Secure Provisioning of Cloud Services
based on SLA Management

SPECS Project - Deliverable 3.4.2

SPECS Monitoring Services
Final

Version no. 1.0
30 April 2016

The activities reported in this deliverable are partially supported
by the European Community's Seventh Framework Programme under grant agreement no. 610795.

Deliverable information

Deliverable no.:	D3.4.2
Deliverable title:	Design of the SPECS Monitoring Services - Final
Deliverable nature:	Prototype
Dissemination level:	Public
Contractual delivery:	30 April 2016
Actual delivery date:	
Author(s):	Silviu Panica (IeAT)
Contributors:	Marina Bregou (CSA), Valentina Casola (CeRICT), Alessandra De Benedictis (CeRICT)
Reviewers:	Madalina Erascu (IeAT), Jolanda Modic (XLAB)
Task contributing to the deliverable:	T3.4
Total number of pages:	36

Executive summary

This deliverable is associated with the prototype implementation of the Monitoring module (Task 3.4).

The goal of this document is to:

- report the final status of implementation activities;
- discuss the updates done to the architecture of the Monitoring core (two components, Event Aggregators and SLOM Exporter, were suppressed and their functionalities where transferred to other monitoring components);
- discuss about the integration of all the monitoring components into the SPECS framework with respect to M18 report;
- give instructions on how to install and use the code for the new components developed in task T3.4;
- provide links to the SPECS public repository where the source code is available.

Moreover, this document tackles the scalability and performance aspect of the Monitoring module. The analysis is focused on both component level, where each Monitoring module component is discussed from this perspective, and at module level where the discussion focuses over the entire Monitoring module architecture.

Table of contents

Index of figures

Index of tables

1. Introduction

The SPECS Monitoring module collects information about the state of target services that is relevant to the set of signed SLAs, and by forwarding notifications of possible alerts and violations to the Enforcement module. Any changes in target services that may affect the validity of any signed SLA are reported to the Enforcement module, which is in charge of the main reasoning and analysis parts (cf. Diagnosis component presented in deliverable D4.3.3).

To summarize the Monitoring module components, we next briefly present the monitoring process (detailed in D3.3). The monitoring process comprises a set of steps that transform and filter the monitoring data collected from the target services. A simplified view over the monitoring process is depicted in Figure 1. *Observe* and *Collect* is an activity covered by the **Monitoring Agents**. The collected monitoring data is first filtered and aggregated by the **Monitoring Adapters** (an extension to the Monitoring agents) and transformed into monitoring events (the Events) by expressing the collecting information using the SPECS Monitoring format (D3.3, Section 4.1). Next, the Events are sent to the SPECS platform for further processing. The Events are published to the monitoring core router, the **Event Hub**. From here, the Events are stored in the **Event Archiver** database (used for auditing or post processing) and filtered by the **Monitoring Policy Filter (Monipoli Filter)** component, that correlates the Events with the filtering rules defined by the Monitoring policy filter. If the monitoring policy is broken, some of the filtered Events are possible alerts or violations, a notification is sent, using the **SLOM Exporter**, to the Enforcement module.

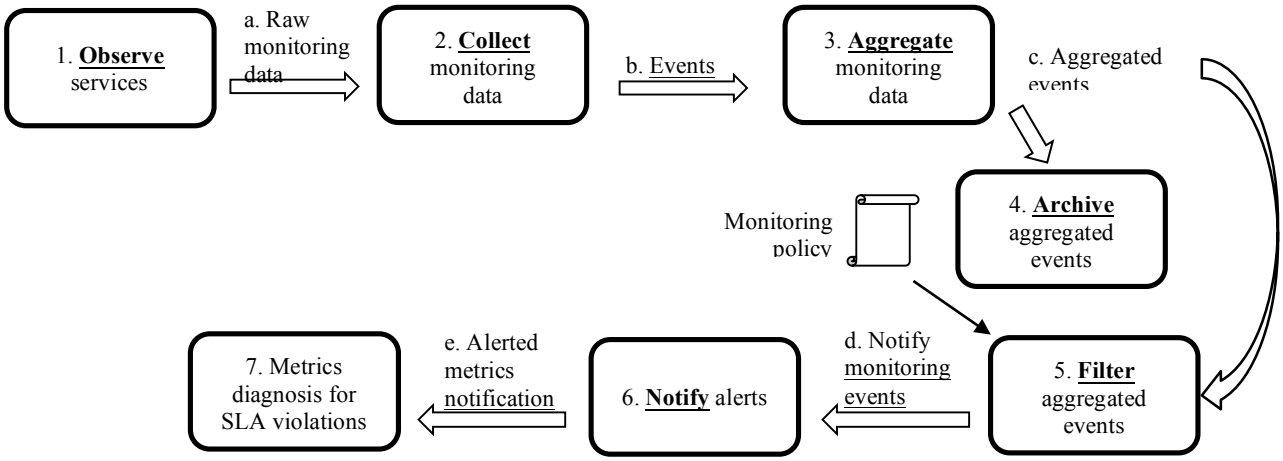

Figure 1. Simplified monitoring process.

Monitoring module components are configured by the Enforcement module in the SLA implementation phase after each new SLA is signed. The Enforcement module extracts negotiated security metrics from the SLA in order to properly configure monitoring systems for associated target services (to determine *what* to observe), and to translate metric values (the SLOs in the signed SLA) into alert and violation thresholds (to determine *when* to notify the Enforcement module about a given monitoring event). See deliverable D3.3 for all design details for the Monitoring module and deliverable D4.3.3 for the final prototypes of the Enforcement module.

In this deliverable, we report the current status of development of the SPECS Monitoring module and of associated monitoring systems. Compared to M18, where only the Event Hub component and the adapters for the SVA, OpenVAS, OSSEC, and Nmap monitoring systems were

available, at current state all Monitoring core components have been completed and are available. Actually, as discussed in detail in Section 3, the architecture of the Monitoring core has been updated in order to better fulfil the requirements originally defined. In particular, two of the Monitoring core components (i.e., *SLOM Exporter* and *MoniPoli Filter*) were substituted by the new *MoniPoli Filter* component and the functionalities offered by the Event Aggregators were integrated into the adapters. For what regards the monitoring systems, in this deliverable we report on the implementation of two additional adapters: one adapter has been developed for Nmap, used to monitor the SPECS Platform core components to detect anomalies in the functionality of these components; another adapter has been developed for the CloudTrust Protocol (CTP) from CSA. In addition to these two, we also developed a specific adapter for the ViPR system from EMC. However, it is closed-source as the rest of the adapters, so no details are given in this deliverable.

The document is structured as follows. In Section 2 we describe the relationships with other deliverables. Section 3 is dedicated to the core Monitoring components developed and not covered or updated with respect to the components described in D3.4.1. It reports the status of development activities and provides installation and usage guides for the Event Archiver and the MoniPoli Filter, which also integrates, in the final version, the functionalities of the SLOM Exporter component. Section 4 describes the scalability potential of the Monitoring module and the performance tests conducted. Section 5 describes the monitoring systems Nmap and CTP, which are used for evaluating the overall status of the SLA Platform core components, and for establishing digital trust between a Cloud Service Customer (CSC) and a Cloud Service Provider (CSP), respectively.

2. Relationship with other deliverables

The work presented in this document is related mainly to activities of other tasks in WP3. The deliverable D3.1 provided an overview of existing monitoring tools and frameworks; D3.2 discussed the requirements that the integrated monitoring systems should respect, D3.3 presents the design of the entire Monitoring module and D3.4.1 described the initial set of components that were already developed. Moreover, there are also indirect relations with deliverable D4.3.3 (the Monipoli Filter configuration process, Enforcement components interaction with Event Archiver and security mechanisms' monitoring adapters interaction with the Event Hub component) and D1.5.2 where integration scenarios are described and they rely on the Monitoring module components.

Figure 2 shows the relationships described above.

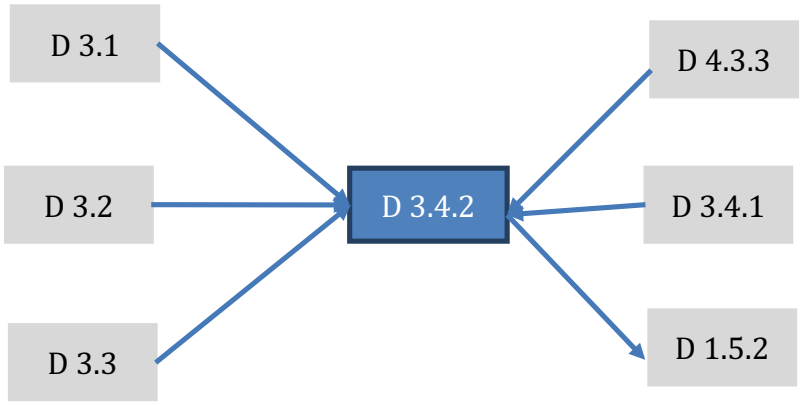

Figure 2. Relationships with other deliverables

3. Monitoring core

The design of the architecture of the Monitoring module was covered by D3.3 The same document also describes into detail the components of the Monitoring module. Briefly, the Monitoring module consists of the following components (their role in the monitoring process was explained in Section 1):

- the Event Hub (central monitoring event router),
- the Event Archiver (monitoring events database used for audit or post processing),
- the Monipoli Filter (filtering service that detect anomalies),
- the SLOM Exporter (the component used to notify the Enforcement module),
- the Monitoring Agents and Monitoring Adapters (used to observe and collect monitoring data from target services),
- the Monitoring Aggregators (used to aggregate the monitoring data).

Due to the implementation process, some of the functionalities of two initially stand-alone components, Event Aggregators and SLOM Exporter, were transferred to be supported by the Monitoring Adapters (Adapters) and the Monipoli Filter.

In terms of artefacts, the list was updated as we have merged the two artefacts SLOM Exporter and MoniPoli Filter in a new one, named Monipoli and we have covered all Event Aggregators functionalities with the Adapters and deprecated the Event aggregator artefact. In conclusion, three artefacts were deprecated and removed from the final design. With respect to this changes we also updated the Monitoring module architecture, depicted in Figure 3.

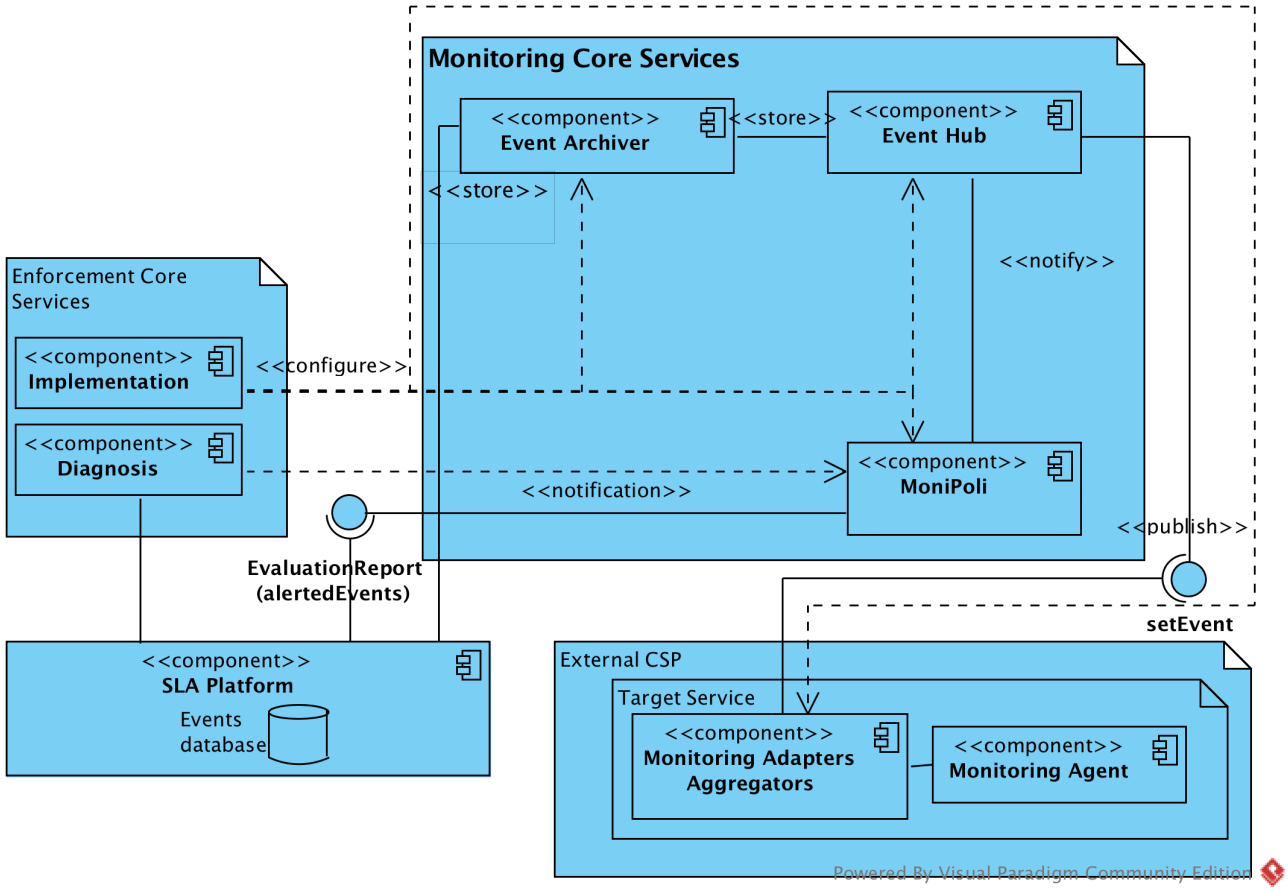

Figure 3. Monitoring module - Final architecture

The SLOM Exporter component was designed to translate the monitoring events messages, described in a custom format, into notifications that are handled by the Enforcement component, which uses a different format. While initially this was designed as a standalone component, in the final version we integrate it into the MoniPoli Filter. The reason to do this, was to simplify the overall architecture in order to address the scalability problem. The Event Aggregators were also moved into the Adapters based on the same reason. Moreover, having the aggregators tide with the Adapters also boosts the performance because it distributes the aggregation operation instead of having it delivered by a central component.

In the same line, some monitoring components were slightly modified in terms of backend support by adding a distributed database. The distributed database makes the scalability process more simple and effective. More details about the performance and scalability analysis are reported in Section 4.

3.1. Status of development activities

In Table 1 we report the list of Monitoring module components under development, as discussed in D1.1.2, D3.2, D3.3, and D3.4.1, together with the requirements they respectively cover.

Monitoring module Requirements	SPECS Software Components			
	Event Archiver	Event Hub	MoniPoli (supersedes SLOM Exporter and MoniPoli Filter)	Adapters
MON_STA_R1				x
MON_STA_R2			x	
MON_STA_R3				x
MON_STA_R4	x			
MON_STA_R5	deprecated			
MON_STA_R6	x			
MON_STA_R7	x		x	
MON_DSH_R1			x	
MON_DSH_R2				x
MON_DSH_R3				x
MON_DSH_R4			x	
MON_DSH_R5			x	
MON_DSH_R6	covered by the User Management component			
MON_SWC_R1				x
MON_SWC_R2				x
MON_SWC_R3				x
MON_SWC_R4	superseded by MON_BSC_R5			
MON_SWC_R5		x		
MON_SWC_R6	superseded by MON_BSC_R4			
MON_SWC_R7	x			
MON_SWC_R8	deprecated			
MON_SWC_R9	superseded by MON_BSC_R1			
MON_SWC_R10	covered by the Enforcement module			
MON_SWC_R11	superseded by MON_BSC_R1			

MON_SWC_R12	superseded by MON_BSC_R2			
MON_SWC_R13	superseded by MON_BSC_R3			
MON_SWC_R14	superseded by MON_BSC_R4			
MON_SWC_R15	superseded by MON_BSC_R5			
MON_SWC_R16	superseded by MON_BSC_R6			
MON_SWC_R17	superseded by MON_BSC_R7			
MON_SWC_R18	superseded by MON_BSC_R8			
MON_SWC_R19				x
MON_SWC_R20			x	
MON_SWC_R21			x	
MON_SWC_R22	deprecated			
MON_SWC_R23				x
MON_SST_R1				x
MON_SST_R2				x
MON_SST_R3	superseded by MON_BSC_R6			
MON_SST_R4	superseded by MON_BSC_R8			
MON_SST_R5	superseded by MON_BSC_R12			
MON_SST_R6				x
MON_SST_R7				x
MON_SST_R8	superseded by MON_BSC_R6			
MON_SST_R9	superseded by MON_BSC_R8			
MON_SST_R10	superseded by MON_BSC_R12			
MON_NEG_R1			x	
MON_NEG_R2			x	
MON_NEG_R3			x	
MON_COS_R1				x
MON_COS_R2				x
MON_COS_R3				x
MON_COS_R4				x
MON_COS_R5				x
MON_COS_R6	covered by the Enforcement module			
MON_COS_R7	covered by the Enforcement module			
MON_COS_R8				x
MON_COS_R9				x
MON_COS_R10	x			x
MON_COS_R11				x
MON_COS_R12	superseded by MON_BSC_R9			
MON_COS_R13		x		x
MON_BSC_R1				x
MON_BSC_R2		x		x
MON_BSC_R3		x		x
MON_BSC_R4				x
MON_BSC_R5			x	
MON_BSC_R6			x	
MON_BSC_R7			x	x
MON_BSC_R8	x	x	x	
MON_BSC_R9		x		x
MON_BSC_R10		x		x

MON_BSC_R11		x		x
MON_BSC_R12	x	x		
MON_BSC_R13	x		x	x
MON_DRE_R1				x
MON_DRE_R2				x
MON_DRE_R3				x
MON_DRE_R4				x
MON_DRE_R5				x
MON_DRE_R6				x
MON_DRE_R7				x
MON_DFE_R1				x
MON_DFE_R2				x
MON_SSB_R1				x
MON_SSB_R2				x
MON_SSB_R3				x
MON_SSB_R4				x
MON_SSB_R5				x
MON_ENF_R1	covered by the Enforcement module			
MON_ENF_R2	superseded by MON_BSC_R5			
MON_ENF_R3	x	x	x	

Table 1. SPECS Components related to the Monitoring module and related requirements

There are 28 total requirements that are relevant to the final prototypes of the core Monitoring components. 19 requirements have been superseded, 8 requirements are deprecated or covered by other SPECS modules and 37 requirements are exclusively associated to Monitoring Adapters that are part of security mechanisms (Enforcement module) and thus out of scope of this deliverable (for design and implementation details of Monitoring Adapters see D4.3.2 and D4.3.3).

With the final core prototypes, we have covered 100% of all core requirements with unit tests reported in Section 3.2.1, 3.3.3 and 3.4.3.

As reported in Deliverable D3.3, we developed a set of Adapters to integrate already available monitoring systems that monitor specific security metrics associated with the scenarios and SPECS applications developed in WP5. Furthermore, with respect to the previous prototype implementation, we have covered all Event Aggregator related requirements with the Monitoring Adapter components and deprecated the Event Aggregator component. In Table 2, we report the final development status of all SPECS artefacts associated with the Monitoring module. In particular, as widely illustrated in design related deliverables (cf. D3.3 and D1.1.2), these artefacts include both the components and models that had to be developed in the tasks of WP3 before the end of the project.

Module	Artefacts under development	Status
Monitoring module	Components: Event Hub	Available
	Components: Event Archiver	Available
	Components: MoniPoli	Available
	Components: Adapters	Available
	Model:MoniPoli	Available

Table 2. Monitoring Module Implementation Status

Note that the Adapters artefacts include all those components to connect different kind of Monitoring Systems that can be enforced depending on the metrics included in the signed SLA. Six different Adapters have been developed, four of these were already presented in D3.4.1 (cf. Section 4), last two (Nmap and CTP) have been finalized and presented in the remainder of this document.
All the artefacts related to monitoring core services are publicly available on the SPECS Bitbucket repository [14] and integrated within the SPECS Framework (as presented in deliverables D1.5.1 and D1.5.2).

3.2. Event Hub

The Event Hub is responsible for routing monitoring events between the other components of the Monitoring module. The Event Hub was described in details in deliverable D3.4.1. In this document we only present the functional tests conducted to prove the correctness of the requirements coverage.

3.2.1. Tests

The following tables include a list of functional tests conducted in order to test if the event hub requirements are properly covered. Two requirements, MON_BSC_R9 and MON_BSC_R10, are not included in this functional tests as they are related to Event Hub performance and are covered by the performance tests and analysis, in Section 4.2. The tests are performed using Java JUnit library[1] and are available at:

- https://bitbucket.org/specs-team/specs-monitoring-unit-testing

Test ID	test_eh_test_event_submit
Test objective	Test if the Event Hub (EH) is able to receive monitoring events and correctly routing them based on the `labels` values.
Verified requirements	*MON_SWC_R5, MON_COS_R13, MON_BSC_R3, MON_BSC_R8, MON_ENF_R3*
Inputs	One of a list of monitoring events.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_eh_test_monitoring_event_format
Test objective	Test if the EH is publishing a received monitoring event in the correct SPECS Monitoring Event format.
Verified requirements	*MON_BSC_R2*
Inputs	Intermittent monitoring events.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

3.3. Event Archiver

This component aims to store all the monitoring data and events for a defined period of time. The information regarding a specific monitored SLA is stored by the archiver during the SLA

[1] http://junit.org/junit4/

lifecycle. The architecture of the Event Archiver is depicted in Figure 4 and the complete design details are described in D3.3.

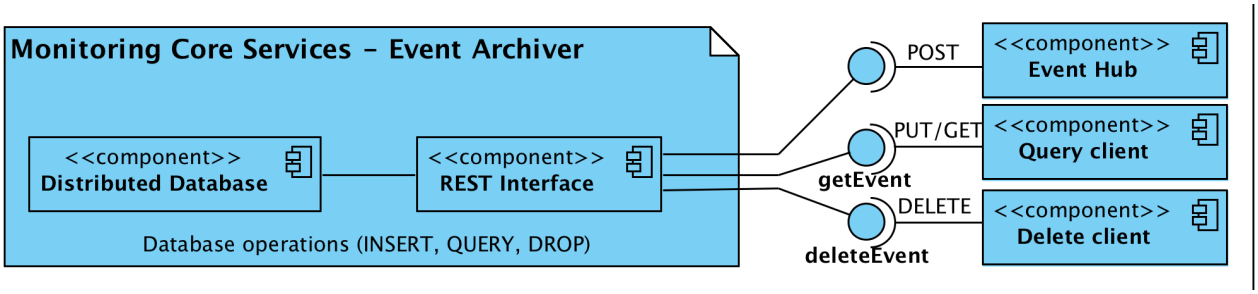

Figure 4. Monitoring core - Event Archiver architecture

3.3.1. Installation and configuration

The Event Archiver is publicly available on SPECS Bitbucket repository and all the details on the installation and configuration procedures are described in the dedicated repository wiki page:

- https://bitbucket.org/specs-team/specs-core-monitoring-event-archiver

The Event Archiver was designed to run on a Unix/Linux environment that has support for the following requirements:

- MongoDB - Distributed NoSQL Database - version 3.2.x [1];
- Python Programming Language - version 2.7.x;
- Python libraries: Flask, Flask-PyMongo, PyMongo, Pip, Virtualenv;
- Mercurial for repository download [2];

Assuming that the running environment where the Event Archiver is intended to be installed meets the above requirements, the following command lines will download, install and configure the Event Archiver:

```
      mkdir -p /opt/specs-monitoring-event-archiver
      cd /opt/specs-monitoring-event-archiver
      hg clone https://bitbucket.org/specs-team/specs-core-monitoring-
event-archiver .
      virtualenv pyenv
      virtualenv pyenv --relocatable
      source pyenv/bin/activate
      pip install -r requirements.txt
      sed -i 's/VIRTUAL_ENV=\".*/VIRTUAL_ENV=\"\/opt\/specs-monitoring-
event-archiver\/pyenv\"/g' pyenv/bin/activate
```

To start the Event Archiver:

```
      (start MongoDB database)
      /opt/specs-monitoring-event-archiver/bootstrap.sh start
```

If the start process was successful the Event Archiver should now be available at *http://localhost:10101/monitoring/events* . The database backend is automatically configured if the storage system is working and is accessible.

For the complete guide on how to install and configure the Event Archiver, we recommend following the dedicated wiki page available on the Bitbucket repository.

3.3.2. Usage

The Event Archiver exposes a REST-based web interface that enables the following operations for dealing with monitoring events data:

Register monitoring events

Resource URL	*http://localhost:10101/monitoring/events*	
POST	*Request body*	SPECS Monitoring Event format [4]

Query for monitoring events

Operation for simple queries (don't exceed 255 caracters in length):

Resource URL	*http://localhost:10101/monitoring/events*	
GET	*Request query*	/?filter={}&sort={}
	Description	filter and sort are strings expressed using MongoDB Query Language [3]

Operation for complex queries:

Resource URL	*http://localhost:10101/monitoring/events*	

POST	*Request body*	{
		"filter" : {},
		"sort" : {}
		}

	Description	filter and sort are string expressed using MongoDB Query Language [3]

Delete monitoring events

Resource URL	*http://localhost:10101/monitoring/events*	

DELETE	*Request body*	{
		"filter" : {},
		"sort" : {}
		}

	Description	*filter* and *sort* are string expressed using MongoDB Query Language [3]

For more detailed usage instructions please follow the recommendations available on the dedicated repository wiki page.

3.3.3. Tests

In the following table we present a set of tests conducted in order to check the requirements coverage. The tests are performed using Java JUnit library and are available at:

- https://bitbucket.org/specs-team/specs-monitoring-unit-testing

Test ID	test_ea_write_event
Test objective	Test if the Event Archiver (EA) is storing the events.
Verified requirements	*MON_SWC_R7, MON_BSC_R8*
Inputs	One event or a list of events.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_ea_delete_events
Test objective	Test if the EA is able to delete one event or a list of events.
Verified requirements	*MON_STA_R7*
Inputs	One or a list of event identifiers.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_ea_test_search
Test objective	Test if the EA is able return a set of events based on a search criteria.
Verified requirements	*MON_STA_R6, MON_COS_R10, MON_BSC_R13, MON_ENF_R3*
Inputs	One or a list of event identifiers.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

3.4. MoniPoli

MoniPoli Filter component is in charge of the filtering of the monitoring events. Based on a set of predefined filtering rules, it triggers the Enforcement module with a notification of a possible SLA alert or violation. In the latest version, MoniPoli component incorporates also the functionality of the SLOM Exporter component. SLOM Exporter is used to notify the Enforcement module using a specific message format (translated from the monitoring event format). The MoniPoli was subject to some architectural changes by adding as a backend support a distributed database, namely MongoDB. The new architecture is depicted in Figure 5. The overall architecture and functionality described in D3.3 didn't change substantially but only the backend was changed from local file storing into a database backend.
The MoniPoli is publicly available on SPECS Bitbucket repository and all the details on the installation and configuration procedures are described in the dedicated repository wiki page:

- https://bitbucket.org/specs-team/specs-core-monitoring-monipoli

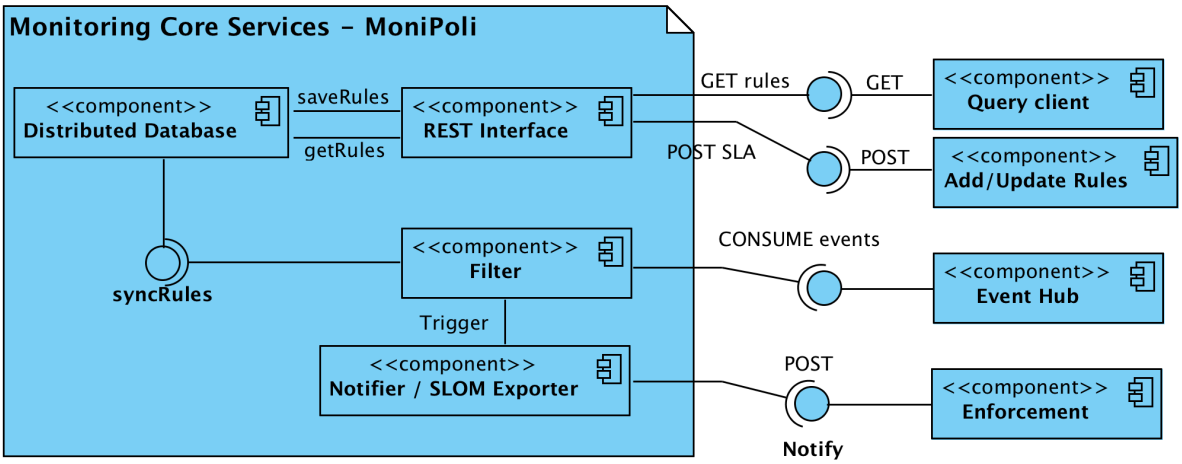

Figure 5. Monitoring core - MoniPoli architecture

3.4.1. Installation and configuration

The MoniPoli is entirely written in JavaScript and it has the following requirements:

- NodeJS 4.2.x or greater;
- NodeJS packages: express, xml2js and xmlbuilder (for compatibility the MoniPoli comes with these packages bundled);
- MongoDB - Distributed NoSQL Database - version 3.2.x;
- Mercurial for repository download;

Assuming that the running environment where the MoniPoli is intended to be installed meets the above requirements, the following command lines will download, install, and configure the MoniPoli:

```
mkdir -p /opt/specs-monitoring-monipoli
cd /opt/specs-monitoring-monipoli
hg clone https://bitbucket.org/specs-team/specs-core-monitoring-
monipoli .
```

Next, the MoniPoli service can be controlled using:

```
/opt/specs-monitoring-monipoli/bootstrap.sh [start|stop]
```

The configuration parameters of the MoniPoli are defined in (explanatory comments are provided for each configuration parameter):

```
/opt/specs-monitoring-monipoli/monipoli.sh
```

The database backend is automatically configured if the storage system is working and accessible. The default configuration template assumes that the Diagnosis component (Enforcement module) and the monitoring router (the Event Hub) are hosted locally.

3.4.2. Usage

Interaction with the MoniPoli is possible through the RESTful API exposed by the component. The default HTTP port is 5000. MoniPoli exposes three functional operations:

- Create rules definition based on an agreed SLA (new or updated rules);

- List the filtering rules;
- Delete the filtering rules associated with a specific SLA;

Create monipoli filtering rules

Resource URL	*http://localhost:5000/monipoli*	
POST	*Request body*	SLA document
	Description	MoniPoli will generate the filtering rules based on the content of the SLA document.

List the filtering rules

Resource URL	*http://localhost:5000/monipoli*	
GET	*Request query*	Empty
	Description	this operation will return the list of filtering form, in plain text, having the following structure: *Rule1:* *specs_webpool_M1* *level_of_redundancy_m1* *3* *geq* *a3dfddg234fw* *Rule2:* *specs_webpool_M2* *level_of_diversity_m2* *2* *geq* *a3dfddg234fw* Where: • line1: represents the rule number; • line 2: represents the metric name; • line 3: represents the measurement identifier; • line 4: represents the expected value of the measurement identifier; • line 5: represents the logical operator to be applied on the value; • line 6: represents the SLA identifier; • line 7: empty line - delimiter.

Delete the filtering rules associated with a specific SLA

Resource URL	*http://localhost:5000/monipoli*	
DELETE	*Request body*	{ "sla_id" : "a3dfddg234fw" }
	Description	MoniPoli will delete rules assigned to a specific SLA.

3.4.3. Tests

In the following table we present a set of tests conducted in order to check the requirements coverage. The tests are performed using Java JUnit library and are available at:

- https://bitbucket.org/specs-team/specs-monitoring-unit-testing

Test ID	test_mp_submit_SLA
Test objective	Test if the Monipoli(MP) is able to parse and extract the required filtering rules from an SLA document.
Verified requirements	*MON_STA_R2*
Inputs	One SLA document.
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_mp_test_delete_SLA
Test objective	Test if the MP is able to delete the filtering rules associated with a given SLA identifier.
Verified requirements	*MON_STA_R2, MON_NEG_R1, MON_STA_R7*
Inputs	One SLA identifier
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_mp_test_update_SLA
Test objective	Test if the MP is able to update its filtering rules based on a new SLA document.
Verified requirements	*MON_BSC_R5*
Inputs	One SLA document
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_mp_test_registered_fitering_rules_SLA
Test objective	Test if the MP is correctly generating the filtering rules based on an SLA document.
Verified requirements	*MON_BSC_R6*
Inputs	One SLA document
Expected results	All operations execute successfully.
Outputs	None.
Comments	All operations executed as expected.

Test ID	test_mp_test_alert_violation_behaviour
Test objective	Test if the MP is correctly filtering the monitoring events that might be alerts or violations.
Verified requirements	*MON_DSH_R4, MON_NEG_R2, MON_NEG_R3, MON_BSC_R7, MON_DSH_R1, MON_BSC_R8, MON_ENF_R3, MON_DSH_R5*
Inputs	Monitoring events that simulate alerts or violations
Expected results	All operations execute successfully.

Outputs	None.
Comments	All operations executed as expected.

4. Monitoring scalability and performance

4.1. Monitoring scalability

The monitoring module is designed to handle a large amount of data in a short period of time. This requirement was naturally raised because of the nature of the SPECS applications that are offered to the End-users and (ii) the number of end-users that SPECS is supposed to handle. These two aspects are translated into a large number of Adapters that are supposed to be deployed. As more and more new End-users are using the SPECS Platform and deploy different new applications, the Monitoring module may need to add extra resources in order to sustain the new workload. This property is called scalability. Monitoring module needs to be able to scale, with respect to the workload that it has to handle.

In order to establish whether the Monitoring module is scalable or not, we need to analyse every component that is part of the module. Each component that might be affected by the increased workload effect needs to be scalable as well. Therefore, in the remainder of this section, each main component is described from the scalability point of view and then the entire Monitoring module is discussed, pointing out how scalability can be implemented.

The main components of the monitoring module are:

- the Event Hub;
- the Event Archiver;
- the MoniPoli;
- the Monitoring Agents (which are important in the discussion but not part of the monitoring core);

The Distributed Event Hub

The Event Hub acts like a router of messages among components and delivers the messages based on a stream subscriber policy to the endpoints waiting for specific messages. The streams are preconfigured and do not change during the lifecycle of the Platform nor SLA. Based on this assumption, multiple instances of the Event Hub can be started without affecting the routing process. As the Event Hub is the entry point of the messages collected by the Monitoring core, all its' instances should have one entry point descriptor. At network level, SPECS uses unique hostnames for each component. When the Enabling Platform (described in D1.6.1 and D1.6.2) deploys the component it registers the component into the internal DNS service. In the case of the Event Hub, each instance will register its unique IP address under the same hostname (Figure 6). In this way all the clients will use one unique hostname to discover all the instances of the Event Hub.

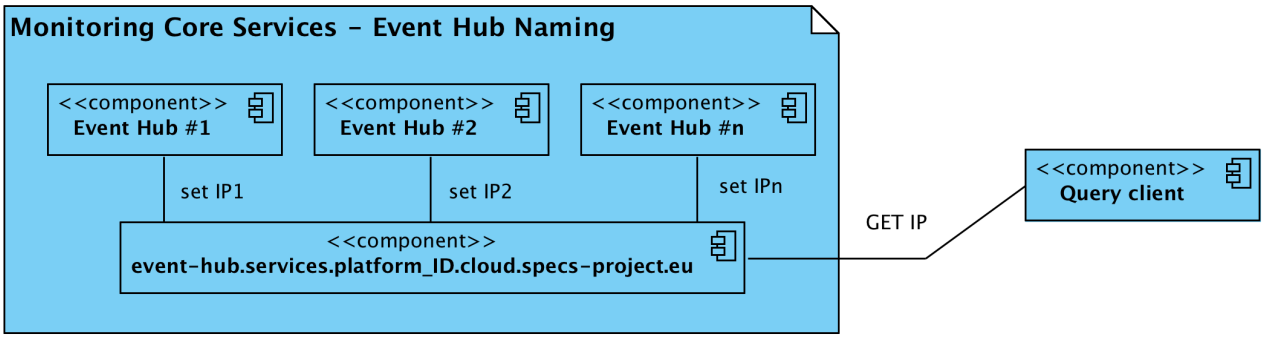

Figure 6. Event Hub - Global Naming Convention

The Distributed Event Archiver

All the monitoring data needs to be archived during the life cycle of an SLA. SPECS core services are using the archiving service to extract information about the events. These core services have to have a single view over the archived data. Having multiple instances of the Event Archiver imposes the need of a synchronization mechanism over the stored data, across all the instances. This property is known as eventual consistency in a distributed system. The Event Archiver component uses as a backend a distributed database called MongoDB. Each Event Archiver instance is deployed with its own database instance. In order to achieve the consistency, the database is setup in replica mode [10].

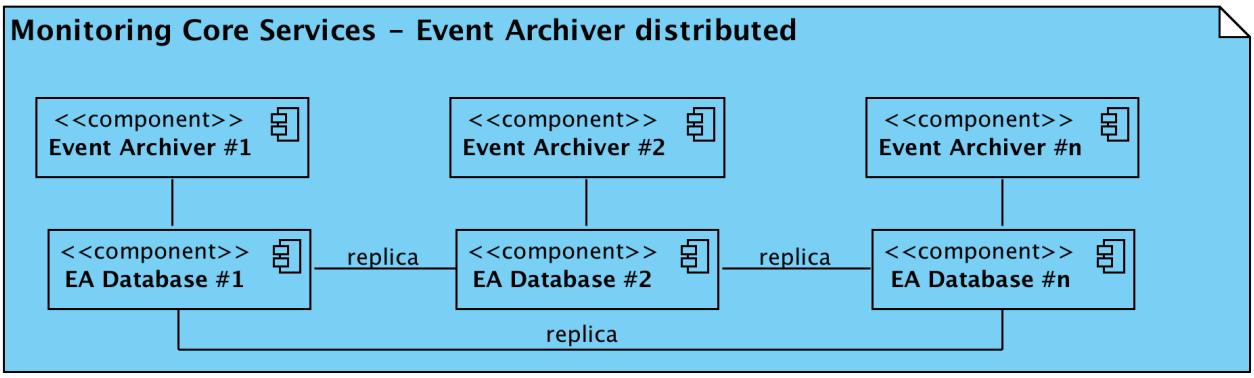

Figure 7. Event Archiver - Distributed Architecture

In this mode, multiple instances of the Event Archiver will write the monitoring data on its own database instance. The distributed database will handle the write operations so that each instance will be synchronized with the others to share the same view of the stored data. The distributed architecture is depicted in Figure 7.

The Distributed MoniPoli

The monitoring data is filtered in order to detect deviations from the agreed SLAs. MoniPoli Filter uses filtering rules that are generated based on the agreed SLA documents. In case of a distributed MoniPoli Filter (multiple MoniPoli Filter instances that are filtering events within the same SLA), the filtering rules should be the same across all the instances. In other words, the filtering rules needs to be synchronised. In order to achieve this, the MoniPoli also relies on a distributed database, by using the same MongoDB technology (Figure 8).

At this point, when a new SLA is submitted or current filtering rules need to be updated, the instance where the operation call is made will update the filtering rules dataset. The distributed database will handle the synchronization of the replicas so all the instances should have the same view over the filtering rules dataset.

One problem remains: how the instances are forced to re-read the new filtering rules from the database? The instances are configured to read the filtering rules table at each second. The instance that receives *create* or *update* operation call, will set a `dirty` bit into the filtering rules dataset. In this way, at the next iteration (of 1 second), all the instances will have the same view and use the same filtering rules for processing the monitoring data.

In case of sending out the notifications to the Diagnosis component (Enforcement module), each instance will use the same naming convention (as described in the Event Hub paragraph above) to get the IP address of the Diagnosis component. In this way, if multiple Diagnosis components are deployed, there is a single point of access to all the instances by using the

hostname that retains all the IP addresses of the instances (similar approach like in the Event Hub case).

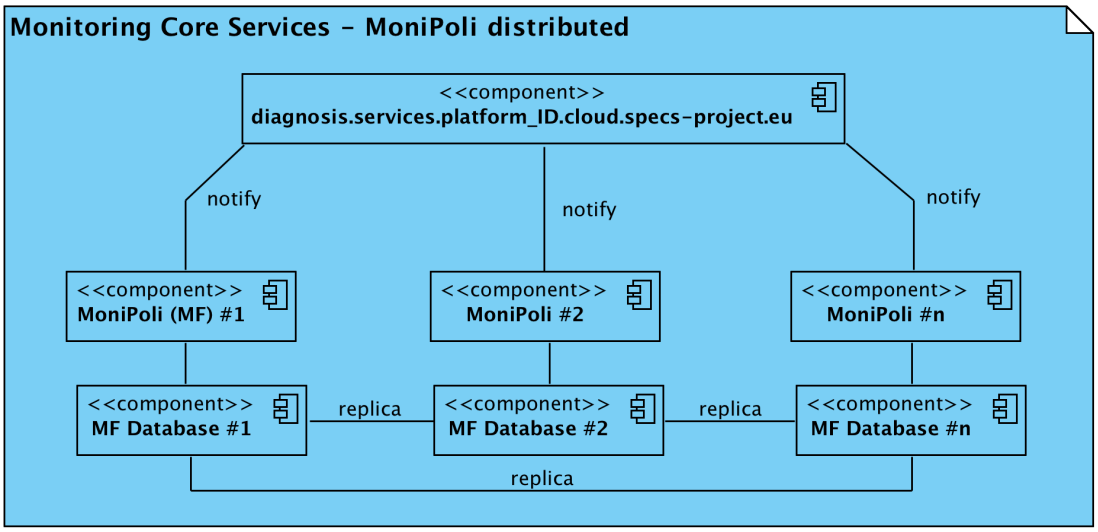

Figure 8. MoniPoli - Distributed Architecture

The Monitoring Agents

The monitoring agents interact only with the Event Hub for sending out the monitoring data collected from the hosts. Service level specific data is translated into SPECS monitoring events by the Monitoring Agents. The Monitoring Agents are configured to use the Event Hub unique hostname (described in the Event Hub paragraph above) to access the router. One consideration: if the Event Hub unique hostname retains more than one IP address, then the monitoring agents need to be able to connect to the addresses using timeouts. If within a time frame the router does not respond, it should try the next IP address as some instances may be not reachable or faulty.

The scalability implementation

In the end, we present the scalable architecture of the monitoring core components. Initially, the SPECS Platform will start with one instance of each monitoring core component. In case that the Monitoring module needs to scale, to handle extra workload, minimum two new instances of the core components will be added. Minimum two new instances are required to ensure the minimum optimal number of members that are needed by the distributed database to have a quorum. The quorum is required in replica mode for executing specific distributed operations. The scalable architecture is presented in Figure 9.

Each new monitoring core instance will have:

- one Event Hub;
- one Event Archiver (with the archival database synchronised);
- one MoniPoli (with the filtering rules synchronised);
- distributed database added as a replica in the existent distributed database deployment.

This approach minimizes the deployment and management complexity of both monitoring core components and distributed databases used in the scaling process.

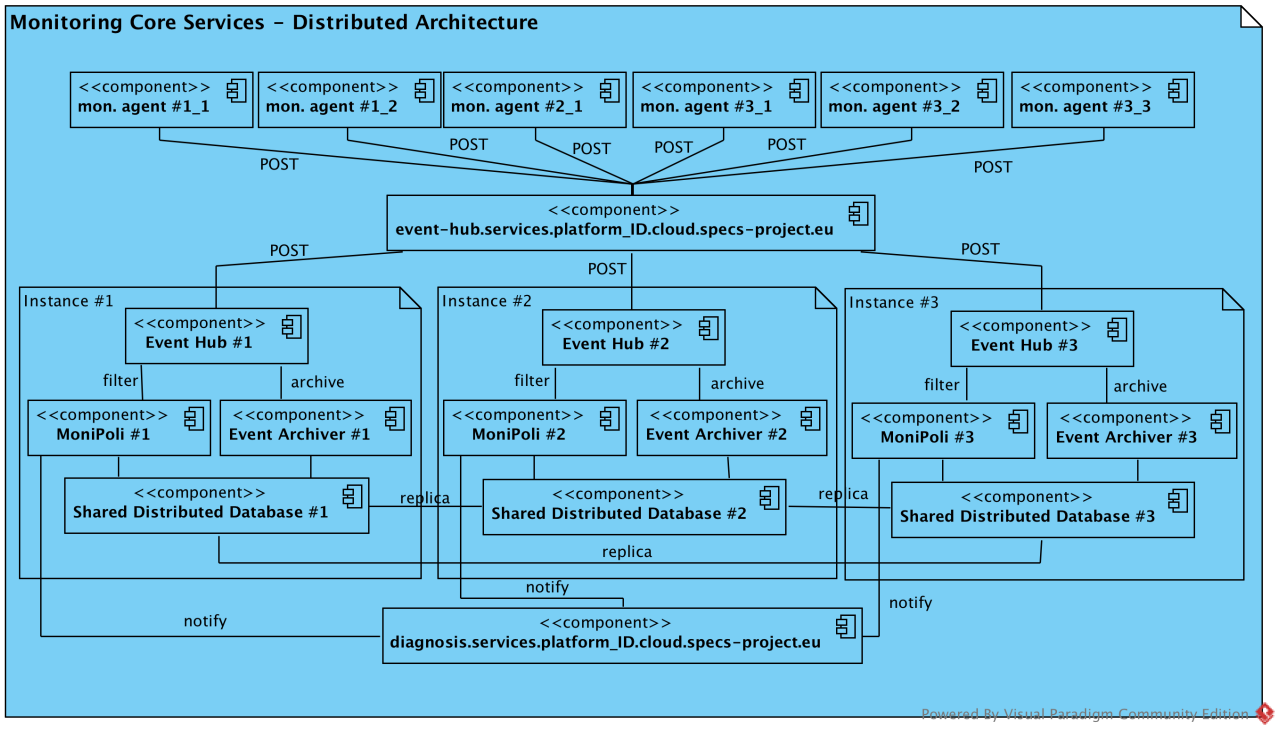

Figure 9. Monitoring module - Distributed architecture

4.2. Monitoring performance

Following the performance testing methodology described in D1.5.2, *Gatling Load Testing Tool* [11] was used to simulate heavy load on the Monitoring module. The Monitoring module assumed to have all the components started and configured like in a real environment. Due to the fact that Monitoring module deals with events and not users, the testing methodology is different as follows: (i) instead of using user profiles, we are simulating monitoring agents, that are configured for different SLAs and will feed the Event Hub with events, (ii) we added also hardware resources consumption reports to better understand where is the bottleneck, and (iii) the testing profile includes all the Monitoring module components so that we test the entire module performance and not only the individual components. Gatling profile used for performance testing is available at:

- https://bitbucket.org/specs-team/specs-performance-monitoring

The testing environment consisted of two virtual machines, one used to host the monitoring module instance and the second one used to generate the traffic load using Gatling.

In the testing scenario, we considered three different SLAs to be monitored, each SLA with an increasing number of monitoring agents, at each iteration. Gatling was configured to feed the Event Hub with monitoring events, with a linear ramp over 120 seconds. The number of monitoring agents increases between the iterations, from 1000 up to 30000 instances (Figure 10). The virtual machine used as a hosting environment had the following hardware specifications:

- CPU: 1 core - Intel Xeon E5504 2.00Ghz;
- RAM: 1GB DDR3;
- Disk: 20GB virtio[2] over SAS physical HDD.

[2] Virtualized I/O - http://wiki.libvirt.org/page/Virtio

The results show that the Monitoring module is able to support up to 30000 monitoring agents that can produce a load of 249 requests per second as outlined on Figure 11 (optimal and measured throughput have the same value) using only one instance. The throughput rate is sustained by a good response time. From this point above the errors start to occur (hardware resources are over commited) and the Monitoring module should scale by adding extra instances, to balance the high workload, or by changing the virtual machine type and adding more hardware resources.

Number of Agents	Response (ms)	Throughput (req/s)	Optimal Throughput	CPU (%)	RAM (MB)	Errors (%)
1000	3	8,34	8,33	13	175	0
2000	3	16,6	16,67	17	181	0
3000	5	25,03	25,00	27	187	0
6000	5	50,12	50,00	50	192	0
9000	6	75,01	75,00	74	201	0
12000	212	93,97	100,00	83	238	0
15000	1420	123,85	125,00	97	251	0
30000	2574	249,18	250,00	100	259	1,01

Figure 10. Performance test results

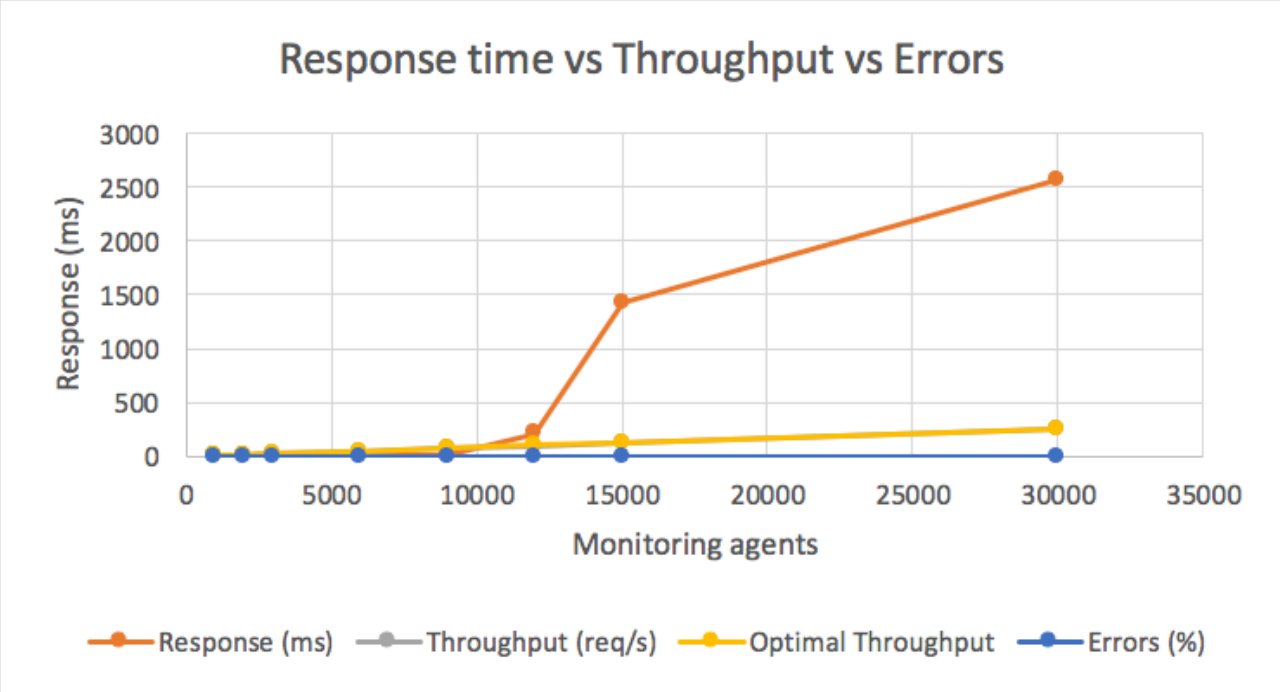

Figure 11. Performance tests: Response vs Throughput vs Errors

The bottleneck identified during this performance tests is represented by the CPU utilization. As outlined in Figure 12, the RAM footprint is relatively low but the CPU usage starts to increase substantially. But the total CPU utilization is not entirely consumed by the components processes but also by operating system service that handles the logging output (standard output and error) of the processes. Also the I/O operations performance of the virtual machine draws back the performance of the Monitoring module. The Monitoring components were configured to use normal logging simulating a real running environment.

Due to all these aspects, the components performance is good, even as running on a slower hardware, and it is in line with the requirements declared at the design time. The

performance requirement states that the Monitoring Event Hub must handle *a couple of hundreds of requests per second* and the entire Monitoring module is able to handle the required rate of requests.

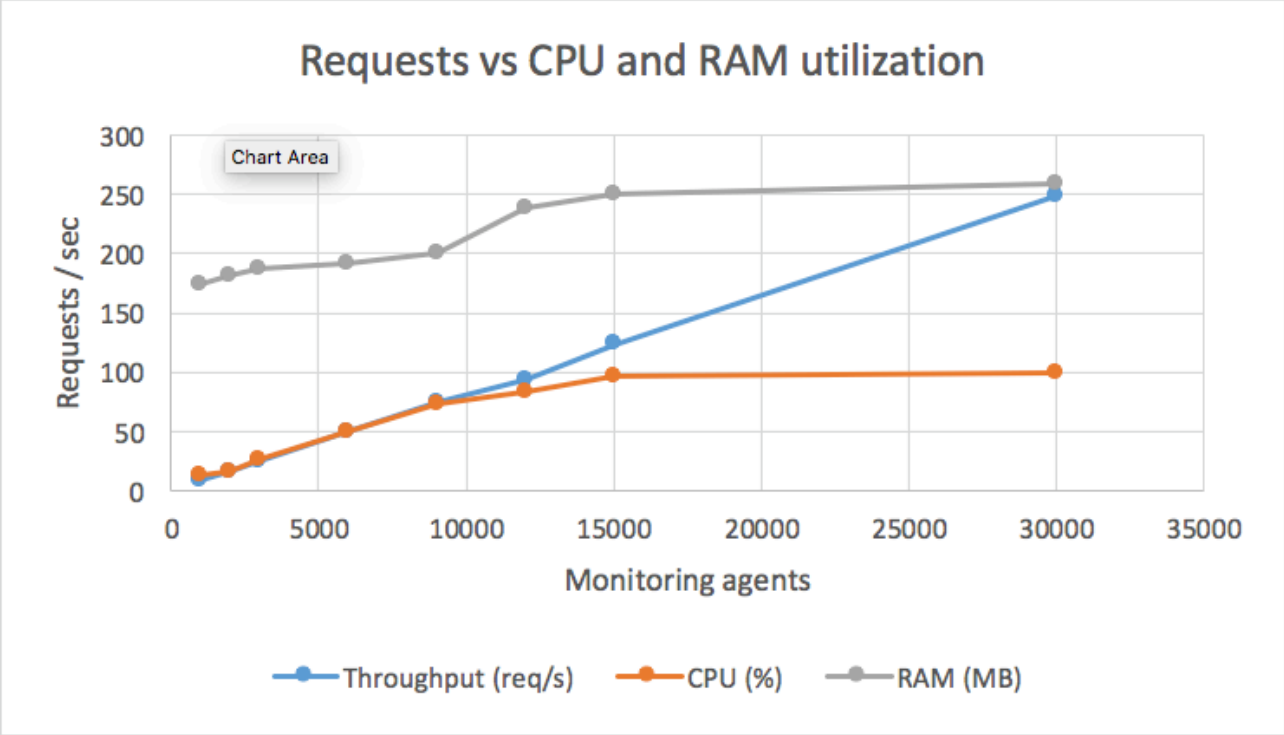

Figure 12. Performance tests: Requests vs CPU vs RAM

5. Monitoring systems

This section presents the technical details regarding the Monitoring Systems used in SPECS context to evaluate the overall status of the SLA Platform core components (NMAP) and to establish a digital trust between a CSC and a CSP (CTP). The last subsection briefly talks about ViPR Monitoring System that is presented in D5.3.

5.1. NMAP Monitoring System and Adapter

The Nmap Monitoring System and Adapter (Nmap) is a distributed monitoring system able to evaluate, among other features, the availability of the remote services. In this way, Nmap is used to monitor SPECS Platform core components to detect anomalies in the functionality of these components. All the monitoring results are pushed in the Event Hub.
The Nmap Monitoring System is composed of five components: *FrontEnd*, *Scheduler*, *Scanner*, *Converter*, and *Presenter* that can be deployed individually. *FrontEnd* component is responsible for receiving the monitoring requests. Scheduling of jobs for execution is done by the *Scheduler* component. *Scanner* component is the one that actually executes the monitoring scans in order to obtain information about the system under observation. *Converter* component applies transformations on the raw monitoring results while *Presenter* component sends the monitoring results to other systems. Communication between components is done via a message queue and a NoSQL database. Jobs to the Nmap Monitoring System can be submitted via a HTTP POST request to the *FrontEnd* component containing a JSON describing the job.

5.1.1. Installation and configuration

There are two ways of installing and configuring the Nmap Monitoring System and Adapter:

1. Using the Chef recipes[3] provided for each component. This is the easiest way, because all the configurations and dependencies are resolved automatically.
2. Manual installing each component and its dependencies. This method allows for a better customization of all the configuration parameters and the version of the dependencies.

For the first method, the only requirement is to have a virtual or a physical machine with Chef Client installed on it. The Chef recipes can be manually copied on the machine or transferred via the Chef Server. A recipe is provided for each of the four components. Each recipe downloads the needed artifacts and configures them in order to assembly the Nmap Monitoring System. Once all the recipes are executed, the system should be up and running, awaiting for monitoring requests.

The second method involves manually deploying each component and its dependencies. The following dependencies must be manually installed:

- RabbitMQ server – v3.4.4 or newer
- MongoDB – v2.6.3 or newer
- Nmap – v6.40 or newer
- JDK v1.8 or newer
- Maven v3.3.3 or newer

The components of the Nmap Monitoring System can be built from source using the *buildAll.sh* script located in the deployment directory in the repository [8]. Alternatively, the artifacts provided in the repository [13] can be used.

[3] Chef Recipes, https://docs.chef.io/recipes.html

Each of the components follows the directory structure from Figure 13. There are four main directories: *bin*, *etc*, *lib*, and *var*. The *bin* directory contains the script for starting and stopping the component. The configuration file is located in the *etc* folder and has the name *conf.properties*. This file is a standard java properties file and examples can be found in the repository [36]**Error! Reference source not found.** The *conf.properties* file contains details related to the database and message queue that will be used by the component. In the *lib* folder we can find the JAR artifacts of the component. A file containing the process identifier and the log files of the component can be found in the *var* directory.

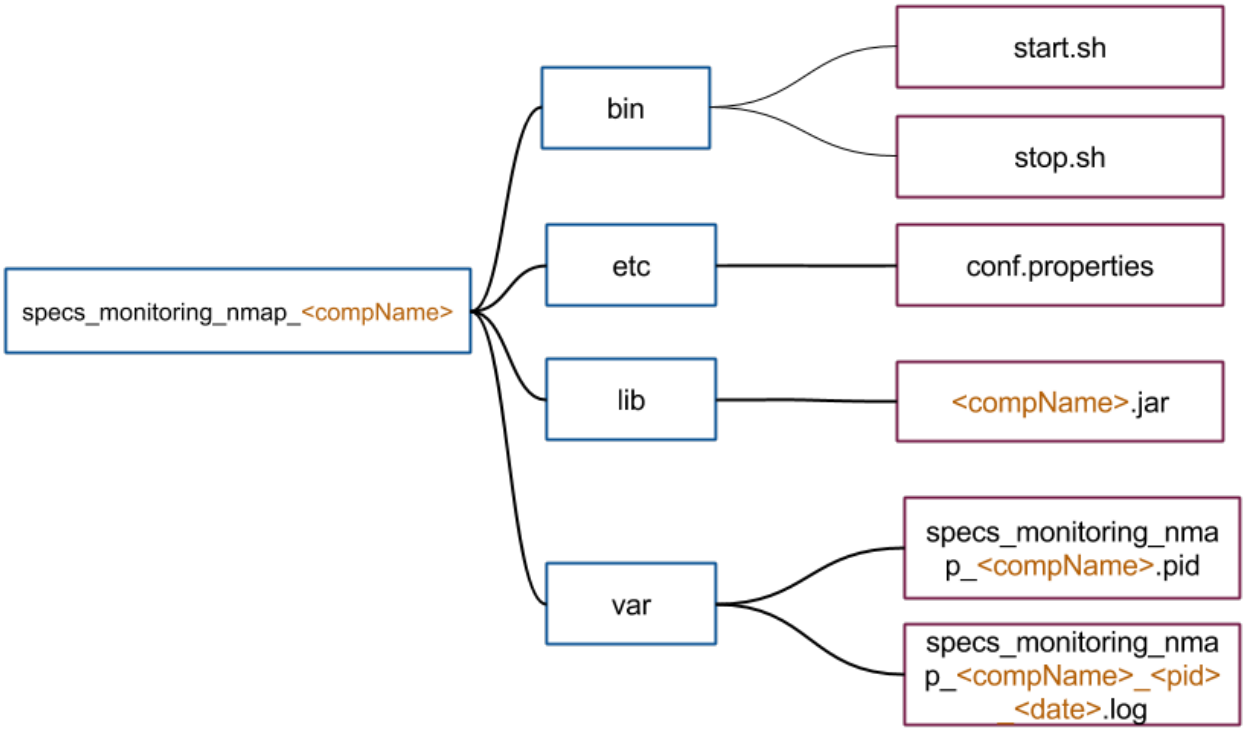

Figure 13. The structure of component deployment

5.1.2. Usage

The monitoring system accepts a scanning request in the form of an HTTP POST request. The structure of the request and examples can be found at [36]. Four types of requests are supported at the moment and multiple filters that can be combined in order to obtain the results in the desired format. Table 3 presents the commands, filters and the way they can be combined.

Commands	Filters
availability <url>	HttpStatusCodeFilter | HttpBodyFilter | HttpStatusCodeAndBodyFilter
security tls <url>	XmlToJsonConverter & TlsCiphersuitesFilter
security ecrypt2lvl <url>	XmlToJsonConverter & TlsCiphersuitesFilter & TlsEcrypt2Level
security open_ports <url>	XmlToJsonConverter

Table 3: Commands and Filters accepted by the Nmap Monitoring System

5.2. CloudTrust Protocol Monitoring System

The Cloud Trust Protocol (CTP) is designed to be a mechanism by which CSCs can ask for and receive information related to the security of the services they use in the cloud, promoting transparency and trust. As such, it is designed to create digital trust between a CSC and a CSP, on a continuous basis.

The CTP API is designed to be a RESTful protocol that CSCs can use to query a CSP on current security attributes related to a cloud service and specified in the SLA that is signed by the CSP and the CSC, such as the current level of availability of the service or information on the last vulnerability assessment. This is normally done through a classical query-response approach driven by the customer. CTP has access control mechanisms in order to assure that each CSC has only access to information related to its assets and not to assets of other customers.

The CTP server is designed with some useful security and scalability features:

- The CTP server uses OAUTH 2 "bearer tokens" for authentication.
- The CTP server can be configured to use TLS with server-side certificates (all CTP compliant servers must offer this possibility).
- The CTP server relies on MongoDB as a backend and can be scaled up if necessary to run on multiple servers.

CTP structures information provided to customers according to the following hierarchy:

- Customers subscribe to a set of services, called "service views"
- "service views" group together a set of "assets" (e.g. virtual machines, databases, etc.)
- "assets" have a set of "attributes" (e.g. uptime, key encryption strength, etc.)
- "attributes" are associated with measurement results (e.g. uptime=99.89834 %)
- "attributes" are associated with objectives (e.g. uptime>99.5%) through JavaScript-like expressions.

In SPECS, CTP is used to inform customers about the current level of security of their system, even if no alerts or violations have occurred. For example, a customer can query the current level of availability of their services in real-time through the CTP. The information presented to the customer is based on events collected by the SPECS Monitoring module and information about customer's existing SLAs. To enable this approach to work, the SPECS-CTP integration is structured around 2 simple principles:

1. Each time a new SLA is created, the CTP server creates the relevant CTP customer, service view, assets, attributes, and objectives by parsing the content of the SPECS SLA.
2. Each time a new event related to an SLA is generated, the CTP analyses the event and uses it to associate measurement results to a specific attribute.

Since both CTP and SPECS use a slightly different approach to structure information, the integration of SPECS and CTP makes use of an "adaptor" which acts like a translator between both platforms.

Hence, the CTP-SPECS integration involves 4 types of exchanges, each supported by a distinct RESTful API call:

1. SPECS will notify the appropriate CTP API of the creation of a new SLA, identified by an SLA ID.
2. CTP will query the appropriate SPECS API about the content of the SLA notified in step 1.

3. SPECS will push relevant events to the appropriate CTP API.
4. SPECS will notify the appropriate CTP API of the termination of new SLA, identified by an SLA ID.

The figure below summarizes the CTP integration architecture in SPECS.

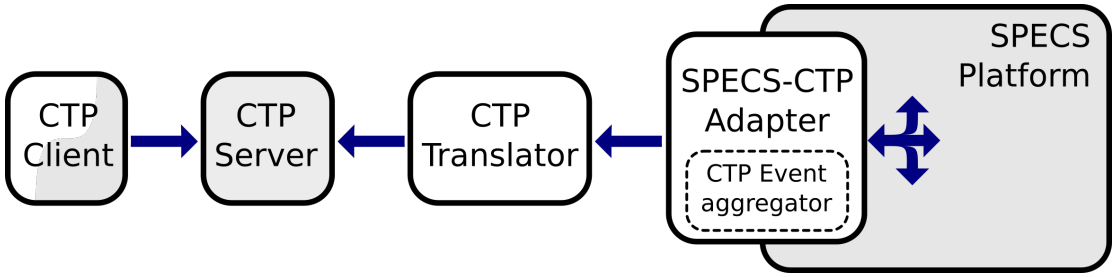

Figure 14: CTP integration with SPECS Platform

The grey components in the figure depict the ones that were already implemented, while the others were designed and developed especially for the integration of the two elements (SPECS Platform – CTP).

The components are defined as follows:

- **CTP Server**: The CTP Server implements the official CTP public API, as well as the unofficial CTP Back Office API[4].
- **CTP Translator**: The CTP Translator translates RESTful API requests originating from the SPECS CTP adapter into one or more RESTful API requests to the CTP Back Office API.
- **SPECS-CTP Adapter**: The SPECS CTP Adapter gathers data from the SPECS Platform and formulates RESTful API requests to the CTP Translator. The SPECS CTP Adapter includes an event aggregator, which is registered to the SPECS Event Hub.
- **CTP Client**: The CTP client offers a UI interface to present data to the customer. The client collects the necessary data from the CTP Server using the CTP public API.

5.2.1. Installation and configuration

Cloud Trust Protocol Daemon Prototype

Cloud Trust Protocol Daemon (ctpd) is a unix-style server, which is written in go[5] language and has MongoDB as a database backend. To compile ctpd a prerequisite is to install go and MongoDB. ctpd is a software implementation of the Cloud Trust Protocol Server (CTP Server) described above.

It has been tested on Ubuntu/Debian Linux and Mac OS X. Ctpd for SPECS implements a 'back office' API that allows the update of the database managed by ctpd. This extra API is not part of the official CTP specification.

After installing MongoDB, the simplest way to install ctpd[6] is to execute

```
go get github.com/cloudsecurityalliance/ctpd
```

[4] https://bitbucket.org/specs-team/specs-monitoring-cloud-trust-protocol-server
[5] http://golang.org/
[6] https://github.com/cloudsecurityalliance/ctpd/blob/master/INSTALL

and it gets installed automatically.

For compiling and running ctpd directly, one can type *'go run ctpd.go'*. To get more options, the command is *'go run ctpd.go –help'.* By default, ctpd runs on port 8080, and can be tested that is working with a simple curl command:

```
curl –H "Authorization: Bearer 1234" http://localhost:8080/api/1.0/
```

Alternative ways to run ctpd, more details on its use in SPECS as well as its source code are hosted in the SPECS bitbucket repository which is public, here:

- https://bitbucket.org/specs-team/specs-monitoring-cloud-trust-protocol-server

In SPECS, the ctpd is preconfigured in the RPM package that is used by the chef-orchestrating platform.

SPECS - CTP Adaptor

As mentioned above, the SPECS - CTP Adaptor is the interface between the SPECS platform and the CTP Server.

The code for the Adaptor relies on part of the CTP Server code, which must be present for compilation. This can be assured with the same command as above:

```
go get github.com/cloudsecurityalliance/ctpd
```

Compilation of this code can be executed with the command:

```
go build specs-ctp-adaptor.go
```

While running the code for the Adaptor, the program will search for the first configuration file it finds in the following locations:

the file **specs-adaptor.conf** the current working directory.
the file **.specs-ctp-adaptor.conf** in **$HOME**.
the file **/etc/specs-ctp-adaptor.conf**.

The file must be set to be readable only by the user currently running the process.

This configuration file should be customised to the platform characteristics as are mentioned in the SPECS Bitbucket repository, together with the source code:

- https://bitbucket.org/specs-team/specs-monitoring-cloud-trust-protocol-adaptor

5.2.2. Usage

The CTP Server (ctpd) listens on a specific port (default is 8080) for client connections. Clients are expected to use this connection to query the CTP server for about the current level of security of their service. The CTP server contains an optional javascript client that can be activated in the CTPD configuration file. This javascript client presents authenticated customers with a dashboard. This way the customers can have an instant overview of the their SPECS SLAs by making use of the CTP API, which provides a description of the level of security of the cloud system in near real-time through a set of attributes.

The following two figures give a general idea of how CTP works when a cloud services customer uses its API to query about the security attributes of services offered.

Figure 15, depicts how the CSC uses CTP to query a CSP about the service availability level that it is committed to provide. In CTP the result of this query is called an "objective" — or "service level objective" — since it describes what the provider aims to achieve, as typically described in an SLA.

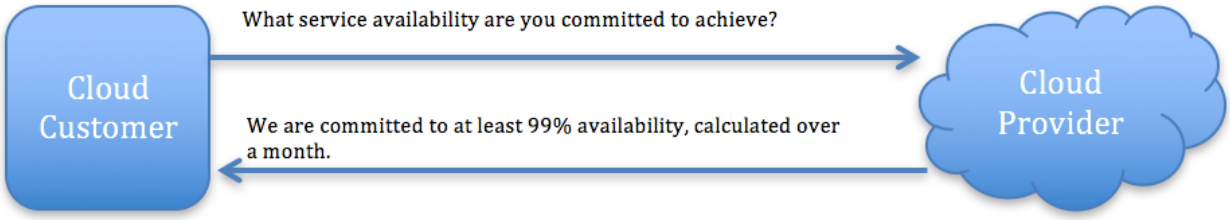

Figure 15. CTP - Service committed to provide

Figure 16, depicts the case where the CSC queries the CSP about the service availability level that was actually achieved in the past month for the customers assets. The result of this query is called a "measurement result" in CTP, since it describes the result of a service level measurement reported by the cloud provider. Both this measurement result and the objective in the previous example apply to the same security attribute informally called "availability" here.

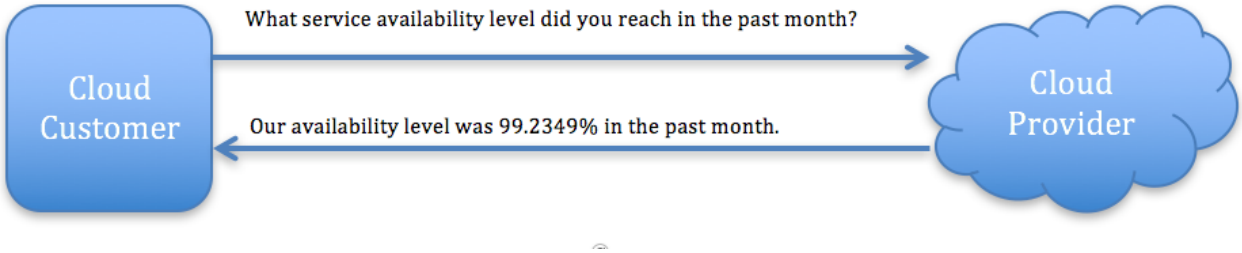

Figure 16. CTP - Service reached in the past month

5.3. ViPR Monitoring adapter

ViPR Monitoring adapter is developed as part of the *Next Generation Data Center* application. This application is developed by EMC partner and is closed-source. Due to this fact, the monitoring adapter and any other monitoring related information are exclusively described in D5.3.

6. Conclusions

This document presents final implementation of the Monitoring module, namely the monitoring core and final update on the monitoring systems able to monitor a set of metrics associated to the developed SPECS services, applications and scenarios.

The problem of scalability and performance was also tackled by describing how the scalability can be achieved both at the component and the module level. The performance tests proved that the Monitoring module performance requirement is respected. The analysis also revealed that, although the overall design of the Monitoring module was not affected, some changes were performed at the module level in order to address the scalability problem (for example, in the MoniPoli Filter component).

The state of the art of monitoring frameworks was comprehensively covered in deliverable D3.1. Moreover, in [15] this analysis was updated by adding new identified solutions which have relevance in this field. By analysing the reports about the academic prototypes or commercial services that tackle the SLA monitoring problem (or security monitoring) and correlate them with the challenges of security monitoring (in Cloud environments, described in [16]) we concluded that there are still no solutions identified to be available for Sec-SLA based Cloud monitoring. Most of the analysed SLA monitoring frameworks are covering the monitoring of a SLA lifecycle but not from the particular SLA security monitoring point of view.

With SPECS we managed to introduce a new concept, in terms of monitoring: the SLA monitoring of the security metrics. With SPECS a user can specify a list of desired SLOs (service level objectives) that he/she wants to enforce. The SLOs are defined in a security metric catalogue developed within the project. Further, SPECS platform is able create secure environment that matches the security requirements and monitor if those requirements are fulfilled during the SLA life-cycle.

The proposed implementation of the monitoring solution for SLA security metrics is a prototype tested in a private environment that matches TRL3 through TRL4 compatible levels, according to [12]. This proof of concept solution demonstrates that the monitoring of SLA security metrics is possible and the analysis of a large number of monitored data can be made in real-time.

Also, we summarized the status of implementation and integration activities and reported the current coverage of the requirements that were located during the requirement analysis and design phases.

In the table below we present the list of objectives associated to the task T3.4 and report the outcomes which verify the benefits of the results achieved in this task in the entire duration of the project.

Sub-objective	**Achievements**
SO3.1 - Identify the requirements for monitoring the fulfillment of service level agreements in what concerns the SPECS measures of	We developed a comprehensive list of monitoring requirements that are able to cover from simple to complex SLA security monitoring scenarios. Using these requirements, we managed to propose a monitoring framework that can be used to monitor

interest	specific SLA agreements metrics. Moreover it can be extended to support even different SLA metrics, not only security oriented.
SO3.2 - Evaluate the appropriateness of the state-of-the-art services for SPECS monitoring	We evaluated the state-of-the-art solutions derived from research projects, standalone project and commercial solutions. Based on this analysis we identified the solutions that we can reuse and integrate into the SPECS solution. Based on this analyses we decided what components need to be designed and implemented from scratch to fulfill the requirements identified in **SO3.1**.
SO3.3 - Propose innovative monitoring services	Based on SO3.1 and SO3.2 outcomes we designed an innovative monitoring architecture that is able to **observe**, **collect**, **aggregate**, **filter** information about the targeted metrics. Moreover the monitoring solution is able to **notify** other specialized components in case of anomaly detection.
SO3.4 - Provide proof-of-concept open-source monitoring service	In the end we managed to develop all of the proposed monitoring services. More over the proof-of-concept is validated through a valid testing methodology (correctness and stress testing). In this way we can prove that the software code is stable enough to be used by third parties. All the monitoring services are released as open-source software repositories that can be used and integrated in other projects or solutions.

As illustrated, all Monitoring module components are completed and publicly available. The code is available on-line on the SPECS repository [14] and its description is provided with all information needed to install and correctly use it within the SPECS Platform.

7. Bibliography

[1] MongoDB NoSQL Distributed Database, https://www.mongodb.org/, last accessed 04.2016
[2] Mercurial Versioning system, https://www.mercurial-scm.org/, last accessed 04.2016
[3] MongoDB Query Language, https://docs.mongodb.org/manual/tutorial/query-documents/, last accessed 04.2016
[4] SPECS Monitoring Event Format, https://bitbucket.org/specs-team/specs-core-monitoring-event-hub - markdown-header-specs-messages
[5] https://bitbucket.org/specs-team/specs-monitoring-cloud-trust-protocol-server
[6] GO Programming Language, http://golang.org/, last accessed 04.2016
[7] https://github.com/cloudsecurityalliance/ctpd/blob/master/INSTALL
[8] https://bitbucket.org/specs-team/specs-monitoring-nmap/
[9] Chef technology, https://www.chef.io/, last accessed 04.2016
[10] MongoDB Replication Documentation, https://docs.mongodb.org/manual/replication/, last accessed 04.2016
[11] Gatling Load Testing Tool, http://www.gatling.io/, last accessed 04.2016
[12] EU Technology readiness levels (TRL) Horizon 2020 - WP 2014-2015 (http://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf)
[13] https://bitbucket.org/specs-team/specs-core-monitoring-nmap/downloads
[14] SPECS Bitbucket repository, http://bitbucket.org/specs-team/
[15] D. Petcu, S. Panica, B. Irimie, G. Macarie, On Security SLA-based Monitoring as a Service, EAI International Conference on Cloud, Networking for IoT systems, Edition Lecture Notes of ICST (LNICST) , Springer-Verlag , In Press
[16] Mazhar, A., Khan, S.U., Vasilakos, A.V.: Security in Cloud Computing: Opportu- nities and Challenges. Information Sciences 305, 357-383 (2015)