UNIVERSITÀ DEGLI STUDI DI BOLOGNA

ATTIVITÀ PROGETTUALE DI INFRASTRUCTURES FOR CLOUD COMPUTING AND BIG DATA

Assessment of Cloud Monitoring Tools

Di
Chiara Simoni

Index

1 Introduction

In the last few years, Cloud Computing has become an even more widespread concept through corporations and organizations, the possibility to acquire on-demand services over the internet as software, platforms and infrastructures is a solid reality.

Several examples of software platforms which deliver the concept of Infrastructure-as-a-service (IaaS) are Amazon Elastic Compute Cloud (EC2), Amazon Web Services, Google Compute Engine, and Microsoft Azure.

In cloud models, applications are executed in virtual machines that share the resources of the physical machine, called host, and as a result the management of cloud data centers and the volumes of traffic require precise monitoring.

The concept of monitoring is based on the understanding of the complete state of a machine in near real-time, in order to describe the resource usage and utilization, like CPU or memory, and as such, it is a non-trivial task, given the high number of involved variables and entities.

In order to achieve this knowledge from both the virtual platform and the physical infrastructure, we need metrics, which are abstractions of the actual measurements done on the system for the involved entities; these metrics are then stored in a database system in order to make planning and take decisions based on them.

Given the complexity of Cloud computing systems, monitoring systems have become a key necessity for decision-making improvement in scenarios where administrators have to react rapidly to saturation or failure events. For example it could be vital to monitor the health of instances, in order to provide efficient and swift recovery options in case something went wrong, as well as the status of the network. There are several objectives for this report.

First of all we're going to give a brief overview of monitoring systems (both proprietary and open-source) for Cloud infrastructures; after this overview, we're going to explore one of the most promising solutions, Prometheus.

Next, Openstack Monitoring solutions will be analyzed, and in this case the Monasca module will be the main focus. In addition, deployment specific information will be given for Monasca and Prometheus solutions; finally, we're going to summarize and discuss the results.

2 Cloud Monitoring

Among Cloud Monitoring solutions figures **Nagios**, an open-source monitoring tool for distributed systems and networks which has been adapted to cloud environments. It uses a plug-in-based model to extract information about different performance metrics from resources of physical and virtual machines.

It is based on a centralized approach that may lead to a single point of failure and a bottleneck situation on the server storing the metrics, which could affect the reliability and response time in which the administrators of data centers receive monitoring information.
There exists the possibility to extend Nagios using a DNX (Domain Name Exchange) to support large deployments, but it still suffers from high-availability problems: if the server goes down, everything is down.

Many other solutions take inspiration from Nagios, like **FlexACMS**, a modular monitoring solution designed for private clouds that supports the creation of monitoring slices, which are composed of a set of monitoring metrics and associated configurations used to monitor cloud slices on cloud platforms; however, fault tolerance and scalability issues are not addressed.

Another solution is **DARGOS**, a distributed cloud monitoring architecture which provides measures of the physical and virtual resources using push/pull approaches. The architecture has two main components: the Node Monitoring Agent (NMA), for collecting the resources statistics, and the Node Supervisor Agent (NSA), as a subscriber to monitoring information being published by NMA.

MonPaaS is another cloud monitoring tool which uses OpenStack as cloud stack and Nagios to allow monitoring the cloud, either by providers and consumers. The problem with MonPaaS is that its last version was implemented in a very old OpenStack version (Folsom) and it is not well documented. Its deployment requires that users have a great understanding of both, the source code of the monitoring tool and the OpenStack architecture and their APIs.

None of these solutions, was natively developed for OpenStack. It is advised that the monitoring tools are part of the Openstack environment: by not having the operator search, install, configure and integrate separately monitoring software, further efficiencies can be achieved. These efficiencies include reduced costs and simplification
.

Another promising solution is **Zabbix**, an enterprise-class open source distributed monitoring software, that monitors numerous parameters of a network and the health and integrity of servers. It uses a flexible notification mechanism that allows users to configure e-mail based alerts for virtually any event.
Some of Zabbix features are: the reporting and data visualisation based on the stored data; polling and trapping support; a web-based front-end that allows to access reports

and statistics from any location.

This software is popular either with small companies either with big ones, and it is free of cost being distributed under the GPL license.

Lastly, **Prometheus** is another tool for monitoring that will be explored more in depth. Prometheus is also an open-source systems monitoring originally built at SoundCloud. Since 2012 it has become a very popular software, maintained independently of any company. Each Prometheus server is standalone, not depending on network storage or other remote services. For this very reason it is realiable when other parts of your infrastructure are broken, and there is no need to setup extensive infrastructure to use it and it fits both machine-centric monitoring as well as monitoring of highly dynamic service-oriented architectures. In a world of microservices, its support for multi-dimensional data collection and querying is a particular strength.

3 Prometheus

Prometheus is a leading open source monitoring and alerting tool. The software was created because of the need to monitor multiple microservices that might be running in the system.
Its architecture is modular and comes with several readily available modules called *exporters*.

Exporters are libraries and integrations which help in exporting existing metrics from third-party systems as Prometheus metrics, this is useful in case it is not feasible to gather metrics directly using Prometheus itself, for example, Linux system stats.

An exporter is basically a collector: there are many available from Prometheus website, but if that is not the case, a custom one can be created. The concept behind exporters is to integrate Prometheus client libraries with third party software. There are exporters for databases, messaging systems, HTTP, APIs.

Prometheus is written in the Go language, and it comes with easily distributed binaries.

In addition, Prometheus was the second project to be accepted by the Cloud Native Computing Foundation (CNCF) after Kubernetes, the leading open source solution for container orchestration.
The Cloud Native Computing Foundation (CNCF) is a non-profit organisation that promotes the advancing of the development of cloud native applications and services by creating a new set of common container technologies guided by technical merit and end user value, inspired by Internet-scale computing.

Prometheus integrates with Kubernetes, to support service discovery and monitoring of dynamically scheduled services, and Kubernetes also supports Prometheus natively. One of the main reasons Prometheus growth has been so significant is thanks to the contributions from the community.

3.1 Architecture

The key modules are as follows.

- Prometheus Server - The heart of the system. It collects metrics, at regular periods of time, from multiple nodes and stores them locally. The core concept is "scraping", that means to invoke the metrics endpoints to of the various nodes that were configured to be monitored. The nodes in question are able to expose the metrics over the endpoints that the Prometheus server scrapes, using the exporters previously mentioned.
- Push gateway - In case the nodes are not exposing an endpoint from which the Prometheus server can collect the metrics, the Prometheus ecosystem has a push gateway. This gateway API is useful for one-off jobs that run, capture the

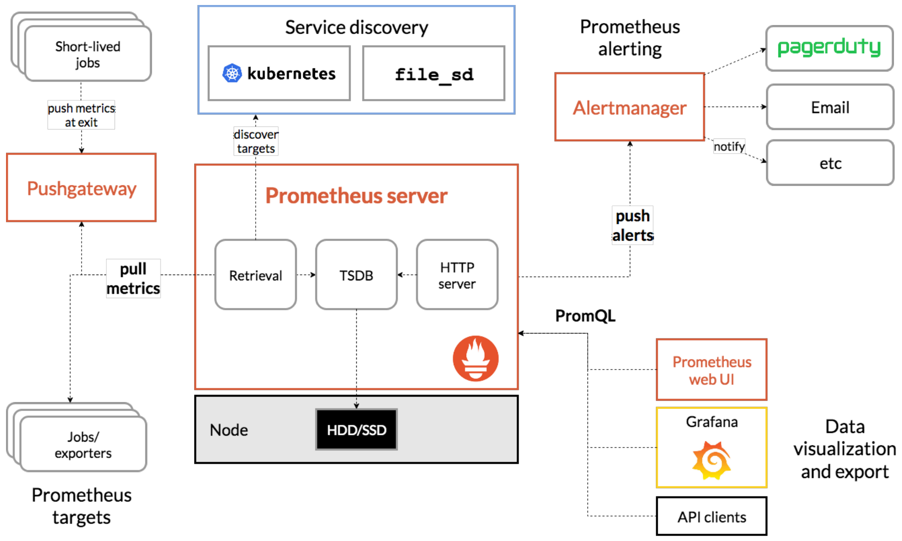


Figure 1: Prometheus architecture

data, transform that data into the Prometheus data format and then push that data into the Prometheus server.

- ALERT MANAGER - It enables the definition of alarms, alerts on those metrics, so that the user can be notified in case of any discrepancies. This is the job of the alerts manager, which stores not just the alert levels but also can deliver these alerts to you over multiple channels like SMS, email, Slack, etc.
- INTERFACE - Prometheus comes with its own user interface that the user can use to check on the configuration, nodes and graphs. Additionally, it is now compatible with Grafana, a leading open source visualisation application, so that Prometheus data is available for viewing inside Grafana. It has to be installed and configured manually.

3.2 Data Model

Prometheus stores all data as *time series*: a unique stream of timestamped values, identified by the same metric and the same set of labels, an optional key-value pair.

The metric name specifies the general feature of a system that is measured (e.g. http_requests_total is the total number of HTTP requests received).

The role of labels is fundamental: by using labels Prometheus identifies different instantiations of the same metric. The query language allows filtering and aggregation based on these dimensions. Changing any label value, including adding or removing a label, will create a new time series.

In summary, given a metric name and a set of labels, time series are frequently identified using this notation:

```
<metric name>{<label name>=<label value>, ...}
```

3.3 Libraries

The Prometheus client libraries offer four core metric types. The Prometheus server does not yet make use of the type information and flattens all data into untyped time series. This may change in the future.

- Counter - a cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart. For example, you can use a counter to represent the number of requests served, tasks completed, or errors.
- Gauge - a metric that represents a single numerical value that can arbitrarily go up and down, typically used for measured values like temperatures or current memory usage, but also "counts" that can go up and down, like the number of concurrent requests.
- Histogram - samples observations (usually things like request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values.
- Summary - samples observations (usually things like request durations and response sizes). While it also provides a total count of observations and a sum of all observed values, it calculates configurable quantiles over a sliding time window.

For more detailed information consult the Prometheus Documentation. All the client libraries have official support for Go, Java, Python and Ruby programming languages.

3.4 Targets

In Prometheus terms, an endpoint you can scrape (monitor) is called an *instance*, usually corresponding to a single process. A collection of instances with the same purpose, a process replicated for scalability or reliability for example, is called a *job*. For example, an API server job with four replicated instances:

job: api-server
instance 1: `1.2.3.4:5670`
instance 2: `1.2.3.4:5671`
instance 3: `5.6.7.8:5670`
instance 4: `5.6.7.8:5671`

When Prometheus scrapes a target, it attaches some labels automatically to the scraped time series which serve to identify the scraped target:

- JOB - The configured job name that the target belongs to.
- INSTANCE - The `<host>:<port>` part of the target's URL that was scraped.

Prometheus allows to define one or more targets, and these can be either on the same system where Prometheus was installed, so defined with the form `localhost:<port>`, or can be external targets, so in the configuration file they will be selected using the `<IP_ADDRESS>:<port>`.

The only difference is how to specify the targets in the configuration file, the monitoring will then proceed in the same way.

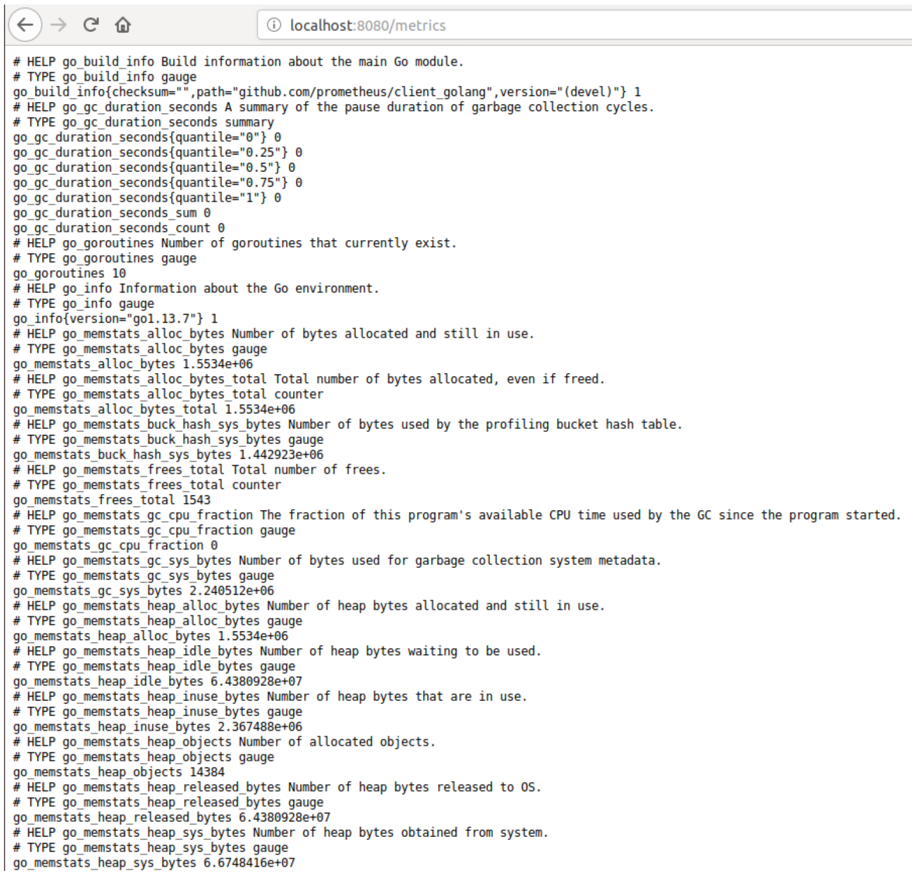
In the first case, "local" targets can be defined by running different processes, as shown in the official documentation, using `go` scripts that launch three client instances that are to be monitored, and their information is collected and available to the Prometheus server.

The `static_configs` section of the targets will be defined as follows:

```
static_configs:
      - targets: ['localhost:8080', 'localhost:8081']
        labels:
          group: 'production'

      - targets: ['localhost:8082']
        labels:
          group: 'canary'
```

The steps needed to do so was to clone locally the "example-random" repository from git, and launch the script from three different terminals, using the command `./<script> -listen-address=: 808x`.

It is critical of course to have available the right libraries to run each script: the example suggested in the tutorial used the go language, so it was mandatory to install the go client libraries and ensure that the environment was running correctly. The user can access directly at the nodes metrics by navigating to the respective localhost:808x address.



```
# HELP go_build_info Build information about the main Go module.
# TYPE go_build_info gauge
go_build_info{checksum="",path="github.com/prometheus/client_golang",version="(devel)"} 1
# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0
go_gc_duration_seconds{quantile="0.25"} 0
go_gc_duration_seconds{quantile="0.5"} 0
go_gc_duration_seconds{quantile="0.75"} 0
go_gc_duration_seconds{quantile="1"} 0
go_gc_duration_seconds_sum 0
go_gc_duration_seconds_count 0
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 10
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.13.7"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 1.5534e+06
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 1.5534e+06
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.442923e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 1543
# HELP go_memstats_gc_cpu_fraction The fraction of this program's available CPU time used by the GC since the program started.
# TYPE go_memstats_gc_cpu_fraction gauge
go_memstats_gc_cpu_fraction 0
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 2.240512e+06
# HELP go_memstats_heap_alloc_bytes Number of heap bytes allocated and still in use.
# TYPE go_memstats_heap_alloc_bytes gauge
go_memstats_heap_alloc_bytes 1.5534e+06
# HELP go_memstats_heap_idle_bytes Number of heap bytes waiting to be used.
# TYPE go_memstats_heap_idle_bytes gauge
go_memstats_heap_idle_bytes 6.4380928e+07
# HELP go_memstats_heap_inuse_bytes Number of heap bytes that are in use.
# TYPE go_memstats_heap_inuse_bytes gauge
go_memstats_heap_inuse_bytes 2.367488e+06
# HELP go_memstats_heap_objects Number of allocated objects.
# TYPE go_memstats_heap_objects gauge
go_memstats_heap_objects 14384
# HELP go_memstats_heap_released_bytes Number of heap bytes released to OS.
# TYPE go_memstats_heap_released_bytes gauge
go_memstats_heap_released_bytes 6.4380928e+07
# HELP go_memstats_heap_sys_bytes Number of heap bytes obtained from system.
# TYPE go_memstats_heap_sys_bytes gauge
go_memstats_heap_sys_bytes 6.6748416e+07
```

Figure 2: Metrics available on the `<localhost>:<port>` address of a target

Prometheus also allows to monitor other external targets. However, to do so, another additional step is mandatory: to install the Node Exporter. This exporter exposes a wide variety of hardware- and kernel-related metrics to monitor Linux Hosts.

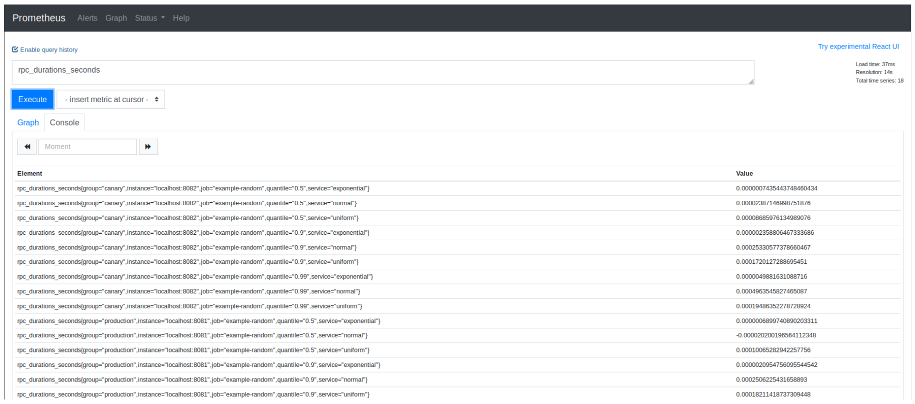
To install it, the user simply needs to download it from the official page and extract it. After doing so, move into the directory and run the `./node_exporter` command.

On the terminal, the port on which the node_exporter will be listening should be displayed.

This is the port that has to be added to the `IP_ADDRESS` when specifying the target in the prometheus.yml file. The target should be defined like this:

```
static_configs:
      - targets: ['192.168.1.x:<port_of_the_node_exporter>']
```

Targets informations is also available in the *Targets* section of the Prometheus Interface, more metrics about the node can be accessed by clicking on the provided link.

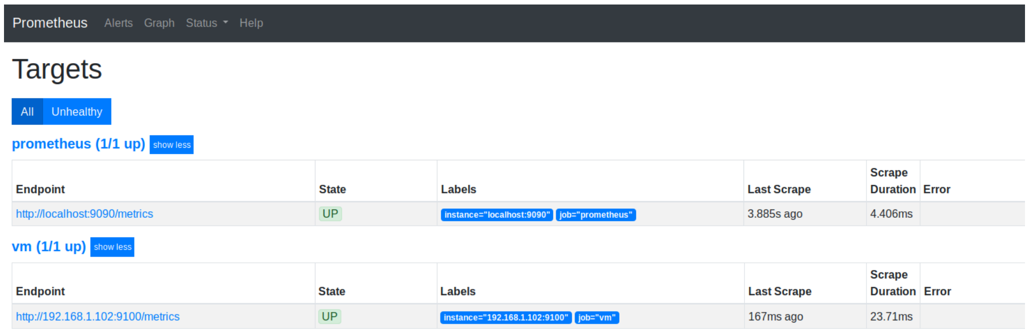


rpc_durations_seconds

Load time: 37ms
Resolution: 14s
Total time series: 18

Element	Value
rpc_durations_seconds{group="canary",instance="localhost:8082",job="example-random",quantile="0.5",service="exponential"}	0.0000007435443748460434
rpc_durations_seconds{group="canary",instance="localhost:8082",job="example-random",quantile="0.5",service="normal"}	0.00002387146998751876
rpc_durations_seconds{group="canary",instance="localhost:8082",job="example-random",quantile="0.5",service="uniform"}	0.00008685976134989076
rpc_durations_seconds{group="canary",instance="localhost:8082",job="example-random",quantile="0.9",service="exponential"}	0.000002358806467333686
rpc_durations_seconds{group="canary",instance="localhost:8082",job="example-random",quantile="0.9",service="normal"}	0.00025330577378660467
rpc_durations_seconds{group="canary",instance="localhost:8082",job="example-random",quantile="0.9",service="uniform"}	0.0001720127288695451
rpc_durations_seconds{group="canary",instance="localhost:8082",job="example-random",quantile="0.99",service="exponential"}	0.0000049881631088716
rpc_durations_seconds{group="canary",instance="localhost:8082",job="example-random",quantile="0.99",service="normal"}	0.0004963545827465087
rpc_durations_seconds{group="canary",instance="localhost:8082",job="example-random",quantile="0.99",service="uniform"}	0.00019486352278728924
rpc_durations_seconds{group="production",instance="localhost:8081",job="example-random",quantile="0.5",service="exponential"}	0.0000006899740890203311
rpc_durations_seconds{group="production",instance="localhost:8081",job="example-random",quantile="0.5",service="normal"}	-0.00002020196564112348
rpc_durations_seconds{group="production",instance="localhost:8081",job="example-random",quantile="0.5",service="uniform"}	0.00010065282942257756
rpc_durations_seconds{group="production",instance="localhost:8081",job="example-random",quantile="0.9",service="exponential"}	0.0000020954756095544542
rpc_durations_seconds{group="production",instance="localhost:8081",job="example-random",quantile="0.9",service="normal"}	0.0002506225431658893
rpc_durations_seconds{group="production",instance="localhost:8081",job="example-random",quantile="0.9",service="uniform"}	0.00018211418737309448

Figure 3: Output on the expression console using multiple targets metrics



Targets

prometheus (1/1 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9090/metrics	UP	instance="localhost:9090" job="prometheus"	3.885s ago	4.406ms	

vm (1/1 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://192.168.1.102:9100/metrics	UP	instance="192.168.1.102:9100" job="vm"	167ms ago	23.71ms	

Figure 4: Output on the expression console using multiple targets metrics

3.5 Alert Manager

The Alertmanager handles alerts sent by client applications such as the Prometheus server. It takes care of deduplicating, grouping, and routing them to the correct receiver integration such as email, PagerDuty, or OpsGenie. It also takes care of silencing and inhibition of alerts.

- Grouping - categorizes alerts of similar nature into a single notification. This is especially useful during larger outages when many systems fail at once and hundreds to thousands of alerts may be firing simultaneously.
- Inhibition - a concept of suppressing notifications for certain alerts if certain other alerts are already firing.
- Silence - a way to mute alerts for a given time.
- High Availability - Alertmanager supports configuration to create a cluster for high availability. This can be configured using the –cluster-* flags.

3.6 Integration with Grafana

Grafana is an analytics platform that allows the user to query, visualize, alert and understand metrics from any location. It is an extremely elastic solution: any datasource can be selected, and then any number of dashboard can be defined. Grafana interface is intuitive and visually appeasing, in a few steps the user is able to create a custom dashboard inserting the metric and other parameters that may be useful for a deeper understanding of the data of interest.

Prometheus supports integration with Grafana. However the user has to manually install and configure the service to use it. The steps to install Grafana are available from the official website. After starting the service the service will be available at the port 3000 of the localhost.

Navigate to it and login using the default credentials (admin:admin). After this first access Grafana will suggest the user to change the password. Once logged, the Prometheus datasource has to be added:

1. Click on the Grafana logo to open the sidebar menu.
2. Click on "Data Sources" in the sidebar.
3. Click on "Add New".
4. Select "Prometheus" as the type.
5. Set the appropriate Prometheus server URL (for example, http://localhost:9090/)
6. Adjust other data source settings as desired (for example, turning the proxy access off).
7. Click "Add" to save the new data source.

4 Openstack Cloud Monitoring

OpenStack is a cloud computing system that controls large pools of compute, storage and networking resources spread over a datacenter, all managed using APIs and common authentication mechanisms. Openstack is defined as an Infrastructure-as-a-Service, but goes behind this concept: based on a modular architecture, the user can add modules depending on the needs.

Using the configuration file Openstack infrastructure will be then configured will all the modules up and running. Additional component may provide orchestration, fault management and service management, as well as monitoring services.

Access to OpenStack infrastructure is available via the OpenStack API or through a dashboard. OpenStack works with popular enterprise and open source technologies making it ideal for heterogeneous infrastructure.

From small to large deployments of OpenStack, monitoring tools are very important. OpenStack environments can include a lot of physical nodes, and to manage the architecture and making sure that everything is running correctly, monitoring is a key process of management.

During the monitoring, OpenStack deployments can get notifications or alarms of each and every issue that occurs in the system, and the administrator is able to plan and take actions to smooth operations and maintain the customer SLAs.

4.1 OpenStack service overview

OpenStack embraces a modular architecture to provide a set of core services that facilitates scalability and elasticity as core design tenets. This chapter briefly reviews OpenStack components, their use cases and security considerations.

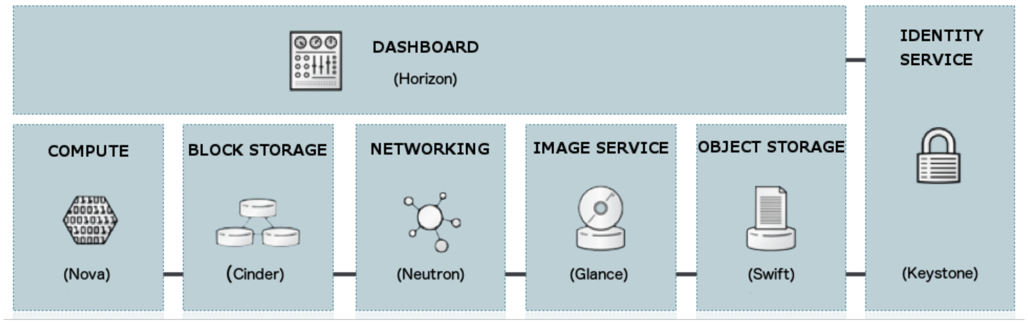


Figure 5: OpenStack service overview

- Compute - Nova supports the managements of virtual machine instances host multi-tiered applications, dev or test environments. The Compute service facilitates this management through an abstraction layer that interfaces with supported hypervisors. Support for strong instance isolation, secure communication between Compute sub-components, and resiliency of public-facing API endpoints is enforced.

- Object Storage - Swift provides support for storing and retrieving arbitrary data in the cloud. The Object Storage service provides both a native API and an Amazon Web Services S3-compatible API. Object storage differs from traditional file system storage and is best used for static data such as media files (MP3s, images, or videos), virtual machine images, and backup files.
- Block Storage - Cinder provides persistent block storage for compute instances. The service is responsible for managing the life-cycle of block devices, from the creation and attachment of volumes to instances, to their release. Security considerations for block storage are similar to that of object storage.
- Network - Neutron provides various networking services to cloud users (tenants) such as IP address management, DNS, DHCP, load balancing, and security groups (network access rules, like firewall policies). It allows cloud tenants to manage their guest network configurations. Security concerns with the networking service include network traffic isolation, availability, integrity, and confidentiality.
- Dashboard - Horizon provides a web-based interface for both cloud administrators and cloud tenants. Using this interface, administrators and tenants can provision, manage, and monitor cloud resources. The dashboard is commonly deployed in a public-facing manner with all the usual security concerns of public web portals.
- Identity Service - Keystone is a shared service that provides authentication and authorization services throughout the entire cloud infrastructure. The Identity service has support for multiple forms of authentication. Security concerns with the Identity service include trust in authentication, the management of authorization tokens, and secure communication.

4.2 Ceilometer before Monasca

Within OpenStack exists a telemetry project that provides a framework to meter and collect infrastructure metrics such as CPU, network, and storage utilization. This project aims to provide the infrastructure needed to collect measurements within OpenStack and then send them to different targets.

Some of the components that are part of this project are: Gnocchi, that was developed to capture measurement data in a time series format to optimise storage and querying and is intended to replace the existing metering database interface; additionally, Aodh is the alarming service which can send alerts when user defined rules are broken and

Panko, that is the event storage project designed to capture document-oriented data such as logs and system event actions.

Ceilometer however, is the main focus of this project. It provides two methods to collect data: since most components are not able to support a notification system through which these events can be sent, Ceilometer uses polling agents for APIs or a local hypervisor to collect information at a regular interval.

The first method is less preferred due to the load it can generate on the API services. The data collected by agents is then published in a pipeline managed by the Notification Agent, that transforms and manipulates data. The processed event and metering data captured by the polling agents are collected by the Collector and after the data validation, sent to the specified target: database, file or http.

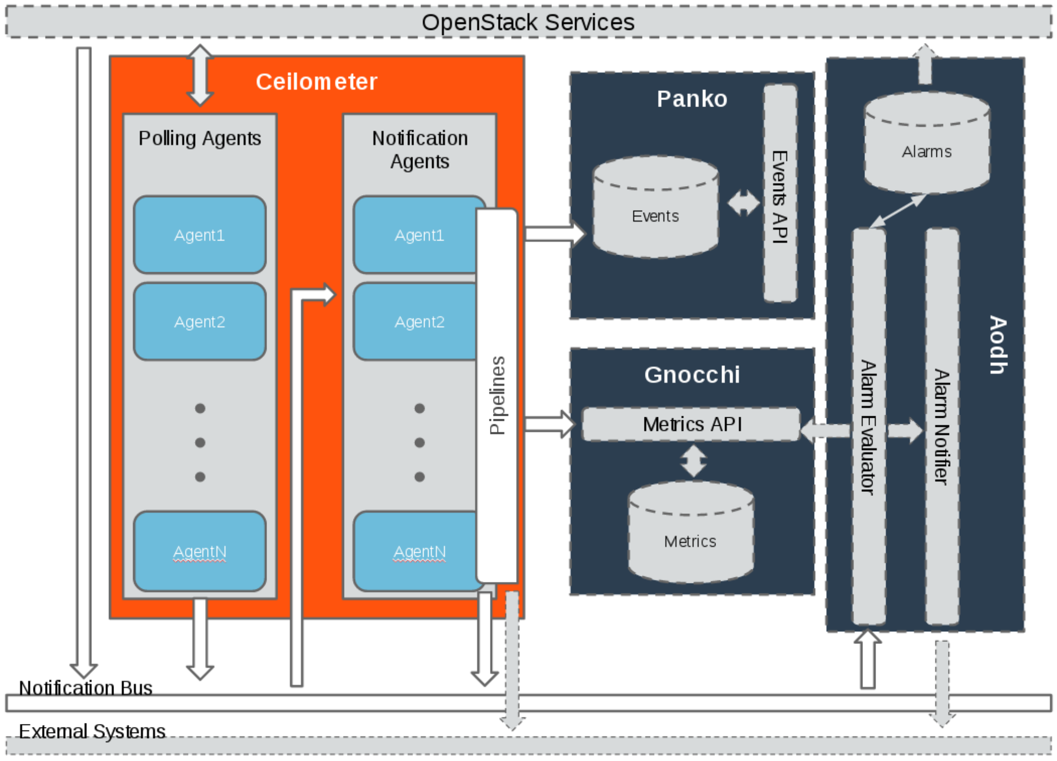


Figure 6: Ceilometer architecture

In summary, the main components of Ceilometer are:

- Polling agent - to poll Openstack services and build meters
- Notification Agent - program that checks if notifications are available from the message queue and converts them into event and samples to transform (pipeline)
- Collector - gathers and records events and data by notification and polling agents

- API - the main purpose is to analize data recorded by the collector

Alarms are also provided: they are set when a metric crosses a threshold and triggers an action, for example, sending the alarm information to an external server.

The Alarm Evaluator is the one responsible to check if there are changes in alarm states and it does so by querying the databases: this process allows an evaluation of the alarm in regards to the metrics stored within a certain period.

However, this means that if an Openstack server produces an alarm for events every once in a while, the evaluation would not be efficient, and it would mean to later add all the event driven alarms to Openstack.

Ceilometer also works along with Heat, the Openstack Orchestration service, in the way that alarms are provided to Heat, and can be triggered to perform actions.

5 Monasca

5.1 Architecture

Monasca is a large framework that involves all aspects of monitoring, including alarms, statistics, measurements specifically for all OpenStack components. Defined as "Monitoring as a Service" (MONaaS), Monasca has the main objective to allow tenants to define what should be measured, what statistics should be collected, what should trigger an alarm, and the best notification method.

The teams responsible for Monasca development mainly belong to major companies, like Hewlett Packard, Fujitsu, Cisco, Rackspace.

The core component is the Monasca API, the gateway for all interactions with other modules. The metrics are usually collected by the Monasca Agent on the host and then sent to the API, that then publishes the metrics to the Kafka queue. Then, the Monasca Persister consumes them and writes them to the specific database. The Monasca Threshold Engine also consumes the metrics and uses them to evaluate alarms. The access to metrics is achieved through the Horizon plugin or the Monasca CLI. The choice to select the database in which store them is also available.

In summary, the main components of Monasca are:

- Monitoring Agent - a Python based monitoring agent that consists of several sub-components and supports system metric, such as cpu utilization and available memory, and many built-in checks for services such as MySQL, RabbitMQ, etc.
- Monasca Api - a RESTful API for monitoring that is primarily focused on the concepts metrics, statistics, alarms and notification methods. It has both Java and Python implementations available.
- Persister - it consumes metrics and stores them in the Metrics and Alarms database. It has both Java and Python implementations available.
- Message Queue - receives published metrics from the Monitoring API and messages from the Threshold Engine that are consumed by other components, such as the Persister and Notification Engine. Currently, a Kafka based MessageQ is supported.
- Metrics and Alarms Database - stores metrics and the alarm state history.
- Config Database - stores a lot of the configuration and other information in the system. Currently, MySQL is supported.
- Monasca Threshold Engine - for computing threshold on metrics and determining alarm states.
- Monasca Transform and Aggregator - engine based on Apache Spark

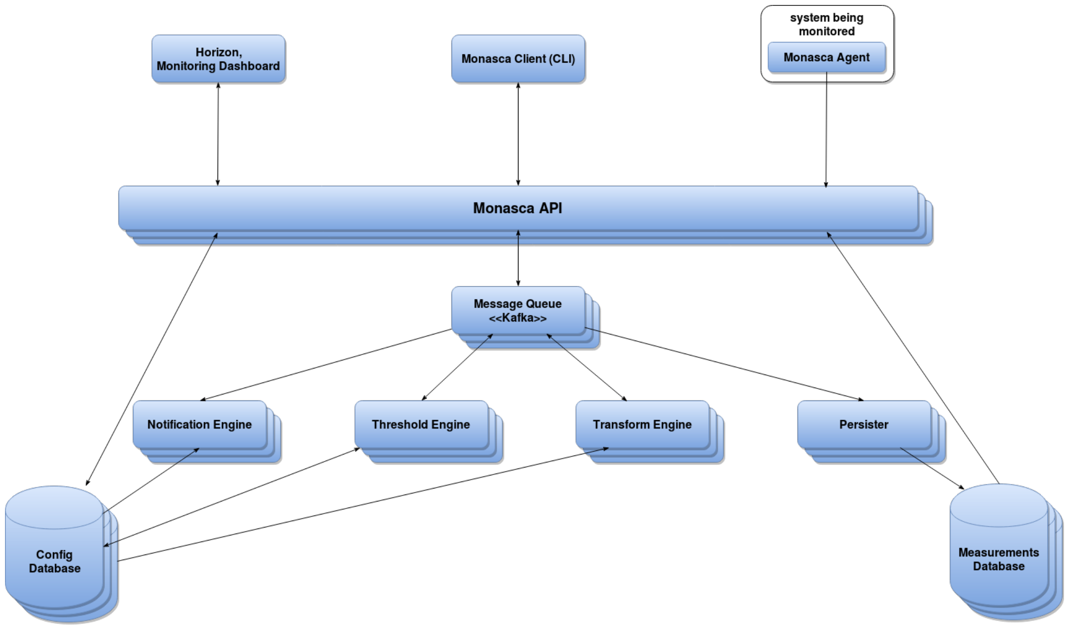


Figure 7: Monasca architecture

and metrics aggregator.

- Monasca Client - a Python command line client and library that communicates and controls the Monitoring API. The Monitoring Client also has a Python library, "monascaclient" similar to the other OpenStack clients, that can be used to quickly build additional capabilities.
- Monitoring UI - a Horizon dashboard for visualizing the overall health and status of an OpenStack cloud.
- Grafana Integration - with keystone authentication it allows to plot data gathered on Grafana Dashboard.
- Monitoring UI - a Horizon dashboard for visualizing the overall health and status of an OpenStack cloud.

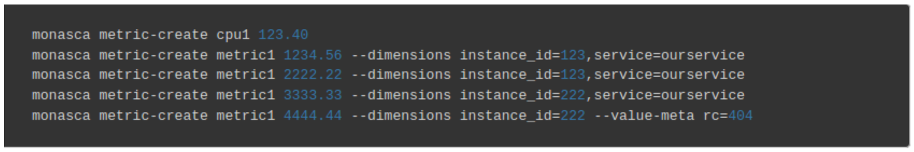
5.2 Metrics

Monasca is mainly focused on the concept of "metric", which is uniquely identified by a name and a specific set of dimensions. The dimension is identified by a (key, value) pair. A *measurement* is a metric instance with a value and a timestamp. Using the dimension, Monasca API can be queried to find related measurements.

In addition, a measurement may also contain extra data about the value, which is known as *value_ meta*: if it is included with a measurement, it is returned when the measurement is read via the Monasca API. Unlike dimensions, value a choice to meta fields are not as a key to find measurements from the Monasca API.

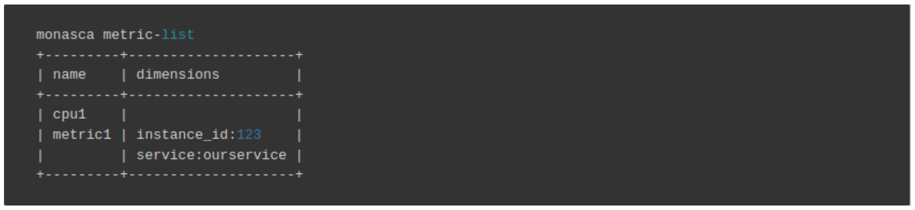
Metrics can be defined manually from the Monasca CLI using the `monasca metric-create` command. There are also other methods to list available metrics or measurements. All the documentation is available on Openstack Docs.

`$ monasca metric-create`



```
monasca metric-create cpu1 123.40
monasca metric-create metric1 1234.56 --dimensions instance_id=123,service=ourservice
monasca metric-create metric1 2222.22 --dimensions instance_id=123,service=ourservice
monasca metric-create metric1 3333.33 --dimensions instance_id=222,service=ourservice
monasca metric-create metric1 4444.44 --dimensions instance_id=222 --value-meta rc=404
```

`$ monasca metric-list`



```
monasca metric-list
+---------+-------------------+
| name    | dimensions        |
+---------+-------------------+
| cpu1    |                   |
| metric1 | instance_id:123   |
|         | service:ourservice |
+---------+-------------------+
```

Following, the schema of how metrics are used in Monasca:

1. A metric is posted to the Monasca API.
2. The Monasca API authenticates and validates the request and publishes the metric to the the Message Queue.
3. The Persister consumes the metric from the Message Queue and stores in the Metrics Store.
4. The Transform Engine consumes the metrics from the Message Queue, performs transform and aggregation operations on metrics, and publishes metrics that it creates back to Message Queue.
5. The Threshold Engine consumes metrics from the Message Queue and evaluates alarms. If a state change occurs in an alarm, an "alarm-state-transitioned-event" is published to the Message Queue.
6. The Notification Engine consumes "alarm-state-transitioned-events" from the Message Queue, evaluates whether they have a Notification Method associated with it, and sends the appropriate notification, such as email.
7. The Persister consumes the "alarm-state-transitioned-event" from the Message Queue and stores it in the Alarm State History Store.

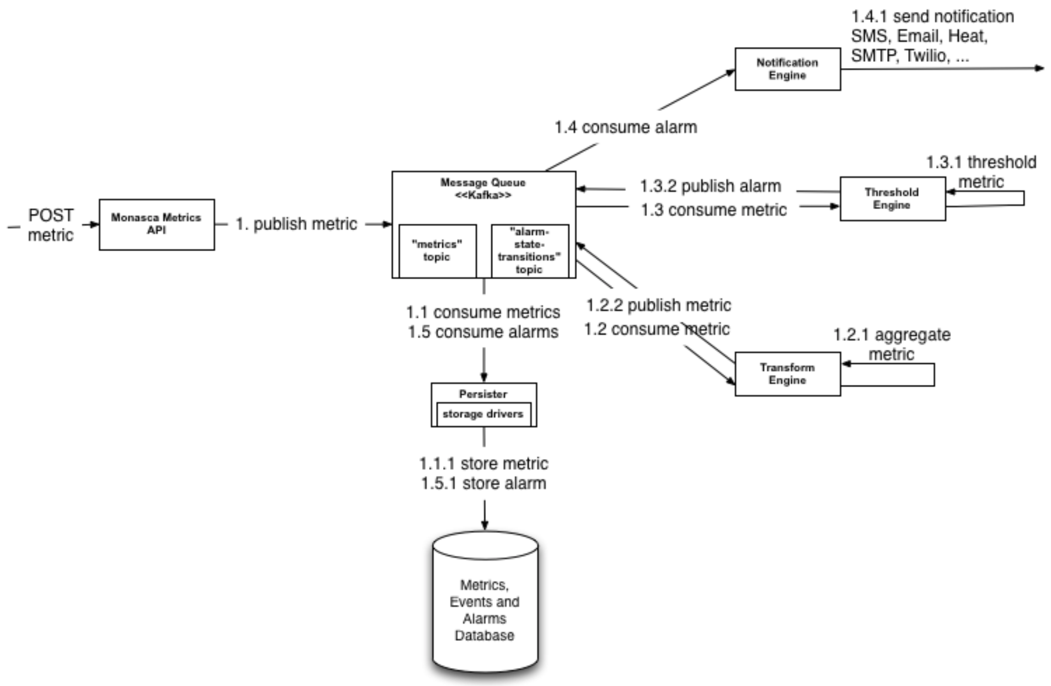


Figure 8: POST metrics schema

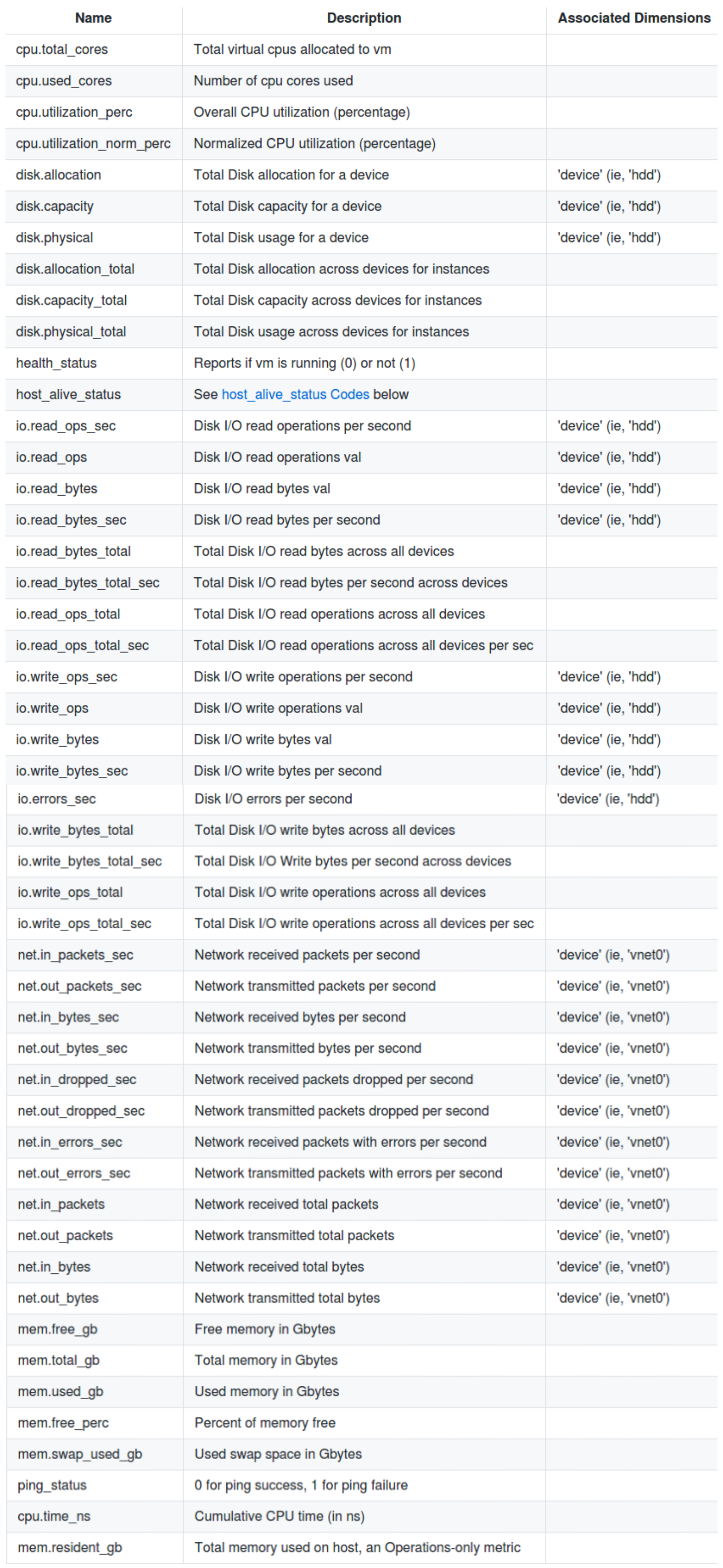
5.3 Libvirt plugin

The Libvirt plugin provides metrics for virtual machines when run on the hypervisor server, as in the case under examination.

The plugin provides two sets of metrics per measurement: one designed for the owner of the VM, and one intended for the owner of the hypervisor server. The monasca-setup program will configure the Libvirt plugin. If nova-compute is running, its nova.conf config file is readable by the Monasca Agent user (default: 'mon-agent'), and python-novaclient is installed.

Since the installation for this project was achieved through the Monasca API Devstack plugin, the setup was done automatically by the script and once terminated, all the plugin and libraries, such as the Libvirt metrics library, were available. It is also important to take into account the fact that Openstack is configured on a single-node, so as said in the beginning, the metrics, nova and all the services are all available.

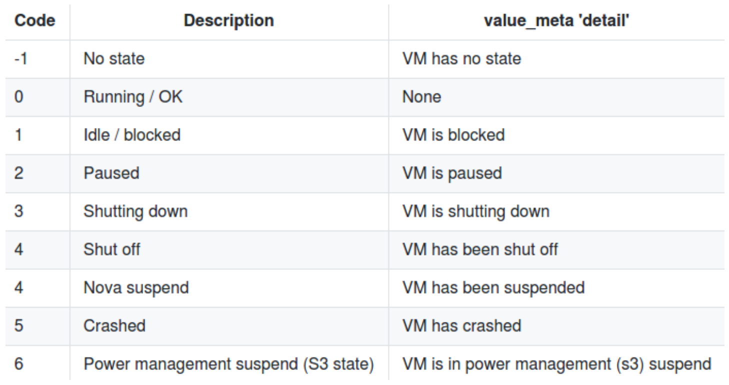
Following the list of the metrics available from Libvirt:



Name	Description	Associated Dimensions
cpu.total_cores	Total virtual cpus allocated to vm	
cpu.used_cores	Number of cpu cores used	
cpu.utilization_perc	Overall CPU utilization (percentage)	
cpu.utilization_norm_perc	Normalized CPU utilization (percentage)	
disk.allocation	Total Disk allocation for a device	'device' (ie, 'hdd')
disk.capacity	Total Disk capacity for a device	'device' (ie, 'hdd')
disk.physical	Total Disk usage for a device	'device' (ie, 'hdd')
disk.allocation_total	Total Disk allocation across devices for instances	
disk.capacity_total	Total Disk capacity across devices for instances	
disk.physical_total	Total Disk usage across devices for instances	
health_status	Reports if vm is running (0) or not (1)	
host_alive_status	See host_alive_status Codes below	
io.read_ops_sec	Disk I/O read operations per second	'device' (ie, 'hdd')
io.read_ops	Disk I/O read operations val	'device' (ie, 'hdd')
io.read_bytes	Disk I/O read bytes val	'device' (ie, 'hdd')
io.read_bytes_sec	Disk I/O read bytes per second	'device' (ie, 'hdd')
io.read_bytes_total	Total Disk I/O read bytes across all devices	
io.read_bytes_total_sec	Total Disk I/O read bytes per second across devices	
io.read_ops_total	Total Disk I/O read operations across all devices	
io.read_ops_total_sec	Total Disk I/O read operations across all devices per sec	
io.write_ops_sec	Disk I/O write operations per second	'device' (ie, 'hdd')
io.write_ops	Disk I/O write operations val	'device' (ie, 'hdd')
io.write_bytes	Disk I/O write bytes val	'device' (ie, 'hdd')
io.write_bytes_sec	Disk I/O write bytes per second	'device' (ie, 'hdd')
io.errors_sec	Disk I/O errors per second	'device' (ie, 'hdd')
io.write_bytes_total	Total Disk I/O write bytes across all devices	
io.write_bytes_total_sec	Total Disk I/O Write bytes per second across devices	
io.write_ops_total	Total Disk I/O write operations across all devices	
io.write_ops_total_sec	Total Disk I/O write operations across all devices per sec	
net.in_packets_sec	Network received packets per second	'device' (ie, 'vnet0')
net.out_packets_sec	Network transmitted packets per second	'device' (ie, 'vnet0')
net.in_bytes_sec	Network received bytes per second	'device' (ie, 'vnet0')
net.out_bytes_sec	Network transmitted bytes per second	'device' (ie, 'vnet0')
net.in_dropped_sec	Network received packets dropped per second	'device' (ie, 'vnet0')
net.out_dropped_sec	Network transmitted packets dropped per second	'device' (ie, 'vnet0')
net.in_errors_sec	Network received packets with errors per second	'device' (ie, 'vnet0')
net.out_errors_sec	Network transmitted packets with errors per second	'device' (ie, 'vnet0')
net.in_packets	Network received total packets	'device' (ie, 'vnet0')
net.out_packets	Network transmitted total packets	'device' (ie, 'vnet0')
net.in_bytes	Network received total bytes	'device' (ie, 'vnet0')
net.out_bytes	Network transmitted total bytes	'device' (ie, 'vnet0')
mem.free_gb	Free memory in Gbytes	
mem.total_gb	Total memory in Gbytes	
mem.used_gb	Used memory in Gbytes	
mem.free_perc	Percent of memory free	
mem.swap_used_gb	Used swap space in Gbytes	
ping_status	0 for ping success, 1 for ping failure	
cpu.time_ns	Cumulative CPU time (in ns)	
mem.resident_gb	Total memory used on host, an Operations-only metric	

Figure 9: Libvirt metrics

In the list is also present a metric called host_alive_status, that assumes different values depending on the host status.



Code	Description	value_meta 'detail'
-1	No state	VM has no state
0	Running / OK	None
1	Idle / blocked	VM is blocked
2	Paused	VM is paused
3	Shutting down	VM is shutting down
4	Shut off	VM has been shut off
4	Nova suspend	VM has been suspended
5	Crashed	VM has crashed
6	Power management suspend (S3 state)	VM is in power management (s3) suspend

Figure 10: Libvirt host_alive metric

Requirements

As said at the beginning of the subsection there are some requirements to be able to use Libvirt plugin.

- Neutron L3 agent in DVR mode (legacy mode is supported on single-node installations, such as devstack).
- Neutron L2 plugin with a tenant network type of vlan or vxlan (other types may be supported, but have not been tested).
- The python-neutronclient library and its dependencies installed and available to the Monasca Agent.

Detection

The monasca-setup detection plugin for libvirt performs some tests and tasks before configuring ping checks. It has the ability, for example, to determine the name of the user under which monasca-agent processes run (eg, mon-agent) and the availability of the python-neutronclient library.

If any of the requirements fail, a WARN-level message is output, describing the problem. The libvirt plugin will continue to function without these requirements, but ping checks will be disabled.

In addition to per-instance metrics, the Libvirt plugin will publish aggregate metrics across all instances. The list is available to consult on the github page /openstack/monasca-agent/../libvirt.md.

5.4 Alarms

Alarms in Monasca are identified by a severity level (LOW, MEDIUM, HIGH, CRITICAL) and an expression, composed by a mathematical function, a metric, a comparator and a threshold.

- The mathematical function value to choose from are: max, min, avg, count, sum, last.
- The metric can be chosen from the Libvirt metrics library or can be defined manually through the Monasca CLI.
- The comparator is simply a mathematical sign with that compares the value of the mathematical function applied to the metric and the threshold define in the following step.
- The threshold value can assume the values: -1, 0, or other positive values.

5.5 Notifications

Monasca already offers integration with email, PagerDuty, WebHook, Jira, Slack and HipChat. The notification engine is the Monasca component that consumes alarm state transition messages from the message queue and sends notifications for alarms, if required. Through either the Horizon Dashboard or using the Monasca CLI, notifications can be created. Depending on the choice regarding the notification method, further configuration may be available.

5.6 Grafana plugin

The Monasca API Devstack plugin allows to access Grafana in a very simple way, without the need to configure ports or authentication.

The Grafana login panel can be accessed from the alarms summary or from the Monitoring overview dashboard, and use as credentials the ones configured in the *agent.yaml* file, (default mini-mon:password).

Once opened the dashboard, many datasources are available: for example Monasca Monitoring retrieves all the dashboard of the components of the whole module, while Monasca API retrieves the dashboard for some of the standard metrics, CPU, memory, IO Count.

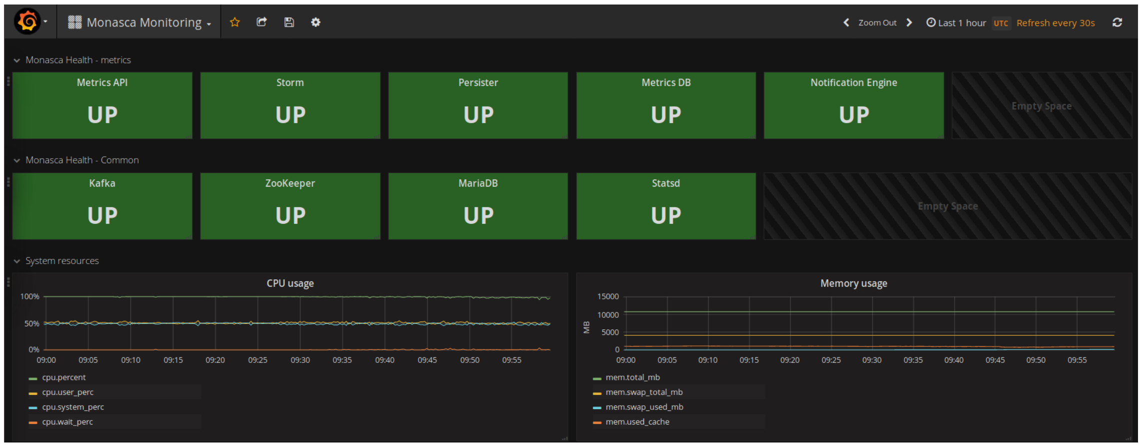


Figure 11: Monasca Monitoring dashboard

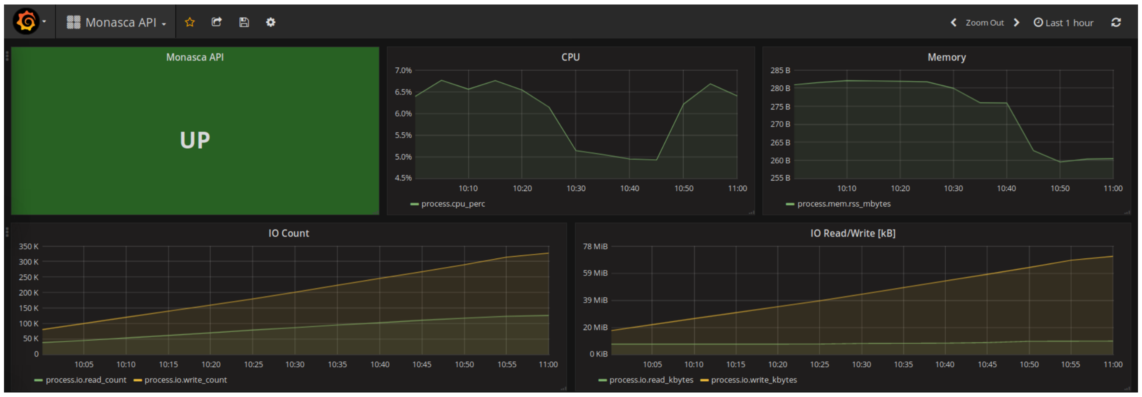


Figure 12: Monasca API dashboard

5.7 Monasca vs Ceilometer

Ceilometer is a suitable and easy to configure tool to manage the virtual layer in a private OpenStack cloud infrastructure, but when fine grained monitoring is required its Collector process does not have an efficient use of RAM in the host system.

Unlike Ceilometer, Monasca is easier to configure for managing the physical layer in an OpenStack cloud infrastructure, since its Persister process has a better use of RAM in the host system.

Although Ceilometer and Monasca are both widespread as Openstack monitoring tools, there are few comparison studies between the two.

On one hand, Ceilometer allows easy configuration for the virtual infrastructure, but is not so easy regarding the physical one. To obtain the hardware metrics if has to run a central agent on one node to poll the SNMP servers running on compute nodes, meaning if the agent goes down, the hardware meters of all compute nodes get lost.

On the other hand, Monasca allows a more manageable monitoring of the physical infrastructure, once the plugins are configured to enable the different metrics provided. The libvirt.yaml plugin allows monitoring of the virtual infrastructe and gathers different metrics per-instrance.

5.8 Ceilosca

Ceilosca is a combination of Ceilometer and Monasca. In Ceilosca data collected using the Ceilometer Agents (Notification Agent, Central/Compute Agent) is sent to the Monasca API and retrieved back using the Ceilometer v2 API for backwards compatibility.

The Telemetry project has deprecated the Ceilometer v2 API starting in Newton release (2016) and the API has been removed in the Queens release (2018) of Ceilometer. Ceilosca continues to support the Ceilometer v2 API through Pike (2017), though users are encouraged to switch to using the Monasca API going forward. Data gathered through the Ceilometer agents will still continue to be published and stored in Monasca.

6 Experimental Results

6.1 Prometheus Deployment

The approach to Prometheus, even for first-timers, is quite friendly.
First, download the release and extract it. For the deployment two VMs with Ubuntu 18.04 (Bionic) were used, one for the server, one as a target to monitor.

Once the download is complete, move into the directory and search for the Prometheus configuration file, called *prometheus.yml*. For the basic Prometheus configuration the file should look like this:

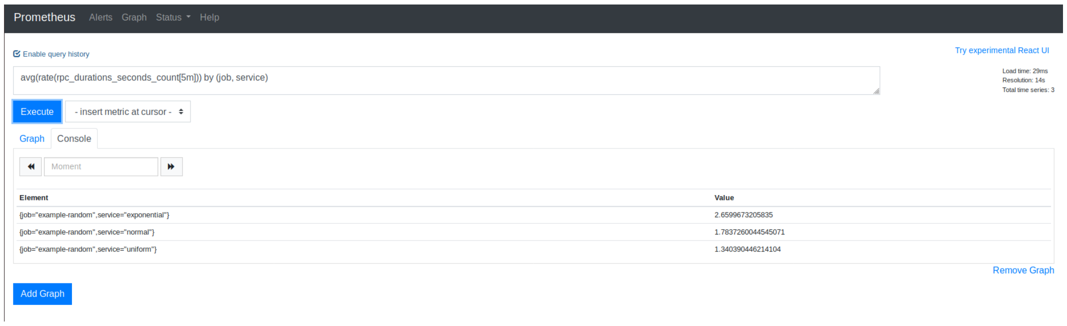
```
global:
  scrape_interval:     15s # By default, scrape targets every 15 seconds.

  # Attach these labels to any time series or alerts when communicating with
  # external systems (federation, remote storage, Alertmanager).
  external_labels:
    monitor: 'codelab-monitor'

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this
    config.
  - job_name: 'prometheus'

    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:9090']
```

It can be noticed that there are two sections: one is called `global` and specifies the default settings, the `scrape_config` section specifies instead a set of targets and parameters describing how to scrape them.
In the general case, as this one, one scrape configuration specifies a single job. In advanced configurations, this may change.

Targets may be statically configured via the `static_configs` parameter, as in the example, or dynamically discovered using one of the supported service-discovery mechanisms. There are many other configuration settings available, extensively documented on the official Prometheus page.

After defining the configuration file, the user has to run it using the `./prometheus --config.file=prometheus.yml`, the Prometheus server should be up right after.

To create metrics and start the monitoring, the user has to navigate to http:localhost:9090, using the target address (in this case ”localhost” is used because the target is on the same system on which Prometheus is installed), as specified in the configuration file. Once there, an expression console will be available: just choose one of the metrics from the drop-down menu and the answer will be promptly provided, as shown in the picture.



```
avg(rate(rpc_durations_seconds_count[5m])) by (job, service)
```

Load time: 29ms
Resolution: 14s
Total time series: 3

Element	Value
{job="example-random",service="exponential"}	2.6599673205835
{job="example-random",service="normal"}	1.7837260044545071
{job="example-random",service="uniform"}	1.340390446214104

Figure 13: Prometheus expression console

One of the most important component of Prometheus is the Alert Manager, and to it has to be manually configured by first downloading it from git and then run it in a similar way to the Prometheus Server.

Following the commands to install the Alert Manager.

```
$ GO15VENDOREXPERIMENT=1 go get github.com/prometheus/alertmanager/cmd/...
$ cd $GOPATH/src/github.com/prometheus/alertmanager
$ make build
$ alertmanager --config.file=<your_file>
```

To run the Alert Manager and use alarms it is first mandatory to configure the Manager itself using the `alertmanager.yml` file, as it was done with the Prometheus configuration file. An example is available on the official website and will not be replicated here due to its length. What is relevant to know however, is that through it, the user is able to define how the Manager can send alerts, using mail or services like Slack, a cloud-based instant messaging platform where chat rooms are structured in channels that are identified by topics of interest.

For the Alert Manager to function properly, the Prometheus Server has to be made aware that it has to connect to the Manager itself. First, add the following lines to the basic configuration settings:

```
rule_files:
  - alert.rules.ymlalerting:
  alertmanagers:
  - static_configs:
    - targets:
      - 'alertmanagerIP:9093'
```

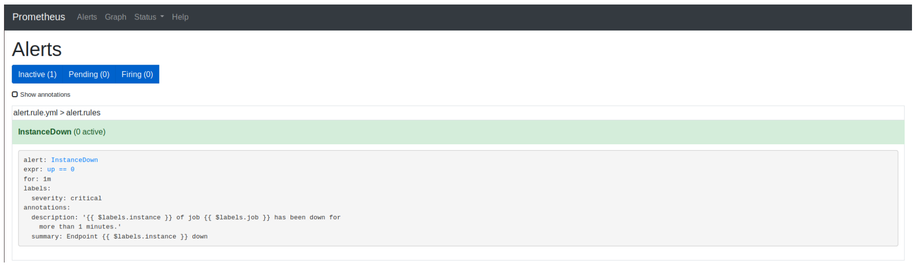
This way a new alerting rule will be defined. Next, how the new `alert.rules.yml` file will look like:

```
alert rule file
groups:
- name: alert.rules
  rules:
  - alert: InstanceDown
    expr: up == 0
    for: 1m
    labels:
      severity: "critical"
    annotations:
      summary: "Endpoint {{ $labels.instance }} down"
      description: "{{ $labels.instance }} of job {{ $labels.job }} has been down for more
    than 1 minutes."
```

6.2 Prometheus Monitoring Test

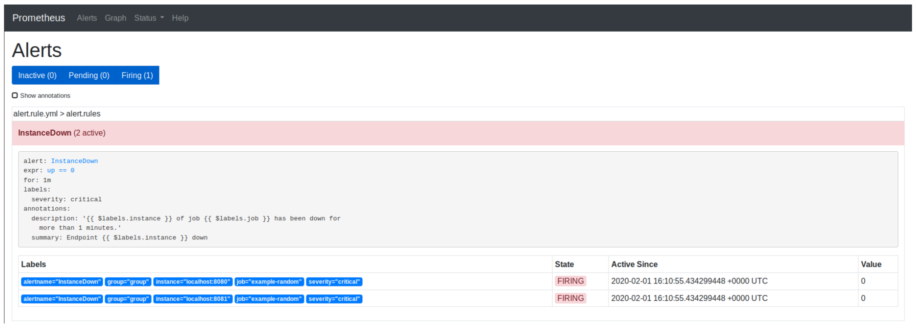
For the testing a simple alert template was used: if one of the targets instances is down for more than one minute, an alarm will be issued.
Now the Alert Manager can be finally started. As the Prometheus Server (started from another terminal window), the Alert Manager interface will be available from the `localhost:<port>` address specified, in this case 9093.

Since no notification method was configured, the only way to see the alert being fired is by going to the Prometheus Server address, go to the `Alerts` section, and observe how alarms behave. In the following images, the Alerts section from the Server is shown.



```
alert: InstanceDown
expr: up == 0
for: 1m
labels:
  severity: critical
annotations:
  description: '{{ $labels.instance }} of job {{ $labels.job }} has been down for
    more than 1 minutes.'
  summary: Endpoint {{ $labels.instance }} down
```

Figure 14: Alerts inactive



```
alert: InstanceDown
expr: up == 0
for: 1m
labels:
  severity: critical
annotations:
  description: '{{ $labels.instance }} of job {{ $labels.job }} has been down for
    more than 1 minutes.'
  summary: Endpoint {{ $labels.instance }} down
```

Labels	State	Active Since	Value
alertname="InstanceDown" group="group" instance="localhost:8080" job="example-random" severity="critical"	FIRING	2020-02-01 16:10:55.434299448 +0000 UTC	0
alertname="InstanceDown" group="group" instance="localhost:8081" job="example-random" severity="critical"	FIRING	2020-02-01 16:10:55.434299448 +0000 UTC	0

Figure 15: Alerts being fired

6.3 Linux Host Monitoring Example

An interesting case of study was using Prometheus to monitor a Linux Host. As previously mentioned, Prometheus may scape metrics from external hosts using the Node Exporter.

In this context another intermediary service was used, the Pushgateway: it allows you to push time series from short-lived service-level batch jobs to an intermediary job which Prometheus can scrape.

This tests wants to simulate the tracking of performance metrics that happens on servers, in case unresponsive instances might block the running of remote commands such as top or htop or in case a simple bottleneck occurs and there is no way to identify it in a simple and quick way.

This solution is also scalable because it may monitor more than one host at the same time.

It is important to understand however, that the Pushgateway should be only used in certaint cases, as the Prometheus documentation suggests: usually, the only valid use case for the Pushgateway is for capturing the outcome of a service-level batch job.

The Pushgateway has to be run on the target as it was done with the Node Exporter: download it, extract the files and move into the directory in order to run the `./pushgateway` command. In the output the port on which the service listens will be available, in this case 9091.

To make full use of it, however, a script to send the metrics gathered to Prometheus is needed. Pushgateway, like Prometheus, works with key value pairs, the key describing the metric monitored.

For the case under examination, what is of interest is the cpu usage and the memory usage and the processes that mostly weight on the performances.
In one case the value of the cpu usage at a certain instant of time is needed, so the metric will be a simple key + value pair, like:

```
cpu_usage <value>
```

In another case, it may be useful to know the cpu usage value associated with a certain process, so it will be in the form:

```
cpu_usage {process=<pid>, <value>}
```

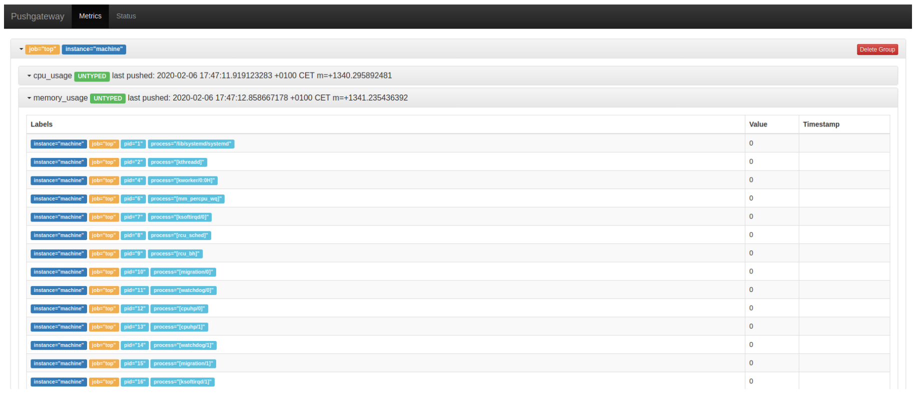
A script was created to execute the `ps aux` command and gather metrics depending on cpu usage, parse the result, transform it and send it to the Pushgateway via the syntax described before. Similarly another script was created for memory usage.

```bash
#!/bin/bash
z=$(ps aux)
while read -r z
do
   var=$var$(awk '{print "cpu_usage{process=\""$11"\", pid=\""$2"\"}", $3z}');
done <<< "$z"
curl -X POST -H  "Content-Type: text/plain" --data "$var
" http://192.168.1.161:9091/metrics/job/top/instance/machine
```

After this, run it from the terminal using the `while sleep 1; do .cpu_usage.sh; done;` command. It will gather the metrics every one second. Run it the same way for memory_usage.

Now everything is ready to start the monitoring: edit the `prometheus.yml` in the "targets" section and add the Pushgateway address, then run the command to start Prometheus.

From the address of the Pushgateway, metrics are already available: there are two sections, one each for the scripts that are running, followed by the data acquired in the format previously described.



job="top" instance="machine"

cpu_usage UNTYPED last pushed: 2020-02-06 17:47:11.919123283 +0100 CET m=+1340.295892481

memory_usage UNTYPED last pushed: 2020-02-06 17:47:12.858667178 +0100 CET m=+1341.235436392

Labels	Value	Timestamp
instance="machine" job="top" pid="1" process="/lib/systemd/systemd"	0	
instance="machine" job="top" pid="2" process="[kthreadd]"	0	
instance="machine" job="top" pid="4" process="[kworker/0:0H]"	0	
instance="machine" job="top" pid="6" process="[mm_percpu_wq]"	0	
instance="machine" job="top" pid="7" process="[ksoftirqd/0]"	0	
instance="machine" job="top" pid="8" process="[rcu_sched]"	0	
instance="machine" job="top" pid="9" process="[rcu_bh]"	0	
instance="machine" job="top" pid="10" process="[migration/0]"	0	
instance="machine" job="top" pid="11" process="[watchdog/0]"	0	
instance="machine" job="top" pid="12" process="[cpuhp/0]"	0	
instance="machine" job="top" pid="13" process="[cpuhp/1]"	0	
instance="machine" job="top" pid="14" process="[watchdog/1]"	0	
instance="machine" job="top" pid="15" process="[migration/1]"	0	
instance="machine" job="top" pid="16" process="[ksoftirqd/1]"	0	

Figure 16: Pushgateway - metrics scraped

Grafana was used to visualize the metrics in a more elaborate way: the user is able to custom almost everything using a variety of settings, graphs and labels, creating the ultimate dashboard.
PromQl, the Prometheus query language will be used for queries, in order to aggregate data using functions such as the sum, the average and the standard deviation.

The monitoring will have five items:

1 - Cpu Usage
FUNCTION - sum(cpu_usage)
TYPE OF GRAPH - Gauge
VALUES - Instant

2 - Memory Usage
FUNCTION - sum(memory_usage)
TYPE OF GRAPH - Gauge
VALUES - Instant

3 - Average Cpu Usage
FUNCTION - avg(cpu_usage)
TYPE OF GRAPH - Gauge
VALUES - Instant

4 - Most cpu-consuming processes
Function - topk(10, cpu_usage)
Type of graph - Line Graph
Values - Historical

5 - Most cpu-consuming processes
Function - topk(10, memory_usage)
Type of graph - Line Graph
Values - Historical

In the first three cases, instant values will be needed, so the Grafana panel will need to be correctly settled.
In the last two cases, instead a Line graph is used: this graph can be particularly useful when there has been some outage in the past and it is relevant to investigate which processes were active at the time or if at a certain process died but a view of its behaviour right before it happened could be helpful. Following, the complete dashboard.

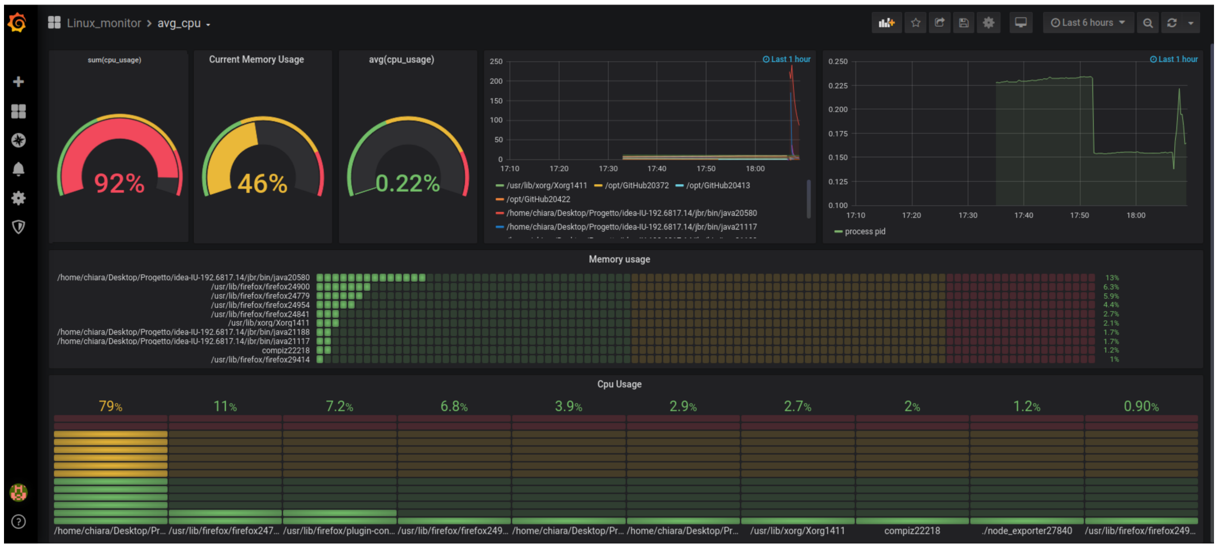
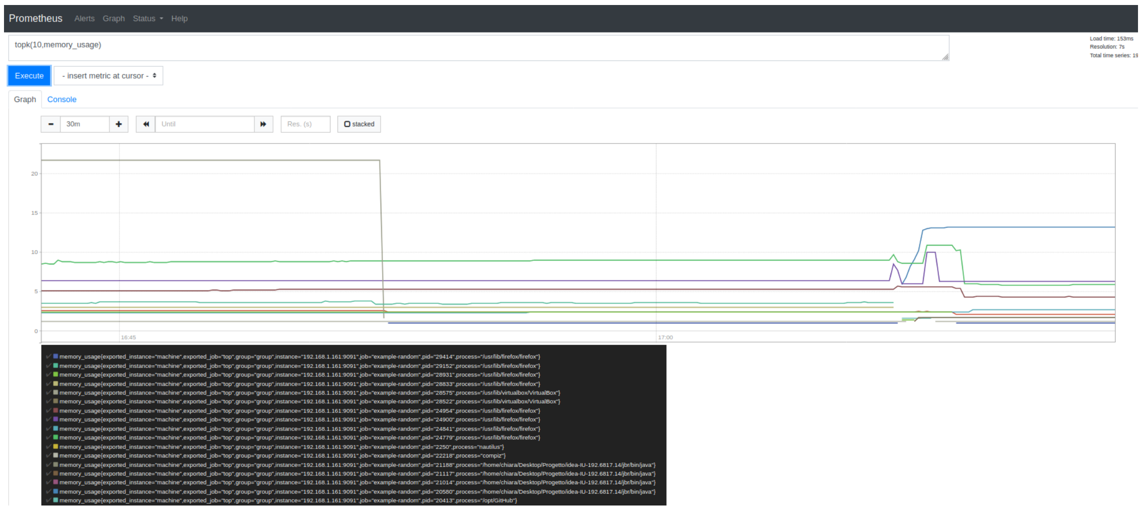

Linux_monitor > avg_cpu
Last 6 hours
sum(cpu_usage)
92%
Current Memory Usage
46%
avg(cpu_usage)
0.22%
Last 1 hour
/usr/lib/xorg/Xorg1411
/opt/GitHub20372
/opt/GitHub20413
/opt/GitHub20422
/home/chiara/Desktop/Progetto/idea-IU-192.6817.14/jbr/bin/java20580
/home/chiara/Desktop/Progetto/idea-IU-192.6817.14/jbr/bin/java21117
process pid
Memory usage
/home/chiara/Desktop/Progetto/idea-IU-192.6817.14/jbr/bin/java20580
/usr/lib/firefox/firefox24900
/usr/lib/firefox/firefox24779
/usr/lib/firefox/firefox24954
/usr/lib/firefox/firefox24841
/usr/lib/xorg/Xorg1411
/home/chiara/Desktop/Progetto/idea-IU-192.6817.14/jbr/bin/java21188
/home/chiara/Desktop/Progetto/idea-IU-192.6817.14/jbr/bin/java21117
compiz22218
/usr/lib/firefox/firefox29414
Cpu Usage
79%
11%
7.2%
6.8%
3.9%
2.9%
2.7%
2%
1.2%
0.90%
/home/chiara/Desktop/Pr...
/usr/lib/firefox/firefox247...
/usr/lib/firefox/plugin-con...
/usr/lib/firefox/firefox249...
/home/chiara/Desktop/Pr...
/home/chiara/Desktop/Pr...
/usr/lib/xorg/Xorg1411
compiz22218
./node_exporter27840
/usr/lib/firefox/firefox249...

Figure 17: Grafana - Linux Host Monitoring Dashboard

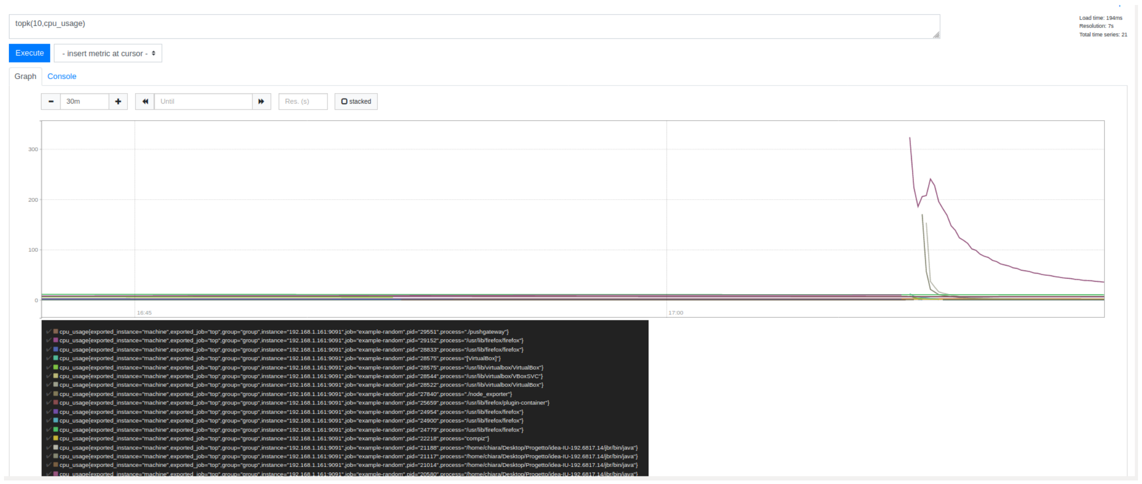
In comparison, Prometheus Interface is definitely a less reader friendly choice to visualize these metrics. The lines are sometimes interrupted depending on the scraping interval, and no additional customization is available. The queries topk(10, memory_usage) and topk(10, cpu_usage) were replicated.



```
topk(10,memory_usage)
```

```
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="29414",process="/usr/lib/firefox/firefox"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="29152",process="/usr/lib/firefox/firefox"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="28931",process="/usr/lib/firefox/firefox"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="28833",process="/usr/lib/firefox/firefox"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="28575",process="/usr/lib/virtualbox/VirtualBox"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="28522",process="/usr/lib/virtualbox/VirtualBox"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="24954",process="/usr/lib/firefox/firefox"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="24900",process="/usr/lib/firefox/firefox"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="24841",process="/usr/lib/firefox/firefox"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="24779",process="/usr/lib/firefox/firefox"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="2250",process="nautilus"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="22218",process="compiz"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="21188",process="/home/chiara/Desktop/Progetto/idea-IU-192.6817.14/jbr/bin/java"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="21117",process="/home/chiara/Desktop/Progetto/idea-IU-192.6817.14/jbr/bin/java"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="21014",process="/home/chiara/Desktop/Progetto/idea-IU-192.6817.14/jbr/bin/java"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="20580",process="/home/chiara/Desktop/Progetto/idea-IU-192.6817.14/jbr/bin/java"}
memory_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="20413",process="/opt/GitHub"}
```

Figure 18: Prometheus - top k processes for memory usage



```
topk(10,cpu_usage)
```

```
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="29551",process="./pushgateway"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="29152",process="/usr/lib/firefox/firefox"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="28833",process="/usr/lib/firefox/firefox"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="28575",process="[VirtualBox]"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="28575",process="/usr/lib/virtualbox/VirtualBox"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="28544",process="/usr/lib/virtualbox/VBoxSVC"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="28522",process="/usr/lib/virtualbox/VirtualBox"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="27840",process="./node_exporter"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="25659",process="/usr/lib/firefox/plugin-container"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="24954",process="/usr/lib/firefox/firefox"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="24900",process="/usr/lib/firefox/firefox"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="24779",process="/usr/lib/firefox/firefox"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="22218",process="compiz"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="21188",process="/home/chiara/Desktop/Progetto/idea-IU-192.6817.14/jbr/bin/java"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="21117",process="/home/chiara/Desktop/Progetto/idea-IU-192.6817.14/jbr/bin/java"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="21014",process="/home/chiara/Desktop/Progetto/idea-IU-192.6817.14/jbr/bin/java"}
cpu_usage{exported_instance="machine",exported_job="top",group="group",instance="192.168.1.161:9091",job="example-random",pid="20580",process="/home/chiara/Desktop/Progetto/idea-IU-192.6817.14/jbr/bin/java"}
```

Figure 19: Prometheus - top k processes for cpu usage

6.4 Monasca Deployment

Openstack release used: Train

The installation was achieved through the devstack script, which supports also the recently developed Monasca API plugin, that allows a seamless configuration of the Monasca API core and all its components without the need for further configuration.

Note that the plugin currently only works on Ubuntu 16.04 and 18.04 (Bionic), more Linux Distributions will probably be added in the future.
Running the Monasca DevStack plugin requires a machine with at least 10GB of RAM. The virtual machine used for the project had 12GB of RAM, 2 processors, 40 GB of memory and two network interfaces, the first private, the second one provides a

network connection between the virtual machine and the host system. Different vm configurations are of course allowed, and some have been tried for testing alongside this one.

For example, install the Openstack infrastructure on Ubuntu Server and use another vm on the same private network as a display to connect to the Horizon Dashboard. To install and run Devstack the only necessary step is to clone the DevStack repo from git, although it is advised to update and upgrade the Ubuntu system before doing so.

Once the repository has been cloned, move into it and copy the configuration file template called *local.conf* into the devstack directory. After that, modify it in order to install an all-in-one Openstack configuration and the Monasca Api components. The final *local.conf* file should look like this:

```
[[local|localrc]]

DATABASE_PASSWORD=secretdatabase
RABBIT_PASSWORD=secretrabbit
ADMIN_PASSWORD=secretadmin
SERVICE_PASSWORD=secretservice
LOGFILE=$DEST/logs/stack.sh.log
LOGDIR=$DEST/logs
LOG_COLOR=False

# MONASCA_PERSISTER_IMPLEMENTATION_LANG=${MONASCA_PERSISTER_IMPLEMENTATION_LANG:-java}
MONASCA_PERSISTER_IMPLEMENTATION_LANG=${MONASCA_PERSISTER_IMPLEMENTATION_LANG:-python}

MONASCA_METRICS_DB=${MONASCA_METRICS_DB:-influxdb}
# MONASCA_METRICS_DB=${MONASCA_METRICS_DB:-cassandra}
# This line will enable all of Monasca.
enable_plugin monasca-api https://opendev.org/openstack/monasca-api
```

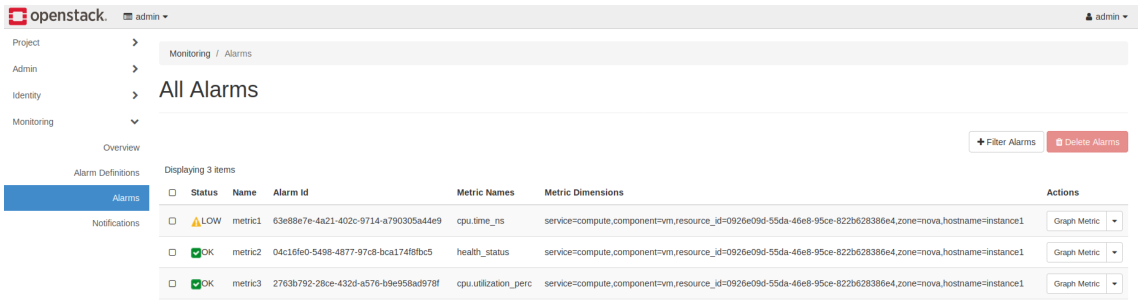
If you want to run Monasca with the bare mininum of OpenStack components you can add the following two lines. In the second one you can specify the services you want to start, in this case only Rabbit, Mysql and Keystone.

```
disable_all_services
enable_service rabbit mysql key
```

Then simply run `./stack.sh` from the root of the devstack directory. The process may take even an hour, depending on the host machine computational power. At the end of the process the Horizon Dashboard will be available at the HOST IP address. All the Monasca-related operations available from the dashboard are also available from the CLI.

6.5 Monasca Monitoring Test

Once Openstack is up and running, the user will be able to access the Monitoring section, and add alarms, notifications and use Grafana to graph the metrics. To define alarms, simply go to the *Monitoring* section and choose *Alarms Definition*. While defining the alarm the corresponding metric defined in JSON format will be shown in the window, as shown in the picture. After defining alarms it will be available

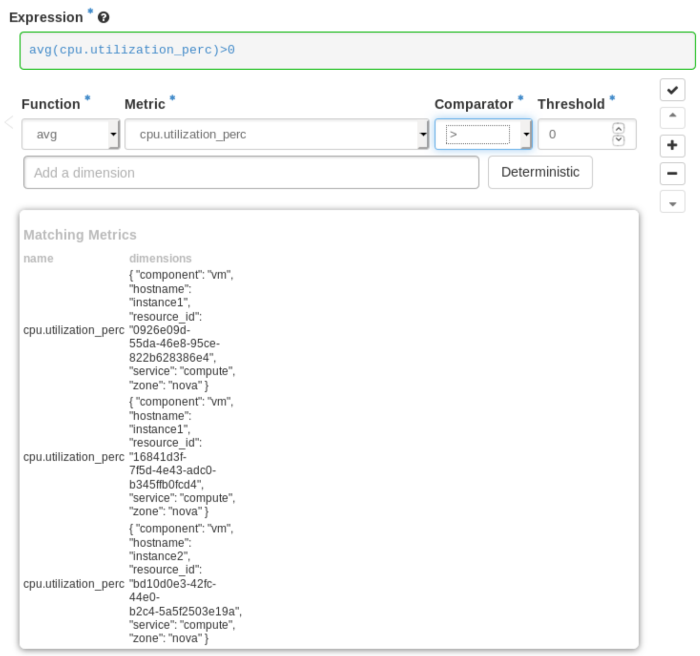


All Alarms

Displaying 3 items

	Status	Name	Alarm Id	Metric Names	Metric Dimensions	Actions
☐	⚠LOW	metric1	63e88e7e-4a21-402c-9714-a790305a44e9	cpu.time_ns	service=compute,component=vm,resource_id=0926e09d-55da-46e8-95ce-822b628386e4,zone=nova,hostname=instance1	Graph Metric
☐	✓OK	metric2	04c16fe0-5498-4877-97c8-bca174f8fbc5	health_status	service=compute,component=vm,resource_id=0926e09d-55da-46e8-95ce-822b628386e4,zone=nova,hostname=instance1	Graph Metric
☐	✓OK	metric3	2763b792-28ce-432d-a576-b9e958ad978f	cpu.utilization_perc	service=compute,component=vm,resource_id=0926e09d-55da-46e8-95ce-822b628386e4,zone=nova,hostname=instance1	Graph Metric

Figure 20: Listing of alarms



Expression *

```
avg(cpu.utilization_perc)>0
```

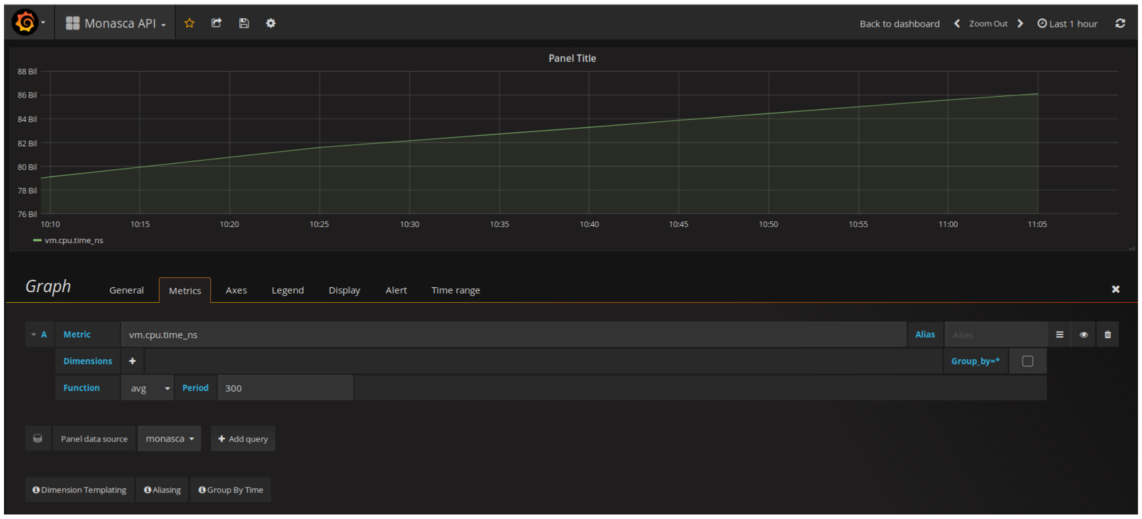
Function *	Metric *	Comparator *	Threshold *
avg	cpu.utilization_perc	>	0

Matching Metrics

name	dimensions
cpu.utilization_perc	{ "component": "vm", "hostname": "instance1", "resource_id": "0926e09d-55da-46e8-95ce-822b628386e4", "service": "compute", "zone": "nova" }
cpu.utilization_perc	{ "component": "vm", "hostname": "instance1", "resource_id": "16841d3f-7f5d-4e43-adc0-b345ffb0fcd4", "service": "compute", "zone": "nova" }
cpu.utilization_perc	{ "component": "vm", "hostname": "instance2", "resource_id": "bd10d0e3-42fc-44e0-b2c4-5a5f2503e19a", "service": "compute", "zone": "nova" }

Figure 21: Definition of one alarm

in the *Alarms* section, in which you can check the metric description, the value (OK, ALARM, UNDETERMINED) and all the related information. To graph a metric, Grafana can be accessed from the alarms section by clickin on "Graph metric". Here the metric will be visualized, and many customization options are available to users.



```
A  Metric      vm.cpu.time_ns
   Dimensions  +                                   Group_by=*
   Function    avg    Period  300
Panel data source  monasca
```

Figure 22: Metrics graphed on Grafana

7 Comparison between tools

Lastly, a table to illustrate the major differences between Monasca and Prometheus, on which this report is mainly focused.

Tool	**Installation time**	**Additional services**	**User Interface**	**Multi-tenancy**
Monasca	Long	Already installed (if specified in the configuration file)	No (uses Grafana)	Yes
Prometheus	Short	To be manually installed	Yes (can also use Grafana)	No

After deploying both tools, it is of course noticeable how they differ in terms of installation: Prometheus Server is available in very few and quick steps, while Monasca takes a very long time to deploy. The fact that Monasca takes longer to install is surely due to the fact that all the services required in the configuration file have to be installed and properly configured, while Prometheus does not install or configure any additional service.

Indeed, if additional settings are required to Prometheus, they have to be configured manually, even a service such as the Alert Manager. On the other hand Monasca (referring to the Devstack Plugin used in the case under examination) is already fully operative and many other services are readily available.

To take into account, however, that Prometheus is easily extended by using exporters, and interacts with many third-party software for a wide range of monitoring activities.

Another difference is the fact that Prometheus has its own User Interface, while Monasca is supported by Grafana, but as it was already mentioned, this is easily overcome since the configuration between the two is already complete. However, having being created for slightly different purposes it makes sense that they differ on several levels.

One of the main drawbacks of Prometheus, according to the community, is that Prometheus offers by default a static configuration, not a dynamic one. If needed, a service discovery will need to be implemented. Zibbix instead, supports automatic discovery of targets. Another deficiency of Prometheus is the fact that multi-tenancy is not supported, wheras it is, for example, in Monasca.

Multi-tenancy in Openstack means that multiple users can reside on the platform, with data isolation that ensure's the privacy of each other tenant's data. Installed with default configuration Openstack has three users, admin, alt-demo, and demo: each one has its own area of competence, its own projects, and its own monitoring environment.

Both support however using chained metrics and expression with "and" or "or" to create more complex metrics.

In comparison to Nagios, on the other hand, Prometheus offers more solutions to manage alerts, since Nagios is primarily about alerting based on the exit codes of scripts and there are no grouping or routing, even if silences are available. In terms of storage, Nagios has no storage per-se, even if there are plugins which can store data such as for visualisation; Prometheus includes a local on-disk time series database, but also optionally integrates with remote storage systems. In both cases the servers are standalone and all configuration is via file.

Nagios is indeed, suitable for basic monitoring of small or static systems where blackbox probing is sufficient. Prometheus is instead considerd a whitebox monitoring tool.

8 Conclusions

This report had the main objective of describing some of the most popular solutions for monitoring that are available, with a focus on Monasca and Prometheus.

Prometheus was born as a monitoring and alerting toolkit for containers and microservices, while Monasca monitoring-as-a-service solution that integrates with OpenStack.

The main reason that Monasca and Prometheus differ is of course due to their scope. Monasca has the objective to monitor instances on Openstack platform and has to adapt to an environment where multiple users can reside and need an highly scalable, performant, fault-tolerant tool to gather information and data. Prometheus on the other hand, has become the mainstream, open source monitoring tool of choice for those that lean heavily on containers and microservices.

Nonetheless, it is important to take notice of how these tools are popular in the community, and the community itself is active in improving them and offer quality solutions as monitoring tools.

In terms of tools used for this report, the Devstack plugin was fundamental for the study of the Monasca module and it allowed a relatively easy configuration and deployment. It is important to notice that Devstack is not production ready but development only, since it allows to configure Openstack in a quick but at the same time "fragile" way. If necessary, the faulty vms can be discarded and new ones can be deployed right away. Much more complex and elaborated considerations are to be taken in enterprise environment. This project has, among the objectives, the one to analyse Monasca and its components.

In conclusion, monitoring any IT system is not a simple task and in the highly dynamic world of the cloud, it is even more complex. In addition to the components and tools mentioned during this report, there are third party open source tools as well as multiple modern commercial ones.

In order to make a real and practical move forward, it is compulsory to first define what is needed to monitor and where to start, be it with compute consumption and usage, detailed network performance or maybe user access for security reasons.

References

[1] Mario A. Gomez-Rodriguez, Victor J. Sosa-Sosa and Jose L. Gonzalez-Compean, Assessment of Private Cloud Infrastructure Monitoring Tools

[2] Ceilometer Openstack Docs, https://docs.openstack.org/ceilometer/latest/

[3] Mitesh, Prometheus With AlertManager

[4] Monasca Devstack plugin, https://github.com/openstack/monasca-api/tree/master/devstack

[5] Monasca Openstack Docs, https://docs.openstack.org/monasca-api/latest/

[6] Monasca Openstack Wiki, https://wiki.openstack.org/wiki/Monasca

[7] Prometheus Documentation, https://prometheus.io/docs/introduction/overview/

[8] Prometheus AlertManager github, https://github.com/prometheus/alertmanager

And also many thanks to the #openstack-monasca community from the official IRC channel.