFirstRequestStartEvent` is published.
- When the last HTTP request is started, the `HTTPLastRequestStartEvent` is published.
- When the first HTTP request is completed, the `HTTPFirstRequestEndEvent` is published.
- When the last HTTP request is completed, the `HTTPLastRequestEndEvent` is published.
- When the first HTTP response is started, the `HTTPFirstResponseStartEvent` is published.
- When the last HTTP response is started, the `HTTPLastResponseStartEvent` is published.

- When the first HTTP response is completed, the `HTTPFirstResponseEndEvent` is published.
- When the last HTTP response is completed, the `HTTPLastResponseEndEvent` is published.

Therefore it's possble to track the progress of the entire repeat batch, aswell as the progress of an individual HTTP event.

Note: This channel will automatically inject an `Authorization` header if a `dcos_auth_token` definition exists, so you don't have to specify it through the `headers` configuration.

Note that a `dcos_auth_token` can be dynamically injected via an authentication task.

8.2.3 MarathonUpdateChannel

class `performance.driver.classes.channel.`**`MarathonUpdateChannel`**(*config*, *eventbus*)

The *Marathon Update Channel* is performing arbitrary updates to existing apps, groups or pods in marathon, based on the given rules.

```
channels:
  - class: channel.MarathonUpdateChannel

    # The base url to marathon
    url: "{{marathon_url}}"

    # One or more updates to perform
    update:

      # `patch_app` action is updating all or some applications
      # and modifies the given properties
      - action: patch_app

        # The properties to patch
        patch:
          instances: 3

        # [Optional] Update only application names that match the regex.
        #            If missing, all applications are selected.
        filter: "^/groups/variable_"

        # [Optional] Update at most the given number of apps.
        #            If missing, all applications are updated.
        limit: 10

        # [Optional] Shuffle apps before picking them (default: yes)
        shuffle: no

    # [Optional] Additional headers to include to the marathon request
    headers:
      x-Originating-From: Python
```

When a parameter is changed, the channel will kill the process and re-launch it with the new command-line.

8.2.4 MarathonDeployChannel

class `performance.driver.classes.channel.`**`MarathonDeployChannel`**`(`**args*, ***kwargs*`)`

The *Marathon Deploy Channel* is performing one or more deployment on marathon based on the given rules.

```
channels:
  - class: channel.MarathonDeployChannel

    # The base url to marathon
    url: "{{marathon_url}}"

    # [Optional] Retry deployments with default configuration
    retry: yes

    # [Optional] Retry with detailed configuration
    retry:

      # [Optional] How many times to re-try
      tries: 10

      # [Optional] How long to wait between retries
      interval: 1s

    # One or more deployments to perform
    deploy:

      # The type of the deployment
      - type: app

        # The app/pod/group spec of the deployment
        spec: |
          {
            "id": "deployment"
          }

        # [Optional] Repeat this deployment for the given number of times
        repeat: 10
        repeat: "{{instances}}"
        repeat: "{{eval(instances * 3)}}"

        # [Optional] How many deployments to perform in parallel
        # When a deployment is completed, another one will be scheduled
        # a soon as possible, but at most `parallel` deployment requests
        # will be active. This is mutually exclusive to `burst`.
        parallel: 1

        # [Optional] [OR] How many deployments to do in a single burst
        # When all the deployments in the burst are completed, a new burst
        # will be posted. This is mutually exclusive to `parallel.
        burst: 100

        # [Optional] Throttle the rate of deployments at a given RPS
        rate: 100

        # [Optional] Stall the deployment for the given time before placing
        # the first HTTP request
        delay: 10s
```

8.3 Observers

The `observer` classes are **monitoring the application being tested** and extract useful events. Such events are either required by the *Policies* in order to evolve the tests, or tracked by the *Trackers* in order to calculate the test results.

8.3.1 HTTPTimingObserver

class `performance.driver.classes.observer.`**`HTTPTimingObserver`**`(`**args*, ***kwargs*`)`

The *HTTP Timing Observer* is performing HTTP requests to the given endpoint and is measuring the request and response times.

```
observers:
  - class: observer.HTTPTimingObserver

    # The URL to send the requests at
    url: http://127.0.0.1:8080/v2/apps

    # [Optional] The interval of the reqeusts (seconds)
    interval: 1

    # [Optional] The body of the HTTP request
    body: |
      {
        "cmd": "sleep 1200",
        "cpus": 0.1,
        "mem": 64,
        "disk": 0,
        "instances": {{instances}},
        "id": "/scale-instances/{{uuid()}}",
        "backoffFactor": 1.0,
        "backoffSeconds": 0
      }

    # [Optional] The HTTP Verb to use (Defaults to 'GET')
    verb: POST

    # [Optional] The HTTP headers to send
    headers:
      Accept: text/plain
```

This observer is publishing a `HTTPTimingResultEvent` every time a sample is taken. Refer to the event documentation for more details.

8.3.2 JMXObserver

class `performance.driver.classes.observer.`**`JMXObserver`**`(`**args*, ***kwargs*`)`

The *JMX Observer* connects to the java management console of a running java application and extracts the given metrics.

```
observers:
  - class: observer.JMXObserver

    # [Optional] Re-send measured values on ParameterUpdateEvent
    resendOnUpdate: yes
    # Connection information
    connect:

      # [Optional] Specify the host/port where to connect
      host: 127.0.0.1
      port: 9010

      # [Optional] Execute the given shell expression and assume the STDOUT
      # contents is the PID where to attach. If available, {{cmdlinepid}}
      # will contain the PID of the last detected PID from the cmdline
      # channel
      # ------------------------------------------------------------------
      # DANGER!! Evaluated as a shell expression
      # ------------------------------------------------------------------
      pid_from: "pgrep -P $(pgrep -P {{cmdlinepid}})"

    # Which metrics to extract
    metrics:

      # Specify the name of the metric and the source
      - field: tagName

        # The java Management Interface MBean to use (Object Name)
        mbean: "java.lang:type=Threading"

        # The attribute value to extract
        attrib: ThreadCount

        # [Optional] Python evaluation expression for the value
        value: "value"

    # Optional event configuration
    events:

      # [Optional] Wait for this event before activating the observer
      activate: MarathonStartedEvent

      # [Optional] If this event is received the observer is deactivated
      deactivate: ExitEvent
```

This observer is going to launch a utility process that is going to attach on the specified JVM instance. Upon successful connection it's going to start extracting all the useful information as `JMXMeasurement` events in the message bus.

Such events can be passed down to metrics using the `JMXTracker` tracker.

8.3.3 LogStaxObserver

class `performance.driver.classes.observer.`**`LogStaxObserver`**`(`**args*, ***kwargs*`)`

The **Logstax Observer** is logstash-like observer for dcos-perf-test-driver that uses some event contents as the line source and a set of rules for creating fields for post-processing.

```
observers:
  - class: observer.LogStaxObserver

    # An array of filters to apply on every line
    filters:
```

```
    # Grok Pattern Matching
    # ----------------------
    - type: grok

      # Match the given field(s) for GROK expressions
      match:
        message: "^%{IP:httpHost} - (?<user>%{WORD}|-).*"

      # [Optional] Overwrite the specified fields with the values
      # extracted from the grok pattern. By default only new fields
      # are appended.
      overwrite: [message, name]

      # [Optional] Add the given fields to the message
      add_field:
        source: grok

      # [Optional] Remove the given fields from the message
      remove_field: [source]

      # [Optional] Add the given tags in the message
      add_tag: [foo, bar]

      # [Optional] Remove the given tags in the message
      remove_tag: [bar, baz]

  # [Optional] Which event(s) to listen for and which fields to extract
  events:

    # By default it's using the `LogLineEvent`
    - name: LogLineEvent
      field: line

  # [Optional] One or more `codecs` to apply on the incoming lines.
  #            These codecs convert one or more log lines into
  codecs:

    # Pass-through every incoming line to a rule matcher
    - type: singleline

    # Group multiple lines into a block and then pass it to the
    # rule matcher as an array of lines. Refer to the `MultilineCodec`
    # for more details.
    - type: multiline
      lines:
        - match: firstLine.*
        - match: secondLine.*
```

This observer is trying to reproduce a logstash set-up, using the LogLineEvent as the only source. It is first passing the events through a *codec* that is going to create a processable messages. Each message contains fields and tags.

By default, the *singleline* codec is populating the *message* field with the contents of the line. The *multiline* codec is more elaborate and can be used in order to extract multi-line blocks from the incoming stream.

The messages are then passed to the filters. If a filter matches the incoming message it is going to apply the transformations described.

When the filter process is completed, the observer is going to braodcast a `LogStaxMessageEvent` that can be processed at a later time by the `LogStaxTracker` in order to extract useful metrics.

MultilineCodec

class `performance.driver.classes.observer.logstax.codecs.`**`MultilineCodec`**`(`*config*`)`

The multi-line codec is able to collect more than one matching lines into a single message.

```
observers:
  - class: observer.LogstaxObserver
    filters:

      - codec:
          class: logstax.codecs.MultlineCodec

          # The multiple lines to match, as one regex rule per line
          lines:

            # Match the given regex on the line
            - match: .*

              # [Optional] Set to `yes` to ignore case
              ignorecase: yes

              # [Optional] Set to `yes` to repeat indefinitely or
              # to a number to repeat up to the given number of times
              repeat: 4

              # [Optional] Set to `yet` to make this rule optional
              optional: no

            # Example: Match the given regex on the next line
            # repeat this pattern 2 times
            - match: .*
              repeat: 2

            # Example: Optionally match this regex on the next line
            - match: .*
              optional: yes

            # Example: Match the given regex until it stops matching
            - match: .*
              repeat: yes

          # [Optional] Set to `yes` to accept incomplete multiline matches
          acceptIncomplete: no

          # [Optional] Set to the new-line character you want to use when joining
          newline: ";""
```

For example, to join together lines that start with "::" you can use:

```
...
lines:
  - match: "^::.*$"
    repeat: yes
```

Or, to join together lines that open a bracket and close it on another line:

```
...
lines:
  - match: "^.*{$"
  - match: "^.*[^}]$"
    repeat: yes
```

SingleLineCodec

class `performance.driver.classes.observer.logstax.codecs.`**`SingleLineCodec`**`(`*config*`)`

The simple line codec is forwarding the line received as-is.

```
observers:
  - class: observer.LogstaxObserver
    filters:

      - codec:
          class: logstax.codecs.SingleLineCodec
```

8.3.4 MarathonEventsObserver

class `performance.driver.classes.observer.`**`MarathonEventsObserver`**`(`**args*, ***kwargs*`)`

The *Marathon Events Observer* is extracting high-level events by subscribing to the Server-Side Events endpoint on marathon.

```
observers:
  - class: observer.MarathonEventsObserver

    # The URL to the marathon SSE endpoint
    url: "{{marathon_url}}/v2/events"

    # [Optional] Use an external curl process for receiving the events
    # instead of the built-in raw SSE client
    curl: no

    # [Optional] Use the timestamp from the event. If set to no, the time
    # the event is arrived to the perf-driver is used
    useEventTimestamp: no

    # [Optional] Additional headers to send
    headers:
      Accept: test/plain
```

Since this observer requires an active HTTP session to marathon, it also publishes the `MarathonStartedEvent` when an HTTP connection was successfully established.

The following events are forwarded from the event bus:

- `MarathonDeploymentStepSuccessEvent`
- `MarathonDeploymentStepFailureEvent`
- `MarathonDeploymentInfoEvent`
- `MarathonDeploymentSuccessEvent`
- `MarathonDeploymentFailedEvent`

Note: In order to properly populcate the event's trace ID, this observer is also listening for *http* channel requests in order to extract the affected application name(s).

Note: This observer will automatically inject an `Authorization` header if a `dcos_auth_token` definition exists, so you don't have to specify it through the `headers` configuration.

Note that a `dcos_auth_token` can be dynamically injected via an authentication task.

8.3.5 MarathonLogsObserver

class `performance.driver.classes.observer.`**`MarathonLogsObserver`**`(`**args*, ***kwargs*`)`

This observer is based on the *LogStaxObserver* functionality in order to find and filter-out the marathon lines.

```
observers:
  - class: observer.MarathonLogsObserver
```

This observer accepts no configuration. It is processing the `LogLineEvent` messages dispatched by other observers or channels (ex. the `CmdlineChannel` channel).

Warning: This observer will currently only work if marathon is launched through a `CmdlineChannel`.

8.3.6 MarathonMetricsObserver

class `performance.driver.classes.observer.`**`MarathonMetricsObserver`**`(`**args*, ***kwargs*`)`

The *Marathon Metrics Observer* is observing for changes in the marathon */stats* endpoint and is emitting events according to it's configuration

```
observers:
  - class: observer.MarathonMetricsObserver

    # The URL to the marathon metrics endpoint
    url: "{{marathon_url}}/metrics"

    # [Optional] Additional headers to send
    headers:
      Accept: test/plain
```

This observer is polling the `/metrics` endpoint 2 times per second and for every value that is changed, a `MetricUpdateEvent` event is published.

Note: The name of the parameter is always the flattened name in the JSON response. For example, a parameter change in the following path:

```
{
  "foo": {
    "bar.baz": {
      "bax": 1
    }
  }
}
```

Will be broadcasted as a change in the following path:

```
foo.bar.baz.bax: 1
```

Note: This observer will automatically inject an `Authorization` header if a `dcos_auth_token` definition exists, so you don't have to specify it through the `headers` configuration.

Note that a `dcos_auth_token` can be dynamically injected via an authentication task.

8.3.7 MarathonPollerObserver

class `performance.driver.classes.observer.`**`MarathonPollerObserver`**`(`**args*, ***kwargs*`)`

The *Marathon Poller Observer* is a polling-based fallback observer that can fully replace the `MarathonEventsObserver` when the SSE event bus is not available.

```
observers:
  - class: observer.MarathonPollerObserver

    # The URL to the marathon base
    url: "{{marathon_url}}"

    # [Optional] Additional headers to send
    headers:
      Accept: test/plain

    # [Optional] How long to wait between consecutive polls (seconds)
    interval: 0.5

    # [Optional] How long to wait before considering the deployment "Failed"
    # If set to 0 the deployment will never fail.
    failureTimeout: 0

    # [Optional] How many times to re-try polling the endpoint before
    # considering the connection closed
    retries: 3

    # [Optional] Event binding
    events:

      # [Optional] Which event to wait to start polling
      start: StartEvent

      # [Optional] Which event to wait to stop polling
      stop: TeardownEvent
```

This observer is polling the `/groups` endpoint as fast as possible and it calculates deferences from the previously observed state. Any differences are propagated as virtual deployment events as:

- `MarathonDeploymentSuccessEvent`

- `MarathonDeploymentFailedEvent`

If requested, the poller is going to look for `MarathonDeploymentStartedEvent` events and is going to wait for it to be completed in a given time. If the time is passed, a synthetic failure event will be generated:

- `MarathonDeploymentFailedEvent`

Note: This observer will automatically inject an `Authorization` header if a `dcos_auth_token` definition exists, so you don't have to specify it through the `headers` configuration.

Note that a `dcos_auth_token` can be dynamically injected via an authentication task.

8.4 Trackers

The `tracker` classes are **monitoring events** in the event bus and are **producing metric values**. You most certainly need them in order to collect your results.

Refer to the *Event Reference* to know the events broadcasted in the bus.

8.4.1 DurationTracker

class `performance.driver.classes.tracker.`**`DurationTracker`**`(`**args*, ***kwargs*`)`

Tracks the duration between a `start` and an `end` event.

```
trackers:
  - class: tracker.DurationTracker

    # The metric where to write the measured value to
    metric: someMetric

    # The relevant events
    events:

      # The event to start counting from
      # (This can be a filter expression)
      start: StartEventFilter

      # The event to stop counting at
      # (This can be a filter expression)
      end: EndEventFilter
```

This tracker always operates within a tracking session, initiated by a `ParameterUpdateEvent` and terminated by the next `ParameterUpdateEvent`, or the completion of the test.

Important: The `start` and `end` events must contain the trace IDs of the originating `ParameterUpdateEvent`. Otherwise they won't be measured.

8.4.2 EventAttributeTracker

class `performance.driver.classes.tracker.`**`EventAttributeTracker`**`(`**args*, ***kwargs*`)`

The *Event Value Tracker* is extracting a value of an attribute of an event into a metric.

```
trackers:
  - class: tracker.EventAttributeTracker

    # The event filter to handle
    event: HTTPResponseEnd

    # One or more attributes to extract
    extract:

      # The metric where to write the result
      - metric: metricName

        # [OR] The attribute to extract the value from
        attrib: attribName

        # [OR] The expression to use to evaluate a value from the event
        eval: "event.attribute1 + event.attribute2"

    # [Optional] Extract the trace ID from the event(s) that match the
    # given filter. If missing, the trace ID of the event is used.
    traceIdFrom: ParameterUpdateEvent
```

This tracker is frequently used in conjunction with observers that broadcast measurements as single events.

For example you can use this tracker to extract JMX measurements as metrics:

```
trackers:
  - class: tracker.EventAttributeTracker
    event: JMXMeasurement
    extract:
      - metric: metricName
        attrib: "fields['jmxFieldName']""
```

Or you can extract raw log line messages as a metric:

```
trackers:
  - class: tracker.EventAttributeTracker
    event: LogLineEvent
    extract:
      - metric: metricName
        attrib: "line"
```

8.4.3 CountTracker

class `performance.driver.classes.tracker.`**`CountTracker`**(**args*, ***kwargs*)

Tracks the occurrences of an `event` within the tracking session.

```
trackers:
  - class: tracker.CountTracker

    # The metric where to write the measured value to
    metric: someMetric

    # The event to count
    # (This can be a filter expression)
    events: SomeEvent
    # [Optional] The increment step (this can be a python expression)
    step: 1
```

This tracker always operates within a tracking session, initiated by a `ParameterUpdateEvent` and terminated by the next `ParameterUpdateEvent`, or the completion of the test.

Important: The `event` must contain the trace IDs of the originating `ParameterUpdateEvent`, otherwise the events won't be measured.

8.4.4 DumpMetricTracker

class `performance.driver.classes.tracker.`**`DumpMetricTracker`**`(`**args*, ***kwargs*`)`

The *Dump Metric Tracker* is dumping metrics collected by observers into the results.

```
trackers:
  - class: tracker.DumpMetricTracker

    # The mapping between the marathon metric and the configured metric
    map:
      gauges.jvm.memory.total.used.value: marathonMemTotalUsage
      gauges.jvm.memory.heap.used.value: marathonMemHeapUsage
      gauges.jvm.threads.count.value: marathonThreadsCount
      gauges.jvm.threads.blocked.count.value: marathonThreadsBlocked
      gauges.jvm.threads.waiting.count.value: marathonThreadsWaiting
```

This tracker is simply translating the name of the metric collected by an observer (usually the `MarathonMetricsObserver`) into the metric collected by the scale test.

8.4.5 LogStaxTracker

class `performance.driver.classes.tracker.`**`LogStaxTracker`**`(`**args*, ***kwargs*`)`

The *Logstax Tracker* is forwarding the values of the LogStax tokens as result metrics.

```
trackers:
  - class: tracker.LogStaxTracker

    # Which tokens to collect
    collect:

      # The name of the metric to store the resulting value
      - metric: nameOfMetric

        # A python expression evaluated at run-time and gives the value
        # to assign to the metric. You can use all definitions, parameters,
        # and field values in your scope
        value: "fieldInMessage * parameter / definition"

        # [Optional] Extract the trace ID from the event(s) that match the
        # given filter.
        traceIdFrom: ParameterUpdateEvent

        # [Optional] The filter to apply on LogStax messages
        filter:

          # [Optional] The message should have all the specified tags present
          all_tags: [ foo, bar ]
          tags: [ foo, bar ]

          # [Optional] The message should have some of the specified tags present
          some_tags: [ baz, bax ]

          # [Optional] The message should have all the specified fields present
          all_fields: [ foo, bar ]

          # [Optional] The message should have some of the specified fields␣
↪present
          some_fields: [ foo, bar ]

          # [Optional] The message should have the given field values
          fields:
            foo: "foovalue"
```

You can use this tracker in combination with `LogStaxObserver` in order to collect useful tokens present in the log lines of the application being tested.

8.5 Tasks

The `tasks` classes contain micro-actions that are executed at some phase of the test and do not participate in the final result.

These tasks are executed on a specified trigger, through the *at:* configuration parameter. For example:

```
tasks:

  - at: setup
    class: ...
    ...
```

8.5.1 Known Triggers

The following table summarises the task triggers available in the driver.

Task Name	Source	Description
`setup`	Session	Called when the sytem is ready and right before the first policy is started. Use this trigger to initialize your app state.
`pretest`	Session	Called before every run is started. Use this trigger to wipe the state before the tests are started.
`posttest`	Session	Called right after every run. Use this trigger to clean-up your system between the runs.
`teardown`	Session	Called when the system has finished all tests and is about to start reporting. Use this trigger to clean-up your system.
`intertest`	Policy	Called inbetween the tests, when a parameter changes. This implementation depends on the policy you are using. Usually you should use this trigger to bring your system into a known state right before every value is applied.

8.5.2 auth.AuthEE

class `performance.driver.classes.tasks.auth.`**`AuthEE`**(*config*, *eventbus*)
Authenticate against an Enterprise-Edition cluster

```
tasks:
  - class: tasks.auth.AuthEE
    at: ...

    # The username to authenticate against
    user: bootstrapuser

    # The password to use
    password: deleteme

    # [Optional] The base cluster URL
    # Instead of specifying this configuration parameter you can specify
    # the `cluster_url` definition (not recommended)
    cluster_url: "https://cluster.dcos.io"
```

This task authenticates against the enterprise cluster and obtains an authentication token.

This task sets the `dcos_auth_token` definition and makes it available for other components to use.

8.5.3 auth.AuthOpen

class `performance.driver.classes.tasks.auth.`**`AuthOpen`**(*config*, *eventbus*)
Authenticate against an Open-Source Edition cluster

```
tasks:
  - class: tasks.auth.AuthOpen
    at: ...

    # The user token to (re-)use
    token: bootstrapuser

    # [Optional] The base cluster URL
    # Instead of specifying this configuration parameter you can specify
    # the `cluster_url` definition (not recommended)
    cluster_url: "https://cluster.dcos.io"
```

This task authenticates against the enterprise cluster and obtains an authentication token.

This task sets the `dcos_auth_token` definition and makes it available for other components to use.

8.5.4 http.Request

class `performance.driver.classes.tasks.http.`**`Request`**(*config*, *eventbus*)
Perform an arbitrary HTTP request as a single-shot task

```
tasks:
  - class: tasks.http.Request
    at: ...

    # The URL to send the requests at
    url: http://127.0.0.1:8080/v2/apps
    # [Optional] The body of the HTTP request
    body: |
      {
        "cmd": "sleep 1200",
        "cpus": 0.1,
        "mem": 64,
        "disk": 0,
        "instances": {{instances}},
        "id": "/scale-instances/{{uuid()}}",
        "backoffFactor": 1.0,
        "backoffSeconds": 0
      }

    # [Optional] The HTTP Verb to use (Defaults to 'GET')
    verb: POST

    # [Optional] How many times to repeat the same HTTP request
    # Note that in this case you can use the {{_i}} macro
    repeat: 10

    # [Optional] The HTTP headers to send
    headers:
      Accept: text/plain
```

Note: This channel will automatically inject an `Authorization` header if a `dcos_auth_token` definition exists, so you don't have to specify it through the `headers` configuration.

Note that a `dcos_auth_token` can be dynamically injected via an authentication task.

8.5.5 marathon.RemoveAllApps

class `performance.driver.classes.tasks.marathon.`**`RemoveAllApps`**`(`**args*, ***kwargs*`)`

Remove matching apps from marathon

```
tasks:
  - class: tasks.marathon.RemoveAllApps
    at: ...

    # The base url to marathon
    url: "{{marathon_url}}"

    # [Optional] Additional headers to include to the marathon request
    headers:
      x-Originating-From: Python
```

This task is enumerating all apps in the root group and delets each one of them.

Note: This task will block the execution of other tasks until all deployments are completed. This is intentional in order allow other tasks to be executed in series.

8.5.6 marathon.RemoveMatchingApps

class `performance.driver.classes.tasks.marathon.`**`RemoveMatchingApps`**(**args*, ***kwargs*)

Removes matching apps from marathon

```
tasks:
  - class: tasks.marathon.RemoveMatchingApps
    at: ...

    # The base url to marathon
    url: "{{marathon_url}}"

    # The string portion in the app name to match
    match: "test-01-"

    # [Optional] Additional headers to include to the marathon request
    headers:
      x-Originating-From: Python
```

This task is enumerating all apps in the root group, checking wich ones contain the string contained in the `match` parameter and removes them.

Note: This task will block the execution of other tasks until all deployments are completed. This is intentional in order allow other tasks to be executed in series.

8.5.7 marathon.RemoveGroup

class `performance.driver.classes.tasks.marathon.`**`RemoveGroup`**(**args*, ***kwargs*)

Removes a specific group from marathon

```
tasks:
  - class: tasks.marathon.RemoveGroup
    at: ...

    # The base url to marathon
    url: "{{marathon_url}}"

    # The group to remove
    group: "tests/01"

    # [Optional] Additional headers to include to the marathon request
    headers:
      x-Originating-From: Python
```

This task removes the given group from marathon.

Note: This task will block the execution of other tasks until all deployments are completed. This is intentional in order allow other tasks to be executed in series.

8.6 Reporters

The `reporter` classes are **reporting the results** into a human-readable, or machine-processable format.

8.6.1 CSVReporter

class `performance.driver.classes.reporter.`**`CSVReporter`**(*config*, *generalConfig*, *eventbus*)

The **CSV Reporter** is creating a comma-separated value (.csv) file with the axis values and summarised metric values for every run.

```
reporters:
  - class: reporter.CSVReporter

    # [Optional] The filename to write the csv file into
    filename: results.csv

    # [Optional] The column separator character to use
    separator: ","

    # [Optional] Which value to use if a parameter is missing
    default: 0
```

This reporter is writing the **summarised** results in a CSV file. The resulting file will have the following columns:

Parameters			Summarised Metrics			Flags		
p1	p2	…	m1 (sum)	m2 (sum)	…	f1	f2	…

The first line will contain the names of the parameters, metrics and flags.

Note: To configure which summariser(s) to use on every metric, use the `summarize` parameter in the *config.metrics* config.

8.6.2 DataDogReporter

class `performance.driver.classes.reporter.`**`DataDogReporter`**(*config*, *generalConfig*, *eventbus*)

The **DataDog Reporter** is uploading the indicators into DataDog for archiving and alerting usage.

```
reporters:
  - class: reporter.DataDogReporter

    # The API Key to use
    api_key: 1234567890abcdef

    # The App Key to use
    app_key: 1234567890abcdef

    # The data points to submit
    points:

      # The name of the metric to submit to DataDog and the
      # indicator to read the data from
      - name: memory
        indicator: meanMemoryUsage

    # [Optional] The hostname to use as the agent name in datadog
    # If missing the network name of the machine will be used
    hostname: test.host

    # [Optional] Prefix of the metrics (Defaults to `dcos.perf.`)
    prefix: "dcos.perf."
```

The DataDog reporter is using the DataDog API to submit one or more indicator values as data points.

Note: This reporter is **only** collecting the `indicators`. Metric values or summaries cannot be reported to DataDog. Use the `reporter.DataDogMetricReporter` instead.

8.6.3 DataDogMetricReporter

class `performance.driver.classes.reporter.`**`DataDogMetricReporter`**(*config*, *generalConfig*, *eventbus*)

The **DataDog Metric Reporter** uploads the raw metric values to DataDog the moment they are collected.

```
reporters:
  - class: reporter.DataDogMetricReporter

    # The API Key to use
    api_key: 1234567890abcdef

    # [Optional] The hostname to use as the agent name in datadog
    # If missing the network name of the machine will be used
    hostname: test.host

    # [Optional] Prefix of the metrics (Defaults to `dcos.perf.`)
    prefix: "dcos.perf."

    # [Optional] How frequently to flush the metrics to DataDog
    flushInterval: 5s

    # [Optional] Report configuration. If missing, the default behavior
    # is to report the summarized metrics at the end of the test.
    report:

      # [Optional] The string `all` indicates that all the metrics should
      # be submitted to DataDog
      metrics: all

      # [Optional] OR a list of metric names can be provided
      metrics:
        - metricA
        - metricB

      # [Optional] OR you can use a dictionary to provide an alias
      metrics:
        metricA: aliasedMetricA
        metricB: aliasedMetricB
      # [Optional] Set to `yes` to submit the raw values the moment they
      # are collected.
      raw: yes

      # [Optional] Set to `yes` to submit summarized values of the metrics
      # at the end of the test run.
      summarized: yes

      # [Optional] The string `all` indicates that all the indicators should
      # be submitted to DataDog
      indicators: all

      # [Optional] OR a list of indicator names can be provided
      indicators:
        - indicatorA
        - indicatorB

      # [Optional] OR you can use a dictionary to provide an alias
      indicators:
        indicatorA: aliasedIndicatorA
        indicatorB: aliasedIndicatorB
```

The DataDog reporter is using the DataDog API to submit the values of the test metrics to DataDog in real-time.

8.6.4 PlotReporter

class `performance.driver.classes.reporter.`**`PlotReporter`**(*config*, *generalConfig*, *eventbus*)

The **Plot Reporter** is creating a PNG plot with the measured values and storing it in the results folder.

```
reporters:
  - class: reporter.PlotReporter

    # [Optional] Default parameter value to use if not specified
    default: 0

    # [Optional] Filename prefix and suffix (without the extension)
    prefix: "plot-"
    suffix: ""

    # [Optional] The X and Y axis scale (for all plots)
    # Can be one of: 'linear', 'log', 'log2', 'log10'
    xscale: linear
    yscale: log2

    # [Optional] The colormap to use when plotting 2D plots
    # Valid options from: https://matplotlib.org/examples/color/colormaps_
↪reference.html
    colormap: winter

    # [Optional] Plot the raw values as a scatter plot and not the summarised
    raw: False

    # [Optional] Reference data structure
    reference:

      # Path to raw reference JSON
      data: http://path.to/refernce-raw.json

      # [Optional] The colormap to use when plotting the reference 2D plots
      ratiocolormap: bwr

      # [Optional] Name of the reference data
      name: ref

      # [Optional] Headers to send along with the request
      headers:
        Authentication: "token={{token}}"
```

This reporter will generate an image plot for every metric defined. The axis is the 1 or 2 parameters of the test.

Warning: The `PlotReporter` can be used only if the total number of parameters is 1 or 2, since it's not possible to display plots with more than 3 axes.

Trying to use it otherwise will result in an exception being thrown.

8.6.5 PostgRESTReporter

class `performance.driver.classes.reporter.`**`PostgRESTReporter`**(*config*, *generalConfig*, *eventbus*)

The **PostgREST Reporter** is uploading the full set of results in a structured manner in a Postgres database using a PostgREST API endpoint.

```
reporters:
  - class: reporter.PostgRESTReporter

    # The URL to the PostgREST endpoint
    url: "http://127.0.0.1:4000"

    # [Optional] The database table prefix
    prefix: "profile_data_"
```

This reporter is uploading the following information

Important: The Postgres database is using uuid-based lookup for every parameter and metric. Therefire it's required to include the `uuid` parameter in the *config.metrics* and *config.parameters* configuration.

This de-couples the representation of the metric across different projects or versions of the same project.

Postgres SQL Schema

This document explains the table schema the `PostgRESTReporter` reporter is expected to which to report the data.

Terminology

- **job** : A scale test job
- **run** : A repeating scale test sampling process in order to collect statistics. One or more *runs* are executed within a single *job*.

- **test** : A scale test that is executed during a *run*
- **phase** : A checkpoint during the execution of a *test* during which a new set of *parameters* is given to the application and a new sampling process begins.
- **parameter** : An input value to the *test*. For example `instances=10`.
- **mmetric** : An output value from the *test*. For example `deployTime=1.42s`

Tables

`*_job` Job Indexing Table

This table keeps track of the high-level job structure. Since more than one project will be using the same database, the `project` field should be populated with the name of the project that started this job.

DDL

```
CREATE TABLE metric_data.perf_test_job
(
    jid uuid NOT NULL,
    started timestamp without time zone NOT NULL,
    completed timestamp without time zone NOT NULL,
    status integer NOT NULL,
    project character varying(128) NOT NULL,
    PRIMARY KEY (jid)
)
WITH (
    OIDS = FALSE
);

ALTER TABLE metric_data.perf_test_job
    OWNER to postgrest;
```

`*_job_meta` Job Metadata

Each job has a set of metadata that can be used to identify the process being executed. For example `environment`, `version`, `git_hash` etc.

They are unique for every run, therefore they are groupped with the run ID.

DDL

```
CREATE TABLE metric_data.perf_test_job_meta
(
    id serial NOT NULL,
    jid uuid NOT NULL,
    name character varying(32) NOT NULL,
    value character varying(128) NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT jid FOREIGN KEY (jid)
        REFERENCES metric_data.perf_test_job (jid) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
);

ALTER TABLE metric_data.perf_test_job_meta
    OWNER to postgrest;
```

`*_job_phases` Job Phases

Eventually the test will go through various *phases* that are repeated during every *run*. Since the *phase* is groupping various parameter/metric combinations, we are using the `job_phases` table to index them.

(This table could actually be merged into the `phase_` tables below)

DDL

```
CREATE TABLE metric_data.perf_test_job_phases
(
    pid uuid NOT NULL,
    jid uuid NOT NULL,
    run integer NOT NULL,
    "timestamp" timestamp without time zone NOT NULL,
    PRIMARY KEY (pid),
    CONSTRAINT jid FOREIGN KEY (jid)
        REFERENCES metric_data.perf_test_job (jid) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
);

ALTER TABLE metric_data.perf_test_job_phases
    OWNER to postgresql;
```

`*_lookup_metrics` Metric lookup table

Since a metric might be renamed or changed over time, we are using UUIDs to refer to metrics. This table contains the lookup information between the UUID and the metric name.

```
CREATE TABLE metric_data.perf_test_lookup_metrics
(
    metric uuid NOT NULL,
    name character varying(32) NOT NULL,
    title character varying(128) NOT NULL,
    units character varying(16),
    PRIMARY KEY (metric)
)
WITH (
    OIDS = FALSE
);

ALTER TABLE metric_data.perf_test_lookup_metrics
    OWNER to postgrest;
```

`*_lookup_parameters` Parameter lookup table

Like the *lookup metrics* table, this table contains the lookup information between the UUID and the parameter name.

```
CREATE TABLE metric_data.perf_test_lookup_parameters
(
    parameter uuid NOT NULL,
    name character varying(32) NOT NULL,
    title character varying(128) NOT NULL,
    units character varying(16),
    PRIMARY KEY (parameter)
)
WITH (
    OIDS = FALSE
);

ALTER TABLE metric_data.perf_test_lookup_parameters
    OWNER to postgrest;
```

`*_phase_flags` Phase Flags

During each *phase* one or more status *flags* might be raised, indicating internal failures or other status information. These flags are submitted when the phase is completed and it's useful to collect them.

```
CREATE TABLE metric_data.perf_test_phase_flags
(
    id serial NOT NULL,
    pid uuid NOT NULL,
    name character varying(32) NOT NULL,
    value character varying(128) NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT pid FOREIGN KEY (pid)
        REFERENCES metric_data.perf_test_job_phases (pid) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
);

ALTER TABLE metric_data.perf_test_phase_flags
    OWNER to postgrest;
```

`*_phase_params` Phase Parameters

During each *phase* the *test* is given some *parameters*. These *parameters* are usually the plot axis that we are interested in. (ex. `instances`)

DDL

```
CREATE TABLE metric_data.perf_test_phase_params
(
    id serial NOT NULL,
    pid uuid NOT NULL,
    parameter uuid NOT NULL,
    value character varying(128) NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT pid FOREIGN KEY (pid)
        REFERENCES metric_data.perf_test_job_phases (pid) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT parameter FOREIGN KEY (parameter)
        REFERENCES metric_data.perf_test_lookup_parameters (parameter) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
);

ALTER TABLE metric_data.perf_test_phase_flags
    OWNER to postgrest;
```

`*_phase_metrics` Phase Metrics

During the *test* various *metrics* are extracted and emmited the moment their sampling is completed. These metrics are effectively the results of the test.

DDL

```
CREATE TABLE metric_data.perf_test_phase_metrics
(
    id serial NOT NULL,
    pid uuid NOT NULL,
    metric uuid NOT NULL,
    value numeric NOT NULL,
    timestamp timestamp without time zone NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT pid FOREIGN KEY (pid)
        REFERENCES metric_data.perf_test_job_phases (pid) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT metric FOREIGN KEY (metric)
        REFERENCES metric_data.perf_test_lookup_metrics (metric) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
);
```

```
ALTER TABLE metric_data.perf_test_phase_flags
    OWNER to postgrest;
```

Querying

The following query can be used to fetch a 1D plot for jobs that have only 1 axis on it's parameters:

```
SELECT
    "metric_data"."perf_test_job_phases".jid,
    "metric_data"."perf_test_phase_params"."value" AS "x",
    "metric_data"."perf_test_phase_metrics"."value" AS "y"

FROM
    "metric_data"."perf_test_phase_params"
    JOIN "metric_data"."perf_test_phase_metrics"
        ON "metric_data"."perf_test_phase_params".pid =
           "metric_data"."perf_test_phase_metrics".pid
    JOIN "metric_data"."perf_test_job_phases"
        ON "metric_data"."perf_test_phase_params".pid =
           "metric_data"."perf_test_job_phases".pid
WHERE
    -- The axis you want to view (assuming only 1 dimention)
    "metric_data"."perf_test_phase_params"."parameter" =
        '4a003e85-e8bb-4a95-a340-eec1727cfd0d' AND

    -- The metric you want to plot
    "metric_data"."perf_test_phase_metrics"."metric" =
        'cfac77fc-eb24-4862-aedd-89066441c416' AND

    -- Job selection based on it's metadata.
    -- In this example we are selecting the latest `master` version.
    "metric_data"."perf_test_job_phases".jid IN (
        SELECT
            "metric_data"."perf_test_job_meta".jid
        FROM
            "metric_data"."perf_test_job_meta"
        WHERE
            "metric_data"."perf_test_job_meta"."name" = 'version' AND
            "metric_data"."perf_test_job_meta"."value" = 'master'
        ORDER BY
            "metric_data"."perf_test_job_meta".id DESC
        LIMIT 1
    )
```

8.6.6 RawReporter

class `performance.driver.classes.reporter.`**`RawReporter`**`(*args, **kwargs)`

The **Raw Reporter** is creating a raw dump of the results in the results folder in JSON format.

```
reporters:
  - class: reporter.RawReporter

    # Where to dump the results
    filename: "results-raw.json"
    # [Optional] Include event traces
    events:

      # [Optional] Include events that pass through the given expression
      include: FilterExpression

      # [Optional] Exclude events that pass through the given expression
      exclude: FilterExpression

      # [Optional] Group the events to their traces
      traces: yes
```

The JSON structure of the data included is the following:

```
{

  // Timing information
  "time": {
    "started": "",
    "completed": ""
  },

  // The configuration used to run this test
  "config": {
    ...
  },

  // The values for the indicators
  "indicators": {
    "indicator": 1.23,
    ...
  },

  // The metadata of the run
  "meta": {
    "test": "1-app-n-instances",
    ...
  },

  // Raw dump of the timeseries for every phase
  "raw": [
    {

      // One or more status flags collected in this phase
      "flags": {
        "status": "OK"
      },

      // The values of all parameter (axes) in this phase
      "parameters": {
        "apps": 1,
        "instances": 1
      },

      // The time-series values for every phase
      "values": {
        "metricName": [
          // Each metric is composed of the timestamp of it's
          // sampling time and the value
          [
            1499696193.822527,
            11
          ],
          ...

        ]
      }
    }
  ],

  // Summarised dump of the raw timeseries above, in the same
  // structure
  "sum": [
    {

      // One or more status flags collected in this phase
      "flags": {
        "status": "OK"
      },

      // The values of all parameter (axes) in this phase
      "parameters": {
        "apps": 1,
        "instances": 1
      },

      // The summarised values of each timeseries
      "values": {
        "metricName": {

          // Here are the summarisers you selected in the `metric`
          // configuration parameter.
          "sum": 123.4,
          "mean": 123.4,
          ...
        }
      }
    }
  ]
}
```

8.6.7 RawEventsReporter

class `performance.driver.classes.reporter.`**`RawEventsReporter`**`(`**args*`)`

The **Raw Events Reporter** is dumping every event in the eventBus to a file that can be used for offline event processing. You can also use this reporter for debugging the performance driver internals.

```
reporters:
  - class: reporter.RawEventsReporter

    # [Optional] Set to `yes` to track TickEvents
    # Note that including tick events might introduce a lot of noise to
    # your data and/or increase the reporting impact.
    tickEvents: no

    # Where to dump the events
    filename: "events.dump"
```

The log file is encoded with the following rules:

1. The events are encoded in plain-text
2. Each event is separated with a new line
3. Each line contains two columns separated with semicolon
4. The first column contains the unix timestamp of the event
5. The second column contains the name of the event
6. The third column contains the field values for the event encoded as a JSON string.

For example:

```
//   Timestamp  //    Name    //      Properties   //
1500891843.976068;SomeEventName;{"prop":"value", ...}
...
```

This format allows for simple grepping and more elaborate parsing. For example

```
cat event.dump | grep ';TickEvent;' | wc -l
```

8.6.8 S3Reporter

class `performance.driver.classes.reporter.`**`S3Reporter`**(**args*, ***kwargs*)

The **S3 Reporter** is uploading a raw dump of the results in a bucket in Amazon's S3 services

```
reporters:
  - class: reporter.S3Reporter

    # The name of the S3 bucket
    bucket: dcos-perf-test-results

    # [Optional] If ommited, you must provide the AWS_ACCESS_KEY_ID
    # environment variable
    aws_access_key_id: ...

    # [Optional] If ommited, you must provide the AWS_SECRET_ACCESS_KEY
    # environment variable
    aws_secret_access_key: ...

    # [Optional] The path in the bucket where to save the file
    path: results-raw.json

    # [Optional] A canned ACL. One of: private, public-read,
    # public-read-write, authenticated-read, bucket-owner-read,
    # bucket-owner-full-control
    acl: private

    # [Optional] Put the filename uploaded on the given index file
    index:
      # The path to the index JSON file
      path: path/to/index.json

      # The index entry to update
      entry: some_key

      # [Optional] How many items to keep under this key
      max_entries: 100

      # [Optional] The bucket name if different than the above
      bucket: dcos-perf-test-results
```

This reporter behaves exactly like *RawReporter*, but the generated JSON blob is uploaded to an S3 bucket instead of a file in your local filesystem.

8.7 Indicators

The `indicator` classes compute a single scalar value from the results. This scalar value can be used as an *indicator* of the outcome fo the test.

8.7.1 NormalizedMeanMetricIndicator

class `performance.driver.classes.indicator.`**`NormalizedMeanMetricIndicator`**(*config*)

Calculates the average of the metrics of all runs, normalized by the given value of parameters.

```
# Note that this needs to be defined in the global `config` section
config:

  indicators:
    - class: indicator.NormalizedMeanMetricIndicator

      # The name of this indicator
      name: someIndicator

      # The metric and summarizer to use in <metric>.<summarizer>
      # format
      metric: someMetric.someSummarizer

      # Normalize each test case measurement to the specified parameter
      normalizeto: normalizeAgainst

      # [Optional] You could also use a python expression. The parameters
      # and the global definitions are available as global variables
      normalizeto: 0.8 * normalizeAgainst
```

8.7.2 NormalizedMinMetricIndicator

class `performance.driver.classes.indicator.`**`NormalizedMeanMetricIndicator`**(*config*)

Calculates the average of the metrics of all runs, normalized by the given value of parameters.

```
# Note that this needs to be defined in the global `config` section
config:

  indicators:
    - class: indicator.NormalizedMeanMetricIndicator

      # The name of this indicator
      name: someIndicator

      # The metric and summarizer to use in <metric>.<summarizer>
      # format
      metric: someMetric.someSummarizer

      # Normalize each test case measurement to the specified parameter
      normalizeto: normalizeAgainst

      # [Optional] You could also use a python expression. The parameters
      # and the global definitions are available as global variables
      normalizeto: 0.8 * normalizeAgainst
```

8.7.3 NormalizedMaxMetricIndicator

class `performance.driver.classes.indicator.`**`NormalizedMeanMetricIndicator`**(*config*)
Calculates the average of the metrics of all runs, normalized by the given value of parameters.

```
# Note that this needs to be defined in the global `config` section
config:

  indicators:
    - class: indicator.NormalizedMeanMetricIndicator

      # The name of this indicator
      name: someIndicator

      # The metric and summarizer to use in <metric>.<summarizer>
      # format
      metric: someMetric.someSummarizer

      # Normalize each test case measurement to the specified parameter
      normalizeto: normalizeAgainst

      # [Optional] You could also use a python expression. The parameters
      # and the global definitions are available as global variables
      normalizeto: 0.8 * normalizeAgainst
```

8.8 Summarizers

The `summarizer` classes contains the functions to calculate a summary value for the given time series.

8.8.1 BuiltInSummarizer

class `performance.driver.core.classes.summarizer.`**`BuiltInSummarizer`**(*config*)
A proxy class that calls the built-in summarizer functions

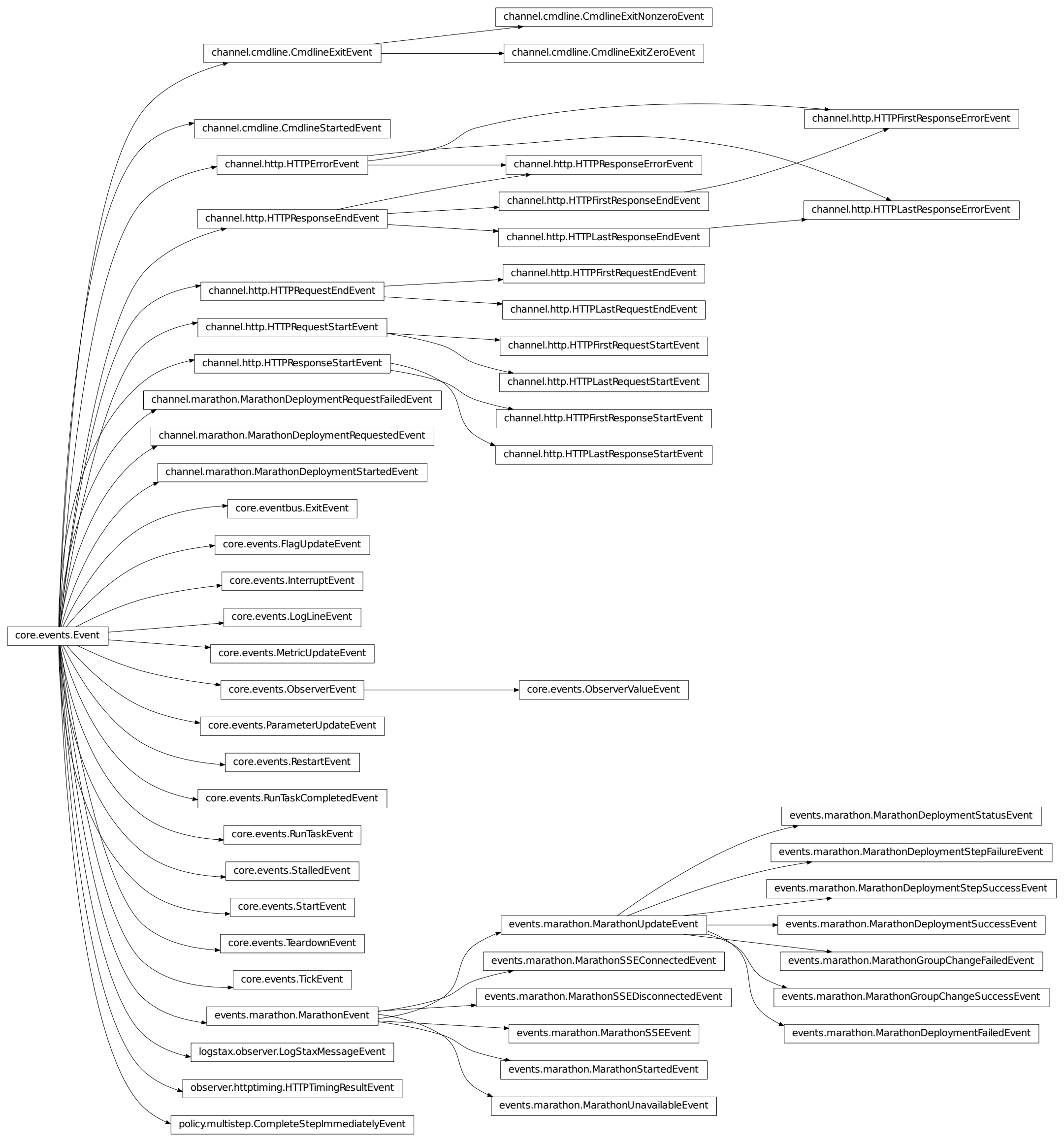
```
# Can be used without configuration, like so:
metrics:
  - name: metric
    ...
    summarize: [mean, min, ]

# Or with configuration like so:
metrics:
  - name: metric
    ...
    summarize:
      - class @mean

        # The name of the metric in the plots
        name: mean

        # [Optional] Set to `yes` to include outliers
        outliers: no
```

The following built-in summarizers are available:

- `mean` : Calculate the mean value of the timeseries
- `mean_err` : Calculate the mean value, including statistical errors
- `min` : Find the minimum value
- `max` : Find the maximum value
- `sum` : Calculate the sum of all timeseries
- `median` : Calculate the median of the timeseries
- `mode` : Calculate the mode of the timeseries
- `variance` : Calculate the variance of the timeseries
- `sdeviation` : Calculate the standard deviation of the timeseries

8.9 Event Reference

This is a reference to all events broadcasted in the internal event bus, including their available attributes.


channel.cmdline.CmdlineExitNonzeroEvent
channel.cmdline.CmdlineExitEvent
channel.cmdline.CmdlineExitZeroEvent
channel.http.HTTPFirstResponseErrorEvent
channel.cmdline.CmdlineStartedEvent
channel.http.HTTPErrorEvent
channel.http.HTTPResponseErrorEvent
channel.http.HTTPFirstResponseEndEvent
channel.http.HTTPLastResponseErrorEvent
channel.http.HTTPResponseEndEvent
channel.http.HTTPLastResponseEndEvent
channel.http.HTTPFirstRequestEndEvent
channel.http.HTTPRequestEndEvent
channel.http.HTTPLastRequestEndEvent
channel.http.HTTPRequestStartEvent
channel.http.HTTPFirstRequestStartEvent
channel.http.HTTPResponseStartEvent
channel.http.HTTPLastRequestStartEvent
channel.marathon.MarathonDeploymentRequestFailedEvent
channel.http.HTTPFirstResponseStartEvent
channel.marathon.MarathonDeploymentRequestedEvent
channel.http.HTTPLastResponseStartEvent
channel.marathon.MarathonDeploymentStartedEvent
core.eventbus.ExitEvent
core.events.FlagUpdateEvent
core.events.InterruptEvent
core.events.LogLineEvent
core.events.Event
core.events.MetricUpdateEvent
core.events.ObserverEvent
core.events.ObserverValueEvent
core.events.ParameterUpdateEvent
core.events.RestartEvent
core.events.RunTaskCompletedEvent
events.marathon.MarathonDeploymentStatusEvent
core.events.RunTaskEvent
events.marathon.MarathonDeploymentStepFailureEvent
core.events.StalledEvent
events.marathon.MarathonDeploymentStepSuccessEvent
core.events.StartEvent
events.marathon.MarathonUpdateEvent
events.marathon.MarathonDeploymentSuccessEvent
core.events.TeardownEvent
events.marathon.MarathonSSEConnectedEvent
events.marathon.MarathonGroupChangeFailedEvent
core.events.TickEvent
events.marathon.MarathonSSEDisconnectedEvent
events.marathon.MarathonGroupChangeSuccessEvent
events.marathon.MarathonEvent
events.marathon.MarathonSSEEvent
events.marathon.MarathonDeploymentFailedEvent
logstax.observer.LogStaxMessageEvent
events.marathon.MarathonStartedEvent
observer.httptiming.HTTPTimingResultEvent
events.marathon.MarathonUnavailableEvent
policy.multistep.CompleteStepImmediatelyEvent

8.9.1 Event Details

class `performance.driver.classes.channel.cmdline.`**`CmdlineExitEvent`**(*exitcode*, ***kwargs*)

This event is published when the process launched through the cmdline channel has completed. The exit code is tracked.

`exitcode` = **None**

The exit code of the application launched by the command-line channel

class `performance.driver.classes.channel.cmdline.`**`CmdlineExitNonzeroEvent`**(*exitcode*, ***kwargs*)

This event is published when the process exited and the exit code is non-zero

class `performance.driver.classes.channel.cmdline.`**`CmdlineExitZeroEvent`**(*exitcode*, ***kwargs*)

This event is published when the process exited and the exit code is zero

class `performance.driver.classes.channel.cmdline.`**`CmdlineStartedEvent`**(*pid*, **args*, ***kwargs*)

This event is published when the process has started. It contains the process ID so the observers can attach to the process and extract useful data.

class `performance.driver.classes.channel.http.`**`HTTPErrorEvent`**(*exception*, **args*, ***kwargs*)

Published when an exception is raised during an HTTP operation (ex. connection error)

`exception` **= None**

The exception that was raised

class `performance.driver.classes.channel.http.`**`HTTPFirstRequestEndEvent`**(*verb*, *url*, *body*, *headers*, **args*, ***kwargs*)

Published when the first request out of many is completed. This is valid when a `repeat` parameter has a value > 1.

class `performance.driver.classes.channel.http.`**`HTTPFirstRequestStartEvent`**(*verb*, *url*, *body*, *headers*, **args*, ***kwargs*)

Published when the first request out of many is started. This is valid when a `repeat` parameter has a value > 1.

class `performance.driver.classes.channel.http.`**`HTTPFirstResponseEndEvent`**(*url*, *body*, *headers*, **args*, ***kwargs*)

Published when the first response out of many has completed. This is valid when a `repeat` parameter has a value > 1.

class `performance.driver.classes.channel.http.`**`HTTPFirstResponseErrorEvent`**(*url*, *body*, *headers*, *exception*, **args*, ***kwargs*)

Published when the first response out of many has an error. This is valid when a `repeat` parameter has a value > 1.

class `performance.driver.classes.channel.http.`**`HTTPFirstResponseStartEvent`**(*url*, **args*, ***kwargs*)

Published when the first response out of many is starting. This is valid when a `repeat` parameter has a value > 1.

class `performance.driver.classes.channel.http.`**`HTTPLastRequestEndEvent`**(*verb*, *url*, *body*, *headers*, **args*, ***kwargs*)

Published when the last request out of many is completed. This is valid when a `repeat` parameter has a value > 1.

class `performance.driver.classes.channel.http.`**`HTTPLastRequestStartEvent`**(*verb*, *url*, *body*, *headers*, **args*, ***kwargs*)

Published when the last request out of many is started. This is valid when a `repeat` parameter has a value > 1.

class `performance.driver.classes.channel.http.`**`HTTPLastResponseEndEvent`**(*url*, *body*, *headers*, **args*, ***kwargs*)

Published when the last response out of many has completed. This is valid when a `repeat` parameter has a value > 1.

class `performance.driver.classes.channel.http.`**`HTTPLastResponseErrorEvent`**(*url*, *body*, *headers*, *exception*, **args*, ***kwargs*)

Published when the last response out of many has an error. This is valid when a `repeat` parameter has a value > 1.

class `performance.driver.classes.channel.http.`**`HTTPLastResponseStartEvent`**(*url*, **args*, ***kwargs*)

Published when the last response out of many is starting. This is valid when a `repeat` parameter has a value > 1.

class `performance.driver.classes.channel.http.`**`HTTPRequestEndEvent`**(*verb*, *url*, *body*, *headers*, **args*, ***kwargs*)

Published when the HTTP request has completed and the response is starting

`body` **= None**

The request body

`headers` **= None**

The request headers

`url` = None
The URL requested

`verb` = None
The HTTP verb that was used (in lower-case). Ex: `get`

class `performance.driver.classes.channel.http.`**`HTTPRequestStartEvent`**(*verb*, *url*, *body*, *headers*, **args*, ***kwargs*)

Published before every HTTP request

`body` = None
The request body

`headers` = None
The request headers

`url` = None
The URL requested

`verb` = None
The HTTP verb that was used (in lower-case). Ex: `get`

class `performance.driver.classes.channel.http.`**`HTTPResponseEndEvent`**(*url*, *body*, *headers*, **args*, ***kwargs*)

Published when the HTTP response has completed

`body` = None
The response body (as string)

`headers` = None
The response headers

`url` = None
The URL requested

class `performance.driver.classes.channel.http.`**`HTTPResponseErrorEvent`**(*url*, *body*, *headers*, *exception*, **args*, ***kwargs*)

Published when an exception was raised while processing an HTTP response. This is valid when a `repeat` parameter has a value = 1.

class `performance.driver.classes.channel.http.`**`HTTPResponseStartEvent`**(*url*, **args*, ***kwargs*)

Published when the HTTP response is starting.

`url` = None
The URL requested

class `performance.driver.classes.channel.marathon.`**`MarathonDeploymentRequestFailedEvent`**(*instance*
sta-
tus_code
re-
spose, **args*, ***kwarg*

class `performance.driver.classes.channel.marathon.`**`MarathonDeploymentRequestedEvent`**(*instance*, **args*, ***kwargs*)

class `performance.driver.classes.channel.marathon.`**`MarathonDeploymentStartedEvent`**(*instance*, **args*, ***kwargs*)

class `performance.driver.classes.observer.events.marathon.`**`MarathonDeploymentFailedEvent`**(*deploy*
af-
fecte-
dIn-
stances
**args*,
***kwa*

class `performance.driver.classes.observer.events.marathon.`**`MarathonDeploymentStatusEvent`**(*deploy*
af-
fecte-
dIn-
stances
**args*,
***kwa*

class `performance.driver.classes.observer.events.marathon.`**`MarathonDeploymentStepFailureEvent`**

class `performance.driver.classes.observer.events.marathon.`**`MarathonDeploymentStepSuccessEvent`**

class `performance.driver.classes.observer.events.marathon.`**`MarathonDeploymentSuccessEvent`**(*deplo*
af-
fecte-
dIn-
stanc
**args*
***kw*

class `performance.driver.classes.observer.events.marathon.`**`MarathonEvent`**(*traceid=None*, *ts=None*)

Base class for all marathon-related events

class `performance.driver.classes.observer.events.marathon.`**`MarathonGroupChangeFailedEvent`**(*deplo group rea- son, *args **kw*

class `performance.driver.classes.observer.events.marathon.`**`MarathonGroupChangeSuccessEvent`**(*dep grou *arg **k*

class `performance.driver.classes.observer.events.marathon.`**`MarathonSSEConnectedEvent`**(*traceid=Non ts=None*)

Raw SSE endpoint was connected

class `performance.driver.classes.observer.events.marathon.`**`MarathonSSEDisconnectedEvent`**(*traceid= ts=None*

Raw SSE endpoint was disconnected

class `performance.driver.classes.observer.events.marathon.`**`MarathonSSEEvent`**(*eventName, event- Data, *args, **kwargs*)

Raw SSE event

class `performance.driver.classes.observer.events.marathon.`**`MarathonStartedEvent`**(*traceid=None, ts=None*)

Marathon is up and accepting HTTP requests

class `performance.driver.classes.observer.events.marathon.`**`MarathonUnavailableEvent`**(*traceid=None, ts=None*)

Marathon is up and accepting HTTP requests

class `performance.driver.classes.observer.events.marathon.`**`MarathonUpdateEvent`**(*deployment, in- stances, *args, **kwargs*)

Base class for update events

class `performance.driver.classes.observer.httptiming.`**`HTTPTimingResultEvent`**(*url*, *verb*, *statusCode*, *requestTime*, *responseTime*, *totalTime*, *contentLength*, **args*, ***kwargs*)

The results of a timing event, initiated by a `HTTPTimingObserver`

`contentLength` **= None**
The length of the response body

`requestTime` **= None**
The time the HTTP request took to complete

`responseTime` **= None**
The time the HTTP response took to complete

`statusCode` **= None**
The HTTP response code

`totalTime` **= None**
The overall time from the beginning of the request, till the end of the response

`url` **= None**
The URL requested

`verb` **= None**
The HTTP verb used to request this resource

class `performance.driver.classes.observer.logstax.observer.`**`LogStaxMessageEvent`**(*message*, ***kwargs*)

class `performance.driver.classes.policy.multistep.`**`CompleteStepImmediatelyEvent`**(*traceid=None*, *ts=None*)

class `performance.driver.core.eventbus.`**`ExitEvent`**(*traceid=None*, *ts=None*)
A local event that instructs the main event loop to exit

class `performance.driver.core.events.`**`Event`**(*traceid=None*, *ts=None*)
Base event

The *traceid* parameter is a unique string or object that is carried along related events and is used to group them together to the same operation.

`hasTrace`(*traceid*)
Check if the event was emmited from the given ID

hasTraces(*traceids*)
Check if at least one of the given trace ids are in the traceids

toDict()
Return dict representation of the event

class performance.driver.core.events.**FlagUpdateEvent**(*name*, *value*, **args*, ***kwargs*)
A flag has changed for this run

class performance.driver.core.events.**InterruptEvent**(*traceid=None*, *ts=None*)
An interrupt event is dispatched when a critical exception has occurred or when the user has instructed to interupt the tests via a keystroke

class performance.driver.core.events.**LogLineEvent**(*line*, *source*, *kind=None*, **args*, ***kwargs*)
A log line from an observer

class performance.driver.core.events.**MetricUpdateEvent**(*name*, *value*, **args*, ***kwargs*)
A metric has changed

class performance.driver.core.events.**ObserverEvent**(*metric*, **args*, ***kwargs*)
A metric change is observed

class performance.driver.core.events.**ObserverValueEvent**(*metric*, *value*, **args*, ***kwargs*)
A metric has changed to a new value

class performance.driver.core.events.**ParameterUpdateEvent**(*newParameters*, *oldParameters*, *changes*, **args*, ***kwargs*)
A parameter change request

class performance.driver.core.events.**RestartEvent**(*traceid=None*, *ts=None*)
A restart event is dispatched in place of StartEvent when more than one test loops has to be executed.

class performance.driver.core.events.**RunTaskCompletedEvent**(*previousEvent*, *exception=None*)
This event is displatched when a task is completed. This is useful if you want to keep track of a lengthy event

class performance.driver.core.events.**RunTaskEvent**(*task*)
This event is dispatched when a policy requires the session to execute a task

class performance.driver.core.events.**StalledEvent**(*traceid=None*, *ts=None*)
An stalled event is dispatched from the session manager when an FSM has stuck to a non-terminal state for longer than expected time.

class performance.driver.core.events.**StartEvent**(*traceid=None*, *ts=None*)
A start event is dispatched when the test configuration is loaded and the environment is ready, in order to start the policies.

class performance.driver.core.events.**TeardownEvent**(*traceid=None*, *ts=None*)
A teardown event is dispatched when all policies are completed and the system is about to be torn down.

class performance.driver.core.events.**TickEvent**(*count*, *delta*, **args*, ***kwargs*)
A clock event is dispatched periodically by the event bus

8.10 Event Filters

class performance.driver.core.eventfilters.**EventFilter**(*expression*)
Various trackers in *DC/OS Performance Test Driver* are operating purely on events. Therefore it's some times needed to use a more elaborate selector in order to filter the correct events.

The following filter expression is currently supported and are closely modeled around the CSS syntax:

```
EventName[attrib1=value,attrib2=value,...]:selector1:selector2:...
```

Where:

- _Event Name_ is the name of the event or `*` if you want to match any event.
- _Attributes_ is a comma-separated list of `<attrib> <operator> <value>` values. For example: `method==post`. The following table summarises the different operators you can use for the attributes.

Operator	Description
= or ==	Equal (case sensitive for strings)
!=	Not equal
>, >=	Grater than / Grater than or equal
<, <=	Less than / Less than or equal
~=	Partial regular expression match
~==	Exact regular expression match
<~	Value in list or key in dictionary (like `in`)

- _Selector_ specifies which event out of many similar to chose. Valid selectors are:

Selector	Description
`:first`	Match the first event in the tracking session
`:last`	Match the last event in the tracking session
`:nth(n)` `:nth(n,grp)`	Match the n-th event in the tracking session. If a `grp` parameter is specified, the counter will be groupped with the given indicator.
`:single`	Match a single event, globally. After the first match all other usages accept by default.
`:after(Xs)`	Trigger after X seconds after the last event
`:notrace`	Ignore the trace ID matching and accept any event, even if they do not belong in the trace session.

For example, to match every `HTTPRequestEvent`:

```
HTTPRequestEvent
```

Or, to match every POST `HTTPRequestEvent`:

```
HTTPRequestEvent[method=post]
```

Or, to match the last `HTTPResponseEndEvent`

```
HTTPResponseEndEvent:last
```

Or, to match the `HTTPRequestStartEvent` that contains the string “foo”:

```
HTTPResponseEndEvent[body~=foo]
```

Or match any first event:

```
*:first
```

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Index