Watcher Documentation

Release 14.1.1.dev3

OpenStack Foundation

Jan 15, 2026

CONTENTS

1 System Architecture **3**
- 1.1 Overview . . . 3
- 1.2 Components . . . 3
  - 1.2.1 AMQP Bus . . . 3
  - 1.2.2 Datasource . . . 4
  - 1.2.3 Watcher API . . . 4
  - 1.2.4 Watcher Applier . . . 4
  - 1.2.5 Watcher CLI . . . 5
  - 1.2.6 Watcher Dashboard . . . 5
  - 1.2.7 Watcher Database . . . 5
  - 1.2.8 Watcher Decision Engine . . . 5
- 1.3 Data model . . . 6
- 1.4 Sequence diagrams . . . 9
  - 1.4.1 Create a new Audit Template . . . 9
  - 1.4.2 Create and launch a new Audit . . . 9
  - 1.4.3 Launch Action Plan . . . 13
- 1.5 State Machine diagrams . . . 15
  - 1.5.1 Audit State Machine . . . 15
  - 1.5.2 Action Plan State Machine . . . 16

2 Contribution Guide **17**
- 2.1 So You Want to Contribute . . . 17
  - 2.1.1 Communication . . . 17
  - 2.1.2 Contacting the Core Team . . . 17
  - 2.1.3 New Feature Planning . . . 17
  - 2.1.4 Task Tracking . . . 18
  - 2.1.5 Reporting a Bug . . . 18
  - 2.1.6 Getting Your Patch Merged . . . 18
  - Project Team Lead Duties . . . 18
- 2.2 Set up a development environment manually . . . 18
  - 2.2.1 Prerequisites . . . 18
  - 2.2.2 Getting the latest code . . . 18
  - 2.2.3 Installing dependencies . . . 19
  - PyPi Packages and VirtualEnv . . . 19
  - 2.2.4 Verifying Watcher is set up . . . 20
  - 2.2.5 Run Watcher tests . . . 20
  - 2.2.6 Build the Watcher documentation . . . 20
  - 2.2.7 Configure the Watcher services . . . 20
  - 2.2.8 Create Watcher SQL database . . . 21

2.2.9 Running Watcher services . . . 21
2.2.10 Interact with Watcher . . . 21
2.2.11 Exercising the Watcher Services locally . . . 21
2.3 Set up a development environment via DevStack . . . 22
2.3.1 Quick Devstack Instructions with Datasources . . . 22
Gnocchi . . . 22
2.3.2 Detailed DevStack Instructions . . . 22
2.3.3 Multi-Node DevStack Environment . . . 24
Setting up SSH keys between compute nodes to enable live migration . . . 24
Configuring NFS Server (ADVANCED) . . . 25
Configuring NFS on Compute Node (ADVANCED) . . . 25
Configuring libvirt to listen on tcp (ADVANCED) . . . 25
VNC server configuration . . . 26
Environment final checkup . . . 26
2.4 Developer Testing . . . 26
2.4.1 Unit tests . . . 26
2.4.2 Tempest tests . . . 27
2.5 Rally job . . . 27
2.5.1 Structure . . . 27
2.5.2 Useful links . . . 27

3 Install Guide **29**
3.1 Infrastructure Optimization service overview . . . 29
3.2 Install and configure . . . 29
3.2.1 Install and configure for Red Hat Enterprise Linux and CentOS . . . 29
Prerequisites . . . 30
Install and configure components . . . 32
Finalize installation . . . 33
3.2.2 Install and configure for Ubuntu . . . 34
Prerequisites . . . 34
Install and configure components . . . 36
Finalize installation . . . 38
3.3 Verify operation . . . 38
3.4 Next steps . . . 41

4 Administrator Guide **43**
4.1 Installing API behind mod_wsgi . . . 43
4.2 Guru Meditation Reports . . . 44
4.2.1 Generating a GMR . . . 44
4.2.2 Structure of a GMR . . . 44
4.3 Policies . . . 44
4.3.1 Constructing a Policy Configuration File . . . 45
4.4 Strategies . . . 47
4.4.1 Actuator . . . 47
Synopsis . . . 47
Requirements . . . 47
Configuration . . . 47
Efficacy Indicator . . . 48
Algorithm . . . 48
How to use it ? . . . 48
External Links . . . 48
4.4.2 Basic Offline Server Consolidation . . . 48

Synopsis . . . 48
Requirements . . . 49
Configuration . . . 51
Efficacy Indicator . . . 51
How to use it ? . . . 51
External Links . . . 51
4.4.3 Host Maintenance Strategy . . . 51
Synopsis . . . 51
Requirements . . . 52
Configuration . . . 54
Efficacy Indicator . . . 54
Algorithm . . . 54
How to use it ? . . . 54
External Links . . . 54
4.4.4 Node Resource Consolidation Strategy . . . 54
Synopsis . . . 54
Requirements . . . 55
Configuration . . . 57
Efficacy Indicator . . . 57
Algorithm . . . 57
How to use it ? . . . 57
External Links . . . 57
4.4.5 Noisy neighbor . . . 57
Synopsis . . . 57
Requirements . . . 58
Configuration . . . 60
Efficacy Indicator . . . 60
Algorithm . . . 60
How to use it ? . . . 60
External Links . . . 60
4.4.6 Outlet Temperature Based Strategy . . . 60
Synopsis . . . 60
Requirements . . . 60
Configuration . . . 63
Efficacy Indicator . . . 63
Algorithm . . . 63
How to use it ? . . . 63
External Links . . . 63
4.4.7 Saving Energy Strategy . . . 63
Synopsis . . . 63
Requirements . . . 64
Configuration . . . 65
Efficacy Indicator . . . 65
Algorithm . . . 65
How to use it ? . . . 66
External Links . . . 66
4.4.8 Storage capacity balance . . . 66
Synopsis . . . 66
Requirements . . . 66
Configuration . . . 68
Efficacy Indicator . . . 68

Algorithm . . . 68
How to use it ? . . . 68
External Links . . . 68
4.4.9 Uniform Airflow Migration Strategy . . . 68
Synopsis . . . 68
Requirements . . . 69
Configuration . . . 71
Efficacy Indicator . . . 71
Algorithm . . . 71
How to use it ? . . . 71
External Links . . . 71
4.4.10 VM Workload Consolidation Strategy . . . 71
Synopsis . . . 71
Requirements . . . 72
Configuration . . . 75
Efficacy Indicator . . . 75
Algorithm . . . 75
How to use it ? . . . 75
External Links . . . 75
4.4.11 Watcher Overload standard deviation algorithm . . . 75
Synopsis . . . 75
Requirements . . . 76
Configuration . . . 78
Efficacy Indicator . . . 78
Algorithm . . . 79
How to use it ? . . . 79
External Links . . . 79
4.4.12 Workload Balance Migration Strategy . . . 79
Synopsis . . . 79
Requirements . . . 80
Configuration . . . 82
Efficacy Indicator . . . 82
Algorithm . . . 82
How to use it ? . . . 82
External Links . . . 82
4.4.13 Zone migration . . . 82
Synopsis . . . 82
Requirements . . . 83
Configuration . . . 85
Efficacy Indicator . . . 86
Algorithm . . . 86
How to use it ? . . . 86
External Links . . . 87
4.5 Datasources . . . 87
4.5.1 Grafana datasource . . . 87
Synopsis . . . 87
Requirements . . . 87
Configuration . . . 87
Example configuration . . . 91
External Links . . . 93
4.5.2 Prometheus datasource . . . 93

Synopsis . . . 93
Requirements . . . 94
Limitations . . . 94
Configuration . . . 95
4.6 Notifications in Watcher . . . 96
4.7 Concurrency . . . 96
4.7.1 Introduction . . . 96
4.7.2 Threadpool . . . 97
4.7.3 Decision engine concurrency . . . 97
AuditEndpoint . . . 97
DecisionEngineThreadPool . . . 97
4.7.4 Applier concurrency . . . 99

5 User Guide **101**
5.1 Ways to install Watcher . . . 101
5.1.1 Prerequisites . . . 101
5.1.2 Installing from Source . . . 101
5.1.3 Installing from packages: PyPI . . . 102
5.1.4 Installing from packages: Debian (experimental) . . . 102
5.2 Watcher User Guide . . . 103
5.2.1 Getting started with Watcher . . . 104
5.2.2 Watcher CLI Command . . . 105
5.2.3 Running an audit of the cluster . . . 105
5.3 Audit using Aodh alarm . . . 106
5.3.1 Step 1: Create an audit with EVENT type . . . 107
5.3.2 Step 2: Create Aodh Alarm . . . 107
5.3.3 Step 3: Trigger the alarm . . . 109
5.3.4 Step 4: Verify the audit . . . 110

6 Configuration Guide **111**
6.1 Configuring Watcher . . . 111
6.1.1 Service overview . . . 111
6.1.2 Install and configure prerequisites . . . 112
Configure the Identity service for the Watcher service . . . 112
Set up the database for Watcher . . . 113
6.1.3 Configure the Watcher service . . . 113
6.1.4 Configure Nova compute . . . 117
6.1.5 Configure Measurements . . . 117
6.1.6 Configure Nova Notifications . . . 117
6.1.7 Configure Cinder Notifications . . . 118
6.1.8 Workers . . . 118
6.2 watcher.conf . . . 118
6.2.1 DEFAULT . . . 118
6.2.2 api . . . 131
6.2.3 cache . . . 133
6.2.4 cinder_client . . . 141
6.2.5 collector . . . 141
6.2.6 database . . . 142
6.2.7 glance_client . . . 146
6.2.8 gnocchi_client . . . 147
6.2.9 grafana_client . . . 147
6.2.10 grafana_translators . . . 149

6.2.11 ironic_client . . . 150
6.2.12 keystone_authtoken . . . 150
6.2.13 keystone_client . . . 157
6.2.14 maas_client . . . 157
6.2.15 monasca_client . . . 158
6.2.16 neutron_client . . . 158
6.2.17 nova_client . . . 159
6.2.18 oslo_concurrency . . . 159
6.2.19 oslo_messaging_kafka . . . 160
6.2.20 oslo_messaging_notifications . . . 162
6.2.21 oslo_messaging_rabbit . . . 163
6.2.22 oslo_policy . . . 171
6.2.23 oslo_reports . . . 174
6.2.24 placement_client . . . 174
6.2.25 prometheus_client . . . 175
6.2.26 watcher_applier . . . 176
6.2.27 watcher_clients_auth . . . 177
6.2.28 watcher_cluster_data_model_collectors.baremetal . . . 179
6.2.29 watcher_cluster_data_model_collectors.compute . . . 179
6.2.30 watcher_cluster_data_model_collectors.storage . . . 179
6.2.31 watcher_datasources . . . 179
6.2.32 watcher_decision_engine . . . 180
6.2.33 watcher_planner . . . 182
6.2.34 watcher_planners.weight . . . 183
6.2.35 watcher_planners.workload_stabilization . . . 183
6.2.36 watcher_strategies.basic . . . 183
6.2.37 watcher_strategies.node_resource_consolidation . . . 184
6.2.38 watcher_strategies.noisy_neighbor . . . 184
6.2.39 watcher_strategies.outlet_temperature . . . 184
6.2.40 watcher_strategies.storage_capacity_balance . . . 184
6.2.41 watcher_strategies.uniform_airflow . . . 185
6.2.42 watcher_strategies.vm_workload_consolidation . . . 185
6.2.43 watcher_strategies.workload_balance . . . 185
6.2.44 watcher_strategies.workload_stabilization . . . 185
6.2.45 watcher_workflow_engines.taskflow . . . 186

7 Plugin Guide **187**
7.1 Create a third-party plugin for Watcher . . . 187
7.1.1 Pre-requisites . . . 187
7.1.2 Third party project scaffolding . . . 187
7.1.3 Implementing a plugin for Watcher . . . 188
7.2 Build a new action . . . 188
7.2.1 Creating a new plugin . . . 188
Input validation . . . 189
7.2.2 Define configuration parameters . . . 190
7.2.3 Abstract Plugin Class . . . 190
7.2.4 Register a new entry point . . . 192
7.2.5 Using action plugins . . . 192
7.2.6 Scheduling of an action plugin . . . 192
7.2.7 Test your new action . . . 192
7.3 Build a new cluster data model collector . . . 192

7.3.1 Creating a new plugin . . . 193
7.3.2 Define a custom model . . . 193
7.3.3 Define configuration parameters . . . 194
7.3.4 Abstract Plugin Class . . . 195
7.3.5 Register a new entry point . . . 196
7.3.6 Add new notification endpoints . . . 196
7.3.7 Using cluster data model collector plugins . . . 197
7.4 Build a new goal . . . 197
7.4.1 Pre-requisites . . . 197
7.4.2 Create a new plugin . . . 198
7.4.3 Abstract Plugin Class . . . 199
7.4.4 Add a new entry point . . . 199
7.4.5 Implement a customized efficacy specification . . . 199
What is it for? . . . 199
Implementation . . . 200
7.5 Build a new planner . . . 201
7.5.1 Creating a new plugin . . . 201
7.5.2 Define configuration parameters . . . 202
7.5.3 Abstract Plugin Class . . . 202
7.5.4 Register a new entry point . . . 203
7.5.5 Using planner plugins . . . 203
7.6 Build a new scoring engine . . . 203
7.6.1 Pre-requisites . . . 204
7.6.2 Create a new scoring engine plugin . . . 204
7.6.3 (Optional) Create a new scoring engine container plugin . . . 205
7.6.4 Abstract Plugin Class . . . 205
7.6.5 Abstract Plugin Container Class . . . 206
7.6.6 Add a new entry point . . . 207
7.6.7 Using scoring engine plugins . . . 208
7.7 Build a new optimization strategy . . . 208
7.7.1 Pre-requisites . . . 208
7.7.2 Create a new strategy plugin . . . 208
7.7.3 Strategy efficacy . . . 209
7.7.4 Define Strategy Parameters . . . 209
7.7.5 Abstract Plugin Class . . . 210
7.7.6 Add a new entry point . . . 212
7.7.7 Using strategy plugins . . . 213
Querying metrics . . . 213
Read usage metrics using the Watcher Datasource Helper . . . 213
7.8 Available Plugins . . . 214
7.8.1 Goals . . . 214
airflow_optimization . . . 214
cluster_maintaining . . . 214
dummy . . . 214
hardware_maintenance . . . 214
noisy_neighbor . . . 214
saving_energy . . . 214
server_consolidation . . . 214
thermal_optimization . . . 215
unclassified . . . 215
workload_balancing . . . 215

7.8.2 Scoring Engines . . . 215
dummy_scorer . . . 215
7.8.3 Scoring Engine Containers . . . 216
dummy_scoring_container . . . 216
7.8.4 Strategies . . . 216
actuator . . . 216
basic . . . 216
dummy . . . 216
dummy_with_resize . . . 217
dummy_with_scorer . . . 217
host_maintenance . . . 217
node_resource_consolidation . . . 218
noisy_neighbor . . . 218
outlet_temperature . . . 218
saving_energy . . . 219
storage_capacity_balance . . . 220
uniform_airflow . . . 220
vm_workload_consolidation . . . 220
workload_balance . . . 221
workload_stabilization . . . 221
zone_migration . . . 222
7.8.5 Actions . . . 222
change_node_power_state . . . 222
change_nova_service_state . . . 222
migrate . . . 222
nop . . . 223
resize . . . 223
sleep . . . 223
volume_migrate . . . 224
7.8.6 Workflow Engines . . . 224
taskflow . . . 224
7.8.7 Planners . . . 224
node_resource_consolidation . . . 224
weight . . . 224
workload_stabilization . . . 225
7.8.8 Cluster Data Model Collectors . . . 225
baremetal . . . 225
compute . . . 225
storage . . . 225

8 Watcher Manual Pages **227**
8.1 watcher-api . . . 227
8.1.1 Service for the Watcher API . . . 227
SYNOPSIS . . . 227
DESCRIPTION . . . 227
OPTIONS . . . 227
FILES . . . 228
BUGS . . . 228
8.2 watcher-applier . . . 229
8.2.1 Service for the Watcher Applier . . . 229
SYNOPSIS . . . 229

DESCRIPTION . . . 229
OPTIONS . . . 229
FILES . . . 230
BUGS . . . 230
8.3 watcher-db-manage . . . 230
8.3.1 Options . . . 230
8.3.2 Usage . . . 231
8.3.3 Command Options . . . 231
create_schema . . . 231
downgrade . . . 232
revision . . . 232
stamp . . . 232
upgrade . . . 232
version . . . 233
purge . . . 233
8.4 watcher-decision-engine . . . 234
8.4.1 Service for the Watcher Decision Engine . . . 234
SYNOPSIS . . . 234
DESCRIPTION . . . 234
OPTIONS . . . 234
FILES . . . 235
BUGS . . . 235
8.5 watcher-status . . . 235
8.5.1 CLI interface for Watcher status commands . . . 235
Synopsis . . . 235
Description . . . 236
Options . . . 236

9 REST API Version History 237
9.1 1.0 (Initial version) . . . 237
9.2 1.1 . . . 237
9.3 1.2 . . . 237
9.4 1.3 . . . 237
9.5 1.4 . . . 237

10 Glossary 239
10.1 Action . . . 239
10.2 Action Plan . . . 239
10.3 Administrator . . . 240
10.4 Audit . . . 240
10.5 Audit Scope . . . 240
10.6 Audit Template . . . 241
10.7 Availability Zone . . . 241
10.8 Cluster . . . 241
10.9 Cluster Data Model (CDM) . . . 241
10.10 Controller Node . . . 242
10.11 Compute node . . . 242
10.12 Customer . . . 243
10.13 Goal . . . 243
10.14 Host Aggregate . . . 243
10.15 Instance . . . 243
10.16 Managed resource . . . 243

10.17 Managed resource type . . . 243
10.18 Efficacy Indicator . . . 244
10.19 Efficacy Specification . . . 244
10.20 Optimization Efficacy . . . 244
10.21 Project . . . 245
10.22 Scoring Engine . . . 245
10.23 SLA . . . 245
10.24 SLA violation . . . 245
10.25 SLO . . . 245
10.26 Solution . . . 245
10.27 Strategy . . . 246
10.28 Watcher Applier . . . 246
10.29 Watcher Database . . . 246
10.30 Watcher Decision Engine . . . 247
10.31 Watcher Planner . . . 247

OpenStack Watcher provides a flexible and scalable resource optimization service for multi-tenant OpenStack-based clouds. Watcher provides a complete optimization loopincluding everything from a metrics receiver, complex event processor and profiler, optimization processor and an action plan applier. This provides a robust framework to realize a wide range of cloud optimization goals, including the reduction of data center operating costs, increased system performance via intelligent virtual machine migration, increased energy efficiency and more!

Watcher project consists of several source code repositories:

- watcher - is the main repository. It contains code for Watcher API server, Watcher Decision Engine and Watcher Applier.
- python-watcherclient - Client library and CLI client for Watcher.
- watcher-dashboard - Watcher Horizon plugin.

The documentation provided here is continually kept up-to-date based on the latest code, and may not represent the state of the project at any specific prior release.

CHAPTER

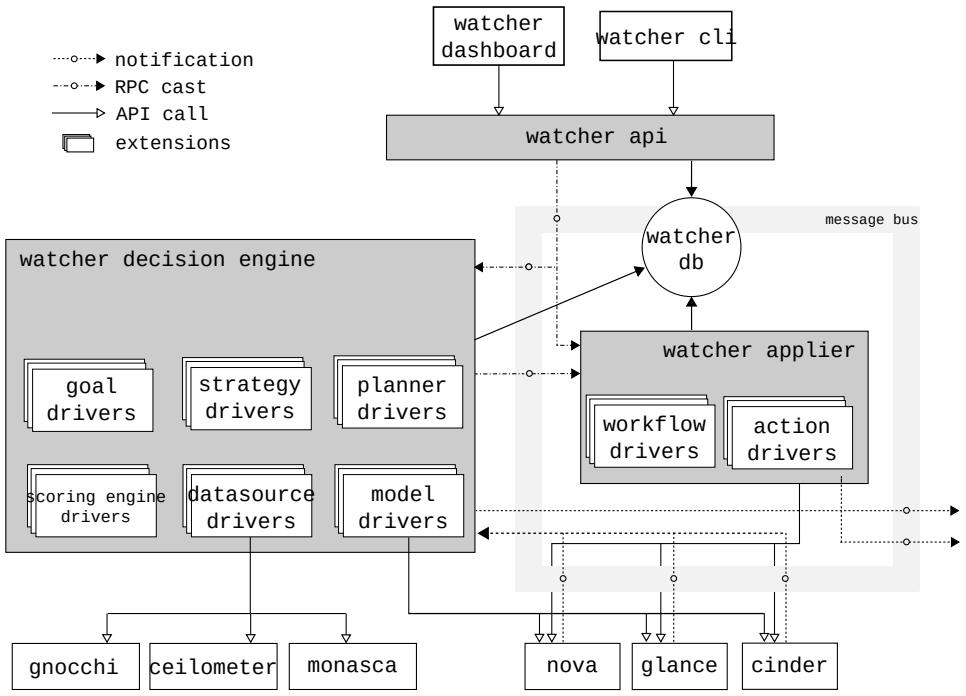
ONE

SYSTEM ARCHITECTURE

This page presents the current technical Architecture of the Watcher system.

1.1 Overview

Below you will find a diagram, showing the main components of Watcher:



1.2 Components

1.2.1 AMQP Bus

The AMQP message bus handles internal asynchronous communications between the different Watcher components.

1.2.2 Datasource

This component stores the metrics related to the cluster.

It can potentially rely on any appropriate storage system (InfluxDB, OpenTSDB, MongoDB,) but will probably be more performant when using Time Series Databases which are optimized for handling time series data, which are arrays of numbers indexed by time (a datetime or a datetime range).

1.2.3 Watcher API

This component implements the REST API provided by the Watcher system to the external world.

It enables the *Administrator* of a *Cluster* to control and monitor the Watcher system via any interaction mechanism connected to this API:

- *CLI*
- Horizon plugin
- Python SDK

You can also read the detailed description of Watcher API.

1.2.4 Watcher Applier

This component is in charge of executing the *Action Plan* built by the *Watcher Decision Engine*. Taskflow is the default workflow engine for Watcher.

It connects to the *message bus* and launches the *Action Plan* whenever a triggering message is received on a dedicated AMQP queue.

The triggering message contains the Action Plan UUID.

It then gets the detailed information about the *Action Plan* from the *Watcher Database* which contains the list of *Actions* to launch.

It then loops on each *Action*, gets the associated class and calls the execute() method of this class. Most of the time, this method will first request a token to the Keystone API and if it is allowed, sends a request to the REST API of the OpenStack service which handles this kind of *atomic Action*.

Note that as soon as *Watcher Applier* starts handling a given *Action* from the list, a notification message is sent on the *message bus* indicating that the state of the action has changed to **ONGOING**.

If the *Action* is successful, the *Watcher Applier* sends a notification message on *the bus* informing the other components of this.

If the *Action* fails, the *Watcher Applier* tries to rollback to the previous state of the *Managed resource* (i.e. before the command was sent to the underlying OpenStack service).

In Stein, added a new config option action_execution_rule which is a dict type. Its key field is strategy name and the value is ALWAYS or ANY. ALWAYS means the callback function returns True as usual. ANY means the return depends on the result of previous action execution. The callback returns True if previous action gets failed, and the engine continues to run the next action. If previous action executes success, the callback returns False then the next action will be ignored. For strategies that arent in action_execution_rule, the callback always returns True. Please add the next section in the watcher.conf file if your strategy needs this feature.

```
[watcher_workflow_engines.taskflow]
action_execution_rule = {'your strategy name': 'ANY'}
```

1.2.5 Watcher CLI

The watcher command-line interface (CLI) can be used to interact with the Watcher system in order to control it or to know its current status.

Please, read the detailed documentation about Watcher CLI.

1.2.6 Watcher Dashboard

The Watcher Dashboard can be used to interact with the Watcher system through Horizon in order to control it or to know its current status.

Please, read the detailed documentation about Watcher Dashboard.

1.2.7 Watcher Database

This database stores all the Watcher domain objects which can be requested by the *Watcher API* or the *Watcher CLI*:

- *Goals*
- *Strategies*
- *Audit templates*
- *Audits*
- *Action plans*
- *Efficacy indicators* via the Action Plan API.
- *Actions*

The Watcher domain being here *optimization of some resources provided by an OpenStack system*.

1.2.8 Watcher Decision Engine

This component is responsible for computing a set of potential optimization *Actions* in order to fulfill the *Goal* of an *Audit*.

It first reads the parameters of the *Audit* to know the *Goal* to achieve.

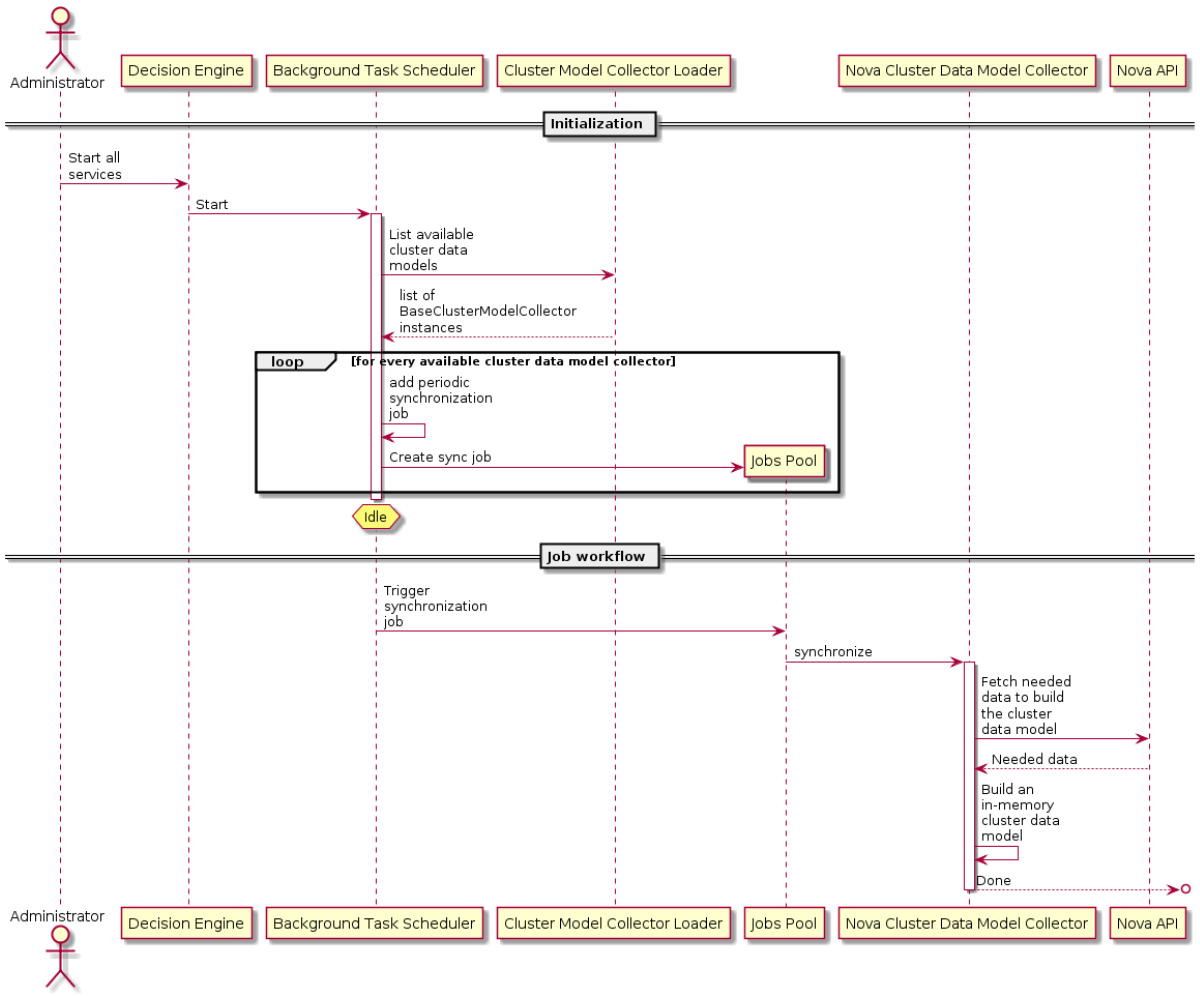
Unless specified, it then selects the most appropriate *strategy* from the list of available strategies achieving this goal.

The *Strategy* is then dynamically loaded (via stevedore). The *Watcher Decision Engine* executes the strategy.

In order to compute the potential *Solution* for the Audit, the *Strategy* relies on different sets of data:

- *Cluster data models* that are periodically synchronized through pluggable cluster data model collectors. These models contain the current state of various *Managed resources* (e.g., the data stored in the Nova database). These models gives a strategy the ability to reason on the current state of a given *cluster*.
- The data stored in the *Cluster Datasource* which provides information about the past of the *Cluster*.

Here below is a sequence diagram showing how the Decision Engine builds and maintains the *cluster data models* that are used by the strategies.

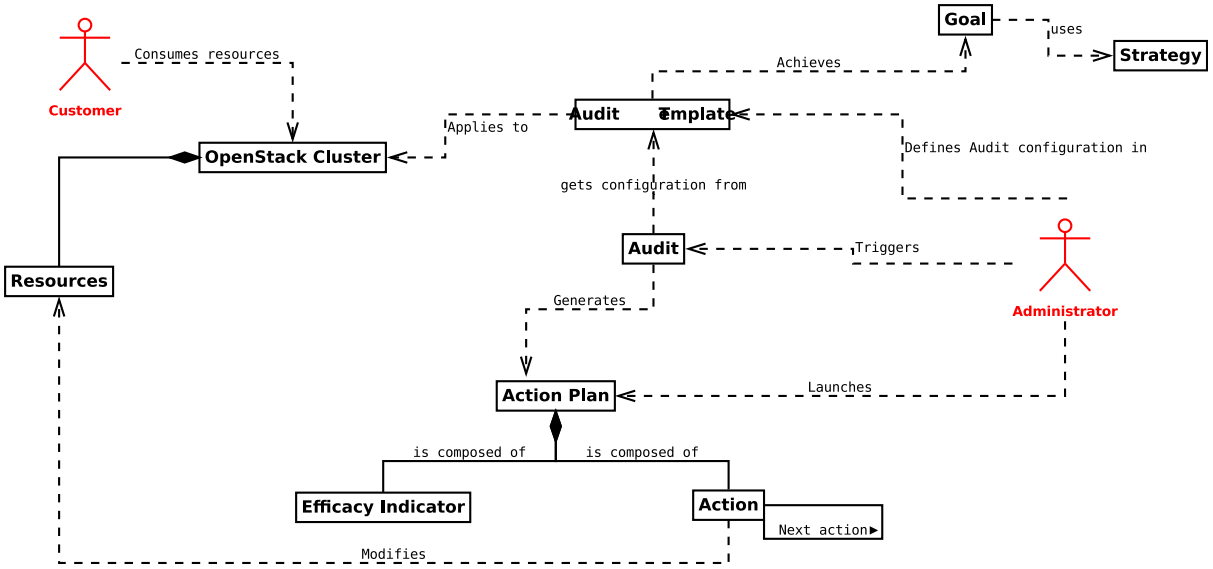


The execution of a strategy then yields a solution composed of a set of *Actions* as well as a set of *efficacy indicators*.

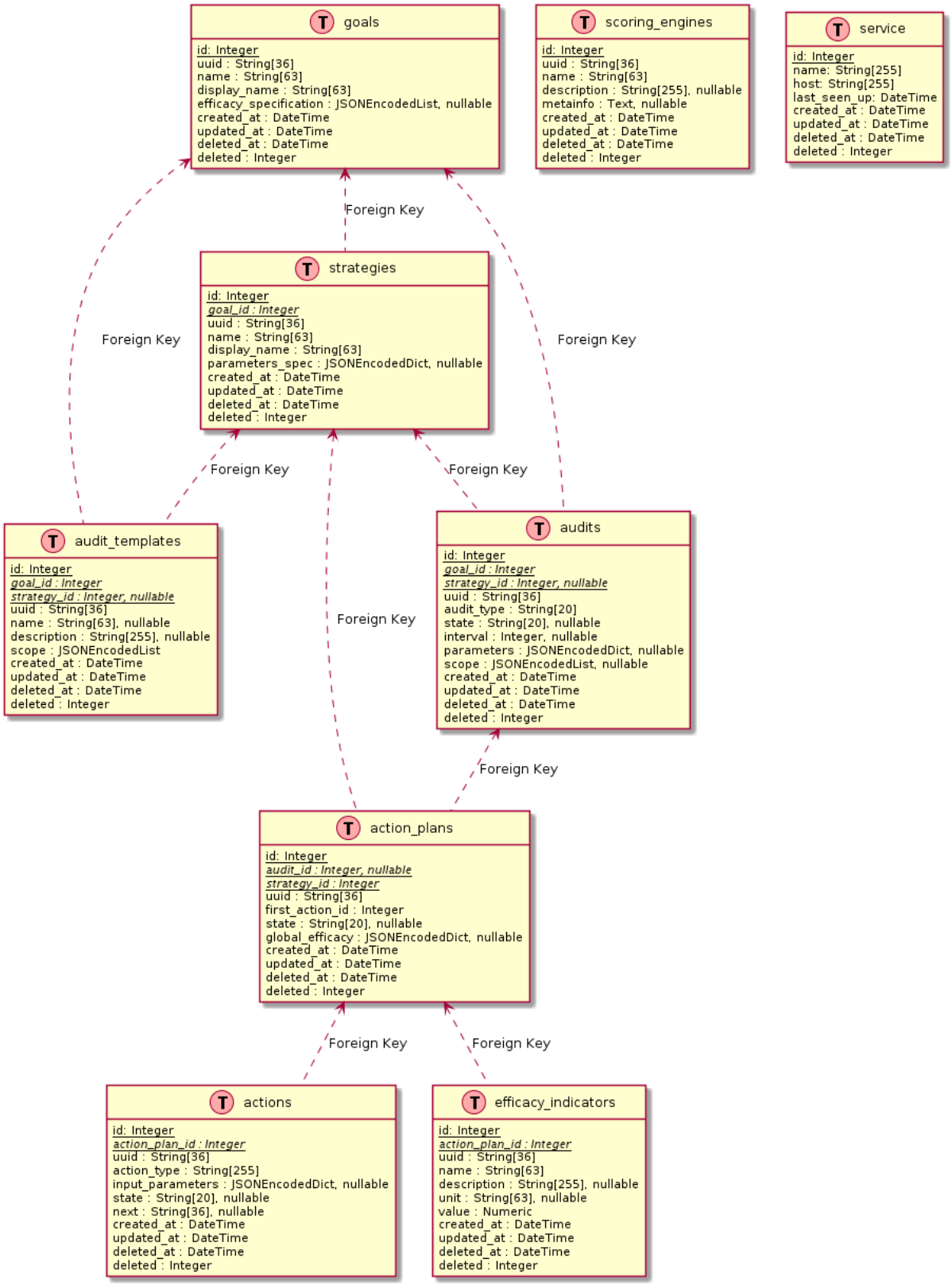
These *Actions* are scheduled in time by the *Watcher Planner* (i.e., it generates an *Action Plan*).

1.3 Data model

The following diagram shows the data model of Watcher, especially the functional dependency of objects from the actors (Admin, Customer) point of view (Goals, Audits, Action Plans,):



Here below is a diagram representing the main objects in Watcher from a database perspective:


goals
id: Integer
uuid : String[36]
name : String[63]
display_name : String[63]
efficacy_specification : JSONEncodedList, nullable
created_at : DateTime
updated_at : DateTime
deleted_at : DateTime
deleted : Integer
scoring_engines
id: Integer
uuid : String[36]
name : String[63]
description : String[255], nullable
metainfo : Text, nullable
created_at : DateTime
updated_at : DateTime
deleted_at : DateTime
deleted : Integer
service
id: Integer
name: String[255]
host: String[255]
last_seen_up: DateTime
created_at : DateTime
updated_at : DateTime
deleted_at : DateTime
deleted : Integer
Foreign Key
strategies
id: Integer
goal_id : Integer
uuid : String[36]
name : String[63]
display_name : String[63]
parameters_spec : JSONEncodedDict, nullable
created_at : DateTime
updated_at : DateTime
deleted_at : DateTime
deleted : Integer
Foreign Key
Foreign Key
Foreign Key
Foreign Key
audit_templates
id: Integer
goal_id : Integer
strategy_id : Integer, nullable
uuid : String[36]
name : String[63], nullable
description : String[255], nullable
scope : JSONEncodedList
created_at : DateTime
updated_at : DateTime
deleted_at : DateTime
deleted : Integer
audits
id: Integer
goal_id : Integer
strategy_id : Integer, nullable
uuid : String[36]
audit_type : String[20]
state : String[20], nullable
interval : Integer, nullable
parameters : JSONEncodedDict, nullable
scope : JSONEncodedList, nullable
created_at : DateTime
updated_at : DateTime
deleted_at : DateTime
deleted : Integer
Foreign Key
Foreign Key
action_plans
id: Integer
audit_id : Integer, nullable
strategy_id : Integer
uuid : String[36]
first_action_id : Integer
state : String[20], nullable
global_efficacy : JSONEncodedDict, nullable
created_at : DateTime
updated_at : DateTime
deleted_at : DateTime
deleted : Integer
Foreign Key
Foreign Key
actions
id: Integer
action_plan_id : Integer
uuid : String[36]
action_type : String[255]
input_parameters : JSONEncodedDict, nullable
state : String[20], nullable
next : String[36], nullable
created_at : DateTime
updated_at : DateTime
deleted_at : DateTime
deleted : Integer
efficacy_indicators
id: Integer
action_plan_id : Integer
uuid : String[36]
name : String[63]
description : String[255], nullable
unit : String[63], nullable
value : Numeric
created_at : DateTime
updated_at : DateTime
deleted_at : DateTime
deleted : Integer

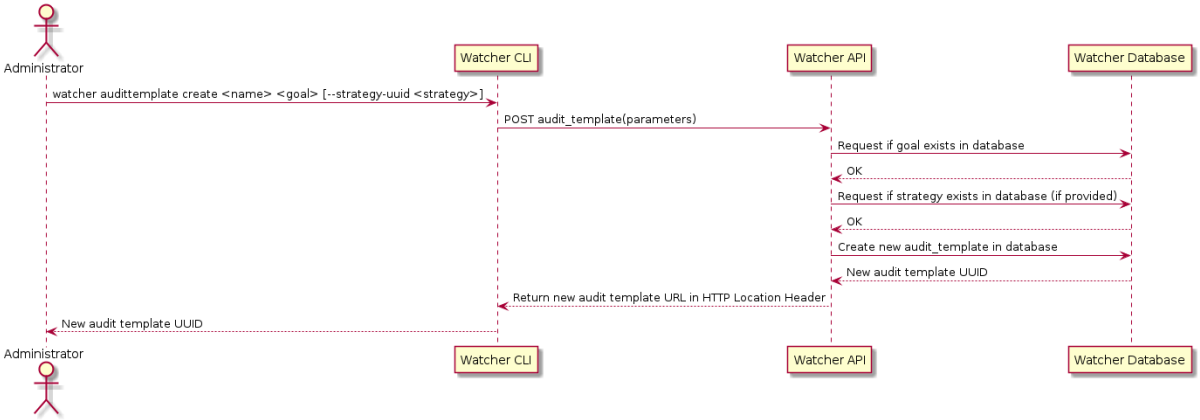
1.4 Sequence diagrams

The following paragraph shows the messages exchanged between the different components of Watcher for the most often used scenarios.

1.4.1 Create a new Audit Template

The *Administrator* first creates an *Audit template* providing at least the following parameters:

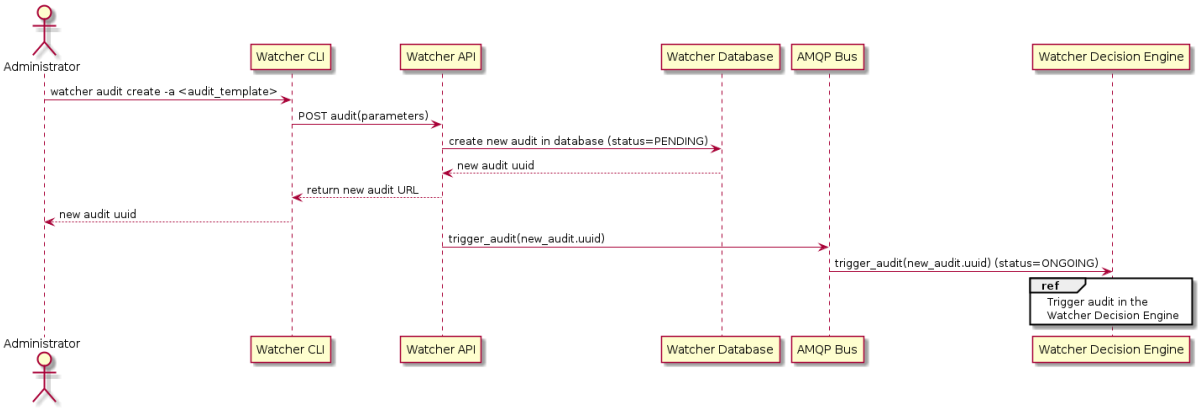
- A name
- A goal to achieve
- An optional strategy



The Watcher API makes sure that both the specified goal (mandatory) and its associated strategy (optional) are registered inside the *Watcher Database* before storing a new audit template in the *Watcher Database*.

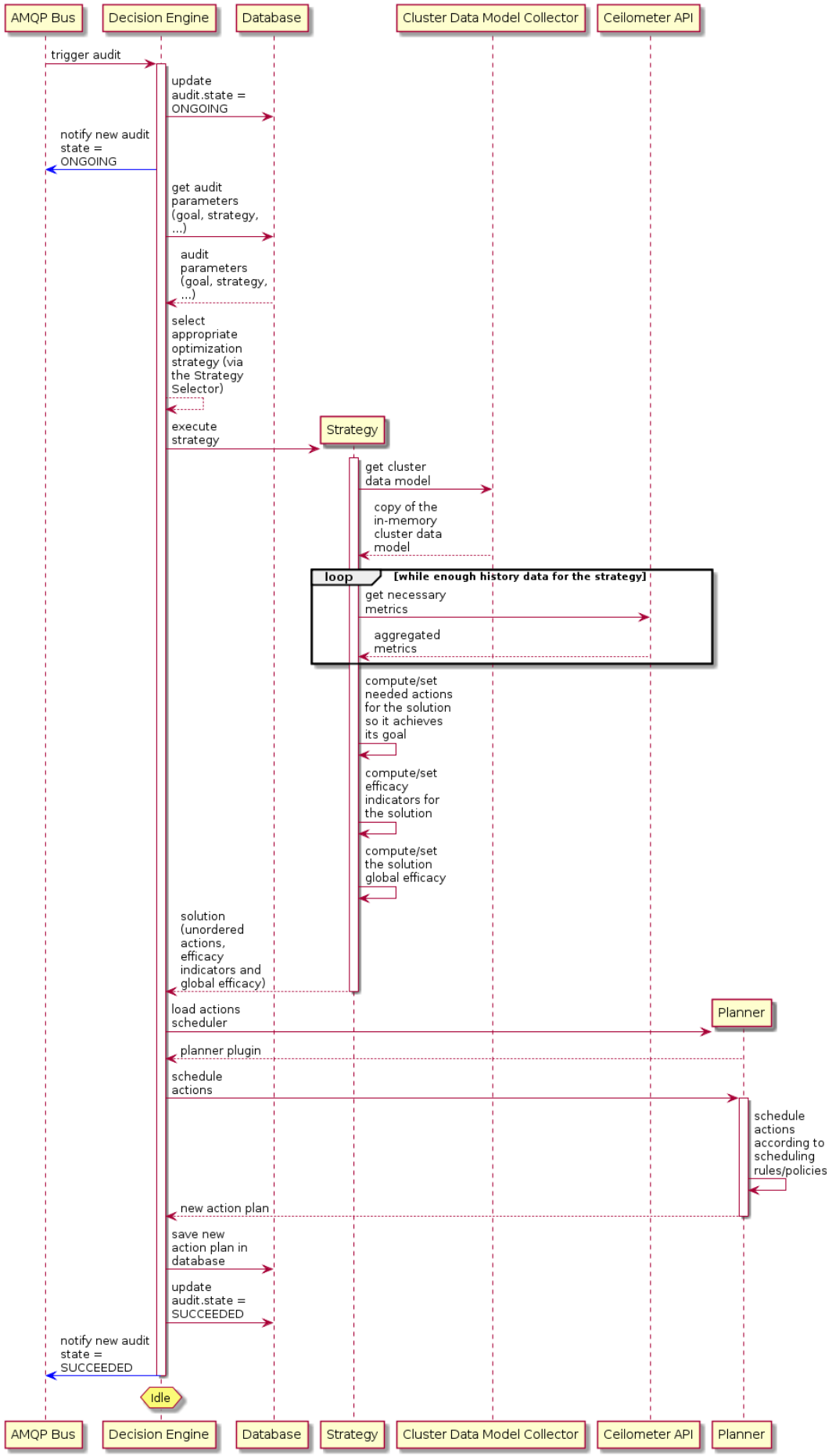
1.4.2 Create and launch a new Audit

The *Administrator* can then launch a new *Audit* by providing at least the unique UUID of the previously created *Audit template*:



The *Administrator* also can specify type of Audit and interval (in case of CONTINUOUS type). There is three types of Audit: ONESHOT, CONTINUOUS and EVENT. ONESHOT Audit is launched once and if it succeeded executed new action plan list will be provided; CONTINUOUS Audit creates action plans with specified interval (in seconds or cron format, cron interval can be used like: `*/5 * * * *`), if action plan has been created, all previous action plans get CANCELLED state; EVENT audit is launched when receiving webhooks API.

A message is sent on the *AMQP bus* which triggers the Audit in the *Watcher Decision Engine*:


AMQP Bus
Decision Engine
Database
Cluster Data Model Collector
Ceilometer API
trigger audit
update audit.state = ONGOING
notify new audit state = ONGOING
get audit parameters (goal, strategy, ...)
audit parameters (goal, strategy, ...)
select appropriate optimization strategy (via the Strategy Selector)
execute strategy
Strategy
get cluster data model
copy of the in-memory cluster data model
loop
[while enough history data for the strategy]
get necessary metrics
aggregated metrics
compute/set needed actions for the solution so it achieves its goal
compute/set efficacy indicators for the solution
compute/set the solution global efficacy
solution (unordered actions, efficacy indicators and global efficacy)
load actions scheduler
Planner
planner plugin
schedule actions
schedule actions according to scheduling rules/policies
new action plan
save new action plan in database
update audit.state = SUCCEEDED
notify new audit state = SUCCEEDED
Idle
AMQP Bus
Decision Engine
Database
Strategy
Cluster Data Model Collector
Ceilometer API
Planner

The *Watcher Decision Engine* reads the Audit parameters from the *Watcher Database*. It instantiates the appropriate *strategy* (using entry points) given both the *goal* and the strategy associated to the parent *audit template* of the *audit*. If no strategy is associated to the audit template, the strategy is dynamically selected by the Decision Engine.

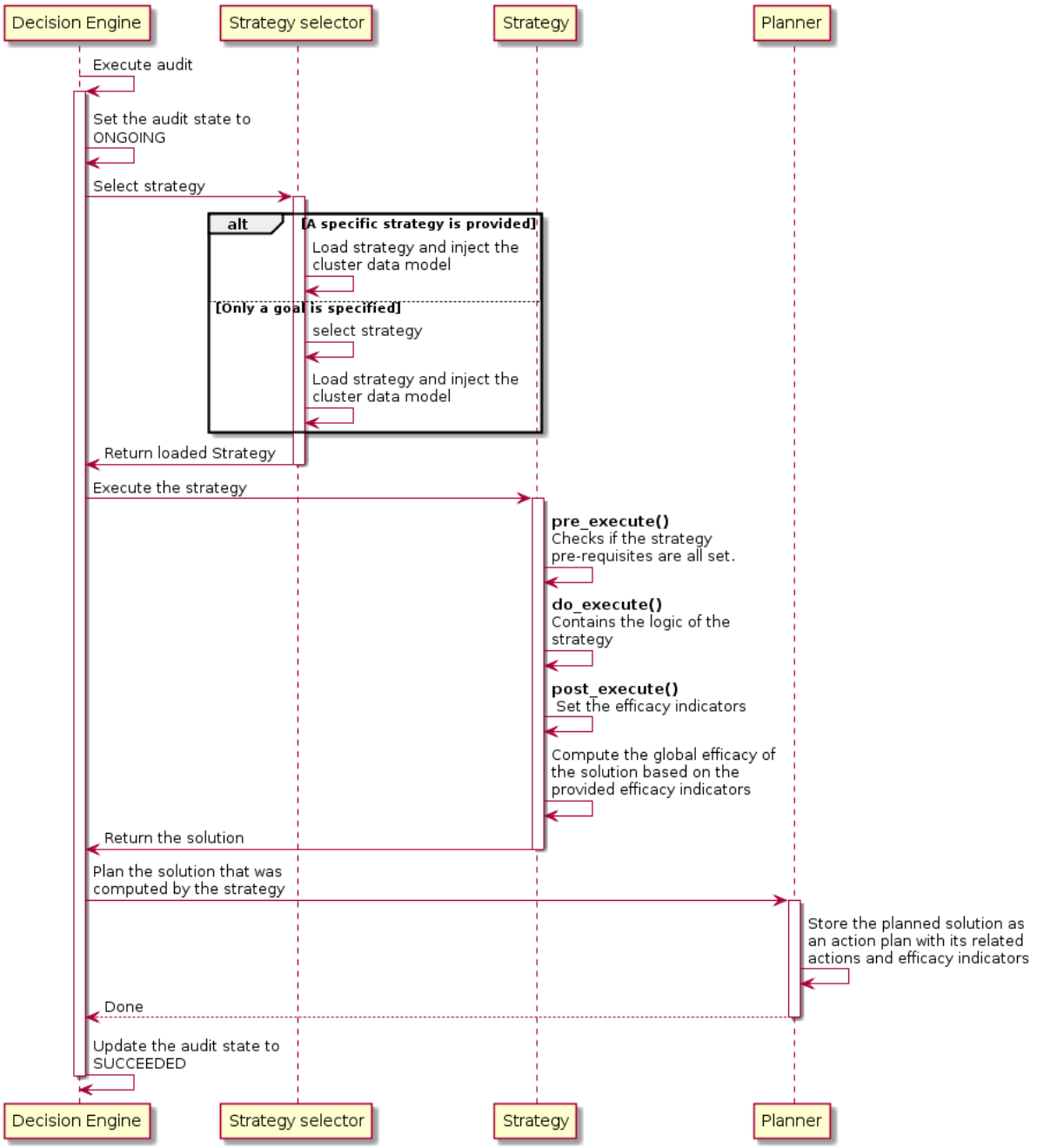
The *Watcher Decision Engine* also builds the *Cluster Data Model*. This data model is needed by the *Strategy* to know the current state and topology of the audited *OpenStack cluster*.

The *Watcher Decision Engine* calls the **execute()** method of the instantiated *Strategy* and provides the data model as an input parameter. This method computes a *Solution* to achieve the goal and returns it to the *Decision Engine*. At this point, actions are not scheduled yet.

The *Watcher Decision Engine* dynamically loads the *Watcher Planner* implementation which is configured in Watcher (via entry points) and calls the **schedule()** method of this class with the solution as an input parameter. This method finds an appropriate scheduling of *Actions* taking into account some scheduling rules (such as priorities between actions). It generates a new *Action Plan* with status **RECOMMENDED** and saves it into the *Watcher Database*. The saved action plan is now a scheduled flow of actions to which a global efficacy is associated alongside a number of *Efficacy Indicators* as specified by the related *goal*.

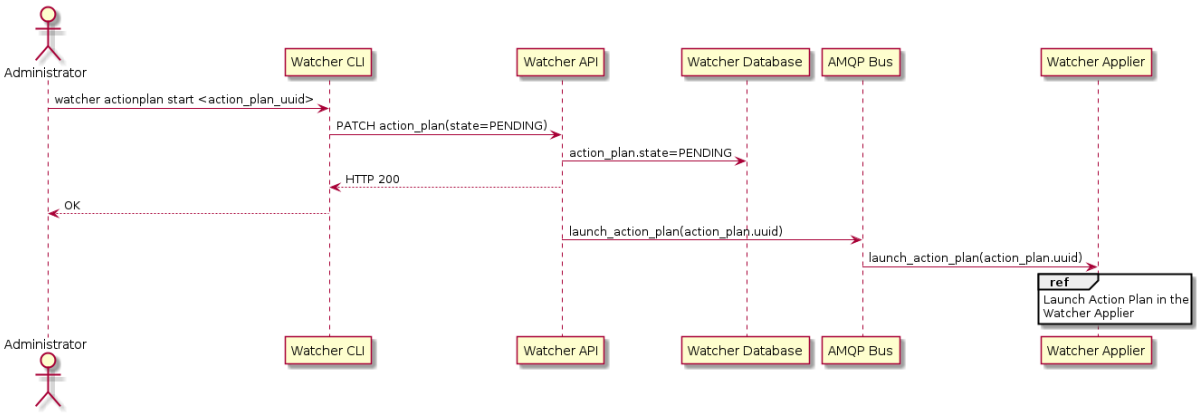
If every step executed successfully, the *Watcher Decision Engine* updates the current status of the Audit to **SUCCEEDED** in the *Watcher Database* and sends a notification on the bus to inform other components that the *Audit* was successful.

This internal workflow the Decision Engine follows to conduct an audit can be seen in the sequence diagram here below:

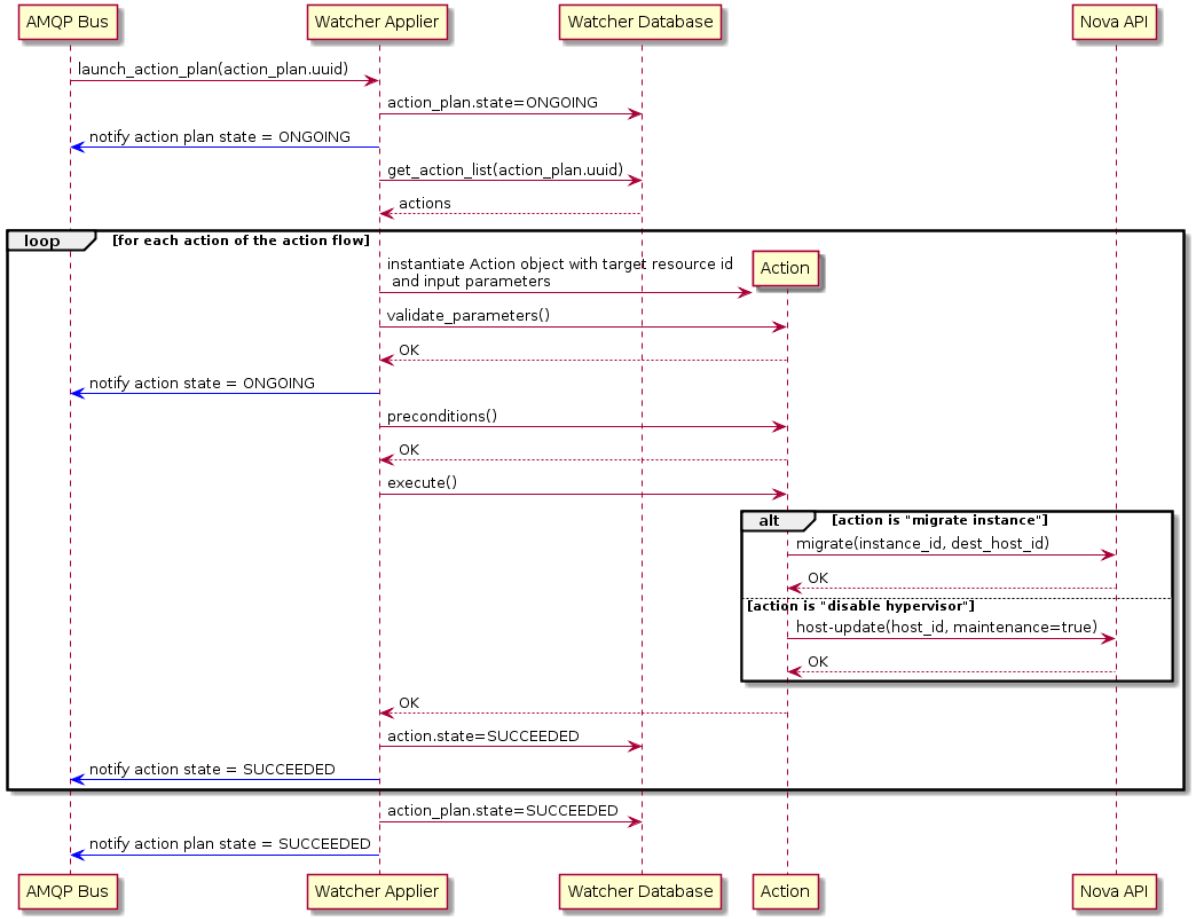


1.4.3 Launch Action Plan

The *Administrator* can then launch the recommended *Action Plan*:



A message is sent on the *AMQP bus* which triggers the *Action Plan* in the *Watcher Applier*:



The *Watcher Applier* will get the description of the flow of *Actions* from the *Watcher Database* and for each *Action* it will instantiate a corresponding *Action* handler python class.

The *Watcher Applier* will then call the following methods of the *Action* handler:

- **validate_parameters**(): this method will make sure that all the provided input parameters are valid:
  - If all parameters are valid, the Watcher Applier moves on to the next step.
  - If it is not, an error is raised and the action is not executed. A notification is sent on the bus informing other components of the failure.
- **preconditions**(): this method will make sure that all conditions are met before executing the action

(for example, it makes sure that an instance still exists before trying to migrate it).

- **execute**(): this method is what triggers real commands on other OpenStack services (such as Nova,) in order to change target resource state. If the action is successfully executed, a notification message is sent on the bus indicating that the new state of the action is **SUCCEEDED**.

If every action of the action flow has been executed successfully, a notification is sent on the bus to indicate that the whole *Action Plan* has **SUCCEEDED**.

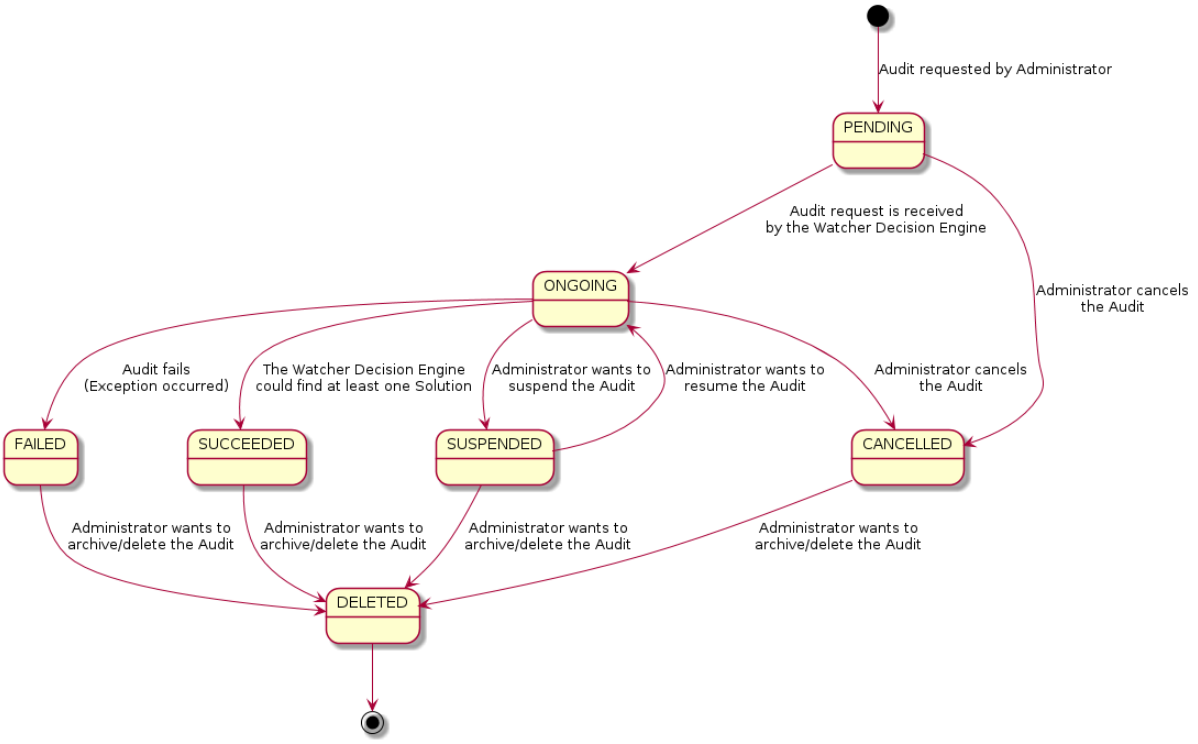
1.5 State Machine diagrams

1.5.1 Audit State Machine

An *Audit* has a life-cycle and its current state may be one of the following:

- **PENDING** : a request for an *Audit* has been submitted (either manually by the *Administrator* or automatically via some event handling mechanism) and is in the queue for being processed by the *Watcher Decision Engine*
- **ONGOING** : the *Audit* is currently being processed by the *Watcher Decision Engine*
- **SUCCEEDED** : the *Audit* has been executed successfully and at least one solution was found
- **FAILED** : an error occurred while executing the *Audit*
- **DELETED** : the *Audit* is still stored in the *Watcher database* but is not returned any more through the Watcher APIs.
- **CANCELLED** : the *Audit* was in **PENDING** or **ONGOING** state and was cancelled by the *Administrator*
- **SUSPENDED** : the *Audit* was in **ONGOING** state and was suspended by the *Administrator*

The following diagram shows the different possible states of an *Audit* and what event makes the state change to a new value:

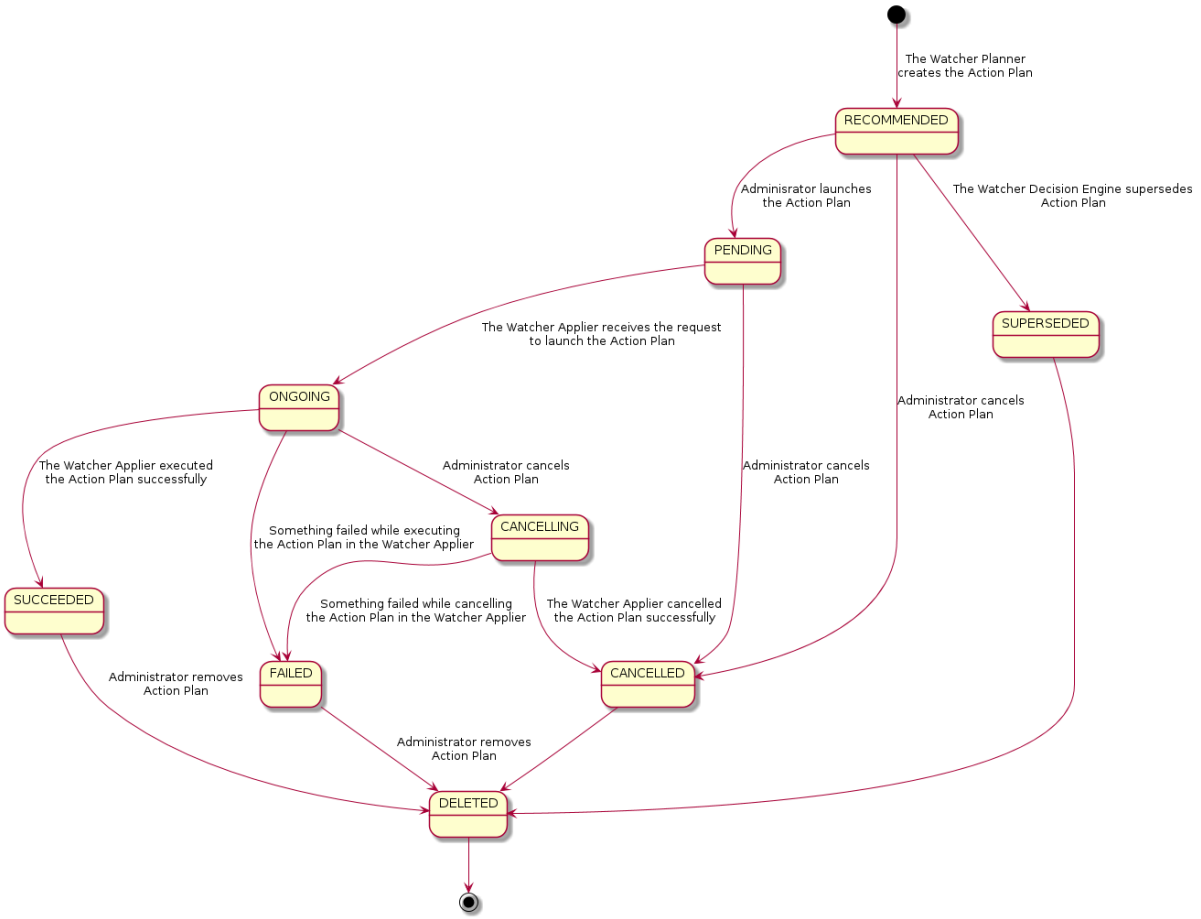


1.5.2 Action Plan State Machine

An *Action Plan* has a life-cycle and its current state may be one of the following:

- **RECOMMENDED** : the *Action Plan* is waiting for a validation from the *Administrator*
- **PENDING** : a request for an *Action Plan* has been submitted (due to an *Administrator* executing an *Audit*) and is in the queue for being processed by the *Watcher Applier*
- **ONGOING** : the *Action Plan* is currently being processed by the *Watcher Applier*
- **SUCCEEDED** : the *Action Plan* has been executed successfully (i.e. all *Actions* that it contains have been executed successfully)
- **FAILED** : an error occurred while executing the *Action Plan*
- **DELETED** : the *Action Plan* is still stored in the *Watcher database* but is not returned any more through the Watcher APIs.
- **CANCELLED** : the *Action Plan* was in **RECOMMENDED**, **PENDING** or **ONGOING** state and was cancelled by the *Administrator*
- **SUPERSEDED** : the *Action Plan* was in RECOMMENDED state and was automatically superseded by Watcher, due to an expiration delay or an update of the *Cluster data model*

The following diagram shows the different possible states of an *Action Plan* and what event makes the state change to a new value:



CHAPTER

TWO

CONTRIBUTION GUIDE

2.1 So You Want to Contribute

For general information on contributing to OpenStack, please check out the contributor guide to get started. It covers all the basics that are common to all OpenStack projects: the accounts you need, the basics of interacting with our Gerrit review system, how we communicate as a community, etc.

Below will cover the more project specific information you need to get started with Watcher.

2.1.1 Communication

IRC Channel
`#openstack-watcher` (changelog)

Mailing list(prefix subjects with `[watcher]`)
http://lists.openstack.org/pipermail/openstack-discuss/

Weekly Meetings
Bi-weekly, on Wednesdays at 08:00 UTC on odd weeks in the `#openstack-meeting-alt` IRC channel (meetings logs)

Meeting Agenda
https://wiki.openstack.org/wiki/Watcher_Meeting_Agenda

2.1.2 Contacting the Core Team

Name	IRC	Email
Li Canwei	licanwei	li.canwei2@zte.com.cn
chen ke	chenke	chen.ke14@zte.com.cn
Corne Lukken	dantalion	info@dantalion.nl
su zhengwei	suzhengwei	sugar-2008@163.com
Yumeng Bao	Yumeng	yumeng_bao@yahoo.com

2.1.3 New Feature Planning

New feature will be discussed via IRC or ML (with [Watcher] prefix). Watcher team uses blueprints in Launchpad to manage the new features.

2.1.4 Task Tracking

We track our tasks in Launchpad. If youre looking for some smaller, easier work item to pick up and get started on, search for the low-hanging-fruit tag.

2.1.5 Reporting a Bug

You found an issue and want to make sure we are aware of it? You can do so HERE.

2.1.6 Getting Your Patch Merged

Due to the small number of core reviewers of the Watcher project, we only need one +2 before +W (merge). All patches excepting for documentation or typos fixes must have unit test.

Project Team Lead Duties

All common PTL duties are enumerated here in the PTL guide.

2.2 Set up a development environment manually

This document describes getting the source from watcher Git repository for development purposes.

To install Watcher from packaging, refer instead to Watcher User Documentation.

2.2.1 Prerequisites

This document assumes you are using Ubuntu or Fedora, and that you have the following tools available on your system:

- Python 2.7 and 3.5
- git
- setuptools
- pip
- msgfmt (part of the gettext package)
- virtualenv and virtualenvwrapper

Reminder: If youre successfully using a different platform, or a different version of the above, please document your configuration here!

2.2.2 Getting the latest code

Make a clone of the code from our `Git repository`:

```
$ git clone https://opendev.org/openstack/watcher.git
```

When that is complete, you can:

```
$ cd watcher
```

2.2.3 Installing dependencies

Watcher maintains two lists of dependencies:

```
requirements.txt
test-requirements.txt
```

The first is the list of dependencies needed for running Watcher, the second list includes dependencies used for active development and testing of Watcher itself.

These dependencies can be installed from PyPi using the Python tool pip.

However, your system *may* need additional dependencies that `pip` (and by extension, PyPi) cannot satisfy. These dependencies should be installed prior to using `pip`, and the installation method may vary depending on your platform.

- Ubuntu 16.04:

  ```
  $ sudo apt-get install python-dev libssl-dev libmysqlclient-dev libffi-dev
  ```

- Fedora 24+:

  ```
  $ sudo dnf install redhat-rpm-config gcc python-devel libxml2-devel
  ```

- CentOS 7:

  ```
  $ sudo yum install gcc python-devel libxml2-devel libxslt-devel mariadb-
  ↪devel
  ```

PyPi Packages and VirtualEnv

We recommend establishing a virtualenv to run Watcher within. virtualenv limits the Python environment to just what youre installing as dependencies, useful to keep a clean environment for working on Watcher.

```
$ mkvirtualenv watcher
$ git clone https://opendev.org/openstack/watcher.git

# Use 'python setup.py' to link Watcher into Python's site-packages
$ cd watcher && python setup.py install

# Install the dependencies for running Watcher
$ pip install -r ./requirements.txt

# Install the dependencies for developing, testing, and running Watcher
$ pip install -r ./test-requirements.txt
```

This will create a local virtual environment in the directory `$WORKON_HOME`. The virtual environment can be disabled using the command:

```
$ deactivate
```

You can re-activate this virtualenv for your current shell using:

```
$ workon watcher
```

For more information on virtual environments, see virtualenv and virtualenvwrapper.

2.2.4 Verifying Watcher is set up

Once set up, either directly or within a virtualenv, you should be able to invoke Python and import the libraries. If youre using a virtualenv, dont forget to activate it:

```
$ workon watcher
```

You should then be able to `import watcher` using Python without issue:

```
$ python -c "import watcher"
```

If you can import watcher without a traceback, you should be ready to develop.

2.2.5 Run Watcher tests

Watcher provides both *unit tests* and *functional/tempest tests*. Please refer to *Developer Testing* to understand how to run them.

2.2.6 Build the Watcher documentation

You can easily build the HTML documentation from `doc/source` files, by using `tox`:

```
$ workon watcher

(watcher) $ cd watcher
(watcher) $ tox -edocs
```

The HTML files are available into `doc/build` directory.

2.2.7 Configure the Watcher services

Watcher services require a configuration file. Use tox to generate a sample configuration file that can be used to get started:

```
$ tox -e genconfig
$ cp etc/watcher.conf.sample etc/watcher.conf
```

Most of the default configuration should be enough to get you going, but you still need to configure the following sections:

- The `[database]` section to configure the *Watcher database*
- The `[keystone_authtoken]` section to configure the *Identity service* i.e. Keystone
- The `[watcher_messaging]` section to configure the OpenStack AMQP-based message bus
- The `watcher_clients_auth` section to configure Keystone client to access related OpenStack projects

So if you need some more details on how to configure one or more of these sections, please do have a look at *Configuring Watcher* before continuing.

2.2.8 Create Watcher SQL database

When initially getting set up, after youve configured which databases to use, youre probably going to need to run the following to your database schema in place:

```
$ workon watcher

(watcher) $ watcher-db-manage create_schema
```

2.2.9 Running Watcher services

To run the Watcher API service, use:

```
$ workon watcher

(watcher) $ watcher-api
```

To run the Watcher Decision Engine service, use:

```
$ workon watcher

(watcher) $ watcher-decision-engine
```

To run the Watcher Applier service, use:

```
$ workon watcher

(watcher) $ watcher-applier
```

Default configuration of these services are available into `/etc/watcher` directory. See *Configuring Watcher* for details on how Watcher is configured. By default, Watcher is configured with SQL backends.

2.2.10 Interact with Watcher

You can also interact with Watcher through its REST API. There is a Python Watcher client library python-watcherclient which interacts exclusively through the REST API, and which Watcher itself uses to provide its command-line interface.

There is also an Horizon plugin for Watcher watcher-dashboard which allows to interact with Watcher through a web-based interface.

2.2.11 Exercising the Watcher Services locally

If you would like to exercise the Watcher services in isolation within a local virtual environment, you can do this without starting any other OpenStack services. For example, this is useful for rapidly prototyping and debugging interactions over the RPC channel, testing database migrations, and so forth.

You will find in the watcher-tools project, Ansible playbooks and Docker template files to easily play with Watcher services within a minimal OpenStack isolated environment (Identity, Message Bus, SQL database, Horizon,).

2.3 Set up a development environment via DevStack

Watcher is currently able to optimize compute resources - specifically Nova compute hosts - via operations such as live migrations. In order for you to fully be able to exercise what Watcher can do, it is necessary to have a multinode environment to use.

You can set up the Watcher services quickly and easily using a Watcher DevStack plugin. See PluginModelDocs for information on DevStacks plugin model. To enable the Watcher plugin with DevStack, add the following to the `[[local|localrc]]` section of your controllers `local.conf` to enable the Watcher plugin:

```
enable_plugin watcher https://opendev.org/openstack/watcher
```

For more detailed instructions, see *Detailed DevStack Instructions*. Check out the DevStack documentation for more information regarding DevStack.

2.3.1 Quick Devstack Instructions with Datasources

Watcher requires a datasource to collect metrics from compute nodes and instances in order to execute most strategies. To enable this a `[[local|localrc]]` to setup DevStack for some of the supported datasources is provided. These examples specify the minimal configuration parameters to get both Watcher and the datasource working but can be expanded is desired.

Gnocchi

With the Gnocchi datasource most of the metrics for compute nodes and instances will work with the provided configuration but metrics that require Ironic such as `host_airflow` and `host_power` will still be unavailable as well as `instance_l3_cpu_cache`

```
[[local|localrc]]

enable_plugin watcher https://opendev.org/openstack/watcher
enable_plugin watcher-dashboard https://opendev.org/openstack/watcher-
↪dashboard
enable_plugin ceilometer https://opendev.org/openstack/ceilometer.git
enable_plugin aodh https://opendev.org/openstack/aodh
enable_plugin panko https://opendev.org/openstack/panko

CEILOMETER_BACKEND=gnocchi
[[post-config|$NOVA_CONF]]
[DEFAULT]
compute_monitors=cpu.virt_driver
```

2.3.2 Detailed DevStack Instructions

1. Obtain N (where N >= 1) servers (virtual machines preferred for DevStack). One of these servers will be the controller node while the others will be compute nodes. N is preferably >= 3 so that you have at least 2 compute nodes, but in order to stand up the Watcher services only 1 server is needed (i.e., no computes are needed if you want to just experiment with the Watcher services). These servers can be VMs running on your local machine via VirtualBox if you prefer. DevStack currently recommends that you use Ubuntu 16.04 LTS. The servers should also have connections to the same network such that they are all able to communicate with one another.

2. For each server, clone the DevStack repository and create the stack user

```
sudo apt-get update
sudo apt-get install git
git clone https://opendev.org/openstack/devstack.git
sudo ./devstack/tools/create-stack-user.sh
```

   Now you have a stack user that is used to run the DevStack processes. You may want to give your stack user a password to allow SSH via a password

```
sudo passwd stack
```

3. Switch to the stack user and clone the DevStack repo again

```
sudo su stack
cd ~
git clone https://opendev.org/openstack/devstack.git
```

4. For each compute node, copy the provided local.conf.compute example file to the compute nodes system at ~/devstack/local.conf. Make sure the HOST_IP and SERVICE_HOST values are changed appropriately - i.e., HOST_IP is set to the IP address of the compute node and SERVICE_HOST is set to the IP address of the controller node.

   If you need specific metrics collected (or want to use something other than Ceilometer), be sure to configure it. For example, in the local.conf.compute example file, the appropriate ceilometer plugins and services are enabled and disabled. If you were using something other than Ceilometer, then you would likely want to configure it likewise. The example file also sets the compute monitors nova configuration option to use the CPU virt driver. If you needed other metrics, it may be necessary to configure similar configuration options for the projects providing those metrics.

5. For the controller node, copy the provided local.conf.controller example file to the controller nodes system at ~/devstack/local.conf. Make sure the HOST_IP value is changed appropriately - i.e., HOST_IP is set to the IP address of the controller node.

   Note

   if you want to use another Watcher git repository (such as a local one), then change the enable plugin line

```
enable_plugin watcher <your_local_git_repo> [optional_branch]
```

   If you do this, then the Watcher DevStack plugin will try to pull the python-watcherclient repo from `<your_local_git_repo>/../`, so either make sure that is also available or specify WATCHERCLIENT_REPO in the `local.conf` file.

   Note

   if you want to use a specific branch, specify WATCHER_BRANCH in the local.conf file. By default it will use the master branch.

Note

watcher-api will default run under apache/httpd, set the variable WATCHER_USE_MOD_WSGI=FALSE if you do not wish to run under apache/httpd. For development environment it is suggested to set WATHCER_USE_MOD_WSGI to FALSE. For Production environment it is suggested to keep it at the default TRUE value.

6. Start stacking from the controller node:

```
./devstack/stack.sh
```

7. Start stacking on each of the compute nodes using the same command.

See also

Configure the environment for live migration via NFS. See the *Multi-Node DevStack Environment* section for more details.

2.3.3 Multi-Node DevStack Environment

Since deploying Watcher with only a single compute node is not very useful, a few tips are given here for enabling a multi-node environment with live migration.

Note

Nova supports live migration with local block storage so by default NFS is not required and is considered an advance configuration. The minimum requirements for live migration are:

- all hostnames are resolvable on each host
- all hosts have a passwordless ssh key that is trusted by the other hosts
- all hosts have a known_hosts file that lists each hosts

If these requirements are met live migration will be possible. Shared storage such as ceph, booting form cinder volume or nfs are recommend when testing evacuate if you want to preserve vm data.

Setting up SSH keys between compute nodes to enable live migration

In order for live migration to work, SSH keys need to be exchanged between each compute node:

1. The SOURCE root users public RSA key (likely in /root/.ssh/id_rsa.pub) needs to be in the DESTINATION stack users authorized_keys file (~stack/.ssh/authorized_keys). This can be accomplished by manually copying the contents from the file on the SOURCE to the DESTINATION. If you have a password configured for the stack user, then you can use the following command to accomplish the same thing:

```
ssh-copy-id -i /root/.ssh/id_rsa.pub stack@DESTINATION
```

2. The DESTINATION hosts public ECDSA key (/etc/ssh/ssh_host_ecdsa_key.pub) needs to be in the SOURCE root users known_hosts file (/root/.ssh/known_hosts). This can be accomplished by running the following on the SOURCE machine (hostname must be used):

```
ssh-keyscan -H DEST_HOSTNAME | sudo tee -a /root/.ssh/known_hosts
```

In essence, this means that every compute nodes root users public RSA key must exist in every other compute nodes stack users authorized_keys file and every compute nodes public ECDSA key needs to be in every other compute nodes root users known_hosts file.

Configuring NFS Server (ADVANCED)

If you would like to use live migration for shared storage, then the controller can serve as the NFS server if needed

```
sudo apt-get install nfs-kernel-server
sudo mkdir -p /nfs/instances
sudo chown stack:stack /nfs/instances
```

Add an entry to `/etc/exports` with the appropriate gateway and netmask information

```
/nfs/instances <gateway>/<netmask>(rw,fsid=0,insecure,no_subtree_check,async,
↪no_root_squash)
```

Export the NFS directories

```
sudo exportfs -ra
```

Make sure the NFS server is running

```
sudo service nfs-kernel-server status
```

If the server is not running, then start it

```
sudo service nfs-kernel-server start
```

Configuring NFS on Compute Node (ADVANCED)

Each compute node needs to use the NFS server to hold the instance data

```
sudo apt-get install rpcbind nfs-common
mkdir -p /opt/stack/data/instances
sudo mount <nfs-server-ip>:/nfs/instances /opt/stack/data/instances
```

If you would like to have the NFS directory automatically mounted on reboot, then add the following to `/etc/fstab`

```
<nfs-server-ip>:/nfs/instances /opt/stack/data/instances nfs auto 0 0
```

Configuring libvirt to listen on tcp (ADVANCED)

Note

By default nova will use ssh as a transport for live migration if you have a low bandwidth connection you can use tcp instead however this is generally not recommended.

Edit `/etc/libvirt/libvirtd.conf` to make sure the following values are set

```
listen_tls = 0
listen_tcp = 1
auth_tcp = "none"
```

Edit `/etc/default/libvirt-bin`

```
libvirtd_opts="-d -l"
```

Restart the libvirt service

```
sudo service libvirt-bin restart
```

VNC server configuration

The VNC server listening parameter needs to be set to any address so that the server can accept connections from all of the compute nodes.

On both the controller and compute node, in `/etc/nova/nova.conf`

```
[vnc]
server_listen = "0.0.0.0"
```

Alternatively, in devstacks `local.conf`:

```
VNCSERVER_LISTEN="0.0.0.0"
```

Environment final checkup

If you are willing to make sure everything is in order in your DevStack environment, you can run the Watcher Tempest tests which will validate its API but also that you can perform the typical Watcher workflows. To do so, have a look at the *Tempest tests* section which will explain to you how to run them.

2.4 Developer Testing

2.4.1 Unit tests

All unit tests should be run using tox. Before running the unit tests, you should download the latest watcher from the github. To run the same unit tests that are executing onto Gerrit which includes `py36`, `py37` and `pep8`, you can issue the following command:

```
$ git clone https://opendev.org/openstack/watcher
$ cd watcher
$ pip install tox
$ tox
```

If you only want to run one of the aforementioned, you can then issue one of the following:

```
$ tox -e py36
$ tox -e py37
$ tox -e pep8
```

If you only want to run specific unit test code and dont like to waste time waiting for all unit tests to execute, you can add parameters `--` followed by a regex string:

```
$ tox -e py37 -- watcher.tests.api
```

2.4.2 Tempest tests

Tempest tests for Watcher has been migrated to the external repo watcher-tempest-plugin.

2.5 Rally job

We provide, with Watcher, a Rally plugin you can use to benchmark the optimization service.

To launch this task with configured Rally you just need to run:

```
rally task start watcher/rally-jobs/watcher-watcher.yaml
```

2.5.1 Structure

- plugins - directory where you can add rally plugins. Almost everything in Rally is a plugin. Benchmark context, Benchmark scenario, SLA checks, Generic cleanup resources, .
- extra - all files from this directory will be copy pasted to gates, so you are able to use absolute paths in rally tasks. Files will be located in ~/.rally/extra/*
- watcher.yaml is a task that is run in gates against OpenStack deployed by DevStack

2.5.2 Useful links

- How to install: https://docs.openstack.org/rally/latest/install_and_upgrade/install.html
- How to set Rally up and launch your first scenario: https://rally.readthedocs.io/en/latest/quick_start/tutorial/step_1_setting_up_env_and_running_benchmark_from_samples.html
- More about Rally: https://docs.openstack.org/rally/latest/
- Rally project info and release notes: https://docs.openstack.org/rally/latest/project_info/index.html
- How to add rally-gates: https://docs.openstack.org/rally/latest/quick_start/gates.html#gate-jobs
- About plugins: https://docs.openstack.org/rally/latest/plugins/index.html
- Plugin samples: https://github.com/openstack/rally/tree/master/samples/

CHAPTER

THREE

INSTALL GUIDE

3.1 Infrastructure Optimization service overview

The Infrastructure Optimization service provides flexible and scalable optimization service for multi-tenant OpenStack based clouds.

The Infrastructure Optimization service consists of the following components:

`watcher` command-line client
A CLI to communicate with `watcher-api` to optimize the cloud.

`watcher-api` service
An OpenStack-native REST API that accepts and responds to end-user calls by processing them and forwarding to appropriate underlying watcher services via AMQP.

`watcher-decision-engine` service
It runs audit and return an action plan to achieve optimization goal specified by the end-user in audit.

`watcher-applier` service
It executes action plan built by watcher-decision-engine. It interacts with other OpenStack components like nova to execute the given action plan.

`watcher-dashboard`
Watcher UI implemented as a plugin for the OpenStack Dashboard.

3.2 Install and configure

This section describes how to install and configure the Infrastructure Optimization service, code-named watcher, on the controller node.

This section assumes that you already have a working OpenStack environment with at least the following components installed: Identity Service, Compute Service, Telemetry data collection service.

Note that installation and configuration vary by distribution.

3.2.1 Install and configure for Red Hat Enterprise Linux and CentOS

This section describes how to install and configure the Infrastructure Optimization service for Red Hat Enterprise Linux 7 and CentOS 7.

Prerequisites

Before you install and configure the Infrastructure Optimization service, you must create a database, service credentials, and API endpoints.

1. Create the database, complete these steps:

   - Use the database access client to connect to the database server as the `root` user:

     ```
     # mysql
     ```

   - Create the `watcher` database:

     ```
     CREATE DATABASE watcher CHARACTER SET utf8;
     ```

   - Grant proper access to the `watcher` database:

     ```
     GRANT ALL PRIVILEGES ON watcher.* TO 'watcher'@'localhost' \
       IDENTIFIED BY 'WATCHER_DBPASS';
     GRANT ALL PRIVILEGES ON watcher.* TO 'watcher'@'%' \
       IDENTIFIED BY 'WATCHER_DBPASS';
     ```

     Replace `WATCHER_DBPASS` with a suitable password.

   - Exit the database access client.

     ```
     exit;
     ```

2. Source the `admin` credentials to gain access to admin-only CLI commands:

   ```
   $ . admin-openrc
   ```

3. To create the service credentials, complete these steps:

   - Create the `watcher` user:

     ```
     $ openstack user create --domain default --password-prompt watcher
       User Password:
       Repeat User Password:
       +---------------------+----------------------------------+
       | Field               | Value                            |
       +---------------------+----------------------------------+
       | domain_id           | default                          |
       | enabled             | True                             |
       | id                  | b18ee38e06034b748141beda8fc8bfad |
       | name                | watcher                          |
       | options             | {}                               |
       | password_expires_at | None                             |
       +---------------------+----------------------------------+
     ```

   - Add the `admin` role to the `watcher` user:

     ```
     $ openstack role add --project service --user watcher admin
     ```

Note

This command produces no output.

- Create the watcher service entities:

```
$ openstack service create --name watcher --description
↪"Infrastructure Optimization" infra-optim
  +-------------+----------------------------------+
  | Field       | Value                            |
  +-------------+----------------------------------+
  | description | Infrastructure Optimization      |
  | enabled     | True                             |
  | id          | d854f6fff0a64f77bda8003c8dedfada |
  | name        | watcher                          |
  | type        | infra-optim                      |
  +-------------+----------------------------------+
```

4. Create the Infrastructure Optimization service API endpoints:

```
$ openstack endpoint create --region RegionOne \
  infra-optim public http://controller:9322
  +-------------+----------------------------------+
  | Field       | Value                            |
  +-------------+----------------------------------+
  | description | Infrastructure Optimization      |
  | enabled     | True                             |
  | id          | d854f6fff0a64f77bda8003c8dedfada |
  | name        | watcher                          |
  | type        | infra-optim                      |
  +-------------+----------------------------------+

$ openstack endpoint create --region RegionOne \
  infra-optim internal http://controller:9322
  +--------------+----------------------------------+
  | Field        | Value                            |
  +--------------+----------------------------------+
  | enabled      | True                             |
  | id           | 225aef8465ef4df48a341aaaf2b0a390 |
  | interface    | internal                         |
  | region       | RegionOne                        |
  | region_id    | RegionOne                        |
  | service_id   | d854f6fff0a64f77bda8003c8dedfada |
  | service_name | watcher                          |
  | service_type | infra-optim                      |
  | url          | http://controller:9322           |
  +--------------+----------------------------------+

$ openstack endpoint create --region RegionOne \
  infra-optim admin http://controller:9322
```

(continues on next page)

(continued from previous page)

```
+--------------+----------------------------------+
| Field        | Value                            |
+--------------+----------------------------------+
| enabled      | True                             |
| id           | 375eb5057fb546edbdf3ee4866179672 |
| interface    | admin                            |
| region       | RegionOne                        |
| region_id    | RegionOne                        |
| service_id   | d854f6fff0a64f77bda8003c8dedfada |
| service_name | watcher                          |
| service_type | infra-optim                      |
| url          | http://controller:9322           |
+--------------+----------------------------------+
```

Install and configure components

1. Install the packages:

```
# sudo yum install openstack-watcher-api openstack-watcher-applier \
  openstack-watcher-decision-engine
```

2. Edit the `/etc/watcher/watcher.conf` file and complete the following actions:

   - In the `[database]` section, configure database access:

```
[database]
...
connection = mysql+pymysql://watcher:WATCHER_DBPASS@controller/
↪watcher?charset=utf8
```

   - In the `[DEFAULT]` section, configure the transport url for RabbitMQ message broker.

```
[DEFAULT]
...
control_exchange = watcher
transport_url = rabbit://openstack:RABBIT_PASS@controller
```

     Replace the RABBIT_PASS with the password you chose for OpenStack user in RabbitMQ.

   - In the `[keystone_authtoken]` section, configure Identity service access.

```
[keystone_authtoken]
...
www_authenticate_uri = http://controller:5000
auth_url = http://controller:5000
memcached_servers = controller:11211
auth_type = password
project_domain_name = default
user_domain_name = default
project_name = service
username = watcher
password = WATCHER_PASS
```

Replace WATCHER_PASS with the password you chose for the watcher user in the Identity service.

- Watcher interacts with other OpenStack projects via project clients, in order to instantiate these clients, Watcher requests new session from Identity service. In the [watcher_clients_auth] section, configure the identity service access to interact with other OpenStack project clients.

```
[watcher_clients_auth]
...
auth_type = password
auth_url = http://controller:5000
username = watcher
password = WATCHER_PASS
project_domain_name = default
user_domain_name = default
project_name = service
```

Replace WATCHER_PASS with the password you chose for the watcher user in the Identity service.

- In the [api] section, configure host option.

```
[api]
...
host = controller
```

Replace controller with the IP address of the management network interface on your controller node, typically 10.0.0.11 for the first node in the example architecture.

- In the [oslo_messaging_notifications] section, configure the messaging driver.

```
[oslo_messaging_notifications]
...
driver = messagingv2
```

3. Populate watcher database:

```
su -s /bin/sh -c "watcher-db-manage --config-file /etc/watcher/watcher.
↪conf upgrade"
```

Finalize installation

Start the Infrastructure Optimization services and configure them to start when the system boots:

```
# systemctl enable openstack-watcher-api.service \
  openstack-watcher-decision-engine.service \
  openstack-watcher-applier.service

# systemctl start openstack-watcher-api.service \
  openstack-watcher-decision-engine.service \
  openstack-watcher-applier.service
```

3.2.2 Install and configure for Ubuntu

This section describes how to install and configure the Infrastructure Optimization service for Ubuntu 16.04 (LTS).

Prerequisites

Before you install and configure the Infrastructure Optimization service, you must create a database, service credentials, and API endpoints.

1. Create the database, complete these steps:

   - Use the database access client to connect to the database server as the `root` user:

     ```
     # mysql
     ```

   - Create the `watcher` database:

     ```
     CREATE DATABASE watcher CHARACTER SET utf8;
     ```

   - Grant proper access to the `watcher` database:

     ```
     GRANT ALL PRIVILEGES ON watcher.* TO 'watcher'@'localhost' \
       IDENTIFIED BY 'WATCHER_DBPASS';
     GRANT ALL PRIVILEGES ON watcher.* TO 'watcher'@'%' \
       IDENTIFIED BY 'WATCHER_DBPASS';
     ```

     Replace `WATCHER_DBPASS` with a suitable password.

   - Exit the database access client.

     ```
     exit;
     ```

2. Source the `admin` credentials to gain access to admin-only CLI commands:

   ```
   $ . admin-openrc
   ```

3. To create the service credentials, complete these steps:

   - Create the `watcher` user:

     ```
     $ openstack user create --domain default --password-prompt watcher
       User Password:
       Repeat User Password:
       +---------------------+----------------------------------+
       | Field               | Value                            |
       +---------------------+----------------------------------+
       | domain_id           | default                          |
       | enabled             | True                             |
       | id                  | b18ee38e06034b748141beda8fc8bfad |
       | name                | watcher                          |
       | options             | {}                               |
       | password_expires_at | None                             |
       +---------------------+----------------------------------+
     ```

   - Add the `admin` role to the `watcher` user:

```
$ openstack role add --project service --user watcher admin
```

Note

This command produces no output.

- Create the watcher service entities:

```
$ openstack service create --name watcher --description
↪"Infrastructure Optimization" infra-optim
  +-------------+----------------------------------+
  | Field       | Value                            |
  +-------------+----------------------------------+
  | description | Infrastructure Optimization      |
  | enabled     | True                             |
  | id          | d854f6fff0a64f77bda8003c8dedfada |
  | name        | watcher                          |
  | type        | infra-optim                      |
  +-------------+----------------------------------+
```

4. Create the Infrastructure Optimization service API endpoints:

```
$ openstack endpoint create --region RegionOne \
  infra-optim public http://controller:9322
  +-------------+----------------------------------+
  | Field       | Value                            |
  +-------------+----------------------------------+
  | description | Infrastructure Optimization      |
  | enabled     | True                             |
  | id          | d854f6fff0a64f77bda8003c8dedfada |
  | name        | watcher                          |
  | type        | infra-optim                      |
  +-------------+----------------------------------+

$ openstack endpoint create --region RegionOne \
  infra-optim internal http://controller:9322
  +--------------+----------------------------------+
  | Field        | Value                            |
  +--------------+----------------------------------+
  | enabled      | True                             |
  | id           | 225aef8465ef4df48a341aaaf2b0a390 |
  | interface    | internal                         |
  | region       | RegionOne                        |
  | region_id    | RegionOne                        |
  | service_id   | d854f6fff0a64f77bda8003c8dedfada |
  | service_name | watcher                          |
  | service_type | infra-optim                      |
  | url          | http://controller:9322           |
  +--------------+----------------------------------+
```

(continues on next page)

(continued from previous page)

```
$ openstack endpoint create --region RegionOne \
  infra-optim admin http://controller:9322
  +--------------+----------------------------------+
  | Field        | Value                            |
  +--------------+----------------------------------+
  | enabled      | True                             |
  | id           | 375eb5057fb546edbdf3ee4866179672 |
  | interface    | admin                            |
  | region       | RegionOne                        |
  | region_id    | RegionOne                        |
  | service_id   | d854f6fff0a64f77bda8003c8dedfada |
  | service_name | watcher                          |
  | service_type | infra-optim                      |
  | url          | http://controller:9322           |
  +--------------+----------------------------------+
```

Install and configure components

1. Install the packages:

```
# apt install watcher-api watcher-decision-engine \
  watcher-applier

# apt install python-watcherclient
```

2. Edit the `/etc/watcher/watcher.conf` file and complete the following actions:

   - In the `[database]` section, configure database access:

```
[database]
...
connection = mysql+pymysql://watcher:WATCHER_DBPASS@controller/
↪watcher?charset=utf8
```

   - In the `[DEFAULT]` section, configure the transport url for RabbitMQ message broker.

```
[DEFAULT]
...
control_exchange = watcher
transport_url = rabbit://openstack:RABBIT_PASS@controller
```

     Replace the RABBIT_PASS with the password you chose for OpenStack user in RabbitMQ.

   - In the `[keystone_authtoken]` section, configure Identity service access.

```
[keystone_authtoken]
...
www_authenticate_uri = http://controller:5000
auth_url = http://controller:5000
memcached_servers = controller:11211
```

(continues on next page)

(continued from previous page)

```
auth_type = password
project_domain_name = default
user_domain_name = default
project_name = service
username = watcher
password = WATCHER_PASS
```

Replace WATCHER_PASS with the password you chose for the watcher user in the Identity service.

- Watcher interacts with other OpenStack projects via project clients, in order to instantiate these clients, Watcher requests new session from Identity service. In the `[watcher_clients_auth]` section, configure the identity service access to interact with other OpenStack project clients.

```
[watcher_clients_auth]
...
auth_type = password
auth_url = http://controller:5000
username = watcher
password = WATCHER_PASS
project_domain_name = default
user_domain_name = default
project_name = service
```

Replace WATCHER_PASS with the password you chose for the watcher user in the Identity service.

- In the `[api]` section, configure host option.

```
[api]
...
host = controller
```

Replace controller with the IP address of the management network interface on your controller node, typically 10.0.0.11 for the first node in the example architecture.

- In the `[oslo_messaging_notifications]` section, configure the messaging driver.

```
[oslo_messaging_notifications]
...
driver = messagingv2
```

3. Populate watcher database:

```
su -s /bin/sh -c "watcher-db-manage --config-file /etc/watcher/watcher.
↪conf upgrade"
```

Finalize installation

Start the Infrastructure Optimization services and configure them to start when the system boots:

```
# systemctl enable watcher-api.service \
  watcher-decision-engine.service \
  watcher-applier.service

# systemctl start watcher-api.service \
  watcher-decision-engine.service \
  watcher-applier.service
```

3.3 Verify operation

Verify operation of the Infrastructure Optimization service.

Note

Perform these commands on the controller node.

1. Source the `admin` project credentials to gain access to admin-only CLI commands:

```
$ . admin-openrc
```

2. List service components to verify successful launch and registration of each process:

```
$ openstack optimize service list
  +----+-------------------------+------------+--------+
  | ID | Name                    | Host       | Status |
  +----+-------------------------+------------+--------+
  |  1 | watcher-decision-engine | controller | ACTIVE |
  |  2 | watcher-applier         | controller | ACTIVE |
  +----+-------------------------+------------+--------+
```

3. List goals and strategies:

```
$ openstack optimize goal list
  +--------------------------------------+----------------------+---------
↪-------------+
  | UUID                                 | Name                 | Display␣
↪name         |
  +--------------------------------------+----------------------+---------
↪-------------+
  | a8cd6d1a-008b-4ff0-8dbc-b30493fcc5b9 | dummy                | Dummy␣
↪goal           |
  | 03953f2f-02d0-42b5-9a12-7ba500a54395 | workload_balancing   |␣
↪Workload Balancing   |
  | de0f8714-984b-4d6b-add1-9cad8120fbce | server_consolidation | Server␣
↪Consolidation |
  | f056bc80-c6d1-40dc-b002-938ccade9385 | thermal_optimization | Thermal␣
↪Optimization |
```

(continues on next page)

(continued from previous page)

```
  | e7062856-892e-4f0f-b84d-b828464b3fd0 | airflow_optimization | Airflow␣
↪Optimization |
  | 1f038da9-b36c-449f-9f04-c225bf3eb478 | unclassified         |␣
↪Unclassified         |
  +--------------------------------------+----------------------+---------
↪-------------+

$ openstack optimize strategy list
+--------------------------------------+--------------------------+------
↪---------------------------------------+----------------------+
| UUID                                 | Name                     |␣
↪Display name                                | Goal                 |
+--------------------------------------+--------------------------+------
↪---------------------------------------+----------------------+
| 98ae84c8-7c9b-4cbd-8d9c-4bd7c6b106eb | dummy                    |␣
↪Dummy strategy                              | dummy                |
| 02a170b6-c72e-479d-95c0-8a4fdd4cc1ef | dummy_with_scorer        |␣
↪Dummy Strategy using sample Scoring Engines | dummy                |
| 8bf591b8-57e5-4a9e-8c7d-c37bda735a45 | outlet_temperature       |␣
↪Outlet temperature based strategy           | thermal_optimization |
| 8a0810fb-9d9a-47b9-ab25-e442878abc54 | vm_workload_consolidation | VM␣
↪Workload Consolidation Strategy             | server_consolidation |
| 1718859c-3eb5-45cb-9220-9cb79fe42fa5 | basic                    |␣
↪Basic offline consolidation                 | server_consolidation |
| b5e7f5f1-4824-42c7-bb52-cf50724f67bf | workload_stabilization   |␣
↪Workload stabilization                      | workload_balancing   |
| f853d71e-9286-4df3-9d3e-8eaf0f598e07 | workload_balance         |␣
↪Workload Balance Migration Strategy         | workload_balancing   |
| 58bdfa89-95b5-4630-adf6-fd3af5ff1f75 | uniform_airflow          |␣
↪Uniform airflow migration strategy          | airflow_optimization |
| 66fde55d-a612-4be9-8cb0-ea63472b420b | dummy_with_resize        |␣
↪Dummy strategy with resize                  | dummy                |
+--------------------------------------+--------------------------+------
↪---------------------------------------+----------------------+
```

4. Run an action plan by creating an audit with dummy goal:

```
$ openstack optimize audit create --goal dummy
  +-------------+--------------------------------------+
  | Field       | Value                                |
  +-------------+--------------------------------------+
  | UUID        | e94d4826-ad4e-44df-ad93-dff489fde457 |
  | Created At  | 2017-05-23T11:46:58.763394+00:00     |
  | Updated At  | None                                 |
  | Deleted At  | None                                 |
  | State       | PENDING                              |
  | Audit Type  | ONESHOT                              |
  | Parameters  | {}                                   |
  | Interval    | None                                 |
  | Goal        | dummy                                |
```

(continues on next page)

(continued from previous page)

```
  | Strategy     | auto                                 |
  | Audit Scope  | []                                   |
  | Auto Trigger | False                                |
  +--------------+--------------------------------------+

$ openstack optimize audit list
  +--------------------------------------+-----------+----------+------
↪-+----------+--------------+
  | UUID                                 | Audit Type | State     | Goal␣
↪ | Strategy | Auto Trigger |
  +--------------------------------------+-----------+----------+------
↪-+----------+--------------+
  | e94d4826-ad4e-44df-ad93-dff489fde457 | ONESHOT    | SUCCEEDED |␣
↪dummy | auto     | False        |
  +--------------------------------------+-----------+----------+------
↪-+----------+--------------+

$ openstack optimize actionplan list
  +--------------------------------------+--------------------------------
↪-------+-------------+-----------+-----------------+
  | UUID                                 | Audit                          ␣
↪       | State       | Updated At | Global efficacy |
  +--------------------------------------+--------------------------------
↪-------+-------------+-----------+-----------------+
  | ba9ce6b3-969c-4b8e-bb61-ae24e8630f81 | e94d4826-ad4e-44df-ad93-
↪dff489fde457 | RECOMMENDED | None       | None            |
  +--------------------------------------+--------------------------------
↪-------+-------------+-----------+-----------------+

$ openstack optimize actionplan start ba9ce6b3-969c-4b8e-bb61-ae24e8630f81
  +--------------------+--------------------------------------+
  | Field              | Value                                |
  +--------------------+--------------------------------------+
  | UUID               | ba9ce6b3-969c-4b8e-bb61-ae24e8630f81 |
  | Created At         | 2017-05-23T11:46:58+00:00            |
  | Updated At         | 2017-05-23T11:53:12+00:00            |
  | Deleted At         | None                                 |
  | Audit              | e94d4826-ad4e-44df-ad93-dff489fde457 |
  | Strategy           | dummy                                |
  | State              | ONGOING                              |
  | Efficacy indicators | []                                   |
  | Global efficacy    | {}                                   |
  +--------------------+--------------------------------------+

$ openstack optimize actionplan list
+--------------------------------------+--------------------------------
↪----+----------+-------------------------+-----------------+
| UUID                                 | Audit                          ␣
↪    | State    | Updated At              | Global efficacy |
```

(continues on next page)

(continued from previous page)

```
+--------------------------------------+--------------------------------------
↪----+-----------+--------------------------+-----------------+
| ba9ce6b3-969c-4b8e-bb61-ae24e8630f81 | e94d4826-ad4e-44df-ad93-
↪dff489fde457 | SUCCEEDED | 2017-05-23T11:53:16+00:00 | None            |
+--------------------------------------+--------------------------------------
↪----+-----------+--------------------------+-----------------+
```

3.4 Next steps

Your OpenStack environment now includes the watcher service.

To add additional services, see https://docs.openstack.org/queens/install/.

The Infrastructure Optimization service (Watcher) provides flexible and scalable resource optimization service for multi-tenant OpenStack-based clouds.

Watcher provides a complete optimization loop including everything from a metrics receiver, complex event processor and profiler, optimization processor and an action plan applier. This provides a robust framework to realize a wide range of cloud optimization goals, including the reduction of data center operating costs, increased system performance via intelligent virtual machine migration, increased energy efficiency and more!

Watcher also supports a pluggable architecture by which custom optimization algorithms, data metrics and data profilers can be developed and inserted into the Watcher framework.

Check the documentation for watcher optimization strategies at Strategies.

Check watcher glossary at Glossary.

This chapter assumes a working setup of OpenStack following the OpenStack Installation Tutorial.

CHAPTER

FOUR

ADMINISTRATOR GUIDE

4.1 Installing API behind mod_wsgi

1. Install the Apache Service:

```
Fedora 21/RHEL7/CentOS7:
  sudo yum install httpd

Fedora 22 (or higher):
  sudo dnf install httpd

Debian/Ubuntu:
  apt-get install apache2
```

2. Copy `etc/apache2/watcher.conf` under the apache sites:

```
Fedora/RHEL7/CentOS7:
  sudo cp etc/apache2/watcher /etc/httpd/conf.d/watcher.conf

Debian/Ubuntu:
  sudo cp etc/apache2/watcher /etc/apache2/sites-available/watcher.conf
```

3. Edit `<apache-configuration-dir>/watcher.conf` according to installation and environment.

   - Modify the `WSGIDaemonProcess` directive to set the `user` and `group` values to appropriate user on your server.
   - Modify the `WSGIScriptAlias` directive to point to the watcher/api/app.wsgi script.
   - Modify the `Directory` directive to set the path to the Watcher API code.
   - Modify the `ErrorLog and CustomLog` to redirect the logs to the right directory.

4. Enable the apache watcher site and reload:

```
Fedora/RHEL7/CentOS7:
  sudo systemctl reload httpd

Debian/Ubuntu:
  sudo a2ensite watcher
  sudo service apache2 reload
```

4.2 Guru Meditation Reports

Watcher contains a mechanism whereby developers and system administrators can generate a report about the state of a running Watcher service. This report is called a *Guru Meditation Report* (*GMR* for short).

4.2.1 Generating a GMR

A *GMR* can be generated by sending the *USR2* signal to any Watcher process with support (see below). The *GMR* will then be outputted as standard error for that particular process.

For example, suppose that `watcher-api` has process id `8675`, and was run with `2>/var/log/watcher/watcher-api-err.log`. Then, `kill -USR2 8675` will trigger the Guru Meditation report to be printed to `/var/log/watcher/watcher-api-err.log`.

4.2.2 Structure of a GMR

The *GMR* is designed to be extensible; any particular service may add its own sections. However, the base *GMR* consists of several sections:

Package
: Shows information about the package to which this process belongs, including version information.

Threads
: Shows stack traces and thread ids for each of the threads within this process.

Green Threads
: Shows stack traces for each of the green threads within this process (green threads dont have thread ids).

Configuration
: Lists all the configuration options currently accessible via the CONF object for the current process.

Plugins
: Lists all the plugins currently accessible by the Watcher service.

4.3 Policies

Warning

JSON formatted policy file is deprecated since Watcher 6.0.0 (Wallaby). This oslopolicy-convert-json-to-yaml tool will migrate your existing JSON-formatted policy file to YAML in a backward-compatible way.

Watchers public API calls may be restricted to certain sets of users using a policy configuration file. This document explains exactly how policies are configured and what they apply to.

A policy is composed of a set of rules that are used in determining if a particular action may be performed by the authorized tenant.

4.3.1 Constructing a Policy Configuration File

A policy configuration file is a simply JSON object that contain sets of rules. Each top-level key is the name of a rule. Each rule is a string that describes an action that may be performed in the Watcher API.

The actions that may have a rule enforced on them are:

- `strategy:get_all`, `strategy:detail` - List available strategies
  - `GET /v1/strategies`
  - `GET /v1/strategies/detail`
- `strategy:get` - Retrieve a specific strategy entity
  - `GET /v1/strategies/<STRATEGY_UUID>`
  - `GET /v1/strategies/<STRATEGY_NAME>`
- `goal:get_all`, `goal:detail` - List available goals
  - `GET /v1/goals`
  - `GET /v1/goals/detail`
- `goal:get` - Retrieve a specific goal entity
  - `GET /v1/goals/<GOAL_UUID>`
  - `GET /v1/goals/<GOAL_NAME>`
- `audit_template:get_all`, `audit_template:detail` - List available audit_templates
  - `GET /v1/audit_templates`
  - `GET /v1/audit_templates/detail`
- `audit_template:get` - Retrieve a specific audit template entity
  - `GET /v1/audit_templates/<AUDIT_TEMPLATE_UUID>`
  - `GET /v1/audit_templates/<AUDIT_TEMPLATE_NAME>`
- `audit_template:create` - Create an audit template entity
  - `POST /v1/audit_templates`
- `audit_template:delete` - Delete an audit template entity
  - `DELETE /v1/audit_templates/<AUDIT_TEMPLATE_UUID>`
  - `DELETE /v1/audit_templates/<AUDIT_TEMPLATE_NAME>`
- `audit_template:update` - Update an audit template entity
  - `PATCH /v1/audit_templates/<AUDIT_TEMPLATE_UUID>`
  - `PATCH /v1/audit_templates/<AUDIT_TEMPLATE_NAME>`
- `audit:get_all`, `audit:detail` - List available audits
  - `GET /v1/audits`
  - `GET /v1/audits/detail`
- `audit:get` - Retrieve a specific audit entity
  - `GET /v1/audits/<AUDIT_UUID>`

- `audit:create` - Create an audit entity
  - `POST /v1/audits`
- `audit:delete` - Delete an audit entity
  - `DELETE /v1/audits/<AUDIT_UUID>`
- `audit:update` - Update an audit entity
  - `PATCH /v1/audits/<AUDIT_UUID>`
- `action_plan:get_all`, `action_plan:detail` - List available action plans
  - `GET /v1/action_plans`
  - `GET /v1/action_plans/detail`
- `action_plan:get` - Retrieve a specific action plan entity
  - `GET /v1/action_plans/<ACTION_PLAN_UUID>`
- `action_plan:delete` - Delete an action plan entity
  - `DELETE /v1/action_plans/<ACTION_PLAN_UUID>`
- `action_plan:update` - Update an action plan entity
  - `PATCH /v1/audits/<ACTION_PLAN_UUID>`
- `action:get_all`, `action:detail` - List available action
  - `GET /v1/actions`
  - `GET /v1/actions/detail`
- `action:get` - Retrieve a specific action plan entity
  - `GET /v1/actions/<ACTION_UUID>`
- `service:get_all`, `service:detail` - List available Watcher services
  - `GET /v1/services`
  - `GET /v1/services/detail`
- `service:get` - Retrieve a specific Watcher service entity
  - `GET /v1/services/<SERVICE_ID>`

To limit an action to a particular role or roles, you list the roles like so

```
{
  "audit:create": ["role:admin", "role:superuser"]
}
```

The above would add a rule that only allowed users that had roles of either admin or superuser to launch an audit.

4.4 Strategies

4.4.1 Actuator

Synopsis

display name: `Actuator`

goal: `unclassified`

Actuator

Actuator that simply executes the actions given as parameter

This strategy allows anyone to create an action plan with a predefined set of actions. This strategy can be used for 2 different purposes:

- Test actions
- Use this strategy based on an event trigger to perform some explicit task

Requirements

Metrics

None

Cluster data model

None

Actions

Default Watchers actions.

Planner

Default Watchers planner:

Weight planner implementation

This implementation builds actions with parents in accordance with weights. Set of actions having a higher weight will be scheduled before the other ones. There are two config options to configure: action_weights and parallelization.

Limitations

- This planner requires to have action_weights and parallelization configs tuned well.

Configuration

Strategy parameters are:

parameter	type	default Value	description
`actions`	array	None	Actions to be executed.

The elements of actions array are:

parameter	type	default Value	description
action_type	string	None	Action name defined in setup.cfg(mandatory)
resource_id	string	None	Resource_id of the action.
input_parameters	object	None	Input_parameters of the action(mandatory).

Efficacy Indicator

None

Algorithm

This strategy create an action plan with a predefined set of actions.

How to use it ?

```
$ openstack optimize audittemplate create \
  at1 unclassified --strategy actuator

$ openstack optimize audit create -a at1 \
  -p actions='[{"action_type": "migrate", "resource_id": "56a40802-6fde-4b59-
↪957c-c84baec7eaed", "input_parameters": {"migration_type": "live", "source_
↪node": "s01"}}]'
```

External Links

None

4.4.2 Basic Offline Server Consolidation

Synopsis

display name: `Basic offline consolidation`

goal: `server_consolidation`

> Good server consolidation strategy
>
> Basic offline consolidation using live migration
>
> Consolidation of VMs is essential to achieve energy optimization in cloud environments such as OpenStack. As VMs are spinned up and/or moved over time, it becomes necessary to migrate VMs among servers to lower the costs. However, migration of VMs introduces runtime overheads and consumes extra energy, thus a good server consolidation strategy should carefully plan for migration in order to both minimize energy consumption and comply to the various SLAs.
>
> This algorithm not only minimizes the overall number of used servers, but also minimizes the number of migrations.
>
> It has been developed only for tests. You must have at least 2 physical compute nodes to run it, so you can easily run it on DevStack. It assumes that live migration is possible on your OpenStack cluster.

Requirements

Metrics

The *basic* strategy requires the following metrics:

metric	service name	plugins	comment
`compute.node.cpu.percent`	ceilometer	none	need to set the `compute_monitors` option to `cpu.virt_driver` in the nova.conf.
`cpu`	ceilometer	none	

Cluster data model

Default Watchers Compute cluster data model:

> Nova cluster data model collector
>
> The Nova cluster data model collector creates an in-memory representation of the resources exposed by the compute service.

Actions

Default Watchers actions:

action	description
`migration`	Migrates a server to a destination nova-compute host This action will allow you to migrate a server to another compute destination host. Migration type live can only be used for migrating active VMs. Migration type cold can be used for migrating non-active VMs as well active VMs, which will be shut down while migrating. The action schema is: `schema = Schema({ 'resource_id': str, # should be a UUID 'migration_type': str, # choices -> "live", "cold" 'destination_node': str, 'source_node': str, })` The *resource_id* is the UUID of the server to migrate. The *source_node* and *destination_node* parameters are respectively the source and the destination compute hostname (list of available compute hosts is returned by this command: `nova service-list --binary nova-compute`). **Note** Nova API version must be 2.56 or above if *destination_node* parameter is given.
`change_nova_service_state`	Disables or enables the nova-compute service, deployed on a host By using this action, you will be able to update the state of a nova-compute service. A disabled nova-compute service can not be selected by the nova scheduler for future deployment of server. The action schema is: `schema = Schema({ 'resource_id': str, 'state': str, 'disabled_reason': str, })` The *resource_id* references a nova-compute service name (list of available nova-compute services is returned by this command: `nova service-list --binary nova-compute`). The *state* value should either be *ONLINE* or *OFFLINE*. The *disabled_reason* references the reason why Watcher disables this nova-compute service. The value should be with *watcher_* prefix, such as *watcher_disabled*.

Planner

Default Watchers planner:

> Weight planner implementation
>
> This implementation builds actions with parents in accordance with weights. Set of actions having a higher weight will be scheduled before the other ones. There are two config options to configure: action_weights and parallelization.
>
> *Limitations*
>
> - This planner requires to have action_weights and parallelization configs tuned well.

Configuration

Strategy parameter is:

parameter	type	default Value	description
migration_	Number	0	Maximum number of combinations to be tried by the strategy while searching for potential candidates. To remove the limit, set it to 0
period	Number	7200	The time interval in seconds for getting statistic aggregation from metric data source

Efficacy Indicator

```
[{'name': 'released_nodes_ratio', 'description': 'Ratio of released compute␣
↪nodes divided by the total number of enabled compute nodes.', 'unit': '%',
↪'value': 0}]
```

How to use it ?

```
$ openstack optimize audittemplate create \
  at1 server_consolidation --strategy basic

$ openstack optimize audit create -a at1 -p migration_attempts=4
```

External Links

None.

4.4.3 Host Maintenance Strategy

Synopsis

display name: `Host Maintenance Strategy`

goal: `cluster_maintaining`

> [PoC]Host Maintenance
>
> *Description*

It is a migration strategy for one compute node maintenance, without having the users application been interrupted. If given one backup node, the strategy will firstly migrate all instances from the maintenance node to the backup node. If the backup node is not provided, it will migrate all instances, relying on nova-scheduler.

Requirements

- You must have at least 2 physical compute nodes to run this strategy.

Limitations

- This is a proof of concept that is not meant to be used in production
- It migrates all instances from one host to other hosts. Its better to execute such strategy when load is not heavy, and use this algorithm with *ONESHOT* audit.
- It assumes that cold and live migrations are possible.

Requirements

None.

Metrics

None

Cluster data model

Default Watchers Compute cluster data model:

Nova cluster data model collector

The Nova cluster data model collector creates an in-memory representation of the resources exposed by the compute service.

Actions

Default Watchers actions:

action	description
migration	Migrates a server to a destination nova-compute host This action will allow you to migrate a server to another compute destination host. Migration type live can only be used for migrating active VMs. Migration type cold can be used for migrating non-active VMs as well active VMs, which will be shut down while migrating. The action schema is: `schema = Schema({ 'resource_id': str, # should be a UUID 'migration_type': str, # choices -> "live", "cold" 'destination_node': str, 'source_node': str, })` The *resource_id* is the UUID of the server to migrate. The *source_node* and *destination_node* parameters are respectively the source and the destination compute hostname (list of available compute hosts is returned by this command: `nova service-list --binary nova-compute`). **Note** Nova API version must be 2.56 or above if *destination_node* parameter is given.
change_nova_service_state	Disables or enables the nova-compute service, deployed on a host By using this action, you will be able to update the state of a nova-compute service. A disabled nova-compute service can not be selected by the nova scheduler for future deployment of server. The action schema is: `schema = Schema({ 'resource_id': str, 'state': str, 'disabled_reason': str, })` The *resource_id* references a nova-compute service name (list of available nova-compute services is returned by this command: `nova service-list --binary nova-compute`). The *state* value should either be *ONLINE* or *OFFLINE*. The *disabled_reason* references the reason why Watcher disables this nova-compute service. The value should be with *watcher_* prefix, such as *watcher_disabled*.

Planner

Default Watchers planner:

> Weight planner implementation
>
> This implementation builds actions with parents in accordance with weights. Set of actions having a higher weight will be scheduled before the other ones. There are two config options to configure: action_weights and parallelization.
>
> *Limitations*
>
> - This planner requires to have action_weights and parallelization configs tuned well.

Configuration

Strategy parameters are:

parameter	type	default Value description
maintenance_nod	String	The name of the compute node which need maintenance. Required.
backup_node	String	The name of the compute node which will backup the maintenance node. Optional.

Efficacy Indicator

None

Algorithm

For more information on the Host Maintenance Strategy please refer to: https://specs.openstack.org/openstack/watcher-specs/specs/queens/approved/cluster-maintenance-strategy.html

How to use it ?

```
$ openstack optimize audit create \
  -g cluster_maintaining -s host_maintenance \
  -p maintenance_node=compute01 \
  -p backup_node=compute02 \
  --auto-trigger
```

External Links

None.

4.4.4 Node Resource Consolidation Strategy

Synopsis

display name: `Node Resource Consolidation Strategy`

goal: `Server Consolidation`

> consolidating resources on nodes using server migration
>
> *Description*

This strategy checks the resource usages of compute nodes, if the used resources are less than total, it will try to migrate server to consolidate the use of resource.

Requirements

- You must have at least 2 compute nodes to run this strategy.
- Hardware: compute nodes should use the same physical CPUs/RAMs

Limitations

- This is a proof of concept that is not meant to be used in production
- It assume that live migrations are possible

Spec URL

http://specs.openstack.org/openstack/watcher-specs/specs/train/implemented/node-resource-consolidation.html

Requirements

None.

Metrics

None

Cluster data model

Default Watchers Compute cluster data model:

Nova cluster data model collector

The Nova cluster data model collector creates an in-memory representation of the resources exposed by the compute service.

Actions

Default Watchers actions:

action	description
`migration`	Migrates a server to a destination nova-compute host This action will allow you to migrate a server to another compute destination host. Migration type live can only be used for migrating active VMs. Migration type cold can be used for migrating non-active VMs as well active VMs, which will be shut down while migrating. The action schema is: `schema = Schema({ 'resource_id': str, # should be a UUID 'migration_type': str, # choices -> "live", "cold" 'destination_node': str, 'source_node': str, })` The *resource_id* is the UUID of the server to migrate. The *source_node* and *destination_node* parameters are respectively the source and the destination compute hostname (list of available compute hosts is returned by this command: `nova service-list --binary nova-compute`). **Note** Nova API version must be 2.56 or above if *destination_node* parameter is given.
`change_nova_service_state`	Disables or enables the nova-compute service, deployed on a host By using this action, you will be able to update the state of a nova-compute service. A disabled nova-compute service can not be selected by the nova scheduler for future deployment of server. The action schema is: `schema = Schema({ 'resource_id': str, 'state': str, 'disabled_reason': str, })` The *resource_id* references a nova-compute service name (list of available nova-compute services is returned by this command: `nova service-list --binary nova-compute`). The *state* value should either be *ONLINE* or *OFFLINE*. The *disabled_reason* references the reason why Watcher disables this nova-compute service. The value should be with *watcher_* prefix, such as *watcher_disabled*.

Planner

Default Watchers planner:

> Weight planner implementation
>
> This implementation builds actions with parents in accordance with weights. Set of actions having a higher weight will be scheduled before the other ones. There are two config options to configure: action_weights and parallelization.
>
> *Limitations*
>
> - This planner requires to have action_weights and parallelization configs tuned well.

Configuration

Strategy parameters are:

parameter	type	default Value	description
host_c	Strin		The way to select the server migration destination node, The value auto means that Nova schedular selects the destination node, and specify means the strategy specifies the destination.

Efficacy Indicator

None

Algorithm

For more information on the Node Resource Consolidation Strategy please refer to: https://specs.openstack.org/openstack/watcher-specs/specs/train/approved/node-resource-consolidation.html

How to use it ?

```
$ openstack optimize audittemplate create \
  at1 server_consolidation \
  --strategy node_resource_consolidation

$ openstack optimize audit create \
  -a at1 -p host_choice=auto
```

External Links

None.

4.4.5 Noisy neighbor

Synopsis

display name: `Noisy Neighbor`

goal: `noisy_neighbor`

Noisy Neighbor strategy using live migration

Description

This strategy can identify and migrate a Noisy Neighbor - a low priority VM that negatively affects performance of a high priority VM in terms of IPC by over utilizing Last Level Cache.

Requirements

To enable LLC metric, latest Intel server with CMT support is required.

Limitations

This is a proof of concept that is not meant to be used in production

Spec URL

http://specs.openstack.org/openstack/watcher-specs/specs/pike/implemented/noisy_neighbor_strategy.html

Requirements

Metrics

The *noisy_neighbor* strategy requires the following metrics:

metric	service name	plugins	comment
cpu_l3_cache	ceilometer	none	Intel CMT is required

Cluster data model

Default Watchers Compute cluster data model:

Nova cluster data model collector

The Nova cluster data model collector creates an in-memory representation of the resources exposed by the compute service.

Actions

Default Watchers actions:

action	description
`migration`	Migrates a server to a destination nova-compute host This action will allow you to migrate a server to another compute destination host. Migration type live can only be used for migrating active VMs. Migration type cold can be used for migrating non-active VMs as well active VMs, which will be shut down while migrating. The action schema is: `schema = Schema({ 'resource_id': str, # should be a UUID 'migration_type': str, # choices -> "live", "cold" 'destination_node': str, 'source_node': str, })` The *resource_id* is the UUID of the server to migrate. The *source_node* and *destination_node* parameters are respectively the source and the destination compute hostname (list of available compute hosts is returned by this command: `nova service-list --binary nova-compute`). **Note** Nova API version must be 2.56 or above if *destination_node* parameter is given.

Planner

Default Watchers planner:

Weight planner implementation

This implementation builds actions with parents in accordance with weights. Set of actions having a higher weight will be scheduled before the other ones. There are two config options to configure: action_weights and parallelization.

Limitations

- This planner requires to have action_weights and parallelization configs tuned well.

Configuration

Strategy parameter is:

parameter	type	default	Value description
cache_threshold	Number	35.0	Performance drop in L3_cache threshold for migration

Efficacy Indicator

None

Algorithm

For more information on the noisy neighbor strategy please refer to: http://specs.openstack.org/openstack/watcher-specs/specs/pike/implemented/noisy_neighbor_strategy.html

How to use it ?

```
$ openstack optimize audittemplate create \
  at1 noisy_neighbor --strategy noisy_neighbor

$ openstack optimize audit create -a at1 \
  -p cache_threshold=45.0
```

External Links

None

4.4.6 Outlet Temperature Based Strategy

Synopsis

display name: `Outlet temperature based strategy`

goal: `thermal_optimization`

> *Good Thermal Strategy*
>
> Towards to software defined infrastructure, the power and thermal intelligences is being adopted to optimize workload, which can help improve efficiency, reduce power, as well as to improve datacenter PUE and lower down operation cost in data center. Outlet (Exhaust Air) Temperature is one of the important thermal telemetries to measure thermal/workload status of server.
>
> This strategy makes decisions to migrate workloads to the hosts with good thermal condition (lowest outlet temperature) when the outlet temperature of source hosts reach a configurable threshold.

Requirements

This strategy has a dependency on the host having Intels Power Node Manager 3.0 or later enabled.

Metrics

The *outlet_temperature* strategy requires the following metrics:

metric	service name	plugins	comment
`hardware.ipmi.node.outlet_temperature`	ceilometer	IPMI	

Cluster data model

Default Watchers Compute cluster data model:

> Nova cluster data model collector
>
> The Nova cluster data model collector creates an in-memory representation of the resources exposed by the compute service.

Actions

Default Watchers actions:

action	description
`migration`	Migrates a server to a destination nova-compute host This action will allow you to migrate a server to another compute destination host. Migration type live can only be used for migrating active VMs. Migration type cold can be used for migrating non-active VMs as well active VMs, which will be shut down while migrating. The action schema is: `schema = Schema({ 'resource_id': str, # should be a UUID 'migration_type': str, # choices -> "live", "cold" 'destination_node': str, 'source_node': str, })` The *resource_id* is the UUID of the server to migrate. The *source_node* and *destination_node* parameters are respectively the source and the destination compute hostname (list of available compute hosts is returned by this command: `nova service-list --binary nova-compute`). **Note** Nova API version must be 2.56 or above if *destination_node* parameter is given.

Planner

Default Watchers planner:

Weight planner implementation

This implementation builds actions with parents in accordance with weights. Set of actions having a higher weight will be scheduled before the other ones. There are two config options to configure: action_weights and parallelization.

Limitations

- This planner requires to have action_weights and parallelization configs tuned well.

Configuration

Strategy parameter is:

parameter	type	default Value	description
threshol	Number	35.0	Temperature threshold for migration
period	Number	30	The time interval in seconds for getting statistic aggregation from metric data source

Efficacy Indicator

None

Algorithm

For more information on the Outlet Temperature Based Strategy please refer to: https://specs.openstack.org/openstack/watcher-specs/specs/mitaka/implemented/outlet-temperature-based-strategy.html

How to use it ?

```
$ openstack optimize audittemplate create \
  at1 thermal_optimization --strategy outlet_temperature

$ openstack optimize audit create -a at1 -p threshold=31.0
```

External Links

- Intel Power Node Manager 3.0

4.4.7 Saving Energy Strategy

Synopsis

display name: `Saving Energy Strategy`

goal: `saving_energy`

Saving Energy Strategy

Description

Saving Energy Strategy together with VM Workload Consolidation Strategy can perform the Dynamic Power Management (DPM) functionality, which tries to save power by dynamically consolidating workloads even further during periods of low resource utilization. Virtual machines are migrated onto fewer hosts and the unneeded hosts are powered off.

After consolidation, Saving Energy Strategy produces a solution of powering off/on according to the following detailed policy:

In this policy, a preset number(min_free_hosts_num) is given by user, and this min_free_hosts_num describes minimum free compute nodes that users expect to have, where free compute nodes refers to those nodes unused but still powered on.

If the actual number of unused nodes(in power-on state) is larger than the given number, randomly select the redundant nodes and power off them; If the actual number of unused nodes(in poweron state) is smaller than the given number and there are spare unused nodes(in poweroff state), randomly select some nodes(unused,poweroff) and power on them.

Requirements

In this policy, in order to calculate the min_free_hosts_num, users must provide two parameters:

- One parameter(min_free_hosts_num) is a constant int number. This number should be int type and larger than zero.
- The other parameter(free_used_percent) is a percentage number, which describes the quotient of min_free_hosts_num/nodes_with_VMs_num, where nodes_with_VMs_num is the number of nodes with VMs running on it. This parameter is used to calculate a dynamic min_free_hosts_num. The nodes with VMs refer to those nodes with VMs running on it.

Then choose the larger one as the final min_free_hosts_num.

Limitations

- at least 2 physical compute hosts

Spec URL

http://specs.openstack.org/openstack/watcher-specs/specs/pike/implemented/energy-saving-strategy.html

Requirements

This feature will use Ironic to do the power on/off actions, therefore this feature requires that the ironic component is configured. And the compute node should be managed by Ironic.

Ironic installation: https://docs.openstack.org/ironic/latest/install/index.html

Cluster data model

Default Watchers Compute cluster data model:

Nova cluster data model collector

The Nova cluster data model collector creates an in-memory representation of the resources exposed by the compute service.

Actions

action	description
change_node_power_state	Compute node power on/off By using this action, you will be able to on/off the power of a compute node. The action schema is: `schema = Schema({ 'resource_id': str, 'state': str, })` The *resource_id* references a baremetal node id (list of available ironic nodes is returned by this command: `ironic node-list`). The *state* value should either be *on* or *off*.

Planner

Default Watchers planner:

> Weight planner implementation
>
> This implementation builds actions with parents in accordance with weights. Set of actions having a higher weight will be scheduled before the other ones. There are two config options to configure: action_weights and parallelization.
>
> *Limitations*
>
> - This planner requires to have action_weights and parallelization configs tuned well.

Configuration

Strategy parameter is:

parameter	type	default Value	description
free_used_pe	Number	10.0	a rational number, which describes the the quotient of min_free_hosts_num/nodes_with_VMs_num
min_free_hos	Int	1	an int number describes minimum free compute nodes

Efficacy Indicator

None

Algorithm

For more information on the Energy Saving Strategy please refer to: http://specs.openstack.org/openstack/watcher-specs/specs/pike/implemented/energy-saving-strategy.html

How to use it ?

step1: Add compute nodes info into ironic node management

```
$ ironic node-create  -d pxe_ipmitool -i ipmi_address=10.43.200.184 \
  ipmi_username=root  -i ipmi_password=nomoresecret -e compute_node_id=3
```

step 2: Create audit to do optimization

```
$ openstack optimize audittemplate create \
  saving_energy_template1 saving_energy --strategy saving_energy

$ openstack optimize audit create -a saving_energy_audit1 \
  -p free_used_percent=20.0
```

External Links

None

4.4.8 Storage capacity balance

Synopsis

display name: `Storage Capacity Balance Strategy`

goal: `workload_balancing`

Storage capacity balance using cinder volume migration

Description

This strategy migrates volumes based on the workload of the cinder pools. It makes decision to migrate a volume whenever a pools used utilization % is higher than the specified threshold. The volume to be moved should make the pool close to average workload of all cinder pools.

Requirements

- You must have at least 2 cinder volume pools to run this strategy.

Limitations

- Volume migration depends on the storage device. It may take a long time.

Spec URL

http://specs.openstack.org/openstack/watcher-specs/specs/queens/implemented/storage-capacity-balance.html

Requirements

Metrics

None

Cluster data model

Storage cluster data model is required:

> Cinder cluster data model collector
>
> The Cinder cluster data model collector creates an in-memory representation of the resources exposed by the storage service.

Actions

Default Watchers actions:

action	description
`volume_migrate`	Migrates a volume to destination node or type By using this action, you will be able to migrate cinder volume. Migration type swap can only be used for migrating attached volume. Migration type migrate can be used for migrating detached volume to the pool of same volume type. Migration type retype can be used for changing volume type of detached volume. The action schema is: `schema = Schema({ 'resource_id': str, # should be a UUID 'migration_type': str, # choices -> "swap", "migrate","retype" 'destination_node': str, 'destination_type': str, })` The *resource_id* is the UUID of cinder volume to migrate. The *destination_node* is the destination block storage pool name. (list of available pools are returned by this command: `cinder get-pools`) which is mandatory for migrating detached volume to the one with same volume type. The *destination_type* is the destination block storage type name. (list of available types are returned by this command: `cinder type-list`) which is mandatory for migrating detached volume or swapping attached volume to the one with different volume type.

Planner

Default Watchers planner:

> Weight planner implementation
>
> This implementation builds actions with parents in accordance with weights. Set of actions having a higher weight will be scheduled before the other ones. There are two config options

to configure: action_weights and parallelization.

Limitations

- This planner requires to have action_weights and parallelization configs tuned well.

Configuration

Strategy parameter is:

parameter	type	default	Value description
`volume_threshold`	Number	80.0	Volume threshold for capacity balance

Efficacy Indicator

None

Algorithm

For more information on the storage capacity balance strategy please refer to: http://specs.openstack.org/openstack/watcher-specs/specs/queens/implemented/storage-capacity-balance.html

How to use it ?

```
$ openstack optimize audittemplate create \
  at1 workload_balancing --strategy storage_capacity_balance

$ openstack optimize audit create -a at1 \
  -p volume_threshold=85.0
```

External Links

None

4.4.9 Uniform Airflow Migration Strategy

Synopsis

display name: `Uniform airflow migration strategy`

goal: `airflow_optimization`

[PoC]Uniform Airflow using live migration

Description

It is a migration strategy based on the airflow of physical servers. It generates solutions to move VM whenever a servers airflow is higher than the specified threshold.

Requirements

- Hardware: compute node with NodeManager 3.0 support

- Software: Ceilometer component ceilometer-agent-compute running in each compute node, and Ceilometer API can report such telemetry airflow, system power, inlet temperature successfully.
- You must have at least 2 physical compute nodes to run this strategy

Limitations

- This is a proof of concept that is not meant to be used in production.
- We cannot forecast how many servers should be migrated. This is the reason why we only plan a single virtual machine migration at a time. So its better to use this algorithm with *CONTINUOUS* audits.
- It assumes that live migrations are possible.

Requirements

This strategy has a dependency on the server having Intels Power Node Manager 3.0 or later enabled.

Metrics

The *uniform_airflow* strategy requires the following metrics:

metric	service name	plugins	comment
`hardware.ipmi.node.airflow`	ceilometer	IPMI	
`hardware.ipmi.node.temperature`	ceilometer	IPMI	
`hardware.ipmi.node.power`	ceilometer	IPMI	

Cluster data model

Default Watchers Compute cluster data model:

> Nova cluster data model collector
>
> The Nova cluster data model collector creates an in-memory representation of the resources exposed by the compute service.

Actions

Default Watchers actions:

action	description
`migration`	Migrates a server to a destination nova-compute host This action will allow you to migrate a server to another compute destination host. Migration type live can only be used for migrating active VMs. Migration type cold can be used for migrating non-active VMs as well active VMs, which will be shut down while migrating. The action schema is: `schema = Schema({ 'resource_id': str, # should be a UUID 'migration_type': str, # choices -> "live", "cold" 'destination_node': str, 'source_node': str, })` The *resource_id* is the UUID of the server to migrate. The *source_node* and *destination_node* parameters are respectively the source and the destination compute hostname (list of available compute hosts is returned by this command: `nova service-list --binary nova-compute`). **Note** Nova API version must be 2.56 or above if *destination_node* parameter is given.

Planner

Default Watchers planner:

Weight planner implementation

This implementation builds actions with parents in accordance with weights. Set of actions having a higher weight will be scheduled before the other ones. There are two config options to configure: action_weights and parallelization.

Limitations

- This planner requires to have action_weights and parallelization configs tuned well.

Configuration

Strategy parameters are:

parameter	type	default Value	description
`threshold_airflow`	Number	400.0	Airflow threshold for migration Unit is 0.1CFM
`threshold_inlet_t`	Number	28.0	Inlet temperature threshold for migration decision
`threshold_power`	Number	350.0	System power threshold for migration decision
`period`	Number	300	Aggregate time period of ceilometer

Efficacy Indicator

None

Algorithm

For more information on the Uniform Airflow Migration Strategy please refer to: https://specs.openstack.org/openstack/watcher-specs/specs/newton/implemented/uniform-airflow-migration-strategy.html

How to use it ?

```
$ openstack optimize audittemplate create \
  at1 airflow_optimization --strategy uniform_airflow

$ openstack optimize audit create -a at1 -p threshold_airflow=410 \
       -p threshold_inlet_t=29.0 -p threshold_power=355.0 -p period=310
```

External Links

- Intel Power Node Manager 3.0

4.4.10 VM Workload Consolidation Strategy

Synopsis

display name: `VM Workload Consolidation Strategy`

goal: `vm_consolidation`

VM Workload Consolidation Strategy

A load consolidation strategy based on heuristic first-fit algorithm which focuses on measured CPU utilization and tries to minimize hosts which have too much or too little load respecting resource capacity constraints.

This strategy produces a solution resulting in more efficient utilization of cluster resources using following four phases:

- Offload phase - handling over-utilized resources

- Consolidation phase - handling under-utilized resources
- Solution optimization - reducing number of migrations
- Disability of unused compute nodes

A capacity coefficients (cc) might be used to adjust optimization thresholds. Different resources may require different coefficient values as well as setting up different coefficient values in both phases may lead to more efficient consolidation in the end. If the cc equals 1 the full resource capacity may be used, cc values lower than 1 will lead to resource under utilization and values higher than 1 will lead to resource overbooking. e.g. If targeted utilization is 80 percent of a compute node capacity, the coefficient in the consolidation phase will be 0.8, but may any lower value in the offloading phase. The lower it gets the cluster will appear more released (distributed) for the following consolidation phase.

As this strategy leverages VM live migration to move the load from one compute node to another, this feature needs to be set up correctly on all compute nodes within the cluster. This strategy assumes it is possible to live migrate any VM from an active compute node to any other active compute node.

Requirements

Metrics

The *vm_workload_consolidation* strategy requires the following metrics:

metric	service name	plugins	comment
`cpu`	ceilometer	none	
`memory.resident`	ceilometer	none	
`memory`	ceilometer	none	
`disk.root.size`	ceilometer	none	
`compute.node.cpu.percent`	ceilometer	none	(optional) need to set the `compute_monitors` option to `cpu.virt_driver` in the nova.conf.
`hardware.memory.used`	ceilometer	SNMP	(optional)

Cluster data model

Default Watchers Compute cluster data model:

Nova cluster data model collector

The Nova cluster data model collector creates an in-memory representation of the resources exposed by the compute service.

Actions

Default Watchers actions:

action	description
`migration`	Migrates a server to a destination nova-compute host This action will allow you to migrate a server to another compute destination host. Migration type live can only be used for migrating active VMs. Migration type cold can be used for migrating non-active VMs as well active VMs, which will be shut down while migrating. The action schema is: `schema = Schema({ 'resource_id': str, # should be a UUID 'migration_type': str, # choices -> "live", "cold" 'destination_node': str, 'source_node': str, })` The *resource_id* is the UUID of the server to migrate. The *source_node* and *destination_node* parameters are respectively the source and the destination compute hostname (list of available compute hosts is returned by this command: `nova service-list --binary nova-compute`). **Note** Nova API version must be 2.56 or above if *destination_node* parameter is given.
`change_nova_service_state`	Disables or enables the nova-compute service, deployed on a host By using this action, you will be able to update the state of a nova-compute service. A disabled nova-compute service can not be selected by the nova scheduler for future deployment of server. The action schema is: `schema = Schema({ 'resource_id': str, 'state': str, 'disabled_reason': str, })` The *resource_id* references a nova-compute service name (list of available nova-compute services is returned by this command: `nova service-list --binary nova-compute`). The *state* value should either be *ONLINE* or *OFFLINE*. The *disabled_reason* references the reason why Watcher disables this nova-compute service. The value should be with *watcher_* prefix, such as *watcher_disabled*.

Planner

Default Watchers planner:

> Weight planner implementation
>
> This implementation builds actions with parents in accordance with weights. Set of actions having a higher weight will be scheduled before the other ones. There are two config options to configure: action_weights and parallelization.
>
> *Limitations*
>
> - This planner requires to have action_weights and parallelization configs tuned well.

Configuration

Strategy parameter is:

parameter	type	default Value	description
`period`	Number	3600	The time interval in seconds for getting statistic aggregation from metric data source

Efficacy Indicator

```
[{'name': 'released_nodes_ratio', 'description': 'Ratio of released compute␣
↪nodes divided by the total number of enabled compute nodes.', 'unit': '%',
↪'value': 0}]
```

Algorithm

For more information on the VM Workload consolidation strategy please refer to: https://specs.openstack.org/openstack/watcher-specs/specs/mitaka/implemented/zhaw-load-consolidation.html

How to use it ?

```
$ openstack optimize audittemplate create \
  at1 server_consolidation --strategy vm_workload_consolidation

$ openstack optimize audit create -a at1
```

External Links

Spec URL https://specs.openstack.org/openstack/watcher-specs/specs/mitaka/implemented/zhaw-load-consolidation.html

4.4.11 Watcher Overload standard deviation algorithm

Synopsis

display name: `Workload stabilization`

goal: `workload_balancing`

Workload Stabilization control using live migration

This is workload stabilization strategy based on standard deviation algorithm. The goal is to determine if there is an overload in a cluster and respond to it by migrating VMs to stabilize the cluster.

This strategy has been tested in a small (32 nodes) cluster.

It assumes that live migrations are possible in your cluster.

Requirements

Metrics

The *workload_stabilization* strategy requires the following metrics:

metric	service name	plugins	comment
`compute.node.cpu.percent`	ceilometer	none	need to set the `compute_monitors` option to `cpu.virt_driver` in the nova.conf.
`hardware.memory.used`	ceilometer	SNMP	
`cpu`	ceilometer	none	
`instance_ram_usag`	ceilometer	none	

Cluster data model

Default Watchers Compute cluster data model:

Nova cluster data model collector

The Nova cluster data model collector creates an in-memory representation of the resources exposed by the compute service.

Actions

Default Watchers actions:

action	description
`migration`	Migrates a server to a destination nova-compute host This action will allow you to migrate a server to another compute destination host. Migration type live can only be used for migrating active VMs. Migration type cold can be used for migrating non-active VMs as well active VMs, which will be shut down while migrating. The action schema is: `schema = Schema({ 'resource_id': str, # should be a UUID 'migration_type': str, # choices -> "live", "cold" 'destination_node': str, 'source_node': str, })` The *resource_id* is the UUID of the server to migrate. The *source_node* and *destination_node* parameters are respectively the source and the destination compute hostname (list of available compute hosts is returned by this command: `nova service-list --binary nova-compute`). **Note** Nova API version must be 2.56 or above if *destination_node* parameter is given.

Planner

Default Watchers planner:

Weight planner implementation

This implementation builds actions with parents in accordance with weights. Set of actions having a higher weight will be scheduled before the other ones. There are two config options to configure: action_weights and parallelization.

Limitations

- This planner requires to have action_weights and parallelization configs tuned well.

Configuration

Strategy parameters are:

parameter	type	default Value	description
metri	array	[instance_cpu_usage, instance_ram_usage]	Metrics used as rates of cluster loads.
thres	object	{instance_cpu_usage: 0.2, instance_ram_usage: 0.2}	Dict where key is a metric and value is a trigger value.
weigh	object	{instance_cpu_usage_weight 1.0, instance_ram_usage_weigh 1.0}	These weights used to calculate common standard deviation. Name of weight contains meter name and _weight suffix.
insta	object	{instance_cpu_usage: compute.node.cpu.percent, instance_ram_usage: hardware.memory.used}	Mapping to get hardware statistics using instance metrics.
host_	string	retry	Method of hosts choice. There are cycle, retry and fullsearch methods. Cycle will iterate hosts in cycle. Retry will get some hosts random (count defined in retry_count option). Fullsearch will return each host from list.
retry	number	1	Count of random returned hosts.
perio	object	{instance: 720, node: 600}	These periods are used to get statistic aggregation for instance and host metrics. The period is simply a repeating interval of time into which the samples are grouped for aggregation. Watcher uses only the last period of all received ones.

Efficacy Indicator

```
[{'name': 'released_nodes_ratio', 'description': 'Ratio of released compute␣
↪nodes divided by the total number of enabled compute nodes.', 'unit': '%',
↪'value': 0}]
```

Algorithm

You can find description of overload algorithm and role of standard deviation here: https://specs.openstack.org/openstack/watcher-specs/specs/newton/implemented/sd-strategy.html

How to use it ?

```
$ openstack optimize audittemplate create \
  at1 workload_balancing --strategy workload_stabilization

$ openstack optimize audit create -a at1 \
  -p thresholds='{"instance_ram_usage": 0.05}' \
  -p metrics='["instance_ram_usage"]'
```

External Links

- Watcher Overload standard deviation algorithm spec

4.4.12 Workload Balance Migration Strategy

Synopsis

display name: `Workload Balance Migration Strategy`

goal: `workload_balancing`

[PoC]Workload balance using live migration

Description

It is a migration strategy based on the VM workload of physical servers. It generates solutions to move a workload whenever a servers CPU or RAM utilization % is higher than the specified threshold. The VM to be moved should make the host close to average workload of all compute nodes.

Requirements

- Hardware: compute node should use the same physical CPUs/RAMs
- Software: Ceilometer component ceilometer-agent-compute running in each compute node, and Ceilometer API can report such telemetry instance_cpu_usage and instance_ram_usage successfully.
- You must have at least 2 physical compute nodes to run this strategy.

Limitations

- This is a proof of concept that is not meant to be used in production
- We cannot forecast how many servers should be migrated. This is the reason why we only plan a single virtual machine migration at a time. So its better to use this algorithm with *CONTINUOUS* audits.
- It assume that live migrations are possible

Requirements

None.

Metrics

The *workload_balance* strategy requires the following metrics:

metric	service name	plugins	comment
`cpu`	ceilometer	none	
`memory.resident`	ceilometer	none	

Cluster data model

Default Watchers Compute cluster data model:

> Nova cluster data model collector
>
> The Nova cluster data model collector creates an in-memory representation of the resources exposed by the compute service.

Actions

Default Watchers actions:

action	description
`migration`	Migrates a server to a destination nova-compute host This action will allow you to migrate a server to another compute destination host. Migration type live can only be used for migrating active VMs. Migration type cold can be used for migrating non-active VMs as well active VMs, which will be shut down while migrating. The action schema is: `schema = Schema({ 'resource_id': str, # should be a UUID 'migration_type': str, # choices -> "live", "cold" 'destination_node': str, 'source_node': str, })` The *resource_id* is the UUID of the server to migrate. The *source_node* and *destination_node* parameters are respectively the source and the destination compute hostname (list of available compute hosts is returned by this command: `nova service-list --binary nova-compute`). **Note** Nova API version must be 2.56 or above if *destination_node* parameter is given.

Planner

Default Watchers planner:

Weight planner implementation

This implementation builds actions with parents in accordance with weights. Set of actions having a higher weight will be scheduled before the other ones. There are two config options to configure: action_weights and parallelization.

Limitations

- This planner requires to have action_weights and parallelization configs tuned well.

Configuration

Strategy parameters are:

parameter	type	default Value	description
metrics	String	instance_cpu_us	Workload balance base on cpu or ram utilization. Choices: [instance_cpu_usage, instance_ram_usage]
threshold	Number	25.0	Workload threshold for migration
period	Number	300	Aggregate time period of ceilometer

Efficacy Indicator

None

Algorithm

For more information on the Workload Balance Migration Strategy please refer to: https://specs.openstack.org/openstack/watcher-specs/specs/mitaka/implemented/workload-balance-migration-strategy.html

How to use it ?

```
$ openstack optimize audittemplate create \
  at1 workload_balancing --strategy workload_balance

$ openstack optimize audit create -a at1 -p threshold=26.0 \
       -p period=310 -p metrics=instance_cpu_usage
```

External Links

None.

4.4.13 Zone migration

Synopsis

display name: `Zone migration`

goal: `hardware_maintenance`

> Zone migration using instance and volume migration
>
> This is zone migration strategy to migrate many instances and volumes efficiently with minimum downtime for hardware maintenance.

Requirements

Metrics

None

Cluster data model

Default Watchers Compute cluster data model:

> Nova cluster data model collector
>
> The Nova cluster data model collector creates an in-memory representation of the resources exposed by the compute service.

Storage cluster data model is also required:

> Cinder cluster data model collector
>
> The Cinder cluster data model collector creates an in-memory representation of the resources exposed by the storage service.

Actions

Default Watchers actions:

action	description
migrate	Migrates a server to a destination nova-compute host This action will allow you to migrate a server to another compute destination host. Migration type live can only be used for migrating active VMs. Migration type cold can be used for migrating non-active VMs as well active VMs, which will be shut down while migrating. The action schema is: `schema = Schema({ 'resource_id': str, # should be a UUID 'migration_type': str, # choices -> "live", "cold" 'destination_node': str, 'source_node': str, })` The *resource_id* is the UUID of the server to migrate. The *source_node* and *destination_node* parameters are respectively the source and the destination compute hostname (list of available compute hosts is returned by this command: `nova service-list --binary nova-compute`). **Note** Nova API version must be 2.56 or above if *destination_node* parameter is given.
volume_migrate	Migrates a volume to destination node or type By using this action, you will be able to migrate cinder volume. Migration type swap can only be used for migrating attached volume. Migration type migrate can be used for migrating detached volume to the pool of same volume type. Migration type retype can be used for changing volume type of detached volume. The action schema is: `schema = Schema({ 'resource_id': str, # should be a UUID 'migration_type': str, # choices -> "swap", "migrate", "retype" 'destination_node': str, 'destination_type': str, })` The *resource_id* is the UUID of cinder volume to migrate. The *destination_node* is the destination block storage pool name.

Planner

Default Watchers planner:

Weight planner implementation

This implementation builds actions with parents in accordance with weights. Set of actions having a higher weight will be scheduled before the other ones. There are two config options to configure: action_weights and parallelization.

Limitations

- This planner requires to have action_weights and parallelization configs tuned well.

Configuration

Strategy parameters are:

parameter	type	default Value	description
`compute_node`	array	None	Compute nodes to migrate.
`storage_pool`	array	None	Storage pools to migrate.
`parallel_tot`	integer	6	The number of actions to be run in parallel in total.
`parallel_per`	integer	2	The number of actions to be run in parallel per compute node.
`parallel_per`	integer	2	The number of actions to be run in parallel per storage pool.
`priority`	object	None	List prioritizes instances and volumes.
`with_attache`	boole	False	False: Instances will migrate after all volumes migrate. True: An instance will migrate after the attached volumes migrate.

The elements of compute_nodes array are:

parameter	type	default Value	description
`src_node`	string	None	Compute node from which instances migrate(mandatory).
`dst_node`	string	None	Compute node to which instances migrate.

The elements of storage_pools array are:

parameter	type	default Value	description
src_pool	string	None	Storage pool from which volumes migrate(mandatory).
dst_pool	string	None	Storage pool to which volumes migrate.
src_type	string	None	Source volume type(mandatory).
dst_type	string	None	Destination volume type (mandatory).

The elements of priority object are:

parameter	type	default Value	description
project	array	None	Project names.
compute_node	array	None	Compute node names.
storage_pool	array	None	Storage pool names.
compute	enum	None	Instance attributes. [vcpu_num, mem_size, disk_size, created_at]
storage	enum	None	Volume attributes. [size, created_at]

Efficacy Indicator

```
[{'name': 'live_instance_migrate_ratio', 'description': 'Ratio of actual live␣
↪migrated instances to planned live migrate instances.', 'unit': '%', 'value
↪': 0}, {'name': 'cold_instance_migrate_ratio', 'description': 'Ratio of␣
↪actual cold migrated instances to planned cold migrate instances.', 'unit':
↪'%', 'value': 0}, {'name': 'volume_migrate_ratio', 'description': 'Ratio of␣
↪actual detached volumes migrated to planned detached volumes migrate.',
↪'unit': '%', 'value': 0}, {'name': 'volume_update_ratio', 'description':
↪'Ratio of actual attached volumes migrated to planned attached volumes␣
↪migrate.', 'unit': '%', 'value': 0}]
```

Algorithm

For more information on the zone migration strategy please refer to: http://specs.openstack.org/openstack/watcher-specs/specs/queens/implemented/zone-migration-strategy.html

How to use it ?

```
$ openstack optimize audittemplate create \
  at1 hardware_maintenance --strategy zone_migration

$ openstack optimize audit create -a at1 \
  -p compute_nodes='[{"src_node": "s01", "dst_node": "d01"}]'
```

External Links

None

4.5 Datasources

4.5.1 Grafana datasource

Synopsis

Grafana can interface with many different types of storage backends that Grafana calls datasources. Since the term datasources causes significant confusion by overlapping definitions used in Watcher these **datasources are called projects instead**. Some examples of supported projects are InfluxDB or Elasticsearch while others might be more familiar such as Monasca or Gnocchi. The Grafana datasource provides the functionality to retrieve metrics from Grafana for different projects. This functionality is achieved by using the proxy interface exposed in Grafana to communicate with Grafana projects directly.

Background

Since queries to retrieve metrics from Grafana are proxied to the project the format of these queries will change significantly depending on the type of project. The structure of the projects themselves will also change significantly as they are structured by users and administrators. For instance, some developers might decide to store metrics about compute_nodes in MySQL and use the UUID as primary key while others use InfluxDB and use the hostname as primary key. Furthermore, datasources in Watcher should return metrics in specific units strictly defined in the baseclass depending on how the units are stored in the projects they might require conversion before being returned. The flexible configuration parameters of the Grafana datasource allow to specify exactly how the deployment is configured and this will enable to correct retrieval of metrics and with the correct units.

Requirements

The use of the Grafana datasource requires a reachable Grafana endpoint and an authentication token for access to the desired projects. The projects behind Grafana will need to contain the metrics for compute_nodes or instances and these need to be identifiable by an attribute of the Watcher datamodel for instance hostname or UUID.

Limitations

- Only the InfluxDB project is currently supported[1].
- All metrics must be retrieved from the same Grafana endpoint (same URL).
- All metrics must be retrieved with the same authentication token.

Configuration

Several steps are required in order to use the Grafana datasource, Most steps are related configuring Watcher to match the deployed Grafana setup such as queries proxied to the project or the type of project for any given metric. Most of the configuration can either be supplied via the traditional configuration file or in a special yaml file.

[1] A base class for projects is available and easily extensible.

token

First step is to generate an access token with access to the required projects. This can be done from the api or from the web interface. Tokens generated from the web interface will have the same access to projects as the user that created them while using the cli allows to generate a key for a specific role.The token will only be displayed once so store it well. This token will go into the configuration file later and this parameter can not be placed in the yaml.

base_url

Next step is supplying the base url of the Grafana endpoint. The base url parameter will need to specify the type of http protocol and the use of plain text http is strongly discouraged due to the transmission of the access token. Additionally the path to the proxy interface needs to be supplied as well in case Grafana is placed in a sub directory of the web server. An example would be: `https://mygrafana.org/api/datasource/proxy/` were `/api/datasource/proxy` is the default path without any subdirectories. Likewise, this parameter can not be placed in the yaml.

To prevent many errors from occurring and potentially filing the logs files it is advised to specify the desired datasource in the configuration as it would prevent the datasource manager from having to iterate and try possible datasources with the launch of each audit. To do this specify `datasources` in the `[watcher_datasources]` group.

The current configuration that is required to be placed in the traditional configuration file would look like the following:

```
[grafana_client]
token = 0JLbF0oB4R3Q2Fl337Gh4Df5VN12D3adBE3f==
base_url = https://mygranfa.org/api/datasource/proxy

[watcher_datasources]
datasources = grafana
```

metric parameters

The last five remaining configuration parameters can all be placed both in the traditional configuration file or in the yaml, however, it is not advised to mix and match but in the case it does occur the yaml would override the settings from the traditional configuration file. All five of these parameters are dictionaries mapping specific metrics to a configuration parameter. For instance the `project_id_map` will specify the specific project id in Grafana to be used. The parameters are named as follow:

- project_id_map
- database_map
- translator_map
- attribute_map
- query_map

These five parameters are named differently if configured using the yaml configuration file. The parameters are named as follows and are in identical order as to the list of the traditional configuration file:

- project
- db

- translator
- attribute
- query

When specified in the yaml the parameters are no longer dictionaries instead each parameter needs to be defined per metric as sub-parameters. Examples of these parameters configured for both the yaml and traditional configuration are described at the end of this document.

project_id

The project ids can only be determined by someone with the admin role in Grafana as that role is required to open the list of projects. The list of projects can be found on `/datasources` in the web interface but unfortunately it does not immediately display the project id. To display the id one can best hover the mouse over the projects and the url will show the project ids for example `/datasources/edit/7563`. Alternatively the entire list of projects can be retrieved using the REST api. To easily make requests to the REST api a tool such as Postman can be used.

database

The database is the parameter for the schema / database that is actually defined in the project. For instance, if the project would be based on MySQL this is were the name of schema used within the MySQL server would be specified. For many different projects it is possible to list all the databases currently available. Tools like Postman can be used to list all the available databases per project. For InfluxDB based projects this would be with the following path and query, however be sure to construct these request in Postman as the header needs to contain the authorization token:

```
https://URL.DOMAIN/api/datasources/proxy/PROJECT_ID/query?q=SHOW%20DATABASES
```

translator

Each translator is for a specific type of project will have a uniquely identifiable name and the baseclass allows to easily support new types of projects such as elasticsearch or prometheus. Currently only InfluxDB based projects are supported as a result the only valid value for this parameter is ‘ influxdb‘.

attribute

The attribute parameter specifies which attribute to use from Watchers data model in order to construct the query. The available attributes differ per type of object in the data model but the following table shows the attributes for ComputeNodes, Instances and IronicNodes.

ComputeNode	Instance	IronicNode
uuid	uuid	uuid
id	name	human_id
hostname	project_id	power_state
status	watcher_exclude	maintenance
disabled_reason	locked	maintenance_reason
state	metadata	extra
memory	state	
disk	memory	
disk_capacity	disk	
vcpus	disk_capacity	
	vcpus	

Many if not all of these attributes map to attributes of the objects that are fetched from clients such as Nova. To see how these attributes are put into the data model the following source files can be analyzed for Nova and Ironic.

query

The query is the single most important parameter it will be passed to the project and should return the desired metric for the specific host and return the value in the correct unit. The units for all available metrics are documented in the datasource baseclass. This might mean the query specified in this parameter is responsible for converting the unit. The following query demonstrates how such a conversion could be achieved and demonstrates the conversion from bytes to megabytes.

```
SELECT value/1000000 FROM memory...
```

Queries will be formatted using the .format string method within Python. This format will currently have give attributes exposed to it labeled `{0}` through `{4}`. Every occurrence of these characters within the string will be replaced with the specific attribute.

{0}
: is the aggregate typically `mean`, `min`, `max` but `count` is also supported.

{1}
: is the attribute as specified in the attribute parameter.

{2}
: is the period of time to aggregate data over in seconds.

{3}
: is the granularity or the interval between data points in seconds.

{4}
: is translator specific and in the case of InfluxDB it will be used for retention_periods.

InfluxDB

Constructing the queries or rather anticipating how the results should look to be correctly interpreted by Watcher can be a challenge. The following json example demonstrates how what the result should look like and the query used to get this result.

```
{
"results": [
    {
        "statement_id": 0,
        "series": [
            {
                "name": "vmstats",
                "tags": {
                    "host": "autoserver01"
                },
                "columns": [
                    "time",
                    "mean"
                ],
                "values": [
                    [
                        1560848284284,
                        7680000
                    ]
                ]
            }
        ]
    }
]
}
```

```
SELECT {0}("{0}_value") FROM "vmstats" WHERE host =~ /^{1}$/ AND
"type_instance" =~ /^mem$/ AND time >= now() - {2}s GROUP BY host
```

Example configuration

The example configurations will show both how to achieve the entire configuration in the config file or use a combination of the regular file and yaml. Using yaml to define all the parameters for each metric is recommended since it has better human readability and supports mutli-line option definitions.

Configuration file

It is important to note that the line breaks shown in between assignments of parameters can not be used in the actual configuration and these are simply here for readability reasons.

```
[grafana_client]
# Authentication token to gain access (string value)
# Note: This option can be changed without restarting.
token = eyJrIjoiT0tTcG1pUlY2RnVKZTFVaDFsNFZXdE9ZWmNrMkZYbk==

# first part of the url (including https:// or http://) up until project id
# part. Example: https://secure.org/api/datasource/proxy/ (string value)
# Note: This option can be changed without restarting.
base_url = https://monitoring-grafana.com/api/datasources/proxy/
```

(continues on next page)

(continued from previous page)

```
# Project id as in url (integer value)
# Note: This option can be changed without restarting.
project_id_map = host_cpu_usage:1337,host_ram_usage:6969,
instance_cpu_usage:1337,instance_ram_usage:9696

# Mapping of grafana databases to datasource metrics. (dict value)
# Note: This option can be changed without restarting.
database_map = host_cpu_usage:monit_production,
host_ram_usage:monit_production,instance_cpu_usage:prod_cloud,
instance_ram_usage:prod_cloud

translator_map = host_cpu_usage:influxdb,host_ram_usage:influxdb,
instance_cpu_usage:influxdb,instance_ram_usage:influxdb

attribute_map = host_cpu_usage:hostname,host_ram_usage:hostname,
instance_cpu_usage:name,instance_ram_usage:name

query_map = host_cpu_usage:SELECT 100-{0}("{0}_value") FROM {4}.cpu WHERE
("host" =~ /^{1}$/ AND "type_instance" =~/^idle$/ AND time > now()-{2}s),
host_ram_usage:SELECT {0}("{0}_value")/1000000 FROM {4}.memory WHERE
 ("host" =~ /^{1}$/) AND "type_instance" =~ /^used$/ AND time >= now()-{2}s
 GROUP BY "type_instance",instance_cpu_usage:SELECT {0}("{0}_value") FROM
 "vmstats" WHERE host =~ /^{1}$/ AND "type_instance" =~ /^cpu$/ AND time >=
 now() - {2}s GROUP BY host,instance_ram_usage:SELECT {0}("{0}_value") FROM
 "vmstats" WHERE host =~ /^{1}$/ AND "type_instance" =~ /^mem$/ AND time >=
 now() - {2}s GROUP BY host

[grafana_translators]

retention_periods = one_week:10080,one_month:302400,five_years:525600

[watcher_datasources]
datasources = grafana
```

yaml

When using the yaml configuration file some parameters still need to be defined using the regular configuration such as the path for the yaml file these parameters are detailed below:

```
[grafana_client]
token = eyJrIjoiT0tTcG1pUlY2RnVKZTFVaDFsNFZXdE9ZWmNrMkZYbk==

base_url = https://monitoring-grafana.com/api/datasources/proxy/

[watcher_datasources]
datasources = grafana

[watcher_decision_engine]
metric_map_path = /etc/watcher/metric_map.yaml
```

Using the yaml allows to more effectively define the parameters per metric with greater human readability due to the availability of multi line options. These multi line options are demonstrated in the query parameters.

```
grafana:
  host_cpu_usage:
    project: 1337
    db: monit_production
    translator: influxdb
    attribute: hostname
    query: >
        SELECT 100-{0}("{0}_value") FROM {4}.cpu
        WHERE ("host" =~ /^{1}$/ AND "type_instance" =~/^idle$/ AND
        time > now()-{2}s)
  host_ram_usage:
    project: 6969
    db: monit_production
    translator: influxdb
    attribute: hostname
    query: >
        SELECT {0}("{0}_value")/1000000 FROM {4}.memory WHERE
         ("host" =~ /^{1}$/) AND "type_instance" =~ /^used$/ AND time >=
         now()-{2}s GROUP BY "type_instance"
  instance_cpu_usage:
    project: 1337
    db: prod_cloud
    translator: influxdb
    attribute: name
    query: >
        SELECT {0}("{0}_value") FROM
         "vmstats" WHERE host =~ /^{1}$/ AND "type_instance" =~ /^cpu$/ AND
         time >= now() - {2}s GROUP BY host
  instance_ram_usage:
    project: 9696
    db: prod_cloud
    translator: influxdb
    attribute: name
    query: >
        SELECT {0}("{0}_value") FROM
         "vmstats" WHERE host =~ /^{1}$/ AND "type_instance" =~ /^mem$/ AND
         time >= now() - {2}s GROUP BY host
```

External Links

- List of Grafana datasources

4.5.2 Prometheus datasource

Synopsis

The Prometheus datasource allows Watcher to use a Prometheus server as the source for collected metrics used by the Watcher decision engine. At minimum deployers must configure the `host` and `port` at which

the Prometheus server is listening.

Requirements

It is required that Prometheus metrics contain a label to identify the hostname of the exporter from which the metric was collected. This is used to match against the Watcher cluster model `ComputeNode.hostname`. The default for this label is `fqdn` and in the prometheus scrape configs would look like:

```
scrape_configs:
- job_name: node
  static_configs:
  - targets: ['10.1.2.3:9100']
    labels:
      fqdn: "testbox.controlplane.domain"
```

This default can be overridden when a deployer uses a different label to identify the exporter host (for example `hostname` or `host`, or any other label, as long as it identifies the host).

Internally this label is used in creating `fqdn_instance_labels`, containing the list of values assigned to the the label in the Prometheus targets. The elements of the resulting fqdn_instance_labels are expected to match the `ComputeNode.hostname` used in the Watcher decision engine cluster model. An example `fqdn_instance_labels` is the following:

```
[
 'ena.controlplane.domain',
 'dio.controlplane.domain',
 'tria.controlplane.domain',
]
```

For instance metrics, it is required that Prometheus contains a label with the uuid of the OpenStack instance in each relevant metric. By default, the datasource will look for the label `resource`. The `instance_uuid_label` config option in watcher.conf allows deployers to override this default to any other label name that stores the `uuid`.

Limitations

The current implementation doesnt support the `statistic_series` function of the Watcher `class DataSourceBase`. It is expected that the `statistic_aggregation` function (which is implemented) is sufficient in providing the **current** state of the managed resources in the cluster. The `statistic_aggregation` function defaults to querying back 300 seconds, starting from the present time (the time period is a function parameter and can be set to a value as required). Implementing the `statistic_series` can always be re-visited if the requisite interest and work cycles are volunteered by the interested parties.

One further note about a limitation in the implemented `statistic_aggregation` function. This function is defined with a `granularity` parameter, to be used when querying whichever of the Watcher `DataSourceBase` metrics providers. In the case of Prometheus, we do not fetch and then process individual metrics across the specified time period. Instead we use the PromQL querying operators and functions, so that the server itself will process the request across the specified parameters and then return the result. So `granularity` parameter is redundant and remains unused for the Prometheus implementation of `statistic_aggregation`. The granularity of the data fetched by Prometheus server is specified in configuration as the server `scrape_interval` (current default 15 seconds).

Configuration

A deployer must set the `datasources` parameter to include `prometheus` under the watcher_datasources section of watcher.conf (or add `prometheus` in datasources for a specific strategy if preferred eg. under the `[watcher_strategies.workload_stabilization]` section).

The watcher.conf configuration file is also used to set the parameter values required by the Watcher Prometheus data source. The configuration can be added under the `[prometheus_client]` section and the available options are duplicated below from the code as they are self documenting:

```
cfg.StrOpt('host',
           help="The hostname or IP address for the prometheus server."),
cfg.StrOpt('port',
           help="The port number used by the prometheus server."),
cfg.StrOpt('fqdn_label',
           default="fqdn",
           help="The label that Prometheus uses to store the fqdn of "
                "exporters. Defaults to 'fqdn'."),
cfg.StrOpt('instance_uuid_label',
           default="resource",
           help="The label that Prometheus uses to store the uuid of "
                "OpenStack instances. Defaults to 'resource'."),
cfg.StrOpt('username',
           help="The basic_auth username to use to authenticate with the "
                "Prometheus server."),
cfg.StrOpt('password',
           secret=True,
           help="The basic_auth password to use to authenticate with the "
                "Prometheus server."),
cfg.StrOpt('cafile',
           help="Path to the CA certificate for establishing a TLS "
                "connection with the Prometheus server."),
cfg.StrOpt('certfile',
           help="Path to the client certificate for establishing a TLS "
                "connection with the Prometheus server."),
cfg.StrOpt('keyfile',
           help="Path to the client key for establishing a TLS "
                "connection with the Prometheus server."),
```

The `host` and `port` are **required** configuration options which have no set default. These specify the hostname (or IP) and port for at which the Prometheus server is listening. The `fqdn_label` allows deployers to override the required metric label used to match Prometheus node exporters against the Watcher ComputeNodes in the Watcher decision engine cluster data model. The default is `fqdn` and deployers can specify any other value (e.g. if they have an equivalent but different label such as `host`).

So a sample watcher.conf configured to use the Prometheus server at `10.2.3.4:9090` would look like the following:

```
[watcher_datasources]

datasources = prometheus
```

(continues on next page)

(continued from previous page)

```
[prometheus_client]

host = 10.2.3.4
port = 9090
fqdn_label = fqdn
```

4.6 Notifications in Watcher

Event type	Notification class	Payload class	Sample
action.cancel.end	ActionCancelNotification	ActionCancelPayload	
action.cancel.error	ActionCancelNotification	ActionCancelPayload	
action.cancel.start	ActionCancelNotification	ActionCancelPayload	
action.create	ActionCreateNotification	ActionCreatePayload	
action.delete	ActionDeleteNotification	ActionDeletePayload	
action.execution.end	ActionExecutionNotificati	ActionExecutionPaylo	
action.execution.error	ActionExecutionNotificati	ActionExecutionPaylo	
action.execution.start	ActionExecutionNotificati	ActionExecutionPaylo	
action_plan.execution.end	ActionPlanActionNotificat	ActionPlanActionPayl	
action_plan.execution.error	ActionPlanActionNotificat	ActionPlanActionPayl	
action_plan.execution.start	ActionPlanActionNotificat	ActionPlanActionPayl	
action_plan.cancel.end	ActionPlanCancelNotificat	ActionPlanCancelPayl	
action_plan.cancel.error	ActionPlanCancelNotificat	ActionPlanCancelPayl	
action_plan.cancel.start	ActionPlanCancelNotificat	ActionPlanCancelPayl	
action_plan.create	ActionPlanCreateNotificat	ActionPlanCreatePayl	
action_plan.delete	ActionPlanDeleteNotificat	ActionPlanDeletePayl	
action_plan.update	ActionPlanUpdateNotificat	ActionPlanUpdatePayl	
action.update	ActionUpdateNotification	ActionUpdatePayload	
audit.strategy.end	AuditActionNotification	AuditActionPayload	
audit.strategy.error	AuditActionNotification	AuditActionPayload	
audit.strategy.start	AuditActionNotification	AuditActionPayload	
audit.create	AuditCreateNotification	AuditCreatePayload	
audit.delete	AuditDeleteNotification	AuditDeletePayload	
audit.update	AuditUpdateNotification	AuditUpdatePayload	
infra.optim.exception	ExceptionNotification	ExceptionPayload	
service.update	ServiceUpdateNotificatior	ServiceUpdatePayload	

4.7 Concurrency

4.7.1 Introduction

Modern processors typically contain multiple cores all capable of executing instructions in parallel. Ensuring applications can fully utilize modern underlying hardware requires developing with these concepts

in mind. The OpenStack foundation maintains a number of libraries to facilitate this utilization, combined with constructs like CPythons GIL the proper use of these concepts becomes more straightforward compared to other programming languages.

The primary libraries maintained by OpenStack to facilitate concurrency are futurist and taskflow. Here futurist is a more straightforward and lightweight library while taskflow is more advanced supporting features like rollback mechanisms. Within Watcher both libraries are used to facilitate concurrency.

4.7.2 Threadpool

A threadpool is a collection of one or more threads typically called *workers* to which tasks can be submitted. These submitted tasks will be scheduled by a threadpool and subsequently executed. In the case of Python tasks typically are bounded or unbounded methods while other programming languages like Java require implementing an interface.

The order and amount of concurrency with which these tasks are executed is up to the threadpool to decide. Some libraries like taskflow allow for either strong or loose ordering of tasks while others like futurist might only support loose ordering. Taskflow supports building tree-based hierarchies of dependent tasks for example.

Upon submission of a task to a threadpool a so called future is returned. These objects allow to determine information about the task such as if it is currently being executed or if it has finished execution. When the task has finished execution the future can also be used to retrieve what was returned by the method.

Some libraries like futurist provide synchronization primitives for collections of futures such as wait_for_any. The following sections will cover different types of concurrency used in various services of Watcher.

4.7.3 Decision engine concurrency

The concurrency in the decision engine is governed by two independent threadpools. Both of these threadpools are GreenThreadPoolExecutor from the futurist library. One of these is used automatically and most contributors will not interact with it while developing new features. The other threadpool can frequently be used while developing new features or updating existing ones. It is known as the DecisionEngineThreadpool and allows to achieve performance improvements in network or I/O bound operations.

AuditEndpoint

The first threadpool is used to allow multiple audits to be run in parallel. In practice, however, only one audit can be run in parallel. This is due to the data model used by audits being a singleton. To prevent audits destroying each others data model one must wait for the other to complete before being allowed to access this data model. A performance improvement could be achieved by being more intelligent in the use, caching and construction of these data models.

DecisionEngineThreadPool

The second threadpool is used for generic tasks, typically networking and I/O could benefit the most of this threadpool. Upon execution of an audit this threadpool can be utilized to retrieve information from the Nova compute service for instance. This second threadpool is a singleton and is shared amongst concurrently running audits as a result the amount of workers is static and independent from the amount of workers in the first threadpool. The use of the `DecisionEngineThreadpool` while building the Nova compute data model is demonstrated to show how it can effectively be used.

In the following example a reference to the `DecisionEngineThreadpool` is stored in `self.executor`. Here two tasks are submitted one with function `self._collect_aggregates` and the other function `self._collect_zones`. With both `self.executor.submit` calls subsequent arguments are passed to the function. All subsequent arguments are passed to the function being submitted as task following the common `(fn, *args, **kwargs)` signature. One of the original signatures would be `def _collect_aggregates(host_aggregates, compute_nodes)` for example.

```
zone_aggregate_futures = {
    self.executor.submit(
        self._collect_aggregates, host_aggregates, compute_nodes),
    self.executor.submit(
        self._collect_zones, availability_zones, compute_nodes)
}
waiters.wait_for_all(zone_aggregate_futures)
```

The last statement of the example above waits on all futures to complete. Similarly, `waiters.wait_for_any` will wait for any future of the specified collection to complete. To simplify the usage of `wait_for_any` the `DecisiongEngineThreadpool` defines a `do_while_futures` method. This method will iterate in a do_while loop over a collection of futures until all of them have completed. The advantage of `do_while_futures` is that it allows to immediately call a method as soon as a future finishes. The arguments for this callback method can be supplied when calling `do_while_futures`, however, the first argument to the callback is always the future itself! If the collection of futures can safely be modified `do_while_futures_modify` can be used and should have slightly better performance. The following example will show how `do_while_futures` is used in the decision engine.

```
# For every compute node from compute_nodes submit a task to gather the node␣
↪it's information.
# List comprehension is used to store all the futures of the submitted tasks␣
↪in node_futures.
node_futures = [self.executor.submit(
    self.nova_helper.get_compute_node_by_name,
    node, servers=True, detailed=True)
    for node in compute_nodes]
LOG.debug("submitted {0} jobs".format(len(compute_nodes)))

future_instances = []
# do_while iterate over node_futures and upon completion of a future call
# self._compute_node_future with the future and future_instances as arguments.
self.executor.do_while_futures_modify(
    node_futures, self._compute_node_future, future_instances)

# Wait for all instance jobs to finish
waiters.wait_for_all(future_instances)
```

Finally, lets demonstrate how powerful this `do_while_futures` can be by showing what the `compute_node_future` callback does. First, it retrieves the result from the future and adds the compute node to the data model. Afterwards, it checks if the compute node has any associated instances and if so it submits an additional task to the `DecisionEngineThreadpool`. The future is appended to the `future_instances` so `waiters.wait_for_all` can be called on this list. This is important as otherwise the building of the data model might return before all tasks for instances have finished.

```
# Get the result from the future.
node_info = future.result()[0]

# Filter out baremetal nodes.
if node_info.hypervisor_type == 'ironic':
    LOG.debug("filtering out baremetal node: %s", node_info)
    return

# Add the compute node to the data model.
self.add_compute_node(node_info)
# Get the instances from the compute node.
instances = getattr(node_info, "servers", None)
# Do not submit job if there are no instances on compute node.
if instances is None:
    LOG.info("No instances on compute_node: {0}".format(node_info))
    return
# Submit a job to retrieve detailed information about the instances.
future_instances.append(
    self.executor.submit(
        self.add_instance_node, node_info, instances)
)
```

Without `do_while_futures` an additional `waiters.wait_for_all` would be required in between the compute node tasks and the instance tasks. This would cause the progress of the decision engine to stall as less and less tasks remain active before the instance tasks could be submitted. This demonstrates how `do_while_futures` can be used to achieve more constant utilization of the underlying hardware.

4.7.4 Applier concurrency

The applier does not use the futurist GreenThreadPoolExecutor directly but instead uses taskflow. However, taskflow still utilizes a greenthreadpool. This threadpool is initialized in the workflow engine called `DefaultWorkFlowEngine`. Currently Watcher supports one workflow engine but the base class allows contributors to develop other workflow engines as well. In taskflow tasks are created using different types of flows such as a linear, unordered or a graph flow. The linear and graph flow allow for strong ordering between individual tasks and it is for this reason that the workflow engine utilizes a graph flow. The creation of tasks, subsequently linking them into a graph like structure and submitting them is shown below.

```
self.execution_rule = self.get_execution_rule(actions)
flow = gf.Flow("watcher_flow")
actions_uuid = {}
for a in actions:
    task = TaskFlowActionContainer(a, self)
    flow.add(task)
    actions_uuid[a.uuid] = task

for a in actions:
    for parent_id in a.parents:
        flow.link(actions_uuid[parent_id], actions_uuid[a.uuid],
                  decider=self.decider)
```

(continues on next page)

(continued from previous page)

```
e = engines.load(
    flow, executor='greenthreaded', engine='parallel',
    max_workers=self.config.max_workers)
e.run()

return flow
```

In the applier tasks are contained in a `TaskFlowActionContainer` which allows them to trigger events in the workflow engine. This way the workflow engine can halt or take other actions while the action plan is being executed based on the success or failure of individual actions. However, the base workflow engine simply uses these notifies to store the result of individual actions in the database. Additionally, since taskflow uses a graph flow if any of the tasks would fail all children of this tasks not be executed while `do_revert` will be triggered for all parents.

```
class TaskFlowActionContainer(...):
    ...
    def do_execute(self, *args, **kwargs):
        ...
        result = self.action.execute()
        if result is True:
            return self.engine.notify(self._db_action,
                                      objects.action.State.SUCCEEDED)
        else:
            self.engine.notify(self._db_action,
                               objects.action.State.FAILED)

class BaseWorkFlowEngine(...):
    ...
    def notify(self, action, state):
        db_action = objects.Action.get_by_uuid(self.context, action.uuid,
                                               eager=True)
        db_action.state = state
        db_action.save()
        return db_action
```

CHAPTER

FIVE

USER GUIDE

5.1 Ways to install Watcher

This document describes some ways to install Watcher in order to use it. If you are intending to develop on or with Watcher, please read *Set up a development environment manually*.

5.1.1 Prerequisites

The source install instructions specifically avoid using platform specific packages, instead using the source for the code and the Python Package Index (PyPi).

Its expected that your system already has python2.7, latest version of pip, and git available.

Your system shall also have some additional system libraries:

On Ubuntu (tested on 16.04LTS):

```
$ sudo apt-get install python-dev libssl-dev libmysqlclient-dev␣
↪libffi-dev
```

On Fedora-based distributions e.g., Fedora/RHEL/CentOS/Scientific Linux (tested on CentOS 7.1):

```
$ sudo yum install gcc python-devel openssl-devel libffi-devel mysql-
↪devel
```

5.1.2 Installing from Source

Clone the Watcher repository:

```
$ git clone https://opendev.org/openstack/watcher.git
$ cd watcher
```

Install the Watcher modules:

```
# python setup.py install
```

The following commands should be available on the command-line path:

- `watcher-api` the Watcher Web service used to handle RESTful requests
- `watcher-decision-engine` the Watcher Decision Engine used to build action plans, according to optimization goals to achieve.

- `watcher-applier` the Watcher Applier module, used to apply action plan
- `watcher-db-manage` used to bootstrap Watcher data

You will find sample configuration files in `etc/watcher`:

- `watcher.conf.sample`

Install the Watcher modules dependencies:

```
# pip install -r requirements.txt
```

From here, refer to *Configuring Watcher* to declare Watcher as a new service into Keystone and to configure its different modules. Once configured, you should be able to run the Watcher services by issuing these commands:

```
$ watcher-api
$ watcher-decision-engine
$ watcher-applier
```

By default, this will show logging on the console from which it was started. Once started, you can use the Watcher Client to play with Watcher service.

5.1.3 Installing from packages: PyPI

Watcher package is available on PyPI repository. To install Watcher on your system:

```
$ sudo pip install python-watcher
```

The Watcher services along with its dependencies should then be automatically installed on your system.

Once installed, you still need to declare Watcher as a new service into Keystone and to configure its different modules, which you can find described in *Configuring Watcher*.

5.1.4 Installing from packages: Debian (experimental)

Experimental Debian packages are available on Debian repositories. The best way to use them is to install them into a Docker container.

Here is single Dockerfile snippet you can use to run your Docker container:

```
FROM debian:experimental
MAINTAINER David TARDIVEL <david.tardivel@b-com.com>

RUN  apt-get update
RUN  apt-get dist-upgrade
RUN  apt-get install vim  net-tools
RUN  apt-get install experimental watcher-api

CMD ["/usr/bin/watcher-api"]
```

Build your container from this Dockerfile:

```
$ docker build -t watcher/api .
```

To run your container, execute this command:

```
$ docker run -d -p 9322:9322 watcher/api
```

Check in your logs Watcher API is started

```
$ docker logs <container ID>
```

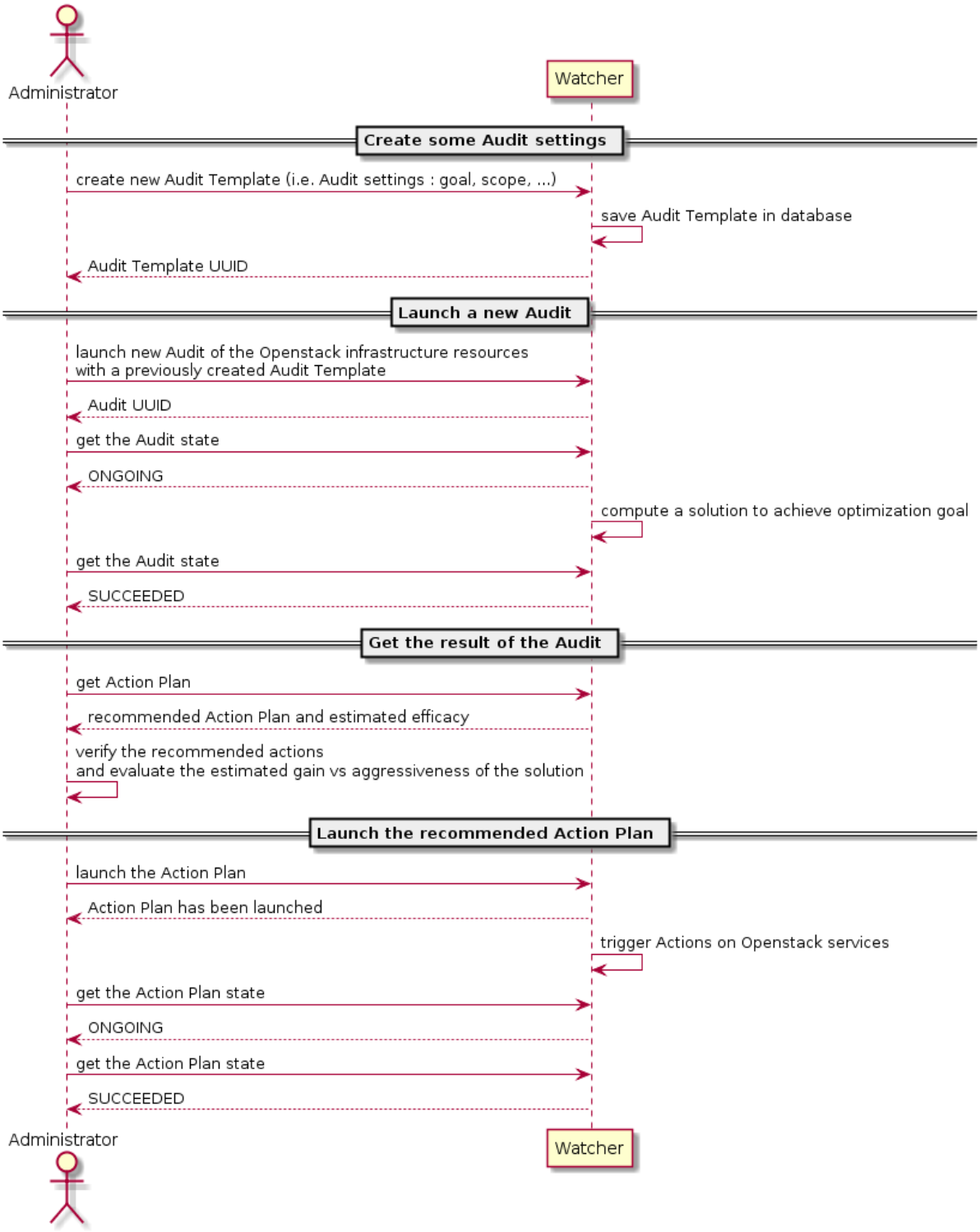
You can run similar container with Watcher Decision Engine (package `watcher-decision-engine`) and with the Watcher Applier (package `watcher-applier`).

5.2 Watcher User Guide

See the architecture page for an architectural overview of the different components of Watcher and how they fit together.

In this guide were going to take you through the fundamentals of using Watcher.

The following diagram shows the main interactions between the *Administrator* and the Watcher system:



5.2.1 Getting started with Watcher

This guide assumes you have a working installation of Watcher. If you get *watcher: command not found* you may have to verify your installation. Please refer to the installation guide. In order to use Watcher, you have to configure your credentials suitable for watcher command-line tools.

You can interact with Watcher either by using our dedicated Watcher CLI named `watcher`, or by using the OpenStack CLI `openstack`.

If you want to deploy Watcher in Horizon, please refer to the Watcher Horizon plugin installation guide.

Note

Notice, that in this guide well use OpenStack CLI as major interface. Nevertheless, you can use Watcher CLI in the same way. It can be achieved by replacing

```
$ openstack optimize ...
```

with

```
$ watcher ...
```

5.2.2 Watcher CLI Command

We can see all of the commands available with Watcher CLI by running the watcher binary without options.

```
$ openstack help optimize
```

5.2.3 Running an audit of the cluster

First, you need to find the *goal* you want to achieve:

```
$ openstack optimize goal list
```

Note

If you get *You must provide a username via either os-username or via env[OS_USERNAME]* you may have to verify your credentials.

Then, you can create an *audit template*. An *audit template* defines an optimization *goal* to achieve (i.e. the settings of your audit).

```
$ openstack optimize audittemplate create my_first_audit_template <your_goal>
```

Although optional, you may want to actually set a specific strategy for your audit template. If so, you may can search of its UUID or name using the following command:

```
$ openstack optimize strategy list --goal <your_goal_uuid_or_name>
```

You can use the following command to check strategy details including which parameters of which format it supports:

```
$ openstack optimize strategy show <your_strategy>
```

The command to create your audit template would then be:

```
$ openstack optimize audittemplate create my_first_audit_template <your_goal>␣
↪\
  --strategy <your_strategy>
```

Then, you can create an audit. An audit is a request for optimizing your cluster depending on the specified *goal*.

You can launch an audit on your cluster by referencing the *audit template* (i.e. the settings of your audit) that you want to use.

- Get the *audit template* UUID or name:

```
$ openstack optimize audittemplate list
```

- Start an audit based on this *audit template* settings:

```
$ openstack optimize audit create -a <your_audit_template>
```

If your_audit_template was created by strategy <your_strategy>, and it defines some parameters (command `watcher strategy show` to check parameters format), your can append `-p` to input required parameters:

```
$ openstack optimize audit create -a <your_audit_template> \
  -p <your_strategy_para1>=5.5 -p <your_strategy_para2>=hi
```

Input parameter could cause audit creation failure, when:

- no predefined strategy for audit template
- no parameters spec in predefined strategy
- input parameters dont comply with spec

Watcher service will compute an *Action Plan* composed of a list of potential optimization *actions* (instance migration, disabling of a compute node,) according to the *goal* to achieve.

- Wait until the Watcher audit has produced a new *action plan*, and get it:

```
$ openstack optimize actionplan list --audit <the_audit_uuid>
```

- Have a look on the list of optimization *actions* contained in this new *action plan*:

```
$ openstack optimize action list --action-plan <the_action_plan_uuid>
```

Once you have learned how to create an *Action Plan*, its time to go further by applying it to your cluster:

- Execute the *action plan*:

```
$ openstack optimize actionplan start <the_action_plan_uuid>
```

You can follow the states of the *actions* by periodically calling:

```
$ openstack optimize action list --action-plan <the_action_plan_uuid>
```

You can also obtain more detailed information about a specific action:

```
$ openstack optimize action show <the_action_uuid>
```

5.3 Audit using Aodh alarm

Audit with EVENT type can be triggered by special alarm. This guide walks you through the steps to build an event-driven optimization solution by integrating Watcher with Ceilometer/Aodh.

5.3.1 Step 1: Create an audit with EVENT type

The first step is to create an audit with EVENT type, you can create an audit template firstly:

```
$ openstack optimize audittemplate create your_template_name <your_goal> \
  --strategy <your_strategy>
```

or create an audit directly with special goal and strategy:

```
$ openstack optimize audit create --goal <your_goal> \
  --strategy <your_strategy> --audit_type EVENT
```

This is an example for creating an audit with dummy strategy:

```
$ openstack optimize audit create --goal dummy \
  --strategy dummy --audit_type EVENT
+---------------+--------------------------------------+
| Field         | Value                                |
+---------------+--------------------------------------+
| UUID          | a3326a6a-c18e-4e8e-adba-d0c61ad404c5 |
| Name          | dummy-2020-01-14T03:21:19.168467     |
| Created At    | 2020-01-14T03:21:19.200279+00:00     |
| Updated At    | None                                 |
| Deleted At    | None                                 |
| State         | PENDING                              |
| Audit Type    | EVENT                                |
| Parameters    | {u'para2': u'hello', u'para1': 3.2}  |
| Interval      | None                                 |
| Goal          | dummy                                |
| Strategy      | dummy                                |
| Audit Scope   | []                                   |
| Auto Trigger  | False                                |
| Next Run Time | None                                 |
| Hostname      | None                                 |
| Start Time    | None                                 |
| End Time      | None                                 |
| Force         | False                                |
+---------------+--------------------------------------+
```

We need to build Aodh action url using Watcher webhook API. For convenience we export the url into an environment variable:

```
$ export AUDIT_UUID=a3326a6a-c18e-4e8e-adba-d0c61ad404c5
$ export ALARM_URL="trust+http://localhost/infra-optim/v1/webhooks/$AUDIT_UUID
↪"
```

5.3.2 Step 2: Create Aodh Alarm

Once we have the audit created, we can continue to create Aodh alarm and set the alarm action to Watcher webhook API. The alarm type can be event(i.e. `compute.instance.create.end`) or gnocchi_resources_threshold(i.e. `cpu_util`), more info refer to alarm-creation

For example:

```
$ openstack alarm create \
  --type event --name instance_create \
  --event-type "compute.instance.create.end" \
  --enable True --repeat-actions False \
  --alarm-action $ALARM_URL
+---------------------------+------------------------------------------------
↪----------------------------------------+
| Field                     | Value                                          ␣
↪                                        |
+---------------------------+------------------------------------------------
↪----------------------------------------+
| alarm_actions             | [u'trust+http://localhost/infra-optim/v1/
↪webhooks/a3326a6a-c18e-4e8e-adba-d0c61ad404c5'] |
| alarm_id                  | b9e381fc-8e3e-4943-82ee-647e7a2ef644           ␣
↪                                        |
| description               | Alarm when compute.instance.create.end event␣
↪occurred.                               |
| enabled                   | True                                           ␣
↪                                        |
| event_type                | compute.instance.create.end                    ␣
↪                                        |
| insufficient_data_actions | []                                             ␣
↪                                        |
| name                      | instance_create                                ␣
↪                                        |
| ok_actions                | []                                             ␣
↪                                        |
| project_id                | 728d66e18c914af1a41e2a585cf766af               ␣
↪                                        |
| query                     |                                                ␣
↪                                        |
| repeat_actions            | False                                          ␣
↪                                        |
| severity                  | low                                            ␣
↪                                        |
| state                     | insufficient data                              ␣
↪                                        |
| state_reason              | Not evaluated yet                              ␣
↪                                        |
| state_timestamp           | 2020-01-14T03:56:26.894416                     ␣
↪                                        |
| time_constraints          | []                                             ␣
↪                                        |
| timestamp                 | 2020-01-14T03:56:26.894416                     ␣
↪                                        |
| type                      | event                                          ␣
↪                                        |
| user_id                   | 88c40156af7445cc80580a1e7e3ba308               ␣
↪                                        |
+---------------------------+------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
↪----------------------------------------+
```

5.3.3 Step 3: Trigger the alarm

In this example, you can create a new instance to trigger the alarm. The alarm state will translate from `insufficient data` to `alarm`.

```
$ openstack alarm show b9e381fc-8e3e-4943-82ee-647e7a2ef644
+---------------------------+-------------------------------------------------
↪------------------------------------------------------------------+
| Field                     | Value                                           ␣
↪                                                                  |
+---------------------------+-------------------------------------------------
↪------------------------------------------------------------------+
| alarm_actions             | [u'trust+http://localhost/infra-optim/v1/
↪webhooks/a3326a6a-c18e-4e8e-adba-d0c61ad404c5']                    |
| alarm_id                  | b9e381fc-8e3e-4943-82ee-647e7a2ef644            ␣
↪                                                                  |
| description               | Alarm when compute.instance.create.end event␣
↪occurred.                                                         |
| enabled                   | True                                            ␣
↪                                                                  |
| event_type                | compute.instance.create.end                     ␣
↪                                                                  |
| insufficient_data_actions | []                                              ␣
↪                                                                  |
| name                      | instance_create                                 ␣
↪                                                                  |
| ok_actions                | []                                              ␣
↪                                                                  |
| project_id                | 728d66e18c914af1a41e2a585cf766af                ␣
↪                                                                  |
| query                     |                                                 ␣
↪                                                                  |
| repeat_actions            | False                                           ␣
↪                                                                  |
| severity                  | low                                             ␣
↪                                                                  |
| state                     | alarm                                           ␣
↪                                                                  |
| state_reason              | Event <id=67dd0afa-2082-45a4-8825-9573b2cc60e5,
↪event_type=compute.instance.create.end> hits the query <query=[]>. |
| state_timestamp           | 2020-01-14T03:56:26.894416                      ␣
↪                                                                  |
| time_constraints          | []                                              ␣
↪                                                                  |
| timestamp                 | 2020-01-14T06:17:40.350649                      ␣
↪                                                                  |
| type                      | event                                           ␣
```

(continues on next page)

(continued from previous page)

```
↪                                                                              |
| user_id                    | 88c40156af7445cc80580a1e7e3ba308                 ␣
↪                                                                              |
+----------------------------+-------------------------------------------------
↪-----------------------------------------------------------------+
```

5.3.4 Step 4: Verify the audit

This can be verified to check if the audit state was SUCCEEDED:

```
$ openstack optimize audit show a3326a6a-c18e-4e8e-adba-d0c61ad404c5
+---------------+--------------------------------------+
| Field         | Value                                |
+---------------+--------------------------------------+
| UUID          | a3326a6a-c18e-4e8e-adba-d0c61ad404c5 |
| Name          | dummy-2020-01-14T03:21:19.168467     |
| Created At    | 2020-01-14T03:21:19+00:00            |
| Updated At    | 2020-01-14T06:26:40+00:00            |
| Deleted At    | None                                 |
| State         | SUCCEEDED                            |
| Audit Type    | EVENT                                |
| Parameters    | {u'para2': u'hello', u'para1': 3.2}  |
| Interval      | None                                 |
| Goal          | dummy                                |
| Strategy      | dummy                                |
| Audit Scope   | []                                   |
| Auto Trigger  | False                                |
| Next Run Time | None                                 |
| Hostname      | ubuntudbs                            |
| Start Time    | None                                 |
| End Time      | None                                 |
| Force         | False                                |
+---------------+--------------------------------------+
```

and you can use the following command to check if the action plan was created:

```
$ openstack optimize actionplan list --audit a3326a6a-c18e-4e8e-adba-
↪d0c61ad404c5
+--------------------------------------+--------------------------------------
↪+-------------+------------+-----------------+
| UUID                                 | Audit                                ␣
↪| State       | Updated At | Global efficacy |
+--------------------------------------+--------------------------------------
↪+-------------+------------+-----------------+
| 673b3fcb-8c16-4a41-9ee3-2956d9f6ca9e | a3326a6a-c18e-4e8e-adba-d0c61ad404c5␣
↪| RECOMMENDED | None       |                 |
+--------------------------------------+--------------------------------------
↪+-------------+------------+-----------------+
```

CHAPTER

SIX

CONFIGURATION GUIDE

6.1 Configuring Watcher

This document is continually updated and reflects the latest available code of the Watcher service.

6.1.1 Service overview

The Watcher system is a collection of services that provides support to optimize your IaaS platform. The Watcher service may, depending upon configuration, interact with several other OpenStack services. This includes:

- the OpenStack Identity service (keystone) for request authentication and to locate other OpenStack services.
- the OpenStack Telemetry service (ceilometer) for collecting the resources metrics.
- the time series database (gnocchi) for consuming the resources metrics.
- the OpenStack Compute service (nova) works with the Watcher service and acts as a user-facing API for instance migration.
- the OpenStack Bare Metal service (ironic) works with the Watcher service and allows to manage power state of nodes.
- the OpenStack Block Storage service (cinder) works with the Watcher service and as an API for volume node migration.

The Watcher service includes the following components:

- `watcher-decision-engine`: runs audit on part of your IaaS and return an action plan in order to optimize resource placement.
- `watcher-api`: A RESTful API that processes application requests by sending them to the watcher-decision-engine over RPC.
- `watcher-applier`: applies the action plan.
- python-watcherclient: A command-line interface (CLI) for interacting with the Watcher service.
- watcher-dashboard: An Horizon plugin for interacting with the Watcher service.

Additionally, the Watcher service has certain external dependencies, which are very similar to other OpenStack services:

- A database to store audit and action plan information and state. You can set the database back-end type and location.
- A queue. A central hub for passing messages, such as RabbitMQ.

Optionally, one may wish to utilize the following associated projects for additional functionality:

- watcher metering: an alternative to collect and push metrics to the Telemetry service.

6.1.2 Install and configure prerequisites

You can configure Watcher services to run on separate nodes or the same node. In this guide, the components run on one node, typically the Controller node.

This section shows you how to install and configure the services.

It assumes that the Identity, Image, Compute, and Networking services have already been set up.

Configure the Identity service for the Watcher service

1. Create the Watcher service user (eg `watcher`). The service uses this to authenticate with the Identity Service. Use the `KEYSTONE_SERVICE_PROJECT_NAME` project (named `service` by default in devstack) and give the user the `admin` role:

   ```
   $ keystone user-create --name=watcher --pass=WATCHER_PASSWORD \
     --email=watcher@example.com \
     --tenant=KEYSTONE_SERVICE_PROJECT_NAME
   $ keystone user-role-add --user=watcher \
     --tenant=KEYSTONE_SERVICE_PROJECT_NAME --role=admin
   ```

   or (by using python-openstackclient 1.8.0+)

   ```
   $ openstack user create  --password WATCHER_PASSWORD --enable \
     --email watcher@example.com watcher \
     --project=KEYSTONE_SERVICE_PROJECT_NAME
   $ openstack role add --project KEYSTONE_SERVICE_PROJECT_NAME \
     --user watcher admin
   ```

2. You must register the Watcher Service with the Identity Service so that other OpenStack services can locate it. To register the service:

   ```
   $ keystone service-create --name=watcher --type=infra-optim \
     --description="Infrastructure Optimization service"
   ```

   or (by using python-openstackclient 1.8.0+)

   ```
   $ openstack service create --name watcher infra-optim \
     --description="Infrastructure Optimization service"
   ```

3. Create the endpoints by replacing YOUR_REGION and `WATCHER_API_[PUBLIC|ADMIN|INTERNAL]_IP` with your region and your Watcher Services API node IP addresses (or FQDN):

   ```
   $ keystone endpoint-create \
   --service-id=the_service_id_above \
   --publicurl=http://WATCHER_API_PUBLIC_IP:9322 \
   --internalurl=http://WATCHER_API_INTERNAL_IP:9322 \
   --adminurl=http://WATCHER_API_ADMIN_IP:9322
   ```

   or (by using python-openstackclient 1.8.0+)

```
$ openstack endpoint create --region YOUR_REGION
  watcher public http://WATCHER_API_PUBLIC_IP:9322

$ openstack endpoint create --region YOUR_REGION
  watcher internal http://WATCHER_API_INTERNAL_IP:9322

$ openstack endpoint create --region YOUR_REGION
  watcher admin http://WATCHER_API_ADMIN_IP:9322
```

Set up the database for Watcher

The Watcher service stores information in a database. This guide uses the MySQL database that is used by other OpenStack services.

1. In MySQL, create a `watcher` database that is accessible by the `watcher` user. Replace WATCHER_DBPASSWORD with the actual password:

```
# mysql

mysql> CREATE DATABASE watcher CHARACTER SET utf8;
mysql> GRANT ALL PRIVILEGES ON watcher.* TO 'watcher'@'localhost' \
IDENTIFIED BY 'WATCHER_DBPASSWORD';
mysql> GRANT ALL PRIVILEGES ON watcher.* TO 'watcher'@'%' \
IDENTIFIED BY 'WATCHER_DBPASSWORD';
```

6.1.3 Configure the Watcher service

The Watcher service is configured via its configuration file. This file is typically located at `/etc/watcher/watcher.conf`.

You can easily generate and update a sample configuration file named *watcher.conf.sample* by using these following commands:

```
$ git clone https://opendev.org/openstack/watcher.git
$ cd watcher/
$ tox -e genconfig
$ vi etc/watcher/watcher.conf.sample
```

The configuration file is organized into the following sections:

- `[DEFAULT]` - General configuration
- `[api]` - API server configuration
- `[database]` - SQL driver configuration
- `[keystone_authtoken]` - Keystone Authentication plugin configuration
- `[watcher_clients_auth]` - Keystone auth configuration for clients
- `[watcher_applier]` - Watcher Applier module configuration
- `[watcher_decision_engine]` - Watcher Decision Engine module configuration
- `[oslo_messaging_rabbit]` - Oslo Messaging RabbitMQ driver configuration

- [cinder_client] - Cinder client configuration
- [glance_client] - Glance client configuration
- [gnocchi_client] - Gnocchi client configuration
- [ironic_client] - Ironic client configuration
- [keystone_client] - Keystone client configuration
- [nova_client] - Nova client configuration
- [neutron_client] - Neutron client configuration
- [placement_client] - Placement client configuration

The Watcher configuration file is expected to be named watcher.conf. When starting Watcher, you can specify a different configuration file to use with --config-file. If you do **not** specify a configuration file, Watcher will look in the following directories for a configuration file, in order:

- ~/.watcher/
- ~/
- /etc/watcher/
- /etc/

Although some configuration options are mentioned here, it is recommended that you review all the *available options* so that the watcher service is configured for your needs.

1. The Watcher Service stores information in a database. This guide uses the MySQL database that is used by other OpenStack services.

   Configure the location of the database via the connection option. In the following, replace WATCHER_DBPASSWORD with the password of your watcher user, and replace DB_IP with the IP address where the DB server is located:

   ```
   [database]
   ...

   # The SQLAlchemy connection string used to connect to the
   # database (string value)
   #connection=<None>
   connection = mysql+pymysql://watcher:WATCHER_DBPASSWORD@DB_IP/watcher?
   ↪charset=utf8
   ```

2. Configure the Watcher Service to use the RabbitMQ message broker by setting one or more of these options. Replace RABBIT_HOST with the IP address of the RabbitMQ server, RABBITMQ_USER and RABBITMQ_PASSWORD by the RabbitMQ server login credentials

   ```
   [DEFAULT]

   # The default exchange under which topics are scoped. May be
   # overridden by an exchange name specified in the transport_url
   # option. (string value)
   control_exchange = watcher
   ```

(continues on next page)

(continued from previous page)

```
# ...
transport_url = rabbit://RABBITMQ_USER:RABBITMQ_PASSWORD@RABBIT_HOST
```

3. Watcher API shall validate the token provided by every incoming request, via keystonemiddleware, which requires the Watcher service to be configured with the right credentials for the Identity service.

   In the configuration section here below:

   - replace IDENTITY_IP with the IP of the Identity server
   - replace WATCHER_PASSWORD with the password you chose for the `watcher` user
   - replace KEYSTONE_SERVICE_PROJECT_NAME with the name of project created for OpenStack services (e.g. `service`)

   ```
   [keystone_authtoken]

   # Authentication type to load (unknown value)
   # Deprecated group/name - [DEFAULT]/auth_plugin
   #auth_type = <None>
   auth_type = password

   # Authentication URL (unknown value)
   #auth_url = <None>
   auth_url = http://IDENTITY_IP:5000

   # Username (unknown value)
   # Deprecated group/name - [DEFAULT]/username
   #username = <None>
   username=watcher

   # User's password (unknown value)
   #password = <None>
   password = WATCHER_PASSWORD

   # Domain ID containing project (unknown value)
   #project_domain_id = <None>
   project_domain_id = default

   # User's domain id (unknown value)
   #user_domain_id = <None>
   user_domain_id = default

   # Project name to scope to (unknown value)
   # Deprecated group/name - [DEFAULT]/tenant-name
   #project_name = <None>
   project_name = KEYSTONE_SERVICE_PROJECT_NAME
   ```

4. Watchers decision engine and applier interact with other OpenStack projects through those projects clients. In order to instantiate these clients, Watcher needs to request a new session from the Identity service using the right credentials.

In the configuration section here below:

- replace IDENTITY_IP with the IP of the Identity server
- replace WATCHER_PASSWORD with the password you chose for the `watcher` user
- replace KEYSTONE_SERVICE_PROJECT_NAME with the name of project created for OpenStack services (e.g. `service`)

```
[watcher_clients_auth]

# Authentication type to load (unknown value)
# Deprecated group/name - [DEFAULT]/auth_plugin
#auth_type = <None>
auth_type = password

# Authentication URL (unknown value)
#auth_url = <None>
auth_url = http://IDENTITY_IP:5000

# Username (unknown value)
# Deprecated group/name - [DEFAULT]/username
#username = <None>
username=watcher

# User's password (unknown value)
#password = <None>
password = WATCHER_PASSWORD

# Domain ID containing project (unknown value)
#project_domain_id = <None>
project_domain_id = default

# User's domain id (unknown value)
#user_domain_id = <None>
user_domain_id = default

# Project name to scope to (unknown value)
# Deprecated group/name - [DEFAULT]/tenant-name
#project_name = <None>
project_name = KEYSTONE_SERVICE_PROJECT_NAME
```

5. Configure the clients to use a specific version if desired. For example, to configure Watcher to use a Nova client with version 2.1, use:

```
[nova_client]

# Version of Nova API to use in novaclient. (string value)
#api_version = 2.56
api_version = 2.1
```

6. Create the Watcher Service database tables:

```
$ watcher-db-manage --config-file /etc/watcher/watcher.conf create_schema
```

7. Start the Watcher Service:

```
$ watcher-api &&  watcher-decision-engine && watcher-applier
```

6.1.4 Configure Nova compute

Please check your hypervisor configuration to correctly handle instance migration.

6.1.5 Configure Measurements

You can configure and install Ceilometer by following the documentation below :

1. https://docs.openstack.org/ceilometer/latest

The built-in strategy basic_consolidation provided by watcher requires **compute.node.cpu.percent** and **cpu** measurements to be collected by Ceilometer. The measurements available depend on the hypervisors that OpenStack manages on the specific implementation. You can find the measurements available per hypervisor and OpenStack release on the OpenStack site. You can use ceilometer meter-list to list the available meters.

For more information: https://docs.openstack.org/ceilometer/latest/admin/telemetry-measurements.html

Ceilometer is designed to collect measurements from OpenStack services and from other external components. If you would like to add new meters to the currently existing ones, you need to follow the documentation below:

1. https://docs.openstack.org/ceilometer/latest/contributor/measurements.html#new-measurements

The Ceilometer collector uses a pluggable storage system, meaning that you can pick any database system you prefer. The original implementation has been based on MongoDB but you can create your own storage driver using whatever technology you want. For more information : https://wiki.openstack.org/wiki/Gnocchi

6.1.6 Configure Nova Notifications

Watcher can consume notifications generated by the Nova services, in order to build or update, in real time, its cluster data model related to computing resources.

Nova emits unversioned(legacy) and versioned notifications on different topics. Because legacy notifications will be deprecated, Watcher consumes Nova versioned notifications.

- In the file `/etc/nova/nova.conf`, the value of driver in the section `[oslo_messaging_notifications]` cant be noop, and the value of notification_format in the section `[notifications]` should be both or versioned

```
[oslo_messaging_notifications]
driver = messagingv2

...

[notifications]
notification_format = both
```

6.1.7 Configure Cinder Notifications

Watcher can also consume notifications generated by the Cinder services, in order to build or update, in real time, its cluster data model related to storage resources. To do so, you have to update the Cinder configuration file on controller and volume nodes, in order to let Watcher receive Cinder notifications in a dedicated `watcher_notifications` channel.

- In the file `/etc/cinder/cinder.conf`, update the section `[oslo_messaging_notifications]`, by redefining the list of topics into which Cinder services will publish events

```
[oslo_messaging_notifications]
driver = messagingv2
topics = notifications,watcher_notifications
```

- Restart the Cinder services.

6.1.8 Workers

You can define a number of workers for the Decision Engine and the Applier.

If you want to create and run more audits simultaneously, you have to raise the number of workers used by the Decision Engine:

```
[watcher_decision_engine]

...

# The maximum number of threads that can be used to execute strategies
# (integer value)
#max_workers = 2
```

If you want to execute simultaneously more recommended action plans, you have to raise the number of workers used by the Applier:

```
[watcher_applier]

...

# Number of workers for applier, default value is 1. (integer value)
# Minimum value: 1
#workers = 1
```

6.2 watcher.conf

The `watcher.conf` file contains most of the options to configure the Watcher services.

6.2.1 DEFAULT

debug

Type
boolean

Default
`False`

Mutable
This option can be changed without restarting.

If set to true, the logging level will be set to DEBUG instead of the default INFO level.

`log_config_append`

Type
string

Default
<None>

Mutable
This option can be changed without restarting.

The name of a logging configuration file. This file is appended to any existing logging configuration files. For details about logging configuration files, see the Python logging module documentation. Note that when logging configuration files are used then all logging configuration is set in the configuration file and other logging configuration options are ignored (for example, log-date-format).

Table 1: Deprecated Variations

Group	Name
DEFAULT	log-config
DEFAULT	log_config

`log_date_format`

Type
string

Default
`%Y-%m-%d %H:%M:%S`

Defines the format string for %(asctime)s in log records. Default: the value above . This option is ignored if log_config_append is set.

`log_file`

Type
string

Default
<None>

(Optional) Name of log file to send logging output to. If no default is set, logging will go to stderr as defined by use_stderr. This option is ignored if log_config_append is set.

Table 2: Deprecated Variations

Group	Name
DEFAULT	logfile

`log_dir`

Type
string

Default
<None>

(Optional) The base directory used for relative log_file paths. This option is ignored if log_config_append is set.

Table 3: Deprecated Variations

Group	Name
DEFAULT	logdir

`watch_log_file`

Type
boolean

Default
`False`

Uses logging handler designed to watch file system. When log file is moved or removed this handler will open a new log file with specified path instantaneously. It makes sense only if log_file option is specified and Linux platform is used. This option is ignored if log_config_append is set.

Warning

This option is deprecated for removal. Its value may be silently ignored in the future.

Reason
This function is known to have bene broken for long time, and depends on the unmaintained library

`use_syslog`

Type
boolean

Default
`False`

Use syslog for logging. Existing syslog format is DEPRECATED and will be changed later to honor RFC5424. This option is ignored if log_config_append is set.

`use_journal`

Type
boolean

Default
`False`

Enable journald for logging. If running in a systemd environment you may wish to enable journal support. Doing so will use the journal native protocol which includes structured metadata in addition to log messages.This option is ignored if log_config_append is set.

`syslog_log_facility`

Type
string

Default
`LOG_USER`

Syslog facility to receive log lines. This option is ignored if log_config_append is set.

`use_json`

Type
boolean

Default
`False`

Use JSON formatting for logging. This option is ignored if log_config_append is set.

`use_stderr`

Type
boolean

Default
`False`

Log output to standard error. This option is ignored if log_config_append is set.

`log_color`

Type
boolean

Default
`False`

(Optional) Set the color key according to log levels. This option takes effect only when logging to stderr or stdout is used. This option is ignored if log_config_append is set.

`log_rotate_interval`

Type
integer

Default
1

The amount of time before the log files are rotated. This option is ignored unless log_rotation_type is set to interval.

`log_rotate_interval_type`

Type
string

Default
`days`

Valid Values
Seconds, Minutes, Hours, Days, Weekday, Midnight

Rotation interval type. The time of the last file change (or the time when the service was started) is used when scheduling the next rotation.

`max_logfile_count`

Type
integer

Default
`30`

Maximum number of rotated log files.

`max_logfile_size_mb`

Type
integer

Default
`200`

Log file maximum size in MB. This option is ignored if log_rotation_type is not set to size.

`log_rotation_type`

Type
string

Default
none

Valid Values
interval, size, none

Log rotation type.

Possible values

interval
Rotate logs at predefined time intervals.

size
Rotate logs once they reach a predefined size.

none
Do not rotate log files.

`logging_context_format_string`

Type
string

Default
`%(asctime)s.%(msecs)03d %(process)d %(levelname)s %(name)s`

```
[%(global_request_id)s %(request_id)s %(user_identity)s]
%(instance)s%(message)s
```

Format string to use for log messages with context. Used by oslo_log.formatters.ContextFormatter

`logging_default_format_string`

Type
string

Default
```
%(asctime)s.%(msecs)03d %(process)d %(levelname)s %(name)s [-]
%(instance)s%(message)s
```

Format string to use for log messages when context is undefined. Used by oslo_log.formatters.ContextFormatter

`logging_debug_format_suffix`

Type
string

Default
```
%(funcName)s %(pathname)s:%(lineno)d
```

Additional data to append to log message when logging level for the message is DEBUG. Used by oslo_log.formatters.ContextFormatter

`logging_exception_prefix`

Type
string

Default
```
%(asctime)s.%(msecs)03d %(process)d ERROR %(name)s
%(instance)s
```

Prefix each line of exception output with this format. Used by oslo_log.formatters.ContextFormatter

`logging_user_identity_format`

Type
string

Default
```
%(user)s %(project)s %(domain)s %(system_scope)s
%(user_domain)s %(project_domain)s
```

Defines the format string for %(user_identity)s that is used in logging_context_format_string. Used by oslo_log.formatters.ContextFormatter

`default_log_levels`

Type
list

Default
```
['amqp=WARN', 'amqplib=WARN', 'boto=WARN', 'qpid=WARN',
'sqlalchemy=WARN', 'suds=INFO', 'oslo.messaging=INFO',
```

```
'oslo_messaging=INFO', 'iso8601=WARN', 'requests.packages.
urllib3.connectionpool=WARN', 'urllib3.connectionpool=WARN',
'websocket=WARN', 'requests.packages.urllib3.util.retry=WARN',
'urllib3.util.retry=WARN', 'keystonemiddleware=WARN',
'routes.middleware=WARN', 'stevedore=WARN', 'taskflow=WARN',
'keystoneauth=WARN', 'oslo.cache=INFO', 'oslo_policy=INFO',
'dogpile.core.dogpile=INFO']
```

List of package logging levels in logger=LEVEL pairs. This option is ignored if log_config_append is set.

publish_errors

Type
boolean

Default
`False`

Enables or disables publication of error events.

instance_format

Type
string

Default
`"[instance: %(uuid)s] "`

The format for an instance that is passed with the log message.

instance_uuid_format

Type
string

Default
`"[instance: %(uuid)s] "`

The format for an instance UUID that is passed with the log message.

rate_limit_interval

Type
integer

Default
`0`

Interval, number of seconds, of log rate limiting.

rate_limit_burst

Type
integer

Default
`0`

Maximum number of logged messages per rate_limit_interval.

`rate_limit_except_level`

Type
string

Default
`CRITICAL`

Valid Values
CRITICAL, ERROR, INFO, WARNING, DEBUG,

Log level name used by rate limiting. Logs with level greater or equal to rate_limit_except_level are not filtered. An empty string means that all levels are filtered.

`fatal_deprecations`

Type
boolean

Default
`False`

Enables or disables fatal status of deprecations.

`enable_authentication`

Type
boolean

Default
`True`

This option enables or disables user authentication via keystone. Default value is True.

`fatal_exception_format_errors`

Type
boolean

Default
`False`

Make exception message format errors fatal.

`pybasedir`

Type
string

Default
`/home/zuul/src/opendev.org/openstack/watcher/watcher`

Directory where the watcher python module is installed.

`bindir`

Type
string

Default
`$pybasedir/bin`

Directory where watcher binaries are installed.

`state_path`

Type
string

Default
`$pybasedir`

Top-level directory for maintaining watchers state.

`periodic_interval`

Type
integer

Default
`60`

Mutable
This option can be changed without restarting.

Seconds between running periodic tasks.

`host`

Type
host address

Default
`npbe7be73f5aa64`

Name of this node. This can be an opaque identifier. It is not necessarily a hostname, FQDN, or IP address. However, the node name must be valid within an AMQP key.

`service_down_time`

Type
integer

Default
`90`

Maximum time since last check-in for up service.

`executor_thread_pool_size`

Type
integer

Default
`64`

Size of executor thread pool when executor is threading or eventlet.

Table 4: Deprecated Variations

Group	Name
DEFAULT	rpc_thread_pool_size

`rpc_response_timeout`

Type
integer

Default
`60`

Seconds to wait for a response from a call.

`transport_url`

Type
string

Default
`rabbit://`

The network address and optional user credentials for connecting to the messaging backend, in URL format. The expected format is:

driver://[user:pass@]host:port[,[userN:passN@]hostN:portN]/virtual_host?query

Example: rabbit://rabbitmq:password@127.0.0.1:5672//

For full details on the fields in the URL see the documentation of oslo_messaging.TransportURL at https://docs.openstack.org/oslo.messaging/latest/reference/transport.html

`control_exchange`

Type
string

Default
`openstack`

The default exchange under which topics are scoped. May be overridden by an exchange name specified in the transport_url option.

`rpc_ping_enabled`

Type
boolean

Default
`False`

Add an endpoint to answer to ping calls. Endpoint is named oslo_rpc_server_ping

`run_external_periodic_tasks`

Type
boolean

Default
`True`

Some periodic tasks can be run in a separate process. Should we run them here?

`backdoor_port`

Type
string

Default
<None>

Enable eventlet backdoor. Acceptable values are 0, <port>, and <start>:<end>, where 0 results in listening on a random tcp port number; <port> results in listening on the specified port number (and not enabling backdoor if that port is in use); and <start>:<end> results in listening on the smallest unused port number within the specified range of port numbers. The chosen port is displayed in the services log file.

Warning

This option is deprecated for removal. Its value may be silently ignored in the future.

Reason
The backdoor_port option is deprecated and will be removed in a future release.

`backdoor_socket`

Type
string

Default
<None>

Enable eventlet backdoor, using the provided path as a unix socket that can receive connections. This option is mutually exclusive with backdoor_port in that only one should be provided. If both are provided then the existence of this option overrides the usage of that option. Inside the path {pid} will be replaced with the PID of the current process.

Warning

This option is deprecated for removal. Its value may be silently ignored in the future.

Reason
The backdoor_socket option is deprecated and will be removed in a future release.

`log_options`

Type
boolean

Default
`True`

Enables or disables logging values of all registered options when starting a service (at DEBUG level).

`graceful_shutdown_timeout`

Type
integer

Default
`60`

Specify a timeout after which a gracefully shutdown server will exit. Zero value means endless wait.

`api_paste_config`

Type
string

Default
`api-paste.ini`

File name for the paste.deploy config for api service

Warning

This option is deprecated for removal. Its value may be silently ignored in the future.

Reason
The api_paste_config option is deprecated and will be removed in a future release.

`wsgi_log_format`

Type
string

Default
`%(client_ip)s "%(request_line)s" status: %(status_code)s len: %(body_length)s time: %(wall_seconds).7f`

A python format string that is used as the template to generate log lines. The following values can beformatted into it: client_ip, date_time, request_line, status_code, body_length, wall_seconds.

Warning

This option is deprecated for removal. Its value may be silently ignored in the future.

Reason
The wsgi_log_format option is deprecated and will be removed in a future release.

`tcp_keepidle`

Type
integer

Default
`600`

Sets the value of TCP_KEEPIDLE in seconds for each server socket. Not supported on OS X.

Warning

This option is deprecated for removal. Its value may be silently ignored in the future.

Reason
The tcp_keepidle option is deprecated and will be removed in a future release.

`wsgi_default_pool_size`

Type
integer

Default
`100`

Size of the pool of greenthreads used by wsgi

Warning

This option is deprecated for removal. Its value may be silently ignored in the future.

Reason
The wsgi_default_pool_size option is deprecated and will be removed in a future release.

`max_header_line`

Type
integer

Default
`16384`

Maximum line size of message headers to be accepted. max_header_line may need to be increased when using large tokens (typically those generated when keystone is configured to use PKI tokens with big service catalogs).

Warning

This option is deprecated for removal. Its value may be silently ignored in the future.

Reason
The max_header_line option is deprecated and will be removed in a future release.

`wsgi_keep_alive`

Type
boolean

Default
`True`

If False, closes the client socket connection explicitly.

Warning

This option is deprecated for removal. Its value may be silently ignored in the future.

Reason
The wsgi_keep_alive option is deprecated and will be removed in a future release.

`client_socket_timeout`

Type
integer

Default
`900`

Timeout for client connections socket operations. If an incoming connection is idle for this number of seconds it will be closed. A value of 0 means wait forever.

Warning

This option is deprecated for removal. Its value may be silently ignored in the future.

Reason
The client_socket_timeout option is deprecated and will be removed in a future release.

`wsgi_server_debug`

Type
boolean

Default
`False`

True if the server should send exception tracebacks to the clients on 500 errors. If False, the server will respond with empty bodies.

Warning

This option is deprecated for removal. Its value may be silently ignored in the future.

Reason
The wsgi_server_debug option is deprecated and will be removed in a future release.

6.2.2 api

`port`

Type
port number

Default
9322

Minimum Value
0

Maximum Value
65535

The port for the watcher API server

`host`

Type
host address

Default
127.0.0.1

The listen IP address for the watcher API server

`max_limit`

Type
integer

Default
1000

The maximum number of items returned in a single response from a collection resource

`workers`

Type
integer

Default
<None>

Minimum Value
1

Number of workers for Watcher API service. The default is equal to the number of CPUs available if that can be determined, else a default worker count of 1 is returned.

`enable_ssl_api`

Type
boolean

Default
False

Enable the integrated stand-alone API to service requests via HTTPS instead of HTTP. If there is a front-end service performing HTTPS offloading from the service, this option should be False; note, you will want to change public API endpoint to represent SSL termination URL with public_endpoint option.

`enable_webhooks_auth`

Type
boolean

Default
`True`

This option enables or disables webhook request authentication via keystone. Default value is True.

6.2.3 cache

`config_prefix`

Type
string

Default
`cache.oslo`

Prefix for building the configuration dictionary for the cache region. This should not need to be changed unless there is another dogpile.cache region with the same configuration name.

`expiration_time`

Type
integer

Default
`600`

Minimum Value
1

Default TTL, in seconds, for any cached item in the dogpile.cache region. This applies to any cached method that doesnt have an explicit cache expiration time defined for it.

`backend_expiration_time`

Type
integer

Default
<None>

Minimum Value
1

Expiration time in cache backend to purge expired records automatically. This should be greater than expiration_time and all cache_time options

`backend`

Type
string

Default
`dogpile.cache.null`

Valid Values
oslo_cache.memcache_pool, oslo_cache.dict, oslo_cache.mongo, oslo_cache.etcd3gw, dogpile.cache.pymemcache, dogpile.cache.memcached,

dogpile.cache.pylibmc, dogpile.cache.bmemcached, dogpile.cache.dbm, dogpile.cache.redis, dogpile.cache.redis_sentinel, dogpile.cache.memory, dogpile.cache.memory_pickle, dogpile.cache.null

Cache backend module. For eventlet-based or environments with hundreds of threaded servers, Memcache with pooling (oslo_cache.memcache_pool) is recommended. For environments with less than 100 threaded servers, Memcached (dogpile.cache.memcached) or Redis (dogpile.cache.redis) is recommended. Test environments with a single instance of the server can use the dogpile.cache.memory backend.

backend_argument

Type
multi-valued

Default
''

Arguments supplied to the backend module. Specify this option once per argument to be passed to the dogpile.cache backend. Example format: <argname>:<value>.

proxies

Type
list

Default
[]

Proxy classes to import that will affect the way the dogpile.cache backend functions. See the dogpile.cache documentation on changing-backend-behavior.

enabled

Type
boolean

Default
False

Global toggle for caching.

debug_cache_backend

Type
boolean

Default
False

Extra debugging from the cache backend (cache keys, get/set/delete/etc calls). This is only really useful if you need to see the specific cache-backend get/set/delete calls with the keys/values. Typically this should be left set to false.

memcache_servers

Type
list

Default
`['localhost:11211']`

Memcache servers in the format of host:port. This is used by backends dependent on Memcached.If `dogpile.cache.memcached` or `oslo_cache.memcache_pool` is used and a given host refer to an IPv6 or a given domain refer to IPv6 then you should prefix the given address with the address family (`inet6`) (e.g `inet6[::1]:11211`, `inet6:[fd12:3456:789a:1::1]:11211`, `inet6:[controller-0.internalapi]:11211`). If the address family is not given then these backends will use the default `inet` address family which corresponds to IPv4

`memcache_dead_retry`

Type
integer

Default
`300`

Number of seconds memcached server is considered dead before it is tried again. (dogpile.cache.memcache and oslo_cache.memcache_pool backends only).

`memcache_socket_timeout`

Type
floating point

Default
`1.0`

Timeout in seconds for every call to a server. (dogpile.cache.memcache and oslo_cache.memcache_pool backends only).

`memcache_pool_maxsize`

Type
integer

Default
`10`

Max total number of open connections to every memcached server. (oslo_cache.memcache_pool backend only).

`memcache_pool_unused_timeout`

Type
integer

Default
`60`

Number of seconds a connection to memcached is held unused in the pool before it is closed. (oslo_cache.memcache_pool backend only).

`memcache_pool_connection_get_timeout`

Type
integer

Default
10

Number of seconds that an operation will wait to get a memcache client connection.

memcache_pool_flush_on_reconnect

Type
boolean

Default
False

Global toggle if memcache will be flushed on reconnect. (oslo_cache.memcache_pool backend only).

memcache_sasl_enabled

Type
boolean

Default
False

Enable the SASL(Simple Authentication and SecurityLayer) if the SASL_enable is true, else disable.

memcache_username

Type
string

Default
<None>

the user name for the memcached which SASL enabled

memcache_password

Type
string

Default
<None>

the password for the memcached which SASL enabled

redis_server

Type
string

Default
localhost:6379

Redis server in the format of host:port

redis_db

Type
integer

Default
`0`

Minimum Value
0

Database id in Redis server

`redis_username`

Type
string

Default
<None>

the user name for redis

`redis_password`

Type
string

Default
<None>

the password for redis

`redis_sentinels`

Type
list

Default
`['localhost:26379']`

Redis sentinel servers in the format of host:port

`redis_socket_timeout`

Type
floating point

Default
`1.0`

Timeout in seconds for every call to a server. (dogpile.cache.redis and dogpile.cache.redis_sentinel backends only).

`redis_sentinel_service_name`

Type
string

Default
`mymaster`

Service name of the redis sentinel cluster.

`tls_enabled`

Type
boolean

Default
`False`

Global toggle for TLS usage when communicating with the caching servers. Currently supported by `dogpile.cache.bmemcache`, `dogpile.cache.pymemcache`, `oslo_cache.memcache_pool`, `dogpile.cache.redis` and `dogpile.cache.redis_sentinel`.

`tls_cafile`

Type
string

Default
<None>

Path to a file of concatenated CA certificates in PEM format necessary to establish the caching servers authenticity. If tls_enabled is False, this option is ignored.

`tls_certfile`

Type
string

Default
<None>

Path to a single file in PEM format containing the clients certificate as well as any number of CA certificates needed to establish the certificates authenticity. This file is only required when client side authentication is necessary. If tls_enabled is False, this option is ignored.

`tls_keyfile`

Type
string

Default
<None>

Path to a single file containing the clients private key in. Otherwise the private key will be taken from the file specified in tls_certfile. If tls_enabled is False, this option is ignored.

`tls_allowed_ciphers`

Type
string

Default
<None>

Set the available ciphers for sockets created with the TLS context. It should be a string in the OpenSSL cipher list format. If not specified, all OpenSSL enabled ciphers will be available. Currently supported by `dogpile.cache.bmemcache`, `dogpile.cache.pymemcache` and `oslo_cache.memcache_pool`.

`enable_socket_keepalive`

Type
boolean

Default
`False`

Global toggle for the socket keepalive of dogpiles pymemcache backend

`socket_keepalive_idle`

Type
integer

Default
1

Minimum Value
0

The time (in seconds) the connection needs to remain idle before TCP starts sending keepalive probes. Should be a positive integer most greater than zero.

`socket_keepalive_interval`

Type
integer

Default
1

Minimum Value
0

The time (in seconds) between individual keepalive probes. Should be a positive integer greater than zero.

`socket_keepalive_count`

Type
integer

Default
1

Minimum Value
0

The maximum number of keepalive probes TCP should send before dropping the connection. Should be a positive integer greater than zero.

`enable_retry_client`

Type
boolean

Default
`False`

Enable retry client mechanisms to handle failure. Those mechanisms can be used to wrap all kind of pymemcache clients. The wrapper allows you to define how many attempts to make and how long to wait between attemots.

`retry_attempts`

Type
integer

Default
2

Minimum Value
1

Number of times to attempt an action before failing.

`retry_delay`

Type
floating point

Default
0

Number of seconds to sleep between each attempt.

`hashclient_retry_attempts`

Type
integer

Default
2

Minimum Value
1

Amount of times a client should be tried before it is marked dead and removed from the pool in the HashClients internal mechanisms.

`hashclient_retry_delay`

Type
floating point

Default
1

Time in seconds that should pass between retry attempts in the HashClients internal mechanisms.

`dead_timeout`

Type
floating point

Default
60

Time in seconds before attempting to add a node back in the pool in the HashClients internal mechanisms.

enforce_fips_mode

Type
boolean

Default
`False`

Global toggle for enforcing the OpenSSL FIPS mode. This feature requires Python support. This is available in Python 3.9 in all environments and may have been backported to older Python versions on select environments. If the Python executable used does not support OpenSSL FIPS mode, an exception will be raised. Currently supported by `dogpile.cache.bmemcache`, `dogpile.cache.pymemcache` and `oslo_cache.memcache_pool`.

6.2.4 cinder_client

api_version

Type
string

Default
`3`

Version of Cinder API to use in cinderclient.

endpoint_type

Type
string

Default
`publicURL`

Type of endpoint to use in cinderclient. Supported values: internalURL, publicURL, adminURL. The default is publicURL.

region_name

Type
string

Default
`<None>`

Region in Identity service catalog to use for communication with the OpenStack service.

6.2.5 collector

collector_plugins

Type
list

Default
`['compute']`

The cluster data model plugin names.

Supported in-tree collectors include:

- `compute` - data model collector for nova
- `storage` - data model collector for cinder
- `baremetal` - data model collector for ironic

Custom data model collector plugins can be defined with the `watcher_cluster_data_model_collectors` extension point.

`api_call_retries`

Type
integer

Default
`10`

Number of retries before giving up on external service calls.

`api_query_timeout`

Type
integer

Default
`1`

Time before retry after failed call to external service.

6.2.6 database

`mysql_engine`

Type
string

Default
`InnoDB`

MySQL engine to use.

`sqlite_synchronous`

Type
boolean

Default
`True`

If True, SQLite uses synchronous mode.

`backend`

Type
string

Default
`sqlalchemy`

The back end to use for the database.

`connection`

Type
string

Default
<None>

The SQLAlchemy connection string to use to connect to the database.

`slave_connection`

Type
string

Default
<None>

The SQLAlchemy connection string to use to connect to the slave database.

`asyncio_connection`

Type
string

Default
<None>

The SQLAlchemy asyncio connection string to use to connect to the database.

`asyncio_slave_connection`

Type
string

Default
<None>

The SQLAlchemy asyncio connection string to use to connect to the slave database.

`mysql_sql_mode`

Type
string

Default
TRADITIONAL

The SQL mode to be used for MySQL sessions. This option, including the default, overrides any server-set SQL mode. To use whatever SQL mode is set by the server configuration, set this to no value. Example: mysql_sql_mode=

`mysql_wsrep_sync_wait`

Type
integer

Default
<None>

For Galera only, configure wsrep_sync_wait causality checks on new connections. Default is None, meaning dont configure any setting.

`connection_recycle_time`

Type
integer

Default
3600

Connections which have been present in the connection pool longer than this number of seconds will be replaced with a new one the next time they are checked out from the pool.

`max_pool_size`

Type
integer

Default
5

Maximum number of SQL connections to keep open in a pool. Setting a value of 0 indicates no limit.

`max_retries`

Type
integer

Default
10

Maximum number of database connection retries during startup. Set to -1 to specify an infinite retry count.

`retry_interval`

Type
integer

Default
10

Interval between retries of opening a SQL connection.

`max_overflow`

Type
integer

Default
50

If set, use this value for max_overflow with SQLAlchemy.

`connection_debug`

Type
integer

Default
0

Minimum Value
0

Maximum Value
100

Verbosity of SQL debugging information: 0=None, 100=Everything.

`connection_trace`

Type
boolean

Default
`False`

Add Python stack traces to SQL as comment strings.

`pool_timeout`

Type
integer

Default
<None>

If set, use this value for pool_timeout with SQLAlchemy.

`use_db_reconnect`

Type
boolean

Default
`False`

Enable the experimental use of database reconnect on connection lost.

`db_retry_interval`

Type
integer

Default
1

Seconds between retries of a database transaction.

`db_inc_retry_interval`

Type
boolean

Default
`True`

If True, increases the interval between retries of a database operation up to db_max_retry_interval.

`db_max_retry_interval`

Type
integer

Default
10

If db_inc_retry_interval is set, the maximum seconds between retries of a database operation.

`db_max_retries`

Type
integer

Default
20

Maximum retries in case of connection error or deadlock error before error is raised. Set to -1 to specify an infinite retry count.

`connection_parameters`

Type
string

Default
''

Optional URL parameters to append onto the connection URL at connect time; specify as param1=value1&param2=value2&

6.2.7 glance_client

`api_version`

Type
string

Default
2

Version of Glance API to use in glanceclient.

`endpoint_type`

Type
string

Default
publicURL

Type of endpoint to use in glanceclient. Supported values: internalURL, publicURL, adminURL. The default is publicURL.

`region_name`

Type
string

Default
<None>

Region in Identity service catalog to use for communication with the OpenStack service.

6.2.8 gnocchi_client

api_version

Type
string

Default
1

Version of Gnocchi API to use in gnocchiclient.

endpoint_type

Type
string

Default
public

Type of endpoint to use in gnocchi client. Supported values: internal, public, admin. The default is public.

region_name

Type
string

Default
<None>

Region in Identity service catalog to use for communication with the OpenStack service.

6.2.9 grafana_client

See https://docs.openstack.org/watcher/latest/datasources/grafana.html for details on how these options are used.

token

Type
string

Default
<None>

Authentication token to gain access

base_url

Type
string

Default
<None>

First part of the url (including https:// or http://) up until project id part. Example: https://secure.org/api/datasource/proxy/

`project_id_map`

Type
dict

Default
```
{'host_cpu_usage': None, 'host_ram_usage': None,
'host_outlet_temp': None, 'host_inlet_temp':
None, 'host_airflow': None, 'host_power': None,
'instance_cpu_usage': None, 'instance_ram_usage': None,
'instance_ram_allocated': None, 'instance_l3_cache_usage':
None, 'instance_root_disk_size': None}
```

Mapping of datasource metrics to grafana project ids. Dictionary values should be positive integers. Example: 7465

`database_map`

Type
dict

Default
```
{'host_cpu_usage': None, 'host_ram_usage': None,
'host_outlet_temp': None, 'host_inlet_temp':
None, 'host_airflow': None, 'host_power': None,
'instance_cpu_usage': None, 'instance_ram_usage': None,
'instance_ram_allocated': None, 'instance_l3_cache_usage':
None, 'instance_root_disk_size': None}
```

Mapping of datasource metrics to grafana databases. Values should be strings. Example: influx_production

`attribute_map`

Type
dict

Default
```
{'host_cpu_usage': None, 'host_ram_usage': None,
'host_outlet_temp': None, 'host_inlet_temp':
None, 'host_airflow': None, 'host_power': None,
'instance_cpu_usage': None, 'instance_ram_usage': None,
'instance_ram_allocated': None, 'instance_l3_cache_usage':
None, 'instance_root_disk_size': None}
```

Mapping of datasource metrics to resource attributes. For a complete list of available attributes see https://docs.openstack.org/watcher/latest/datasources/grafana.html#attribute Values should be strings. Example: hostname

`translator_map`

Type
dict

Default
```
{'host_cpu_usage': None, 'host_ram_usage': None,
'host_outlet_temp': None, 'host_inlet_temp':
```

```
None, 'host_airflow': None, 'host_power': None,
'instance_cpu_usage': None, 'instance_ram_usage': None,
'instance_ram_allocated': None, 'instance_l3_cache_usage':
None, 'instance_root_disk_size': None}
```

Mapping of datasource metrics to grafana translators. Values should be strings. Example: influxdb

query_map

Type
dict

Default
```
{'host_cpu_usage': None, 'host_ram_usage': None,
'host_outlet_temp': None, 'host_inlet_temp':
None, 'host_airflow': None, 'host_power': None,
'instance_cpu_usage': None, 'instance_ram_usage': None,
'instance_ram_allocated': None, 'instance_l3_cache_usage':
None, 'instance_root_disk_size': None}
```

Mapping of datasource metrics to grafana queries. Values should be strings for which the .format method will transform it. The transformation offers five parameters to the query labeled {0} to {4}. {0} will be replaced with the aggregate, {1} with the resource attribute, {2} with the period, {3} with the granularity and {4} with translator specifics for InfluxDB this will be the retention period. These queries will need to be constructed using tools such as Postman. Example: SELECT cpu FROM {4}.cpu_percent WHERE host == {1} AND time > now()-{2}s

http_timeout

Type
integer

Default
60

Minimum Value
0

Mutable
This option can be changed without restarting.

Timeout for Grafana request

6.2.10 grafana_translators

retention_periods

Type
dict

Default
```
{'one_week': 604800, 'one_month': 2592000, 'five_years':
31556952}
```

Keys are the names of retention periods in InfluxDB and the values should correspond with the maximum time they can retain in seconds. Example: {one_day: 86400}

6.2.11 ironic_client

api_version

Type
string

Default
1

Version of Ironic API to use in ironicclient.

endpoint_type

Type
string

Default
publicURL

Type of endpoint to use in ironicclient. Supported values: internalURL, publicURL, adminURL. The default is publicURL.

region_name

Type
string

Default
<None>

Region in Identity service catalog to use for communication with the OpenStack service.

6.2.12 keystone_authtoken

www_authenticate_uri

Type
string

Default
<None>

Complete public Identity API endpoint. This endpoint should not be an admin endpoint, as it should be accessible by all end users. Unauthenticated clients are redirected to this endpoint to authenticate. Although this endpoint should ideally be unversioned, client support in the wild varies. If youre using a versioned v2 endpoint here, then this should *not* be the same endpoint the service user utilizes for validating tokens, because normal end users may not be able to reach that endpoint.

Table 5: Deprecated Variations

Group	Name
keystone_authtoken	auth_uri

auth_uri

Type
string

Default
<None>

Complete public Identity API endpoint. This endpoint should not be an admin endpoint, as it should be accessible by all end users. Unauthenticated clients are redirected to this endpoint to authenticate. Although this endpoint should ideally be unversioned, client support in the wild varies. If youre using a versioned v2 endpoint here, then this should *not* be the same endpoint the service user utilizes for validating tokens, because normal end users may not be able to reach that endpoint. This option is deprecated in favor of www_authenticate_uri and will be removed in the S release.

Warning

This option is deprecated for removal since Queens. Its value may be silently ignored in the future.

Reason
The auth_uri option is deprecated in favor of www_authenticate_uri and will be removed in the S release.

`auth_version`

Type
string

Default
<None>

API version of the Identity API endpoint.

`interface`

Type
string

Default
`internal`

Interface to use for the Identity API endpoint. Valid values are public, internal (default) or admin.

`delay_auth_decision`

Type
boolean

Default
`False`

Do not handle authorization requests within the middleware, but delegate the authorization decision to downstream WSGI components.

`http_connect_timeout`

Type
integer

Default
<None>

Request timeout value for communicating with Identity API server.

`http_request_max_retries`

Type
integer

Default
3

How many times are we trying to reconnect when communicating with Identity API Server.

`cache`

Type
string

Default
<None>

Request environment key where the Swift cache object is stored. When auth_token middleware is deployed with a Swift cache, use this option to have the middleware share a caching backend with swift. Otherwise, use the `memcached_servers` option instead.

`certfile`

Type
string

Default
<None>

Required if identity server requires client certificate

`keyfile`

Type
string

Default
<None>

Required if identity server requires client certificate

`cafile`

Type
string

Default
<None>

A PEM encoded Certificate Authority to use when verifying HTTPs connections. Defaults to system CAs.

`insecure`

Type
boolean

Default
`False`

Verify HTTPS connections.

`region_name`

Type
string

Default
`<None>`

The region in which the identity server can be found.

`memcached_servers`

Type
list

Default
`<None>`

Optionally specify a list of memcached server(s) to use for caching. If left undefined, tokens will instead be cached in-process.

Table 6: Deprecated Variations

Group	Name
keystone_authtoken	memcache_servers

`token_cache_time`

Type
integer

Default
`300`

In order to prevent excessive effort spent validating tokens, the middleware caches previously-seen tokens for a configurable duration (in seconds). Set to -1 to disable caching completely.

`memcache_security_strategy`

Type
string

Default
`None`

Valid Values
None, MAC, ENCRYPT

(Optional) If defined, indicate whether token data should be authenticated or authenticated and encrypted. If MAC, token data is authenticated (with HMAC) in the cache. If ENCRYPT, token data is encrypted and authenticated in the cache. If the value is not one of these options or empty, auth_token will raise an exception on initialization.

`memcache_secret_key`

Type
string

Default
<None>

(Optional, mandatory if memcache_security_strategy is defined) This string is used for key derivation.

`memcache_pool_dead_retry`

Type
integer

Default
300

(Optional) Number of seconds memcached server is considered dead before it is tried again.

`memcache_pool_maxsize`

Type
integer

Default
10

(Optional) Maximum total number of open connections to every memcached server.

`memcache_pool_socket_timeout`

Type
integer

Default
3

(Optional) Socket timeout in seconds for communicating with a memcached server.

`memcache_pool_unused_timeout`

Type
integer

Default
60

(Optional) Number of seconds a connection to memcached is held unused in the pool before it is closed.

`memcache_pool_conn_get_timeout`

Type
integer

Default
10

(Optional) Number of seconds that an operation will wait to get a memcached client connection from the pool.

`memcache_use_advanced_pool`

Type
boolean

Default
`True`

(Optional) Use the advanced (eventlet safe) memcached client pool.

`include_service_catalog`

Type
boolean

Default
`True`

(Optional) Indicate whether to set the X-Service-Catalog header. If False, middleware will not ask for service catalog on token validation and will not set the X-Service-Catalog header.

`enforce_token_bind`

Type
string

Default
`permissive`

Used to control the use and type of token binding. Can be set to: disabled to not check token binding. permissive (default) to validate binding information if the bind type is of a form known to the server and ignore it if not. strict like permissive but if the bind type is unknown the token will be rejected. required any form of token binding is needed to be allowed. Finally the name of a binding method that must be present in tokens.

`service_token_roles`

Type
list

Default
`['service']`

A choice of roles that must be present in a service token. Service tokens are allowed to request that an expired token can be used and so this check should tightly control that only actual services should be sending this token. Roles here are applied as an ANY check so any role in this list must be present. For backwards compatibility reasons this currently only affects the allow_expired check.

`service_token_roles_required`

Type
boolean

Default
`False`

For backwards compatibility reasons we must let valid service tokens pass that dont pass the service_token_roles check as valid. Setting this true will become the default in a future release and should be enabled if possible.

`service_type`

Type
string

Default
<None>

The name or type of the service as it appears in the service catalog. This is used to validate tokens that have restricted access rules.

`memcache_sasl_enabled`

Type
boolean

Default
`False`

Enable the SASL(Simple Authentication and Security Layer) if the SASL_enable is true, else disable.

`memcache_username`

Type
string

Default
`''`

the user name for the SASL

`memcache_password`

Type
string

Default
`''`

the username password for SASL

`auth_type`

Type
unknown type

Default
<None>

Authentication type to load

Table 7: Deprecated Variations

Group	Name
keystone_authtoken	auth_plugin

auth_section

Type
unknown type

Default
<None>

Config Section from which to load plugin specific options

6.2.13 keystone_client

interface

Type
string

Default
admin

Valid Values
internal, public, admin

Type of endpoint to use in keystoneclient.

region_name

Type
string

Default
<None>

Region in Identity service catalog to use for communication with the OpenStack service.

6.2.14 maas_client

url

Type
string

Default
<None>

MaaS URL, example: http://1.2.3.4:5240/MAAS

api_key

Type
string

Default
<None>

MaaS API authentication key.

timeout

Type
integer

Default
60

MaaS client operation timeout in seconds.

6.2.15 monasca_client

api_version

Type
string

Default
2_0

Version of Monasca API to use in monascaclient.

interface

Type
string

Default
internal

Type of interface used for monasca endpoint. Supported values: internal, public, admin. The default is internal.

region_name

Type
string

Default
<None>

Region in Identity service catalog to use for communication with the OpenStack service.

6.2.16 neutron_client

api_version

Type
string

Default
2.0

Version of Neutron API to use in neutronclient.

endpoint_type

Type
string

Default
publicURL

Type of endpoint to use in neutronclient. Supported values: internalURL, publicURL, adminURL. The default is publicURL.

region_name

Type
string

Default
<None>

Region in Identity service catalog to use for communication with the OpenStack service.

6.2.17 nova_client

api_version

Type
string

Default
2.56

Version of Nova API to use in novaclient.

Minimum required version: 2.56

Certain Watcher features depend on a minimum version of the compute API being available which is enforced with this option. See https://docs.openstack.org/nova/latest/reference/api-microversion-history.html for the compute API microversion history.

endpoint_type

Type
string

Default
publicURL

Type of endpoint to use in novaclient. Supported values: internalURL, publicURL, adminURL. The default is publicURL.

region_name

Type
string

Default
<None>

Region in Identity service catalog to use for communication with the OpenStack service.

6.2.18 oslo_concurrency

disable_process_locking

Type
boolean

Default
False

Enables or disables inter-process locks.

`lock_path`

Type
string

Default
<None>

Directory to use for lock files. For security, the specified directory should only be writable by the user running the processes that need locking. Defaults to environment variable OSLO_LOCK_PATH. If external locks are used, a lock path must be set.

6.2.19 oslo_messaging_kafka

`kafka_max_fetch_bytes`

Type
integer

Default
`1048576`

Max fetch bytes of Kafka consumer

`kafka_consumer_timeout`

Type
floating point

Default
`1.0`

Default timeout(s) for Kafka consumers

`consumer_group`

Type
string

Default
`oslo_messaging_consumer`

Group id for Kafka consumer. Consumers in one group will coordinate message consumption

`producer_batch_timeout`

Type
floating point

Default
`0.0`

Upper bound on the delay for KafkaProducer batching in seconds

`producer_batch_size`

Type
integer

Default
`16384`

Size of batch for the producer async send

`compression_codec`

Type
string

Default
`none`

Valid Values
none, gzip, snappy, lz4, zstd

The compression codec for all data generated by the producer. If not set, compression will not be used. Note that the allowed values of this depend on the kafka version

`enable_auto_commit`

Type
boolean

Default
`False`

Enable asynchronous consumer commits

`max_poll_records`

Type
integer

Default
`500`

The maximum number of records returned in a poll call

`security_protocol`

Type
string

Default
`PLAINTEXT`

Valid Values
PLAINTEXT, SASL_PLAINTEXT, SSL, SASL_SSL

Protocol used to communicate with brokers

`sasl_mechanism`

Type
string

Default
`PLAIN`

Mechanism when security protocol is SASL

`ssl_cafile`

Type
string

Default
''

CA certificate PEM file used to verify the server certificate

`ssl_client_cert_file`

Type
string

Default
''

Client certificate PEM file used for authentication.

`ssl_client_key_file`

Type
string

Default
''

Client key PEM file used for authentication.

`ssl_client_key_password`

Type
string

Default
''

Client key password file used for authentication.

6.2.20 oslo_messaging_notifications

`driver`

Type
multi-valued

Default
''

The Drivers(s) to handle sending notifications. Possible values are messaging, messagingv2, routing, log, test, noop

`transport_url`

Type
string

Default
<None>

A URL representing the messaging driver to use for notifications. If not set, we fall back to the same configuration used for RPC.

`topics`

Type
list

Default
`['notifications']`

AMQP topic used for OpenStack notifications.

`retry`

Type
integer

Default
`-1`

The maximum number of attempts to re-send a notification message which failed to be delivered due to a recoverable error. 0 - No retry, -1 - indefinite

6.2.21 oslo_messaging_rabbit

`amqp_durable_queues`

Type
boolean

Default
`False`

Use durable queues in AMQP. If rabbit_quorum_queue is enabled, queues will be durable and this value will be ignored.

`amqp_auto_delete`

Type
boolean

Default
`False`

Auto-delete queues in AMQP.

`rpc_conn_pool_size`

Type
integer

Default
`30`

Minimum Value
1

Size of RPC connection pool.

`conn_pool_min_size`

Type
integer

Default
2

The pool size limit for connections expiration policy

`conn_pool_ttl`

Type
integer

Default
1200

The time-to-live in sec of idle connections in the pool

`ssl`

Type
boolean

Default
False

Connect over SSL.

`ssl_version`

Type
string

Default
''

SSL version to use (valid only if SSL enabled). Valid values are TLSv1 and SSLv23. SSLv2, SSLv3, TLSv1_1, and TLSv1_2 may be available on some distributions.

`ssl_key_file`

Type
string

Default
''

SSL key file (valid only if SSL enabled).

`ssl_cert_file`

Type
string

Default
''

SSL cert file (valid only if SSL enabled).

`ssl_ca_file`

Type
string

Default
''

SSL certification authority file (valid only if SSL enabled).

`ssl_enforce_fips_mode`

Type
boolean

Default
`False`

Global toggle for enforcing the OpenSSL FIPS mode. This feature requires Python support. This is available in Python 3.9 in all environments and may have been backported to older Python versions on select environments. If the Python executable used does not support OpenSSL FIPS mode, an exception will be raised.

`heartbeat_in_pthread`

Type
boolean

Default
`False`

(DEPRECATED) It is recommend not to use this option anymore. Run the health check heartbeat thread through a native python thread by default. If this option is equal to False then the health check heartbeat will inherit the execution model from the parent process. For example if the parent process has monkey patched the stdlib by using eventlet/greenlet then the heartbeat will be run through a green thread. This option should be set to True only for the wsgi services.

Warning

This option is deprecated for removal. Its value may be silently ignored in the future.

Reason
The option is related to Eventlet which will be removed. In addition this has never worked as expected with services using eventlet for core service framework.

`kombu_reconnect_delay`

Type
floating point

Default
`1.0`

Minimum Value
0.0

Maximum Value
4.5

How long to wait (in seconds) before reconnecting in response to an AMQP consumer cancel notification.

`kombu_reconnect_splay`

Type
floating point

Default
`0.0`

Minimum Value
0.0

Random time to wait for when reconnecting in response to an AMQP consumer cancel notification.

`kombu_compression`

Type
string

Default
<None>

EXPERIMENTAL: Possible values are: gzip, bz2. If not set compression will not be used. This option may not be available in future versions.

`kombu_missing_consumer_retry_timeout`

Type
integer

Default
`60`

How long to wait a missing client before abandoning to send it its replies. This value should not be longer than rpc_response_timeout.

Table 8: Deprecated Variations

Group	Name
oslo_messaging_rabbit	kombu_reconnect_timeout

`kombu_failover_strategy`

Type
string

Default
`round-robin`

Valid Values
round-robin, shuffle

Determines how the next RabbitMQ node is chosen in case the one we are currently connected to becomes unavailable. Takes effect only if more than one RabbitMQ node is provided in config.

`rabbit_login_method`

Type
string

Default
`AMQPLAIN`

Valid Values
PLAIN, AMQPLAIN, EXTERNAL, RABBIT-CR-DEMO

The RabbitMQ login method.

`rabbit_retry_interval`

Type
integer

Default
`1`

Minimum Value
1

How frequently to retry connecting with RabbitMQ.

`rabbit_retry_backoff`

Type
integer

Default
`2`

Minimum Value
0

How long to backoff for between retries when connecting to RabbitMQ.

`rabbit_interval_max`

Type
integer

Default
`30`

Minimum Value
1

Maximum interval of RabbitMQ connection retries.

`rabbit_ha_queues`

Type
boolean

Default
`False`

Try to use HA queues in RabbitMQ (x-ha-policy: all). If you change this option, you must wipe the RabbitMQ database. In RabbitMQ 3.0, queue mirroring is no longer controlled by the x-ha-policy argument when declaring a queue. If you just want to make sure that all queues (except those with auto-generated names) are mirrored across all nodes, run: rabbitmqctl set_policy HA ^(?!amq.).* {ha-mode: all}

`rabbit_quorum_queue`

Type
boolean

Default
`False`

Use quorum queues in RabbitMQ (x-queue-type: quorum). The quorum queue is a modern queue type for RabbitMQ implementing a durable, replicated FIFO queue based on the Raft consensus algorithm. It is available as of RabbitMQ 3.8.0. If set this option will conflict with the HA queues (`rabbit_ha_queues`) aka mirrored queues, in other words the HA queues should be disabled. Quorum queues are also durable by default so the amqp_durable_queues option is ignored when this option is enabled.

`rabbit_transient_quorum_queue`

Type
boolean

Default
`False`

Use quorum queues for transients queues in RabbitMQ. Enabling this option will then make sure those queues are also using quorum kind of rabbit queues, which are HA by default.

`rabbit_quorum_delivery_limit`

Type
integer

Default
`0`

Each time a message is redelivered to a consumer, a counter is incremented. Once the redelivery count exceeds the delivery limit the message gets dropped or dead-lettered (if a DLX exchange has been configured) Used only when rabbit_quorum_queue is enabled, Default 0 which means dont set a limit.

`rabbit_quorum_max_memory_length`

Type
integer

Default
`0`

By default all messages are maintained in memory if a quorum queue grows in length it can put memory pressure on a cluster. This option can limit the number of messages in the quorum queue. Used only when rabbit_quorum_queue is enabled, Default 0 which means dont set a limit.

Table 9: Deprecated Variations

Group	Name
oslo_messaging_rabbit	rabbit_quroum_max_memory_length

rabbit_quorum_max_memory_bytes

Type
integer

Default
0

By default all messages are maintained in memory if a quorum queue grows in length it can put memory pressure on a cluster. This option can limit the number of memory bytes used by the quorum queue. Used only when rabbit_quorum_queue is enabled, Default 0 which means dont set a limit.

Table 10: Deprecated Variations

Group	Name
oslo_messaging_rabbit	rabbit_quroum_max_memory_bytes

rabbit_transient_queues_ttl

Type
integer

Default
1800

Minimum Value
0

Positive integer representing duration in seconds for queue TTL (x-expires). Queues which are unused for the duration of the TTL are automatically deleted. The parameter affects only reply and fanout queues. Setting 0 as value will disable the x-expires. If doing so, make sure you have a rabbitmq policy to delete the queues or you deployment will create an infinite number of queue over time.In case rabbit_stream_fanout is set to True, this option will control data retention policy (x-max-age) for messages in the fanout queue rather then the queue duration itself. So the oldest data in the stream queue will be discarded from it once reaching TTL Setting to 0 will disable x-max-age for stream which make stream grow indefinitely filling up the diskspace

rabbit_qos_prefetch_count

Type
integer

Default
0

Specifies the number of messages to prefetch. Setting to zero allows unlimited messages.

heartbeat_timeout_threshold

Type
integer

Default
`60`

Number of seconds after which the Rabbit broker is considered down if heartbeats keep-alive fails (0 disables heartbeat).

`heartbeat_rate`

Type
integer

Default
`3`

How often times during the heartbeat_timeout_threshold we check the heartbeat.

`direct_mandatory_flag`

Type
boolean

Default
`True`

(DEPRECATED) Enable/Disable the RabbitMQ mandatory flag for direct send. The direct send is used as reply, so the MessageUndeliverable exception is raised in case the client queue does not exist.MessageUndeliverable exception will be used to loop for a timeout to lets a chance to sender to recover.This flag is deprecated and it will not be possible to deactivate this functionality anymore

Warning

This option is deprecated for removal. Its value may be silently ignored in the future.

Reason
Mandatory flag no longer deactivable.

`enable_cancel_on_failover`

Type
boolean

Default
`False`

Enable x-cancel-on-ha-failover flag so that rabbitmq server will cancel and notify consumerswhen queue is down

`use_queue_manager`

Type
boolean

Default
`False`

Should we use consistant queue names or random ones

hostname

Type
string

Default
`node1.example.com`

This option has a sample default set, which means that its actual default value may vary from the one documented above.

Hostname used by queue manager. Defaults to the value returned by socket.gethostname().

processname

Type
string

Default
`nova-api`

This option has a sample default set, which means that its actual default value may vary from the one documented above.

Process name used by queue manager

rabbit_stream_fanout

Type
boolean

Default
`False`

Use stream queues in RabbitMQ (x-queue-type: stream). Streams are a new persistent and replicated data structure (queue type) in RabbitMQ which models an append-only log with non-destructive consumer semantics. It is available as of RabbitMQ 3.9.0. If set this option will replace all fanout queues with only one stream queue.

6.2.22 oslo_policy

enforce_scope

Type
boolean

Default
`True`

This option controls whether or not to enforce scope when evaluating policies. If `True`, the scope of the token used in the request is compared to the `scope_types` of the policy being enforced. If the scopes do not match, an `InvalidScope` exception will be raised. If `False`, a message will be logged informing operators that policies are being invoked with mismatching scope.

Warning

This option is deprecated for removal. Its value may be silently ignored in the future.

Reason
This configuration was added temporarily to facilitate a smooth transition to the new RBAC. OpenStack will always enforce scope checks. This configuration option is deprecated and will be removed in the 2025.2 cycle.

`enforce_new_defaults`

Type
boolean

Default
`True`

This option controls whether or not to use old deprecated defaults when evaluating policies. If `True`, the old deprecated defaults are not going to be evaluated. This means if any existing token is allowed for old defaults but is disallowed for new defaults, it will be disallowed. It is encouraged to enable this flag along with the `enforce_scope` flag so that you can get the benefits of new defaults and `scope_type` together. If `False`, the deprecated policy check string is logically ORd with the new policy check string, allowing for a graceful upgrade experience between releases with new policies, which is the default behavior.

`policy_file`

Type
string

Default
`policy.yaml`

The relative or absolute path of a file that maps roles to permissions for a given service. Relative paths must be specified in relation to the configuration file setting this option.

`policy_default_rule`

Type
string

Default
`default`

Default rule. Enforced when a requested rule is not found.

`policy_dirs`

Type
multi-valued

Default
`policy.d`

Directories where policy configuration files are stored. They can be relative to any directory in the search path defined by the config_dir option, or absolute paths. The file defined by policy_file must exist for these directories to be searched. Missing or empty directories are ignored.

`remote_content_type`

Type
string

Default
`application/x-www-form-urlencoded`

Valid Values
application/x-www-form-urlencoded, application/json

Content Type to send and receive data for REST based policy check

`remote_ssl_verify_server_crt`

Type
boolean

Default
`False`

server identity verification for REST based policy check

`remote_ssl_ca_crt_file`

Type
string

Default
<None>

Absolute path to ca cert file for REST based policy check

`remote_ssl_client_crt_file`

Type
string

Default
<None>

Absolute path to client cert for REST based policy check

`remote_ssl_client_key_file`

Type
string

Default
<None>

Absolute path client key file REST based policy check

`remote_timeout`

Type
floating point

Default
`60`

Minimum Value
0

Timeout in seconds for REST based policy check

6.2.23 oslo_reports

`log_dir`

Type
string

Default
<None>

Path to a log directory where to create a file

`file_event_handler`

Type
string

Default
<None>

The path to a file to watch for changes to trigger the reports, instead of signals. Setting this option disables the signal trigger for the reports. If application is running as a WSGI application it is recommended to use this instead of signals.

`file_event_handler_interval`

Type
integer

Default
1

How many seconds to wait between polls when file_event_handler is set

6.2.24 placement_client

Configuration options for connecting to the placement API service

`api_version`

Type
string

Default
1.29

microversion of placement API when using placement service.

`interface`

Type
string

Default
public

Valid Values
internal, public, admin

Type of endpoint when using placement service.

`region_name`

Type
string

Default
<None>

Region in Identity service catalog to use for communication with the OpenStack service.

6.2.25 prometheus_client

See https://docs.openstack.org/watcher/latest/datasources/prometheus.html for details on how these options are used.

`host`

Type
string

Default
<None>

The hostname or IP address for the prometheus server.

`port`

Type
string

Default
<None>

The port number used by the prometheus server.

`fqdn_label`

Type
string

Default
fqdn

The label that Prometheus uses to store the fqdn of exporters. Defaults to fqdn.

`instance_uuid_label`

Type
string

Default
resource

The label that Prometheus uses to store the uuid of OpenStack instances. Defaults to resource.

`username`

Type
string

Default
<None>

The basic_auth username to use to authenticate with the Prometheus server.

`password`

Type
string

Default
<None>

The basic_auth password to use to authenticate with the Prometheus server.

`cafile`

Type
string

Default
<None>

Path to the CA certificate for establishing a TLS connection with the Prometheus server.

`certfile`

Type
string

Default
<None>

Path to the client certificate for establishing a TLS connection with the Prometheus server.

`keyfile`

Type
string

Default
<None>

Path to the client key for establishing a TLS connection with the Prometheus server.

6.2.26 watcher_applier

`workers`

Type
integer

Default
1

Minimum Value
1

Number of workers for applier, default value is 1.

conductor_topic

Type
string

Default
watcher.applier.control

The topic name used for control events, this topic used for rpc call

publisher_id

Type
string

Default
watcher.applier.api

The identifier used by watcher module on the message broker

workflow_engine

Type
string

Default
taskflow

Select the engine to use to execute the workflow

rollback_when_actionplan_failed

Type
boolean

Default
False

If set True, the failed actionplan will rollback when executing. Default value is False.

6.2.27 watcher_clients_auth

cafile

Type
string

Default
<None>

PEM encoded Certificate Authority to use when verifying HTTPs connections.

certfile

Type
string

Default
<None>

PEM encoded client certificate cert file

`keyfile`

Type
string

Default
<None>

PEM encoded client certificate key file

`insecure`

Type
boolean

Default
`False`

Verify HTTPS connections.

`timeout`

Type
integer

Default
<None>

Timeout value for http requests

`collect_timing`

Type
boolean

Default
`False`

Collect per-API call timing information.

`split_loggers`

Type
boolean

Default
`False`

Log requests to multiple loggers.

`auth_type`

Type
unknown type

Default
<None>

Authentication type to load

Table 11: Deprecated Variations

Group	Name
watcher_clients_auth	auth_plugin

auth_section

Type
unknown type

Default
<None>

Config Section from which to load plugin specific options

6.2.28 watcher_cluster_data_model_collectors.baremetal

period

Type
integer

Default
3600

The time interval (in seconds) between each synchronization of the model

6.2.29 watcher_cluster_data_model_collectors.compute

period

Type
integer

Default
3600

The time interval (in seconds) between each synchronization of the model

6.2.30 watcher_cluster_data_model_collectors.storage

period

Type
integer

Default
3600

The time interval (in seconds) between each synchronization of the model

6.2.31 watcher_datasources

datasources

Type
list

Default
`['gnocchi', 'monasca', 'grafana', 'prometheus']`

Datasources to use in order to query the needed metrics. If one of strategy metric is not available in the first datasource, the next datasource will be chosen. This is the default for all strategies unless a strategy has a specific override.

`query_max_retries`

Type
integer

Default
`10`

Minimum Value
1

Mutable
This option can be changed without restarting.

How many times Watcher is trying to query again

Table 12: Deprecated Variations

Group	Name
gnocchi_client	query_max_retries

`query_timeout`

Type
integer

Default
1

Minimum Value
0

Mutable
This option can be changed without restarting.

How many seconds Watcher should wait to do query again

Table 13: Deprecated Variations

Group	Name
gnocchi_client	query_timeout

6.2.32 watcher_decision_engine

`conductor_topic`

Type
string

Default
watcher.decision.control

The topic name used for control events, this topic used for RPC calls

`notification_topics`

Type
list

Default
['nova.versioned_notifications', 'watcher.watcher_notifications']

The exchange and topic names from which notification events will be listened to. The exchange should be specified to get an ability to use pools.

`publisher_id`

Type
string

Default
watcher.decision.api

The identifier used by the Watcher module on the message broker

`max_audit_workers`

Type
integer

Default
2

The maximum number of threads that can be used to execute audits in parallel.

`max_general_workers`

Type
integer

Default
4

The maximum number of threads that can be used to execute general tasks in parallel. The number of general workers will not increase depending on the number of audit workers!

`action_plan_expiry`

Type
integer

Default
24

Mutable
This option can be changed without restarting.

An expiry timespan(hours). Watcher invalidates any action plan for which its creation time -whose number of hours has been offset by this value- is older that the current time.

`check_periodic_interval`

Type
integer

Default
`1800`

Mutable
This option can be changed without restarting.

Interval (in seconds) for checking action plan expiry.

`metric_map_path`

Type
string

Default
`/etc/watcher/metric_map.yaml`

Path to metric map yaml formatted file. The file contains a map per datasource whose keys are the metric names as recognized by watcher and the value is the real name of the metric in the datasource. For example:

```
monasca:
  instance_cpu_usage: VM_CPU
gnocchi:
  instance_cpu_usage: cpu_vm_util
```

This file is optional.

`continuous_audit_interval`

Type
integer

Default
`10`

Mutable
This option can be changed without restarting.

Interval (in seconds) for checking newly created continuous audits.

6.2.33 watcher_planner

`planner`

Type
string

Default
`weight`

The selected planner used to schedule the actions

6.2.34 watcher_planners.weight

weights

Type
dict

Default
```
{'nop': 70, 'volume_migrate': 60, 'change_nova_service_state':
50, 'sleep': 40, 'migrate': 30, 'resize': 20,
'turn_host_to_acpi_s3_state': 10, 'change_node_power_state':
9}
```

These weights are used to schedule the actions. Action Plan will be build in accordance with sets of actions ordered by descending weights.Two action types cannot have the same weight.

parallelization

Type
dict

Default
```
{'turn_host_to_acpi_s3_state': 2, 'resize': 2, 'migrate':
2, 'sleep': 1, 'change_nova_service_state': 1, 'nop': 1,
'change_node_power_state': 2, 'volume_migrate': 2}
```

Number of actions to be run in parallel on a per action type basis.

6.2.35 watcher_planners.workload_stabilization

weights

Type
dict

Default
```
{'turn_host_to_acpi_s3_state': 0, 'resize': 1, 'migrate':
2, 'sleep': 3, 'change_nova_service_state': 4, 'nop': 5}
```

These weights are used to schedule the actions

6.2.36 watcher_strategies.basic

datasources

Type
list

Default
<None>

Datasources to use in order to query the needed metrics. This option overrides the global preference. options: [gnocchi, monasca, grafana, prometheus]

check_optimize_metadata

Type
boolean

Default
`False`

Check optimize metadata field in instance before migration

6.2.37 watcher_strategies.node_resource_consolidation

datasources

Type
list

Default
<None>

Datasources to use in order to query the needed metrics. This option overrides the global preference. options: [gnocchi, monasca, grafana, prometheus]

6.2.38 watcher_strategies.noisy_neighbor

datasources

Type
list

Default
<None>

Datasources to use in order to query the needed metrics. This option overrides the global preference. options: [gnocchi, monasca, grafana, prometheus]

6.2.39 watcher_strategies.outlet_temperature

datasources

Type
list

Default
<None>

Datasources to use in order to query the needed metrics. This option overrides the global preference. options: [gnocchi, monasca, grafana, prometheus]

6.2.40 watcher_strategies.storage_capacity_balance

datasources

Type
list

Default
<None>

Datasources to use in order to query the needed metrics. This option overrides the global preference. options: [gnocchi, monasca, grafana, prometheus]

ex_pools

Type
list

Default
['local_vstorage']

exclude pools

6.2.41 watcher_strategies.uniform_airflow

datasources

Type
list

Default
<None>

Datasources to use in order to query the needed metrics. This option overrides the global preference. options: [gnocchi, monasca, grafana, prometheus]

6.2.42 watcher_strategies.vm_workload_consolidation

datasources

Type
list

Default
<None>

Datasources to use in order to query the needed metrics. This option overrides the global preference. options: [gnocchi, monasca, grafana, prometheus]

6.2.43 watcher_strategies.workload_balance

datasources

Type
list

Default
<None>

Datasources to use in order to query the needed metrics. This option overrides the global preference. options: [gnocchi, monasca, grafana, prometheus]

6.2.44 watcher_strategies.workload_stabilization

datasources

Type
list

Default
<None>

Datasources to use in order to query the needed metrics. This option overrides the global preference. options: [gnocchi, monasca, grafana, prometheus]

6.2.45 watcher_workflow_engines.taskflow

`max_workers`

Type
integer

Default
4

Minimum Value
1

Number of workers for taskflow engine to execute actions.

`action_execution_rule`

Type
dict

Default
{}

The execution rule for linked actions,the key is strategy name and value ALWAYS means all actions will be executed,value ANY means if previous action executes success, the next action will be ignored.None means ALWAYS.

CHAPTER

SEVEN

PLUGIN GUIDE

7.1 Create a third-party plugin for Watcher

Watcher provides a plugin architecture which allows anyone to extend the existing functionalities by implementing third-party plugins. This process can be cumbersome so this documentation is there to help you get going as quickly as possible.

7.1.1 Pre-requisites

We assume that you have set up a working Watcher development environment. So if this not already the case, you can check out our documentation which explains how to set up a *development environment*.

7.1.2 Third party project scaffolding

First off, we need to create the project structure. To do so, we can use cookiecutter and the OpenStack cookiecutter project scaffolder to generate the skeleton of our project:

```
$ virtualenv thirdparty
$ . thirdparty/bin/activate
$ pip install cookiecutter
$ cookiecutter https://github.com/openstack-dev/cookiecutter
```

The last command will ask you for many information, and If you set `module_name` and `repo_name` as `thirdparty`, you should end up with a structure that looks like this:

```
$ cd thirdparty
$ tree .
.
 babel.cfg
 CONTRIBUTING.rst
 doc
ăă  source
ăă      conf.py
ăă      contributing.rst
ăă      index.rst
ăă      installation.rst
ăă      readme.rst
ăă      usage.rst
 HACKING.rst
 LICENSE
 MANIFEST.in
```

(continues on next page)

(continued from previous page)

```
 README.rst
 requirements.txt
 setup.cfg
 setup.py
 test-requirements.txt
 thirdparty
ăă  __init__.py
ăă  tests
ăă      base.py
ăă      __init__.py
ăă      test_thirdparty.py
 tox.ini
```

Note: You should add python-watcher as a dependency in the requirements.txt file:

```
# Watcher-specific requirements
python-watcher
```

7.1.3 Implementing a plugin for Watcher

Now that the project skeleton has been created, you can start the implementation of your plugin. As of now, you can implement the following plugins for Watcher:

- A *goal plugin*
- A *strategy plugin*
- An *action plugin*
- A *planner plugin*
- A workflow engine plugin
- A *cluster data model collector plugin*

If you want to learn more on how to implement them, you can refer to their dedicated documentation.

7.2 Build a new action

Watcher Applier has an external *action* plugin interface which gives anyone the ability to integrate an external *action* in order to extend the initial set of actions Watcher provides.

This section gives some guidelines on how to implement and integrate custom actions with Watcher.

7.2.1 Creating a new plugin

First of all you have to extend the base `BaseAction` class which defines a set of abstract methods and/or properties that you will have to implement:

- The `schema` is an abstract property that you have to implement. This is the first function to be called by the *applier* before any further processing and its role is to validate the input parameters that were provided to it.
- The `pre_condition()` is called before the execution of an action. This method is a hook that can be used to perform some initializations or to make some more advanced validation on its input

parameters. If you wish to block the execution based on this factor, you simply have to `raise` an exception.

- The `post_condition()` is called after the execution of an action. As this function is called regardless of whether an action succeeded or not, this can prove itself useful to perform cleanup operations.
- The `execute()` is the main component of an action. This is where you should implement the logic of your action.
- The `revert()` allows you to roll back the targeted resource to its original state following a faulty execution. Indeed, this method is called by the workflow engine whenever an action raises an exception.

Here is an example showing how you can write a plugin called `DummyAction`:

```
# Filepath = <PROJECT_DIR>/thirdparty/dummy.py
# Import path = thirdparty.dummy
import voluptuous

from watcher.applier.actions import base


class DummyAction(base.BaseAction):

    @property
    def schema(self):
        return voluptuous.Schema({})

    def execute(self):
        # Does nothing
        pass  # Only returning False is considered as a failure

    def revert(self):
        # Does nothing
        pass

    def pre_condition(self):
        # No pre-checks are done here
        pass

    def post_condition(self):
        # Nothing done here
        pass
```

This implementation is the most basic one. So in order to get a better understanding on how to implement a more advanced action, have a look at the `Migrate` class.

Input validation

As you can see in the previous example, we are using Voluptuous to validate the input parameters of an action. So if you want to learn more about how to work with Voluptuous, you can have a look at their documentation:

7.2.2 Define configuration parameters

At this point, you have a fully functional action. However, in more complex implementation, you may want to define some configuration options so one can tune the action to its needs. To do so, you can implement the `get_config_opts()` class method as followed:

```
from oslo_config import cfg

class DummyAction(base.BaseAction):

    # [...]

    def execute(self):
        assert self.config.test_opt == 0

    @classmethod
    def get_config_opts(cls):
        return super(
            DummyAction, cls).get_config_opts() + [
            cfg.StrOpt('test_opt', help="Demo Option.", default=0),
            # Some more options ...
        ]
```

The configuration options defined within this class method will be included within the global `watcher.conf` configuration file under a section named by convention: `{namespace}.{plugin_name}`. In our case, the `watcher.conf` configuration would have to be modified as followed:

```
[watcher_actions.dummy]
# Option used for testing.
test_opt = test_value
```

Then, the configuration options you define within this method will then be injected in each instantiated object via the `config` parameter of the `__init__()` method.

7.2.3 Abstract Plugin Class

Here below is the abstract `BaseAction` class that every single action should implement:

class watcher.applier.actions.base.**BaseAction**(*config*, *osc=None*)

schema

Defines a Schema that the input parameters shall comply to

Returns

A schema declaring the input parameters this action should be provided along with their respective constraints (e.g. type, value range,)

Return type

`voluptuous.Schema` instance

__init__(*config*, *osc=None*)

Constructor

Parameters

- **config** (*dict*) A mapping containing the configuration of this action

- **osc** (OpenStackClients instance, optional) an OpenStackClients instance, defaults to None

abstract execute()

Executes the main logic of the action

This method can be used to perform an action on a given set of input parameters to accomplish some type of operation. This operation may return a boolean value as a result of its execution. If False, this will be considered as an error and will then trigger the reverting of the actions.

Returns
A flag indicating whether or not the action succeeded

Return type
bool

classmethod get_config_opts()

Defines the configuration options to be associated to this loadable

Returns
A list of configuration options relative to this Loadable

Return type
list of oslo_config.cfg.Opt instances

abstract get_description()

Description of the action

abstract post_condition()

Hook: called after the execution of an action

This function is called regardless of whether an action succeeded or not. So you can use it to perform cleanup operations.

abstract pre_condition()

Hook: called before the execution of an action

This method can be used to perform some initializations or to make some more advanced validation on its input parameters. So if you wish to block its execution based on this factor, *raise* the related exception.

abstract revert()

Revert this action

This method should rollback the resource to its initial state in the event of a faulty execution. This happens when the action raised an exception during its execute().

abstract property schema

Defines a Schema that the input parameters shall comply to

Returns
A schema declaring the input parameters this action should be provided along with their respective constraints

Return type
jsonschema.Schema instance

7.2.4 Register a new entry point

In order for the Watcher Applier to load your new action, the action must be registered as a named entry point under the `watcher_actions` entry point of your `setup.py` file. If you are using pbr, this entry point should be placed in your `setup.cfg` file.

The name you give to your entry point has to be unique.

Here below is how you would proceed to register `DummyAction` using pbr:

```
[entry_points]
watcher_actions =
    dummy = thirdparty.dummy:DummyAction
```

7.2.5 Using action plugins

The Watcher Applier service will automatically discover any installed plugins when it is restarted. If a Python package containing a custom plugin is installed within the same environment as Watcher, Watcher will automatically make that plugin available for use.

At this point, you can use your new action plugin in your *strategy plugin* if you reference it via the use of the `add_action()` method:

```
# [...]
self.solution.add_action(
    action_type="dummy",  # Name of the entry point we registered earlier
    applies_to="",
    input_parameters={})
```

By doing so, your action will be saved within the Watcher Database, ready to be processed by the planner for creating an action plan which can then be executed by the Watcher Applier via its workflow engine.

At the last, remember to add the action into the weights in `watcher.conf`, otherwise you will get an error when the action be referenced in a strategy.

7.2.6 Scheduling of an action plugin

Watcher provides a basic built-in *planner* which is only able to process the Watcher built-in actions. Therefore, you will either have to use an existing third-party planner or *implement another planner* that will be able to take into account your new action plugin.

7.2.7 Test your new action

In order to test your new action via a manual test or a Tempest test, you can use the `Actuator` strategy and pass it one or more actions to execute. This way, you can isolate your action to see if it works as expected.

7.3 Build a new cluster data model collector

Watcher Decision Engine has an external cluster data model (CDM) plugin interface which gives anyone the ability to integrate an external cluster data model collector (CDMC) in order to extend the initial set of cluster data model collectors Watcher provides.

This section gives some guidelines on how to implement and integrate custom cluster data model collectors within Watcher.

7.3.1 Creating a new plugin

In order to create a new cluster data model collector, you have to:

- Extend the `BaseClusterDataModelCollector` class.
- Implement its `execute()` abstract method to return your entire cluster data model that this method should build.
- Implement its `audit_scope_handler()` abstract property to return your audit scope handler.
- Implement its `notification_endpoints()` abstract property to return the list of all the `NotificationEndpoint` instances that will be responsible for handling incoming notifications in order to incrementally update your cluster data model.

First of all, you have to extend the `BaseClusterDataModelCollector` base class which defines the `execute()` abstract method you will have to implement. This method is responsible for building an entire cluster data model.

Here is an example showing how you can write a plugin called `DummyClusterDataModelCollector`:

```
# Filepath = <PROJECT_DIR>/thirdparty/dummy.py
# Import path = thirdparty.dummy

from watcher.decision_engine.model import model_root
from watcher.decision_engine.model.collector import base


class DummyClusterDataModelCollector(base.BaseClusterDataModelCollector):

    def execute(self):
        model = model_root.ModelRoot()
        # Do something here...
        return model

    @property
    def audit_scope_handler(self):
        return None

    @property
    def notification_endpoints(self):
        return []
```

This implementation is the most basic one. So in order to get a better understanding on how to implement a more advanced cluster data model collector, have a look at the `NovaClusterDataModelCollector` class.

7.3.2 Define a custom model

As you may have noticed in the above example, we are reusing an existing model provided by Watcher. However, this model can be easily customized by implementing a new class that would implement the `Model` abstract base class. Here below is simple example on how to proceed in implementing a custom Model:

```
# Filepath = <PROJECT_DIR>/thirdparty/dummy.py
# Import path = thirdparty.dummy

from watcher.decision_engine.model import base as modelbase
from watcher.decision_engine.model.collector import base


class MyModel(modelbase.Model):

    def to_string(self):
        return 'MyModel'


class DummyClusterDataModelCollector(base.BaseClusterDataModelCollector):

    def execute(self):
        model = MyModel()
        # Do something here...
        return model

    @property
    def notification_endpoints(self):
        return []
```

Here below is the abstract `Model` class that every single cluster data model should implement:

class watcher.decision_engine.model.base.**Model**

7.3.3 Define configuration parameters

At this point, you have a fully functional cluster data model collector. By default, cluster data model collectors define a `period` option (see `get_config_opts()`) that corresponds to the interval of time between each synchronization of the in-memory model.

However, in more complex implementation, you may want to define some configuration options so one can tune the cluster data model collector to your needs. To do so, you can implement the `get_config_opts()` class method as followed:

```
from oslo_config import cfg
from watcher.decision_engine.model import model_root
from watcher.decision_engine.model.collector import base


class DummyClusterDataModelCollector(base.BaseClusterDataModelCollector):

    def execute(self):
        model = model_root.ModelRoot()
        # Do something here...
        return model

    @property
```

(continues on next page)

(continued from previous page)

```
    def audit_scope_handler(self):
        return None

    @property
    def notification_endpoints(self):
        return []

    @classmethod
    def get_config_opts(cls):
        return super(
            DummyClusterDataModelCollector, cls).get_config_opts() + [
            cfg.StrOpt('test_opt', help="Demo Option.", default=0),
            # Some more options ...
        ]
```

The configuration options defined within this class method will be included within the global `watcher.conf` configuration file under a section named by convention: `{namespace}.{plugin_name}` (see section *Register a new entry point*). The namespace for CDMC plugins is `watcher_cluster_data_model_collectors`, so in our case, the `watcher.conf` configuration would have to be modified as followed:

```
[watcher_cluster_data_model_collectors.dummy]
# Option used for testing.
test_opt = test_value
```

Then, the configuration options you define within this method will then be injected in each instantiated object via the `config` parameter of the `__init__()` method.

7.3.4 Abstract Plugin Class

Here below is the abstract `BaseClusterDataModelCollector` class that every single cluster data model collector should implement:

class watcher.decision_engine.model.collector.base.**BaseClusterDataModelCollector**(**args*, ***kwargs*)

__init__(*config*, *osc=None*)

abstract execute()

Build a cluster data model

abstract get_audit_scope_handler(*audit_scope*)

Get audit scope handler

classmethod get_config_opts()

Defines the configuration options to be associated to this loadable

Returns

A list of configuration options relative to this Loadable

Return type

list of `oslo_config.cfg.Opt` instances

abstract property notification_endpoints

Associated notification endpoints

Returns
Associated notification endpoints

Return type
List of `EventsNotificationEndpoint` instances

synchronize()

Synchronize the cluster data model

Whenever called this synchronization will perform a drop-in replacement with the existing cluster data model

7.3.5 Register a new entry point

In order for the Watcher Decision Engine to load your new cluster data model collector, the latter must be registered as a named entry point under the `watcher_cluster_data_model_collectors` entry point namespace of your `setup.py` file. If you are using pbr, this entry point should be placed in your `setup.cfg` file.

The name you give to your entry point has to be unique.

Here below is how to register `DummyClusterDataModelCollector` using pbr:

```
[entry_points]
watcher_cluster_data_model_collectors =
    dummy = thirdparty.dummy:DummyClusterDataModelCollector
```

7.3.6 Add new notification endpoints

At this point, you have a fully functional cluster data model collector. However, this CDMC is only refreshed periodically via a background scheduler. As you may sometimes execute a strategy with a stale CDM due to a high activity on your infrastructure, you can define some notification endpoints that will be responsible for incrementally updating the CDM based on notifications emitted by other services such as Nova. To do so, you can implement and register a new `DummyEndpoint` notification endpoint regarding a `dummy` event as shown below:

```
from watcher.decision_engine.model import model_root
from watcher.decision_engine.model.collector import base


class DummyNotification(base.NotificationEndpoint):

    @property
    def filter_rule(self):
        return filtering.NotificationFilter(
            publisher_id=r'.*',
            event_type=r'^dummy$',
        )

    def info(self, ctxt, publisher_id, event_type, payload, metadata):
        # Do some CDM modifications here...
```

(continues on next page)

(continued from previous page)

```
        pass


class DummyClusterDataModelCollector(base.BaseClusterDataModelCollector):

    def execute(self):
        model = model_root.ModelRoot()
        # Do something here...
        return model

    @property
    def notification_endpoints(self):
        return [DummyNotification(self)]
```

Note that if the event you are trying to listen to is published by a new service, you may have to also add a new topic Watcher will have to subscribe to in the `notification_topics` option of the `[watcher_decision_engine]` section.

7.3.7 Using cluster data model collector plugins

The Watcher Decision Engine service will automatically discover any installed plugins when it is restarted. If a Python package containing a custom plugin is installed within the same environment as Watcher, Watcher will automatically make that plugin available for use.

At this point, you can use your new cluster data model plugin in your *strategy plugin* by using the `collector_manager` property as followed:

```
# [...]
dummy_collector = self.collector_manager.get_cluster_model_collector(
    "dummy")  # "dummy" is the name of the entry point we declared earlier
dummy_model = dummy_collector.get_latest_cluster_data_model()
# Do some stuff with this model
```

7.4 Build a new goal

Watcher Decision Engine has an external *goal* plugin interface which gives anyone the ability to integrate an external goal which can be achieved by a *strategy*.

This section gives some guidelines on how to implement and integrate custom goals with Watcher. If you wish to create a third-party package for your plugin, you can refer to our *documentation for third-party package creation*.

7.4.1 Pre-requisites

Before using any goal, please make sure that none of the existing goals fit your needs. Indeed, the underlying value of defining a goal is to be able to compare the efficacy of the action plans resulting from the various strategies satisfying the same goal. By doing so, Watcher can assist the administrator in his choices.

7.4.2 Create a new plugin

In order to create a new goal, you have to:

- Extend the `Goal` class.
- Implement its `get_name()` class method to return the **unique** ID of the new goal you want to create. This unique ID should be the same as the name of *the entry point you will declare later on*.
- Implement its `get_display_name()` class method to return the translated display name of the goal you want to create. Note: Do not use a variable to return the translated string so it can be automatically collected by the translation tool.
- Implement its `get_translatable_display_name()` class method to return the translation key (actually the english display name) of your new goal. The value return should be the same as the string translated in `get_display_name()`.
- Implement its `get_efficacy_specification()` method to return the *efficacy specification* for your goal.

Here is an example showing how you can define a new `NewGoal` goal plugin:

```
# filepath: thirdparty/new.py
# import path: thirdparty.new

from watcher._i18n import _
from watcher.decision_engine.goal import base
from watcher.decision_engine.goal.efficacy import specs

class NewGoal(base.Goal):

    @classmethod
    def get_name(cls):
        return "new_goal"  # Will be the name of the entry point

    @classmethod
    def get_display_name(cls):
        return _("New Goal")

    @classmethod
    def get_translatable_display_name(cls):
        return "New Goal"

    @classmethod
    def get_efficacy_specification(cls):
        return specs.Unclassified()
```

As you may have noticed, the `get_efficacy_specification()` method returns an `Unclassified()` instance which is provided by Watcher. This efficacy specification is useful during the development process of your goal as it corresponds to an empty specification. If you want to learn more about what efficacy specifications are used for or to define your own efficacy specification, please refer to the *related section below*.

7.4.3 Abstract Plugin Class

Here below is the abstract `Goal` class:

class watcher.decision_engine.goal.base.**Goal**(*config*)

classmethod get_config_opts()

Defines the configuration options to be associated to this loadable

Returns
A list of configuration options relative to this Loadable

Return type
list of `oslo_config.cfg.Opt` instances

abstract classmethod get_display_name()

The goal display name for the goal

abstract get_efficacy_specification()

The efficacy spec for the current goal

abstract classmethod get_name()

Name of the goal: should be identical to the related entry point

abstract classmethod get_translatable_display_name()

The translatable msgid of the goal

7.4.4 Add a new entry point

In order for the Watcher Decision Engine to load your new goal, the goal must be registered as a named entry point under the `watcher_goals` entry point namespace of your `setup.py` file. If you are using pbr, this entry point should be placed in your `setup.cfg` file.

The name you give to your entry point has to be unique and should be the same as the value returned by the `get_name()` class method of your goal.

Here below is how you would proceed to register `NewGoal` using pbr:

```
[entry_points]
watcher_goals =
    new_goal = thirdparty.new:NewGoal
```

To get a better understanding on how to implement a more advanced goal, have a look at the `watcher.decision_engine.goal.goals.ServerConsolidation` class.

7.4.5 Implement a customized efficacy specification

What is it for?

Efficacy specifications define a set of specifications for a given goal. These specifications actually define a list of indicators which are to be used to compute a global efficacy that outlines how well a strategy performed when trying to achieve the goal it is associated to.

The idea behind such specification is to give the administrator the possibility to run an audit using different strategies satisfying the same goal and be able to judge how they performed at a glance.

Implementation

In order to create a new efficacy specification, you have to:

- Extend the `EfficacySpecification` class.
- Implement `get_indicators_specifications()` by returning a list of `IndicatorSpecification` instances.
  - Each `IndicatorSpecification` instance should actually extend the latter.
  - Each indicator specification should have a **unique name** which should be a valid Python variable name.
  - They should implement the `schema` abstract property by returning a `Schema` instance. This schema is the contract the strategy will have to comply with when setting the value associated to the indicator specification within its solution (see the *architecture of Watcher* for more information on the audit execution workflow).
- Implement the `get_global_efficacy()` method: it should compute the global efficacy for the goal it achieves based on the efficacy indicators you just defined.

Here below is an example of an efficacy specification containing one indicator specification:

```
from watcher._i18n import _
from watcher.decision_engine.goal.efficacy import base as efficacy_base
from watcher.decision_engine.goal.efficacy import indicators
from watcher.decision_engine.solution import efficacy


class IndicatorExample(IndicatorSpecification):
    def __init__(self):
        super(IndicatorExample, self).__init__(
            name="indicator_example",
            description=_("Example of indicator specification."),
            unit=None,
        )

    @property
    def schema(self):
        return voluptuous.Schema(voluptuous.Range(min=0), required=True)


class UnclassifiedStrategySpecification(efficacy_base.EfficacySpecification):

    def get_indicators_specifications(self):
        return [IndicatorExample()]

    def get_global_efficacy(self, indicators_map):
        return efficacy.Indicator(
          name="global_efficacy_indicator",
          description="Example of global efficacy indicator",
          unit="%",
          value=indicators_map.indicator_example % 100)
```

To get a better understanding on how to implement an efficacy specification, have a look at `ServerConsolidationSpecification`.

Also, if you want to see a concrete example of an indicator specification, have a look at `ReleasedComputeNodesCount`.

7.5 Build a new planner

Watcher *Decision Engine* has an external *planner* plugin interface which gives anyone the ability to integrate an external *planner* in order to extend the initial set of planners Watcher provides.

This section gives some guidelines on how to implement and integrate custom planners with Watcher.

7.5.1 Creating a new plugin

First of all you have to extend the base `BasePlanner` class which defines an abstract method that you will have to implement. The `schedule()` is the method being called by the Decision Engine to schedule a given solution (`BaseSolution`) into an *action plan* by ordering/sequencing an unordered set of actions contained in the proposed solution (for more details, see *definition of a solution*).

Here is an example showing how you can write a planner plugin called `DummyPlanner`:

```
# Filepath = third-party/third_party/dummy.py
# Import path = third_party.dummy
from oslo_utils import uuidutils
from watcher.decision_engine.planner import base


class DummyPlanner(base.BasePlanner):

    def _create_action_plan(self, context, audit_id):
        action_plan_dict = {
            'uuid': uuidutils.generate_uuid(),
            'audit_id': audit_id,
            'first_action_id': None,
            'state': objects.action_plan.State.RECOMMENDED
        }

        new_action_plan = objects.ActionPlan(context, **action_plan_dict)
        new_action_plan.create(context)
        new_action_plan.save()
        return new_action_plan

    def schedule(self, context, audit_id, solution):
        # Empty action plan
        action_plan = self._create_action_plan(context, audit_id)
        # todo: You need to create the workflow of actions here
        # and attach it to the action plan
        return action_plan
```

This implementation is the most basic one. So if you want to have more advanced examples, have a look at the implementation of planners already provided by Watcher like `DefaultPlanner`. A list with all available planner plugins can be found *here*.

7.5.2 Define configuration parameters

At this point, you have a fully functional planner. However, in more complex implementation, you may want to define some configuration options so one can tune the planner to its needs. To do so, you can implement the `get_config_opts()` class method as followed:

```
from oslo_config import cfg

class DummyPlanner(base.BasePlanner):

    # [...]

    def schedule(self, context, audit_uuid, solution):
        assert self.config.test_opt == 0
        # [...]

    @classmethod
    def get_config_opts(cls):
        return super(
            DummyPlanner, cls).get_config_opts() + [
            cfg.StrOpt('test_opt', help="Demo Option.", default=0),
            # Some more options ...
        ]
```

The configuration options defined within this class method will be included within the global `watcher.conf` configuration file under a section named by convention: `{namespace}.{plugin_name}`. In our case, the `watcher.conf` configuration would have to be modified as followed:

```
[watcher_planners.dummy]
# Option used for testing.
test_opt = test_value
```

Then, the configuration options you define within this method will then be injected in each instantiated object via the `config` parameter of the `__init__()` method.

7.5.3 Abstract Plugin Class

Here below is the abstract `BasePlanner` class that every single planner should implement:

class watcher.decision_engine.planner.base.**BasePlanner**(*config*)

classmethod get_config_opts()

Defines the configuration options to be associated to this loadable

Returns
A list of configuration options relative to this Loadable

Return type
list of `oslo_config.cfg.Opt` instances

abstract schedule(*context*, *audit_uuid*, *solution*)

The planner receives a solution to schedule

Parameters

- **solution** (`BaseSolution` subclass instance) A solution provided by a strategy for scheduling
- **audit_uuid** (*str*) the audit uuid

Returns
Action plan with an ordered sequence of actions such that all security, dependency, and performance requirements are met.

Return type
`watcher.objects.ActionPlan` instance

7.5.4 Register a new entry point

In order for the Watcher Decision Engine to load your new planner, the latter must be registered as a new entry point under the `watcher_planners` entry point namespace of your `setup.py` file. If you are using pbr, this entry point should be placed in your `setup.cfg` file.

The name you give to your entry point has to be unique.

Here below is how you would proceed to register `DummyPlanner` using pbr:

```
[entry_points]
watcher_planners =
    dummy = third_party.dummy:DummyPlanner
```

7.5.5 Using planner plugins

The *Watcher Decision Engine* service will automatically discover any installed plugins when it is started. This means that if Watcher is already running when you install your plugin, you will have to restart the related Watcher services. If a Python package containing a custom plugin is installed within the same environment as Watcher, Watcher will automatically make that plugin available for use.

At this point, Watcher will use your new planner if you referenced it in the `planner` option under the `[watcher_planner]` section of your `watcher.conf` configuration file when you started it. For example, if you want to use the `dummy` planner you just installed, you would have to select it as followed:

```
[watcher_planner]
planner = dummy
```

As you may have noticed, only a single planner implementation can be activated at a time, so make sure it is generic enough to support all your strategies and actions.

7.6 Build a new scoring engine

Watcher Decision Engine has an external *scoring engine* plugin interface which gives anyone the ability to integrate an external scoring engine in order to make use of it in a *strategy*.

This section gives some guidelines on how to implement and integrate custom scoring engines with Watcher. If you wish to create a third-party package for your plugin, you can refer to our *documentation for third-party package creation*.

7.6.1 Pre-requisites

Because scoring engines execute a purely mathematical tasks, they typically do not have any additional dependencies. Additional requirements might be defined by specific scoring engine implementations. For example, some scoring engines might require to prepare learning data, which has to be loaded during the scoring engine startup. Some other might require some external services to be available (e.g. if the scoring infrastructure is running in the cloud).

7.6.2 Create a new scoring engine plugin

In order to create a new scoring engine you have to:

- Extend the `watcher.decision_engine.scoring.base.ScoringEngine` class
- Implement its `get_name()` method to return the **unique** ID of the new scoring engine you want to create. This unique ID should be the same as the name of *the entry point we will declare later on*.
- Implement its `get_description()` method to return the user-friendly description of the implemented scoring engine. It might contain information about algorithm used, learning data etc.
- Implement its `get_metainfo()` method to return the machine-friendly metadata about this scoring engine. For example, it could be a JSON formatted text with information about the data model used, its input and output data format, column names, etc.
- Implement its `calculate_score()` method to return the result calculated by this scoring engine.

Here is an example showing how you can write a plugin called `NewScorer`:

```
# filepath: thirdparty/new.py
# import path: thirdparty.new
from watcher.decision_engine.scoring import base


class NewScorer(base.ScoringEngine):

    def get_name(self):
        return 'new_scorer'

    def get_description(self):
        return ''

    def get_metainfo(self):
        return """{
            "feature_columns": [
                "column1",
                "column2",
                "column3"],
            "result_columns": [
                "value",
                "probability"]
            }"""

    def calculate_score(self, features):
        return '[12, 0.83]'
```

As you can see in the above example, the `calculate_score()` method returns a string. Both this class and the client (caller) should perform all the necessary serialization or deserialization.

7.6.3 (Optional) Create a new scoring engine container plugin

Optionally, its possible to implement a container plugin, which can return a list of scoring engines. This list can be re-evaluated multiple times during the lifecycle of *Watcher Decision Engine* and synchronized with *Watcher Database* using the `watcher-sync` command line tool.

Below is an example of a container using some scoring engine implementation that is simply made of a client responsible for communicating with a real scoring engine deployed as a web service on external servers:

```
class NewScoringContainer(base.ScoringEngineContainer):

    @classmethod
    def get_scoring_engine_list(self):
        return [
            RemoteScoringEngine(
                name='scoring_engine1',
                description='Some remote Scoring Engine 1',
                remote_url='http://engine1.example.com/score'),
            RemoteScoringEngine(
                name='scoring_engine2',
                description='Some remote Scoring Engine 2',
                remote_url='http://engine2.example.com/score'),
        ]
```

7.6.4 Abstract Plugin Class

Here below is the abstract `watcher.decision_engine.scoring.base.ScoringEngine` class:

class watcher.decision_engine.scoring.base.**ScoringEngine**(*config*)

A base class for all the Scoring Engines.

A Scoring Engine is an instance of a data model, to which the learning data was applied.

Please note that this class contains non-static and non-class methods by design, so that its easy to create multiple Scoring Engine instances using a single class (possibly configured differently).

abstract calculate_score(*features*)

Calculates a score value based on arguments passed.

Scoring Engines might be very different to each other. They might solve different problems or use different algorithms or frameworks internally. To enable this kind of flexibility, the method takes only one argument (string) and produces the results in the same format (string). The consumer of the Scoring Engine is ultimately responsible for providing the right arguments and parsing the result.

Parameters
features (*str*) Input data for Scoring Engine

Returns
A score result

Return type
str

classmethod get_config_opts()

Defines the configuration options to be associated to this loadable

Returns
A list of configuration options relative to this Loadable

Return type
list of oslo_config.cfg.Opt instances

abstract get_description()

Returns the description of the Scoring Engine.

The description might contain any human readable information, which might be useful for Strategy developers planning to use this Scoring Engine. It will be also visible in the Watcher API and CLI.

Returns
A Scoring Engine description

Return type
str

abstract get_metainfo()

Returns the metadata information about Scoring Engine.

The metadata might contain a machine-friendly (e.g. in JSON format) information needed to use this Scoring Engine. For example, some Scoring Engines require to pass the array of features in particular order to be able to calculate the score value. This order can be defined in metadata and used in Strategy.

Returns
A Scoring Engine metadata

Return type
str

abstract get_name()

Returns the name of the Scoring Engine.

The name should be unique across all Scoring Engines.

Returns
A Scoring Engine name

Return type
str

7.6.5 Abstract Plugin Container Class

Here below is the abstract ScoringContainer class:

class watcher.decision_engine.scoring.base.ScoringEngineContainer(*config*)

A base class for all the Scoring Engines Containers.

A Scoring Engine Container is an abstraction which allows to plugin multiple Scoring Engines as a single Stevedore plugin. This enables some more advanced scenarios like dynamic reloading of Scoring Engine implementations without having to restart any Watcher services.

classmethod get_config_opts()

Defines the configuration options to be associated to this loadable

Returns
A list of configuration options relative to this Loadable

Return type
list of `oslo_config.cfg.Opt` instances

abstract classmethod get_scoring_engine_list()

Returns a list of Scoring Engine instances.

Returns
A list of Scoring Engine instances

Return type

class
~.scoring_engine.ScoringEngine

7.6.6 Add a new entry point

In order for the Watcher Decision Engine to load your new scoring engine, it must be registered as a named entry point under the `watcher_scoring_engines` entry point of your `setup.py` file. If you are using pbr, this entry point should be placed in your `setup.cfg` file.

The name you give to your entry point has to be unique and should be the same as the value returned by the `get_name()` method of your strategy.

Here below is how you would proceed to register `NewScorer` using pbr:

```
[entry_points]
watcher_scoring_engines =
    new_scorer = thirdparty.new:NewScorer
```

To get a better understanding on how to implement a more advanced scoring engine, have a look at the `DummyScorer` class. This implementation is not really using machine learning, but other than that it contains all the pieces which the real implementation would have.

In addition, for some use cases there is a need to register a list (possibly dynamic, depending on the implementation and configuration) of scoring engines in a single plugin, so there is no need to restart *Watcher Decision Engine* every time such list changes. For these cases, an additional `watcher_scoring_engine_containers` entry point can be used.

For the example how to use scoring engine containers, please have a look at the `DummyScoringContainer` and the way it is configured in `setup.cfg`. For new containers it could be done like this:

```
[entry_points]
watcher_scoring_engine_containers =
    new_scoring_container = thirdparty.new:NewContainer
```

7.6.7 Using scoring engine plugins

The Watcher Decision Engine service will automatically discover any installed plugins when it is restarted. If a Python package containing a custom plugin is installed within the same environment as Watcher, Watcher will automatically make that plugin available for use.

At this point, Watcher will scan and register inside the *Watcher Database* all the scoring engines you implemented upon restarting the *Watcher Decision Engine*.

In addition, `watcher-sync` tool can be used to trigger *Watcher Database* synchronization. This might be used for dynamic scoring containers, which can return different scoring engines based on some external configuration (if they support that).

7.7 Build a new optimization strategy

Watcher Decision Engine has an external *strategy* plugin interface which gives anyone the ability to integrate an external strategy in order to make use of placement algorithms.

This section gives some guidelines on how to implement and integrate custom strategies with Watcher. If you wish to create a third-party package for your plugin, you can refer to our *documentation for third-party package creation*.

7.7.1 Pre-requisites

Before using any strategy, you should make sure you have your Telemetry service configured so that it would provide you all the metrics you need to be able to use your strategy.

7.7.2 Create a new strategy plugin

In order to create a new strategy, you have to:

- Extend the `UnclassifiedStrategy` class
- Implement its `get_name()` class method to return the **unique** ID of the new strategy you want to create. This unique ID should be the same as the name of *the entry point we will declare later on*.
- Implement its `get_display_name()` class method to return the translated display name of the strategy you want to create. Note: Do not use a variable to return the translated string so it can be automatically collected by the translation tool.
- Implement its `get_translatable_display_name()` class method to return the translation key (actually the English display name) of your new strategy. The value return should be the same as the string translated in `get_display_name()`.
- Implement its `execute()` method to return the solution you computed within your strategy.

Here is an example showing how you can write a plugin called `NewStrategy`:

```
# filepath: thirdparty/new.py
# import path: thirdparty.new
import abc
from watcher._i18n import _
from watcher.decision_engine.strategy.strategies import base


class NewStrategy(base.UnclassifiedStrategy):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, osc=None):
        super(NewStrategy, self).__init__(osc)

    def execute(self, original_model):
        self.solution.add_action(action_type="nop",
                                 input_parameters=parameters)
        # Do some more stuff here ...
        return self.solution

    @classmethod
    def get_name(cls):
        return "new_strategy"

    @classmethod
    def get_display_name(cls):
        return _("New strategy")

    @classmethod
    def get_translatable_display_name(cls):
        return "New strategy"
```

As you can see in the above example, the `execute()` method returns a `BaseSolution` instance as required. This solution is what wraps the abstract set of actions the strategy recommends to you. This solution is then processed by a *planner* to produce an action plan which contains the sequenced flow of actions to be executed by the *Watcher Applier*. This solution also contains the various *efficacy indicators* alongside its computed *global efficacy*.

Please note that your strategy class will expect to find the same constructor signature as BaseStrategy to instantiate you strategy. Therefore, you should ensure that your `__init__` signature is identical to the `BaseStrategy` one.

7.7.3 Strategy efficacy

As stated before, the `NewStrategy` class extends a class called `UnclassifiedStrategy`. This class actually implements a set of abstract methods which are defined within the `BaseStrategy` parent class.

One thing this `UnclassifiedStrategy` class defines is that our `NewStrategy` achieves the `unclassified` goal. This goal is a peculiar one as it does not contain any indicator nor does it calculate a global efficacy. This proves itself to be quite useful during the development of a new strategy for which the goal has yet to be defined or in case a *new goal* has yet to be implemented.

7.7.4 Define Strategy Parameters

For each new added strategy, you can add parameters spec so that an operator can input strategy parameters when creating an audit to control the `execute()` behavior of strategy. This is useful to define some threshold for your strategy, and tune them at runtime.

To define parameters, just implements `get_schema()` to return parameters spec with jsonschema format. It is strongly encouraged that provide default value for each parameter, or else reference fails if operator specify no parameters.

Here is an example showing how you can define 2 parameters for `DummyStrategy`:

```
class DummyStrategy(base.DummyBaseStrategy):

    @classmethod
    def get_schema(cls):
        return {
            "properties": {
                "para1": {
                    "description": "number parameter example",
                    "type": "number",
                    "default": 3.2,
                    "minimum": 1.0,
                    "maximum": 10.2,
                },
                "para2": {
                    "description": "string parameter example",
                    "type": "string",
                    "default": "hello",
                },
            },
        }
```

You can reference parameters in `execute()`:

```
class DummyStrategy(base.DummyBaseStrategy):

    def execute(self):
        para1 = self.input_parameters.para1
        para2 = self.input_parameters.para2

        if para1 > 5:
            ...
```

Operator can specify parameters with following commands:

```
$ watcher audit create -a <your_audit_template> -p para1=6.0 -p para2=hi
```

Pls. check user-guide for details.

7.7.5 Abstract Plugin Class

Here below is the abstract `BaseStrategy` class:

class watcher.decision_engine.strategy.strategies.base.**BaseStrategy**(*config*, *osc=None*)

A base class for all the strategies

A Strategy is an algorithm implementation which is able to find a Solution for a given Goal.

DATASOURCE_METRICS = []

Contains all metrics the strategy requires from a datasource to properly execute

__init__(*config*, *osc=None*)

Constructor: the signature should be identical within the subclasses

Parameters

- **config** (Struct) Configuration related to this plugin
- **osc** (OpenStackClients instance) An OpenStackClients instance

property baremetal_model

Cluster data model

Returns
Cluster data model the strategy is executed on

Rtype model
ModelRoot instance

property compute_model

Cluster data model

Returns
Cluster data model the strategy is executed on

Rtype model
ModelRoot instance

abstract do_execute(*audit=None*)

Strategy execution phase

Parameters
audit (Audit instance) An Audit instance

This phase is where you should put the main logic of your strategy.

execute(*audit=None*)

Execute a strategy

Parameters
audit (Audit instance) An Audit instance

Returns
A computed solution (via a placement algorithm)

Return type
BaseSolution instance

classmethod get_config_opts()

Defines the configuration options to be associated to this loadable

Returns
A list of configuration options relative to this Loadable

Return type
list of oslo_config.cfg.Opt instances

abstract classmethod get_display_name()

The goal display name for the strategy

classmethod get_goal()

The goal the strategy achieves

abstract classmethod get_goal_name()

The goal name the strategy achieves

abstract classmethod get_name()

The name of the strategy

classmethod get_schema()

Defines a Schema that the input parameters shall comply to

Returns
A jsonschema format (mandatory default setting)

Return type
dict

abstract classmethod get_translatable_display_name()

The translatable msgid of the strategy

abstract post_execute()

Post-execution phase

This can be used to compute the global efficacy

abstract pre_execute()

Pre-execution phase

This can be used to fetch some pre-requisites or data.

property storage_model

Cluster data model

Returns
Cluster data model the strategy is executed on

Rtype model
`ModelRoot` instance

7.7.6 Add a new entry point

In order for the Watcher Decision Engine to load your new strategy, the strategy must be registered as a named entry point under the `watcher_strategies` entry point of your `setup.py` file. If you are using pbr, this entry point should be placed in your `setup.cfg` file.

The name you give to your entry point has to be unique and should be the same as the value returned by the `get_name()` class method of your strategy.

Here below is how you would proceed to register `NewStrategy` using pbr:

```
[entry_points]
watcher_strategies =
    new_strategy = thirdparty.new:NewStrategy
```

To get a better understanding on how to implement a more advanced strategy, have a look at the `BasicConsolidation` class.

7.7.7 Using strategy plugins

The Watcher Decision Engine service will automatically discover any installed plugins when it is restarted. If a Python package containing a custom plugin is installed within the same environment as Watcher, Watcher will automatically make that plugin available for use.

At this point, Watcher will scan and register inside the *Watcher Database* all the strategies (alongside the goals they should satisfy) you implemented upon restarting the *Watcher Decision Engine*.

You should take care when installing strategy plugins. By their very nature, there are no guarantees that utilizing them as is will be supported, as they may require a set of metrics which is not yet available within the Telemetry service. In such a case, please do make sure that you first check/configure the latter so your new strategy can be fully functional.

Querying metrics

A large set of metrics, generated by OpenStack modules, can be used in your strategy implementation. To collect these metrics, Watcher provides a DataSourceManager for two data sources which are Ceilometer (with Gnocchi as API) and Monasca. If you wish to query metrics from a different data source, you can implement your own and use it via DataSourceManager from within your new strategy. Indeed, strategies in Watcher have the cluster data models decoupled from the data sources which means that you may keep the former while changing the latter. The recommended way for you to support a new data source is to implement a new helper that would encapsulate within separate methods the queries you need to perform. To then use it, you would just have to add it to appropriate watcher_strategies.* section in config file.

If you want to use Ceilometer but with your own metrics database backend, please refer to the Ceilometer developer guide. The list of the available Ceilometer backends is located here. The Ceilosca project is a good example of how to create your own pluggable backend. Moreover, if your strategy requires new metrics not covered by Ceilometer, you can add them through a Ceilometer plugin.

Read usage metrics using the Watcher Datasource Helper

The following code snippet shows how datasource_backend is defined:

```
from watcher.datasource import manager as ds_manager

@property
def datasource_backend(self):
    if not self._datasource_backend:

        # Load the global preferred datasources order but override it
        # if the strategy has a specific datasources config
        datasources = CONF.watcher_datasources
        if self.config.datasources:
            datasources = self.config

        self._datasource_backend = ds_manager.DataSourceManager(
            config=datasources,
            osc=self.osc
        ).get_backend(self.DATASOURCE_METRICS)
    return self._datasource_backend
```

Using that you can now query the values for that specific metric:

```
avg_meter = self.datasource_backend.statistic_aggregation(
    instance.uuid, 'instance_cpu_usage', self.periods['instance'],
    self.granularity,
    aggregation=self.aggregation_method['instance'])
```

7.8 Available Plugins

In this section we present all the plugins that are shipped along with Watcher. If you want to know which plugins your Watcher services have access to, you can use the *Guru Meditation Reports* to display them.

7.8.1 Goals

airflow_optimization

AirflowOptimization

This goal is used to optimize the airflow within a cloud infrastructure.

cluster_maintaining

ClusterMaintenance

This goal is used to maintain compute nodes without having the users application being interrupted.

dummy

Dummy

Reserved goal that is used for testing purposes.

hardware_maintenance

HardwareMaintenance

This goal is to migrate instances and volumes on a set of compute nodes and storage from nodes under maintenance

noisy_neighbor

NoisyNeighborOptimization

This goal is used to identify and migrate a Noisy Neighbor - a low priority VM that negatively affects performance of a high priority VM in terms of IPC by over utilizing Last Level Cache.

saving_energy

SavingEnergy

This goal is used to reduce power consumption within a data center.

server_consolidation

ServerConsolidation

This goal is for efficient usage of compute server resources in order to reduce the total number of servers.

thermal_optimization

ThermalOptimization

This goal is used to balance the temperature across different servers.

unclassified

Unclassified

This goal is used to ease the development process of a strategy. Containing no actual indicator specification, this goal can be used whenever a strategy has yet to be formally associated with an existing goal. If the goal achieve has been identified but there is no available implementation, this Goal can also be used as a transitional stage.

workload_balancing

WorkloadBalancing

This goal is used to evenly distribute workloads across different servers.

7.8.2 Scoring Engines

dummy_scorer

Sample Scoring Engine implementing simplified workload classification.

Typically a scoring engine would be implemented using machine learning techniques. For example, for workload classification problem the solution could consist of the following steps:

1. Define a problem to solve: we want to detect the workload on the machine based on the collected metrics like power consumption, temperature, CPU load, memory usage, disk usage, network usage, etc.
2. The workloads could be predefined, e.g. IDLE, CPU-INTENSIVE, MEMORY-INTENSIVE, IO-BOUND, Or we could let the ML algorithm to find the workloads based on the learning data provided. The decision here leads to learning algorithm used (supervised vs. non-supervised learning).
3. Collect metrics from sample servers (learning data).
4. Define the analytical model, pick ML framework and algorithm.
5. Apply learning data to the data model. Once taught, the data model becomes a scoring engine and can start doing predictions or classifications.
6. Wrap up the scoring engine with the class like this one, so it has a standard interface and can be used inside Watcher.

This class is a greatly very simplified version of the above model. The goal is to provide an example how such class could be implemented and used in Watcher, without adding additional dependencies like machine learning frameworks (which can be quite heavy) or over-complicating its internal implementation, which can distract from looking at the overall picture.

That said, this class implements a workload classification manually (in plain python code) and is not intended to be used in production.

7.8.3 Scoring Engine Containers

dummy_scoring_container

Sample Scoring Engine container returning a list of scoring engines.

Please note that it can be used in dynamic scenarios and the returned list might return instances based on some external configuration (e.g. in database). In order for these scoring engines to become discoverable in Watcher API and Watcher CLI, a database re-sync is required. It can be executed using watcher-sync tool for example.

7.8.4 Strategies

actuator

Actuator

Actuator that simply executes the actions given as parameter

This strategy allows anyone to create an action plan with a predefined set of actions. This strategy can be used for 2 different purposes:

- Test actions
- Use this strategy based on an event trigger to perform some explicit task

basic

Good server consolidation strategy

Basic offline consolidation using live migration

Consolidation of VMs is essential to achieve energy optimization in cloud environments such as OpenStack. As VMs are spinned up and/or moved over time, it becomes necessary to migrate VMs among servers to lower the costs. However, migration of VMs introduces runtime overheads and consumes extra energy, thus a good server consolidation strategy should carefully plan for migration in order to both minimize energy consumption and comply to the various SLAs.

This algorithm not only minimizes the overall number of used servers, but also minimizes the number of migrations.

It has been developed only for tests. You must have at least 2 physical compute nodes to run it, so you can easily run it on DevStack. It assumes that live migration is possible on your OpenStack cluster.

dummy

Dummy strategy used for integration testing via Tempest

Description

This strategy does not provide any useful optimization. Its only purpose is to be used by Tempest tests.

Requirements

<None>

Limitations

Do not use in production.

Spec URL

<None>

dummy_with_resize

Dummy strategy used for integration testing via Tempest

Description

This strategy does not provide any useful optimization. Its only purpose is to be used by Tempest tests.

Requirements

<None>

Limitations

Do not use in production.

Spec URL

<None>

dummy_with_scorer

A dummy strategy using dummy scoring engines.

This is a dummy strategy demonstrating how to work with scoring engines. One scoring engine is predicting the workload type of a machine based on the telemetry data, the other one is simply calculating the average value for given elements in a list. Results are then passed to the NOP action.

The strategy is presenting the whole workflow: - Get a reference to a scoring engine - Prepare input data (features) for score calculation - Perform score calculation - Use scorers metadata for results interpretation

host_maintenance

[PoC]Host Maintenance

Description

> It is a migration strategy for one compute node maintenance, without having the users application been interrupted. If given one backup node, the strategy will firstly migrate all instances from the maintenance node to the backup node. If the backup node is not provided, it will migrate all instances, relying on nova-scheduler.

Requirements

- You must have at least 2 physical compute nodes to run this strategy.

Limitations

- This is a proof of concept that is not meant to be used in production
- It migrates all instances from one host to other hosts. Its better to execute such strategy when load is not heavy, and use this algorithm with *ONESHOT* audit.
- It assumes that cold and live migrations are possible.

node_resource_consolidation

consolidating resources on nodes using server migration

Description

This strategy checks the resource usages of compute nodes, if the used resources are less than total, it will try to migrate server to consolidate the use of resource.

Requirements

- You must have at least 2 compute nodes to run this strategy.
- Hardware: compute nodes should use the same physical CPUs/RAMs

Limitations

- This is a proof of concept that is not meant to be used in production
- It assume that live migrations are possible

Spec URL

http://specs.openstack.org/openstack/watcher-specs/specs/train/implemented/node-resource-consolidation.html

noisy_neighbor

Noisy Neighbor strategy using live migration

Description

This strategy can identify and migrate a Noisy Neighbor - a low priority VM that negatively affects performance of a high priority VM in terms of IPC by over utilizing Last Level Cache.

Requirements

To enable LLC metric, latest Intel server with CMT support is required.

Limitations

This is a proof of concept that is not meant to be used in production

Spec URL

http://specs.openstack.org/openstack/watcher-specs/specs/pike/implemented/noisy_neighbor_strategy.html

outlet_temperature

[PoC] Outlet temperature control using live migration

Description

It is a migration strategy based on the outlet temperature of compute hosts. It generates solutions to move a workload whenever a servers outlet temperature is higher than the specified threshold.

Requirements

- Hardware: All computer hosts should support IPMI and PTAS technology
- Software: Ceilometer component ceilometer-agent-ipmi running in each compute host, and Ceilometer API can report such telemetry `hardware.ipmi.node.outlet_temperature` successfully.

- You must have at least 2 physical compute hosts to run this strategy.

Limitations

- This is a proof of concept that is not meant to be used in production
- We cannot forecast how many servers should be migrated. This is the reason why we only plan a single virtual machine migration at a time. So its better to use this algorithm with *CONTINUOUS* audits.
- It assume that live migrations are possible

Spec URL

https://github.com/openstack/watcher-specs/blob/master/specs/mitaka/implemented/outlet-temperature-based-strategy.rst

saving_energy

Saving Energy Strategy

Description

Saving Energy Strategy together with VM Workload Consolidation Strategy can perform the Dynamic Power Management (DPM) functionality, which tries to save power by dynamically consolidating workloads even further during periods of low resource utilization. Virtual machines are migrated onto fewer hosts and the unneeded hosts are powered off.

After consolidation, Saving Energy Strategy produces a solution of powering off/on according to the following detailed policy:

In this policy, a preset number(min_free_hosts_num) is given by user, and this min_free_hosts_num describes minimum free compute nodes that users expect to have, where free compute nodes refers to those nodes unused but still powered on.

If the actual number of unused nodes(in power-on state) is larger than the given number, randomly select the redundant nodes and power off them; If the actual number of unused nodes(in poweron state) is smaller than the given number and there are spare unused nodes(in poweroff state), randomly select some nodes(unused,poweroff) and power on them.

Requirements

In this policy, in order to calculate the min_free_hosts_num, users must provide two parameters:

- One parameter(min_free_hosts_num) is a constant int number. This number should be int type and larger than zero.
- The other parameter(free_used_percent) is a percentage number, which describes the quotient of min_free_hosts_num/nodes_with_VMs_num, where nodes_with_VMs_num is the number of nodes with VMs running on it. This parameter is used to calculate a dynamic min_free_hosts_num. The nodes with VMs refer to those nodes with VMs running on it.

Then choose the larger one as the final min_free_hosts_num.

Limitations

- at least 2 physical compute hosts

Spec URL

http://specs.openstack.org/openstack/watcher-specs/specs/pike/implemented/energy-saving-strategy.html

storage_capacity_balance

Storage capacity balance using cinder volume migration

Description

This strategy migrates volumes based on the workload of the cinder pools. It makes decision to migrate a volume whenever a pools used utilization % is higher than the specified threshold. The volume to be moved should make the pool close to average workload of all cinder pools.

Requirements

- You must have at least 2 cinder volume pools to run this strategy.

Limitations

- Volume migration depends on the storage device. It may take a long time.

Spec URL

http://specs.openstack.org/openstack/watcher-specs/specs/queens/implemented/storage-capacity-balance.html

uniform_airflow

[PoC]Uniform Airflow using live migration

Description

> It is a migration strategy based on the airflow of physical servers. It generates solutions to move VM whenever a servers airflow is higher than the specified threshold.

Requirements

- Hardware: compute node with NodeManager 3.0 support
- Software: Ceilometer component ceilometer-agent-compute running in each compute node, and Ceilometer API can report such telemetry airflow, system power, inlet temperature successfully.
- You must have at least 2 physical compute nodes to run this strategy

Limitations

- This is a proof of concept that is not meant to be used in production.
- We cannot forecast how many servers should be migrated. This is the reason why we only plan a single virtual machine migration at a time. So its better to use this algorithm with *CONTINUOUS* audits.
- It assumes that live migrations are possible.

vm_workload_consolidation

VM Workload Consolidation Strategy

A load consolidation strategy based on heuristic first-fit algorithm which focuses on measured CPU utilization and tries to minimize hosts which have too much or too little load respecting resource capacity constraints.

This strategy produces a solution resulting in more efficient utilization of cluster resources using following four phases:

- Offload phase - handling over-utilized resources

- Consolidation phase - handling under-utilized resources
- Solution optimization - reducing number of migrations
- Disability of unused compute nodes

A capacity coefficients (cc) might be used to adjust optimization thresholds. Different resources may require different coefficient values as well as setting up different coefficient values in both phases may lead to more efficient consolidation in the end. If the cc equals 1 the full resource capacity may be used, cc values lower than 1 will lead to resource under utilization and values higher than 1 will lead to resource overbooking. e.g. If targeted utilization is 80 percent of a compute node capacity, the coefficient in the consolidation phase will be 0.8, but may any lower value in the offloading phase. The lower it gets the cluster will appear more released (distributed) for the following consolidation phase.

As this strategy leverages VM live migration to move the load from one compute node to another, this feature needs to be set up correctly on all compute nodes within the cluster. This strategy assumes it is possible to live migrate any VM from an active compute node to any other active compute node.

workload_balance

[PoC]Workload balance using live migration

Description

> It is a migration strategy based on the VM workload of physical servers. It generates solutions to move a workload whenever a servers CPU or RAM utilization % is higher than the specified threshold. The VM to be moved should make the host close to average workload of all compute nodes.

Requirements

- Hardware: compute node should use the same physical CPUs/RAMs
- Software: Ceilometer component ceilometer-agent-compute running in each compute node, and Ceilometer API can report such telemetry instance_cpu_usage and instance_ram_usage successfully.
- You must have at least 2 physical compute nodes to run this strategy.

Limitations

- This is a proof of concept that is not meant to be used in production
- We cannot forecast how many servers should be migrated. This is the reason why we only plan a single virtual machine migration at a time. So its better to use this algorithm with *CONTINUOUS* audits.
- It assume that live migrations are possible

workload_stabilization

Workload Stabilization control using live migration

This is workload stabilization strategy based on standard deviation algorithm. The goal is to determine if there is an overload in a cluster and respond to it by migrating VMs to stabilize the cluster.

This strategy has been tested in a small (32 nodes) cluster.

It assumes that live migrations are possible in your cluster.

zone_migration

Zone migration using instance and volume migration

This is zone migration strategy to migrate many instances and volumes efficiently with minimum downtime for hardware maintenance.

7.8.5 Actions

change_node_power_state

Compute node power on/off

By using this action, you will be able to on/off the power of a compute node.

The action schema is:

```
schema = Schema({
 'resource_id': str,
 'state': str,
})
```

The *resource_id* references a baremetal node id (list of available ironic nodes is returned by this command: `ironic node-list`). The *state* value should either be *on* or *off*.

change_nova_service_state

Disables or enables the nova-compute service, deployed on a host

By using this action, you will be able to update the state of a nova-compute service. A disabled nova-compute service can not be selected by the nova scheduler for future deployment of server.

The action schema is:

```
schema = Schema({
 'resource_id': str,
 'state': str,
 'disabled_reason': str,
})
```

The *resource_id* references a nova-compute service name (list of available nova-compute services is returned by this command: `nova service-list --binary nova-compute`). The *state* value should either be *ONLINE* or *OFFLINE*. The *disabled_reason* references the reason why Watcher disables this nova-compute service. The value should be with *watcher_* prefix, such as *watcher_disabled*, *watcher_maintaining*.

migrate

Migrates a server to a destination nova-compute host

This action will allow you to migrate a server to another compute destination host. Migration type live can only be used for migrating active VMs. Migration type cold can be used for migrating non-active VMs as well active VMs, which will be shut down while migrating.

The action schema is:

```
schema = Schema({
 'resource_id': str,  # should be a UUID
 'migration_type': str,  # choices -> "live", "cold"
 'destination_node': str,
 'source_node': str,
})
```

The *resource_id* is the UUID of the server to migrate. The *source_node* and *destination_node* parameters are respectively the source and the destination compute hostname (list of available compute hosts is returned by this command: `nova service-list --binary nova-compute`).

Note

Nova API version must be 2.56 or above if *destination_node* parameter is given.

nop

logs a message

The action schema is:

```
schema = Schema({
 'message': str,
})
```

The *message* is the actual message that will be logged.

resize

Resizes a server with specified flavor.

This action will allow you to resize a server to another flavor.

The action schema is:

```
schema = Schema({
 'resource_id': str,  # should be a UUID
 'flavor': str,  # should be either ID or Name of Flavor
})
```

The *resource_id* is the UUID of the server to resize. The *flavor* is the ID or Name of Flavor (Nova accepts either ID or Name of Flavor to resize() function).

sleep

Makes the executor of the action plan wait for a given duration

The action schema is:

```
schema = Schema({
 'duration': float,
})
```

The *duration* is expressed in seconds.

volume_migrate

Migrates a volume to destination node or type

By using this action, you will be able to migrate cinder volume. Migration type swap can only be used for migrating attached volume. Migration type migrate can be used for migrating detached volume to the pool of same volume type. Migration type retype can be used for changing volume type of detached volume.

The action schema is:

```
schema = Schema({
    'resource_id': str,  # should be a UUID
    'migration_type': str,  # choices -> "swap", "migrate","retype"
    'destination_node': str,
    'destination_type': str,
})
```

The *resource_id* is the UUID of cinder volume to migrate. The *destination_node* is the destination block storage pool name. (list of available pools are returned by this command: `cinder get-pools`) which is mandatory for migrating detached volume to the one with same volume type. The *destination_type* is the destination block storage type name. (list of available types are returned by this command: `cinder type-list`) which is mandatory for migrating detached volume or swapping attached volume to the one with different volume type.

7.8.6 Workflow Engines

taskflow

Taskflow as a workflow engine for Watcher

Full documentation on taskflow at https://docs.openstack.org/taskflow/latest

7.8.7 Planners

node_resource_consolidation

Node Resource Consolidation planner implementation

This implementation preserves the original order of actions in the solution and try to parallelize actions which have the same action type.

Limitations

- This is a proof of concept that is not meant to be used in production

weight

Weight planner implementation

This implementation builds actions with parents in accordance with weights. Set of actions having a higher weight will be scheduled before the other ones. There are two config options to configure: action_weights and parallelization.

Limitations

- This planner requires to have action_weights and parallelization configs tuned well.

workload_stabilization

Workload Stabilization planner implementation

This implementation comes with basic rules with a set of action types that are weighted. An action having a lower weight will be scheduled before the other ones. The set of action types can be specified by weights in the `watcher.conf`. You need to associate a different weight to all available actions into the configuration file, otherwise you will get an error when the new action will be referenced in the solution produced by a strategy.

Limitations

- This is a proof of concept that is not meant to be used in production

7.8.8 Cluster Data Model Collectors

baremetal

Baremetal cluster data model collector

The Baremetal cluster data model collector creates an in-memory representation of the resources exposed by the baremetal service.

compute

Nova cluster data model collector

The Nova cluster data model collector creates an in-memory representation of the resources exposed by the compute service.

storage

Cinder cluster data model collector

The Cinder cluster data model collector creates an in-memory representation of the resources exposed by the storage service.

CHAPTER

EIGHT

WATCHER MANUAL PAGES

8.1 watcher-api

8.1.1 Service for the Watcher API

Author
openstack@lists.launchpad.net

Copyright
OpenStack Foundation

Manual section
1

Manual group
cloud computing

SYNOPSIS

watcher-api [options]

DESCRIPTION

watcher-api is a server daemon that serves the Watcher API

OPTIONS

General options

-h, help
Show the help message and exit

version
Print the version number and exit

-v, verbose
Print more verbose output

noverbose
Disable verbose output

-d, debug
Print debugging output (set logging level to DEBUG instead of default WARNING level)

nodebug
Disable debugging output

use-syslog
Use syslog for logging

nouse-syslog
Disable the use of syslog for logging

syslog-log-facility SYSLOG_LOG_FACILITY
syslog facility to receive log lines

config-dir DIR
Path to a config directory to pull *.conf files from. This file set is sorted, to provide a predictable parse order if individual options are over-ridden. The set is parsed after the file(s) specified via previous config-file, arguments hence over-ridden options in the directory take precedence. This means that configuration from files in a specified config-dir will always take precedence over configuration from files specified by config-file, regardless to argument order.

config-file PATH
Path to a config file to use. Multiple config files can be specified by using this flag multiple times, for example, config-file <file1> config-file <file2>. Values in latter files take precedence.

log-config-append PATH log-config PATH
The name of logging configuration file. It does not disable existing loggers, but just appends specified logging configuration to any other existing logging options. Please see the Python logging module documentation for details on logging configuration files. The log-config name for this option is deprecated.

log-format FORMAT
A logging.Formatter log message format string which may use any of the available logging.LogRecord attributes. Default: None

log-date-format DATE_FORMAT
Format string for %(asctime)s in log records. Default: None

log-file PATH, logfile PATH
(Optional) Name of log file to output to. If not set, logging will go to stdout.

log-dir LOG_DIR, logdir LOG_DIR
(Optional) The directory to keep log files in (will be prepended to log-file)

FILES

/etc/watcher/watcher.conf
Default configuration file for Watcher API

BUGS

- Watcher bugs are tracked in Launchpad at OpenStack Watcher

8.2 watcher-applier

8.2.1 Service for the Watcher Applier

Author
openstack@lists.launchpad.net

Copyright
OpenStack Foundation

Manual section
1

Manual group
cloud computing

SYNOPSIS

watcher-applier [options]

DESCRIPTION

Watcher Applier

OPTIONS

General options

-h, help
Show the help message and exit

version
Print the version number and exit

-v, verbose
Print more verbose output

noverbose
Disable verbose output

-d, debug
Print debugging output (set logging level to DEBUG instead of default WARNING level)

nodebug
Disable debugging output

use-syslog
Use syslog for logging

nouse-syslog
Disable the use of syslog for logging

syslog-log-facility SYSLOG_LOG_FACILITY
syslog facility to receive log lines

config-dir DIR
Path to a config directory to pull *.conf files from. This file set is sorted, to provide a predictable parse order if individual options are over-ridden. The

set is parsed after the file(s) specified via previous config-file, arguments hence over-ridden options in the directory take precedence. This means that configuration from files in a specified config-dir will always take precedence over configuration from files specified by config-file, regardless to argument order.

config-file PATH
Path to a config file to use. Multiple config files can be specified by using this flag multiple times, for example, config-file <file1> config-file <file2>. Values in latter files take precedence.

log-config-append PATH log-config PATH
The name of logging configuration file. It does not disable existing loggers, but just appends specified logging configuration to any other existing logging options. Please see the Python logging module documentation for details on logging configuration files. The log-config name for this option is deprecated.

log-format FORMAT
A logging.Formatter log message format string which may use any of the available logging.LogRecord attributes. Default: None

log-date-format DATE_FORMAT
Format string for %(asctime)s in log records. Default: None

log-file PATH, logfile PATH
(Optional) Name of log file to output to. If not set, logging will go to stdout.

log-dir LOG_DIR, logdir LOG_DIR
(Optional) The directory to keep log files in (will be prepended to log-file)

FILES

/etc/watcher/watcher.conf
Default configuration file for Watcher Applier

BUGS

- Watcher bugs are tracked in Launchpad at OpenStack Watcher

8.3 watcher-db-manage

The `watcher-db-manage` utility is used to create the database schema tables that the watcher services will use for storage. It can also be used to upgrade (or downgrade) existing database tables when migrating between different versions of watcher.

The Alembic library is used to perform the database migrations.

8.3.1 Options

This is a partial list of the most useful options. To see the full list, run the following:

```
watcher-db-manage --help
```

-h, --help
Show help message and exit.

--config-dir <DIR>
Path to a config directory with configuration files.

--config-file <PATH>
Path to a configuration file to use.

-d, --debug
Print debugging output.

-v, --verbose
Print more verbose output.

--version
Show the programs version number and exit.

upgrade, downgrade, stamp, revision, version, create_schema, purge
The *command* to run.

8.3.2 Usage

Options for the various *commands* for **watcher-db-manage** are listed when the *-h* or *--help* option is used after the command.

For example:

```
watcher-db-manage create_schema --help
```

Information about the database is read from the watcher configuration file used by the API server and conductor services. This file must be specified with the *--config-file* option:

```
watcher-db-manage --config-file /path/to/watcher.conf create_schema
```

The configuration file defines the database backend to use with the *connection* database option:

```
[database]
connection=mysql://root@localhost/watcher
```

If no configuration file is specified with the *--config-file* option, **watcher-db-manage** assumes an SQLite database.

8.3.3 Command Options

watcher-db-manage is given a command that tells the utility what actions to perform. These commands can take arguments. Several commands are available:

create_schema

-h, --help
Show help for create_schema and exit.

This command will create database tables based on the most current version. It assumes that there are no existing tables.

An example of creating database tables with the most recent version:

```
watcher-db-manage --config-file=/etc/watcher/watcher.conf create_schema
```

downgrade

-h, --help

Show help for downgrade and exit.

--revision <REVISION>

The revision number you want to downgrade to.

This command will revert existing database tables to a previous version. The version can be specified with the *--revision* option.

An example of downgrading to table versions at revision 2581ebaf0cb2:

```
watcher-db-manage --config-file=/etc/watcher/watcher.conf downgrade --
↪revision 2581ebaf0cb2
```

revision

-h, --help

Show help for revision and exit.

-m <MESSAGE>, **--message** <MESSAGE>

The message to use with the revision file.

--autogenerate

Compares table metadata in the application with the status of the database and generates migrations based on this comparison.

This command will create a new revision file. You can use the *--message* option to comment the revision.

This is really only useful for watcher developers making changes that require database changes. This revision file is used during database migration and will specify the changes that need to be made to the database tables. Further discussion is beyond the scope of this document.

stamp

-h, --help

Show help for stamp and exit.

--revision <REVISION>

The revision number.

This command will stamp the revision table with the version specified with the *--revision* option. It will not run any migrations.

upgrade

-h, --help

Show help for upgrade and exit.

--revision <REVISION>

The revision number to upgrade to.

This command will upgrade existing database tables to the most recent version, or to the version specified with the *--revision* option.

If there are no existing tables, then new tables are created, beginning with the oldest known version, and successively upgraded using all of the database migration files, until they are at the specified version. Note that this behavior is different from the *create_schema* command that creates the tables based on the most recent version.

An example of upgrading to the most recent table versions:

```
watcher-db-manage --config-file=/etc/watcher/watcher.conf upgrade
```

Note

This command is the default if no command is given to **watcher-db-manage**.

Warning

The upgrade command is not compatible with SQLite databases since it uses ALTER TABLE commands to upgrade the database tables. SQLite supports only a limited subset of ALTER TABLE.

version

-h, --help

Show help for version and exit.

This command will output the current database version.

purge

-h, --help

Show help for purge and exit.

-d, --age-in-days

The number of days (starting from today) before which we consider soft deleted objects as expired and should hence be erased. By default, all objects soft deleted are considered expired. This can be useful as removing a significant amount of objects may cause a performance issues.

-n, --max-number

The maximum number of database objects we expect to be deleted. If exceeded, this will prevent any deletion.

-t, --goal

Either the UUID or name of the goal to purge.

-e, --exclude-orphans

This is a flag to indicate when we want to exclude orphan objects from deletion.

`--dry-run`
: This is a flag to indicate when we want to perform a dry run. This will show the objects that would be deleted instead of actually deleting them.

This command will purge the current database by removing both its soft deleted and orphan objects.

8.4 watcher-decision-engine

8.4.1 Service for the Watcher Decision Engine

Author
: openstack@lists.launchpad.net

Copyright
: OpenStack Foundation

Manual section
: 1

Manual group
: cloud computing

SYNOPSIS

watcher-decision-engine [options]

DESCRIPTION

Watcher Decision Engine

OPTIONS

General options

-h, help
: Show the help message and exit

version
: Print the version number and exit

-v, verbose
: Print more verbose output

noverbose
: Disable verbose output

-d, debug
: Print debugging output (set logging level to DEBUG instead of default WARNING level)

nodebug
: Disable debugging output

use-syslog
: Use syslog for logging

nouse-syslog
: Disable the use of syslog for logging

syslog-log-facility SYSLOG_LOG_FACILITY
syslog facility to receive log lines

config-dir DIR
Path to a config directory to pull *.conf files from. This file set is sorted, to provide a predictable parse order if individual options are over-ridden. The set is parsed after the file(s) specified via previous config-file, arguments hence over-ridden options in the directory take precedence. This means that configuration from files in a specified config-dir will always take precedence over configuration from files specified by config-file, regardless to argument order.

config-file PATH
Path to a config file to use. Multiple config files can be specified by using this flag multiple times, for example, config-file <file1> config-file <file2>. Values in latter files take precedence.

log-config-append PATH log-config PATH
The name of logging configuration file. It does not disable existing loggers, but just appends specified logging configuration to any other existing logging options. Please see the Python logging module documentation for details on logging configuration files. The log-config name for this option is deprecated.

log-format FORMAT
A logging.Formatter log message format string which may use any of the available logging.LogRecord attributes. Default: None

log-date-format DATE_FORMAT
Format string for %(asctime)s in log records. Default: None

log-file PATH, logfile PATH
(Optional) Name of log file to output to. If not set, logging will go to stdout.

log-dir LOG_DIR, logdir LOG_DIR
(Optional) The directory to keep log files in (will be prepended to log-file)

FILES

/etc/watcher/watcher.conf
Default configuration file for Watcher Decision Engine

BUGS

- Watcher bugs are tracked in Launchpad at OpenStack Watcher

8.5 watcher-status

8.5.1 CLI interface for Watcher status commands

Synopsis

```
watcher-status <category> <command> [<args>]
```

Description

watcher-status is a tool that provides routines for checking the status of a Watcher deployment.

Options

The standard pattern for executing a **watcher-status** command is:

```
watcher-status <category> <command> [<args>]
```

Run without arguments to see a list of available command categories:

```
watcher-status
```

Categories are:

- upgrade

Detailed descriptions are below:

You can also run with a category argument such as upgrade to see a list of all commands in that category:

```
watcher-status upgrade
```

These sections describe the available categories and arguments for **Watcher-status**.

Upgrade

watcher-status upgrade check

Performs a release-specific readiness check before restarting services with new code. For example, missing or changed configuration options, incompatible object states, or other conditions that could lead to failures while upgrading.

Return Codes

Return code	Description
0	All upgrade readiness checks passed successfully and there is nothing to do.
1	At least one check encountered an issue and requires further investigation. This is considered a warning but the upgrade may be OK.
2	There was an upgrade status check failure that needs to be investigated. This should be considered something that stops an upgrade.
255	An unexpected error occurred.

History of Checks

2.0.0 (Stein)

- Sample check to be filled in with checks as they are added in Stein.

3.0.0 (Train)

- A check was added to enforce the minimum required version of nova API used.

CHAPTER

NINE

REST API VERSION HISTORY

This documents the changes made to the REST API with every microversion change. The description for each version should be a verbose one which has enough information to be suitable for use in user documentation.

9.1 1.0 (Initial version)

This is the initial version of the Watcher API which supports microversions.

A user can specify a header in the API request:

```
OpenStack-API-Version: infra-optim <version>
```

where <version> is any valid api version for this API.

If no version is specified then the API will behave as if version 1.0 was requested.

9.2 1.1

Added the parameters `start_time` and `end_time` to create audit request. Supported for start and end time of continuous audits.

9.3 1.2

Added `force` into create audit request. If `force` is true, audit will be executed despite of ongoing actionplan.

9.4 1.3

Added list data model API.

9.5 1.4

Added Watcher webhook API. It can be used to trigger audit with `event` type.

CHAPTER

TEN

GLOSSARY

This page explains the different terms used in the Watcher system.

They are sorted in alphabetical order.

10.1 Action

An *Action* is what enables Watcher to transform the current state of a *Cluster* after an *Audit*.

An *Action* is an atomic task which changes the current state of a target *Managed resource* of the OpenStack *Cluster* such as:

- Live migration of an instance from one compute node to another compute node with Nova
- Changing the power level of a compute node (ACPI level,)
- Changing the current state of a compute node (enable or disable) with Nova

In most cases, an *Action* triggers some concrete commands on an existing OpenStack module (Nova, Neutron, Cinder, Ironic, etc.).

An *Action* has a life-cycle and its current state may be one of the following:

- **PENDING** : the *Action* has not been executed yet by the *Watcher Applier*
- **ONGOING** : the *Action* is currently being processed by the *Watcher Applier*
- **SUCCEEDED** : the *Action* has been executed successfully
- **FAILED** : an error occurred while trying to execute the *Action*
- **DELETED** : the *Action* is still stored in the *Watcher database* but is not returned any more through the Watcher APIs.
- **CANCELLED** : the *Action* was in **PENDING** or **ONGOING** state and was cancelled by the *Administrator*

Some default implementations are provided, but it is possible to *develop new implementations* which are dynamically loaded by Watcher at launch time.

10.2 Action Plan

An *Action Plan* specifies a flow of *Actions* that should be executed in order to satisfy a given *Goal*. It also contains an estimated *global efficacy* alongside a set of *efficacy indicators*.

An *Action Plan* is generated by Watcher when an *Audit* is successful which implies that the *Strategy* which was used has found a *Solution* to achieve the *Goal* of this *Audit*.

In the default implementation of Watcher, an action plan is composed of a list of successive *Actions* (i.e., a Workflow of *Actions* belonging to a unique branch).

However, Watcher provides abstract interfaces for many of its components, allowing other implementations to generate and handle more complex *Action Plan(s)* composed of two types of Action Item(s):

- simple *Actions*: atomic tasks, which means it can not be split into smaller tasks or commands from an OpenStack point of view.
- composite Actions: which are composed of several simple *Actions* ordered in sequential and/or parallel flows.

An *Action Plan* may be described using standard workflow model description formats such as Business Process Model and Notation 2.0 (BPMN 2.0) or Unified Modeling Language (UML).

To see the life-cycle and description of *Action Plan* states, visit *the Action Plan state machine*.

10.3 Administrator

The *Administrator* is any user who has admin access on the OpenStack cluster. This user is allowed to create new projects for tenants, create new users and assign roles to each user.

The *Administrator* usually has remote access to any host of the cluster in order to change the configuration and restart any OpenStack service, including Watcher.

In the context of Watcher, the *Administrator* is a role for users which allows them to run any Watcher commands, such as:

- Create/Delete an *Audit Template*
- Launch an *Audit*
- Get the *Action Plan*
- Launch a recommended *Action Plan* manually
- Archive previous *Audits* and *Action Plans*

The *Administrator* is also allowed to modify any Watcher configuration files and to restart Watcher services.

10.4 Audit

In the Watcher system, an *Audit* is a request for optimizing a *Cluster*.

The optimization is done in order to satisfy one *Goal* on a given *Cluster*.

For each *Audit*, the Watcher system generates an *Action Plan*.

To see the life-cycle and description of an *Audit* states, visit *the Audit State machine*.

10.5 Audit Scope

An Audit Scope is a set of audited resources. Audit Scope should be defined in each Audit Template (which contains the Audit settings).

10.6 Audit Template

An *Audit* may be launched several times with the same settings (*Goal*, thresholds,). Therefore it makes sense to save those settings in some sort of Audit preset object, which is known as an *Audit Template*.

An *Audit Template* contains at least the *Goal* of the *Audit*.

It may also contain some error handling settings indicating whether:

- *Watcher Applier* stops the entire operation
- *Watcher Applier* performs a rollback

and how many retries should be attempted before failure occurs (also the latter can be complex: for example the scenario in which there are many first-time failures on ultimately successful *Actions*).

Moreover, an *Audit Template* may contain some settings related to the level of automation for the *Action Plan* that will be generated by the *Audit*. A flag will indicate whether the *Action Plan* will be launched automatically or will need a manual confirmation from the *Administrator*.

10.7 Availability Zone

Please, read the official OpenStack definition of an Availability Zone.

10.8 Cluster

A *Cluster* is a set of physical machines which provide compute, storage and networking resources and are managed by the same OpenStack Controller node. A *Cluster* represents a set of resources that a cloud provider is able to offer to his/her *customers*.

A data center may contain several clusters.

The *Cluster* may be divided in one or several *Availability Zone(s)*.

10.9 Cluster Data Model (CDM)

A *Cluster Data Model* (or CDM) is a logical representation of the current state and topology of the *Cluster Managed resources*.

It is represented as a set of *Managed resources* (which may be a simple tree or a flat list of key-value pairs) which enables Watcher *Strategies* to know the current relationships between the different *resources* of the *Cluster* during an *Audit* and enables the *Strategy* to request information such as:

- What compute nodes are in a given *Audit Scope*?
- What *Instances* are hosted on a given compute node?
- What is the current load of a compute node?
- What is the current free memory of a compute node?
- What is the network link between two compute nodes?
- What is the available bandwidth on a given network link?
- What is the current space available on a given virtual disk of a given *Instance* ?
- What is the current state of a given *Instance*?

-

In a word, this data model enables the *Strategy* to know:

- the current topology of the *Cluster*
- the current capacity for each *Managed resource*
- the current amount of used/free space for each *Managed resource*
- the current state of each *Managed resources*

In the Watcher project, we aim at providing a some generic and basic *Cluster Data Model* for each *Goal*, usable in the associated *Strategies* through a plugin-based mechanism which are called cluster data model collectors (or CDMCs). These CDMCs are responsible for loading and keeping up-to-date their associated CDM by listening to events and also periodically rebuilding themselves from the ground up. They are also directly accessible from the strategies classes. These CDMs are used to:

- simplify the development of a new *Strategy* for a given *Goal* when there already are some existing *Strategies* associated to the same *Goal*
- avoid duplicating the same code in several *Strategies* associated to the same *Goal*
- have a better consistency between the different *Strategies* for a given *Goal*
- avoid any strong coupling with any external *Cluster Data Model* (the proposed data model acts as a pivot data model)

There may be various *generic and basic Cluster Data Models* proposed in Watcher helpers, each of them being adapted to achieving a given *Goal*:

- For example, for a *Goal* which aims at optimizing the network *resources* the *Strategy* may need to know which *resources* are communicating together.
- Whereas for a *Goal* which aims at optimizing thermal and power conditions, the *Strategy* may need to know the location of each compute node in the racks and the location of each rack in the room.

Note however that a developer can use his/her own *Cluster Data Model* if the proposed data model does not fit his/her needs as long as the *Strategy* is able to produce a *Solution* for the requested *Goal*. For example, a developer could rely on the Nova Data Model to optimize some compute resources.

The *Cluster Data Model* may be persisted in any appropriate storage system (SQL database, NoSQL database, JSON file, XML File, In Memory Database,). As of now, an in-memory model is built and maintained in the background in order to accelerate the execution of strategies.

10.10 Controller Node

Please, read the official OpenStack definition of a Controller Node.

In many configurations, Watcher will reside on a controller node even if it can potentially be hosted on a dedicated machine.

10.11 Compute node

Please, read the official OpenStack definition of a Compute Node.

10.12 Customer

A *Customer* is the person or company which subscribes to the cloud provider offering. A customer may have several *Project(s)* hosted on the same *Cluster* or dispatched on different clusters.

In the private cloud context, the *Customers* are different groups within the same organization (different departments, project teams, branch offices and so on). Cloud infrastructure includes the ability to precisely track each customers service usage so that it can be charged back to them, or at least reported to them.

10.13 Goal

A *Goal* is a human readable, observable and measurable end result having one objective to be achieved.

Here are some examples of *Goals*:

- minimize the energy consumption
- minimize the number of compute nodes (consolidation)
- balance the workload among compute nodes
- minimize the license cost (some software have a licensing model which is based on the number of sockets or cores where the software is deployed)
- find the most appropriate moment for a planned maintenance on a given group of host (which may be an entire availability zone): power supply replacement, cooling system replacement, hardware modification,

10.14 Host Aggregate

Please, read the official OpenStack definition of a Host Aggregate.

10.15 Instance

A running virtual machine, or a virtual machine in a known state such as suspended, that can be used like a hardware server.

10.16 Managed resource

A *Managed resource* is one instance of *Managed resource type* in a topology with particular properties and dependencies on other *Managed resources* (relationships).

For example, a *Managed resource* can be one virtual machine (i.e., an *instance*) hosted on a *compute node* and connected to another virtual machine through a network link (represented also as a *Managed resource* in the *Cluster Data Model*).

10.17 Managed resource type

A *Managed resource type* is a type of hardware or software element of the *Cluster* that the Watcher system can act on.

Here are some examples of *Managed resource types*:

- Nova Host Aggregates

- Nova Servers
- Cinder Volumes
- Neutron Routers
- Neutron Networks
- Neutron load-balancers
- Sahara Hadoop Cluster
-

It can be any of the official list of available resource types defined in OpenStack for HEAT.

10.18 Efficacy Indicator

An efficacy indicator is a single value that gives an indication on how the *solution* produced by a given *strategy* performed. These efficacy indicators are specific to a given *goal* and are usually used to compute the *global efficacy* of the resulting *action plan*.

In Watcher, these efficacy indicators are specified alongside the goal they relate to. When a strategy (which always relates to a goal) is executed, it produces a solution containing the efficacy indicators specified by the goal. This solution, which has been translated by the *Watcher Planner* into an action plan, will see its indicators and global efficacy stored and would now be accessible through the *Watcher API*.

10.19 Efficacy Specification

An efficacy specification is a contract that is associated to each *Goal* that defines the various *efficacy indicators* a strategy achieving the associated goal should provide within its *solution*. Indeed, each solution proposed by a strategy will be validated against this contract before calculating its *global efficacy*.

10.20 Optimization Efficacy

The *Optimization Efficacy* is the objective measure of how much of the *Goal* has been achieved in respect with constraints and *SLAs* defined by the *Customer*.

The way efficacy is evaluated will depend on the *Goal* to achieve.

Of course, the efficacy will be relevant only as long as the *Action Plan* is relevant (i.e., the current state of the *Cluster* has not changed in a way that a new *Audit* would need to be launched).

For example, if the *Goal* is to lower the energy consumption, the *Efficacy* will be computed using several *efficacy indicators* (KPIs):

- the percentage of energy gain (which must be the highest possible)
- the number of *SLA violations* (which must be the lowest possible)
- the number of virtual machine migrations (which must be the lowest possible)

All those indicators are computed within a given timeframe, which is the time taken to execute the whole *Action Plan*.

The efficacy also enables the *Administrator* to objectively compare different *Strategies* for the same goal and same workload of the *Cluster*.

10.21 Project

Projects represent the base unit of ownership in OpenStack, in that all *resources* in OpenStack should be owned by a specific *project*. In OpenStack Identity, a *project* must be owned by a specific domain.

Please, read the official OpenStack definition of a Project.

10.22 Scoring Engine

A *Scoring Engine* is an executable that has a well-defined input, a well-defined output, and performs a purely mathematical task. That is, the calculation does not depend on the environment in which it is running - it would produce the same result anywhere.

Because there might be multiple algorithms used to build a particular data model (and therefore a scoring engine), the usage of scoring engine might vary. A metainfo field is supposed to contain any information which might be needed by the user of a given scoring engine.

10.23 SLA

SLA means Service Level Agreement.

The resources are negotiated between the *Customer* and the Cloud Provider in a contract.

Most of the time, this contract is composed of two documents:

- *SLA* : Service Level Agreement
- *SLO* : Service Level Objectives

Note that the *SLA* is more general than the *SLO* in the sense that the former specifies what service is to be provided, how it is supported, times, locations, costs, performance, and responsibilities of the parties involved while the *SLO* focuses on more measurable characteristics such as availability, throughput, frequency, response time or quality.

You can also read the Wikipedia page for SLA which provides a good definition.

10.24 SLA violation

A *SLA violation* happens when a *SLA* defined with a given *Customer* could not be respected by the cloud provider within the timeframe defined by the official contract document.

10.25 SLO

A Service Level Objective (SLO) is a key element of a *SLA* between a service provider and a *Customer*. SLOs are agreed as a means of measuring the performance of the Service Provider and are outlined as a way of avoiding disputes between the two parties based on misunderstanding.

You can also read the Wikipedia page for SLO which provides a good definition.

10.26 Solution

A *Solution* is the result of execution of a *strategy* (i.e., an algorithm). Each solution is composed of many pieces of information:

- A set of *actions* generated by the strategy in order to achieve the *goal* of an associated *audit*.

- A set of *efficacy indicators* as defined by the associated goal
- A *global efficacy* which is computed by the associated goal using the aforementioned efficacy indicators.

A *Solution* is different from an *Action Plan* because it contains the non-scheduled list of *Actions* which is produced by a *Strategy*. In other words, the list of Actions in a *Solution* has not yet been re-ordered by the *Watcher Planner*.

Note that some algorithms (i.e. *Strategies*) may generate several *Solutions*. This gives rise to the problem of determining which *Solution* should be applied.

Two approaches to dealing with this can be envisaged:

- **fully automated mode**: only the *Solution* with the highest ranking (i.e., the highest *Optimization Efficacy*) will be sent to the *Watcher Planner* and translated into concrete *Actions*.
- **manual mode**: several *Solutions* are proposed to the *Administrator* with a detailed measurement of the estimated *Optimization Efficacy* and he/she decides which one will be launched.

10.27 Strategy

A *Strategy* is an algorithm implementation which is able to find a *Solution* for a given *Goal*.

There may be several potential strategies which are able to achieve the same *Goal*. This is why it is possible to configure which specific *Strategy* should be used for each goal.

Some strategies may provide better optimization results but may take more time to find an optimal *Solution*.

10.28 Watcher Applier

This component is in charge of executing the *Action Plan* built by the *Watcher Decision Engine*.

See: *System Architecture* for more details on this component.

10.29 Watcher Database

This database stores all the Watcher domain objects which can be requested by the Watcher API or the Watcher CLI:

- Audit templates
- Audits
- Action plans
- Actions
- Goals

The Watcher domain being here *optimization of some resources provided by an OpenStack system*.

See *System Architecture* for more details on this component.

10.30 Watcher Decision Engine

This component is responsible for computing a set of potential optimization *Actions* in order to fulfill the *Goal* of an *Audit*.

It first reads the parameters of the *Audit* from the associated *Audit Template* and knows the *Goal* to achieve.

It then selects the most appropriate *Strategy* depending on how Watcher was configured for this *Goal*.

The *Strategy* is then executed and generates a set of *Actions* which are scheduled in time by the *Watcher Planner* (i.e., it generates an *Action Plan*).

See *System Architecture* for more details on this component.

10.31 Watcher Planner

The *Watcher Planner* is part of the *Watcher Decision Engine*.

This module takes the set of *Actions* generated by a *Strategy* and builds the design of a workflow which defines how-to schedule in time those different *Actions* and for each *Action* what are the prerequisite conditions.

It is important to schedule *Actions* in time in order to prevent overload of the *Cluster* while applying the *Action Plan*. For example, it is important not to migrate too many instances at the same time in order to avoid a network congestion which may decrease the *SLA* for *Customers*.

It is also important to schedule *Actions* in order to avoid security issues such as denial of service on core OpenStack services.

Some default implementations are provided, but it is possible to *develop new implementations* which are dynamically loaded by Watcher at launch time.

See *System Architecture* for more details on this component.