



## D4.4 Technical documentation of the ASAP subsystem



Software Workbench for Interactive, Time Critical and Highly self-adaptive Cloud applications

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 643963 (SWITCH project).

Start date of project: 01.02.2015. Duration: 36 months until 31.01.2018

<b>Due Date:</b>	31st July 2017
<b>Delivery:</b>	31st July 2017
<b>Lead Partner:</b>	UL
<b>Dissemination Level*:</b>	PU
<b>Type**:</b>	R
<b>Status:</b>	FINAL
<b>Approved:</b>	All partners
<b>Version:</b>	1.0.1

<b>*Dissemination Level</b>	
<b>PU</b>	Public
<b>CI</b>	Classified, information as referred to in Commission Decision 2001/844/EC.
<b>CO</b>	Confidential, only for members of the consortium (including the Commission Services)
<b>**Type</b>	
<b>R</b>	Document, report (excluding the periodic and final reports)
<b>DEM</b>	Demonstrator, pilot, prototype, plan designs
<b>DEC</b>	Websites, patents filing, press & media actions, videos, etc.
<b>OTHER</b>	Software, technical diagram, etc.

## Contributors

The contributors to this deliverable are:

Contributor	Role
Jernej Trnkoczy, Petar Kochovski, Vlado Stankovski	Editors
Vlado Stankovski, Jernej Trnkoczy, Petar Kochovski, Salman Taherizadeh, Sandi Gec, Uroš Paščinski, Polona Štefanič, Matej Cigale, Vlad Poenaru	Authors
Andrew Jones, George Suciu Jr.	Internal reviewers

## Document History

Version	Date	Description
V0.1	17.5.2017	TOC
V0.2	20.5.2017	First draft version
V0.3	21.5.2017	Formatted version
V0.4	30.6.2017	Contribution from all authors added
V0.5	7.7.2017	Incorporated changes proposed internally
V0.6	21.7.2017	Version for internal review
V0.7	24.7.2017	Incorporated changes suggested by CU
V0.8	25.7.2017	Incorporated changes suggested by BEIA
V1.0	27.7.2017	Minor updates; candidate version for delivery to European Commission
V1.0.1	30.7.2017	Minor updates; final version for delivery to European Commission

**Keywords:** Autonomous system adaptation platform, monitoring, adaptation, performance diagnosis, knowledge base, time series database, Cloud, runtime control, time-critical applications

# Table of Contents

<b>1</b>	<b>Executive Summary.....</b>	<b>5</b>
<b>2</b>	<b>Introduction .....</b>	<b>5</b>
<b>3</b>	<b>ASAP architecture.....</b>	<b>7</b>
<b>4</b>	<b>Detailed component specification.....</b>	<b>8</b>
4.1	Monitoring server .....	8
4.1.1	Functionality.....	8
4.1.2	API description.....	9
4.1.3	Developed software .....	11
4.2	Monitoring Agent .....	12
4.2.1	Functionality.....	13
4.2.2	Developed software .....	13
4.3	Time Series DataBase.....	14
4.3.1	Functionality.....	14
4.3.2	API description.....	14
4.3.3	TSDB format .....	16
4.4	Alarm trigger .....	17
4.4.1	Functionality.....	18
4.4.2	API description.....	18
4.4.3	Developed software .....	21
4.5	Self-adapter Decision Maker .....	23
4.5.1	Functionality.....	23
4.5.2	Learning Classifier System.....	23
4.5.3	Generated rules used by the "Self-adapter Decision Maker" .....	26
4.6	Self-adapter Setup&Control .....	26
4.6.1	Functionality.....	26
4.6.2	API description.....	27
4.6.3	Developed software .....	35
4.7	Performance diagnose Model Generator .....	35
4.7.1	Functionality.....	35
4.7.2	Time and space complexity improvement.....	36
4.8	Knowledge Base.....	37
4.8.1	Functionality.....	37
4.8.2	API description.....	37
4.8.3	Developed software .....	39
<b>5</b>	<b>Testbed description .....</b>	<b>40</b>
<b>6</b>	<b>Videoconferencing use-case .....</b>	<b>41</b>
6.1	Use case description .....	41
6.2	Experiment description.....	43
6.2.1	CPU consumption.....	43
6.2.2	Bandwidth usage.....	44
6.2.3	PSNR .....	45
6.2.4	Frame latency .....	46
6.2.5	Service start-up time .....	47
<b>7</b>	<b>File Upload use-case .....</b>	<b>48</b>
7.1	Use case description .....	48
7.2	Experiment description.....	48
7.2.1	Metrics for the QoS model .....	48

7.2.2 Measurements..... 50

7.2.3 QoS model ..... 52

**8 Summary ..... 53**

8.1 Software functionality in public releases..... 53

8.2 Innovation..... 53

**9 Bibliography..... 54**

**Abbreviations..... 55**

# 1 Executive Summary

Deliverable D4.4 Technical documentation of the ASAP subsystem is a logical continuation of the previous WP4 deliverables D4.1 [1], D4.2 [2] and D4.3 [3]. We present the overall architecture and detailed description of each individual component. Their APIs and algorithms used are given. The deliverable also contains the explanation of the ASAP functionality on two concrete time critical – File Upload and VaaS (Videoconferencing as a Service) use cases, that were developed by UL particularly for the purpose of demonstration.

Deliverable D4.4 therefore contains the following sections: (1) A short overview of requirements on the ASAP subsystem, (2) the upgraded status and detailed specification of the ASAP subsystem components, (3) testbed description, and (4) experiment descriptions with container-based applications – File Upload and VaaS. The applications were used to gather data for the development of the learning algorithm and the strategies for self-adaptation.

The planned work in the 3rd project year, under the WP4 included testbed enlargement, experimentation with the developed ASAP services, further integration with the SIDE GUI. Experiments with the three industrial SWITCH demo applications are ongoing and will be reported in other deliverables by the end of the project. Future experiments are geared towards: fine tuning of the learning algorithms and strategies (e.g. incremental learning from monitoring data), improvement of the ontology, which is used for the SWITCH Knowledge Base, the development of various multi-tier (e.g. cloud-edge-fog) application design patterns and similar. Particular focus is given to the possibility to manage Non-Functional Requirements from the SWITCH Interactive Development Environment. For example, the HTTPS service of the File Upload application can be annotated to execute in an edge-computing mode, which would address the requirement for fast upload. The goal is to provide means for specification of multi-tier application patterns in the SIDE GUI, which can then be used by the software engineer when developing time-critical component-based applications.

# 2 Introduction

The SWITCH workbench consists of three subsystems, each taking primary responsibility for one of the three key parts of the time-critical application lifecycle on cloud infrastructure: development, provisioning and adaptation. The ASAP subsystem is responsible for the adaptation functionality. The notion of ‘time-critical application’, as expressed in the original SWITCH description of work, refers specifically to applications that must satisfy one or more response-time constraints imposed on some subset of the application’s constituent components, e.g. to respond within a certain time window to new sensor data, or to minimise the latency in video streaming application. The adaptation of the cloud-based time-critical applications is needed for different reasons (e.g. varying number of users, component failures etc.), nevertheless its core objective is to continuously satisfy the QoE requirements of the application end users in the constantly changing cloud environment. The goal is not to provide the highest possible experience, but rather to ensure stable performance within strict boundaries in the most cost-effective manner feasible. The adaptation for a time-critical application therefore requires careful monitoring of the host infrastructure and applications and based on this monitoring information the system should perform appropriate adaptation strategies to satisfy the requirements towards QoE perceived by the end users. This requires particularly robust architecture, which must take into account all the parameters relevant for the QoE, including the number of users and/or requests it will be receiving, the geographical location of the clients, the computational and space complexity of the problem etc.

Due to the complexity of the adaptation problem, the adaptation scope in this project was narrowed down to **event-driven adaptation**. The role of event-driven adaptation is to find appropriate cloud resources and instantiate application services that can satisfy the requested QoE level, each time a new event (i.e. service request) occurs. The events therefore represent the need for new resources due to new service requests. For example, in MOG application [4], when a new live-event (e.g. 2017 FIFA World Cup in India) has to be covered, the application consisting of several services (input distributor, proxy transcoder, switcher etc.) has to be deployed. This requires a new set of containers to be deployed, however the question is where these

containers need to be started and what computational resources need to be allocated to them. Obviously, this depends on the event context (e.g. the location of the event to cover, the number and resolution of video cameras, etc.) and on the strategy that needs to be fulfilled (e.g. low production cost or highest video quality etc.). To determine appropriate VMs where Docker containers will be instantiated, ASAP couples event context with an appropriate performance model, which represents the strategy selected by the service user.

Each time an event needs to be served, the following phases need to be performed: (1) context capturing, (2) decision making, (3) container deployment, (4) actual service that is serving the event (Figure 2-1) and (5) the destruction of the service when the event is over.

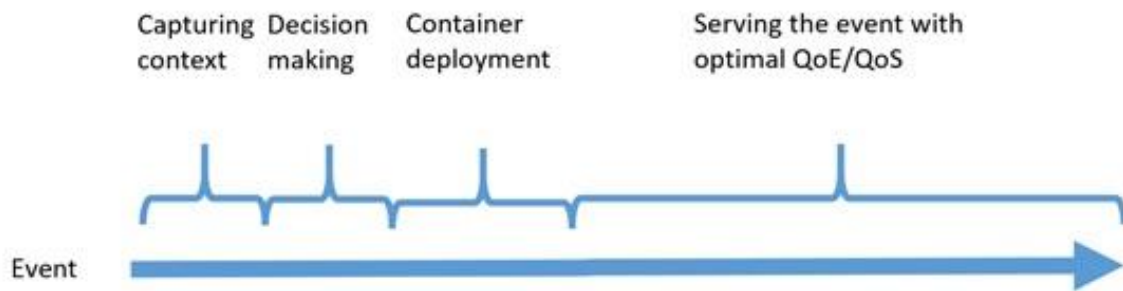


Figure 2-1 Phases in the ASAP subsystem adaptation to time-critical events process.

This approach, of course, assumes that (1) serving each and every time-critical event requires specific tailoring of the application components and choice of the Cloud provider (based on the identified context), and (2) the services serving a time-critical event do not require further adaptation during the application runtime. The only runtime adaptation of the application covered by ASAP is due to component failure. In this case, a new instantiated container replaces the failed service. With our event-driven approach it should be possible to reduce the operational cost of the service, since there will be no idle hosts at any time. Additionally, elasticity to a varying number of users will be achieved automatically.

The role of ASAP subsystem is therefore to select the appropriate infrastructure and deploy the application containers such that during the runtime the appropriate QoE levels will be achieved. During the application runtime (which serves the one-at-a-time event), the application is non-intrusively monitored, and in case of failure detection a new component is immediately instantiated, the users are notified and their application can be re-launched. It is important to note that due to complexity growing exponentially with the number of services composing the application (i.e. the number of containers) ASAP took the approach of “model per component” – this means that the performance models are not made for the entire application, but rather for each individual component separately. In other word - we assume that all of the components of the application are running on the same VM and they share resources of this VM. It is also important to note that we implemented only one model, representing the “best QoE” strategy, in which the goal of the VM selection is to provide the best possible Quality of Experience to the end users.

In the process of adaptation, the decision-making phase is particularly important. In ASAP decision-making represents the selection of the appropriate VM where the service instance will be instantiated for a specific event. The decision-making phase is composed of two steps: 1) filtering and 2) selection based on monitored data and decision model representing the strategy. Filtering is necessary to reduce a huge number of potential host VMs to a smaller subset of “suitable” VMs. The filtering is based on user-specified requirements and provided application context. These are provided in a form of rules (e.g. the VM has to have more than 1GB RAM and more than 2vCPU cores and should be located on the same continent as users) which are then checked against all the potential host VMs descriptions in the Knowledge Base. The filtering is needed to reduce the number of potential candidates to only a few VMs, where monitoring can be started in real-time and the best of the VMs according to the decision model is selected.

In order to support the above-mentioned event-driven adaptation a suitable ASAP architecture, consisting of several ASAP components was established. The architecture and the detailed specification of individual components are described in the following chapters.

### 3 ASAP architecture

The ASAP subsystem has several components that are required for its operation. In Figure 3-1 we present a high-level overview of the components and their interconnections.

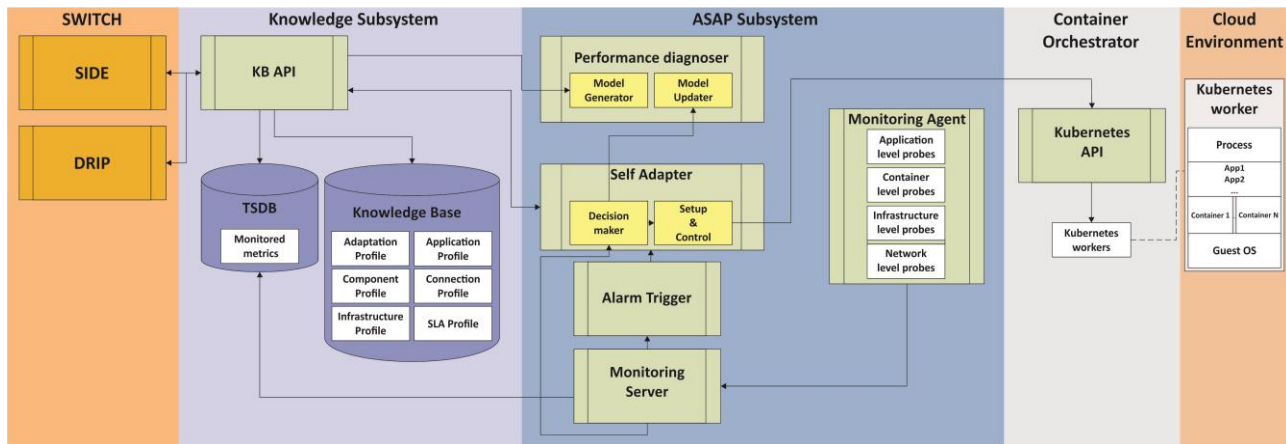


Figure 3-1 Architecture of the ASAP subsystem

Furthermore, the architecture of ASAP can be divided in roughly four different subsystems: 1) monitoring 2) adaptation logic 3) setup & control and 4) data storage.

The monitoring subsystem is responsible for monitoring the current state of the system. It represents the baseline of any autonomous system. In SWITCH, the monitoring system consists of Monitoring Server, Monitoring Agent and Alarm-Trigger. These will be specified in sections **Error! Reference source not found.** and 4.4. The data collected by the monitoring system has three functions. First, it is used as input to real-time Alarm-Trigger (section 4.4) component that is constantly monitoring the state of the application and raises an alarm in case of component failures. Second, the collected monitoring data is used by the Model Generator and Model Updater that analyse the monitored application and provides the up-to-date model representing a specific user-defined strategy. Third, the monitoring data is used by the Decision Maker component, which decides on which VM to instantiate the service(s) serving the event.

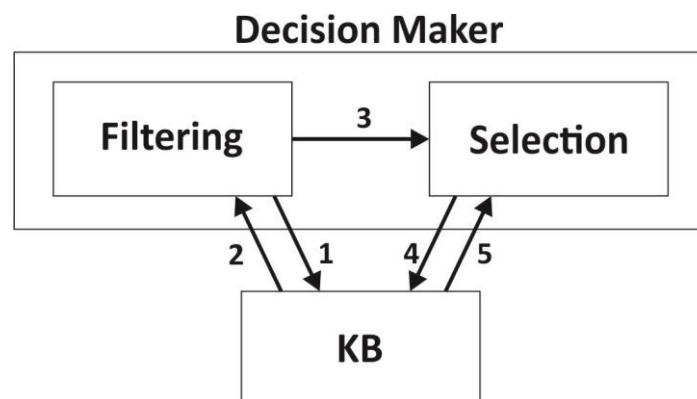


Figure 3-2 Decision making process

The adaptation logic is responsible for calculating the model(s) representing user-defined strategy(ies) for the adaptation, and based on the calculated model(s) performing the decision where to instantiate the service that will serve particular event. It is composed of three components. Model Generator (section **Error! Reference source not found.**) generates the models of the application according to the user-selected strategy. Model Updater (section **Error! Reference source not found.**) is using the monitoring data in order to decide when to update the model, which is needed due to the changes in the environment. Decision Maker (section **Error! Reference source not found.**) is the component that selects the appropriate VM where the services serving

particular event will be instantiated. The process (Figure 3-2) of selection is composed of two steps, filtering and selection. The filtering is based on the measurements done during the infrastructure level real time monitoring process. Furthermore, geolocation might be used as additional filtering constraint, because it may provide an approximation for network based metrics that influence the QoS [5] [6]. As a result, a subset of the closest VMs to the user is created. The selection follows the filtering process. It obtains the QoS model and measurements such as the jitter, latency, packet loss and estimated throughput. The measurements and the QoS model are used to rank the VMs and select the best VM where the service can be instantiated.

Setup & control part of the ASAP subsystem (section 3) is responsible for the actual services deployment. When the adaptation logic determines the virtual (or physical) machine where the services need to be deployed, the Setup & Control component calls appropriate Kubernetes Cluster API and initiates the starting of the appropriate Kubernetes Pods in which the containers representing the services are running. When the event is over this component needs to destruct the created services.

The Data Storage of ASAP is composed of TSDB and the Knowledge Base (section 4.8). While TSDB stores the real-time monitoring, metrics values, the KB is responsible for inter-entity relationships, constraints and complex data analysis mechanisms.

## 4 Detailed component specification

### 4.1 Monitoring server

In the SWITCH project, there is one Monitoring Server for each application. Monitoring Server is a component that receives measured metrics from the Monitoring Agent [7]. Moreover, for each application environment, there is one Monitoring Agent, which consists of one StatsD and one Monitoring Adapter. The Monitoring Agent includes two entities: (I) StatsD and (II) Monitoring Adapter. Monitoring Adapter is the actual component that aggregates individual metrics' values via StatsD and submits all monitoring data to the Monitoring Server. The Monitoring Server can recognize all running containers based on the measured metrics. It receives all measured metrics associated with the running containers and then stores them into a database.

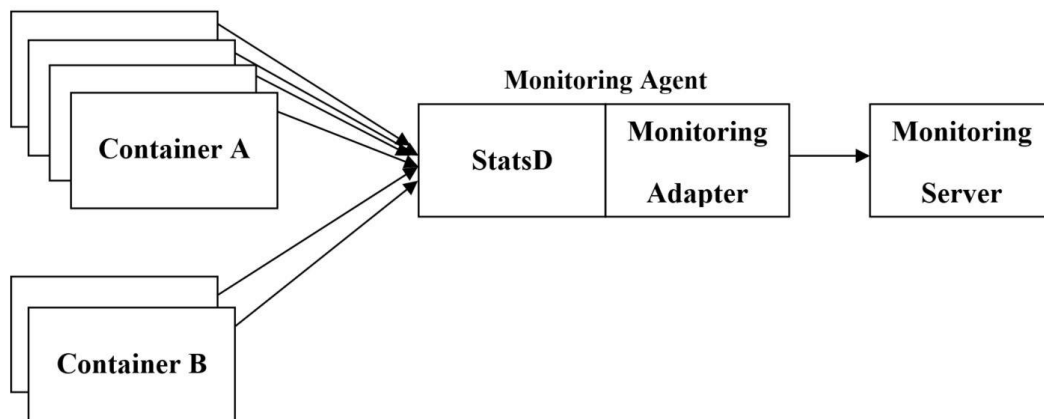


Figure 4-1 Overview of the Monitoring System for the SWITCH workbench

#### 4.1.1 Functionality

In order to develop a monitoring system to measure metrics, JCatascopia [8], which is a VM-based monitoring system has been chosen as baseline technology, which was extended in this work to fulfil the requirements of containerized applications, and having Alarm-Trigger component. Monitoring Server is the key component, which obtains the monitoring data transmitted by the Monitoring Agent in the application environment. In the application environment, there could be more than one type of component as an application may have different containerized services. For example, in the BEIA use case, there are two virtualized components: the CC Server



and the DB Server. Therefore, the Monitoring Server receives all monitoring data that could consist of different monitoring types. For example, for the containers with type A, the monitoring data could be the CPU and memory resource utilisation, and for the containers with type B, the monitoring data could be the response time of the associated service. Therefore, the Monitoring Server receives all the monitoring data that belongs to running containers and the stores these measured values into a database. Each must contain the information about the IP address of the container, metric name and metric value.

Before the deployment of containers, which are providing the service, the Monitoring Server should be instantiated and ready to register the containers and receive measured data. Each user of the SWITCH workbench can have a Monitoring Server for its own running application wherever it is required. To this end, the following command can be executed by container management systems such as Kubernetes.

```
docker run -p 8080:8080 -p 4242:4242 -p 4245:4245 -p 7199:7199 -p 7000:7000 -p 7001:7001 -p 9160:9160 -p 9042:9042 -p 8012:8012 -p 61621:61621 salmant/monitoring_server_container_image
```

After running the Monitoring Server, containers can register themselves in it via the Monitoring Agent and start working. Running containers can be seen in the following Web page:

<http://monitoringServerIP:8080/JCatascopia-Web/home.jsp>

### 4.1.2 API description

Monitoring Server provides various APIs accessible by other entities in the SWITCH project to fetch different types of monitoring data. These APIs have been described as follows:

REST endpoint	<b>Fetch the list of Containers</b>
Method	<b>GET</b>
Description	<b>This API gives the list of monitored containers and their information including IP, status and name.</b>
Input parameters	<b>None</b>
URL template	<b><a href="http://{monitoringServerIP}:8080/JCatascopia-Web/restAPI/agents/">http://{monitoringServerIP}:8080/JCatascopia-Web/restAPI/agents/</a></b>
Example of URL	<b><a href="http://194.249.0.192:8080/JCatascopia-Web/restAPI/agents/">http://194.249.0.192:8080/JCatascopia-Web/restAPI/agents/</a></b>
Example of result	<pre>{   "agents": [     {       "agentID": "99470131e7c44eeb8e2609d7f25e66ea",       "agentIP": "194.249.1.46",       "status": "UP",       "agentName": "194.249.1.46"     },     {       "agentID": "ce8bca7033a14198a2611046d9919e33",       "agentIP": "194.249.1.28",       "status": "UP",       "agentName": "194.249.1.28"     }   ] }</pre>

REST endpoint	<b>Fetch the list of measured metrics associated with a specified container</b>
Method	<b>GET</b>
Description	<b>This API gives the list of measured metrics (and their associated information e.g. metric ID, metric name, metric type, metric group and so on) associated with a container specified via an ID.</b>

Input parameters	<b>Container ID</b>
URL template	<b>http://[monitoringServerIP]:8080/JCatascopia-Web/restAPI/agents/{agentID}/availableMetrics</b>
Example of URL	<b>http://194.249.0.192:8080/JCatascopia-Web/restAPI/agents/99470131e7c44eeb8e2609d7f25e66ea/availableMetrics</b>
Example of result	<pre>{   "metrics": [     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:arch",       "name": "arch",       "units": "",       "type": "STRING",       "group": "StaticInfo"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:btime",       "name": "btime",       "units": "",       "type": "STRING",       "group": "StaticInfo"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:cpuOwait",       "name": "cpuOwait",       "units": "%",       "type": "DOUBLE",       "group": "CPU"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:cpuldle",       "name": "cpuldle",       "units": "%",       "type": "DOUBLE",       "group": "CPU"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:cpuNum",       "name": "cpuNum",       "units": "",       "type": "STRING",       "group": "StaticInfo"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:cpuSystem",       "name": "cpuSystem",       "units": "%",       "type": "DOUBLE",       "group": "CPU"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:cpuTotal",       "name": "cpuTotal",       "units": "%",       "type": "DOUBLE",       "group": "CPU"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:cpuUser",       "name": "cpuUser",       "units": "%",       "type": "DOUBLE",       "group": "CPU"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:diskFree",       "name": "diskFree",       "units": "MB",       "type": "LONG",       "group": "Disk"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:diskTotal",       "name": "diskTotal",       "units": "MB",       "type": "LONG",       "group": "Disk"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:diskUsed",       "name": "diskUsed",       "units": "%",       "type": "DOUBLE",       "group": "Disk"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:iotime",       "name": "iotime",       "units": "%",       "type": "DOUBLE",       "group": "DiskStats"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:memCache",       "name": "memCache",       "units": "KB",       "type": "INTEGER",       "group": "Memory"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:memFree",       "name": "memFree",       "units": "KB",       "type": "INTEGER",       "group": "Memory"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:memSwapFree",       "name": "memSwapFree",       "units": "KB",       "type": "INTEGER",       "group": "Memory"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:memSwapTotal",       "name": "memSwapTotal",       "units": "KB",       "type": "INTEGER",       "group": "Memory"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:memTotal",       "name": "memTotal",       "units": "KB",       "type": "INTEGER",       "group": "Memory"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:memUsed",       "name": "memUsed",       "units": "KB",       "type": "INTEGER",       "group": "Memory"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:memUsedPercent",       "name": "memUsedPercent",       "units": "%",       "type": "DOUBLE",       "group": "Memory"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:netBytesIN",       "name": "netBytesIN",       "units": "bytes/s",       "type": "DOUBLE",       "group": "Network"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:netBytesOUT",       "name": "netBytesOUT",       "units": "bytes/s",       "type": "DOUBLE",       "group": "Network"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:netPacketsIN",       "name": "netPacketsIN",       "units": "packets/s",       "type": "DOUBLE",       "group": "Network"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:netPacketsOut",       "name": "netPacketsOut",       "units": "packets/s",       "type": "DOUBLE",       "group": "Network"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:os",       "name": "os",       "units": "",       "type": "STRING",       "group": "StaticInfo"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:readkbps",       "name": "readkbps",       "units": "KB/s",       "type": "DOUBLE",       "group": "DiskStats"     },     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:writekbps",       "name": "writekbps",       "units": "KB/s",       "type": "DOUBLE",       "group": "DiskStats"     }   ] }</pre>

REST endpoint	<b>Fetch the last value of a metric</b>
Method	<b>GET</b>
Description	<b>This API gives the last value of the metric specified via metric ID.</b>
Input parameters	<b>Metric ID</b>
URL template	<b>http://[monitoringServerIP]:8080/JCatascopia-Web/restAPI/metrics/&lt;metricID&gt;</b>
Example of URL	<b>http://194.249.0.192:8080/JCatascopia-Web/restAPI/metrics/99470131e7c44eeb8e2609d7f25e66ea:cpuTotal</b>
Example of result	<pre>{   "metricID": "99470131e7c44eeb8e2609d7f25e66ea:cpuTotal",   "values": [     {       "metricID": "99470131e7c44eeb8e2609d7f25e66ea:cpuTotal",       "name": "cpuTotal",       "units": "%",       "type": "DOUBLE",       "group": "CPU"     }   ] }</pre>

	<pre>"value":"0.30000000000000004", "timestamp":"10:47:38"]}</pre>
--	--

REST endpoint	<b>To fetch the list of Virtual Clusters</b>
Method	<b>GET</b>
Description	<b>This API gives the ID and name of all created Virtual Clusters. A Virtual Cluster represents a group of containers that are providing together the same service. This concept comes when the SWITCH suggests the horizontal scalability of running instances. Therefore, if three running container instances are providing the same service, these three containers make one Virtual Cluster.</b>
Input parameters	<b>None</b>
URL template	<b>http://”monitoringServerIP”:8080/JCatascopia-Web/restAPI/subscriptions/</b>
Example of URL	<b>http://194.249.0.192:8080/JCatascopia-Web/restAPI/subscriptions/</b>
Example of result	<pre>{   "subs": [     {       "subID": "32022e7042c749c79735ccb95409cca6",       "subName": "test2"     },     {       "subID": "12acfaed2fcc47afae4650da56140e8e",       "subName": "test"     }   ] }</pre>

REST endpoint	<b>Fetch the list of containers in a Virtual Cluster</b>
Method	<b>GET</b>
Description	<b>This API gives the list of containers (their associated information e.g. ID and IP) which are the members of a Virtual Cluster specified via cluster ID.</b>
Input parameters	<b>Cluster ID</b>
URL template	<b>http://”monitoringServerIP”:8080/JCatascopia-Web/restAPI/subscriptions/&lt;clusterID&gt;/agents</b>
Example of URL	<b>http://194.249.0.192:8080/JCatascopia-Web/restAPI/subscriptions/12acfaed2fcc47afae4650da56140e8e/agents</b>
Example of result	<pre>{   "agents": [     {       "agentID": "9761b206be714c8289668e49c11a3ce7",       "agentIP": "194.249.1.46"     },     {       "agentID": "b0dbdf77ccd34937b556f75925ffac2c",       "agentIP": "194.249.1.28"     }   ] }</pre>

### 4.1.3 Developed software

In the SWITCH project, the Monitoring Server has been containerized to be deployed automatically by container management systems such as Kubernetes. To this end, two files (“Dockerfile” and “start.sh”) which include associated code to prepare the containerized Monitoring Server are as follows:

```
FROM poklet/cassandra:latest
#-----SETUP OF THE JCATASCOPIA MONITORING AGENT-----
RUN yum install -y wget
RUN yum install -y tar
#change the workdir
WORKDIR /root
RUN wget https://www.dropbox.com/s/lodwx237u4fets/JCatascopia-Server-0.0.2-SNAPSHOT.tar.gz
RUN tar xvfz JCatascopia-Server-0.0.2-SNAPSHOT.tar.gz
RUN export TERM=xterm
RUN cp -r JCatascopia-Server-0.0.2-SNAPSHOT/JCatascopiaServerDir /usr/local/bin/
RUN chmod +x /etc/init.d/JCatascopia-Server
```

```

RUN mv -f JCatascopia-Server-0.0.2-SNAPSHOT/JCatascopia-Server-CELAR /etc/init.d/JCatascopia-Server
RUN wget http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.55/bin/apache-tomcat-7.0.55.tar.gz
RUN tar xvfz apache-tomcat-7.0.55.tar.gz -C /usr/share/
RUN mv /usr/share/apache-tomcat-7.0.55 /usr/share/tomcat/
RUN wget https://www.dropbox.com/s/gr4celempy7sybu/JCatascopia-Web.war
RUN cp JCatascopia-Web.war /usr/share/tomcat/webapps/
#RUN java -jar JCatascopia-Server-0.0.2-SNAPSHOT/JCatascopiaServerDir/JCatascopia-Server-0.0.2-SNAPSHOT.jar
JCatascopia-Server-0.0.2-SNAPSHOT/JCatascopiaServerDir /var/lock/JCatascopia-Server-lock &
#-----EXPOSE THE PORTS AND START THE SCRIPT THAT SHOULD START TOMCAT-----
#expose the ports of the container
EXPOSE 8080 4242 4245 7199 7000 7001 9160 9042 8012 61621
#configure and start components with an external script
COPY start.sh /root/start.sh
RUN chmod 777 /root/start.sh
ENTRYPOINT ["/root/start.sh"]

```

Figure 4-2 Dockerfile

```

#!/bin/bash
sed -i 's/securerandom.source=file:/dev/random/securerandom.source=file:/dev/urandom/g'
/usr/lib/jvm/jre/lib/security/java.security
sh /usr/share/tomcat/bin/startup.sh
cd /etc/init.d/
cassandra start
exec java -jar /root/JCatascopia-Server-0.0.2-SNAPSHOT/JCatascopiaServerDir/JCatascopia-Server-0.0.2-SNAPSHOT.jar /root/JCatascopia-Server-0.0.2-SNAPSHOT/JCatascopiaServerDir /var/lock/JCatascopia-Server-lock

```

Figure 4-3 Instructions' script "start.sh"

## 4.2 Monitoring Agent

In the base architecture of JCatascopia, there is a concept called "Monitoring Probe". In this architecture, Monitoring Probes are in charge of gathering monitoring metrics, which are then aggregated by the individual, associated Monitoring Agent. These probes are small java classes, which are loaded by the agent and are used to pull data. The base architecture of JCatascopia has two properties, which makes hard to be integrated with the containers used in the SWITCH project. Firstly, JCatascopia is written in Java and has no client SDKs for other languages. Therefore, unfortunately the container requires a lot of memory and many packages to be able to run a JVM. The second is the fact that the design of the Monitoring Probes is quite static and it is hard to re-configure the probe or the agent under which it runs to allow multiple containers being monitored by the same probe/agent. The main idea of a new design for the Monitoring Agent in the SWITCH project is using only one entity as an intermediary, which is able to receive monitoring data from containers through UDP-based StatsD protocol available for many programming languages and then forwarded to the Monitoring server. The protocol is available on GitHub:

<https://github.com/etsy/StatsD>

This new architecture of the monitoring system has been created due to the following criteria:

- Only one agent per application environment: a single agent is needed in the environment, which can receive the monitoring data (measured data) from a large number of containers running to provide the service.
- Easy to configure containerized services: Using this new architecture of the monitoring system, there is only one Monitoring Agent to be prepared, each container just needs to know the IP address of this Monitoring Agent (shown in the following figure) as a parameter. This can be performed as an environmental variable when a container is launched. Besides that, in order to forward the monitoring data, the containerized Monitoring Agent only needs to know the IP address of the Monitoring Server.
- Low footprint and easy to integrate: As there are client SDKs available and plugins for software like NGINX, MySQL and so on, the footprint of new monitoring architecture in any container is very low

(usually the order of tens of kilobytes). The monitoring part runs in-process so containers remain simple and there is no need for complicated process management.

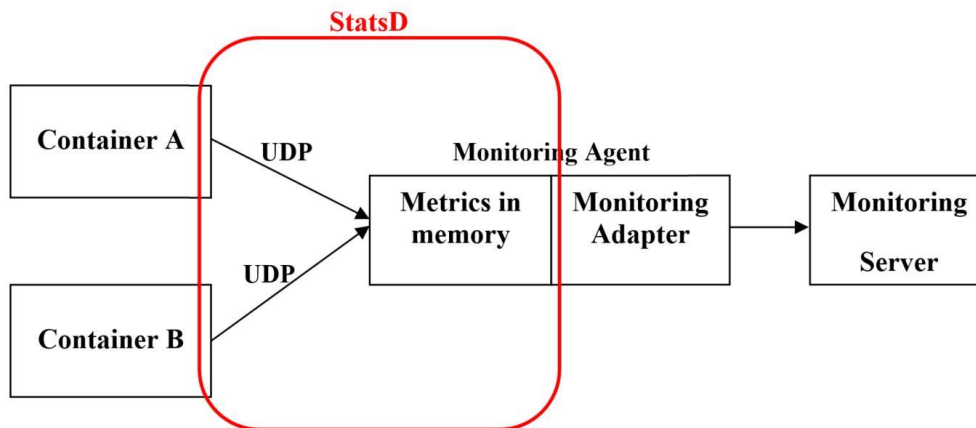


Figure 4-4 UDP-based StatsD as simple way to send measured metrics.

### 4.2.1 Functionality

In the SWITCH project, the JCatascopia monitoring system has been extended additionally to also measure container-level metrics. This monitoring system is appropriate for a large number of monitored containers. This monitoring solution, which consists of the Monitoring Agent, requires certain information to be passed for each measured metric through StatsD protocol between containers and the Monitoring Agent. To this end, a format for the metric key has been defined as follows:

```
eu.switch.<application-environment>.<container-id>.<container-ip>.<metric-group-name>.<metric-name>.<units>
```

The considered fields in the above-mentioned format are:

- `eu.switch.<application-environment>` - This parameter helps the Monitoring Adapter filter measured metrics which are not supposed to be sent to the Monitoring Server.
- `<container-id>` - This parameter represents the container ID. It is suggested to use a randomly generated string, which includes different digits to prevent duplication issue (e.g. UUID).
- `<container-ip>` - This parameter shows the container IP. As StatsD keys use dots to separate each part, the IP address will be converted by dashes instead of dots (e.g. 127.0.0.1 would be converted to 127-0-0-1).
- `<metric-group-name>` - This parameter represents the metric group. There are some reserved group names, which have been used for the assigned monitoring values (e.g. 'StatsInfoProbe', 'CPUProbe', 'DiskStatsProbe', 'NetworkProbe', 'MemoryProbe').
- `<metric-name>` - This parameter shows the metric name in the monitoring system. There are some reserved names for monitoring metrics which have been used for the assigned monitoring values (e.g. 'cpuTotal', 'cpuUser', 'cpuSystem', 'cpuIdle', 'cpuIOwait', 'memTotal', 'memFree').
- `<units>` - This parameter represents the metric units to display in the monitoring system. This is an optional parameter.

### 4.2.2 Developed software

The Monitoring Agent has been containerized and uploaded in the Docker Hub on beia/monitoring\_agent. In order to start this container, two environment variables should be set as following:

- `MONITORING_SERVER` - This parameter should be initialized as the address of the Monitoring Server.
- `MONITORING_PREFIX` - This parameter should be initialized as the prefix of all the metric keys to be processed and sent forward by the Monitoring Agent (For example: "eu.switch.beia").

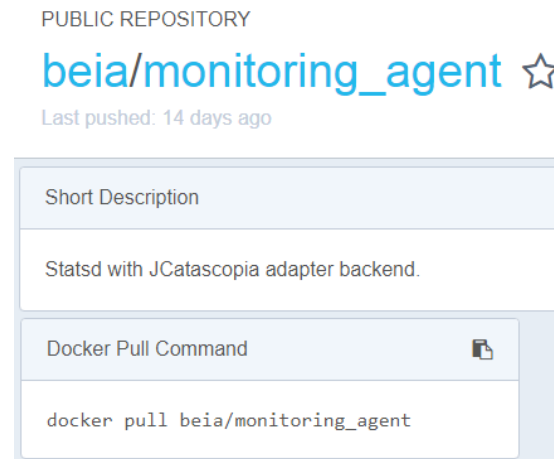


Figure 4-5 Container image for the Monitoring Agent.

### 4.3 Time Series DataBase

A Time Series Database (TSDB) has been employed in the SWITCH project for storing all measured values indexed by time. This TSDB used in the SWITCH project per application environment has been implemented by the Apache Cassandra Server, which is a distributed database, designed to manage huge amounts of time-ordered monitoring data. In essence, the data streams coming from the Monitoring Agent are received by the Monitoring Server and then stored in the TSDB that is capable of handling large amount of monitoring data [9].

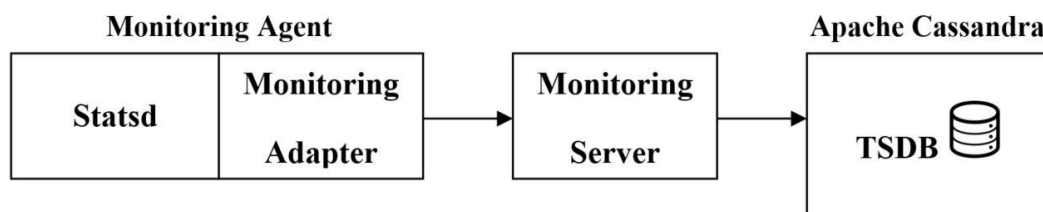


Figure 4-6 TSDB as a part of the Monitoring System in the SWITCH project.

#### 4.3.1 Functionality

A TSDB is a software component that is optimized for handling a large amount of time series monitoring data indexed by time. In the SWITCH project, a NoSQL database system called Apache Cassandra has been chosen for these types of TSDB challenges. The Apache Cassandra's data model is an appropriate fit for handling monitoring data as the measured data in sequence regardless of datatype or size.

#### 4.3.2 API description

According to the Cassandra documentations, the API to Cassandra is “Cassandra Query Language” (CQL). In order to apply CQL, it is needed to connect to the database via one of the following options:

- *cqlsh* tool
- Client driver for Cassandra (as shown in the following code used in the SWITCH project as an example).

```

import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.ResultSet;
import com.datastax.driver.core.Row;
import com.datastax.driver.core.Session;
import com.datastax.driver.core.exceptions.InvalidQueryException;
import com.datastax.driver.core.exceptions.NoHostAvailableException;
import com.datastax.driver.core.policies.ConstantReconnectionPolicy;
  
```

```

import com.datastax.driver.core.policies.DowngradingConsistencyRetryPolicy;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.*;
import java.util.logging.Level;
import java.util.logging.Logger;
public class CassandraDB {
    private Session session;
    private static Logger logger;
    public static void main(String[] args) {
        // init app
        new CassandraDB(logger, "db", ".*.*.*", 9042);
    }
    public CassandraDB(Logger logger, String databaseName, String cassandraIP, int port) {
        this.logger = logger;
        try {
            final InetAddress ip = InetAddress.getByName(cassandraIP);
            initDatabase(ip, port, databaseName);
        } catch (UnknownHostException e) {
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public void initDatabase(InetAddress ip, int port, String databaseName) {
        try {
            //connect to cassandra cluster
            Cluster cluster = Cluster.builder().addContactPoints(ip.getHostAddress()).withCredentials
                ("*****_username", "*****_password").withPort(port).withRetryPolicy
                (DowngradingConsistencyRetryPolicy.INSTANCE).withReconnectionPolicy(new
                ConstantReconnectionPolicy(1000L)).build();
            session = cluster.connect(databaseName);
            System.out.println("Logged to keyspace: " + session.getLoggedKeyspace());
            // simple CQL query:
            ResultSet resultSet = this.session.execute("select * from agent_table limit 20;");
            Set<String> list = new HashSet<String>();
            // list query results
            for (Row row : resultSet) {
                final String id = row.getString("agentid");
                System.out.println("agentid: " + row.getString("agentip"));
                System.out.println("agentname: " + row.getString("agentname"));
                System.out.println("status: " + row.getString("status"));
                System.out.println("tags: " + row.getString("tags"));
                System.out.println("tstart: " + row.getUUID("tstart"));
                System.out.println("tstop: " + row.getUUID("tstop"));
                System.out.println("-----");
                list.add(id);
            }
        } catch (InvalidQueryException e) {
            if (e.getMessage().equals("Keyspace asapDB does not exist")) {
                Cluster cluster = Cluster.builder().addContactPoints("194.249.0.185").withCredentials
                    ("*****_username", "*****_password").build();
                Session session = cluster.connect("asapdb");
                session.getState();
            }
            else {
                e.printStackTrace();
            }
        } catch (NoHostAvailableException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Figure 4-7 Driver for Cassandra used in the SWITCH project.

### 4.3.3 TSDB format

The TSDB format in the Cassandra server for containers is as follows:

```
CREATE TABLE asapdb.agent_table (
  agentid text PRIMARY KEY,
  agentip text,
  agentname text,
  status text,
  tags text,
  tstart timeuuid,
  tstop timeuuid
) WITH bloom_filter_fp_chance = 0.01
  AND caching = '{"keys":"ALL", "rows_per_partition":"NONE"}'
  AND comment = ''
  AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy'}
  AND compression = {'sstable_compression': 'org.apache.cassandra.io.compress.LZ4Compressor'}
  AND dclocal_read_repair_chance = 0.1
  AND default_time_to_live = 0
  AND gc_grace_seconds = 864000
  AND max_index_interval = 2048
  AND memtable_flush_period_in_ms = 0
  AND min_index_interval = 128
  AND read_repair_chance = 0.0
  AND speculative_retry = '99.0PERCENTILE';
```

The TSDB format in the Cassandra server for metrics monitored by the monitoring system is as follows:

```
CREATE TABLE asapdb.metric_table (
  agentid text,
  metricid text,
  is_sub text,
  mgroup text,
  name text,
  type text,
  units text,
  PRIMARY KEY (agentid, metricid)
) WITH CLUSTERING ORDER BY (metricid ASC)
  AND bloom_filter_fp_chance = 0.01
  AND caching = '{"keys":"ALL", "rows_per_partition":"NONE"}'
  AND comment = ''
  AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy'}
  AND compression = {'sstable_compression': 'org.apache.cassandra.io.compress.LZ4Compressor'}
  AND dclocal_read_repair_chance = 0.1
  AND default_time_to_live = 0
  AND gc_grace_seconds = 864000
  AND max_index_interval = 2048
  AND memtable_flush_period_in_ms = 0
  AND min_index_interval = 128
  AND read_repair_chance = 0.0
  AND speculative_retry = '99.0PERCENTILE';
```

The TSDB format in the Cassandra server for measured values of metrics is as follows:

```
CREATE TABLE asapdb.metric_value_table (
  metricid text,
  event_date text,
  event_timestamp timeuuid,
  mgroup text,
  name text,
  type text,
  units text,
  value text,
  PRIMARY KEY ((metricid, event_date), event_timestamp)
) WITH CLUSTERING ORDER BY (event_timestamp DESC)
  AND bloom_filter_fp_chance = 0.01
  AND caching = '{"keys":"ALL", "rows_per_partition":"NONE"}'
```



```

AND comment = "
AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy'}
AND compression = {'sstable_compression': 'org.apache.cassandra.io.compress.LZ4Compressor'}
AND dclocal_read_repair_chance = 0.1
AND default_time_to_live = 0
AND gc_grace_seconds = 864000
AND max_index_interval = 2048
AND memtable_flush_period_in_ms = 0
AND min_index_interval = 128
AND read_repair_chance = 0.0
AND speculative_retry = '99.0PERCENTILE';

```

The TSDB format in the Cassandra server for virtual clusters is as follows:

```

CREATE TABLE asapdb.subscription_agents_table (
  subid text,
  agentid text,
  agentip text,
  PRIMARY KEY (subid, agentid)
) WITH CLUSTERING ORDER BY (agentid ASC)
AND bloom_filter_fp_chance = 0.01
AND caching = {'keys': "ALL", "rows_per_partition": "NONE"}
AND comment = "
AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy'}
AND compression = {'sstable_compression': 'org.apache.cassandra.io.compress.LZ4Compressor'}
AND dclocal_read_repair_chance = 0.1
AND default_time_to_live = 0
AND gc_grace_seconds = 864000
AND max_index_interval = 2048
AND memtable_flush_period_in_ms = 0
AND min_index_interval = 128
AND read_repair_chance = 0.0
AND speculative_retry = '99.0PERCENTILE';

```

## 4.4 Alarm trigger

The Alarm-Trigger is a component which checks the incoming monitoring data and notifies other components of the system (such as Self-Adapter and SWITCH GUI) when the application is experiencing abnormal behaviour based on the predefined thresholds for each monitoring metric [10]. As shown in Figure 4-8, the Alarm-Trigger is capable of triggering alerts if a threshold is violated. It periodically investigates all measured data and compares their values with the thresholds.

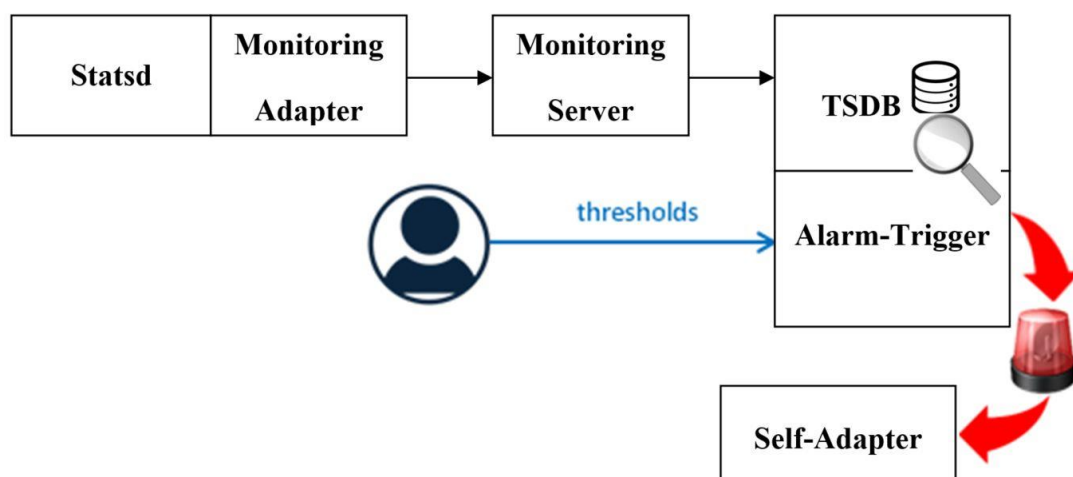


Figure 4-8 Alarm-Trigger as a part of the Monitoring System in the SWITCH project.

### 4.4.1 Functionality

For each metric at each level of monitoring (e.g. container or application), there is a threshold which is a value indicating the acceptable performance level for that metric. The user through the SWITCH GUI can set the threshold. The Monitoring Server, the TSDB and the Alarm Trigger are tightly coupled, i.e. running on the same virtual machine to optimize network bandwidth and computational resources needed for data distribution.

### 4.4.2 API description

Alarm-Trigger has two APIs. First gives the Alarm-Trigger the input in form of a YAML file which includes the list of metrics and their associated type, subtype, class\_id, label, data\_type, action, unit, period, minimum, maximum, warning\_value (warning threshold), warning\_operator, critical\_value (critical threshold) and critical\_operator.

REST endpoint	<b>Fetch the input (YAML file) for the Alarm-Trigger component</b>
Method	<b>GET</b>
Description	<b>This API gives the input for the Alarm-Trigger that includes all metrics and their associated thresholds to be periodically investigated.</b>
Input parameters	<b>None</b>
URL template	<b>http://”SIDEsideIP”:port/SWITCH/rest/side/getAlarmTriggerInput</b>
Example of URL	<b>http://194.249.0.192:8080/SWITCH/rest/side/getAlarmTriggerInput</b>
Example	<pre> metric1:   type: container_level   subtype: CPUProbe   class_id: "1ccba0cc92174ce788695cfc0a027b57"   properties:     label: cpuTotal     data_type: double     action: average     unit: %     period: 20   range:     minimum: 0.0     maximum: 100.0   alarm:     warning:       warning_value: 80.0       warning_operator: "&gt;="     critical:       critical_value: 100.0       critical_operator: "&gt;=" ##### metric2:   type: container_level   subtype: MemoryProbe   class_id: "1ccba0cc92174ce788695cfc0a027b57"   properties:     label: memTotal     data_type: double     action: average     unit: %     period: 20   range:     minimum: 0.0     maximum: 100.0   alarm:     warning:       warning_value: 80.0       warning_operator: "&gt;=" </pre>

```

critical:
  critical_value: 100.0
  critical_operator: ">="
#####
metric3:
  type: application_level
  subtype: Service1Probe
  class_id: "1ccba0cc92174ce788695cfc0a027b57"
  properties:
    label: AvgResponseTime
    data_type: double
    action: average
    unit: ms
    period: 20
  range:
    minimum: 0.0
    maximum: 1000.0
  alarm:
    warning:
      warning_value: 15.0
      warning_operator: ">="
    critical:
      critical_value: 30.0
      critical_operator: ">="
#####

```

The example mentioned above for a service (such as the BEIA CC Server) possibly including more than one container could be considered as follows:

if ((cpuTotal>=80%) or (memTotal>=80%) and (AvgResponseTime>=15ms)) then Send\_Alert\_To\_Self-Adapter in order to initiate new container instance.

In this regard, an experiment has been performed. As an example, assume that the workload includes two steps. In the first step, the number of incoming requests for the CC Server is slowly rising from 100 to 1600 requests per six seconds. In contrast, during the second step, workload density drops smoothly from 1600 to 100 requests. As shown in the following figure, the number of containers is increasing in the first step of the workload scenario and it is decreasing in the second step according to the number of arrived requests at execution time. Therefore, six new containers have been gradually started up during the experiment in the first step of this workload scenario. Red arrows in this figure indicate the time intervals when new containers have been initiated during the rising workload and purple arrows show the time intervals when the running containers have been removed from the cluster during the falling workload.

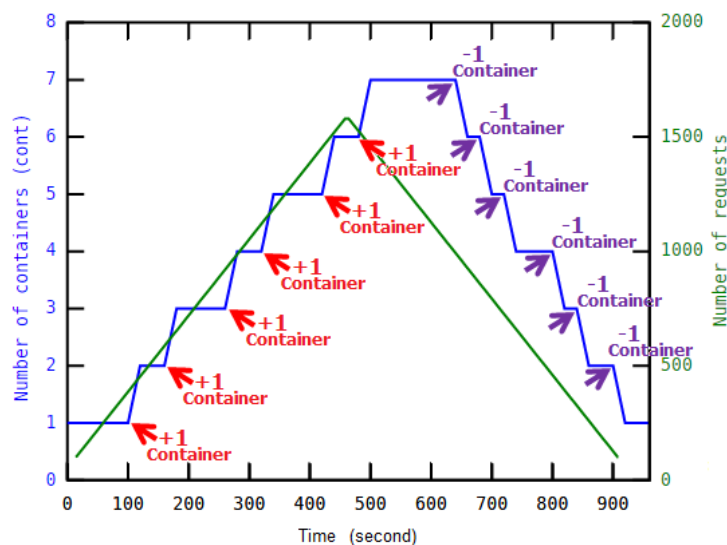


Figure 4-9 Number of containers vs number of requests.

Moreover, the following figure shows the average CPU utilization of all running containers in the cluster according to the changing workload at runtime. This figure implies that in seven monitoring intervals, enumerated from 1 to 7, the average CPU usage is over the CPU threshold which is 80% (defined in the YAML file mentioned above). However, the previous figure shows that there have been just six container initiations. Therefore, during one of these seven monitoring intervals—the 6th interval—the average response time of the application has been less than 15ms (the threshold for the response time of the CC Server that was defined in the YAML file mentioned above) indicating that the system is able to handle incoming requests without any performance problem yet. Hence, no container start up occurs for this interval, which is numbered six.

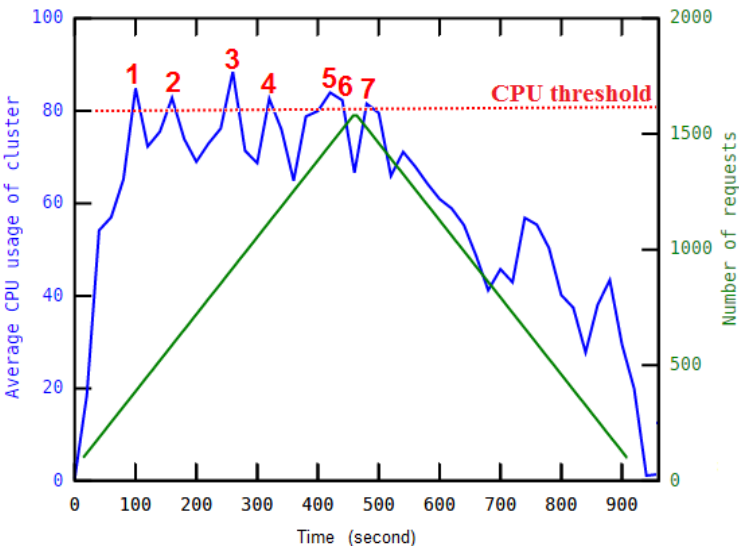


Figure 4-10 Average CPU usage of containers vs number of requests.

Furthermore, the following figure shows the average memory usage of all running containers in the cluster according to the changing workload at runtime for this experiment. From this figure, it is simply concluded that the conducted service in this experiment is not a memory-intensive service, as the average memory usage was almost steady in the conducted experiment—around ~28% of the whole memory.

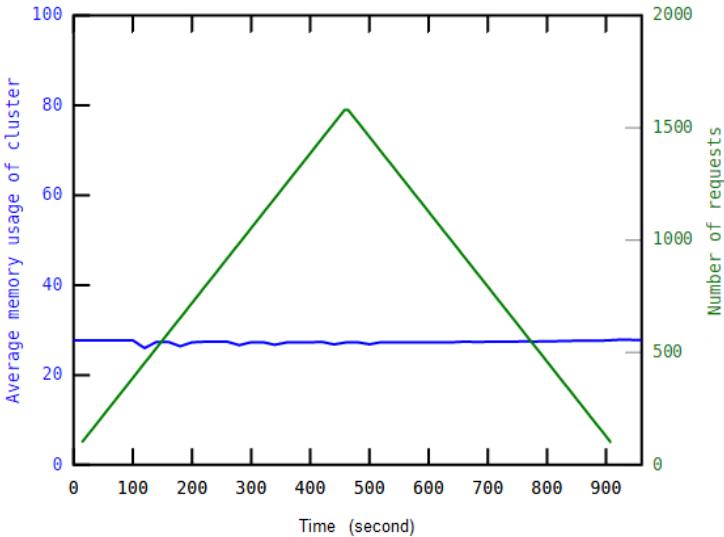


Figure 4-11 Average memory usage of the cluster.

Another API of the Alarm-Trigger sends notifications to the Self-Adapter if any threshold is reached.

REST endpoint	Send the notification to the Self-Adapter
Method	GET

Description	<b>This API sends alerts to the Self-Adapter if thresholds are violated at run-time.</b>
Input parameters	/
URL template	<b>http://Self-AdapterIP:Port/SWITCH/rest/asap/alarm_trigger_executed?DateTime=&lt;datetime&gt;&amp; label=&lt;label&gt;&amp;class_id=&lt;class_id&gt;&amp;value=&lt;value&gt;&amp;warning_or_critical=&lt;warning_or_critical&gt;</b>

### 4.4.3 Developed software

The following YAML code is considered as an example for the input of the Alarm-Trigger component. In this example, the first metric labelled “cpuTotal” represents the CPU resource utilisation of the machine on which the container is running. This value is between 0 and 100 percent. Metrics such as 'cpuTotal', 'cpuUser' and 'cpuSystem' have the same type called “subtype”. The data type for these metrics is “double” and their unit is percentage (%). Each metric has its own period for checking whether it is violated or not. For this example, if the total CPU utilization is over 80%, the Alarm-Trigger will notify the Self-Adapter by a warning alert. In addition, if the total CPU utilization is over 100%, the Alarm-Trigger will notify the Self-Adapter by a critical alert.

```
#####
metric1:
  type: vm_level
  subtype: CPUProbe
  class_id: "1ccba0cc92174ce788695cfc0a027b57"
  properties:
    label: cpuTotal
    data_type: double
    action: average
    unit: %
    period: 20
  range:
    minimum: 0.0
    maximum: 100.0
  alarm:
    warning:
      warning_value: 80.0
      warning_operator: ">="
    critical:
      critical_value: 100.0
      critical_operator: ">="
#####
metric2:
  type: vm_level
  subtype: CPUProbe
  class_id: "1ccba0cc92174ce788695cfc0a027b57"
  properties:
    label: cpuSystem
    data_type: double
    action: average
    unit: %
    period: 20
  range:
    minimum: 0.0
    maximum: 100.0
  alarm:
    warning:
      warning_value: 40.0
      warning_operator: ">="
    critical:
      critical_value: 50.0
      critical_operator: ">="
#####
metric3:
```

```

type: container_level
subtype: NetworkProbe
class_id: "1ccb0cc92174ce788695cfc0a027b57"
properties:
  label: netBytesIN
  data_type: double
  action: average
  unit: bytes/s
  period: 10
range:
  minimum: 0.0
  maximum: 30000000.0
alarm:
  warning:
    warning_value: 20000000.0
    warning_operator: ">="
  critical:
    critical_value: 25000000.0
    critical_operator: ">="
#####
metric4:
  type: application_level
  subtype: FrameProbe
  class_id: "1ccb0cc92174ce788695cfc0a027b57"
  properties:
    label: frameRate
    data_type: double
    action: average
    unit: frames/s
    period: 10
  range:
    minimum: 0.0
    maximum: 25.1
  alarm:
    warning:
      warning_value: 25.0
      warning_operator: "<="
    critical:
      critical_value: 24.9
      critical_operator: "<="
#####
metric5:
  type: P2P_level
  subtype: ConnectionQuality
  class_id: "1ccb0cc92174ce788695cfc0a027b57"
  properties:
    label: Jitter
    data_type: double
    action: maximum
    unit: ms
    period: 30
  range:
    minimum: 0.0
    maximum: infinite
  alarm:
    warning:
      warning_value: 0.2
      warning_operator: ">="
    critical:
      critical_value: 1.0
      critical_operator: ">="
#####

```

Figure 4-12 An example as input for the Alarm-Trigger in the SWITCH project.

In the YAML file shown in the above figure, “type” can have different options: ['vm\_level', 'container\_level', 'application\_level']. Moreover, “action” can include various options: ['average', 'minimum', 'maximum', 'whatever']. Besides that, “operator” can be defined as: ['<', '<=', '>', '>=', '==', '!=', 'whatever']. In addition, “unit” can be chosen as one of these options: ['%', 'bytes/s', 'KB/s', 'ms', 'frames/s', 'whatever'].

## 4.5 Self-adapter Decision Maker

### 4.5.1 Functionality

To achieve adequate QoS for applications in the SWITCH project, runtime variations in running conditions intrinsic to the cloud and edge environments should be monitored. These types of systems should therefore be continuously monitored and hence adapted at various levels including infrastructure, container and application levels. To this end, the "Self-adapter Decision Maker" that applies a new incremental learning approach, explained in this section as Learning Classifier System (LCS), has been proposed based on multi-level monitoring data. The "Self-adapter Decision Maker" is able to detect runtime changes in the application environment and define the way of reacting to continuously adapt the running services for optimal performance. The "Self-adapter Decision Maker" dynamically generates a set of rules that allow us to find potential performance bottlenecks and then make the decisions to achieve suitable application adaptation actions.

### 4.5.2 Learning Classifier System

The LCS is aimed at defining a set of adaptation rules stored in the Knowledge Base. The LCS module includes three fundamental components: (I) Environment, (II) Learning machine, and (III) Rule compaction.

**(I) Environment :** The environment means the source of monitoring data by which the LCS learns. In the early warning system, the preparation of this environment has four different pre-processing steps: (A) Producing monitoring data, (B) Averaging monitoring data, (C) Re-formatting monitoring data, and (D) Converting monitoring data from numeric to binary format.

**(A) Producing monitoring data:** In the general sense, the high-level structure of monitoring data constructed by the proposed monitoring framework in the SWITCH project has been shown in Figure 4-13. To enhance the performance of an application component (e.g. DB Server in the BEIA use case), one adaptation action can be adding more instances of this component to the pool of servers so that load can be spread across multiple instances for one application component. In the following figure, rows belong to various metrics measured periodically in different times for all DB Servers, all CC Servers and all point-to-point links. The value of metrics can be Long, Double or Float.

	Metric ID	Metric Name	Metric Type	Metric Unit	Measuring Time	Value
DB Server 1	-	-	-	-	-	-
	-	-	-	-	-	-
	-	-	-	-	-	-
DB Server 2	-	-	-	-	-	-
	-	-	-	-	-	-
	-	-	-	-	-	-
	-	-	-	-	-	-
	-	-	-	-	-	-
	-	-	-	-	-	-
	-	-	-	-	-	-
	-	-	-	-	-	-
	-	-	-	-	-	-

Figure 4-13 Original format of the monitoring data.

**(B) Averaging monitoring data:** Our solution periodically measures the average value for all metrics at each time, e.g. the average read and write latency of all DB Servers. In this way, shown in Figure 4-14, for each time period, there exist average values of all metrics for application components, not for individual instances.

	Metric Name	Measuring Time	Average Value
DB Server	.	.	.
	.	.	.
	.	.	.
CC Server	.	.	.
	.	.	.
	.	.	.
	.	.	.
	.	.	.
	.	.	.
	.	.	.
	.	.	.
	.	.	.

Figure 4-14 Averaging the original monitoring data.

**(C) Re-formatting monitoring data:** Each row in the original format of monitoring data is a measurement record for one metric. The original format needs to be modified since it should be consistent with the environment usable by the LCS algorithm. In this step of preparation procedure, all measurements belonging to the same time interval have been gathered together as one row in the new format (Figure 4-15). According to the new structure, the overall application performance is the prediction class (endpoint) which we need to enhance as the main goal and hence deliver the result as early as possible for the best real-time user experience.

Application-level metrics for CC Server	Application-level metrics for DB Server	Container-level metrics for CC Server	Container-level metrics for DB Server	Infrastructure-level metrics for CC Server	Infrastructure-level metrics for DB Server	End-to-end network quality-level metrics	Overall application performance (endpoint)
3 metrics	2 metrics	8 metrics	8 metrics	9 metrics	9 metrics	4 metrics	1 metric
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

Figure 4-15 Re-formatting monitoring data.

**(D) Converting monitoring data from numeric to binary format:** Numeric values of all features stored in the reformatted monitoring data have been converted to binary values. Therefore, in our proposed the "Self-adaptor Decision Maker", all rows representing states of the environment and consequently all rules include attributes that have the binary value. To this end, the thresholds of all monitoring metrics have been used. For instance, the threshold for average CPU usage for each application component in the infrastructure level can be considered 80 percent. Then, if the average CPU utilization is less than 80 percent, it has been converted to 0 and on the other hand, if it is over 80 percent, it has been converted to 1.

**(II) Learning machine:** The learning machine iterates over the dataset repeatedly until some stop criteria are met, or the maximum number of learning iterations is reached. As the result, the learning machine generates a set of rules used by the "Self-adaptor Decision Maker". Rules typically take the form of an {IF:THEN} expression, e.g. {IF 'condition' THEN 'action'}. An individual rule is not itself a prediction model, since the rule is only applicable when its condition is satisfied. The entire population of rules collectively forms the prediction model. Each attribute in a rule can be 0, 1, or '#' as "don't care" symbol (also referred as wildcard). For example, the rule (#1###0### ~ 1) as {condition ~ action} can be interpreted in this way: IF the second attribute = 1 AND the sixth attribute = 0 (regardless of other attributes) THEN the prediction class = 1. In the example, the second and sixth attributes have been specified in this rule, while the others were generalized. A rule along with its associated parameters (such as accuracy, fitness and numerosity) is often referred as a classifier. In Michigan-style LCS as the most common type of LCS algorithm, classifiers are contained within a population ([P]) that has a user defined maximum number of classifiers. The [P] starts out empty (i.e. there is no need to randomly initialize a rule population). Classifiers will instead be initially introduced to [P] with a covering mechanism. The learning machine includes different components that operate in a step-wise



learning cycle [11]: (Step 1) Training, (Step 2) Matching, (Step 3) Covering, (Step 4) Updating, (Step 5) Subsuming, (Step 6) Genetic algorithm, (Step 7) Deleting.

**(Step 1) Training:** The beginning step in incremental learning is getting a training instance from the environment. For online learning, LCS will obtain a completely new training instance for each iteration from the environment.

**(Step 2) Matching:** The next step is finding all rules in the population [P] that have a condition matching the attributes values of the training instance. In other words, every rule in [P] is now compared to the training instance to see which rules match. A rule matches a training instance if all feature values specified in the rule condition are equivalent to the corresponding feature value in the training instance. For example, assuming the training instance is (001001 ~ 0), these rules would match: (###0## ~ 0), (00###1 ~ 0), (#01001 ~ 1), but these rules would not (1##### ~ 0), (000##1 ~ 0), (#0#1#0 ~ 1). In matching step, the endpoint (action or prediction class) specified by the rule is not taken into consideration. At the end, matching rules are moved to a match set [M]. As a result, the [M] may contain classifiers that propose conflicting actions. Afterwards, since we are performing supervised learning, [M] will be divided into a correct set [C] and an incorrect set [I]. A matching rule goes into the [C] if it proposes the correct action (based on the known action of the training instance), otherwise it goes into [I]. At this point, if no rule has been made into either [M] or [C], then the covering step will be applied.

**(Step 3) Covering (as rule discovery):** Covering is one of two mechanisms that can introduce new rules to [P] also known as “rule discovery”. Covering randomly generates a rule that matches the current training instance. It works by generating a rule condition, which randomly specifies a subset of attribute values in the current training instance, and applies wild cards (“#”) to the rest. The action or prediction class for the rule is set to the class of the current training instance. Assuming the training instance is (001001 ~ 0), covering might generate any of the following rules: (#0#0## ~ 0), (001001 ~ 0), (#010## ~ 0). Covering step not only ensures that during each learning cycle there is at least one correct, matching rule in [C], but also any rule initialized into the population [P] will match at least one training instance. As mentioned before, [P] typically starts empty. Because of this, covering step serves as a form of smart population initialization.

**(Step 4) Updating:** Parameters of any rule in [M] are updated to reflect the new experience gained from the current training instance. For example, we can simply update the accuracy of a rule. Rule accuracy is calculated by dividing the number of times the rule was in the correct set [C] by the number of times it was in the match set [M]. Rule fitness is also updated in this step, and is commonly calculated as a power function based on the inverse of rule accuracy. Numerosity of a classifier means the number of copies of this classifier in the population [P] (if there are multiple copies). Classifiers in the correct set [C] will see an increase in both accuracy as well as fitness. Classifiers in the incorrect set [I] will see a decrease in the accuracy and fitness.

**(Step 5) Subsuming:** In particular, rules that specify fewer attributes are likely to appear in match set [M] more frequently. Subsuming step is a generalization mechanism that merges classifiers that cover redundant parts of the problem space. In this way, it helps to decrease the size of population set [P] by subsuming a classifier to a more general classifier (and its numerosity has been increased). In other words, the subsuming step examines pairs of rules and looks for a situation in which one of the rules is a subsumer of another one. For example, rule (#####0 ~ 0) is a subsumer of (##1#00 ~ 0). A subsumer rule must cover all of the problem space of another rule, and must be more general and accurate while the more specific rule is eliminated from the population [P].

**(Step 6) Genetic algorithm (as rule discovery):** This step applies a simple Genetic Algorithm (GA) as the second type of rule discovery mechanisms. While other heuristics could be used to discover rules, the GA is most commonly used. In fact, only two new ‘offspring’ rules are typically generated by the GA and added to the rule population [P] during each learning cycle.

**(Step 7) Deleting:** The last step in the LCS learning cycle is to enforce the limited size of the rule population using deletion in order to maintain the maximum population size. The probability of a classifier being selected for deletion is inversely proportional to its fitness. Other factors such as the classifier’s numerosity can be applied to increase the probability of deletion (e.g. numerosity divided by fitness). This keeps [P] from being overrun by just a few rules with large numerosities. When a classifier is selected for deletion, its numerosity parameter is reduced by one. When the numerosity of a classifier is reduced to zero, it is removed entirely from the population [P].

**(III) Rule compaction:** Once the last learning iteration is reached, the resulting rules can be applied by the “Self-adaptor Decision Maker”. However, there is often a post-processing step called “rule compaction” applied to the resulting model after the last learning iteration. Rule compaction strategies typically seek to remove poor, redundant or inexperienced rules from the prediction model. In this way, rule compaction simplifies the model, improves interpretability, and even can enhance predictive performance.

### 4.5.3 Generated rules used by the "Self-adaptor Decision Maker"

In this section, few examples of rules that can be used by the "Self-adaptor Decision Maker" have been provided. Following is an example of a generated rule:

(rule 1) (#####0#####0#####0#### ~ 0)

In this rule, the first attribute with the value of 0 is "cpuUsedPercent" at infrastructure level for CC Server (Apache Tomcat), the second and third attributes (defined as 0) are "memUsedPercent" and "diskUsed" also at infrastructure level for DB Server (Cassandra). Other attributes have been set as "don't care" (#). This rule can be interpreted in this way: If the average CPU utilization of the host(s) on which CC Server(s) is(are) running does not exceed its threshold, also if the average memory usage and the amount of used disk capacity of the host(s) on which DB Server(s) is(are) running do not violate their thresholds, the overall application performance of early warning system is acceptable (0) considering users' satisfaction. Therefore, in situations when the overall application performance is not favourable, these three metrics should be investigated to define if they are violated.

For instance, if "memUsedPercent" and "diskUsed" for DB Server are not presenting a problem, however "cpuUsedPercent" for CC Server is inappropriately very high, regardless of other monitoring attributes, the Self-Adaptor suggests increasing the CPU power of the existing virtual machine(s) on which CC Server(s) is(are) running. In this situation, if vertical scaling is not feasible for example since maximum CPU capacity is already reached, the proposed adaptation plan could be other approaches e.g. live-service migration by moving running CC Server container(s) from the current infrastructure to another one either at the same data centre or at a different cloud to offer better fitted computational resources.

As another example, the following rule can be also inferred:

(#1##### ~ 1)

In this rule, the attribute with the value of 1 is "processingTime" at application level for CC Server (Apache Tomcat). Other attributes have been also defined as "don't care" (#). This rule can be interpreted in this way: If the average response time of CC Server component is unsuitable since it is more than associated threshold, the overall application performance is not acceptable (1). Therefore, in situations when the overall application performance is not appropriate, this metric ("processingTime") should be considered to determine if it is violated. For instance, if it is over the threshold, regardless of other monitoring attributes, the number of running containerized CC Server should be increased in order to enhance the overall application performance.

## 4.6 Self-adaptor Setup&Control

### 4.6.1 Functionality

In the proposed event-driven applications, service instances should be running across multiple hosts in clusters owned by different cloud providers. Self-adaptor Setup & Control component is used to abstract away the management of such geographically dispersed service instances. The purpose of this component is to execute the intended actions planned by the Decision Making module on the managed components that constitute the application. It allows us to add/remove application component instances, manage the amount of RAM and CPU share of individual instances and determine the subset of hosts or individual host, where the service instance will be placed. For management of individual clusters we assume the use of Kubernetes orchestration technology, which allows for horizontal and vertical scaling and resource management (RAM and CPU) within the individual cluster. Individual cluster is setup within a relatively performance, reliable and cheap network due to the design of scheduling and network routing domains. This confines the cluster boundaries to a single

cloud provider/availability zone. Self-Adapter Setup and Control component allows SWITCH applications to run over different cloud providers and multiple data centers and availability zones, by unifying the API to multiple federated Kubernetes clusters. To be able to do this the component communicates with the Knowledge Base, which stores the information needed for the orchestration of applications across clusters (i.e. information about the cluster credentials, master nodes endpoints, status of the environment etc.).

## 4.6.2 API description

The Self-Adapter Setup and Control exposes REST-based API and is using JSON over HTTP. The description in the below tables includes the REST endpoints that have been prepared in order to be able to orchestrate SWITCH applications across multiple Kubernetes clusters. It is important to understand that instead of direct management of individual containers, Kubernetes operates on the level of “pods” and uses “services” to expose the “pods” to the clients outside of the cluster. The Unifying API endpoints therefore allow for creation and destruction of “pods” and “services” rather than individual containers.

<b>REST endpoint</b> <div>POST</div>																			
/API/v01/{theIdOfTheCluster}/namespaces/{theNameOfNamespace}/pods																			
<b>Brief description</b> This endpoint is used to create a new Pod in a given cluster (identified by the {theIdOfTheCluster} parameter) and given namespace (identified by the {theNameOfNamespace} parameter). The specification of the Pod is given as a parameter in the body of HTTP request, and it is of mime type application/json. If this specification does not include the Pod name then Kubernetes friendly short UUID is generated and assigned as the name of the Pod. The endpoint is asynchronous, which means that it returns response immediately after the creation of the pod is initiated. A successful response therefore does not mean successful creation of the Pod.																			
<b>Parameters</b> <table> <tr> <th>Parameter</th><th>Type</th><th>Data Type</th><th>Description</th></tr> <tr> <td>theIdOfTheCluster</td><td>query</td><td>String</td><td>The ID of the cluster in which to start the Pod. This parameter is used to obtain the credentials and Master node endpoint from the Knowledge Base. These are then used to connect to the desired Kubernetes cluster.</td></tr> <tr> <td>theNameOfNamespace</td><td>query</td><td>String</td><td>The name of the Kubernetes namespace in which to start the Pod.</td></tr> <tr> <td>body</td><td>body</td><td>application/json</td><td>The specification of the Pod. It is in JSON format and represents the serialized <i>io.fabric8.kubernetes.api.model.Pod</i> object. The model schema is given below. Detailed description of the model schema can be found at <a href="http://kubernetes.io/docs/api-reference/v1/definitions/#_v1_Pod">http://kubernetes.io/docs/api-reference/v1/definitions/#_v1_Pod</a></td></tr> </table>				Parameter	Type	Data Type	Description	theIdOfTheCluster	query	String	The ID of the cluster in which to start the Pod. This parameter is used to obtain the credentials and Master node endpoint from the Knowledge Base. These are then used to connect to the desired Kubernetes cluster.	theNameOfNamespace	query	String	The name of the Kubernetes namespace in which to start the Pod.	body	body	application/json	The specification of the Pod. It is in JSON format and represents the serialized <i>io.fabric8.kubernetes.api.model.Pod</i> object. The model schema is given below. Detailed description of the model schema can be found at <a href="http://kubernetes.io/docs/api-reference/v1/definitions/#_v1_Pod">http://kubernetes.io/docs/api-reference/v1/definitions/#_v1_Pod</a>
Parameter	Type	Data Type	Description																
theIdOfTheCluster	query	String	The ID of the cluster in which to start the Pod. This parameter is used to obtain the credentials and Master node endpoint from the Knowledge Base. These are then used to connect to the desired Kubernetes cluster.																
theNameOfNamespace	query	String	The name of the Kubernetes namespace in which to start the Pod.																
body	body	application/json	The specification of the Pod. It is in JSON format and represents the serialized <i>io.fabric8.kubernetes.api.model.Pod</i> object. The model schema is given below. Detailed description of the model schema can be found at <a href="http://kubernetes.io/docs/api-reference/v1/definitions/#_v1_Pod">http://kubernetes.io/docs/api-reference/v1/definitions/#_v1_Pod</a>																
<b>Body schema:</b> <pre>{   "kind": "string",   "apiVersion": "string",   "metadata": {     "name": "string",     "labels": {       "name": "string"     },     "generateName": "string",     "namespace": "string",     "annotations": [ "string" ]   } }</pre>																			

```
},
"spec": {
  "containers": [
    {
      "name": "string",
      "image": "string",
      "command": [ "string" ],
      "args": [ "string" ],
      "env": [
        {
          "name": "string",
          "value": "string"
        }
      ],
      "imagePullPolicy": "string",
      "ports": [
        {
          "containerPort": int,
          "name": "string",
          "protocol": "string"
        }
      ],
      "resources": {
        "cpu": "string"
        "memory": "string"
      }
    }
  ],
  "restartPolicy": "string",
  "volumes": [
    {
      "name": "string",
      "emptyDir": {
        "medium": "string"
      },
      "secret": {
        "secretName": "string"
      }
    }
  ]
}
```

**Response Messages**

HTTP status code	Response model
200	String representing the name of the created pod.
500	The server encountered an unexpected condition which prevented it from fulfilling the request.

**REST endpoint**

**DELETE**


/API/v01/{theIdOfTheCluster}/namespaces/{theNameOfNamespace}/pods/{theNameOfThePod}

**Brief description**

This endpoint is used to delete a certain Pod (identified by the {theNameOfThePod} parameter) in a given cluster (identified by the {theIdOfTheCluster} parameter) and given namespace (identified by the {theNameOfNamespace} parameter). The endpoint is asynchronous, which means that it returns response immediately after the deletion of the pod is initiated. A successful response therefore does not mean that the Pod is already deleted. The Pod might remain in the “Running” phase for a while until Kubernetes Master node successfully terminates it.

<b>Parameters</b>			
Parameter	Type	Data Type	Description
theIdOfTheCluster	query	String	The ID of the cluster in which to stop the Pod. This parameter is used to obtain the credentials and Master node endpoint from the Knowledge Base. These are then used to connect to the desired Kubernetes cluster.
theNameOfNamespace	query	String	The name of the Kubernetes namespace in which to stop the Pod.
theNameOfThePod	query	String	The name of the Pod to delete.

<b>Response Messages</b>	
HTTP status code	Response model
200	String representing the name of the deleted pod.
500	The server encountered an unexpected condition which prevented it from fulfilling the request.

<b>REST endpoint</b>			
 <b>/API/v01/{theIdOfTheCluster}/namespaces/{theNameOfNamespace}/pods/{theNameOfThePod}/hostIP</b>			
<b>Brief description</b>			
<p>This endpoint is used to obtain the external IP of the host where a certain Pod (identified by the {theNameOfThePod} parameter) in a given cluster (identified by the {theIdOfTheCluster} parameter) and given namespace (identified by the {theNameOfNamespace} parameter) is running.</p>			
<b>Parameters</b>			
Parameter	Type	Data Type	Description
theIdOfTheCluster	query	String	The ID of the cluster in which the Pod is running. This parameter is used to obtain the credentials and Master node endpoint from the Knowledge Base. These are then used to connect to the desired Kubernetes cluster.
theNameOfNamespace	query	String	The name of the Kubernetes namespace in which the Pod is running.
theNameOfThePod	query	String	The name of the Pod.

<b>Response Messages</b>	
HTTP status code	Response model

200	String representing the external IP number of the host machine where the Pod is running.
500	The server encountered an unexpected condition which prevented it from fulfilling the request.

**REST endpoint**

GET

/API/v01/{theIdOfTheCluster}/namespaces/{theNameOfNamespace}/pods/{theNameOfThePod}/notifyWhenRunning

**Brief description**

This endpoint is used to check when the Pod in a given cluster (identified by the {theIdOfTheCluster} parameter), given namespace (identified by the {theNameOfNamespace} parameter), and with given name (identified by the {theNameOfThePod} parameter) actually reaches the state "Running". It is needed because the creation of the Pod (as described above) is asynchronous. If the Pod is in the "Running", "Succeeded" or "Failed" phase at the time of request then the response will be sent back immediately. If the Pod is in "Pending" phase, then a listener will be registered and the method will block until the Pod reaches the "Running" phase. More information on Pod phases can be found at <http://kubernetes.io/docs/user-guide/pod-states/>

**Parameters**

Parameter	Type	Data Type	Description
theIdOfTheCluster	query	String	The ID of the cluster in which the Pod is running. This parameter is used to obtain the credentials and Master node endpoint from the Knowledge Base. These are then used to connect to the desired Kubernetes cluster.
theNameOfNamespace	query	String	The name of the Kubernetes namespace in which the Pod was initiated.
theNameOfThePod	query	String	The name of the Pod.

**Response Messages**

HTTP status code	Response model
200	Information on the Pod phase and status in application/json format representing the serialized switch.jernej.trnkoczy.PodIsRunningInfo object. The model schema is given below: {"podReachedRunningStatus":true,"podPhaseInfo":"string"}
500	The server encountered an unexpected condition which prevented it from fulfilling the request.

**REST endpoint**

GET

**/API/v01/{theIdOfTheCluster}/namespaces/{theNameOfNamespace}/pods/{theNameOfThePod}/status**

#### **Brief description**

This endpoint is used to retrieve the current phase of the Pod in a given cluster (identified by the {theIdOfTheCluster} parameter), given namespace (identified by the {theNameOfNamespace} parameter), and with given name (identified by the {theNameOfThePod} parameter). The endpoint is synchronous, i.e. the result containing the current phases of the pod will be returned immediately. Valid Pod phases are Pending, Running, Succeeded, Failed and, Unknown. More information on Pod phases can be found at <http://kubernetes.io/docs/user-guide/pod-states/>

#### **Parameters**

Parameter	Type	Data Type	Description
theIdOfTheCluster	query	String	The ID of the cluster where the Pod is located. This parameter is used to obtain the credentials and Master node endpoint from the Knowledge Base. These are then used to connect to the desired Kubernetes cluster.
theNameOfNamespace	query	String	The name of the Kubernetes namespace in which the Pod was initiated.
theNameOfThePod	query	String	The name of the Pod.

#### **Response Messages**

HTTP status code	Response model
200	Information on the Pod phase. The data type is String that can contain the following values: Pending, Running, Succeeded, Failed or Unknown.
500	The server encountered an unexpected condition, which prevented it from fulfilling the request.

#### **REST endpoint**

POST

**/API/v01/{theIdOfTheCluster}/namespaces/{theNameOfNamespace}/services**

#### **Brief description**

This endpoint is used to create a new Service in a given cluster (identified by the {theIdOfTheCluster} parameter) and given namespace (identified by the {theNameOfNamespace} parameter). The specification of the Service is given as a parameter in the body of HTTP request, and is of type application/json. If this specification does not include the Service name then Kubernetes friendly short UUID is generated and assigned as the name of the Service. The endpoint is asynchronous, which means that it returns response immediately after the creation of the Service is initiated. A successful response therefore does not mean successful creation of the Service.

**Parameters**

Parameter	Type	Data Type	Description
theIdOfTheCluster	query	String	The ID of the cluster in which to start the Service. This parameter is used to obtain the credentials and Master node endpoint from the Knowledge Base. These are then used to connect to the desired Kubernetes cluster.
theNameOfNamespace	query	String	The name of the Kubernetes namespace in which to start the Service.
body	body	application/json	The specification of the Service. It is in JSON format and represents the serialized <i>io.fabric8.kubernetes.api.model.Service</i> object. The model schema is given below. Detailed specification of the model schema can be found at <a href="http://kubernetes.io/docs/api-reference/v1/definitions/#_v1_service">http://kubernetes.io/docs/api-reference/v1/definitions/#_v1_service</a>

**Body schema:**

```
{
  "kind": "string",
  "apiVersion": "string",
  "metadata": {
    "finalizers": ["string"],
    "labels": {
      "name": "string"
    },
    "name": "string",
    "ownerReferences": [
      {
        "name": "string"
      }
    ]
  },
  "spec": {
    "selector": {
      "name": "string"
    },
    "ports": [
      {
        "name": "string",
        "nodePort": int,
        "port": int,
        "protocol": "string",
        "targetPort": 8080
      }
    ],
    "clusterIP": "string",
    "type": "string",
    "externalIPs": ["string"],
    "deprecatedPublicIPs": ["string"],
    "sessionAffinity": "string",
    "loadBalancerIP": "string",
    "loadBalancerSourceRanges": ["string"],
    "externalName": "string"
  }
}
```

**Response Messages**

HTTP status code	Response model
------------------	----------------



200	String representing the name of the created Service.
500	The server encountered an unexpected condition which prevented it from fulfilling the request.

**REST endpoint****DELETE**

/API/v01/{theIdOfTheCluster}/namespaces/{theNameOfNamespace}/services/{theNameOfTheService}

**Brief description**

This endpoint is used to delete a certain Service (identified by the {theNameOfTheService} parameter) in a given cluster (identified by the {theIdOfTheCluster} parameter) and given namespace (identified by the {theNameOfNamespace} parameter). The endpoint is asynchronous, which means that it returns response immediately after the deletion of the Service is initiated. A successful response therefore does not mean that the Service has been successfully removed from the system.

**Parameters**

Parameter	Type	Data Type	Description
theIdOfTheCluster	query	String	The ID of the cluster in which to stop the Service. This parameter is used to obtain the credentials and Master node endpoint from the Knowledge Base. These are then used to connect to the desired Kubernetes cluster.
theNameOfNamespace	query	String	The name of the Kubernetes namespace in which to stop the Service.
theNameOfTheService	query	String	The name of the Service to delete.

**Response Messages**

HTTP status code	Response model
200	String representing the name of the deleted Service.
500	The server encountered an unexpected condition which prevented it from fulfilling the request.

**REST endpoint****GET**

/API/v01/{theIdOfTheCluster}/namespaces/{theNameOfNamespace}/services/{theNameOfTheService}/nodePorts

**Brief description**

This endpoint is used to retrieve the list of nodePorts assigned to the Service in given cluster (identified by the {theIdOfTheCluster} parameter), given namespace (identified by the {theNameOfNamespace} parameter), and with given name (identified by the {theNameOfTheService} parameter). The endpoint is synchronous, i.e. the result containing the list of nodePorts assigned to the service will be returned immediately. More information on the nodePort mechanism of Kubernetes Services can be found at <http://kubernetes.io/docs/user-guide/services/>.

**Parameters**

Parameter	Type	Data Type	Description
theIdOfTheCluster	query	String	The ID of the cluster where the Service is located. This parameter is used to obtain the credentials and Master node endpoint from the Knowledge Base. These are then used to connect to the desired Kubernetes cluster.
theNameOfNamespace	query	String	The name of the Kubernetes namespace in which the Service was initiated.
theNameOfTheService	query	String	The name of the Serviced.

**Response Messages**

HTTP status code	Response model
200	A list (List<Integer> datatype in JSON format) of nodePorts assigned to the Service.
500	The server encountered an unexpected condition which prevented it from fulfilling the request.

**REST endpoint**

GET

/API/v01/{theIdOfTheCluster}/namespaces/{theNameOfNamespace}/availableNodePorts

**Brief description**

This endpoint is used to retrieve the list of currently available nodePorts in given cluster (identified by the {theIdOfTheCluster} parameter) and given namespace (identified by the {theNameOfNamespace} parameter). The endpoint is synchronous, i.e. the result containing the list of available nodePorts at the moment of request is returned immediately. More information on the nodePort mechanism and the default port range can be found at <http://kubernetes.io/docs/user-guide/services/#publishing-services---service-types> . If the default port range needs to be extended it is possible to do so - <https://github.com/kubernetes/kubernetes/issues/11690> - however it is not recommended due to port conflicts.

<b>Parameters</b>			
<b>Parameter</b>	<b>Type</b>	<b>Data Type</b>	<b>Description</b>
theIdOfTheCluster	query	String	The ID of the cluster in which the available nodePorts need to be retrieved. This parameter is used to obtain the credentials and Master node endpoint from the Knowledge Base. These are then used to connect to the desired Kubernetes cluster.
theNameOfNamespace	query	String	The name of the Kubernetes namespace in which the available nodePorts need to be retrieved.

<b>Response Messages</b>	
<b>HTTP status code</b>	<b>Response model</b>
200	A list (List<Integer> datatype in JSON format) of nodePorts available in a given cluster and namespace.
500	The server encountered an unexpected condition which prevented it from fulfilling the request.

### 4.6.3 Developed software

The API has been developed using Java Jersey (Jersey RESTful Web Services) framework, an open source, production quality, framework for developing RESTful Web Services in Java. Our implementation uses Fabric8 Kubernetes API library, a Java library providing easy access to the Kubernetes and OpenShift API. This allowed us to abstract away the complexities related to security aspects (authentication, authorization) and asynchronous nature of Kubernetes API. Furthermore, Fabric8 Kubernetes API library allows for automatic parsing of the raw JSON data returned by the individual Kubernetes management API servers. The Java source code of the interface can be found in GitHub at <https://github.com/switch-project/WP4-SelfAdapterSetupAndControl>. All the REST endpoints described above were successfully deployed and tested on Apache Tomcat 8 server in a form of WAR (Web Archive) file. Besides deployment in a form of .war file a Docker image containing Tomcat server running the API has been built and is available in DockerHub: <https://hub.docker.com/r/jernejtrnkoczy/jcontrolagent01> . The Docker file used to build this image, together with instructions on image usage, all necessary configuration files and API packed as WAR file can be found on GitHub at <https://github.com/switch-project/WP4-SelfAdapterSetupAndControl/tree/master/BuildDockerImage> .

## 4.7 Performance diagnose Model Generator

### 4.7.1 Functionality

In terms of performance-wise adaptability of the SWITCH platform, decision-making process is the key component. It individually for every cloud application selects the best possible cluster for execution regarding QoS towards the application end-users. As a prerequisite, the automatic decision maker relies on a QoS model, which is generated automatically based on QoS observations prepared by cloud application developers during the development and testing process. Therefore, cloud application developers are expected to define a QoS metric for the application, which, hopefully, reflects in as good QoE as possible for all the parties engaged in the application. Beside QoS metric, developers are also expected to provide additional metrics – let us say non-QoS metrics – that can be easily monitored on a cluster or network level without having to execute the application itself (in other words, the rest of the metrics are not on an application level). This assumption allows for the decision making process to perform simple measurements before executing the application without having to run the application itself while relying on the QoS model to identify the relevance of the

non-QoS metrics for the QoS. The model employs qualitative modelling techniques and has been fully described in D4.3. For the sake of completeness, it is briefly summarised here, as follows.

Let  $M$  denote metrics table with  $n$  rows and  $m$  columns, which was prepared by application developers during the application development and testing process. Rows correspond to measurements repeated on various machine and cluster configurations while columns correspond to different metrics employed. We assume that the first column contains the QoS metric, and the rest of the columns contain non-QoS and non-application-level metrics. To build a QoS model from measurements  $M$ , proceed with the following:

1. Compute differences  $D$  along rows of  $M$  for each unique pair of rows in  $M$ ;
2. In  $S$ , for each element of  $D$  keep only the sign, with -1 encoding -, 0 encoding 0, and +1 encoding +;
3. Detect correlation between metrics: for each pair of correlated metrics mark one of them as being correlated with the other. Two metrics are defined to be correlated, if they match in sign for all the given examples. Correlations are marked into a vector  $c$ ;
4. Compare into  $R$  each non-QoS column of  $S$  with the QoS column of  $S$ . For each column element take 1 for match and 0 for no match;
5. At this step,  $R$  contains only non-QoS columns. Into  $w$ , perform a summation alongside columns to find the number of examples for each column where there is a match with the QoS metric;
6. Finally, normalise  $w$  such that the range of values fits between -1 and 1.

Result  $w$  is a row vector of weights corresponding to non-QoS metrics, where values express the relevance of each non-QoS metric for the QoS metric. A positive value indicates that the direction of change in the QoS metric in most number of the given examples also imply the same direction of change in the respective non-QoS metric, while a negative value indicates opposite direction of change between the QoS and the respective non-QoS metric. To the decision maker, the sign of the  $w$ 's elements tell whether to minimise or maximise the respective metrics. The magnitude of  $w$ 's values (i.e. the absolute value of  $w$ 's values) denotes the degree of relevance towards the QoS for each of the non-QoS metrics.

The presented algorithm is not too sensitive on the normalisation of values in a sense that different metrics have different units and ranges of values. Therefore, no normalisation of values is involved on the input table  $M$ .

#### 4.7.2 Time and space complexity improvement

The algorithm for computing QoS model based on measurements table  $M$ , as presented above, has a quadratic time and space complexity in number of examples  $n$  for all the steps, except for steps 3 and 6. In fact, the computational complexity of steps 1, 2, 4 and 5 requires  $\binom{n}{2}m$  operations and the same amount of space for storing the result (step 5 requires to store  $m - 1$  values only). Worse, step 3 is quadratic in both,  $n$  and  $m$ . For a large number of example cases (i.e. rows  $n$ ) this might be undesirable. Because the result is a row vector with  $m - 1$  elements, the space complexity can be easily reduced to  $O(nm)$ , which is the size of the input table  $M$  and is therefore optimal. To achieve this, omit computation of correlations (i.e. step 3). Then start with zero vector  $w$  and work on an arbitrary pair of rows of  $M$  at a time, while combining steps 1, 2, 4 and 5 before processing the next pair of rows from  $M$ . After each iteration, add the intermediate vector for each pair to  $w$ .

The above update of the algorithm does improve the space but not the time complexity – time improvements can be achieved by considering less row pairs from  $M$ . This might require careful consideration of the distribution of the values of examples and performing random sampling with respect to the distribution. We followed an easier approach, in which small subsets of row pairs from  $M$  are randomly and independently chosen. Then, for each subset of pairs  $w$  is computed independently. Finally, variants of  $w$  are compared in the distribution of metric weights. If there is no significant difference between them, then any of them is returned as a solution. If not, the computation is repeated with more pairs observed. This optimisation is used only for large enough  $n$  – about 1000 or more.

Moreover, if it is desired to compute correlations between metrics (step 3), it can be done as follows. Start with  $w$  just before it is normalised (i.e. after step 5). We claim that if two metrics are correlated, then the

corresponding absolute values in  $w$  should be the same. However, matched absolute values in  $w$  for correlated pair of metrics is a necessary condition, but is not sufficient. We suggest to perform computation of correlations only for matched scalar pairs in  $w$  (i.e. when a pair of values from  $w$  matches in their absolute values), which should on average notably reduce the number of vector pairs to consider.

Lastly, to better leverage on the scalability of clouds, the computation of QoS model can be easily parallelised. Each compute thread (or processor) independently computes partial  $w$  (up to step 5) on a small and unique subset of row pairs of  $M$  and requires no synchronisation. Finally, the results of processors are reduced into a single  $w$  before it is normalised (step 6). Simple parallelisation is especially beneficial for online updating of the QoS model from various sources.

## 4.8 Knowledge Base

### 4.8.1 Functionality

The proposed solution uses knowledge base as a main ASAP storage system that represents the data in a RDF graph based representation. The main advantage of the technology is the development of a robust ontology, where the data has a higher level of logic representation such as richer inter-entity relationships, constraints and complex data analysis mechanisms (e.g. validators and reasoning mechanisms). Therefore, the aim of the knowledge base is to store a subset of a representative monitoring data, such as alarm trigger violations, that can be further analysed and even extended with new assumptions and inferences. Results can be reflected as an optimization tool for algorithms with high time complexity, for example to optimize cloud environment [12]. The basic idea of the paper is to reduce the input data through the Knowledge Base by preserving the quality of solution and at the same time significantly speed up the algorithm execution.

### 4.8.2 API description

<b><u>REST endpoint</u></b>			
GET			
<b>http://193.2.72.83:7070/SWITCH/rest/asap/getMonitoringInfo</b>			
<b><u>Brief description</u></b>			
This endpoint is used to obtain information about the running monitoring service over a running application. The monitoring service is localized through the IP of the monitoring server and application ID. The response is a map providing all available monitoring/state information and even knowledge base stored info such as violated monitoring metrics.			
<b><u>Parameters</u></b>			
Parameter	Type	Data Type	Description
ip	query	String	IP of the running monitoring server.
app_id	query	String	Application ID that specifies the desired monitoring information to be queried.
<b><u>Response Messages</u></b>			
HTTP status code	Response model		
200	A map (Map<String,String>) datatype in JSON format that specifies the monitoring information such as running state, number of monitoring agents, applied monitoring metrics, violated monitoring metrics (see REST getViolatedMonitoringMetrics) and others.		

500	The server encountered an unexpected condition which prevented it from fulfilling the request.
-----	--

**REST endpoint**

GET

**http://193.2.72.83:7070/SWITCH/rest/asap/getAvailableMonitoringMetrics****Brief description**

This endpoint returns the available monitoring metrics of a running monitoring agent. It is usually used to register new monitoring metrics in the monitoring system (e.g. as a combo box in the GUI).

**Parameters**

Parameter	Type	Data Type	Description
monitoring_agent_id	query	String	The ID of a running monitoring agent where we want to get available monitoring metrics.

**Response Messages**

HTTP status code	Response model
200	A map (Map<String,String>) datatype in JSON format that specifies the monitoring metrics with monitoring metric ID as a key and monitoring human readable description as a string. Additional extension is possible to return also the group of a monitoring metric.
500	The server encountered an unexpected condition, which prevented it from fulfilling the request.

**REST endpoint**

GET

**http://193.2.72.83:7070/SWITCH/rest/asap/getAvailableAsapClusters****Brief description**

This endpoint returns a list of all available and running ASAP cloud clusters. No input parameters are included.

**Response Messages**

HTTP status code	Response model
200	A map (Map<String,String>) datatype in JSON format that specifies the cluster information (e.g. ID, name, cluster type, current state, number of running pods etc.).
500	The server encountered an unexpected condition which prevented it from fulfilling the request.

**REST endpoint**

GET

http://193.2.72.83:7070/SWITCH/rest/asap/getViolatedMonitoringMetrics

**Brief description**

This endpoint returns a list of violated monitoring metrics. In cases when monitoring triggers executes the alarm trigger notifies the self adapter and the violation information is stored into knowledge base according the ASAP ontology. The presented input parameters are not mandatory and are used like a filter to reduce the result list.

**Parameters**

Parameter	Type	Data Type	Description
monitoring_metric_id	query	String	The ID of a monitoring metric that we want to investigate.
cloud_id	query	String	The ID of a cloud to be checked the monitoring metric violations.
application_id	query	String	The ID of a specific application where the violations occurred.

**Response Messages**

HTTP status code	Response model
200	A map (Map<String,String>) datatype in JSON format that specifies all available information of the violated monitoring metrics (e.g. monitoring metric, metric value, unit, timestamp, performed adaptation action etc.).
500	The server encountered an unexpected condition which prevented it from fulfilling the request.

### 4.8.3 Developed software

The API has been developed using Java Jersey (Jersey RESTful Web Services) framework. The most important libraries used in the API are:

- Jena Fuseki<sup>1</sup>: used for knowledge base manipulation (RDF CRUD operations, application of reasoning mechanisms, validators, rules etc.),
- Fabric8<sup>2</sup>: used for directly interaction with ASAP clusters that are founded on Kubernetes, and
- Cassandra<sup>3</sup>: used for communication with TSDB where monitoring information is needed to be obtained.

The Java source code of the ASAP Knowledge base API can be in GitHub at [https://github.com/switch-project/WP4-KB\\_API](https://github.com/switch-project/WP4-KB_API). All the REST endpoints described above were deployed and tested on Apache Tomcat 8 server in a form of WAR (Web Archive). The main used build technology is Apache Maven, which does not need to encapsulate the libraries in the project but are downloaded from the official Maven repository. Deployed API is currently deployed on Tomcat 7 and can be accessed on the following URL <http://193.2.72.83:7070/manager/html>. Jena Fuseki server runs as standalone service on the following URL

<sup>1</sup> <https://jena.apache.org/documentation/fuseki2/>

<sup>2</sup> <https://fabric8.io/>

<sup>3</sup> <http://cassandra.apache.org/>

<http://193.2.72.83:3030/index.html>. To facilitate the development and deployment process a continuous build integration tool will be used and described in the purposive deliverable.

## 5 Testbed description

In order to test the functionality of the ASAP, and be able to perform the measurements described in (section 6) we had to establish a testbed. Table 5-1 summarizes the testbed that was set up. As can be seen the testbed comprises 14 machines in seven geographically distributed Kubernetes clusters, deployed over three different cloud providers.

VM	Cluster	Cloud provider	Location	# vCPU	Host CPU type	vCPU schedule	RAM	Disk	Disk type	Docker version	Docker network driver	Docker graph driver	Host OS
1	1	FlexiOps	Edinburgh, UK	4	AMD Opteron 1.8 GHz	Pre-emptive	4 GB	100 GB	Magnetic	1.11.2	flanne ld	overlay	CoreOS 1185.5.0
2	1	FlexiOps	Edinburgh, UK	4	AMD Opteron 1.8 GHz	Pre-emptive	4 GB	100 GB	Magnetic	1.11.2	flanne ld	overlay	CoreOS 1185.5.0
3	2	Arnes	Ljubljana, Slovenia	1	Intel Core (Haswell), 2.4 GHz	Pre-emptive	4 GB	80 GB	Magnetic	1.10.3	flanne ld	overlay	CoreOS 1122.2.0
4	2	Arnes	Ljubljana, Slovenia	1	Intel Core (Haswell), 2.4 GHz	Pre-emptive	4 GB	80 GB	Magnetic	1.10.3	flanne ld	overlay	CoreOS 1122.2.0
5	3	GCP, Gke-us-west	The Dalles Oregon, USA	1	2.2 GHz Intel Xeon E5 v4 (Broadwell)	dedicated HW thread per vCPU	3.7 5 GB	100 GB	Magnetic	1.11.2	HW bridg e	overlay	COS 57
6	3	GCP, Gke-us-west	The Dalles Oregon, USA	1	2.2 GHz Intel Xeon E5 v4 (Broadwell)	dedicated HW thread per vCPU	3.7 5 GB	100 GB	Magnetic	1.11.2	HW bridg e	overlay	COS 57
7	4	GCP, Gke-eu-west	St. Ghislain, Belgium	1	2.3 GHz Intel Xeon E5 v3 (Haswell) platform	dedicated HW thread per vCPU	3.7 5 GB	100 GB	Magnetic	1.11.2	HW bridg e	overlay	COS 57
8	4	GCP, Gke-eu-west	St. Ghislain, Belgium	1	2.3 GHz Intel Xeon E5 v3 (Haswell) platform	dedicated HW thread per vCPU	3.7 5 GB	100 GB	Magnetic	1.11.2	HW bridg e	overlay	COS 57
9	5	GCP, Gke-asia-east	Changhua County, Taiwan	1	2.5 GHz Intel Xeon E5 v2 (Ivy Bridge) platform	dedicated HW thread per vCPU	3.7 5 GB	100 GB	Magnetic	1.11.2	HW bridg e	overlay	COS 57
10	5	GCP, Gke-asia-east	Changhua County, Taiwan	1	2.5 GHz Intel Xeon E5 v2 (Ivy Bridge) platform	dedicated HW thread per vCPU	3.7 5 GB	100 GB	Magnetic	1.11.2	HW bridg e	overlay	COS 57
11	6	GCP, Gke-asia-southeast	Jurong West, Singapore	1	2.2 GHz Intel Xeon E5 v4 (Broadwell)	dedicated HW thread per vCPU	3.7 5 GB	100 GB	Magnetic	1.11.2	HW bridg e	overlay	COS 57



12	6	GCP, Gke-asia-southeast	Jurong West, Singapore	1	2.2 GHz Intel Xeon E5 v4 (Broadwell)	dedicated HW thread per vCPU	3.75 GB	100 GB	Magnetic	1.11.2	HW bridge	overlay	COS 57
13	7	GCP, Gke-asia-northeast	Tokyo, Japan	1	2.2 GHz Intel Xeon E5 v4 (Broadwell)	dedicated HW thread per vCPU	3.75 GB	100 GB	Magnetic	1.11.2	HW bridge	overlay	COS 57
14	7	GCP, Gke-asia-northeast	Tokyo, Japan	1	2.2 GHz Intel Xeon E5 v4 (Broadwell)	dedicated HW thread per vCPU	3.75 GB	100 GB	Magnetic	1.11.2	HW bridge	overlay	COS 57

Table 5-1 Testbed for ASAP functionality testing.

## 6 Videoconferencing use-case

### 6.1 Use case description

To demonstrate ASAP functionality UL developed an event-driven VaaS (Videoconferencing as a Service) system [13]. The developed system is based on Jitsi-meet open source components. It is a WebRTC based multiparty videoconferencing solution that uses a real-time multimedia streaming based on RTP/UDP protocols. The system does not use full-mesh (peer-to-peer) connectivity, instead it uses a centralized SFU (Selective Forwarding Unit) based architecture. The application consists of several software components that are handling signalization and media streams forwarding. The SFU component is called Jitsi Videobridge. It is deployed in conjunction with a web server hosting JitsiMeet – a JavaScript WebRTC application that is used by end-users through web browsers. The signalization of the application is based on XMPP/Jingle protocols and is handled by Jicofo, a component that acts as a conference focus initiating sessions between the endpoints. The system uses the XMPP-capable messaging server for messages exchange; for this component, we selected the Prosody XMPP server.

In line with our event-driven approach, we do not intend to use a dedicated server(s) to host the VC services. The service is created and destroyed dynamically for each individual videoconference. The overall architecture of the developed prototype is presented on Figure 6-1.

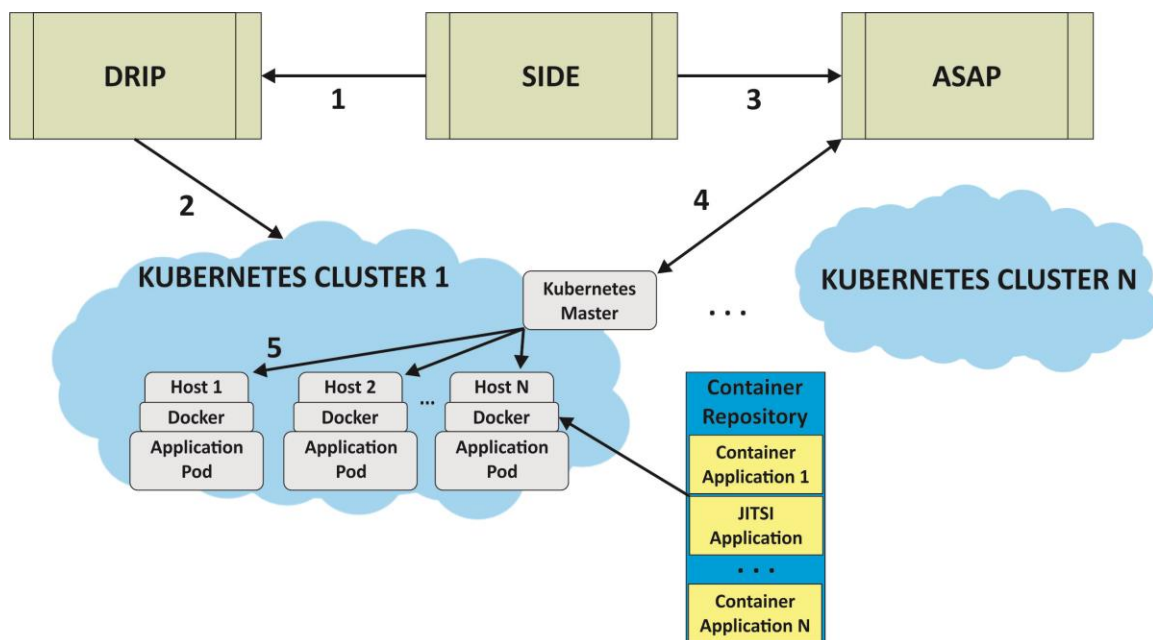


Figure 6-1 Video-conferencing use case.

In this picture multiple clusters are forming the testbed, but as will be shown in section 5., our testbed was actually composed of 14 machines in 7 different Kubernetes clusters, geographically distributed around the world. In order to start the Jitsi-meet VC application instances in a form of Docker containers that are managed by Kubernetes container orchestrator tool, we built Docker images of the components constituting the VC application (<https://github.com/switch-project/WP4-JitsiMeet/tree/master/JitsiMeetSeparateComponents/>). It is important to note that different application architectures could be considered when deploying the VC services. The decision, which components would be shared between multiple tenants and across multiple VC sessions and which of the components would be instantiated per VC session, had to be made. For example, the signalling components that are needed to establish VC sessions, and are not computationally and bandwidth demanding, neither do they request a low delay to the end users, could be centralized and shared by all the customers of the VaaS service. However, we decided that we will completely isolate the VC sessions and instantiate both signalling and media services. Therefore in our prototype for each new VC sessions all four Docker containers (jvb, jicofo, prosody and nginx) are run.

The GUI access to the application and management of the application instances, the optimisation strategy selection and application context capturing is provided through SIDE graphical user interface. Here a user can provide the context of the application, and select optimisation strategy, that is then used by the ASAP subsystem to select the appropriate machine where the service instance will be started. The context includes the number of users that will participate in a videoconferencing session, the application configuration (Last-N, simulcasting, user camera resolution, etc.) and the IPs/Geolocations of these users. The optimization strategy allows to fine tune the decision-making process of the ASAP subsystem. In particular, the user can select whether DM wants to optimise average QoS towards all parties or make it good QoS for some users, while it is acceptable to be degraded for others, as can happen in the case that parties are widely spread geographically.

The number of users and application configuration values are used to determine the appropriate amount of CPU and network interface bandwidth resources that will be needed for the service instance. CPU and bandwidth properties of the machine hosting SFU-based videoconferencing service are the properties that critically influence the application QoE. CPU power is needed for coding/decoding and encrypting/decrypting video streams. Network interface bandwidth should be high enough for transmission of video and audio streams. SFU-based systems are in general CPU-efficient, but the trade-off is paid in less bandwidth efficiency (if compared to MCU-based systems). Bandwidth efficiency of SFU-based systems is usually achieved by not forwarding the streams of all users, instead the application can be configured that a selection algorithm decides which packets to forward to which endpoints (this is called Last-N). The SFU-based systems can typically receive multiple streams of various qualities, and then based on the current networking properties of the connections to the clients choose which stream will be forwarded. This is called simulcasting. Therefore, to determine the needed resources of the host machine not only the number of users is important, but also the configuration of the application. If we know these properties, then the minimum required resources can be calculated and used in the “filtering” phase of the Decision Making. Furthermore, the IPs/Geolocation of users, which was provided as application context, is used to determine the region in which the service should be instantiated. This information is then also used in the “filtering” phase, where from a large number of possible VMs where to instantiate the service, a smaller subset of adequate VMs is calculated.

After the “filtering” phase ASAP has a short list of VMs that have adequate resources for the service and are located somewhere close to the users. Since our VaaS platform is supposed to work on the open Internet, the networking conditions between clients and servers are changing all the time. Therefore, in the “selection” phase; the network level metrics are monitored for all of the VMs in the short (i.e. filtered) list in the real-time. Then the best possible VM is selected according to the performance model. The output of the decision maker is therefore the exact VM where the VaaS services (four application components described above) should be started. It is important to note here that we make the assumption that only the SFU component (i.e. Jitsi Videobridge) has stringent resource requirements. We assume that signalling components (Jicofo and Prosody) and the web server serving the web application are not “resource hungry” if compared to the SFU component, therefore the quality of the signalling phase before the videoconferencing session happens was not modelled and taken into account.

After the decision, making the Setup and Control component is called. This component uses the Fabric8 libraries to deploy the VC instance on the right VM machine. Since the architectural decision was made that all four components will be instantiated for every videoconference call (i.e. all four component instances serve only one VC session), it is logical that they are deployed together (on the same host machine), therefore the four Docker containers are scheduled on Kubernetes cluster as a single Kubernetes pod.

## 6.2 *Experiment description*

In order to estimate the CPU and bandwidth requirements for the service instance, we performed an experiment in which we were varying the number of users and we monitored the CPU consumption (section 6.2.1) and bandwidth usage (section 6.2.2).

Since the service is intended to work over the best-effort public Internet, it is expected that by wise selection of the location, where the service will be instantiated it will be possible to raise the achieved QoE level. In order to see how the application quality depends on the selected location of the service we performed the measurement of PSNR (chapter 6.2.3) and frame latency (chapter 6.2.4). Unfortunately, as will be explained the measurement of PSNR failed.

In order to estimate if our event-driven (new service instance for every VC session request) approach is acceptable by the user we measured the times that are needed for the service creation (the time Kubernetes needs to instantiate the Pod) – see section 6.2.5.

### 6.2.1 CPU consumption

SFU-based VC systems only copy and forward user packets and should be less CPU demanding than MCU-based systems, which perform video coding/decoding/mixing as well. However, the CPU resources consumed by SFU units are still substantial. This can be explained by the use of Secure Real-time Transport Protocol (SRTP). Each packet has to be decrypted and encrypted again to be transmitted to each of the participants as every connection has its own encryption keys. This, combined with the very high bit-rate of the media, is a CPU intensive process. To estimate how the CPU requirements of SFU unit vary with a growing number of users (that are participating in the same videoconference session) we used a benchmarking tool called Jitsi Hammer. With this tool, we were able to simulate a varying number of users, while we used our monitoring system to record the CPU consumption level on the machine hosting Jitsi Videobridge (SFU unit).

On machine 5 in cluster 3, (see Table 5-1) we run a Docker container containing Jitsi Videobridge. We selected one of the Google machines for testing – because it does not use overlay networking inside Kubernetes cluster, which could further deteriorate the CPU usage. The Docker image was prepared so that the logging of the Jitsi Videobridge was disabled. This is important since the logging in Kubernetes with its Fluentd and Elasticsearch (<https://github.com/kubernetes/kubernetes/tree/master/cluster/addons/fluentd-elasticsearch/fluentd-es-image>) mechanisms can use a substantial amount of CPU power.

Then we started the Jitsi Hammer benchmarking tool on some other machine. However, the Jitsi Hammer does not use the single UDP port (port 10000) option of the Videobridge. This is why we could not start Jitsi Hammer on machines that are outside the Kubernetes cluster hosting the Videobridge service. Anyway, it is better to start the Hammer on a machine that is on the same LAN, since the networking between the two machines should be fast enough to handle all of the traffic (we are studying CPU, not network). Therefore, we developed a Docker image containing the Hammer tool. Then we started the Hammer, and signalling part of VC services (nginx, jicofo, prosody) on machine 6 in cluster 3. We had to make sure that this machine was not the bottleneck, so we temporarily (for the duration of the test) increased the resources of the machine (it was upgraded to n1-standard-2 with 2vCPU and 7.5GB memory). For the simulation of users, we used a video file that is included in the Hammer tool (badger-audio.rtpdump and badger-video.rtpdump - <https://github.com/jitsi/jitsi-hammer/tree/master/resources>). The configuration LastN, adaptiveLastN, simulcasting, video muting and congestion control configuration of the VC application were all disabled. Then we started the Hammer tool 13 times simulating 1,2,3,4,5,7,10,13,14,15,16,17, and 18 users in a VC session. In every run, we monitored the CPU consumption for 120 seconds, each second taking one measurement. We measured both CPU consumed by the container (Figure 2-1, blue line) and the CPU consumed by the VM

(Figure 2-1, red line). Then we calculated the average and standard deviation of these measurements. We obtained the following graph:

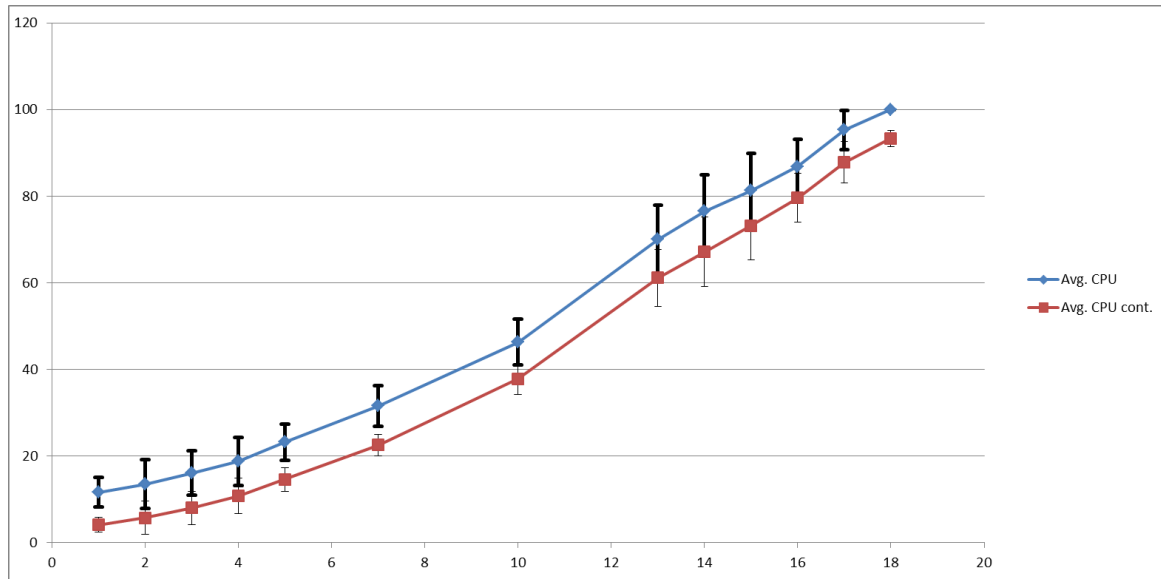


Figure 6-2 Average CPU consumption.

The graph shows that on the n1-standard-1 VM with 1CPU (single hardware hyper-thread on a Broadwell 2.2 GHz Intel Xeon E5 v4 processor) and 3.75GB RAM the average VM CPU usage reaches 80% with approximately 15 users in the same VC session.

## 6.2.2 Bandwidth usage

The same experiment (the same machines and configuration) described in the previous chapter was conducted, but this time we were measuring the bandwidth of the media streams on the network interface of Videobridge. We measured the received packets (coming from simulated users to Videobridge – Avg. NET rx.) and transmitted packets (forwarded from Videobridge to simulated users – Avg. NET tx.). The obtained graph is presented in Figure 6-3.

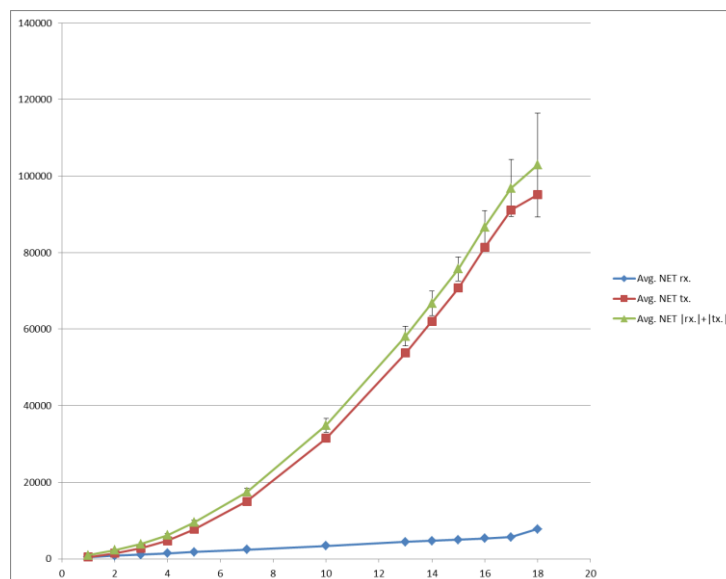


Figure 6-3 Bandwidth usage.

We can see from the graph that the required bandwidth for receiving video streams on the Videobridge network interface grows linearly with number of users. This is because each user is sending only one video stream to the Videobridge. On the other hand, the required bandwidth for sending (forwarding) video streams on the Videobridge network interface grows quadratically with the number of sending endpoints (users). This is because the number of forwarded video streams increases quadratically with the number of users, because each of the users also acts as a destination that traffic from everyone else needs to be delivered to. Of course this assumes that the Last-N configuration is disabled, representing the worst-case scenario where every user in the VC session wants to see all other participants.

### 6.2.3 PSNR

In this experiment, we run the VaaS service on geographically distinct VMs. The QoS metric in this case is PSNR (Peak Signal-to-Noise Ratio), a metric that represents the amount of video distortion between the transmitted and received the video stream. The final goal of this experiment is to find the correlation between the low-level network metrics, which can be measured by our monitoring system (bandwidth, delay and jitter between clients and SFU) and achieved the quality of the videoconference in terms of PSNR. If such correlation is found it would be possible to determine the best location where the SFU service should be started, according to the measured values (bandwidth, delay and jitter between clients and SFU).

The transmission related video degradation, measured with PSNR metric is mostly related to the video packet loss on the entire communication path. This includes loss at intermediate routers, as well as loss related to buffer overflow at either receiver and/or sender side. Another source of video degradation is due to the video resolution resampling and the amount of the video compression applied at the sender side. The video codecs used in WebRTC apply lossy compression techniques to the video obtained from user's camera. The client dynamically adapts its bitrate to the network conditions by selecting the appropriate video resolution and video compression level. The clients are running a congestion control algorithm known as the GCC (Google Congestion Control) algorithm. The receiver estimates the available throughput capacity based on variation in frame inter-arrival times and the Receiver Estimated Maximum Bandwidth (REMB) and a RTCP report is sent to the sender for performing congestion control. This suggests that the video quality degradation is not a simple function of the video slices loss (due to packet loss on lower network levels) but other network characteristics are important as well. The correlation functions between network parameters and PSNR value is therefore an extremely complex function. In order to model this function we would require a testbed capable of precisely controlled variations of the various network properties (throughput, delay, jitter, buffer sizes, etc.) and each of these should be controllable in isolation from the others. Unfortunately, we do not have such a testbed. The purpose of this experiment was therefore just to estimate if the geographical positioning of service influences on the measured PSNR value at all. Therefore, we run the service on the 14 machines of our testbed and evaluated the PSNR obtained.

The PSNR value was measured with the Jitsi Torture benchmarking tool. This tool is used to unit-test the application, therefore performs various functionality tests, and among them also the PSNR calculation. The PSNR test is based on stamping the input video with QR codes, sending this video through Videobridge back to the same application (client), capturing the received frames, finding the corresponding input frames (based on QR code identification) and calculating the PSNR value. Of course, since we are studying the impact of the intermediate network on the PSNR, we have to make sure that both the machine with Jitsi Torture and Videobridge have enough CPU, RAM and high-bandwidth network interfaces.

In our preliminary tests, we first prepared the input video/audio stream data, which is requested by Jitsi Torture tool. We obtained 1280x720pix, 60fps video file (KristenAndSara\_1280x720\_60.y4m) from <https://media.xiph.org/video/derf/y4m/>. This video is appropriate for the PSNR assessment since it contains a lot of movement. We run scripts to annotate the video with QR codes and decompose video into .png images (used when calculating PSNR of each individual frame). Then we run the Jitsi Torture test and measured the obtained PSNR when streaming through any of the 14 machines in our testbed. Unfortunately we obtained more or less the same values for each experiment – indicating that there is no PSNR difference if we setup Videobridge in Taiwan or if we set it up very close to our faculty (on Arnes). This was somewhat surprising and we then run normal videoconference through different machines – and observed (human observer) the quality. What we found out (with our human eyes) is that the blocking artefacts that are a symptom of packet



loss never occur – regardless where is the Videobridge. However, we noticed that the video “freezing” occurs substantially more often if Videobridge is far away. Obviously the packet loss and/or delay manifests in video “freezing” and not blocking artefacts. However, Jitsi Torture with its PSNR calculation based on QR code identification is not capable of detecting video “freezing” – therefore we should find some other tool for PSNR calculation (that takes into account also the “freezing”).

## 6.2.4 Frame latency

Videoconferencing applications, besides low packet loss, require also low media delay (voice and video). Such delays affect interactive human communication, and therefore knowing the delay is a major factor in judging the expected quality of experience of the conferencing system. For video, the delay can be measured in terms of delays of individual video frames – i.e. the time from when the video frame was captured with sender camera to the time when the same frame was rendered on the receiver screen. Frame delay is caused by processing of media streams on the entire communication path (sender encoding, SFU processing, receiver decoding etc.) and by network latency (buffering and routing in the intermediate switches and routers, propagation delay etc.). For the purpose of the frame delay measurements, we choose VideoLat measurement tool (<http://www.videolat.org/>). VideoLat can measure roundtrip video delay of a complete video chain (requires only one VideoLat device), and one-way video delay measurements (requires two VideoLat devices). In our experiment, we measured roundtrip video delay. The purpose of the experiment was to find out if the location of the SFU unit affects the round trip frame delay. We measured the frame delay for all 14 different machines in our testbed (there was an error in measurements on machine gke-asia-northeast-2 – so the results for this machine can be ignored). Alongside with frame delay we measured also the packet delay on network layer. We used PING tool. Both videoconferencing clients were located at our premises at the faculty. We made sure that both clients and SFU had plenty of compute resources to handle the two-user VC session. The client machines were two laptops with Intel® Core™ i7-4710MQ processors, 16GB RAM, 64bit WinOs, and Chrome browsers for videoconference establishment. The results that we obtained are presented in Figure 6-4.

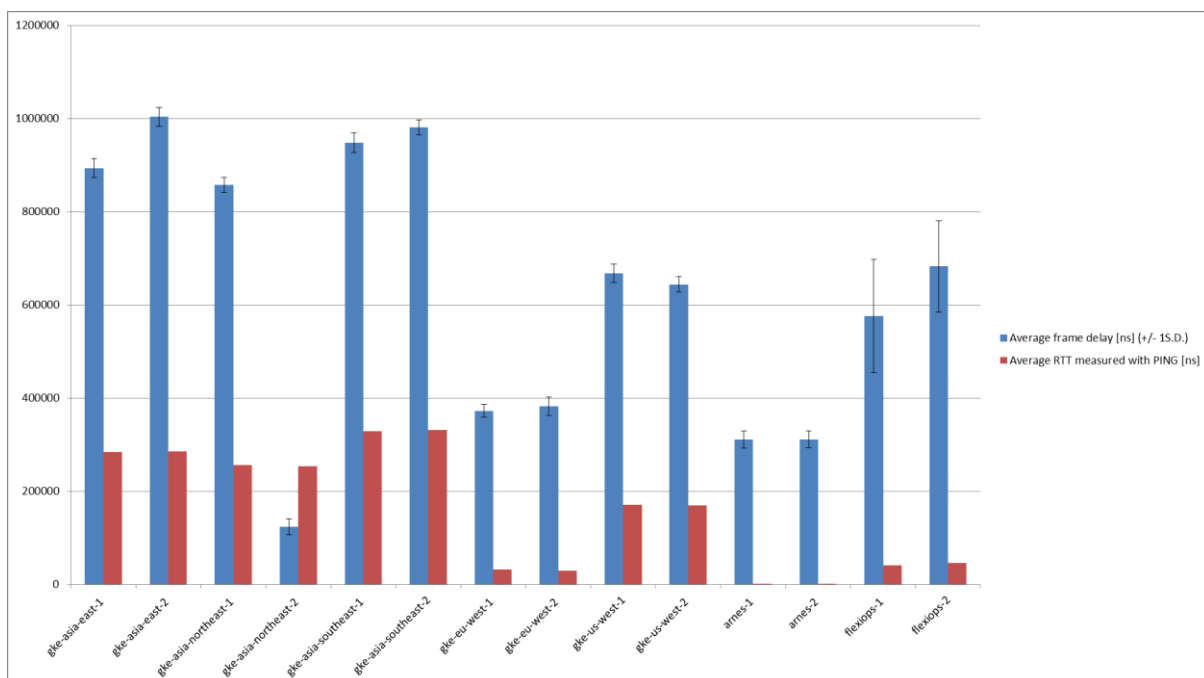


Figure 6-4 Frame latency.

The graph shows that the frame delay rises with the distance between SFU and our two video conferencing clients (located in Slovenia). As can be seen the delays are highest with machines located in Asia (Taiwan, Singapore, Japan), then machines in US follow (Oregon), and then machines in Europe (UK, Belgium, Slovenia). What is surprising is that the frame delay is much higher than the network-level delay measured

with PING. The processing of the media on clients and centralized SFU obviously introduces a substantial amount of delay. Again, to find out how exactly are the network-level delay and application-level frame delay correlated, we would require a (simulated) testbed capable of precisely controlled variations of the network-level delay.

## 6.2.5 Service start-up time

The time from the request for a service, to the actual creation and availability of the service is very important to the users. If the sum of phases 1, 2, 3 in is too large then the users will not be satisfied with the service. In this experiment, we measured the duration of the container deployment phase. The time of Context Capturing and Decision-making are not taken into account in this experiment. Since we are deploying the containers with Kubernetes, and all four VC containers are deployed as a single Kubernetes pod, we actually measured the time from the command for pod creation that was sent to the Kubernetes cluster, until the notification/callback indicating that the pod (and all four containers inside) is in “running” phase. The time of deployment was measured on two different machines (Arnes-1 and FlexiOps-1) and in two different scenarios: 1) Docker images already present on the host machine (indicated by “only run”), and 2) Docker images not present on host machine and need to be pulled from Docker registry (indicated by “pull + run”). All four Docker images were located in public Docker Hub registry. Uploading container images to a registry allows Docker hosts to pull down the image and spin up container instances by simply knowing the image name. However, the downside to having images in registries is that they are not local to the network on which the application is being deployed. This means that every layer of every deployment might need to be downloaded across the Internet in order to deploy an application. As it will be shown, the Internet latencies can have a big impact on software deployments. The sizes of the four images were the following: Jicofo=982MB, JVB=720MB, nginx=409MB, prosody=254MB. It is important to note that it is not necessary to transfer all 982+720+409+254MB of data when pulling the four Docker images, since Docker images are built in layers and some of these layers shared between different images. This can be then optimized by good image layering design and thin layers between image versions, which are easy to move around the Internet. The obtained results are presented in Table 6-1.

Exp. #	Arnes-1		FlexiOps-1	
	pull+run [ms]	only run [ms]	pull+run [ms]	only run [ms]
1	579268	5046	334273	4883
2	426866	6090	310815	5439
3	290720	5796	306856	4728
4	152784	4879	304379	4654
5	151140	4642	302163	4895
6	129751	5966	305383	4570
7	131279	8108	303439	4355
8	118000	4150	303728	4951

Table 6-1 Service start-up time.

From the results, it is clear that the actual service creation time is relatively short and consistent from experiment to experiment. In general, it takes around 5 seconds to start all four VC services in a form of single Kubernetes pod in case all four Docker images are already present on the machine where services are started. However if the images have to be pulled from Docker registry beforehand, the time for service creation is considerably higher, and it can vary from experiment to experiment (e.g. from 118 seconds to 579 seconds on Arnes-1 machine). The reason for this is that high amounts of data need to be transferred from the image storage server to the machine where we want to instantiate the service. There are a number of varying parameters affecting this transfer time: network bandwidth changes over time, the I/O speed of the storage varies over time, etc. From the experiment performed on Arnes-1 machine, we observed that in first experiment, the time to obtain the images from Docker Hub and run them was 579 seconds, and this time was rapidly falling with subsequent requests. The exact physical location of the images in Docker Hub was unknown to us, and we believe that the difference in pull times comes from Docker Hub using CDN for content

delivery. It seems that the CDN moved images to storage servers that are located closer to our Arnes-1 machine when we made several subsequent requests. The experiment on FlexiOps-1 machine was performed after the Arnes-1 experiment. Since both of these machines are located in Europe, we believe that in the case of FlexiOps-1 experiments the image was already somewhere in “European region” – and the times did not change much with further subsequent requests. This shows us two things: 1) the time needed to pull the images from storage is considerably higher than the time needed to actually start the images, and 2) the pull time is highly varying, depending on the storage performance, and the networking conditions during data transfer. To obtain the consistent service start-up-times it is better to pull the images to the potential machines where services will be started in advance, before the request for the service instantiation comes.

## 7 File Upload use-case

### 7.1 Use case description

File Upload is a simple use case, which – unlike videoconferencing – involves a single end-user participant in the service request: an end-user wants to upload a file to a remote location. Traditionally, she is expected to visit a certain web page that allows uploading files and then to perform an upload by selecting a local file for upload. For large files, an upload might take some time to complete and can contribute to a sluggish experience of the end-users, especially if perceived upload is slow and/or fails during the data transfer. Within SWITCH, ASAP tries to improve upon QoS of the application and hopefully to address the QoE aspect, depending on how well is QoS, as defined by the application developers, related to the QoE. Through SIDE the end-user first requests to start the File Upload service and then ASAP has to decide in which cluster to start the corresponding container resembling the File Upload service. To this end, ASAP utilizes the monitoring system to perform certain measures about the current load on the clusters and to determine the network conditions between the end-user and the available clusters. The decision for service placement is made based on the QoS model stored in KB and the observed conditions. After the decision is made, the service is started and the end-user can interact with the File Upload application as usual. Finally, when the file is uploaded, the container service can be stopped and its resources released. However, in a real-case scenario the service might want to store the file in a permanent storage before the container is destroyed. File Upload was developed as a simple Tomcat servlet.

### 7.2 Experiment description

The primary goal of the experimental evaluation was to evaluate the Performance model generator component of the ASAP subsystem. The required input for this component is a set of metrics defined by the application developers. One of the expected metrics is the QoS.

#### 7.2.1 Metrics for the QoS model

For File Upload we defined metrics shown in Table 7-1.

Metric	Description
Upload time [s]	File upload time in seconds.
Upload speed [B/s]	Upload speed or the goodput expressed in bytes per second.
File size [B]	Size of the uploaded file in bytes.
RTT [s]	Network packet round-trip-time in seconds between the end-user and the cluster.
Jitter [s]	Network packet RTT jitter (i.e. standard deviation) in seconds between the end-user and the cluster.
Packet loss [%]	Percentage of lost packets between the end-user and the cluster.



Metric	Description
Hops	Number of hops on the network path between the source and the target endpoint.
Container startup time [s]	The time in seconds to start the container.
Tomcat startup time [s]	The time in seconds to start the File Upload service.
Total time [s]	The total time in seconds of the container provision time, the Tomcat start time and the file upload time combined.
CPU usage [%]	Average CPU usage on the VM in the cluster during servicing file upload request (and possibly few others).
CPU iowait [%]	The percentage of CPU time used in waiting for significant I/O operation to complete.
Free memory [MB]	Average amount of free main memory on the observed machine during the file upload event.
I/O read [KB/s]	Average I/O read during servicing file upload in kilobytes per second.
I/O write [KB/s]	Same as I/O read but for write operation.
Normalised upload speed [B/s]	The QoS metric: defined as “File size” / “Total time”.

**Table 7-1 File Upload use case input metrics for the Performance model generator.**

Upload time, upload speed, file size, total time and normalised transfer speed are application specific metrics. Of those, upload time, upload speed and file size directly depend on each other, as upload speed is expressed as the quotient between file size and upload time. The “Normalised upload speed” is also an application-level metric and is similar to “Upload time”, but it incorporates also container and service startup times. These two metrics, however, do not depend on the file size, but rather on the characteristics of the VM and the respective container image – its size, the proximity of the cache locations and the quality of the network connections between the image (cache) repository and the cluster. The “Normalised transfer speed” was chosen as the QoS metric, because it directly relates to the perceived speed of the end-user when uploading files. All of these metrics can be measured either on a client or on a server side, but the measurements might slightly differ (i.e. the server observes shorter times than client does).

RTT, jitter and packet loss are network-level metrics that are measured between two endpoints. It is common to measure them with ping utility using ICMP, but other protocols and tools are possible too. A number of hops can be measured with traceroute or tracepath utilities, but the exact network path taken by successive packets might differ. The presence of firewalls and NATs on either side present challenges in measuring these metrics, but the resolution to this problem is out of scope.

Container and service startup times are metrics specific to our event-driven approach, as multi-tenant with instantly on services do not have to start the service first, except in auto-scaling and migration scenarios. These two metrics have been already discussed in Section 6.2.5.

CPU usage and CPU iowait can be measured on the container- or VM-level. CPU usage is always greater or equal to CPU iowait as the later is incorporated into the former. File Upload is not CPU demanding application, although with pre-emptively scheduled vCPUs of VMs on host machines and network-accessed VM disks the I/O wait can be significant.

Finally, I/O read and write are related to VM disk utilisation. Due to buffering, they might occur in batches, when for write operations buffers are filled up or when reading unbuffered content.

Not all non-QoS metrics presented in Table 7-1 are suitable for our QoS modelling. These include application-level metrics, because they cannot be measured prior to application start up. Next, “File size” was required in order to compute the QoS metric and the “Upload speed”, but is not particularly useful for the QoS model. For the rest of the metrics, their relevance to the QoS model is not clear, as some might be affected due to the

application. For example, I/O write or CPU iowait might increase due to File Upload application writing uploaded content on the disk, but only during that time. Therefore, these metrics might require the application running and performing upload during the measurements requested by the decision-making process, prior to the application deployment. On the other hand, the decision maker might know how to use them by knowing the general performance characteristics of the clusters and without having to run the application during the decision-making process.

## 7.2.2 Measurements

With monitoring system deployed on the entire target clusters of the experiment, the metrics' values were populated by simulating file uploads from a single client location towards the services deployed in several geographically dispersed clusters. Files in transport were of different sizes: 1 kB, 10 kB, 100 kB, 1 MB, 10 MB and 100 MB. The content of these files was randomly generated to avoid the undesired effects of HTTP requests body compression. Client application used was curl command utility. Every test was repeated 20 times. During each application session, all of the metrics were populated. Every file upload event corresponds to a single entry in a metrics table, which means that over a time window of the event duration, interval metrics were aggregated into a single, averaged value. To avoid Docker image pull delays, the File Upload container image was pre-pulled on all the machines involved in the experiment.

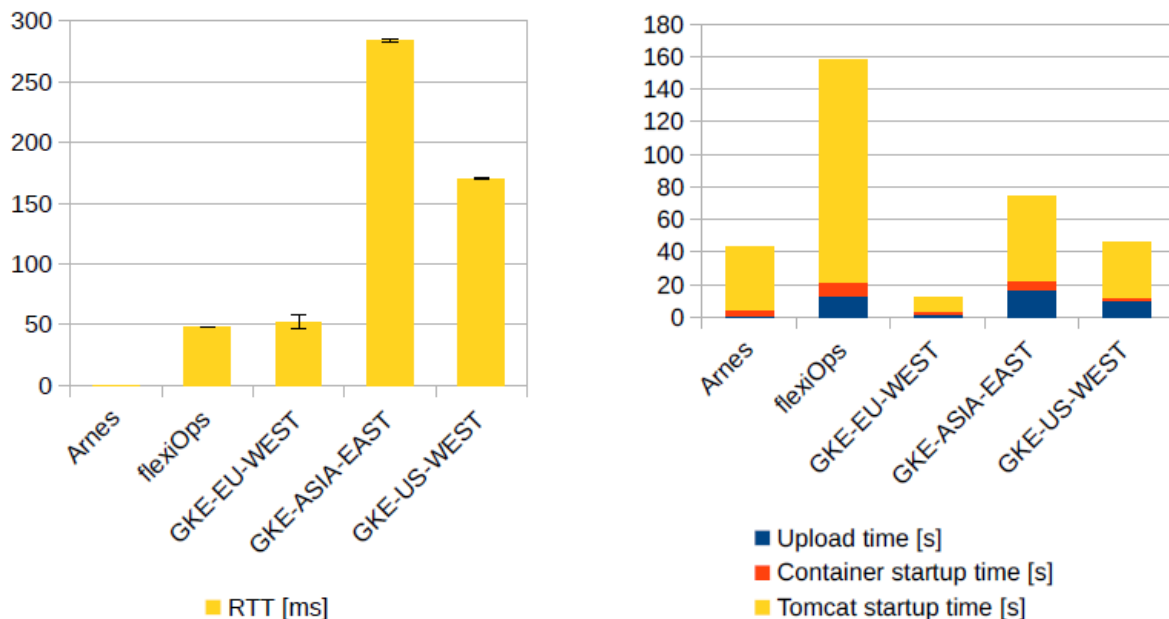
Figures Figure 7-1,

Figure 7-2,

Figure 7-3 show measurements of the metrics from Table 7-1 for the File Upload application deployed and executed on five different clusters. The RTT (Figure 7-1(A)) and the number of hops (

Figure 7-2 (A)) are good estimators of the upload speed in our case, with flexiOps being an exception. It relates well with the proximity of the servers relative to the client, as well as with the average upload speed. Quite surprisingly, the Tomcat service start up time,

Figure 7-1(B), was in our case the main candidate for optimisation, because for the average file size of 18.5 MB, the most time was spent in Tomcat startup time. The reason for this is believed to be due to the initialisation of secure strings during Tomcat starting, which relies on random source of numbers. Because the source of random on Linux operating system partially relies on non-deterministic source provided by human users through input devices, the source (known as an entropy) is limited for servers, due to the lack of the user input, compared to desktop computers. Therefore, the source of randomness is slow. It is possible to mitigate this issue either by installing an additional service, which introduces randomness, or by using cryptographically less secure, albeit faster counterpart. Clearly, no single non-QoS metric is an exact predictor of the QoS metric.



(A) RTT (shown as bars) and jitter (shown as y error bars).

(B) Average upload time, container and service startup times.

Figure 7-1 Average upload time, startup time, RTT and number of hops.

Figure 7-2(A) shows the network packet loss and the number of hops. The plot indicates 8% packet loss for GCP Asia East region, but in fact, it occurred less frequently, if at all. The reason is in the measuring methodology, in which RTT ping timeout was set too low. As a result, the last packet in a sequence timeouted before it was able to be confirmed. Instead of repeating the measurements, the packet loss was left as a noise to the QoS modeller.

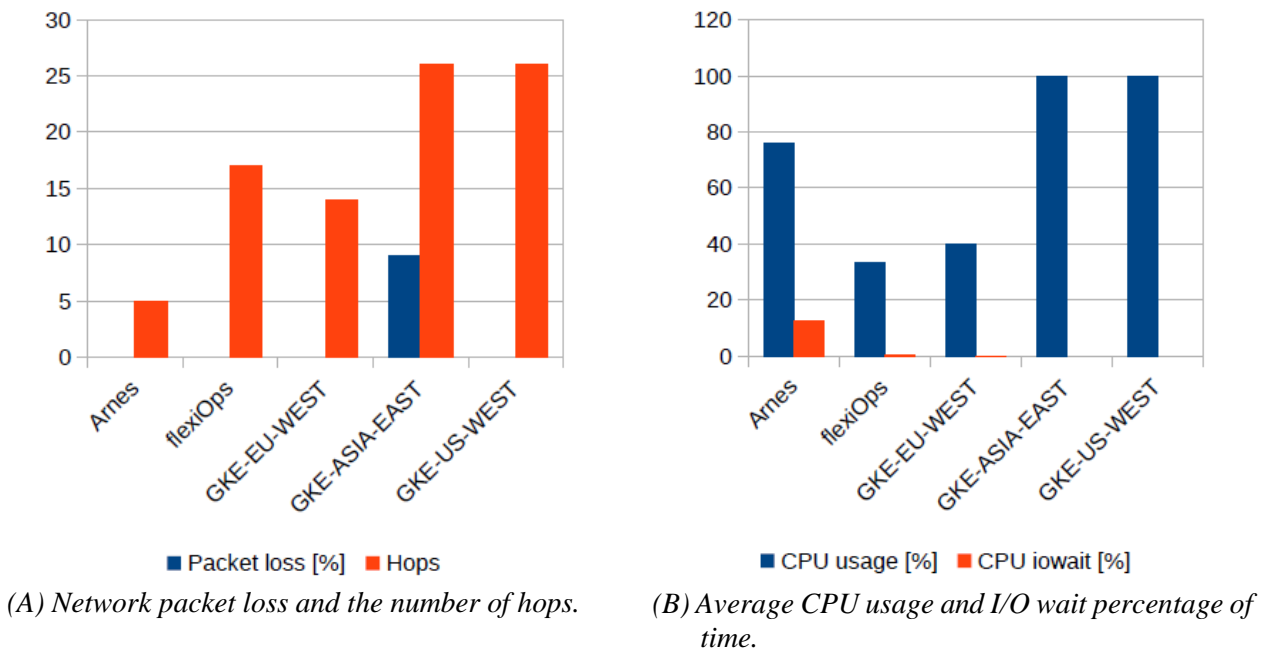


Figure 7-2 Network packet loss, number of hops, CPU usage, I/O wait percentage.

Figure 7-2(B) shows the average CPU usage and I/O wait percentage of time. Intentionally a low-priority CPU load was put on GCP in Asian and US regions to disturb the QoS modeller. High CPU usage for Arnes relates to short upload times, as most CPU time was spent in starting up the service and not in servicing the file upload itself. Due to the high upload rate, CPU I/O wait was quite noticeable on Arnes. Furthermore, judging only by the upload speed (

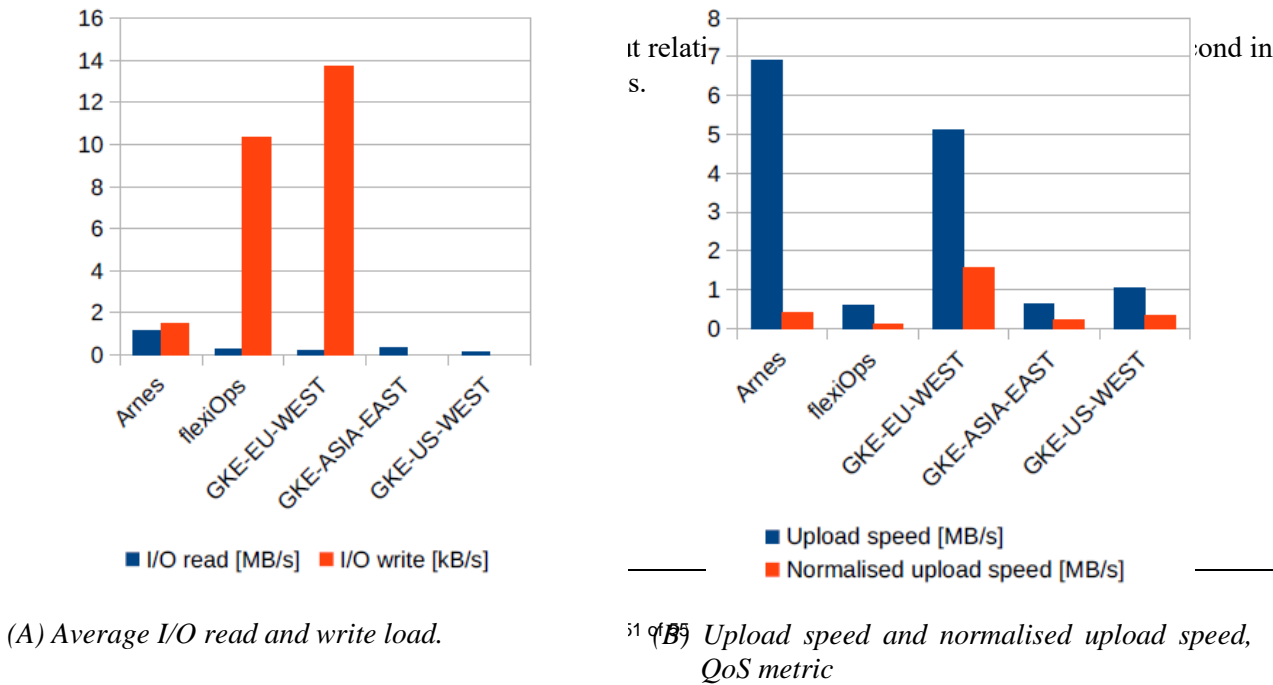


Figure 7-3 Average I/O read, write load and upload speed

### 7.2.3 QoS model

Starting with the full set of measurements presented in Section 7.2.2, which in total contains 150 examples, 30 per each of the five clusters, the Performance diagnoser Model Generator component of ASAP computes a QoS model as depicted in Figure 7-4, where the metrics are sorted according to their magnitude. Negative values suggest that the respective metrics should be minimised for better QoS, while positive metrics should be maximised. The model ranks highly I/O write, upload speed, packet loss and file size metrics. Unfortunately, only the packet loss is to some extent possible to estimate prior to execution of the application. Besides, we intentionally introduced some noise for the packet loss. The upload speed (or the goodput) metric seems reasonable selection, but is an application metric. We could replace it with throughput metric, which should relate well to the upload speed, and is measurable without having the application running. As CPU iowait metric was only noticeable on Arnes cluster, and depends on the high demand for I/O write operations, it is not a good predictor for the QoS. However, since on Arnes cluster upload alone never lasted more than four seconds, higher CPU iowait time as compared to the other clusters is somewhat reasonable. However, the QoS model does not appear to agree with this reasoning, because the corresponding value for CPU iowait is negative. It rather suggests that CPU iowait observations from the other clusters contributed more to this metric. Number of network hops metric received similar weight than CPU iowait, but is instead a better selection in our opinion.

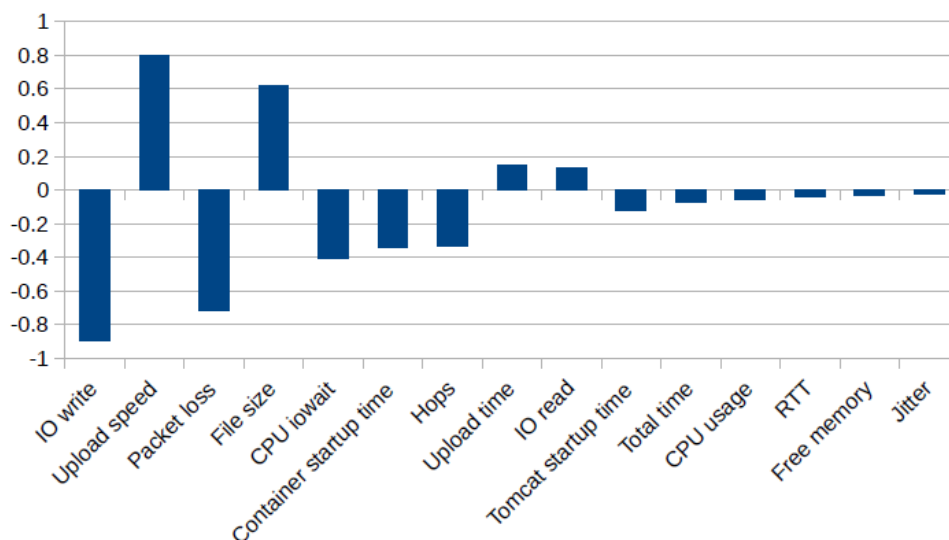


Figure 7-4 The QoS model.

In another experiment, we were interested in building QoS model from only a subset of observations and were curious what is the impact of the size of the subset and the ordering of the observations on the resulting QoS

model. In particular, the original metrics table was reduced to a table of multiples of 10 percentage of the original set, for all the multiples between 10% and 100%. The observations appeared sequentially in a cluster-major ordering, i.e. starting with one cluster and then continuing with the next. If the subset was formed of the sequential observations, smaller subsets led to very different results compared to the reference QoS model built from the full set. In this way, small subsets contained mostly observations from one cluster and were therefore biased toward a single cluster behaviour. Randomly sampled subsets tend to perform better, meaning that even small subsets resulted in a model resembling the reference one. This is important information for at least two aspects: for parallel computation of a QoS model and for updating QoS model from several sources. However, more research is needed to find out whether it is better to independently build intermediate QoS models, each from its own source, and then combine the models into a single one, or is it better to interleave the observations from various sources before computing the model.

## 8 Summary

### 8.1 Software functionality in public releases

Architecture components (defined in D2.2)	Functionality in V1	Functionality in V2	Key Performance Indicators (KPI)	Current status
Monitoring Server	Yes	Yes	Functionality	Implemented
Monitoring Agent	Yes	Yes	Functionality	Implemented
Alarm-Trigger	Yes	Yes	Functionality	Implemented
Performance diagnoser Model Generator	Yes	Yes	Functionality, robustness, updatability	Partial: needs more evaluation
Performance diagnoser Decision Maker	Yes	Yes	Functionality, speed	Partial: needs more evaluation
Setup & control	Yes	Yes	Functionality	Achieved KPI
Knowledge Base API	Yes	Yes	Functionality, data management	Achieved KPI

### 8.2 Innovation

Component (in release)	Current state of the Art	Innovation
Monitoring Server	Receiving measured metrics from the Monitoring Agent.	Addressing the requirement of containerized applications.
Monitoring Agent	Receiving monitoring data from containers.	Lightweight monitoring approach based on a non-intrusive design.
Alarm-Trigger	Checking the incoming monitoring data and notifying other components of the system.	Checking the incoming monitoring data measured in different levels such as infrastructure level, container level and application level.
Performance diagnoser Model Generator	Accurate QoS models require a vast number of measurements and uses substantial computational power; models may be also hard to update and do not adequately address dynamic nature of clouds.	Qualitative approach to QoS modelling trades less precision for higher efficiency. Model is easy to compute and update from observations and does not put restrictions upon observable metrics. It also does not require a lot of observed data.
Performance diagnoser Decision Maker	Most works focus on always-on services and geolocation information.	By using the QoS model and pre-deployment measurements, it starts containers on-demand and per-event,

		tailored to improve QoS towards end-users involved in the event.
Setup & control	Orchestrators such as Kubernetes, Mesos, Marathon	No innovation – uses Kubernetes.
Knowledge Base API	Queries per second manageable over RDF triple store. Supported semantic data providing and inference (through reasoning) approaches among specific domains.	Supporting algorithms through a RDF using semantic approach by providing filtered input data to reduce overall execution time.

## 9 Bibliography

- [1] S. Taherizadeh, V. Stankovski, J. Trnkoczy, U. Paščinski and M. Breška, “D4.1 Prototype runtime monitoring system,” SWITCH consortium, 2016.
- [2] M. Cigale, V. Stankovski, J. Trnkoczy, S. Taherizadeh, S. Gec, P. Štefanič, U. Paščinski and M. Breška, “D4.2 Design specification of the ASAP subsystem,” SWITCH consortium, 2016.
- [3] M. Cigale, V. Stankovski, J. Trnkoczy, S. Taherizadeh, S. Gec, M. Breška, P. Štefanič, P. Kochovski, J. Česnik and U. Paščinski, “D4.3 Learning strategies for self-adaptive control mechanism for time critical Cloud applications,” SWITCH consortium, 2017.
- [4] K. Evans, A. Jones, P. Martin, F. Quevedo, D. Rogers, I. Taylor, V. Stankovski, A. Taal, S. Taherizadeh, J. Trnkoczy, J. Wang, Z. Zhao, BEIA, MOG and W. T. teams, “D2.1 Technical Requirements and State of the Art Review for Time-Critical and Self-Adaptive Applications in Cloud Environments,” SWITCH consortium, 2015.
- [5] P. Štefanič, D. Kimovski, G. J. Siciu and V. Stankovski, “Non-Functional Requirements Optimisation for,” in *Cloud and Big Data Computing*, San Francisco, USA, 2017.
- [6] P. Kochovski and V. Stankovski, “[Under review] Supporting smart construction with dependable edge computing infrastructures and applications,” *Automation in Construction*, no. Special Issue on Smart Infrastructure, Construction and Building Internet of Things, 2017.
- [7] S. Taherizadeh and V. Stankovski, “Quality of Service Assurance for Internet of Things Time-Critical Cloud Applications,” in *6th International Congress on Advanced Applied Informatics (AAI 2017)*, Hamamatsu, Japan, 2017.
- [8] D. Trihinas, G. Pallis and M. D. Dikaiakos, “Jcatascopia: Monitoring elastically adaptive applications in the clou,” in *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2014.
- [9] S. Taherizadeh and V. Stankovski, “Incremental Learning from Multi-level Monitoring Data and Its Application to Component Based Software Engineering,” in *41st Annual Computer Software and Applications Conference (COMPSAC 2017)*, Turin, Italy, 2017.
- [10] S. Taherizadeh and V. Stankovski, “Dynamic Multi-level Rules for Container-based Elastic Applications. [Under review],” *The Computer Journal*, 2017.
- [11] M. . V. Butz, “Learning Classifier Systems,” in *Springer Handbook of Computational Intelligence*, Berlin, Heidelberg, Springer, 2015, pp. 961-981.
- [12] S. Gec, D. Kimovski, P. Uroš, R. Prodan and V. Stankovski, “Semantic approach for multi-objective optimisation of the ENTICE distributed Virtual Machine and container images repository,” *Concurrency Computat: Pract Exper*, no. e4264, 2017.
- [13] J. Trnkoczy, U. Paščinski, S. Gec and V. Stankovski, “SWITCH-ing from multi-tenant to event-driven videoconferencing services,” in *1ST WORKSHOP ON AUTONOMIC MANAGEMENT OF LARGE SCALE CONTAINER-BASED SYSTEMS co-located with the 2017 IEEE International Conference on Cloud and Autonomic Computing (ICCAC)*, Arizona, USA, 2017.

- [14] P. Kochovski and V. Stankovski, “Data-centric systems and dependability. [In press],” in *Security and Resilience in Intelligent Data-Centric Systems and Communication Networks*, Elsevier, 2017.
- [15] P. Štefanič, M. Cigale, A. Jones and V. Stankovski, “Quality of Service models for Micro-services and their integration into the SWITCH IDE,” in *1ST WORKSHOP ON AUTONOMIC MANAGEMENT OF LARGE SCALE CONTAINER-BASED SYSTEMS co-located with the 2017 IEEE International Conference on Cloud and Autonomic Computing (ICCAC)*, Arizona, USA, 2017.
- [16] U. Paščinski, S. Gec, J. Trnkoczy, M. Cigale and V. Stankovski, “Orchestrating Containers Across Software Defined Data Centres, [Major revision June 2017],” *Journal of Grid Computing*, no. Springer, 2017.

## Abbreviations

Abbreviation	Expansion
API	Application Programming Interface
ASAP	Autonomous Self-Adaptation Platform
CPU	Central Processing Unit
CQL	Cassandra Query Language
DRIP	Dynamic Real-time Infrastructure Planner
GCC	Google Congestion Control
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IP	Internet Protocol
JSON	JavaScript Object Notation
KB	Knowledge Base
OS	Operating System
PSNR	Peak Signal to Noise Ratio
QoE	Quality of Experience
QoS	Quality of Service
RAM	Random Access Memory
REMB	Receiver Estimated Maximum Bandwidth
REST	Representational State Transfer
RTCP	Real-time Control Protocol
RTT	Round Trip Time
SFU	Selective Forwarding Unit
SIDE	Switch Interactive Development Environment
TSDB	Time Series Database
VM	Virtual Machine
VaaS	Videoconferencing-as-a-Service
WAR	Web archive
XMPP	Extensible Messaging and Presence Protocol