TNOVA

NETWORK FUNCTIONS AS-A-SERVICE
OVER VIRTUALISED INFRASTRUCTURES

GRANT AGREEMENT NO. 619520

Deliverable D4.42

Monitoring and Maintenance - Final

Editor G. Gardikis (SPH)

Contributors I. Koutras, G. Mavroudis, S. Costicoglou (SPH), G. Dimosthenous, D. Christofi (PTL), M. Di Girolamo (HPE), K. Karras (FINT), G. Xilouris, C. Sakkas, E. Trouva (NCSRD), M. Arnaboldi (ITALTEL), P. Harsh (ZHAW), E. Markakis (TEIC)

Version 1.0

Date March 31th, 2016

Distribution PUBLIC (PU)

Executive Summary

This deliverable is the final report of the work carried out in Task 4.4 (Monitoring and Maintenance). The task focuses on the implementation and integration of a monitoring framework, able to extract, process and communicate monitoring information from both physical and virtual nodes as well as VNFs at IVM level.

The first step is the consolidation of IVM requirements, as expressed in Deliverable D2.32, in order to derive the specific requirements for the monitoring framework. The latter include: monitoring of all NFVI domains (hypervisor/compute/storage/network) as well as VNF applications; processing and generation of events and alarms; communication of monitoring information as well as events/alarms to the Orchestrator in a scalable manner.

In parallel, a comprehensive survey of cloud and network monitoring tools is performed, in order to identify technologies which can be re-used for VIM monitoring. Special emphasis is put on frameworks which integrate smoothly with Openstack, in particular Openstack Telemetry/Ceilometer, Monasca, Gnocchi, Cyclops, Zabbix, Nagios as well as relevant OPNFV projects (Doctor and Prediction). It seems that most of the existing technological enablers for VIM monitoring, can only partially address all the aforementioned challenges in a lightweight and resource-efficient manner. Although most of them are indeed open and modular, they are already quite complicated and resource-demanding and therefore further expanding them to cover these needs would require considerable effort and would raise efficiency issues. We thus propose a "clean-slate" approach towards NFV monitoring at VIM level, exploiting only some basic enablers and adding only the required functionalities.

The T-NOVA VIM monitoring framework is introduced as a contribution towards this direction. The framework is built around the VIM Monitoring Manager (VIM MM), which is the key component devoted to monitoring at VIM level. The VIM MM exploits OpenStack and OpenDaylight APIs to retrieve a set of metrics for both physical and virtual nodes, which should be sufficient for most NFV handling requirements. However, in order to gain a more detailed insight on the VNF status and operation, a Monitoring Agent, based on the collectd framework, is also introduced in each VNF VM, collecting a large variety of metrics at frequent intervals.

The VIM MM consists of the following components:

- Openstack and OpenDaylight connectors, used to periodically poll the two platforms via their monitoring APIs.
- A VNF Application connector, which accepts data periodically dispatched by the VNF application. These metrics are specific to each VNF.
- A time-series database (InfluxDB) for data persistence.
- An alarming/anomaly detection engine which utilises statistical methods based on pre-defined but also dynamic thresholds in order to identify possible anomalies in the NFV service and to produce the corresponding alarms/events to be forwarded to the Orchestrator/VNFM.

- A Graphical User Interface (GUI), based on Grafana, which visualizes the stored metrics and presents them as live, time-series graphs.
- A Northbound API, which communicates selected metrics and events to the Orchestrator and, in turn, to the VNF Manager(s). The provided REST API allows metrics to be communicated in either push or pull mode.

The VIM monitoring framework is integrated, validated, evaluated and released as open-source in the frame of the project.

It is concluded that, with the proposed approach, the goal of delivering an effective, efficient and scalable monitoring solution for the T-NOVA IVM layer is achieved. The developed solution is able to expose to the Orchestrator and to the Marketplace enhanced awareness of the IVM status and resources, while at the same time keeping the communication and signalling overhead at minimum.

Table of Contents

1. INTRODUCTION

This deliverable is the final report of the work currently being carried out in Task 4.4 (Monitoring and Maintenance[1]). Task 4.4 focuses on the implementation and integration of a monitoring framework, able to extract, process and communicate monitoring information from both physical and virtual nodes as well as VNFs at IVM level. In other words, the operational scope of the monitoring framework being developed in Task 4.4 corresponds to the two lower layers of the T-NOVA architecture, namely the NFVI and VIM. The metrics[2] collected, along with alarms/events generated, are in turn communicated to the upper layers (Orchestrator and Marketplace), so that the latter have a comprehensive view of the status of the infrastructure resources as well as the network services running on them.

The present document is structured as follows:

- Chapter 2 overviews and consolidates the T-NOVA system and IVM requirements which directly or indirectly affect the monitoring framework.
- Chapter 3 presents a survey of the most prominent enabling technologies for NFV monitoring, as well as relevant Openstack, OpenDaylight and OPNFV projects and presents a justification for the technologies used.
- Chapter 4 presents the architecture and the functional blocks of the T-NOVA VIM monitoring framework.
- Chapter 5 presents the testing and validation of the framework against specific test cases.
- Finally, Chapter 6 concludes the document.

Compared to the interim version of this deliverable (D4.41), the present document contains the following major updates:

- Addition of Section 4.6.5 (Anomaly detection) to reflect the new work done in this field, as well as Sec. 5.3 (Assessment of anomaly detection methods)
- Addition of Annex II (VIM Monitoring Manager API Reference)
- Addition on Sec. 5.1.2 (Northbound API tests)
- Update on the mechanism of FPGA-based VNFs (Sec. 4.5)
- Significant update of VNF-specific metrics (Sec.4.4)
- Update on OPNFV Doctor and Prediction projects (Sec. 3.7)
- Update on the requirements fulfillment status (Sec. 5.4)

[1] It must be clarified that although some kind of maintenance actions were in the initial scope of T4.4, during the design and the specification phases of the project it was decided that all control operations are to be decided by the Orchestrator and carried out mostly via the VNFM. Thus, maintenance actions, as a reaction to monitored status data, are in scope of WP3.

[2] It must be also clarified that Task 4.4 focuses on the collection of dynamic metrics, i.e. metrics which change frequently in relation to resource usage. Static information reflecting the status and capabilities of infrastructure, e.g. number of installed compute nodes, processing resources per node etc. are assumed to be handled by Task 3.2 (Infrastructure Repository).

2. REQUIREMENTS OVERVIEW AND CONSOLIDATION

Deliverable D2.32 [D232] has defined and identified architectural concepts and requirements for the IVM (NFVI and VIM) layers. The technical requirements which drive the specification and development of the T-NOVA monitoring framework can be directly derived/inherited by the specific IVM requirements. Table 1 below identifies the IVM requirements that –either directly or indirectly- are associated to IVM monitoring, focusing on NFVI-PoP resources and describes how each of these are translated to a specific requirement for the monitoring framework.

Table 1. IVM requirements which affect the monitoring framework

IVM Req.ID	IVM Requirement Name	Requirement for the Monitoring Framework
VIM.1	Ability to handle heterogeneous physical resources	The MF must provide a vendor agnostic mechanism for physical resource monitoring.
VIM.2	Ability to provision virtual instances of the infrastructure resources	The MF must be able to report the status of virtualized resources as well as from physical resources in order to assist placement decisions
VIM.3	API Exposure	The MF must provide an interface to the Orchestrator for the communication of monitoring metrics.
VIM.6	Translation of references between logical and physical resource identifiers	The MF must re-use resource identifiers when linking metrics to resources.
VIM.8	Control and Monitoring	The MF must monitor in real time the physical network infrastructure as well as the vNets instantiated on top of it, providing measurements of the metrics relevant to service level assurance.
VIM.9	Scalability	The MF must keep up with dynamic increase of the number of resources to be monitored
VIM.18	Query API and Monitoring	The MF must provide an API for communicating metrics (in either push or pull mode)
VIM.21	Virtualised Infrastructure Metrics	The MF must collect performance and utilisation metrics from the virtualised resources in the NFVI.
C.7	Compute Domain Metrics	The MF must collect compute domain metrics.
C.12	Hardware accelerator metrics	The MF must collect hardware accelerator metrics
H.1	Compute Domain Metrics	The MF must collect compute metrics from the Hypervisor.

H.2	Network Domain Metrics	The MF must collect network domain metrics from the Hypervisor.
H.12	Alarm/Error Publishing	The MF must process and dispatch alarms.
N.5	Usage monitoring	The MF must collect metrics from physical and virtual networking devices.
N.8	SDN Management	The MF must leverage SDN monitoring capabilities.

By consolidating the aforementioned requirements, it becomes clear that the basic required functionalities of the IVM monitoring framework are as follows:

1. Collection of IT and networking metrics from virtual and physical devices of the NFVI. It should be noted that at the IVM level, metrics correspond only to physical and virtual nodes and are not associated to services since the VIM does not have knowledge of the end-to-end Network Service. Metrics are mapped to Network Services at Orchestrator level;
2. Processing and generation of events and alarms;
3. Communication of monitoring information and events/alarms to the Orchestrator in a scalable manner;

The following chapter overviews several technological frameworks for NFV monitoring which could be partially exploited towards fulfilling these requirements.

3. TECHNOLOGIES AND FRAMEWORKS FOR NFV MONITORING

This chapter presents a brief overview of the most relevant monitoring frameworks which can be applied to the NFV domain. This section mostly focuses on monitoring tools which provide a satisfactory degree of integration with OpenStack and can be extended for NFV monitoring; a more comprehensive survey of other, most generic IT and network/SDN monitoring tools can be found in Annex 2.

3.1. OpenStack Telemetry/Ceilometer

The goal of the Telemetry project within OpenStack [Telemetry], is to reliably collect measurements of the utilisation of physical and virtual resources, comprising deployed clouds, store such data for offline usage, and trigger actions on the occurrence of given events. It includes three different services (Aodh, Ceilometer and Gnocchi – see Sec. 3.3), providing the different stages of the data monitoring functional chain: Aodh delivers alarming functions, Ceilometer deals with data collection, Gnocchi provides a time-series database with resource indexing.

The actual data collection service in the Telemetry project is Ceilometer. Ceilometer is an OpenStack service which performs collection of data, normalizes and duly transforms them, making them available to other services (starting from the Telemetry ones). Ceilometer efficiently collects the metering data of virtual machines (VMs) and the computing hosts (Nova), the network, the Operating System images (Glance), the disk volumes (Cinder), the identities (Keystone), the object storage (Swift), the orchestration (Heat), the energy consumption (Kwapi) and also user-defined meters.

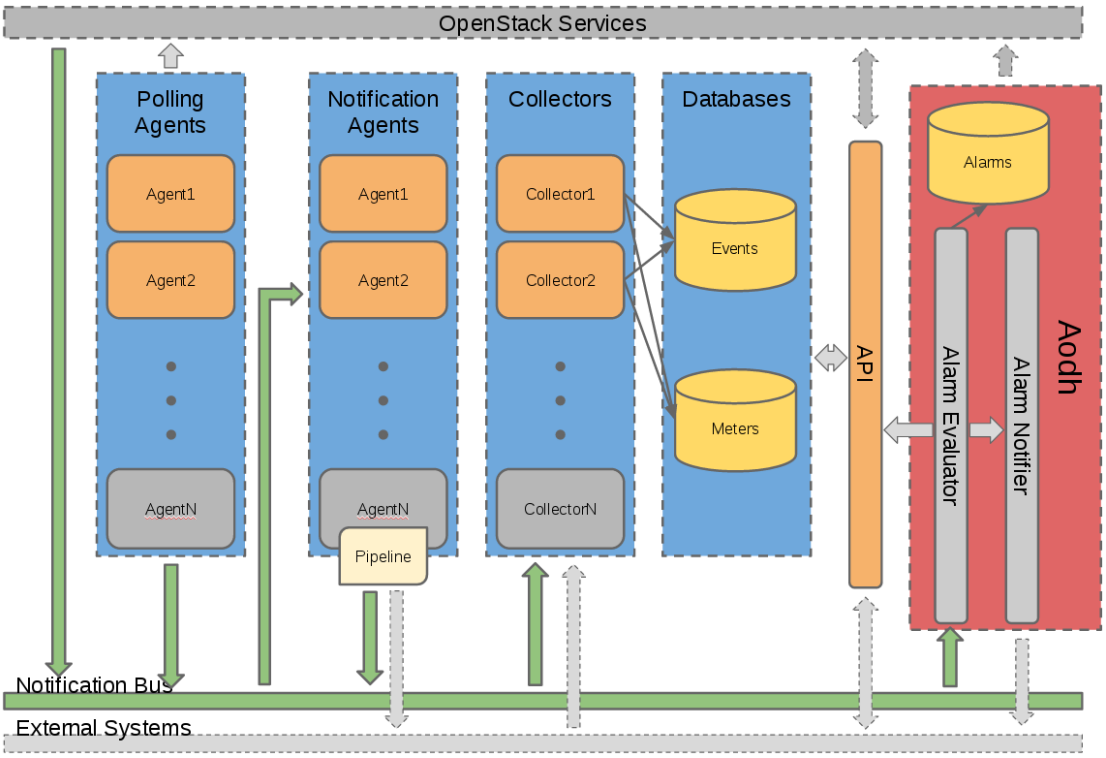

Figure 1. OpenStack Telemetry/Ceilometer architecture

Figure 1 depicts an overall summary of the Telemetry/Ceilometer logical architecture. Each of the Telemetry services are designed to scale horizontally. Additional workers and nodes can be added depending on the expected load. The system consists of the following basic components:

- *Polling agents*; these are:
  - compute agents (ceilometer-agent-compute): they run on each compute node and poll for resource utilisation statistics;
  - central agents (ceilometer-agent-central): it runs on one or more central management servers to poll for resource utilisation statistics for resources not tied to instances or compute nodes;
- *Notification agents*; these run on one or more central management servers to monitor the message queues (for notifications and for metering data coming from the agent);
- *Collectors* (ceilometer-collector): designed to gather and record event and metering data created by notification and polling agents.
- *Databases*, containing Events, Meters and Alarms; these are capable of handling concurrent writes (from one or more collector instances) and reads (from the API module);
- An *Alarm Evaluator and Notifier* (ceilometer-alarm-notifier): Runs on one or more central management servers to allow configuration of alarms based on threshold evaluation for a collection of samples. This functionality is now undertaken by the Aodh module, as will be described later.
- An *API* module (ceilometer-api): Runs on one or more central management servers to provide access to the data from the data store.

Ceilometer offers two independent ways to collect metering data, allowing easy integration of any Openstack-related project which needs to be monitored:

- By listening to events generated on the notification bus, transformed into Ceilometer samples. This is the preferred method of data collection, since it is the most simple and straightforward. It requires, however, that the monitored entity uses the bus to publish events, which may not be the case for all OpenStack-related projects.
- By polling information via the APIs of monitored components at regular intervals to collect information. The data are usually stored in a database and are available through the Ceilometer REST API. This method is least preferred due to the inherent difficulty in making such a component resilient.

Each meter measures a particular aspect of resource usage or on-going performance. All meters have a string name, a unit of measurement, and a type indicating whether values are monotonically increasing (cumulative), interpreted as a change from the previous value (delta), or a standalone value relating only to the current duration (gauge). Samples are individual data points associated with a particular meter and have a timestamp and a value. The aggregation of a set of samples for a specified duration (start-end time) is called a statistic. Each statistic has also an associated time period, which is a repeating interval of time that the samples are grouped for aggregation. Currently there are five aggregation functions implemented: *count*, *max*, *min*, *avg* and *sum*.

Another feature of Telemetry is alarming, which used to be internal to Ceilometer, but moved to a separate project, Aodh [Aodh]. An alarm is a set of rules defining a monitor of a statistic that will trigger when a threshold condition is breached. An alarm can be set on a single meter, or on a combination of meters and can have three states:

- *alarm* (the threshold condition is breached)
- *ok* (the threshold condition is not met)
- *insufficient data* (not enough data has been gathered to determine if the alarm should fire or not).

The transition to these states can have an associated action, which is either writing to a log file or an http post to a URL. The concept of *meta-alarm* is also supported; meta-alarms aggregate over the current state of a set of other basic alarms combined via a logical operator (AND/OR). For example, a meta-alarm could be triggered when three basic alarms become active at the same time.

3.2. Monasca

Monasca [Monasca] is an OpenStack project, aiming at developing an open-source multi-tenant, highly scalable, performant, fault-tolerant monitoring-as-a-service solution, which is integrated within the OpenStack framework. Monasca uses a REST API for high-speed metrics processing and querying, and has a streaming alarm and notification engine. Monasca is being developed by HPE, Rackspace and IBM.

Monasca is conceived to scale up to service provider level of metrics throughput (in the order of 100,000 metrics/sec). The Monasca architecture is natively designed to

support scaling, performance and high-availability. Retention period of historical data is not less than one year. Storage of metrics values, and metrics database query, use an HTTP REST API. Monasca is multi-tenant, and exploits OpenStack authentication mechanisms (Keystone) to control submission and access to metrics.

The metric definition model consists of a (key, value) pair named *dimension*. Basic threshold-based real-time alarms are available on metrics. Furthermore, complex alarm events can be defined end instrumented, based on a simple description grammar with specific expressions and operators.

Monasca agents embed a number of built-in system and service level checks, plus Nagios checks and statsd.

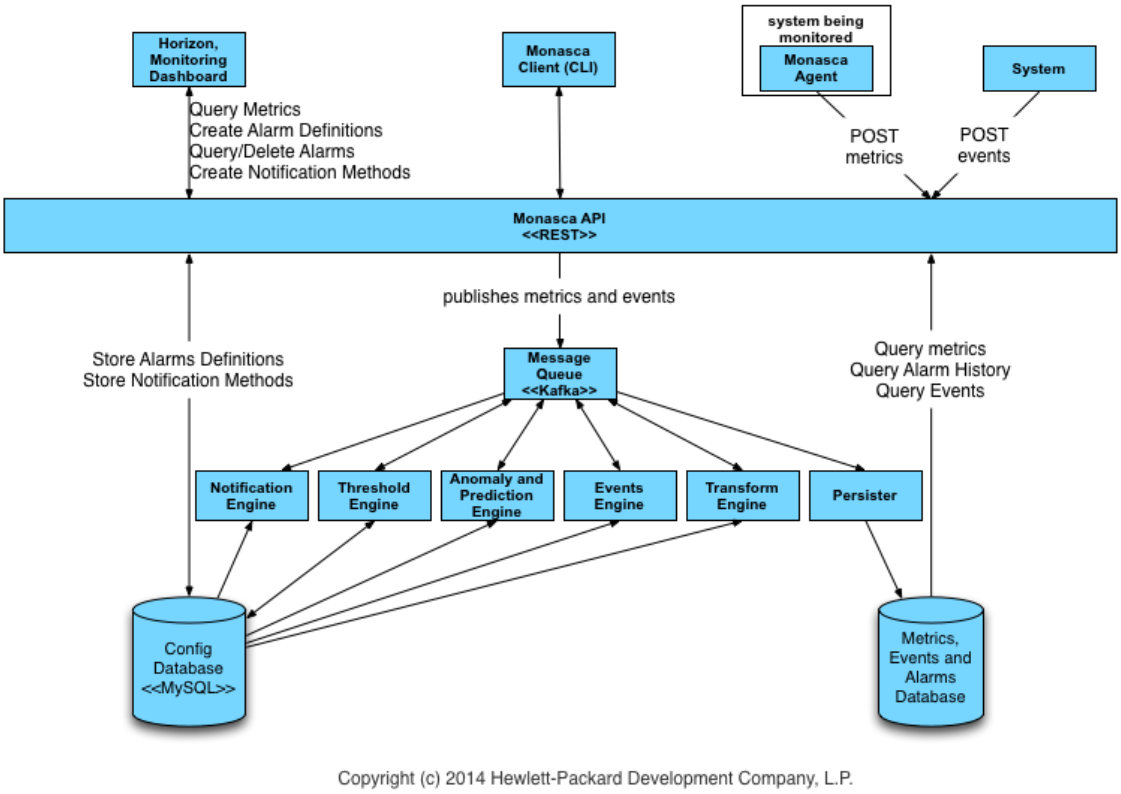

Figure 2. Monasca architecture

Monasca agents are Python based, and consist of several sub-components and supports system metrics, such as cpu utilization and available memory, Nagios plugins, statsd and many built-in checks for services such as MySQL, RabbitMQ, and many others.

The REST API provides an exhaustive set of functions:

- Real-time storage and querying of large amounts of metrics;
- Statistics query for metrics;
- Alarm definition management (create, delete, update);
- Query and cleanup of historical metrics database;
- Compound alarms definition;
- Alarm severity ranking;
- Full storage of alarm transition pattern;

- Management of alarm notification mechanisms;
- Java and Python API available

Published metrics and events are pushed into a Kafka[3] based message queue, from which a component named Persister pulls them out and stores them into the metrics database (HPE Vertica, InfluxDB and Cassandra are supported). Other engine components look after compound metric creation, predictive metrics, notification, and alarm threshold management.

Monasca also includes a multi-publisher plugin for OpenStack ceilometer, able to convert and publish metric samples to the Monitoring API, plus an OpenStack Horizon dashboard as user interface.

Monasca features like real-time alarm processing, integration with OpenStack and scalability/extendibility make it a monitoring system potentially well suitable to be employed within NFV platforms.

3.3. Gnocchi

Gnocchi [Gnocchi] s a project incubated under the OpenStack Telemetry program umbrella, addressing the development of a TDBaaS (Time Series Database as a Service) framework. Its paramount goal is to fix the significant performance issues experienced by Ceilometer in the time series data collection and storage. The root cause of such issues is the highly generic nature of Ceilometer's data model, which gave the needed design flexibility in the initial OpenStack releases, but imposed a performance penalty which is no longer deemed acceptable (storing a large amount of metrics on several weeks makes substantially collapse the storage backend). The current data model on one hand encompasses many options never appearing in real user requests, on the other hand doesn't handle use cases which are overcomplex or too slow to be run. From the aforementioned remarks, the idea of a brand new solution for metrics sample collection was ignited, which brought to the inception of Gnocchi.

Diving deeper into the problem, whereas event collection model in Ceilometer is pretty robust, metrics collection and storage suffers the aforementioned performance flaws. The root of the problem is in the free form metadata associated to each metric, storing a bevy of redundant information which is hard to efficiently query. Gnocchi proposes a faster and scalable pair of time series storage/resource indexer, with a REST API returning an entity (the measured thing) and a resource (information metadata). Differently from Ceilometer, in Gnocchi data stores are separated for metrics and metadata.

[3] http://kafka.apache.org/

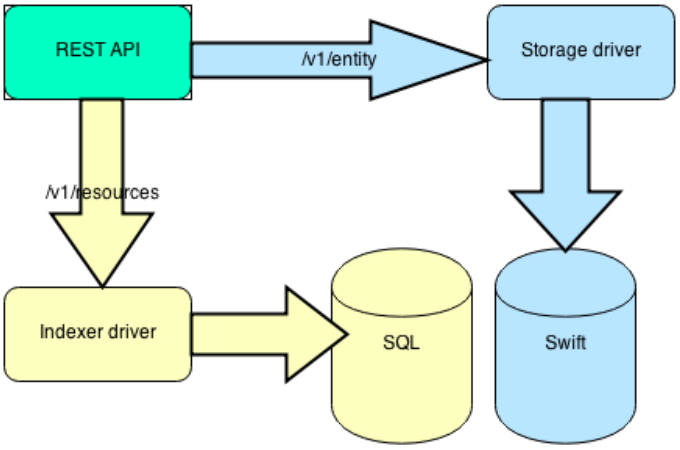

Figure 3. Gnocchi architecture

The storage driver (abstracted) is in charge of metrics storage. Aggregated metrics are actually pre-aggregated before the storage operation occurs, based on the user request at entity time creation. The canonical implementation of time series data (TSD) storage uses Pandas and Swift.

The indexer driver (abstracted as well) uses SQLAlchemy, to exploit the speed and indexable nature of SQL, very well fitting indexing storage. In Gnocchi vision, there will be predefined resource schemas (image, instance,…) to improve indexing and querying at the maximum extent.

Additional functional updates envisioned in Gnocchi include configurable per time-series retention policies.

In future perspective, Gnocchi API should be transitioned to Ceilometer API V3, and its TSDB interaction fully moved into the Ceilometer collector. In an initial phase, Gnocchi should be integrated as self-standing code inside the Ceilometer workflow (via Ceilometer Database Dispatcher).

3.4. Cyclops

Cyclops [Cyclops] is a generic rating-charging-billing (rcb) framework that allows arbitrary pricing and billing strategies to be implemented. The business rules are registered and processed inside of a Drools BPM rule engine [Drools]. The framework itself is organized as a set of micro-services with clear separation of functionalities and communication among them carried over well defined RESTful interfaces.

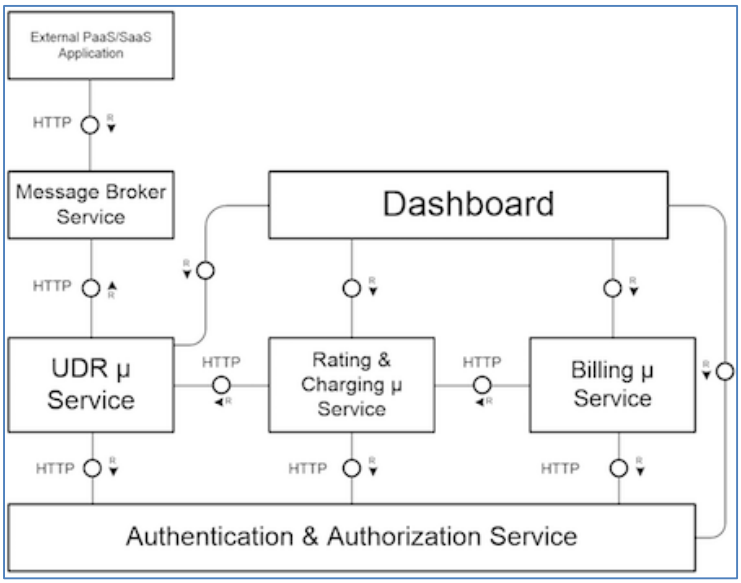

Figure 4. Cyclops architecture

Figure 4 above shows the key micro services and modules that make up the Cyclops framework. A brief explanation of each module is described next:

- udr µ-service: this service is responsible of metric collection for natively supported cloud platforms, currently OpenStack and CloudStack are supported. The metric collection drivers for other popular frameworks including PaaS as public cloud vendors are in the plans. It also persists the collected metrics data into TSDB (InfluxDB 0.9.x data store). For non-natively supported applications, it processes the metrics pushed into the messaging queues as and when they arrive thus enabling the framework to support all sorts of composite and converged billing needs.
- rc µ-service: the rating and charging service processes the udr-records and transforms them into charge-records with cost details. The rating part of the service generates / fetches the rate value for identified services or resources that form in the product portfolio of any provider. The rating rules are processed through drools BPM rules engine, enabling providers to activate dynamic rating and maximize the revenue potential of their portfolio while maintaining consumer satisfaction and loyalty.
- billing µ-service: as the name suggests, this service processes and aggregates all the charge records created by rc µ-service; furthermore it processes any SLA violations and associated penalties for a specified time period. This service also processes any pending service credits, discounts and seasonal offers and applicable regional tax rates before generating the final bill document.
- Auth-n/z µ-service: Cyclops micro-services validate API requests using secure tokens.
- Messaging service: this feature allows external non-natively supported applications and platforms to send in their usage metrics data for further processing by the Cyclops framework.

- Dashboard: this module provides a rich graphical user interface for customers to manage and view their usage charts and bills, and allows admins to control various parameters of the framework and also manage the pricing and billing rules.

As the framework is implemented as a distributed platform, the health status monitoring of various service is critical. For this, currently Sensu [Sensu] is used to track the aliveness of each service. Sensu could also be used to manage the data collection tasks scheduling and triggering. Although the framework designers are migrating towards a self-contained scheduler for their data collection and processing requirements. The usage metrics collection depends heavily on the granularity of the service monitoring implementation.

3.5. Zabbix

Zabbix [Zabbix] is an open source, general-purpose, enterprise-class network and application monitoring tool that can be customised for use with OpenStack. It can be used to automatically collect and parse data from monitored cloud resources. It also provides distributed monitoring with centralised Web administration, a high level of performance and capacity, JMX monitoring, SLAs and ITIL KPI metrics on reporting, as well as agent-less monitoring. An OpenStack Telemetry plugin for Zabbix is already available.

Using Zabbix the administrator can monitor servers, network devices and applications, gathering statistics and performance data. Monitoring performance indicators such as CPU, memory, network, disk space and processes can be supported through an agent, which is available as a native process for Linux, UNIX and Windows platforms. For the OpenStack infrastructure it can currently monitor:

- Core OpenStack services: Nova, Keystone, Neutron, Ceilometer (OpenStack Telemetry), Horizon, Cinder, Glance, Swift Object Storage, and OVS (Open vSwitch)
- Core infrastructure components: MySQL, RabbitMQ, HAProxy, memchached, and libvirtd.
- Operating system statistics: Disk I/O, CPU load, free RAM, etc.

Zabbix is not limited to OpenStack cloud infrastructures: it can be used to monitor VMware vCenter and vSphere installations for various VMware hypervisor and virtual machine properties and statistics.

3.6. Nagios

Nagios is an open source tool that provides monitoring and reporting for network services and host resources [Nagios]. The entire suite is based on the open-source Nagios Core which provides monitoring of all IT infrastructure components - including applications, services, operating systems, network protocols, system metrics, and network infrastructure. Nagios does not come as a one-size-fits-all monitoring system with thousands of monitoring agents and monitoring functions; it is rather a small,

lightweight system reduced to the bare essential of monitoring. It is also very flexible since it makes use of plugins in order to setup its monitoring environment.

Nagios Fusion enables administrators to gain insight into the health of the organisation's entire network through a centralised view of their monitored infrastructure. In addition, they can automate the response to various incidents through the usage of Nagios Incident Manager and Reactor. The Network Analyser, which is part of the suite, provides an extensive view of all network traffic sources and potential security threats allowing administrators to quickly gather high-level information regarding the status and utilisation of the network as well as detailed data for complete and thorough network analysis. All monitoring information is stored in the Log Server that provides monitoring of all mission-critical infrastructure components – including applications, services, operating systems, network protocols, systems metrics, and network infrastructure.

Nagios and Telemetry are quite complementary products which can be used in an integrated solution. The ICCLab, which operates within the ZHAW's Institute of Applied Information Technology, has developed a Nagios plugin which can be used to capture metrics through the Telemetry API, thus allowing Nagios to monitor VMs inside OpenStack. Finally, the Telemetry plugin can be used to define thresholds and triggers in the Nagios alerting system.

3.7. OPNFV Projects

3.7.1. Doctor

Doctor (Fault Management) [Doctor] is an active OPNFV requirements project. Started December 2014, its aim is to build fault management and maintenance framework for high availability of Network Services on top of virtualized infrastructure. The project is supported by engineers from several major telecom vendors as well as telco providers.

So far, the project has produced a report deliverable which was recently released (October 2015) [DoctorDel]. This report identifies use cases and requirements for an NFV fault detection and management system. In specific, the following requirements are identified for a VIM-layer monitoring system:

- Monitoring of resources
- Detection of unavailability and failures
- Correlation and Cognition (especially correlation of faults among resources)
- Notification by means of alarms
- Fencing, i.e. isolation of a faulty resource
- Recovery actions

Doctor has also specified an architectural blueprint for the fault management functional blocks within the NFV infrastructure, as shown in Figure 5. In particular, it is envisaged that certain functionalities for control, monitoring, notification and inspection need to be included in the VIM.

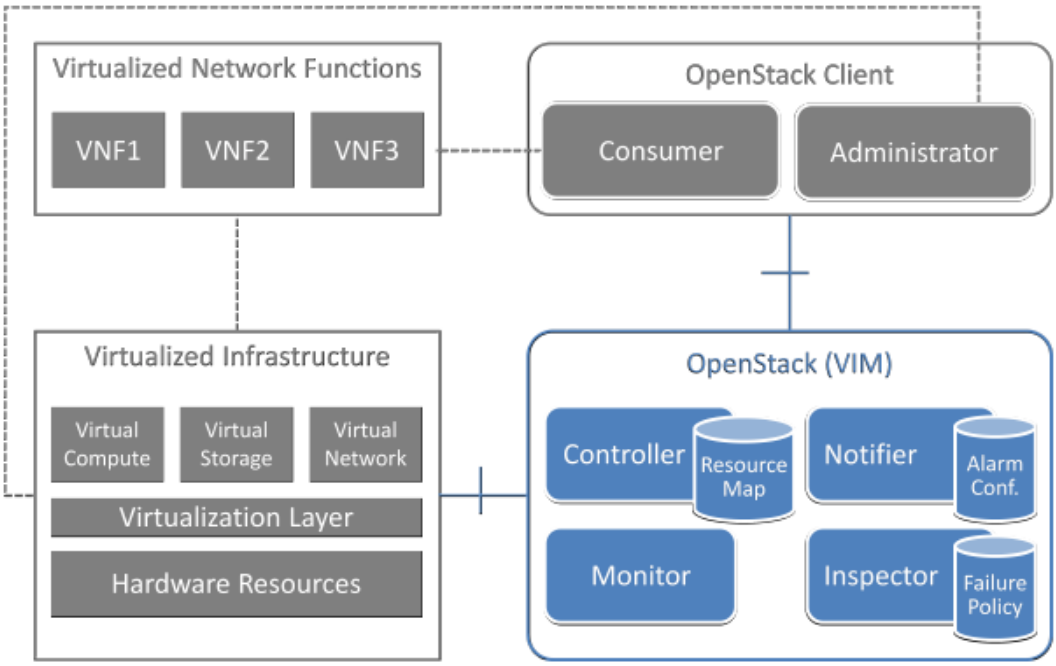

Figure 5. Doctor functional blocks

As a first implementation proposal, the report proposes to re-use and integrate some off-the shelf solutions for these functionalities, namely Ceilometer (see Sec. 3.1) for the Notifier, Zabbix (see Sec. 3.5) for the Monitor and Monasca (see Sec. 3.2) for the Inspector.

However, it is evident that integrating these frameworks results in considerable overlaps, since many functionalities are present in all of them (e.g. metrics collection, storage, alarming etc.) and thus may produce an unnecessarily complex and overprovisioned system. In addition, some key requirements mentioned in the document, such as correlation and root cause detection are not covered by the present versions of these frameworks.

The first version of the Doctor platform has been included in the Brahmaputra release of OPNFV (February 2016).

3.7.2. Prediction

Data Collection for Failure Prediction [Prediction] is another OPNFV project, aiming to implement a system for predicting failures. Notifications produced can be dispatched to the fault management system (see previous section), so that the latter can proactively respond to faults, before these actually happen.

The scope of the project is very promising indeed and also very relevant. However, it is still at a very early stage. Recently, it released a document [Pred-del] in which some early gap analysis is performed with regard to existing monitoring frameworks.

3.8. OpenDaylight monitoring

Since OpenDaylight has been selected as the SDN network controller for T-NOVA, it is relevant to investigate the monitoring capabilities it provides.

The OpenDaylight Statistics Manager module, implements statistics collection, sending statistics requests to all enabled nodes (managed switches) and storing responses in the statistics operational subtree. The Statistics Manager collects information on the following:

- node-connector (switch port)
- flow
- meter
- table
- group statistics

In the Hydrogen and Helium releases, monitoring metrics were exposed via the northbound Statistics REST API.

The Lithium release introduces the Model-Driven Service Abstraction Layer (MDSAL), which stores status-related data in the form of a document object model (DOM), known as a "data tree." MDSAL 's RESTful interfaces for configuration and monitoring are designed based on RESTCONF protocol. These interfaces are generated dynamically at runtime based on YANG models that define its data.

3.9. Other relevant monitoring frameworks

Apart from the frameworks surveyed in the present sections, there exists a large number of IT/Cloud and Network/SDN monitoring tools, many of them open-source, which could be re-used as components of an NFV monitoring platform. Some of the most popular tools are presented in Figure 6, and are briefly overviewed in Annex I.

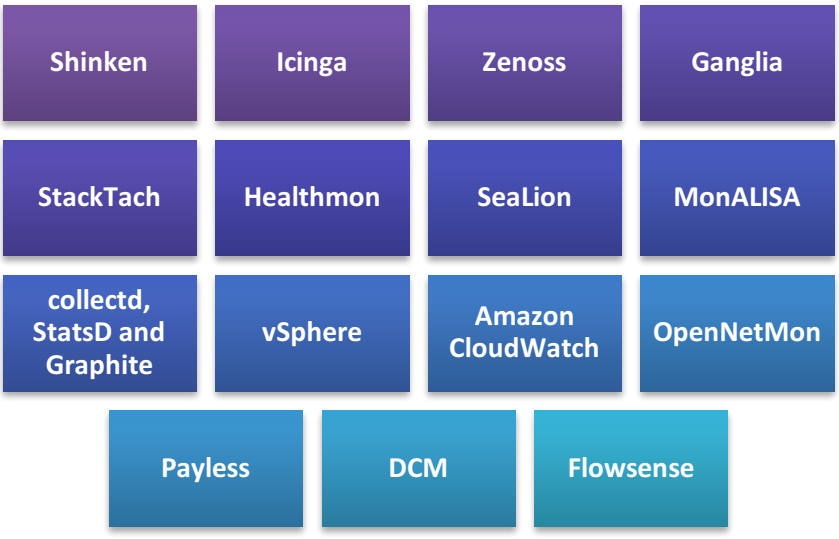

Figure 6. Other relevant Cloud/SDN Monitoring frameworks

While most of these frameworks require considerable effort in order to be adapted to suit the needs of NFV monitoring, there are certain components which are quite mature

and can be reused. For example, in T-NOVA, the collectd module (the core version) is adopted as monitoring agent for VNFs and compute nodes, as will be described in the next section.

3.10. Technology selection and justification

With regard to the basic functionalities identified in Section 2 as requirements for VIM monitoring, metrics collection (Functionality 1) can already be achieved by re-using a number of the pre-existing monitoring mechanisms for virtualised infrastructures, as surveyed in the following section. Apart from selecting and properly integrating the appropriate technologies and possibly selecting the appropriate set of metrics, limited progress beyond the state-of-the-art should be expected in this field.

On the other hand, the actual challenges and envisaged innovation of the monitoring framework are seen to be associated with Functionalities 2 and 3. Specifically, the following challenges have been identified:

- *Events and alarms generation:* Moving beyond the typical approach, which is found in most monitoring systems and is based on static thresholds (i.e. generate an alarm when a metric has crossed a pre-defined threshold) the aim is to study and adopt more dynamic methods for fault detection. Such methods should be based on statistical methods and self-learning approaches, identifying outliers in system behaviour and triggering alarms reactively or even proactively (e.g. before the actual fault has occurred). This anomaly detection procedure, in the context of T-NOVA, can clearly benefit from the fact that the monitored services are composed of VNFs rather than generic VMs. As virtual appliances dedicated to traffic processing, VNFs are expected to expose some common characteristics (e.g. the CPU load is expected to proportionally rise, not necessarily linearly, with the increase of processed traffic). A significant deviation from this correlation could, for example, indicate a potential malfunction.
- *Communication with the Orchestrator*: With this functionality, scalability is the key requirement that needs to be fulfilled. In an operational environment, the Orchestrator is expected to manage tens or hundreds of NFVI-PoPs (or even thousands, if micro-data centres distributed in the access network are envisaged). It is thus impossible for the Orchestrator to handle the full set of metrics from all managed physical and virtual nodes. The challenge is to optimise the communication of monitoring information to the Orchestrator so that only necessary information is transmitted. This optimisation does not only imply fine-tuning of polling frequency, careful definition of a minimal set of metrics or proper design of the communication protocol, but also requires an intelligent aggregation procedure at VIM level. This procedure should achieve the grouping/aggregation of various metrics from different parts of the infrastructure as well as of alarms, and the dynamic identification of the information that is of actual value to the Orchestrator.

To achieve the aforementioned innovations, Task 4.4 work plan involves in its initial stage the establishment of a baseline framework which fulfils the basic functionalities

by collecting and communicating metrics and, as a second step, the study, design and incorporation of innovative techniques for anomaly detection and metrics aggregation.

There are many alternative ways towards this direction, whose assessment is overviewed in the following table.

Table 2. Assessment of various implementation choices for VIM monitoring

Implementation choice for T-NOVA VIM monitoring	Pros	Cons
Integration of required functionalities (push meters, statistical processing, integration of guest OS and VNF metrics) into Ceilometer and Aodh.	Direct integration into Openstack, contribution to a mainstream project. Takes advantage of Ceilometer's open and modular architecture.	Will require intrusive interventions into Ceilometer. Solution will be Openstack-specific and also version-specific. Also, Ceilometer suffers specific scalability issues.
Adoption of Monasca, with some extensions (push meters, statistical processing, integration of guest OS and VNF metrics)	Monasca is a complete monitoring system with remarkable scalability and also quite mature. Its REST API already provides an exhaustive set of functions. Monasca has an open and modular architecture.	Monasca is quite complex and resource-demanding, involving many capabilities which are not required in T-NOVA, given that metrics are also processed at Orchestrator. Requires a special monitoring agent (monasca-agent).
Extension of Gnocchi (as a TDBaaS framework) with all necessary communication and processing tools	Gnocchi is quite mature and advancing rapidly, is also well integrated with Ceilometer to provide scalability.	Will need to implement several extensions for communication and processing, since Gnocchi mainly provides a storage solution.
Extension of Cyclops with all necessary communication and processing tools	Quite mature solution, know-how available within T-NOVA consortium (Cyclops is developed by ZHAW)	Will require extensive modification since Cyclops is mostly a rating-charging-billing platform.
Extension of Nagios or Zabbix with all necessary communication and processing tools	Both are well-proven monitoring frameworks and provide support for multiple systems and applications	Nagios and Zabbix already involve many features and capabilities which are not needed in T-NOVA, and thus their extension would be inefficient, also requiring several modifications.
Integration of specific enablers (agent, time-series	Will result in a tailored solution for T-NOVA needs.	Some functionalities (such as alarming) will have to

DB, existing APIs) into a new framework.	Lightweight and directly configurable.	be redeveloped from scratch.

It seems that most of the existing technological enablers for VIM monitoring, as previously overviewed, can only partially address all the aforementioned challenges in a lightweight and resource-efficient manner. Although most of them are indeed open and modular (such as Monasca), they are already quite complicated and resource-demanding and therefore further expanding them to cover these needs would require considerable effort and would raise efficiency issues. We argue that a "clean-slate" approach towards NFV monitoring at VIM level, exploiting some basic enablers and adding only the required functionalities, is a more optimized approach.

The VIM monitoring framework, which we describe in the next section, aims to provide a lightweight and NFV-tailored contribution towards this direction.

4. THE T-NOVA VIM MONITORING FRAMEWORK

4.1. Architecture and functional entities

The overall architecture of the T-NOVA VIM monitoring framework can be defined by taking into account the technical requirements, as identified in Section 2, as well as the technical choices made for the NFVI and VIM infrastructure. The specification phase has concluded that the OpenStack platform will be used for the control of the virtualised IT infrastructure, as well as the OpenDaylight controller for the management of the SDN network elements.

In this context, it is proper to leverage the OpenDaylight (Statistics API) and OpenStack (Telemetry API) capabilities for collecting metrics, rather than directly polling the network elements and the hypervisors at NFVI layer, respectively.

Theoretically, it would be possible for the Orchestrator to directly poll the cloud and network controllers of each NFVI-PoP and retrieve resource metrics respectively. This approach, although simple and straightforward, would only poorly address the challenges outlined in Section 3.10 and in particularly would introduce significant scalability issues on the Orchestrator side.

Thus, it seems appropriate to introduce a mediator/processing entity at the VIM level to collect, consolidate, process metrics and communicate them to the Orchestrator. We call this entity *VIM Monitoring Manager (VIM MM)*, as a stand-alone software component. As justified in Sec. 3.10, VIM MM is re-designed and developed in T-NOVA as a novel component, without depending on the modification of existing monitoring frameworks.

With regard to the collection of monitoring information, OpenStack and OpenDaylight already provide a rich set of metrics for both physical and virtual nodes, which should be sufficient for most T-NOVA requirements. However, in order to gain a more detailed insight on the VNF and the NFVI status and operation, we consider advisable to also collect a rich set of metrics from the guest OS of the VNF container (VM) - including information which cannot be obtained via the hypervisor – as well as the compute node itself.

For this purpose, we introduce an additional *VNF Monitoring Agent,* deployed within the VNF VMs. The agent intends to augment VNF monitoring capabilities, by collecting a large variety of metrics, as declared in the VNF Descriptor document (VNFD) of each VNF and also at a higher temporal resolution compared to Ceilometer.

The monitoring agent can be either pre-installed in the VNF image or installed upon VNF deployment. It must be noted, however, that in some cases the presence of an agent might not be desirable by the VNF developer for several reasons (e.g. resource constraints, incompatibilities etc.). In this case, the system can also work in agent-less mode, solely relying on Ceilometer data for VNFs which do not have an agent installed.

In addition to collecting generic VNF and infrastructure metrics, the VIM MM is also expected to retrieve VNF-specific metrics from the VNF application itself. For this purpose, we have developed specific lightweight libraries (currently in Python, but

planned to expand to other languages), which can be used by the VNF developer to dispatch application-specific metrics to the VIM MM.

Although traditionally the VNF metrics are supposed to be directly sent to the VNF Manager, for the sake of simplicity we chose to exploit the already established VIM monitoring framework to collect and forward VNF metrics to the VNF Manager through the VIM, rather than implement a second parallel "monitoring channel".

Based on the design choices, outlined above, the architecture of the T-NOVA VIM monitoring framework can be defined as shown in Figure 7 below.

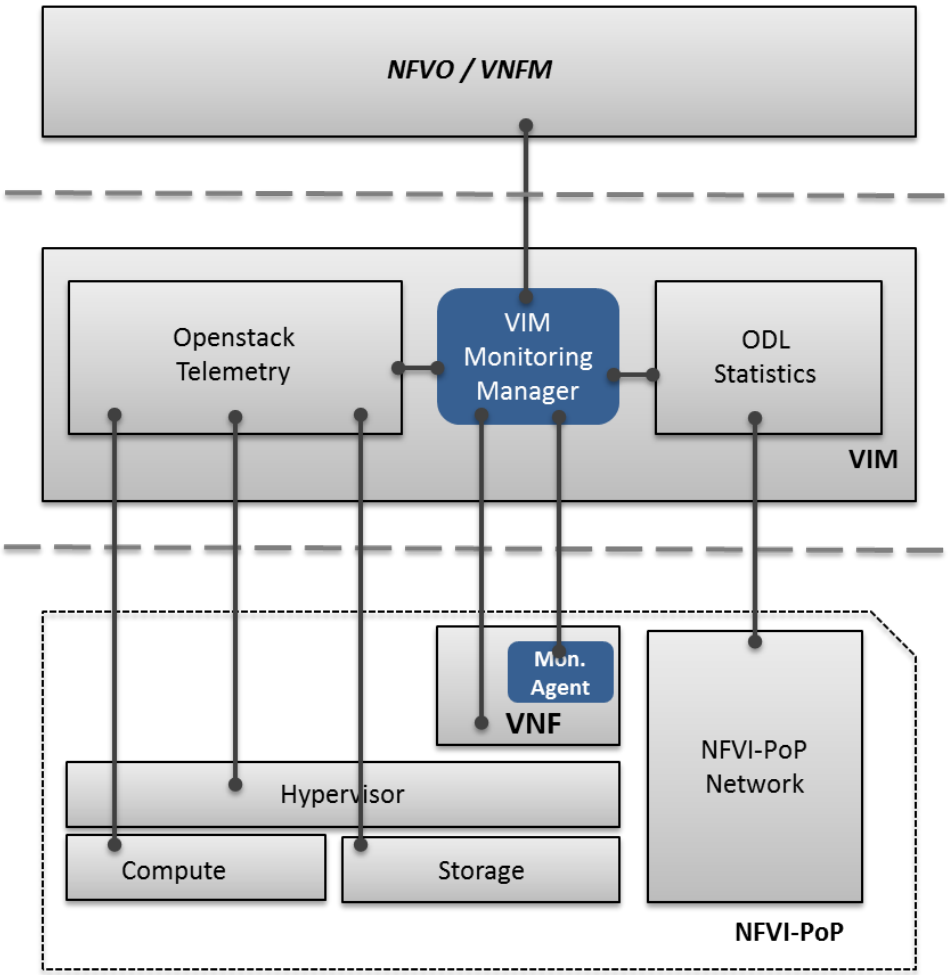

Figure 7. Overview of the VIM monitoring modules

The VIM MM aggregates metrics by polling the cloud and network controllers and by receiving additional information from the monitoring agents as well as the VNF applications, consolidates these metrics, produces events/alarms if appropriate and communicates them to the Orchestrator. For the sake of scalability and efficiency, it was decided that metrics will be pushed by the VIM MM to the Orchestrator, rather than being polled by the latter. Moreover, the process of metrics collection/communication and event generation can be partially configured by the Orchestrator via a relevant configuration service to be exposed by the VIM MM. More details on the introduced modules can be found in the sections to follow.

4.2. Monitoring metrics list

4.2.1. Generic metrics

A crucial task when defining the T-NOVA approach for monitoring is the identification of metrics that need to be collected from the virtualised infrastructure. Although the list of metrics that are available via the existing controllers can be quite extensive, it is necessary, for the sake of scalability and efficiency, to restrict this list to include only the information that is actually needed for the implementation of the T-NOVA Use Cases, as defined in Deliverable D2.1. Table 3 belows summarises a list of such metrics, which are "generic" in the sense that they are not VNF application-specific[4]. This list is meant to be continuously updated throughout the project in order to align with the technical capabilities and requirements of the components under development and the use cases which are implemented.

Table 3. List of generic monitoring metrics

Domain	Metric	Units	Origin	Relevant UCs
VM/VNF	CPU utilisation (user & system)	%	VNF Mon.Agent	UC3, UC4
VM/VNF	Free space in root FS	MB	VNF Mon.Agent	UC3, UC4
VM/VNF	RAM available	MB	VNF Mon.Agent	UC3, UC4
VM/VNF	System load (short/mid/long term)	%	VNF Mon.Agent	UC3, UC4
VM/VNF	No. of processes (running/sleeping etc)	#	VNF Mon.Agent	UC3, UC4
VM/VNF	Network Interface in/out bitrate	Mbps	VNF Mon.Agent	UC3, UC4
VM/VNF	Network Interface in/out packet rate	pps	VNF Mon.Agent	UC3, UC4
VM/VNF	No. of processes	#	VNF Mon.Agent	UC4
Compute Node	CPU utilisation	%	OS Telemetry	UC2, UC3, UC4
Compute Node	RAM available	MB	OS Telemetry	UC2, UC3, UC4
Compute Node	Disk read/write rate	MB/s	OS Telemetry	UC3, UC4
Compute Node	Network i/f in/out rate	Mbps	OS Telemetry	UC3, UC4

[4] Please refer to Sec. 4.2.2 for a list of VNF-specific metrics.

Storage (Volume)	Read/write rate	MB/s	OS Telemetry	UC3, UC4
Storage (Volume)	Free space	GB	OS Telemetry	UC2, UC3, UC4
Network (virtual/physical switch)	Port in/out bit rate	Mbps	ODL Statistics	UC2, UC3, UC4
Network (virtual/physical switch)	Port in/out packet rate	pps	ODL Statistics	UC3, UC4
Network (virtual/physical switch)	Port in/out drops	#	ODL Statistics	UC3, UC4

With regard to metrics identification, a very relevant reference is the ETSI GV NFV-INF 010 document [NVFINF010] which was released December 2014. This document aims at defining and describing metrics which relate to the service quality, as perceived by the NFV Consumer. These metrics are overviewed in the table below.

Table 4. NFV Service Quality Metrics (Source: [NFVINF010])

Service Metric Category	Speed	Accuracy	Reliability
Orchestration Step 1 (e.g. Resource Allocation, Configuration and Setup)	VM Provisioning Latency	VM Placement Policy Compliance	VM Provisioning Reliability VM Dead-on-Arrival (DOA) Ratio
VirtualMachine operation	VM Stall (event duration and frequency) VM Scheduling Latency	VM Clock Error	VM Premature Release Ratio
Virtual Network Establishment	VN Provisioning Latency	VN Diversity Compliance	VN Provisioning Reliability
Virtual Network operation	Packet Delay Packet Delay Variation (Jitter) Delivered Throughput	Packet Loss Ratio	Network Outage
Orchestration Step 2 (e.g. Resource Release)			Failed VM Release Ratio
Technology Component as-a-Service	TcaaS Service Latency	-	TcaaS Reliability (e.g.defective transaction ratio) TcaaS Outage

It can be seen that, apart from the service latency metrics which are related to the provisioning and/or reconfiguration of the service and essentially refer to the response of management commands (e.g. VM start), the rest metrics can be directly or indirectly derived from the elementary metrics identified in Table 3 as well as the events/alarms associated. However, it is up to the Orchestrator, which has a complete view of the service, to assemble/exploit VIM metrics in order to derive the service quality metrics to be exposed to the SP and the Customer via the Dashboard. These metrics will be used as input to enforce the Service Level Agreement (SLA) that will be finally evaluated at Marketplace level for the applicability of possible rewards to the customer in case of failure (see D6.4 – SLAs and billing).

4.2.2. VNF-specific metrics

Apart from the generic metrics identified in the previous section, each VNF generates specific dynamic metrics to monitor its internal status and performance.

These metrics:

- are specified inside the VNF Descriptor (VNFD) as monitoring-parameters (both for the VDUs and for the whole VNF) to define the expected performance of the VNF under certain resource requirements.
- are sent by the VNF application to the VIM Monitoring Manager, either via the agent or directly (see details in Sec. 4.4)
- are processed, aggregated and forwarded, if required, to the upper layers (Orchestrator and Marketplace).

At the Orchestration level, some of the VNF-specific metrics can be used for automating the selection of the most efficient VNF flavour in terms of usage of resources, to achieve a given SLA (for example using automated scaling procedures – see D3.3).

At the Marketplace level, those VNF-specific metrics that may be part of the SLA agreed between SP and customer will be evaluated for business and commercial clauses (e.g: penalties, rewards, etc.) that will finally impact in the billing procedure (see D6.4)

The subsections to follow overview a list of VNF-specific metrics for each of the VNFs being developed in T-NOVA. The lists which follow are tentative and are meant to be continuously updated as the VNF applications evolve.

Please also note that most of these metrics refer to the specific functionality of each VNF as well as its component software modules. For a detailed description of the T-NOVA VNFs, please refer to Deliverable D5.32 [D532].

4.2.2.1. vSBC metrics

The vSBC components (VNFCs) able to generate metrics are: LB, IBCF, BGF and O&M (please refer to [D532] for more details).

These data are collected by the O&M component, and sent to the Monitoring Manager via the monitoring agent, using the SNMP protocol.

The following table sums up the VNF-specific metrics for the vSBC functions.

Table 5. vSBC monitoring metrics

Metric	Description	Units
total_sip_sessions	Total number of confirmed SIP sessions	Integer (incremental)
rtp_pack_in	Number of incoming RTP packets	Integer (incremental)

rtp_pack_out	Number of outgoing RTP	Integer (incremental)
rtp_pack_in_byte	Number of incoming RTP bytes	[Byte] (incremental)
rtp_pack_out_byte	Number of outgoing RTP bytes	[Byte] (incremental)
rtp_frame_loss	RTP frame loss	Integer (incremental)
average_latency	Average RTP delay	[Msec]
max_latency	Maximum RTP delay	[Msec]
average_interarrival_jitter	Average inter-packet arrival jitter	[Msec]
max_interarrival_jitter	Maximum inter-packet arrival jitter	[Msec]
number_of_in_transcoding	Number of incoming transcoding procedures	Integer (incremental)
number_of_out_transcoding	Number of outgoing transcoding procedures	Integer (incremental)
number_of_in_transrating	Number of incoming transrating procedures	Integer (incremental)
number_of_out_transrating	Number of outgoing transrating procedures	Integer (incremental

We point out that:

- the SIP metrics are related to the control plane monitoring
- the rest metrics are related to the media plane monitoring

At the receipt of a new SNMP GET request coming from the monitoring agent, all these metrics are reset, while their enhancement starts again.

These metrics may be strongly influenced by:

- incoming packet sizes(i.e : 64, 128, 256,……, 1518 byte)
- hardware and software acceleration technologies (i.e: DPDK or GPU). In particular the GPU hardware accelerators might be used, in tandem with standard processors, in case of intensive processing (i.e: video trascoding/transrating).

4.2.2.2. vTC metrics

Table 6 overviews the metrics reported by the virtual Traffic Classifier VNF (vTC).

Table 6. vTC monitoring metrics

Metric	Description	Units
pps	Packets per second processed	pps
flows	Flows per second	# (average)
totalflows	Total flows	# (incremenetal)
protocols	Application Protocols	# (incremental)
mbits_packets_all	Total Throughput	Mbps
mbits_packets_bittorrent	BitTorrent application rate	Mbps
mbits_packets_dns	DNS application rate	Mbps
mbits_packets_dropbox	Dropbox application rate	Mbps
mbits_packets_google	Google application rate	Mbps
mbits_packets_http	HTTP application rate	Mbps
mbits_packets_icloud	iCloud application rate	Mbps
mbits_packets_skype	Skype application rate	Mbps
mbits_packets_twitter	Twitter application rate	Mbps
mbits_packets_viber	Viber application rate	Mbps
mbits_packets_youtube	Twitter application rate	Mbps

4.2.2.3. vSA metrics

The metrics reported by the virtual Security Appliance (vSA) are shown in Table 7. The vSA metrics correspond to the two vSA components (snort and pfsense).

Table 7. vSA monitoring metrics

Metric	Description	Units
vsa_pfsense_lan_inerrs	Number of errors coming into the lan interface of pfsense	# (incremental)
vsa_pfsense_lan_outerrs	Number of errors going out of the lan interface of pfsense	# (incremental)
vsa_pfsense_wan_inerrs	Number of errors coming into the wan interface of pfsense	# (incremental)
vsa_pfsense_wan_outerrs	Number of errors going out of the wan interface of pfsense	# (incremental)
vsa_pfsense_lan_inbytes	Number of bytes coming into the lan interface of pfsense	# (incremental)
vsa_pfsense_lan_outbytes	Number of bytes going out of the lan interface of pfsense	# (incremental)

vsa_pfsense_wan_inbytes	Number of bytes coming into the wan interface of pfsense	# (incremental)
vsa_pfsense_wan_outbytes	Number of bytes going out of the wan interface of pfsense	# (incremental)
vsa_pfsense_lan_inpkts	Number of packets coming into the lan interface of pfsense	# (incremental)
vsa_pfsense_lan_outpkts	Number of packets going out of the lan interface of pfsense	# (incremental)
vsa_pfsense_wan_inpkts	Number of packets coming into the wan interface of pfsense	# (incremental)
vsa_pfsense_wan_outpkts	Number of packets going out of the wan interface of pfsense	# (incremental)
vsa_pfsense_cpu	Cpu usage of pfsense	float (0.0-1.0)
vsa_pfsense_mem	Memory usage of pfsense	float (0.0-1.0)
vsa_pfsense_dis	Hardware usage of pfsense	float (0.0-1.0)
vsa_pfsense_load_avg	Average load of pfsense	float (0.0-1.0)
vsa_pfsense_pfstate	State table size of pfsense	float (0.0-1.0)
vsa_snort_cpu	Cpu usage of snort	float (0.0-1.0)
vsa_snort_memory	Memory usage of snort	float (0.0-1.0)
vsa_snort_pkt_drop_percent	Percent of dropped packets, generated by snort	%
vsa_snort_alerts_per_second	Number of alerts per second, generated by snort	int
vsa_snort_kpackets_per_sec.realtime	How many thousands of Packets per second in realtime through vsa, generate by snort	float
vsa_pfsense_uptime	Pfsense uptime	String

4.2.2.4. vHG metrics

Table 8 summarises the virtual Home Gateway (vHG) metrics.

Table 8. vHG monitoring metrics

Metric	Description	Units
remaining_storage_size	Remaining Storage Size	Bytes
transcoding_score	Transoding Score	double
httpnum	Number of HTTP requests received	# (incremental)

hits	Cache hits percentage of all requests for the last 5 minutes	%
hits_bytes	Cache hits percentage of bytes sent for the last 5 minutes	%
cachediskutilization	Cache disk utilization	%
cachememkutilization	Cache memory utilization	%
usernum	Number of users accessing the proxy	#

4.2.2.5. vProxy metrics

The metrics reported by the virtual proxy (vProxy) VNF are summarized in Table 9. Most of them are bound to the specific proxy implementation (squid), but can be extended to match other implementations as well.

Table 9. vProxy monitoring metrics

Metric	Description	Units
httpnum	Number of HTTP requests received	# (incremental)
hits	Cache hits percentage of all requests for the last 5 minutes	%
hits_bytes	Cache hits percentage of bytes sent for the last 5 minutes	%
memoryhits	Memory hits percentage for the last 5 minutes (hits that are logged as TCP_MEM_HIT)	%
diskhits	Disk hits percentage for the last 5 minutes (hits that are logged as TCP_HIT)	%
cachediskutilization	Cache disk utilization	%
cachememkutilization	Cache memory utilization	%
usernum	Number of users accessing the proxy	#
cpuusage	CPU consumed by Squid for the last 5 minutes	%

4.3. VNF Monitoring Agent

The VNF Monitoring Agent comes either pre-installed within the VM image hosting the VNFC or installed upon VNFC deployment. It will be automatically launched upon VNF start-up and run continuously in the background. The agent collects a wide range of metrics from the local OS.

For the implementation of the monitoring agent, we exploit the the popular collectd-core module [collectd] (also see Annex I, Sec. 9.1.1.8.). Collectd-core comes in a

package already available in most Linux distributions and can be directly installed with relatively minimal overhead.

Given that the list of available collectd plugins is quite extensive, we have selected a basic set of plugins to be used in T-NOVA, in order to cover all generic metrics, as identified in Sec. 4.2.1 but also to capture most vital meters of the system, without on the other hand introducing too much overhead. These plugins, accompanied by a brief description and the metrics which are collected, are overviewed in Table 10 below.

Table 10. Collectd plugins used in T-NOVA

Plugin	Description	Metrics
CPU	Collects the amount of time spent by the CPU in various states, most notably executing user code, executing system code, waiting for IO-operations and being idle.	• user • interrupt • softirq • steal • nice • system • idle • wait
Memory	Collects physical memory utilization.	• used • buffered • cached • free
Disk	Collects performance statistics of hard-disks and, where supported, partitions.	• octets.read • octets.write • ops.read • ops.write • time.read • time.write • merged.read • merged.write
Interface	Collects information about the traffic (octets per second), packets per second and errors of interfaces	• if_octects.rx • if_octects.tx • if_packets.rx • if_packets.tx • if_errors.rx • if_errors.tx
Processes	Collects the number of processes, grouped by their state (e. g. running, sleeping, zombies, etc.).	• ps_state-running • ps_state-sleeping • ps_state-zombies • ps_state-stopped • ps_state-paging • ps_state-blocked • fork_rate

For the communication of metrics, the Monitoring Agent features a TCP or UDP dispatcher which pushes measurements to the VIM MM periodically. The push frequency will be configurable (either manually or automatically). The set of metrics (selection among all available ones) to be communicated will also vary among VNFs, and will be defined in the VNFD.

4.4. Collection of VNF-specific metrics

The VIM monitoring framework provides several options for collecting VNF metrics; each VNF developer may choose the most appropriate option which suits their requirements, policies and constraints.

The *direct* communication method involves the VNF application itself reporting selected metrics as key-value pairs to the VIM MM at arbitrary intervals. For this purpose, we have developed a set of lightweight libraries (currently in Python and Java) which the VNF provider/developer can integrate in the application. This way, the VNFP can use the methods provided to easily and quickly dispatch internal application metrics without knowing the internals and interfaces of the monitoring framework.

The *indirect* communication method implies that all VNF metrics are collected by the monitoring agent (collectd) by means of plugins. This can be done in three alternative ways:

1. Using the collectd "*Snmp*" plugin. This option is appropriate in cases when the VNF already exposes an SNMP service. In this case, the metrics to be collected are described by a Management Information Base (MIB). The MIB includes the VNF/VDU identifiers, and uses a hierarchical namespace containing object identifiers (OIDs); each OID identifies a variable that can be read via the SNMP protocol (see RFC 2578). The T-NOVA agent (collectd) issues standard SNMP GET requests periodically to the VNF SNMP service for these specific OIDs and gets back the values, which in turn communicates to the VIM MM.
2. Using the collectd "*Tail*" plugin. This is the simplest method which requires minimal integration with the VNF. With this approach, the VNF application dumps metrics as entries in a log file with known format. The collectd Tail plugin parses the log file after each update, extracts the metrics and communicates to the VIM MM.
3. Using the collectd "*Custom*" plugin. This is the most complicated method and requires the VNFP to develop a special collectd plugin for the VNF. However, this might be the preferred choice for some VNFPs who in any case want to add collectd support in their VNF, given that collectd is very widely used also outside T-NOVA and already integrated with some of the most popular monitoring frameworks.

4.5. Monitoring of FPGA-based VNFs

The T-NOVA project attempts to expand the NFV purview to heterogeneous compute architecture such as GPUs and FPGAs. Utilizing such specialized hardware has direct effects on the monitoring infrastructure that should be used to support it. This section explores the corollaries of the use of programmable logic as compute nodes in the T-NOVA environment.

Monitoring programmable logic devices represents unique challenges in comparison to standard CPUs. Many of the notions present in the latter are not present in programmable logic devices and furthermore programmable logic-based systems can show large disparities which makes the task of providing one overarching concept exceedingly difficult.

As a starting point for our measurement architecture definition we use the programmable cloud platform architecture introduced in D4.1. In this architecture we assume that a FPGA SoC (System-on-Chip) is used as the compute node. An FPGA SoC consists of a Processing System (PS), which comprises one or more CPUs and the Programmable Logic (PL). In this architecture the PS execute the OpenStack worker and any software requirement for the management of the programmable resources available in the PL, while the actual VNFCs to be monitored are deployed to the PL.

In this scheme the monitoring infrastructure is by necessity also divided into two components. One component resides in SW and is executed in the PS. It consists of a software stack that collects statistics from the HW components and forwards them to the monitoring manager and can also be used to monitor the performance of the PS if this is desired.

This software stack comprises several components and is further divided into a kernel-mode part, which interacts with the DMA (Direct Memory Access) driver to send data to and receive them from the programmable logic and a user-mode part that packages the data and sends them to the remote monitoring manager. The kernel-mode part utilizes the same DMA driver that's used to transfer the application data and complements that with a data mover program that exposes a block-level device to the user-mode part, which allows the latter to exchange data with the driver. The implementation of this mechanism is based on the same zero-copy principle as the application data exchange in order to optimize performance. The user-mode part of the data mover then writes the measurement data into a pre-defined storage location whence they're fetched by a shell script which interfaces with the monitoring manager.

The HW component on the other hand is responsible for collecting all of the relevant parameters on the HW side and forwarding them to the software component. These parameters are specific to each VNFC and thus it's up to the uses to provide the appropriate connections and circuits for it. The FPGA SoC platform provides the user with infrastructure which can be used to send that data on to the software component.

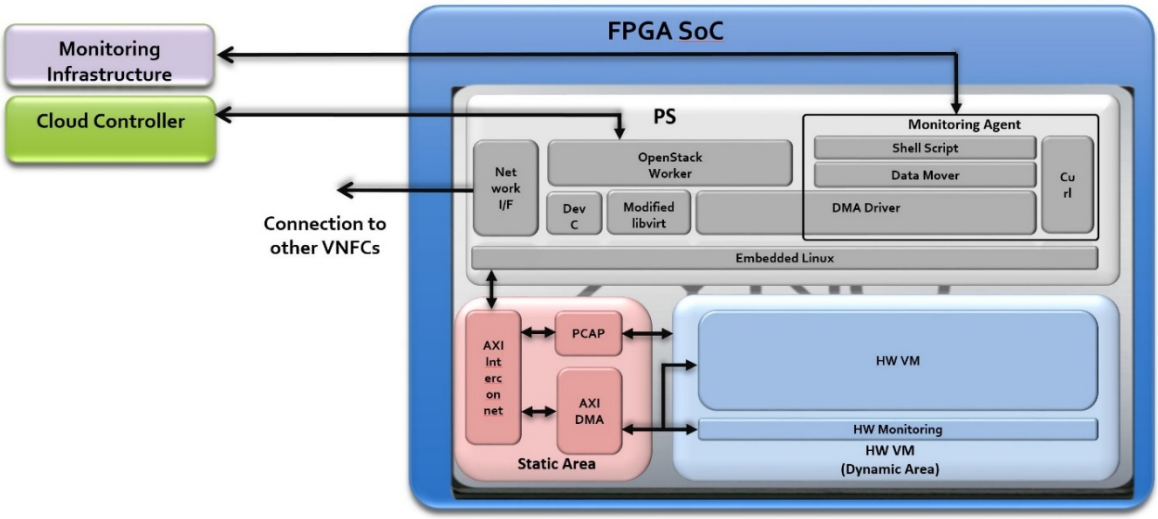

Figure 8. Monitoring architecture for the T-NOVA FPGA SoC

The HW monitoring component shares the DMA with the VNFC also deployed on the FPGA. This is enabled by using separate DMA channels for the data transfers, a feature offered by the DMA block used in the design. The DMA block offers one interface for all inputs and uses additional signal to discern the user which provided the data. It then transfers that to the appropriate memory address space from which the SW monitoring application can read the data.

This scheme provides a clear, expandable, standard interface for the HW VM to transfers its monitoring data to the SW component and a straightforward method for the SW to read the data and perform any processing required before sending it on to the monitoring manager.

4.6. VIM Monitoring Manager architecture and components

4.6.1. VIM MM Architecture

Aligned with the requirements and the design choices set in the previous sections, the functional components of the VIM Monitoring Manager are depicted in Figure 9 and described in this section and in the ones which follow.

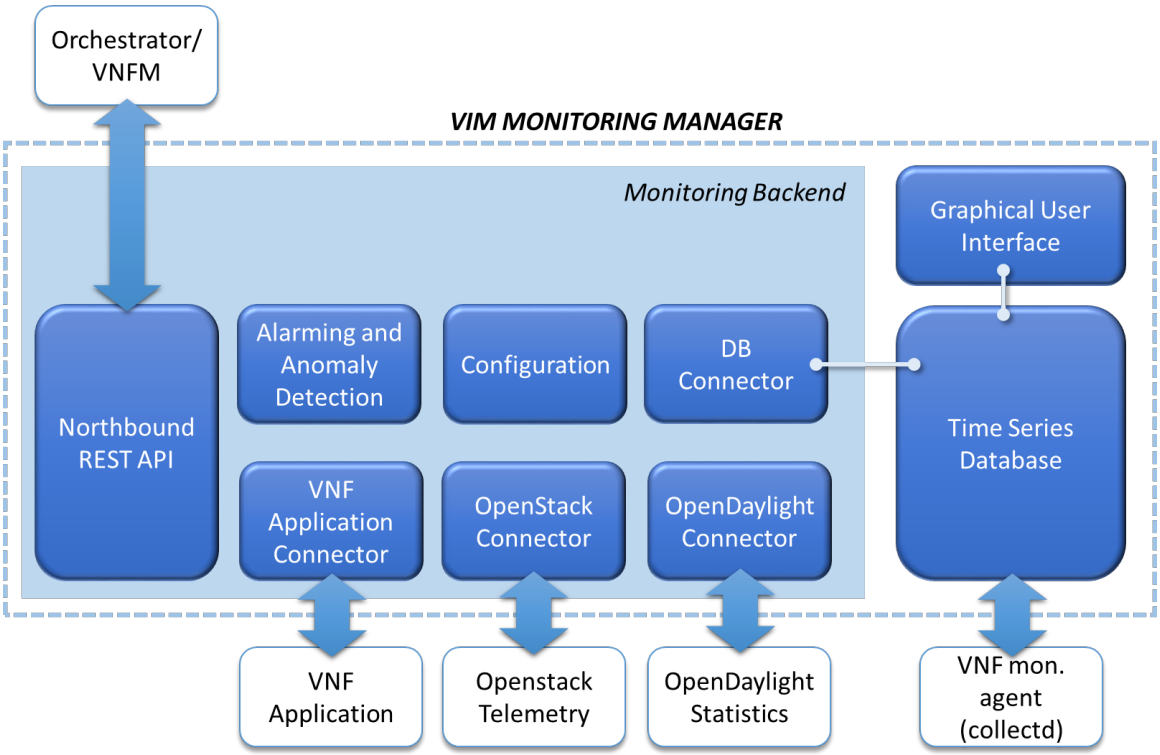

Figure 9. VIM MM functional components

The Monitoring Backend is the core component of the monitoring framework. It is developed in JavaScript and uses the node.js [nodejs] framework to run as a server-side application. The reason behind this choice is that JavaScript matches an asynchronous, event-driven programming style, optimal for building scalable network applications. The main functionality of VIM MM is data communication and the node.js ecosystem offers several services to facilitate communication, especially via web services, as well as event-driven networking.

The backend itself is divided to the following modules:

- **Database connector.** This module accesses the time-series database (see Sec. 4.6.4) in order to write and to read measurements. This module uses influent[5], an InfluxDB Javascript driver.
- **OpenStack and OpenDaylight connectors.** These modules perform requests to various OpenStack and OpenDaylight services in order to acquire cloud- and network-related metrics (see Sec. 4.6.2). The Openstack connector communicates with Keystone, the OpenStack Identity service, in order to generate tokens that can be used for authentication and authorisation during the rest of the OpenStack queries. It polls Nova, the OpenStack Compute service, in order to get the available instances. Finally, it polls Ceilometer, the OpenStack Telemetry service, in order to receive the latest measurements

[5] https://github.com/gobwas/influent

of the instances. The request-promise[6] npm (node.js package) module provides here the HTTP client to perform all these requests.

- **Northbound REST API.** This module exposes all the recorded measurements via HTTP and offers the ability to subscribe to specific measurement events. hapi.js[7] has been used as a framework to build this. hapi.js plugins were also used, such as joi[8] for validation, hapi-swaggered[9] and hapi-swagger[10] for Swagger documentation generation. See Sec. 4.6.3 for more details.
- **Alarming and anomaly detection.** This module performs statistical processing using outlier detection methods in order to derive events and alarms from multiple metrics, without individual thresholds (see Sec. 4.6.5 for more details)
- **VNF Application connector**. It accepts data periodically dispatched by each VNF application, filters them and stored. These metrics are specific to each VNF (e.g. number of flows, sessions etc.). The list of metrics to be collected as well as the dispatch frequency are described in the VNF Descriptor (VNFD).
- **Configuration.** This module allows the use of local files in order to load settings. Node-config[11] has been used here to define a set of default parameters and extend them for different deployment environments, e.g., development, QA, staging and production. Configurations are stored in configuration files within the backend application and can be overridden and extended by environment variables.

The default config file is called *config/default.json* and the administrator may create multiple files in the config directory with the same format, which can later be used by the backend application if the environment variable *NODE_ENV* is set to the configuration file without the .json suffix, e.g. for config/production.json the following command needs to be invoked on a Bash-compatible shell:

export NODE_ENV=production

The configuration parameters that are currently available are the following:

Config Parameter	Description
loggingLevel	Sets the logging level. Available levels are *debug*, *warn* and *info*.
database	Connection information for the time-series database. Required information should be entered in the following strings: *host*, *port*, *username*, *password* and *name* (for the target database name).
identity	Connection information for the OpenStack Keystone service. Required information should be entered in the following

[6] https://www.npmjs.com/package/request-promise
[7] http://hapijs.com/
[8] https://github.com/hapijs/joi
[9] https://github.com/z0mt3c/hapi-swaggered
[10] https://github.com/glennjones/hapi-swagger
[11] https://github.com/lorenwest/node-config

	strings: *host*, *port*, *tenantName*, *username* and *password*. It should be noted that the tenant whose credentials must have sufficient privileges to access all the necessary OpenStack VNF instances.
ceilometer	Connection information for the OpenStack Ceilometer service. Required information should be entered in the following strings: *pollingInterval*, *host* and *port*. The *pollingInterval* sets the time period during which the backend polls Ceilometer for measurements.
nova	Connection information for the OpenStack Nova service. Required information should be entered in the following strings: *host* and *port*.

4.6.2. Interfaces to cloud and network controllers

Monitoring of computing, hypervisor and storage status and resources are performed directly via the OpenStack Ceilometer framework. The VIM MM (OpenStack connector) periodically polls the Telemetry API for metrics regarding all deployed physical and virtual resources. Although these metrics could be retrieved by directly accessing the Ceilometer database, since the scheme of the latter may evolve in future OpenStack versions, it is more appropriate to use the REST-based Telemetry API. The VIM MM issues GET requests to the service referring to a specific resource and meter, and the result are returned in JSON format.

Fortunately, the Telemetry support for the hypervisor selected for T-NOVA (libvirt) offers the widest possible list of available monitoring metrics, compared to other hypervisors, such as Xen or vSphere.

The current version of the OpenStack connector has the following workflow:

- **Token management.** Communication with the OpenStack API requires always a valid token. The backend uses the Openstack Keystone service to acquire a valid token, which is used for every transaction. The token is being checked before submitting any request and if it is expired, it gets renewed.
- **Instance information retrieval.** The backend does not know a priori which instances are to be monitored. By posting a request at the Nova API, it gets a list of the active instances in order to proceed with the measurement request.
- **Measurement retrieval** Once the backend knows the existence of an active OpenStack instance, it is able to retrieve specific measurements for it. Currently CPU utilisation, incoming and outgoing bytes rate are supported, but the list is quickly expanded with other metrics that OpenStack Ceilometer supports.

This workflow is being performed with a time period that can be set with the *pollingInterval* parameter as aforementioned.

Moreover, collecting metrics via the API allows exploiting additional features of Telemetry such as:

- *Meter grouping:* it is possible to define set of metrics and retrieve an entire set with a single query;
- *Sample processing:* it is possible to define basic aggregation rules (average, max/min etc.) and retrieve only the aggregate instead of a set of metrics;
- *Alarming:* it is possible to set alarms based on thresholds for the collection of samples. An alarm can depend on a single meter, or a combination. The VIM MM may use the API to set an alarm and define an HTTP callback service to be called when the alarm has been set off.

Monitoring of physical and virtual elements is achieved via the OpenDayLight interface. The VIM MM backend uses the available REST API of OpenDayLight to receive respective statistics. Each REST API request is encapsulated with an Authorisation Header, which contains a username and password with sufficient privileges. Information of every network device is periodically requested and stored to the time-series database, so that users can later access them through the monitoring API.

The statistics that are currently requested and stored are the following:

- **Port Statistics:** rx packet count, tx packet count, rx byte count, tx byte count, rx drop count, tx drop count.
- **Table Statistics:** active count, lookup count, matched count, maximum supported entries.

4.6.3. Northbound API to Orchestrator

The VIM Monitoring Framework offers a Northbound API to the Orchestrator in order to inform the latter of the newest measurements and in the future for possible alerts. An HTTP RESTful interface provides the latest measurements upon requests and the ability to subscribe to measurements.

The latest draft of the ETSI NFV IFA document [NFVIFA005] which provides an insight to the Or-Vi reference point, contains a high-level specification of the requirements and the data structures which need to be adopted for infrastructure management and monitoring. The requirements related to monitoring can be summarized into the following:

- The VIM must support querying information regarding consumable virtualised resources
- The VIM must issue notifications of changes to information regarding resources
- The VIM must offer full support for alarming (alarm creation/ modification/ subscription/ issue/ deletion)
- The VIM must issue notifications for infrastructure faults

The Northbound API provided by the T-NOVA VIM Monitoring Framework intends to align with these requirements.

4.6.3.1. Querying

First, a set of REST GET endpoints support the transmission of the latest measurements of every available type for every instance being monitored.

The template of such URL is **/api/measurements/*{instance}.{type}***, where *instance* is the Universally Unique Identifier (UUID) given by the OpenStack deployment to the instance and *type* one of the supported measurement types. The currently supported measurement types are:

- *cpu_util* (CPU utilisation)
- *cpuidle* (CPU idle usage)
- *fsfree* (free space on the root filesystem)
- *memfree* (free memory space)
- *network_incoming* (the rate of incoming bytes) and
- *network_outgoing* (the rate of outgoing bytes)

The format of the answer is a JSON object whose fields are the following:

- *timestamp*: shows the timestamp the measurement was taken
- *value*: shows the actual measurement value
- *units*: shows the measurement units

These endpoints require constant polling in order to retrieve their values. If a system requires a constant stream of measurements at specific interval times, then it could use the subscription endpoint.

4.6.3.2. Meters/notifications push

The VIM MM enables a publish-subscribe communication model for pushing of metrics and events to the Orchestrator. In order to subscribe for measurement events, it is required to provide the following information in the form of a JSON object:

- *types*: This is an array of the measurement types. The supported types are the same ones as the ones in the GET endpoints.
- *instances*: This is an array of the instances that have to be monitored. The UUIDs of the instances are also used here.
- *interval*: This is the interval time the monitoring backend has to wait before sending a new set of measurements. The time should be given in minutes.
- callbackUrl: This is the URL the monitoring backend has to callback in order to submit the newest measurements.

This JSON object has to be submitted as a payload in a POST request to the endpoint **/api/subscribe**. Upon transmission, a confirmation message is sent back as response and after the specified interval, a message is given to the *callbackUrl*, similar to the ones one can get via the GET endpoints.

4.6.3.3. Alarming

The VIM MM offers methods for creating alarms and dispatching callbacks whenever the status of alarm changes. Via the API, it is possible to:

- Create an alarm trigger, defining an instance ID, a metric, a comparison operator (less, less or equal, not equal, greater or equal, greater) and a threshold. The creation request is also accompanied by a callback URL; as soon as the expression becomes true, the alarm notification is dispatched to a callback URL. The alarm notification contains a reference to the expression which was validated, the instance(s) which produced the alarm and the timestamp of the measurement.
- Delete an alarm
- Retrieve the details of a set alarm

More details on the above functions can be found in Annex II.

4.6.3.4. Live API documentation

For the convenience of API consumers, a Swagger-UI endpoint is given at **/docs**, where users can refer to for up-to-date information (Figure 10)

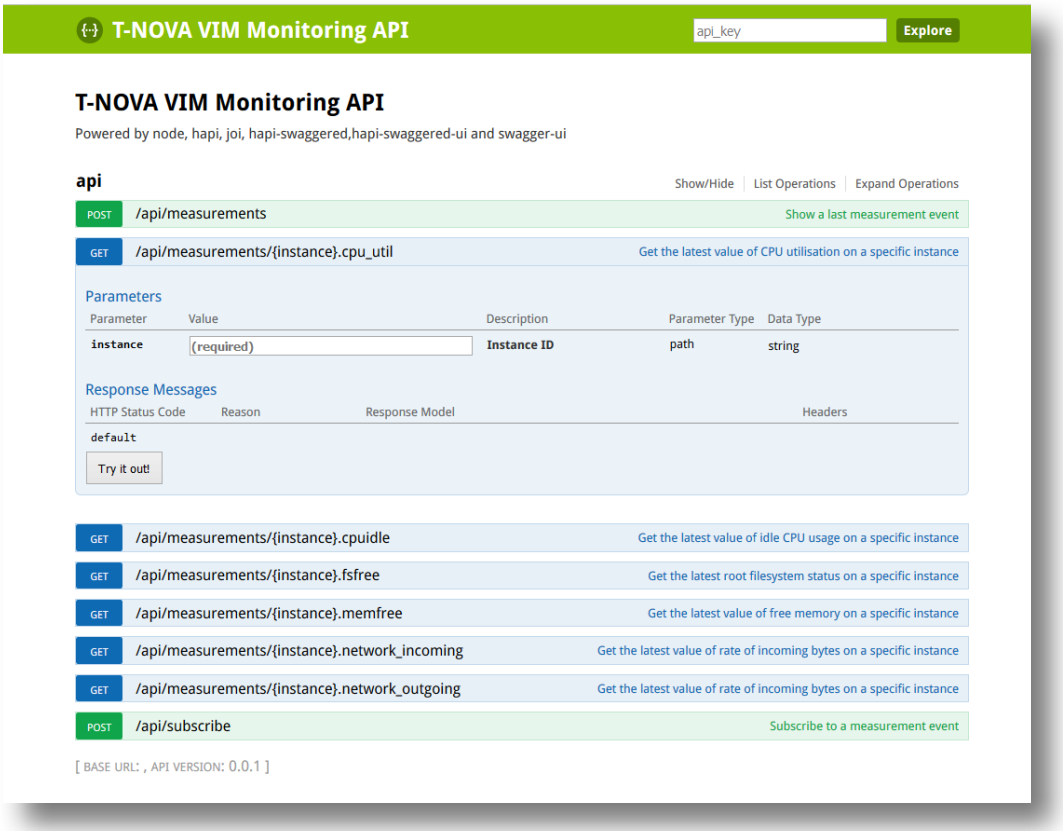

Figure 10. Live API documentation via Swagger

Also, Annex II of this document provides a complete API reference for the VIM Monitoring Manager.

4.6.4. Time-series Database

Since the T-NOVA VIM Monitoring Backend handles primarily measurements, we have selected a time-series database as optimal. For its implementation we have opted to use InfluxDB [InfluxDB], a time-series database written in Go. By concentrating all data in a performant DB and relying on periodical feeds, we can simplify workflows, reduce inter-component signaling and thus eliminate the need for a message queue, which is commonly used in monitoring frameworks.

Although, InfluxDB is a distributed database, for the time being we are evaluating it on a single node until storage issues appear.

The Backend requires that a database has already been created in InfluxDB. The use of a retention policy is also highly recommended, since the database could store potentially multiple gigabytes of measurement data every day. For the development and QA testing of the backend we use a retention policy of 30 days. After 30 days the measurements are erased, in order to free up disk and memory space.

Each meter is stored in a separate table, where multiple instances may store values of the specific measurement type. In addition to the actual value, a timestamp and an instance tag are also stored, in order to identify the measurement's origin and time. Finally, the data type of every measurement is float64.

Queries are performed in the Line protocol[12] by using the influent module. An example query is the following:

SELECT last(**value**) **FROM** measurementType **WHERE** host='instanceA'

This query retrieves the last measurement of a certain type and a certain VNF instance.

4.6.5. Anomaly detection

For detecting critical situations and producing alarms, the T-NOVA VIM MM supports operations via statically defined thresholds. The Orchestrator can use the alarming methods (see Sec. 4.6.3.3.) to define and modify alarms. In turn the VIM MM checks the affected measurements, evaluates the expressions given by the rules and sends eventually a notification if the expression is true. This is a standard feature also provided by several monitoring frameworks, as surveyed in Chap. 3.

However, alarming using single-metric thresholds is not always effective. For example, in a VNF, a high CPU usage might be a result of either normal operation (under high traffic volume) or malfunction. For this reason, it would be more appropriate to jointly consider more than one metric and use multiple rules for alarming combined with

[12] https://influxdb.com/docs/v0.9/write_protocols/write_syntax.html

logical operations (AND, OR etc.) In our case, a high CPU usage combined with a low traffic volume could be a reason for alarm.

Even in this case though, finding the right set of rules to accommodate various anomalous behaviours is not always easy and can lead to rather misleading results. Instead, it would be much more effective to observe the normal VNF behaviour under various usage scenarios and loads and dynamically (i.e. without pre-defined thresholds) identify any deviation from the "normal" state, taking into account also the cross-correlation of the various metrics, rather than observing each one individually. This is the concept of NFV anomaly detection.

The first step towards this direction is to identify a set of metrics which best illustrate the VNF status and decide, at any given time, whether the operating conditions are normal or not. Let us assume a vector of samples of various metrics, each of which can be either instantaneous or average over an arbitrary time window:

$$\underline{m} = (m_1, m_2, \dots m_n)$$

For example, m_1 can be the current CPU load, m_2 the memory utilization, m_3 a VNF-specific application metric etc. The question is whether from a given set of values we can assume whether the VNF is functioning normally or not.

At first glance, this resembles a binary classification problem, for which several approaches could be used, such as e.g. support vector machines, neural networks or logistic regression. The problem with all these methods is that they are associated with supervised learning, i.e. they need a diverse set of both positive and negative samples for training. In other words, they need a sufficiently large dataset from a VNF operating in both abnormal and normal state. While the normal state metrics can be derived after a comprehensive measurements campaign with the VNF operating under multiple usage scenarios and load profile, it is quite difficult to predict, plan and reproduce all (or even, most) abnormal situations. For this reason, a supervised learning method cannot be easily applied in our case.

Another approach would be to resort to statistical multivariate outlier detection methods, which essentially aim at identifying samples (outliers) which significantly diverge from the rest of the dataset. For the needs of T-NOVA, we choose two methods which are quite efficient and at the same time can be implemented with reasonable complexity. In particular, we consider and evaluate multivariate outlier detection based on i) multiple linear regression and ii) Mahalanobis distance. These two statistical methods are briefly overviewed in the next paragraphs, while their actual evaluation in a test case involving the virtual Traffic Classifier (vTC) is presented in Sec. 5.3.

4.6.5.1. Anomaly detection based on multiple linear regression

In order to adopt the linear regression method, it is first essential to categorise the metrics into independent and dependent ones (this classification is not essential in the Mahalanobis method, which is described in the next paragraph). In an NFV environment, we should consider as independent the metrics which are associated to

the external load imposed to the VNF (e.g. the bit rate or the number of flows of the traffic fed to the VNF), while dependent metrics would reflect the actual status of the VNF application and the associated VDU, such as e.g. CPU load, memory usage, etc.

Multiple linear regression attempts to deduce a linear dependency between one or more dependent metric and one or more independent ones. If we denote as $\underline{m_D}$ the vector of the dependent metrics and $\underline{m_I}$ the vector of independent ones, then the multiple linear regression estimator for $\underline{m_D}$ should be

$$\underline{\widehat{m_D}} = A\underline{m_I}^T + B$$

where A and B are matrices whose values can be calculated using various methods, e.g. using the least squares model.

Using this method, anomaly detection is pretty straightforward and can be achieved using the following steps:

1. The VNF is put in normal operation and a set of measurements is derived under as diverse as possible operating conditions, in order to collect the dataset.
2. The model parameters are specified by performing the linear fit on the dataset.
3. The RMSE (root mean square error) of the predictor is calculated for each individual dependent metric m_{Ii} , using the actually measured values and the predicted ones:

$$RMSE = \sqrt{\frac{\sum_n (m_{Di} - \widehat{m_{Di}})^2}{n}}$$

where n is the number of samples in the dataset.

4. An operating threshold is chosen, typically an integer multiple of RMSE. As will be shown in the evaluation section (Sec. 5.3), the choice of the threshold will need to actually achieve an optimal trade-off between precision and recall.
5. During operation phase, VNF samples are gathered periodically, and when the absolute deviation between a measured and an estimated dependent metric exceeds the operating threshold, an anomaly is indicated. In order to reduce the number of false alarms, it might be decided that an alarm is triggered only if a number of consecutive high deviations are detected.

The concept of outlier detection using linear regression is visualized in Figure 11, where, for the sake of visualization, single instead of multiple regression is used. That is, a single dependent metric (m_{Di}) is predicted from a single independent metric (m_{Ii}). The two outliers shown are detected as deviating significantly from the predicted values.

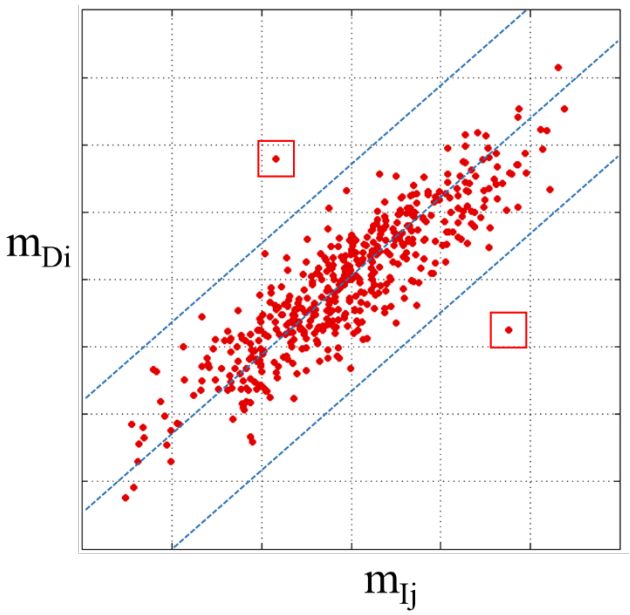

Figure 11. Anomaly detection using linear regression (univariate example)

It must be noted that these two outliers could not be detected by setting thresholds on the two metrics individually, since they are both quite close to the mean values of m_{Di} and m_{Ij}. This highlights the value of multivariate analysis –even in its simplest form- for VNF anomaly detection.

For the VIM Monitoring Manager to use this method, it is essential to know a priori for each VNF (or, more precisely, VNFC) the metrics to be used, the A and B matrices and the operating threshold.

It must be also noted that the linear regression method –as its name implies- assumes linear (or close-to-linear) dependency between the various metrics. If this is not the case, a transformation should be applied to some or all the metrics in order to approach linearity. However, as shown in the assessment campaign of Sec. 5.3., the linearity assumption is more than sufficient for most cases.

4.6.5.2. Anomaly detection based on Mahalanobis distance

Unlike the linear regression method, the Mahalanobis approach [Mahalan36] does not require a categorization of metrics into dependent and independent ones. The Mahalanobis distance considers an N-dimensional space (in our case, N is the number of different metrics) and calculates the distance of a sample from the mean of the distribution of the existing dataset.

The advantage of the Mahalanobis distance with respect to the standard Euclidean distance is that, unlike the latter, it also takes into account the covariance of the different metrics.

In specific, the Mahalanobis distance of a sample of metrics

$$\underline{m} = (m_1, m_2, \dots m_n)$$

from a dataset of metrics observation with mean

$$\underline{\mu} = (\mu_1, \mu_2, \dots \mu_n)$$

and covariance matrix S, is defined as:

$$D_M(\underline{m}) = \sqrt{(\underline{m} - \underline{\mu})^T S^{-1} (\underline{m} - \underline{\mu})}$$

The concept of using the Mahalanobis distance to detect outliers is shown in Figure 12. For the sake of visualization, we use only two metrics although the method is valid for any N number of different metrics. The ellipses shown represent points with the same Mahalanobis distance from the dataset centre. By choosing an appropriate distance threshold, we can classify as outliers samples whose distance is greater than the threshold.

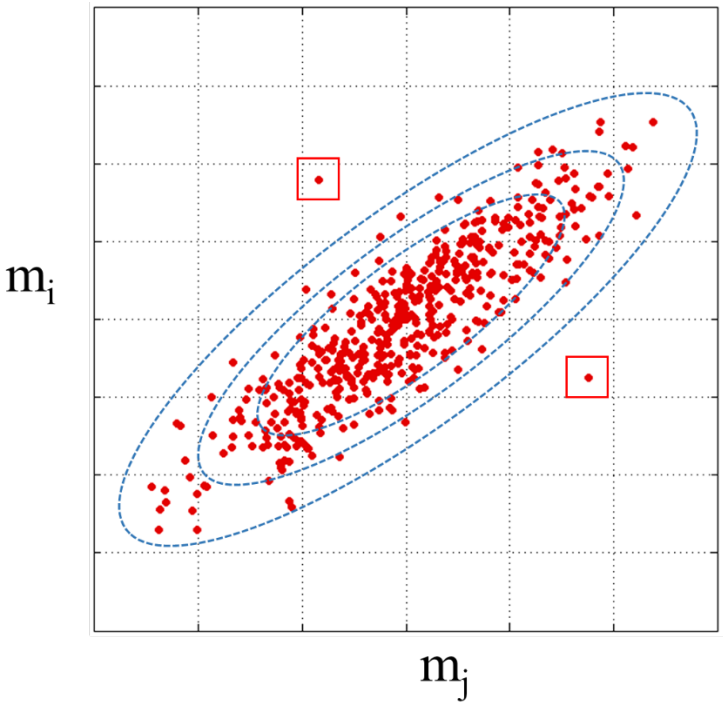

Figure 12. Anomaly detection using the Mahalanobis distance

As it can be seen, the Mahalanobis formula essentially uses the covariance matrix to build the ellipsoid that best represents the set's probability distribution. The Mahalanobis distance is simply the distance of the test point from the center of mass divided by the width of the ellipsoid in the direction of the test point.

In order to achieve anomaly detection using the Mahalanobis distance, the following steps are foreseen:

1. The VNF is put in normal operation and a set of measurements is derived under as diverse as possible operating conditions, in order to collect the dataset.
2. The mean for each individual metric is derived.
3. The covariance matrix S is calculated, as well as its inverse (S^{-1})
4. The Mahalanobis distance of all existing samples in the dataset is calculated, and the standard deviation of the distances σ_M is derived. An operating threshold is chosen, often as an integer multiple of σ_M. Again, as it will be shown in the evaluation section (Sec. 5.3), the choice of the threshold will need to actually achieve a trade-off between precision and recall.
5. During operation phase, VNF samples are gathered periodically, and when the Mahalanobis distance of a sample exceeds the operating threshold, an anomaly is indicated. In order to reduce the number of false alarms, it might be decided that an alarm is triggered only if a number of consecutive high deviations are detected.

For the VIM Monitoring Manager to use this method, it is essential to know a priori for each VNF (or, more precisely, VNFC) the metrics to be used, the covariance matrix and the operating threshold.

Similarly to the linear regression mentioned in the previous section, the application of the Mahalanobis distance also assumes linear dependence between metrics. If this is not the case, as aforementioned, a transformation should be applied.

4.6.6. Graphical user interface

The main interface of the VIM Monitoring Framework is the HTTP API of the backend, as described in Sec. 4.6.3. During the development of the monitoring framework, the VNF developers have, however, requested for a graphical way of accessing the monitoring data their VNF instances and VNF applications more specifically, produce. This lead to the integration of a Grafana server. Grafana [Grafana] is a graph builder for visualising time series metrics. It supports InfluxDB as a data source and thus, it is easy to visualise all the available measurements directly from the database.

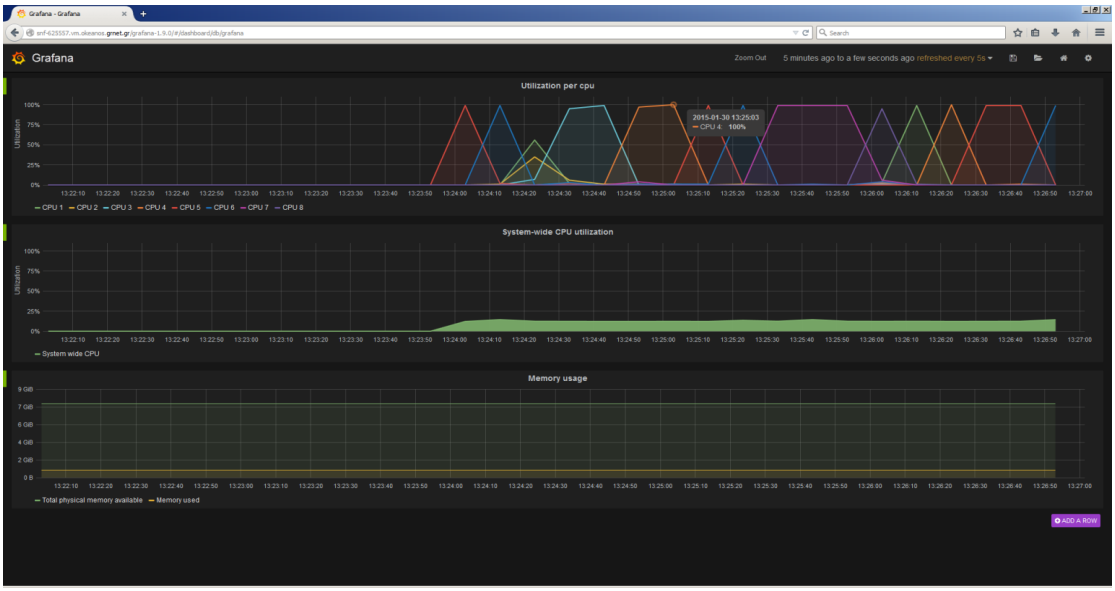

Figure 13. Visualization of measurements with Grafana

4.7. Packaging, documentation and open-source release

In an effort to contribute to maximising the impact of T-NOVA on the NFV community, the T-NOVA monitoring framework is released [GH-VIM] under the GNU General Public License v3.0[13]. Interested stakeholders and prospective contributors are welcome to download and do pull requests on the public GitHub repository[14]. The plan is to move the project to the overall T-NOVA Github account, as soon as the latter becomes available.

A Docker image containing node and the application is also provided inside this repository. A Docker image allows the seamless usage of the backend in any OpenStack deployment, regardless of deploying it on a physical or virtual machine. Most of the configuration parameters are exposed in Docker environment variables and can be set up during container creation. A YAML file (docker-compose.yml) is also provided inside the repository, so that users can combine the VIM monitoring backend, InfluxDB and Grafana in the same way the backend is being developed and tested.

For further documentation of the backend, please refer to the README file[15] and the documentation directory[16] of the repository. The information will be kept up-to-date while development progresses. For the API documentation, please refer to the /docs endpoint of a working deployment, where the Swagger-UI is hosted.

[13] https://github.com/spacehellas/tnova-vim-backend/blob/master/LICENSE.txt
[14] https://github.com/spacehellas/tnova-vim-backend
[15] https://github.com/spacehellas/tnova-vim-backend/blob/master/README.md
[16] https://github.com/spacehellas/tnova-vim/blob/master/documentation

5. VALIDATION

5.1. Functional testing

For the purpose of functional testing and benchmarking of the current release of the T-NOVA VIM monitoring framework, the latter was integrated into the T-NOVA IVM testbed as shown in Figure 14 below.

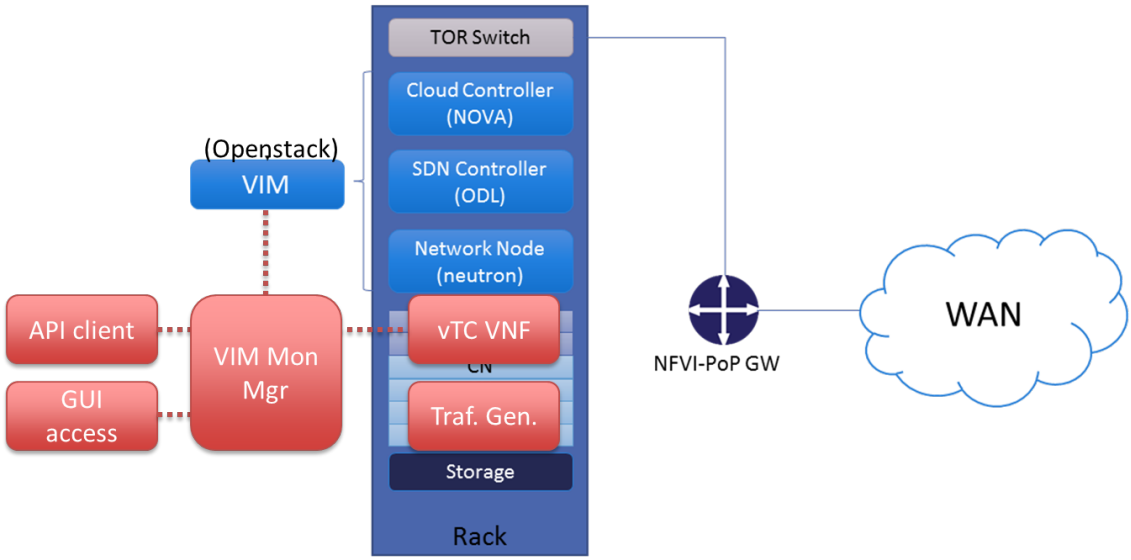

Figure 14. Testbed configuration for testing VIM Monitoring

The VIM Monitoring Manager was deployed as a Docker container in a separate physical host, as part of the VIM management and monitoring framework. It interfaced with Openstack for the collection of metrics via Ceilometer.

The workload to produce the metrics was the latest version of the vTC (virtual Traffic Classifier) VNF, deployed in a VM. The vTC used the Python library provided (see Sec. 4.4) in order to dispatch VNF metrics to the VIM MM. At the same time, the collectd agent was also installed in the VNF VM, to dispatch generic metrics.

In order to emulate realistic operational conditions, a traffic generator hosted in a separate VM was used to play back a real network traffic dump containing a mix of various services.

Behind the VIM Mon. Mgr. two clients were used; one for accessing the metrics via the REST API and a second one accessing the web-based GUI.

5.1.1. Metrics acquisition and integration test

The chosen functional test intended to validate most of the functional capabilities of the VIM MM, namely:

- Interfacing with Ceilometer
- Collection of agent metrics
- Collection of VNF metrics
- Persistence of measurements
- GUI operation

The GUI was configured by the user to display the following metrics, integrated in a single view:

- VNF CPU utilization, retrieved from Openstack
- VNF memory usage and network traffic (cumulative packet count), as reported by guest OS via the monitoring agent
- VNF-specific metrics. Specifically, the vTC is able to report the packet rate of different applications detected e.g. Skype, Bittorrent, Dropbox, Google, Viber etc.)

Figure 15. End-to-end functional test: the VIM MM GUI screenshot, integrating metrics from various sources

It was verified that the multiple samples were collected and displayed properly. The validity of the measurements was verified by accessing the console view of the VNF VM and:

- Checking the system metrics via command-line tools (top, netstat etc.)

- Checking the vTC VNF logs which contained periodic dumps of the VNF metrics (per-application rate). It is clarified that what is tested here is the ability to communicate and collect metrics and not the accuracy of the vTC.

System stability was also checked by allowing the system to run continuously; the vTC and VIM MM was found to be operating normally after more than two days of uptime and continuous operation, until it was manually stopped.

5.1.2. Northbound API tests

All methods exposed by the northbound API were tested and the reception of the proper reply was verified. In specific, the following functionalities were tested:

- Listing of available measurement types
- Getting the latest measurements
- Fetching individual generic metrics
- Fetching individual VNF-specific metrics
- Subscribing to measurements
- Listing subscriptions
- Deleting subscriptions
- Creating alarm triggers
- Listing alarm triggers
- Deleting alarm triggers

The sample responses which were derived via the tests, along with the REST calls which created them, are listed in detail in Annex II.

5.2. Benchmarking

As aforementioned, apart from the GUI, the VIM also exposes a programmatic interface (API) to the orchestration and VNFM components. We tested the scalability and performance of the VIM MM by loading it with a variable number of GET requests, asking for a single specific metric (CPU load) of the vDPI VNF. We used the httperf software [httperf] to generate synthetic HTTP GET requests at various rates and measured the rate of responses received. Then, we repeated the procedure, this time directly polling Ceilometer for the same metric.

The two sets of measurements were made on platforms with similar hardware capabilities. The results are depicted in Fig.5.

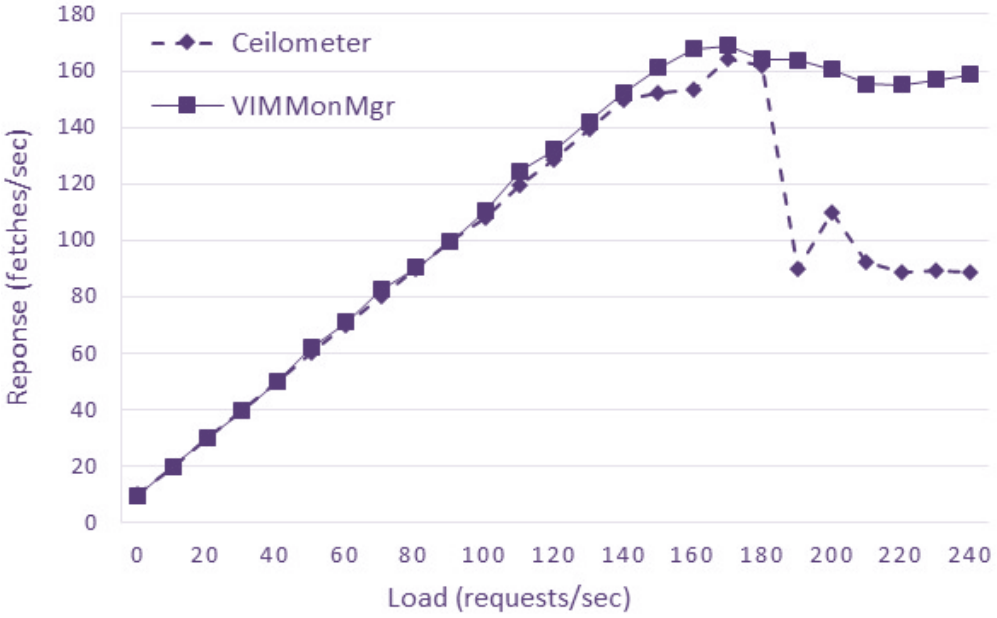

Figure 16. VIM Monitoring Manager performance

It can be shown that the VIM MM can expose metrics with performance comparable to native Ceilometer. It also seems to exhibit better stability when overloaded (at more than 160 requests/sec for the given hardware configuration).

An important added value of VIM MM is the communication overhead, which has been reduced to the minimum to improve scalability. Figure 17 compares the length (in bytes) of the responses to a single GET request for a specific metric of a VNF VM (CPU load, memory utilization and disk usage). The response of Ceilometer is quite verbose, since it also includes detailed instance information. We try to alleviate this effect by including in the response body only the absolutely necessary elements, i.e. the metric name, the value and the timestamp. The result is a decrease in overhead by about 95%.

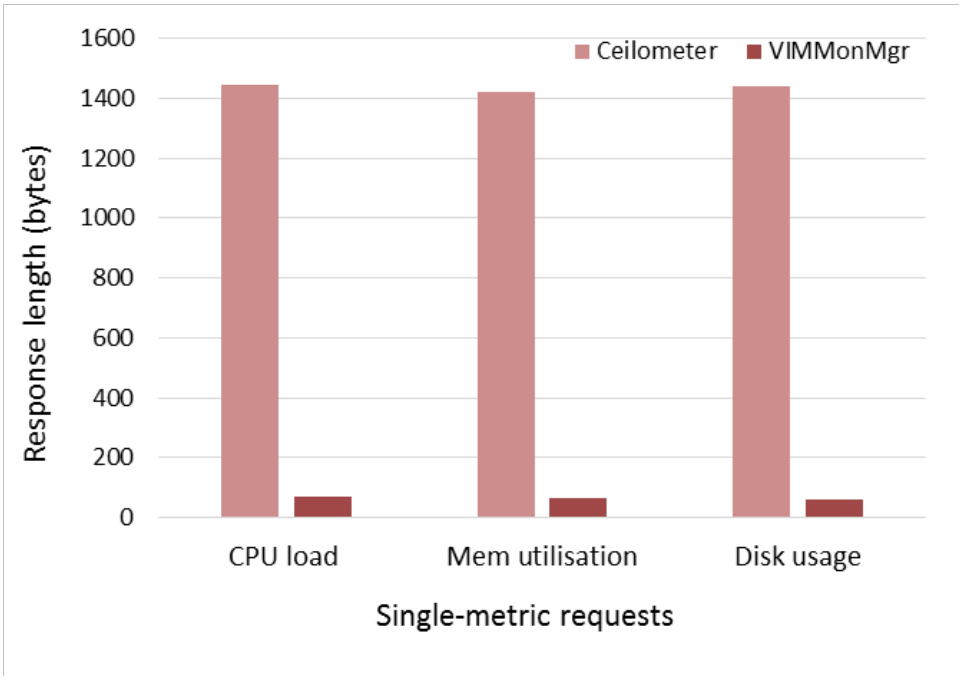

Figure 17. Length of responses for single-metric requests

It is thus seen that the VIM MM exhibits acceptable performance when it comes to communicating metrics.

Regarding Ceilometer, however, it must be noted that the performance limitations of Ceilometer are known and expected to be alleviated in the upcoming releases - and the Telemetry project is already targeting at fulfilling NFV requirements. In this context, Ceilometer could also in the near future offer an efficient solution for NFV monitoring, bringing at the same time strong community support as well as wide industrial uptake, being a core Openstack component.

5.3. Assessment of anomaly detection methods

Another step in the assessment of the monitoring system is the evaluation of the anomaly detection mechanism using the two methods described in Sec. 4.6.5. For the tests, we used a deployed instance of the T-NOVA virtual Traffic Classifier (vTC) VNF. The VNF was deployed in a "m1.small" flavor, with 2 CPU cores, 4GB of RAM and 40GB of HDD.

To feed the vTC, we used an actual traffic dump containing a mix of various services, which was played back (using tcpreplay) at various speeds, ranging from 0 Mbps (no traffic) to 600 Mbps (maximum) at steps of 60 Mbps. We gathered a set of metrics originating from the VNF monitoring agent, from Openstack Ceilometer and from the VNF application itself. In total, 122 samples (metrics vectors) were gathered and form the dataset which corresponds to the VNF normal operation and which was used to "train" the anomaly detection algorithms.

The next step is to identify these metrics which can be actually used for the anomaly detection process. We visually examined, via graphs but also by calculating the

covariance, the fluctuation of various metrics as a response to the various traffic loads as well as their cross-correlation. As examples, the following figures depict the variation of CPU (system + user) and memory usage respectively (dependent variables) corresponding to the traffic bitrate (independent variable).

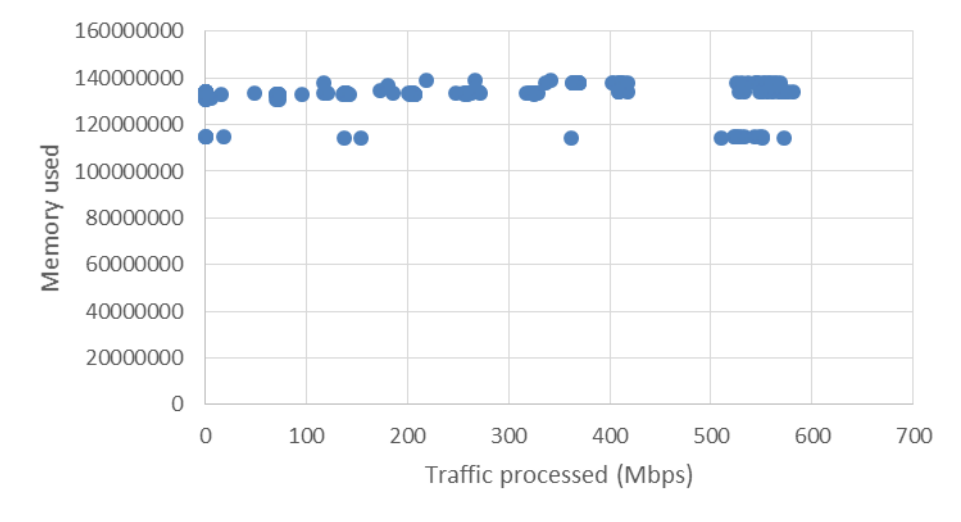

Figure 18. Memory used vs. traffic bitrate for the vTC VNF

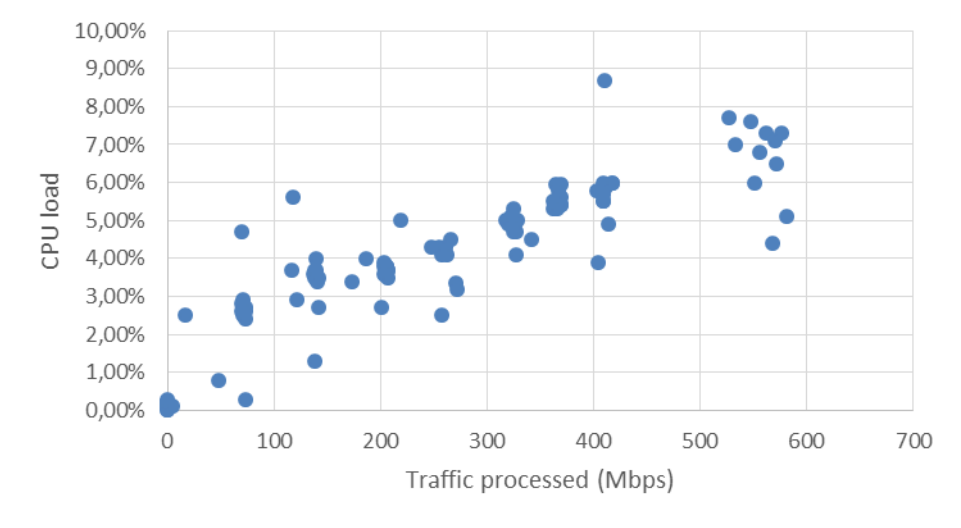

Figure 19. CPU usage vs. traffic bitrate for the vTC VNF

From Figure 18 it can be deduced that memory utilization does not actually fluctuate with workload, so it might be more appropriate to be subject to individual, threshold-based alarming rather than multivariate analysis.

On the other hand, as shown in Figure 19, CPU increases proportionally –almost linearly- with traffic bitrate (which is expected of course), so it would make sense to exploit the correlation between these two metrics for anomaly detection. In the following sections, we use and evaluate the two statistical methods introduced (linear regression and Mahalanobis distance). For the sake of simplicity, we restrict the analysis to these two metrics only, which anyway present by far the most value for our analysis

compared to the rest. However, the methods can be applied as is to an arbitrary number of metrics.

For the evaluation of each method, we use three quality measures which are commonly used to assess detection methods:

Precision is defined as the ratio of true positives (alarms which actually corresponded to an anomaly incident) over all reported positives (i.e. all alarms, including true and false positives).

Recall is defined as the ratio of true positives over all actual positives (i.e. all anomaly incidents, including the ones which were not detected).

Finally, **F1-score** is a single number which combines precision and recall to produce a single merit score for the detector. It is defined as:

$$F1 = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

5.3.1. Linear regression

The usage of multiple linear regression for VNF anomaly detection was detailed in Sec. 4.6.5.1. and is applied here to the vTC case.

For the evaluation, we consider the CPU usage (%) as dependent variable and the processed traffic bitrate (Mbps) as independent.

By processing the normal operation dataset and using linear fitting, a predictor for the CPU usage from the traffic bitrate (BW) can be derived as follows:

$$CPU = 0.0107 \cdot BW + 1.3$$

(CPU usage: percent, BW: Mbps)

Using this predictor, the Root Mean Squared Error (RMSE) over the normal dataset was 0.921879. These calculated values (linear model coefficients and RMSE) are the ones to feed into the model for anomaly detection.

To assess the method, we need to drive the vTC into an abnormal state. While, as also mentioned, it is hard to predict and reproduce all anomalies, we used two specific scenarios:

1. VNF application fault; the vTC application was manually suspended, although the VNF still forwarded the traffic.
2. Process malfunction; a dummy process in parallel with the vTC application was initiated to consume extra CPU resources, emulating either a malicious intervention or a malfunctioning application.

In addition to the 122 samples corresponding to the normal operation, we also gathered 14 samples of each of the anomalous scenarios and used the linear regression method to detect the abnormality. We used three different detection thresholds, corresponding to the RMSE multiplied by 2, 4 and 6. The results shown in the table below refer to all 150 samples overall (normal operation & 2 anomaly scenarios).

Table 11. Performance of the linear regression-based detector

Detection threshold	2RMSE	4RMSE	6RMSE
Precision	84.4%	100.0%	100.0%
Recall	100.0%	92.6%	88.9%
F1-score	0.915254	0.961538	0.941176

It can be deduced that a selection of 4RMSE as detection threshold yields a satisfactory tradeoff between precision and recall. The threshold can be further adjusted according to the requirements of the NFV service (e.g. minimization of either missed incidents or false alarms) as well as the exact types of anomalies which need to be tracked.

5.3.2. Mahalanobis distance

The usage of the Mahalanobis distance for VNF anomaly detection was detailed in Sec. 4.6.5.2. and is applied here to the vTC case.

For the evaluation, we consider the same pair of metrics: CPU usage (%) and processed traffic bitrate (Mbps). Unlike the previous method, there is no need to identify metrics as dependent or independent.

By processing the normal operation dataset, we calculate in turn:

- the mean of each metric
- the covariance matrix and its inverse
- the Mahalanobis distance of each sample of the normal dataset
- the standard deviation (σ) of the distances

The metric means, inverse of covariance matrix and standard deviation σ are the parameters to feed into the detector.

To assess the method, we use the same induced anomalies as described in the previous section. Overall we have again 150 samples corresponding to normal and abnormal operation. As detection thresholds, we use the standard deviation σ of the normal dataset multiplied by 2, 4 and 6. The results are shown in the table below.

Table 12. Performance of the Mahalanobis distance-based detector

Detection threshold	2σ	4σ	6σ

Precision	40.9%	87.1%	100.0%
Recall	100.0%	100.0%	96.3%
F1-score	0.580645	0.931034	0.981132

It can be seen that, using 6σ as threshold, the detector presents satisfactory performance.

Overall, via the evaluation of the two methods it can be deduced that, even under this limited assessment in two specific scenarios, they can be quite useful in detecting outliers in VNF operation and thus quickly identifying anomalies. They both exhibit similar performance and also introduce little computational overhead, suitable for real-time operation. Crucial aspects for the success of the method are: the proper selection of metrics to be used, an exhaustive measurement campaign for the normal operation dataset under diverse conditions and also a careful selection of the detection threshold.

5.4. Fulfillment of requirements

Following the successful execution of the aforementioned validation and assessment tests, the table below explains how the implemented and tested VIM monitoring framework eventually fulfills (or is planned to fulfill) the requirements which were set in Section 2.

Table 13. Compliance to requirements

Requirement for the Monitoring Framework	Status	Justification
The MF must provide a vendor agnostic mechanism for physical resource monitoring.	Compliance	The mechanisms introduced for measurements collection are vendor agnostic for most metrics (CPU, memory, storage, network etc.)
The MF must provide an interface to the Orchestrator for the communication of monitoring metrics.	Compliance	The MF exposes a northbound REST API for metrics/alarms communication in either push or pull mode.
The MF must re-use resource identifiers when linking metrics to resources.	Compliance	The VIM MM re-uses the Openstack UUIDs and hostnames for linking metrics to resources.
The MF must monitor in real time the physical network infrastructure as well as the vNets instantiated on top of it.	Compliance	Network-related measurements are derived from the monitoring agents (for network interfaces) and also OpenDaylight (for virtual links and networks)
The MF must provide an API for communicating metrics (in either push or pull mode)	Compliance	The MF exposes a northbound REST API for metrics/alarms communication in either push or pull mode.

The MF must collect utilisation metrics from the virtualised resources in the NFVI.	Compliance	The VIM MM collects metrics from deployed VMs and established vNets.
The MF must collect compute domain metrics.	Compliance	Direct access to compute domain metrics is achieved by means of the collect monitoring agent.
The MF must collect hardware accelerator metrics	Compliance	Hardware accelerator metrics are accessible by means of specific agent (collectd) plugins.
The MF must collect compute metrics from the Hypervisor.	Compliance	The VIM MM collects hypervisor metrics indirectly via the Ceilometer API.
The MF must collect network domain metrics from the Hypervisor.	Compliance	The VIM MM collects hypervisor metrics indirectly via the Ceilometer API.
The MF must process and dispatch alarms.	Compliance	The VIM MM allows configuring and subscribing to alarms.
The MF must collect metrics from physical and virtual networking devices.	Compliance	The VIM MM interfaces with OpenDaylight to collect network device statistics.
The MF must leverage SDN monitoring capabilities.	Compliance	The VIM MM collects (mostly port) statistics for SDN devices from OpenDaylight.

6. CONCLUSIONS

This document described the design and development of a monitoring framework for the T-NOVA IVM layer. Using a comprehensive state-of-the-art survey as well as a consolidation of T-NOVA requirements, the architecture of the T-NOVA VIM monitoring framework was specified. Taking into account the use of OpenDaylight and OpenStack as the controller technologies in the VIM, infrastructure metrics and statistics available from these controllers are collected. Furthermore, a VNF monitoring agent was also introduced, as an optional component, collecting a rich set of metrics from within VMs and VNF applications. All these metrics are aggregated and filtered into a centralised Monitoring Manager, which exposes status and resource information of the NFVI-PoP to the Orchestrator, as configured by the latter.

It is concluded that, with the proposed approach, the goal of delivering an effective, efficient and scalable monitoring solution for the T-NOVA IVM layer is achieved. The developed solution is able to expose to the Orchestrator and to the Marketplace enhanced awareness of the IVM status and resources, while at the same time keeping the communication and signalling overhead at minimum.

The current release has been integrated with the T-NOVA IVM testbed and has been demonstrated in operation in IEEE IM 2015 and IEEE SDN/NFV conferences as part of an integrated demonstrator of the T-NOVA project, monitoring an NFV service with the vTC VNF.

7. REFERENCES

[Aodh] Openstack Telemetry alarming, https://github.com/openstack/Aodh

[Cloudwatch] Amazon CloudWatch, http://aws.amazon.com/cloudwatch

[Collectd] collectd – The system statistics collection daemon, https://collectd.org/

[Cyclops] Cyclops framework, http://icclab.github.io/cyclops/

[D232] M. McGrath (Ed.) et al, "Specification of the Infrastructure Virtualisation, Management and Orchestration – Final", T-NOVA Deliverable D2.32, October 2015

[D532] "Network Functions Implementation and Testing – Final", T-NOVA Deliverable D5.32, June 2016

[DCM] Ye Yu, C. Qian, and X. Li, "Distributed and Collaborative Traffic Monitoring in Software Defined Networks," presented at the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'14), Chicago, IL, USA, 2014.

[Doctor] OPNFV Wiki - Project: Fault Management (Doctor), https://wiki.opnfv.org/doctor

[DoctorDel] Doctor Deliverable: Fault Management and Maintenance, Release 1.0.0, October 2015, http://artifacts.opnfv.org/doctor/DoctorFaultManagementandMaintenance.pdf

[Drools] Drools BPM engine, http://www.drools.org/

[Flowsense] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. Madhyastha, "FlowSense: Monitoring Network Utilization with Zero Measurement Cost," in Passive and Active Measurement. vol. 7799, M. Roughan and R. Chang, Eds., ed: Springer Berlin Heidelberg, 2013, pp. 31-41.

[Ganglia] Ganglia Monitoring System, http://ganglia.sourceforge.net

[GH-VIM] https://github.com/spacehellas/tnova-vim-backend

[Gnocchi] Openstack Gnocchi project, https://wiki.openstack.org/wiki/Gnocchi

[Grafana] Grafana: An open source, feature rich metrics dashboard and graph editor for Graphite, InfluxDB & OpenTSDB, http://grafana.org/

[Graphite] Graphite: A Highly Scalable Real-time Graphing System, https://github.com/graphite-project/graphite-web

[Hodge04] V. Hodge and J. Austin, "A Survey of Outlier Detection Methodologies", Artificial Intelligence Review 22 (2004), pp. 85-126

[httperf] https://github.com/httperf/httperf

[Icinga] ICINGA., https://www.icinga.org/

[InfluxDB] InfluxDB: An open-source, distributed, time series database with no external dependencies, http://influxdb.com/

[Mahalan36] Mahalanobis, Prasanta Chandra (1936). "On the generalised distance in statistics". Proceedings of the National Institute of Sciences of India 2 (1): 49–55.

[Monalisa] MONitoring Agents using a Large Integrated Services Architecture, http://monalisa.caltech.edu/monalisa.htm

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

[Nagios] Nagios Is The Industry Standard In IT Infrastructure Monitoring, http://www.nagios.org/

[NFVIFA005] Network Functions Virtualisation (NFV); Management and Orchestration; Or-Vi reference point – Interface and Information Model Specification, work in progress, November 2015

[nodejs] https://nodejs.org/en/

[NVFINF010] ETSI GS NFV-INF 010 V1.1.1 (2014-12), Network Functions Virtualisation (NFV); Service Quality Metrics

[OpenNetMon] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow Software-Defined Networks," in Network Operations and Management Symposium (NOMS), 2014 IEEE, 2014, pp. 1-8.

[Payless] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "PayLess: A low cost network monitoring framework for Software Defined Networks," in Network Operations and Management Symposium (NOMS), 2014 IEEE, 2014, pp. 1-9.

[Prediction] OPNFV Wiki – Data Collection for Failure Prediction, https://wiki.opnfv.org/prediction

[Pred-del] OPNFV Prediction Project, Release draft, February 2016, http://artifacts.opnfv.org/prediction/brahmaputra/prediction_docs/prediction_docs.pdf

[SeaLion] Sealion; Quickly Diagnose Problems with you Linux Servers, https://sealion.com/

[Shinken] Shinken, http://www.shinken-monitoring.org/

[Stacktach] Stacktach, Event-based Monitoring & Billing solution for OpenStack, https://github.com/rackerlabs/stacktach

[Statsd] StatsD; Simple daemon for easy stats aggregation, https://github.com/etsy/statsd/

[Telemetry] Openstack Telemetry, https://wiki.openstack.org/wiki/Telemetry

[vSphere] vmware vSphere, http://www.vmware.com/products/vsphere

[Zabbix] ZABBIX, The Enterprise-class Monitoring Solution for Everyone, http://www.zabbix.com/

[Zenoss] Zenoss User Community, http://www.zenoss.org/

8. LIST OF ACRONYMS

Acronym	Explanation
API	Application Programming Interface
CPU	Central Processing Unit
DPDK	Data Packet Development Kit
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HW	Hardware
KPI	Key Performance Indicator
NFV	Network Functions Virtualisation
NFVI	NFV Infrastructure
NFVI PoP	NFVI Point-of-Presence
NFVO	NFV Orchestrator
OID	Object Identifier
OPNFV	Open Platform for NFV
OS	Operating System
REST	Representational State Transfer
SNMP	Simple Network Management Protocol
SoC	System-on-Chip
VDU	Virtual Deployment Unit
VIM	Virtualised Infrastructure Manager
VIM MM	VIM Monitoring Manager
VM	Virtual Machine
VNF	Virtual Network Function
VNFD	VNF Descriptor
VNFM	VNF Manager
VNFP	VNF Provider
vTC	Virtual Traffic Classifier
YANG	Yet Another Next Generation

9. ANNEX I: SURVEY OF RELEVANT IT/NETWORK MONITORING TOOLS

This section presents a brief overview of existing frameworks for monitoring virtualized IT infrastructures as well as SDN-enabled networks, and discusses technologies which could be partially re-used in T-NOVA.

9.1.1. IT/Cloud monitoring

9.1.1.1. Shinken

Shinken is an open source system and network monitoring application [Shinken]. It is fully compatible with Nagios plugins. It started as a proof of concept for a new Nagios architecture, but since the proposal was turned down by the Nagios authors, Shinken became an independent tool. It is not a fork of Nagios; it is a total rewrite in Python. It watches hosts and services, gathers performance data and alerts users when error conditions occur and again when the conditions clear. Shinken's architecture is focused on offering easier load balancing and high availability capabilities. The main differences and advantages toward Nagios are:

- A more efficient distributed monitoring and high availability architecture
- Graphite integration in the Web UI
- Better performance, mostly due to the use of a distributed database (MongoDB)

9.1.1.2. Icinga

Icinga is an open-source network and system monitoring application which was born out of a Nagios fork [Icinga]. It maintains configuration and plug-in compatibility with the latter. Its new features are as follows:

- A modern Web 2.0 style user interface;
- An interface for mobile devices;
- Additional database connectors (for MySQL, Oracle, and PostgreSQL);
- RESTful API.

Currently there are two flavours of Icinga that are maintained by two different development branches: Icinga 1 (the original Nagios fork) and Icinga 2 (where the core framework is being replacement by a full rewrite).

9.1.1.3. Zenoss

Zenoss is an open source monitoring platform released under the GPLv2 license [Zenoss] It provides an easy-to-use Web UI to monitor performance, events, configuration, and inventory. Zenoss is one of the best options for unified monitoring as it is cloud-agnostic and is open source. Zenoss provides powerful plug-ins named Zenpacks, which support monitoring on hypervisors (ESX, KVM, Xen and HyperV),

private cloud platforms (CloudStack, OpenStack and vCloud/vSphere), and public cloud (AWS). In OpenStack Zenoss integrates with Nova, Keystone and OpenStack Telemetry.

9.1.1.4. Ganglia

Ganglia is a scalable distributed system monitor tool for high-performance computing systems such as clusters and grids [Ganglia]. Its structure is based on a hierarchical design using a tree of point-to-point connections among cluster nodes. Ganglia is based on an XML data representation, XDR for compact and RRDtool for data storage and virtualisation. The Ganglia system contains:

1. Two unique daemons, gmond and gmetad
2. A PHP-based web front-end
3. Other small programs

gmond runs on each node to monitor changes in the host state, to announce applicable changes, to listen to the state of all Ganglia nodes via a unicast or multicast channel based on installation, and to respond to requests. gmetad (Ganglia Meta Daemon) polls at regular intervals a collection of data sources, parses the XML and saves all metrics to round-robin databases. Aggregated XML can then be exported.

The Ganglia web frontend is written in PHP. It uses graphs generated by gmetad and provides the collected information like CPU utilisation for the past day, week, month, or year. Ganglia has been used to link clusters across university campuses and around the world and can scale to handle clusters with 2000 nodes. However, further work is required in order for it to become more cloud-agnostic.

9.1.1.5. StackTach

StackTach is a debugging and monitoring utility for OpenStack that can work with multiple Data Centres, including multi-cell deployment [Stacktach]. It was initially created as a browser-based debugging tool for OpenStack Nova. Since that time, StackTach has evolved into a tool that can perform debugging, monitoring and auditing. StackTach is quickly moving into Metrics, SLA and Monitoring territory with version 2 and the inclusion of Stacky, the command line interface to StackTach. StackTach contains a worker that reads notifications from the OpenStack's RabbitMQ queues and stores them in a database. From there, StackTach reviews the stream of notifications to glean usage information and assemble it in an easy-to-query fashion. Users can inquire on instances, requests, servers, etc. using the browser interface or the Stacky command line tool. Rackspace is working on StackTach integration with Telemetry.

9.1.1.6. SeaLion

SeaLion is a cloud-based system monitoring tool for Linux servers. It installs an agent in the system, which can be run as an unprivileged user [SeaLion]. The agent collects data at regular intervals across servers and this data will be available on your workspace. Sealion provides a high-level view (graphical overview) of Linux server

activity. The monitoring data are transmitted over SSL to the SeaLion servers. The service provides graphs, charts and access to the raw gathered data.

9.1.1.7. MonALISA

MONitoring Agents using a Large Integrated Services Architecture (MonaLISA) is based on Dynamic Distributed Service Architecture and is able to provide complete monitoring, control and global optimisation services for complex systems[Monalisa]. The MonALISA system is designed as a collection of autonomous multi-threaded, self-describing agent-based subsystems which are registered as dynamic services, and are able to collaborate and cooperate in performing a wide range of information gathering and processing tasks.

The agents can analyse and process the information in a distributed way, in order to provide optimisation decisions in large-scale distributed applications. The scalability of the system derives from the use of a multithreaded execution engine, that hosts a variety of loosely coupled self-describing dynamic services or agents, and the ability of each service to register itself and then to be discovered and used by any other services, or clients that require such information. The system is designed to easily integrate existing monitoring tools and procedures and to provide this information in a dynamic, customised, self-describing way to any other services or clients.

By using MonALISA the administrator is able to monitor all aspects of complex systems, including:

- System information for computer nodes and clusters;
- Network information (traffic, flows, connectivity, topology) for WAN and LAN;
- Monitoring the performance of applications, jobs or services; and
- End-user systems and end-to-end performance measurements.

9.1.1.8. collectd, StatsD and Graphite

Cloud instances may also be monitored by using a collection of separate open source tools. collectd is a daemon which collects system performance statistics periodically and provides mechanisms to store the values in a variety of ways [Collectd]. collectd gathers statistics about the system it is running on and stores this information. These statistics can then be used to find current performance bottlenecks (i.e. performance analysis) and predict future system load (i.e., capacity planning). collectd is written in C for performance and portability, allowing it to run on systems without scripting language or cron daemon, such as embedded systems. At the same time it includes optimisations and features to handle big amounts of data sets. StatsD [Statsd] is a Node.JS daemon that listens for messages on a UDP to TCP port. StatsD listens for statistics, like counters and timers and then parses the messages, extracts metrics data, and periodically flushes the data to other services in order to build graphs. A tool that can be used to build graphs afterwards is Graphite [Graphite], which is able to store numeric time-series data and render graphs of the data on demand.

9.1.1.9. vSphere

The vSphere statistics subsystem collects data on the resource usage of inventory objects [vSphere]. Data on a wide range of metrics is collected at frequent intervals, processed and archived in a database. Statistics regarding the network utilisation are collected at Cluster, Host and Virtual Machine levels. In addition vSphere supports performance monitoring of guest operating systems, gathering statistics regarding network utilisation among others.

9.1.1.10. Amazon CloudWatch

Amazon CloudWatch is a monitoring service for AWS cloud resources and the applications running on AWS [Cloudwatch]. It provides real-time monitoring to Amazon's EC2 customers on their resource utilisation such as CPU, disk and network. However, CloudWatch does not provide any memory, disk space, or load average metrics without running additional software on the instance. It was primarily designed for use with Amazon Elastic Load Balancing and Auto Scaling with load balancing in mind: the service checks CPU usage on multiple instances and automatically creates additional ones when the load increases.

9.1.2. Network Monitoring

Network monitoring is a domain that has attracted significant attention from the research community over the past decades, with well-established technologies and standards with regard to measurement processes (active and passive) as well as the communication of monitoring metrics (SNMP, IPFIX, sFlow etc.).

In the context of T-NOVA, where network management, at least within each NFVI-PoP is based on OpenFlow, the measurement process will leverage OpenFlow's monitoring capabilities.

OpenFlow provides the capability to report per-flow and per-port metrics, reported by the switch itself. These metrics are then collected by the Controller and communicated to SDN control applications via the northbound API of the Controller it-self (Figure 20). Almost all SDN controllers offer the capability to expose monitoring metrics, either via API calls or language bindings. In this respect, the OpenFlow-based architecture provides the capability to monitor all network elements in a uniform and vendor-agnostic manner.

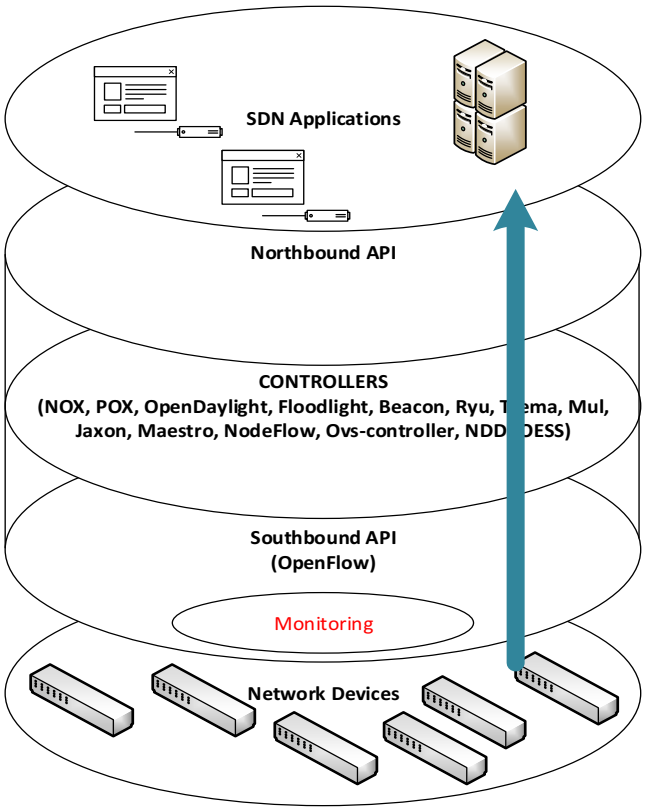

Figure 20. Communication of monitoring metrics in an OpenFlow-enabled architecture

In this context, several monitoring applications have been developed, leveraging OpenFlow capabilities for integrated network management tasks. Some of these applications are overviewed in the table below.

Table 14. OpenFlow monitoring applications

Monitoring Application	Brief description	Control-ler Used	Open Source	Available at
OpenNetMon	OpenNetMon [OpenNetMon] continuously monitors all flows between predefined link destination pairs on throughput, packet loss and delay	POX	Yes	https://github.com/TU DelftNAS/SDN-OpenNetMon/
Payless	Payless [Payless] provides a flexible RESTful API for flow statistics collection at different aggregation levels. It uses an adaptive statistics collection algorithm that delivers highly accurate information in real-time without incurring significant network overhead.	POX, NOX, OpenDay Light	Yes	http://github.com/srcv irus/floodlight.

DCM	DCM [DCM] allows switches to collaboratively achieve flow-monitoring tasks and balance measurement load.	None (native OF)	No	Not available
FlowSense	FlowSense [Flowsense] achieves a push-based approach to performance monitoring in flow-based networks, where the network informs of performance changes, rather than query it.	None (native OF)	No	Not available

In addition, many of the monitoring frameworks mentioned in Section 9.1.1 for cloud infrastructures can be also used for monitoring OpenFlow infrastructures, via the appropriate plugins.

10. Annex II: VIM Monitoring Manager API Reference

The VIM Monitoring Back-End offers a read-only API mostly on measurement events. Additionally, support for subscription is included, both for regular measurement events and alarm triggers.

For accessing the latest, up-to-date API documentation, it is always advisable to consult the Swagger UI which is available on every back-end deployment at the /docs endpoint of the HTTP service.

To access the REST-based API, one may also use CLI tools, such as curl or wget. In the following sections curl is going to be used as an example command and the example response will be shown.

10.1. Measurement-related methods

10.1.1. List available metrics

```
GET /measurementTypes
```

Example:

```
curl -X GET --header 'Accept: application/json'
'http://monitoring_backend_url/api/measurementTypes'
```

Example response (truncated):

```
[
  {
    "endPoint": "cpuidle",
    "description": "Get the latest value of idle CPU usage on a
specific instance"
  },
  {
    "endPoint": "cpu_util",
    "description": "Get the latest value of CPU utilisation on a
specific instance"
  },
  {
    "endPoint": "fsfree",
    "description": "Get the latest root filesystem status on a specific
instance"
  },
  {
    "endPoint": "memfree",
    "description": "Get the latest value of free memory on a specific
instance"
  },
  {
    "endPoint": "network_incoming",
    "description": "Get the latest value of rate of incoming bytes on
a specific instance"
```

```
  },
…
```

10.1.2. Batch retrieval of latest measurements

```
POST /measurements
```

This endpoint serves as a quick way to show multiple measurements from a variable number of VNFs. The expected response is identical to the response coming from a subscription event. It uses the following parameters:

```
{
  "types": [
    "string"
  ],
  "instances": [
    "string"
  ]
}
```

where *types* are the measurement types as appear in the previous endpoint and *instances* are the OpenStack UUIDs of the target VNFs.

Example:

```
curl -X POST --header 'Content-Type: application/json' --header
'Accept: application/json' -d '{
  "types": [
    "cpu_util", "hits"
  ],
  "instances": [
    "faeb3e00-12e2-470b-85cf-55f89a30bffa"
  ]
}' 'http://monitoring_backend_url/api/measurements'
```

Example response:

```
[
  {
    "instance": "faeb3e00-12e2-470b-85cf-55f89a30bffa",
    "measurements": [
      {
        "timestamp": "2016-03-03T12:10:13.325115753Z",
        "value": 0.53332637964622,
        "units": "percentage",
        "type": "cpu_util"
      },
      {
        "timestamp": "2016-03-15T13:54:03.399055022Z",
        "value": 0,
```

```
        "units": "percentage",
        "type": "hits"
      }
    ]
  }
]
```

10.1.3. Retrieval of individual measurements

Generic measurements contain measurements that every running VNF should report back to the monitoring back-end.

Example 1: Get the latest value of CPU utilisation on a specific instance

```
GET /measurements/{instance}.cpu_util
```

Example:

```
curl -X GET --header 'Accept: application/json'
'http://monitoring_backend_url/api/measurements/faeb3e00-12e2-470b-
85cf-55f89a30bffa.cpu_util'
```

Example response:

```
{
  "timestamp": "2016-03-03T12:10:13.325115753Z",
  "value": 0.53332637964622,
  "units": "percentage"
}
```

Example 2: Get the latest root filesystem status on a specific instance

```
GET /measurements/{instance}.fsfree
```

Example:

```
curl -X GET --header 'Accept: application/json'
'http://monitoring_backend_url/api/measurements/faeb3e00-12e2-470b-
85cf-55f89a30bffa.fsfree'
```

Example response:

```
{
  "timestamp": "2016-02-12T11:15:45.729599Z",
  "value": "17.2",
  "unit": "GB"
}
```

10.2. Subscriptions

10.2.1. List all the active subscriptions

```
GET /subscriptions
```

Example:

```
curl -X GET --header 'Accept: application/json'
'http://monitoring_backend_url/api/subscriptions'
```

Example response:

```
[
  {
    "id": "_pvr14ogpn",
    "instances": [
      "faeb3e00-12e2-470b-85cf-55f89a30bffa"
    ],
    "measurementTypes": [
      "cpu_util",
      "hits"
    ],
    "interval": 5,
    "callbackUrl": "http://callback.url"
  }
]
```

10.2.2. Subscribe to a measurement event

```
POST /subscriptions
```

If one wants to subscribe to a measurement event, then he/she has to use the following parameters:

```
{
  "types": [
    "string"
  ],
  "instances": [
    "string"
  ],
  "interval": 0,
  "callbackUrl": "string"
}
```

where *types* and *instances* are known from previous examples, while *interval* is the interval between sending new measurements in minutes and *callbackUrl* the callback URL where the subscription service is supposed to send the measurements.

Example:

```
curl -X POST --header 'Content-Type: application/json' --header
'Accept: text/html' -d '{
  "types": [
    "cpu_util", "hits"
  ],
  "instances": [
    "faeb3e00-12e2-470b-85cf-55f89a30bffa"
  ],
  "interval": 5,
  "callbackUrl": "http://callback.url"
}' 'http://monitoring_backend_url/api/subscriptions'
```

Example response:

```
Your subscription request has been registered successfully under ID
_pvr14ogpn
```

10.2.3. Delete a specific subscription

```
DELETE /subscriptions/{id}
```

Example:

```
curl -X DELETE --header 'Accept: text/html'
'http://monitoring_backend_url/api/subscriptions/_pvr14ogpn'
```

Example response:

```
Subscription was deleted
```

10.2.4. Get a specific subscription's details

```
GET /subscriptions/{id}
```

Example:

```
curl -X GET --header 'Accept: application/json'
'http://monitoring_backend_url/api/subscriptions/_pvr14ogpn'
```

Example response:

```
{
  "instances": [
    "faeb3e00-12e2-470b-85cf-55f89a30bffa"
  ],
  "measurementTypes": [
    "cpu_util",
    "hits"
  ],
  "interval": 5,
  "callbackUrl": "http://callback.url"
}
```

10.3. Alarm management

10.3.1. List all the active alarm triggers

```
GET /alarms
```

Example:

```
curl -X GET --header 'Accept: application/json'
'http://monitoring_backend_url/api/alarms'
```

Example response:

```
[
  {
    "id": "_xvazxyrfc",
    "triggers": [
      {
        "type": "memfree",
        "comparisonOperator": "gt",
        "threshold": 8018198432
      }
    ],
    "instances": [
      "instance1",
      "instance2"
    ],
    "callbackUrl": "http://callback_url1"
  },
  {
    "id": "_23zy3xgdg",
    "triggers": [
      {
        "type": "cpu_util",
        "comparisonOperator": "gt",
        "threshold": 0.80
      }
    ],
    "instances": [
      "instance1",
      "instance2"
    ],
    "callbackUrl": "http://callback_url2"
  }
]
```

10.3.2. Create an alarm trigger

```
POST /alarms
```

If one wants to create an alarm trigger, then he/she has to use the following parameters:

```
{
  "triggers": [
    {
      "type": "string",
      "comparisonOperator": "string",
      "threshold": 0
    }
  ],
  "instances": [
    "string"
  ],
  "callbackUrl": "string"
}
```

where *type*, *instances* and *callbackUrl* are known from previous examples, *comparisonOperator* is "lt" for "less", "le" for "less or equal, "eq" for "equal", "ne" for "not equal", "ge" for "greater or equal" and "gt" for "greater" and *threshold* is the value to be compared with.

Example:

```
curl -X POST --header 'Content-Type: application/json' --header
'Accept: application/json' -d '{
  "triggers": [
    {
      "type": "memfree",
      "comparisonOperator": "gt",
      "threshold": 8018198432
    }
  ],
  "instances": [
    "instance1", "instance2"
  ],
  "callbackUrl": "http://callback_url1"
}' 'http://monitoring_backend_url/api/alarms'
```

Example response:

```
{
  "status": "alarm created",
  "id": "_23zy3xgdg"
}
```

A possible trigger of the alarm in this example should send to http://callback_url1 the following JSON object:

```
{
    "measurementType": "memfree",
    "expected": "gt 8018198432",
    "alarmingInstances": [
        {
            "instance": "instance1",
            "value": 8018194432,
```

```
            "time": "2016-04-16T11:39:32.069012Z"
        }
    ]
}
```

where *measurementType* is the measurement type of one trigger, *expected* is the condition that was set in that trigger, *alarmingInstances* are the instances which triggered the alarm and more specifically, *instance* is the instance UUID, *value* is the measurement value and *time* the timestamp of the measurement.

10.3.3. Delete a specific alarm trigger

```
DELETE /alarms/{id}
```

Example:

```
curl -X DELETE --header 'Accept: text/html'
'http://monitoring_backend_url/api/alarms/_xvazxyrfc'
```

Example response:

```
{
  "id": "_xvazxyrfc",
  "status": "alarm deleted"
}
```

10.3.4. Get a specific alarm trigger's details

```
GET /alarms/{id}
```

Example:

```
curl -X GET --header 'Accept: application/json'
'http://monitoring_backend_url/api/alarms/_xvazxyrfc'
```

Example response:

```
{
  "id": "_xvazxyrfc",
  "triggers": [
    {
      "type": "memfree",
      "comparisonOperator": "gt",
      "threshold": 8018198432
    }
  ],
  "instances": [
    "archie",
    "archie2"
  ],
  "callbackUrl": "http://callback_url1"
}
```