Automatic, multi-grained elasticity-provisioning for the Cloud

Cloud Monitoring Tool V1

Deliverable no.: 4.1
September 30th, 2013

CELAR is funded by the European Commission DG-INFSO
Seventh Framework Programmed, Contract no.: 317790

Table of Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
DB	Database
EU	European Union
HTTP	HyperText Transfer Protocol
IaaS	Infrastructure as a Service
I/O	Input / Output
IT	Information Technology
JSON	JavaScript Object Notation
MS	Monitoring System
OS	Operating System
REST	REpresentational State Transfer
SLA	Service Level Agreement
VM	Virtual Machine
XML	eXtensible Markup Language

Deliverable Title	**Cloud Monitoring Tool V1**
Deliverable No.	**4.1**
Filename	CELAR_D4.1_finalrelease.docx
Author(s)	DemetrisTrihinas, Nicholas Loulloudes, Daniel Moldovan, StaloSofokleous, George Pallis, Marios D. Dikaiakos
Date	27.09.2013

Start of the project:1.10.2012
Duration:36 Months
Project coordinator organization:ATHENA RESEARCH AND INNOVATION CENTER IN INFORMATION COMMUNICATION & KNOWLEDGE TECHNOLOGIES (ATHENA)

Deliverable title:Cloud Monitoring Tool V1
Deliverable no.:4.1

Due date of deliverable: 30 September 2013
Actual submission date: 30 September 2013

Dissemination Level

X	PU	Public
	PP	Restricted to other programme participants (including the Commission Services)
	RE	Restricted to a group specified by the consortium (including the Commission Services)
	CO	Confidential, only for members of the consortium (including the Commission Services)

Deliverable status version control

Version	Date	Author
0.1	05.08.2013	DemetrisTrihinas
0.2	08.08.2013	DemetrisTrihinas, Nicholas Loulloudes, StaloSofokleous
0.3	28.08.2013	DemetrisTrihinas, Nicholas Loulloudes, Daniel Moldovan, StaloSofokleous
0.4	02.09.2013	DemetrisTrihinas, Nicholas Loulloudes, Daniel Moldovan, StaloSofokleous, George Pallis, Marios D. Dikaiakos
0.5	06.09.2013	DemetrisTrihinas, Nicholas Loulloudes, Daniel

		Moldovan, StaloSofokleous, George Pallis, Marios D. Dikaiakos
0.6	19.09.2013	DemetrisTrihinas, Nicholas Loulloudes, Daniel Moldovan, StaloSofokleous, George Pallis, Marios D. Dikaiakos
1.0	27.09.2013	DemetrisTrihinas, Nicholas Loulloudes, Daniel Moldovan, StaloSofokleous, George Pallis, Marios D. Dikaiakos

Abstract

The aim of this deliverable is to present the first version (v1.0) of the CELAR Cloud Information and Performance Monitor layer. This document focuses on theCELAR Monitoring System and the Multi-Level Metrics Evaluation Module. The CELAR Monitoring System (MS)is a fully automated, scalable,multi-layer,platform independent Cloud Monitoring System. The Multi-Level Metrics Evaluation Module performs composable cost evaluation and estimation when integrated with the CELAR Monitoring System to receive raw monitoring metrics, thus extending the Cloud Information and Performance Monitor layer's functionality. Both components are coupled together to support the decision-making process in CELAR with monitoring metrics that can be enriched with cost information. Initially, this document provides a study of the State of the Art in the field of Cloud Monitoring and Cost Evaluation. Described in detail are the requirements that a Cloud Monitoring System and a Multi-Level Metrics Evaluation Module must have, together with the challenges to overcome when monitoringuser applicationsin a rapidly adapting application execution environment. Furthermore, a thorough description of the CELAR Cloud Information and Performance Monitor layer is provided, focusing on the Monitoring System's architecture, the components that comprise the MS and their basic functionality available in v1.0. A detail analysis of the main functionality of the Multi-Level Evaluation Module is also provided. Finally, we elaborate on how the Cloud Information and Performance Monitor layer interacts and exchanges data with other CELAR modules, concentrating also on the strong relationship between the Monitoring System and the Multi-Level Metrics Evaluation Module.

Keywords

Cloud Computing, Cloud Monitoring, Real-Time Monitoring, Application Monitoring, Cost-Evaluation, Elastic Adaption, JCatascopia, MELA

1 Introduction

Elasticity in Cloud Computing[1], as defined by the EU Cloud Expert Group [Schubert 2012], allows the Cloud environment to -ideally automatically- assign a dynamic number of resources to a task, aiming to ensure that the amount of resources actually needed for its execution is actually provided to the respective task or service. Till this date, Elasticity in Cloud Computing is still considered a primary open research issue [Armbrust 2010, Aceto 2013, and Schubert 2010].

Many Cloud providers and systems claim that they offer elastic resource allocation but usually offer only scalability in terms of increasing availability through horizontal elasticity[2]. Scaling-down[3] or vertically[4] user applications and services to efficiently allocate the exact amount of resources are areas that either have limited success or still remain untouched by the Cloud community.

In an attempt to resolve the aforementioned open issues in the field of Cloud elasticity, the consortium aims at developing CELAR, an automated multi-grained, elastic resource provisioning platform for Cloud applications. The platform will ensure the allocation of just the right amount of computing and storage resources to Cloud applications. The innovative capabilities offered by the CELAR System will result in better utilization of the Cloud infrastructure and lower administrative costs for both the Cloud providers and Cloud end-users respectively.

Depicted in Figure 1 is the CELAR System architecture that is introduced and thoroughly described in CELAR Deliverable D1.1 [D1.1]. The CELAR System consists of three major layers: the Application Management Platform, the Elasticity Platform, and theCloud Information and Performance Monitor.

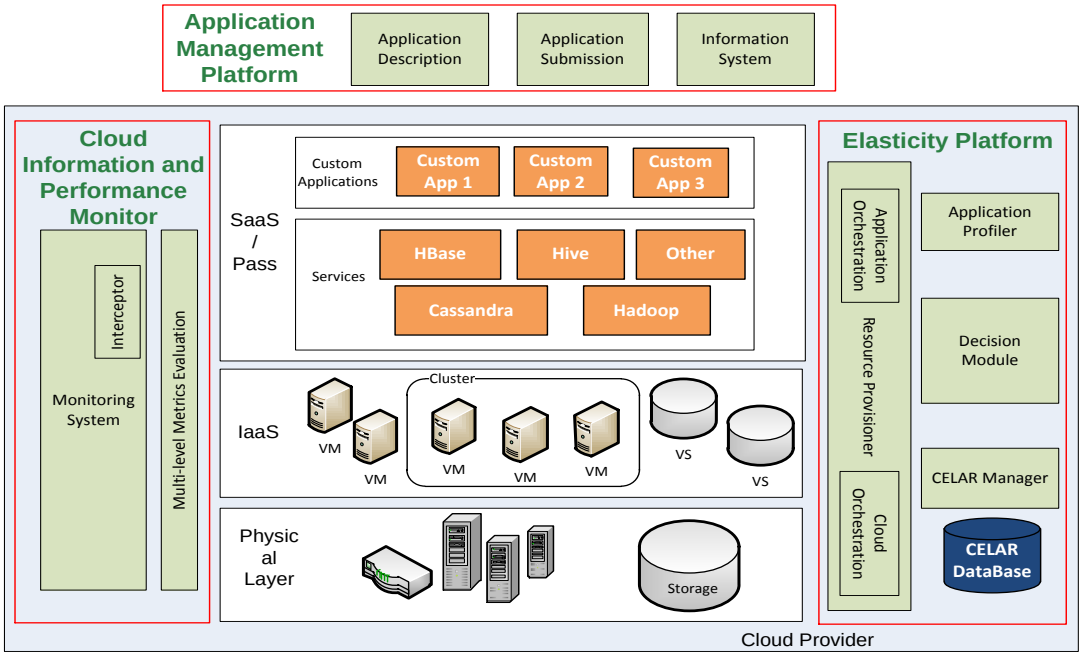

Figure 1: CELAR System Architecture

[1]Cloud computing [NIST 2011] is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage) that can be rapidly provisioned and released with minimal management effort or service provider interaction

[2]Horizontal elasticity (scale up): adding more instances of a resource to an application (e.g. adding another VM)

[3]Scale down: deducting a number of instances of a resource to an application (e.g. removing a number of VMs)

[4]Scale vertically: expanding part of the resources allocated by an instance of the application (e.g. allocating more disk storage, memory, etc. to a single instance)

The *Application Management Platform*provides an interface between Cloud end-users and the CELAR system. Specifically itserves as the information source through which the application description and elastic requirements will be made available to the resource provisioning and decision-making components. The *Elasticity Platform* uses intelligent decision-making algorithms to elastically allocate the correct amount of resources for the deployed user applications and to reconfigure the deployment accordingly. To support the decision-making process, accurate real-time monitoring metrics must be collected from the Cloud infrastructure and the deployed user applications. These collected metrics must be pre-processed, structured and enriched, so that the decision-making algorithms can understand them and take appropriate decisions.

The *Cloud Information and Performance Monitor layer* consists of (i) a scalable, multi-layer, real-timeCloud Monitoringand InformationSystem that will provide the decision-making module with accurate, real-time, monitoring metrics collected from multiple levels of the Cloud infrastructure and performance metrics from the deployed user applications, and (ii) a Multi-Level Metrics Evaluation Module that will enrich monitoring metrics with cost information.

Monitoring applications residing in large-scale infrastructures, such as Clouds, is essential for detecting and understanding the behavior of the application under various circumstances and unexpected workloads, preventing faults and security breaches and also detecting performance issues that could not be detected in the development cycle.

In this deliverable we present the**Cloud Information and Performance Monitor layer**, focusing on the CELAR Monitoring System and the Multi-Level Metrics Evaluation Module which are currently under development.

The **CELAR Monitoring System (MS)**is a fully automated, scalable, multi-layer, platform independent Cloud Monitoring System that aims at supporting automated multi-grained Cloud platforms which offer elastic resource provisioning for deployed Cloud applications.The CELAR MS can be utilized to collect monitoring metrics from multiple layers of the underlying infrastructure as well as performance metrics from deployed applications, and subsequently distribute them to subscribed users and platform operators. In contrast to existing Cloud monitoring systems, or systems with workarounds to support Cloud monitoring, the CELAR MS provides self-adaptive mechanisms to minimize the intrusiveness of its presence, and enhanced filtering to reduce communication costs for metric distribution. The Monitoring System's elastic properties are further enhanced with the ability to dynamically add/remove monitoring instances located on multiple levels of the infrastructure without the need to restart the monitoring system. Furthermore, the CELAR MS supports aggregation and grouping of low-level metrics to generate high-level metrics at runtime.

The **Multi-Level Metrics Evaluation Module** provides the CELAR System with composable cost evaluation and estimation. Cost information, in most cases, cannot be monitored directly, as pricing schemes are dependent on marketing strategies of individual Cloud providers. Thus, for evaluating cost, bothCloud pricing schemes and monitoring metrics must be utilizedtoprovide structured, aggregated and analyzed cost-wise information, such that the Decision Module can understand and take appropriate decisions. To achieve this, the Multi-Level Metrics Evaluation Module will process monitoring metrics retrieved from multiple levels of the Cloud infrastructure, evaluate cost and aggregate the monitoring data according to the application structure model (D5.1 [D5.1 - Section 4.1.1]), providing a complete overview of the application behavior.

Moreover, the Multi-Level Metrics Evaluation Module will provide estimations about the future application cost, quality and resource utilization, providing support for the Decision Module to take proactive decisions.

1.1 Purpose of this Document

This document presents the first version (v1.0) of the Cloud Information and Performance Monitor layer and the progress made until month 12 in Work Package 4 of the CELAR Project. Specifically, this deliverable mainly focuses on describing the CELAR Monitoring System. We present the challenges that must be addressed when monitoring elastically adaptive applications deployed on Cloud infrastructures and provide a detailed description of the architecture, the main components and the implementation of the CELAR Monitoring System (Task 4.1). Secondly, we present an initial analysis of how cost-evaluation is performed and provide a description and the architecture of the Multi-Level Metrics Evaluation Module (Task 4.2). The functional and non-functional requirements are defined, and all the necessary communication and data exchange between other CELAR modules is also described. Special emphasis is given to present how the CELAR Monitoring System integrates with the Multi-Level Metrics Evaluation Module to provide the CELAR System, and consequently the decision-making process, with monitoring metrics enriched with cost information.

1.2 Document Structure

The rest of the document is structured as follows: Section 2 presents a study of the State of the Art in monitoring elastic applications in Cloud infrastructures and cost-evaluation.Section 3, presents the actors, the use-cases driven by these actors and the respective functional and non-functional requirementsof the Monitoring System and the Multi-Level Metrics Evaluation Module. Section 4 focuses on the analysis and architecture of the Monitoring System and the Multi-Level Metrics Evaluation Module. Sections 5 and 6, document all the implementation specific details of the Monitoring System and the Multi-Level Metrics Evaluation Module respectively. Section 7, elaborates on the interactions and integration of the Cloud Information and Performance Monitor layer with other CELAR modules. The last section concludes the deliverable and outlines the future work.

2 State of the Art in Cloud Monitoring

Over the past decade, Cloud computing has rapidly become a widely accepted paradigm with core concepts such as elasticity, scalability and on-demand automatic resource provisioning emerging as next generation Cloud service -must have- properties. Justifiably, the concept of Cloud computing is dominating the interests of organizations, as they seek to empower their business units to promptly address market opportunities, while at the same time aiming to minimize their running IT costs. Automatically provisioning resources for applications deployed on Cloud infrastructures is not a trivial task, requiring the deployed applications and the platform, to be constantly monitored, capturing information at different levels and time granularity. Aceto et al. [Aceto 2013] advocates that accurate and fine-grained monitoring activities are required to efficiently operate Cloud platforms and to manage their increasing complexity.

In the following sections, we present a study of the State of the Art in the field of Cloud monitoring and cost evaluation.

2.1 Cloud and Application Monitoring Tools

Commercial monitoring platforms such as Amazon's CloudWatch [CloudWatch], Windows Azure's Cloud Services [Azure] and RackSpace's Cloud Monitor [RackSpace] provide monitoring as a service to their customers. Similarly, VMware offers HypericHQ [HypericHQ] for organizations wanting to monitor their virtual infrastructure or private Cloud running VMware's vCenterand vSphere [vCenter, vSphere]. Customers can select from a vast collection of metrics, from low-level metrics such as CPU, memory, network usage, to high-level application metrics such as response time, latency, availability, for dedicated provider services, accessible via RESTful API's or a graphical web interface. The metric collecting methodology though is considered proprietary and is not revealed. At the same time, customers can set thresholds and will receive email or sms alerts when these thresholds have been violated. The aforementioned Cloud platforms also enable their customers to inject custom metrics generated by their applications and services to the monitoring system, though, metrics are limited to fixed collecting time intervals (e.g. currently CloudWatch's minimum collecting period is 5min, requiring for a premium fee to lower the period to 1min).

CloudWatch can be integrated with Amazon's AutoScaling [AutoScaling] to provide elasticity, allowing for Amazon EC2 instances to be seamlessly and automatically added when demand increases, based upon Boolean formulas given manually by users (e.g. AvgCPU10minTotal > 70%) and metrics collected by CloudWatch. Windows Azure can be integrated with third party Paraleap'sAzureWatch [AzureWatch] to provide elasticity in a similar manner to CloudWatch. Despite the fact that the previous monitoring tools are easy to use, fully documented and well-integrated with the underlying platform, the biggest disadvantage they impose is that they cannot be used on any Cloud platform, thus bounding them to their providers' infrastructure. This is not the case for the CELAR Monitoring System. CELAR aims at providing a genericand platform independent monitoring system that will not be limited to the consortium IaaS providers[5].

Ganglia [Massie 2004], Nagios [Nagios], Zabbix [Zabbix], MonALISA [Newman 2003] and GridICE [Andreozzi 2005] are existing open-source; robust, monitoring tools used traditionally by system administrators to monitor slowly changing and fixed large-scale

[5] GRNET's Okeanos Public Cloud [Okeanos] and Flexiant's FCO Cloud Orchestrator [FCO]

distributed infrastructures, such as Computing Grids or Clusters. Cloud providers tend to adopt such solutions to monitor their infrastructures as well. However, Cloud platforms are inherently more complex than grid infrastructures, consisting of multiple layers and service paradigms (IaaS, PaaS, SaaS) providing users and applications with "on-demand" resources through a, theoretically, infinite pool of virtual resources making the previously referred monitoring tools unsuitable for addressing a rapidly adapting and dynamic Cloud infrastructure where for example, a virtual instance is deployed for several minutes (or hours) on one physical node and after a short interval it can migrate to another node.

Nevertheless, expansions to monitoring tools, such as Nagios or Ganglia, have been proposed and can be utilized to monitor virtual infrastructures (e.g., Xen [Xen] and VMware [VMware]). For instance, sFlow [sFlow], a network monitoring tool, can be integrated with Ganglia to monitor virtualized environments, focusing mainly on both VM clusters and Java Virtual Machines. Carvalho et al. [Carvalho 2011] propose the use of passive checks by each physical host to notify the *Nagios Server* via a push notification mechanism about the virtual instances currently running on the system. Whereas, Katsaros et al. [Katsaros 2011] extends Nagios through the implementation of *NEB2REST*, which is a RESTful*Event Brokering* module utilized to provide elastic capabilities through an abstraction layer between monitoring agents, pushing collected metrics, to the management layer.

Clayman et al. [Clayman 2010] propose *Lattice*, a scalable Cloud monitoring framework which succeeds in monitoring not only physical hosts but also virtual resources. *Lattice* can be utilized to monitor elastically adapting environments. The process of determining the existence of new virtual instances in an application deployment is performed at the hypervisor level, where a*Hypervisor Controller* is deployed as the responsible entity for retrieving (at fixed intervals) a list of running virtual instances from the hypervisor, detecting if a new VM is added/removed, consequently adding/removing metric collecting probes to/from the virtual instance. CELAR aims at providing a multi-layer monitoring system, capable of collecting heterogeneous types of information of different granularity across multiple levels (i.e. VM, Virtual Cluster, Cloud, Application level) of the Cloud infrastructure. In contrast to *Lattice* and the proposed Nagios extensions, to support elasticity, the CELAR MS does not require special entities deployed on physical nodes, nor does it need to request information from the hypervisor regarding the current running VMs. By utilizing a variation of the *publish and subscribe*[Eugster 2003] message pattern, the CELAR MS is able to automatically detect and adopt newly deployed monitoring entities, thus overcoming the need of restarting the whole system or any of its sub-components that shadows many of the previously mentioned systems.

Wang et al. [Wang 2012] propose VScope, a flexible and scalable (performance tests conducted with 1000+ VMs) monitoring and analysis middleware for troubleshooting multi-tier data intensive applications residing on Cloud infrastructures. VScope can operate on any set of nodes or software components, thus metrics can be collected within a tier, via aggregation functions from across multiple tiers, or across different software levels. In contrast to VScope which targets data-intensive Cloud applications, Montes et al. [Montes 2013] propose GMonE, a general-purpose Cloud monitoring tool applicable to all areas of Cloud monitoring. GMonE, allows for monitoring instances to be deployed on any level of the Cloud (physical, hypervisor, virtualization, platform and application level) and provides a pluggable model where users can add/remove

monitoring instances (named GMonEMon) and their own custom metrics to the MS. However, it is not clear if instances or metrics can be added/removed at runtime and if metrics must be collected at regular time intervals.

Similar to VScope and GMonE, Kutate et al. [Kutare 2011] propose Monalytics, an online monitoring and application behavior troubleshooting system that targets virtualized systems and Cloud infrastructures. The developers of Monalytics though, propose as a novelty that monitoring instances should impose an amount of logic in order to analyze, in place, the usefulness of collected raw metrics rather than distributing over the network metrics of little significance. Meng et al. [Meng 2011] propose a violation-likelihood sampling model which offers the option and flexibility to dynamically adjust the monitoring intensity based on how likely a state (threshold) violation will be detected. The CELAR MS aims at providing monitoring instances equipped with intelligent light-weight metric collectors that can analyze in place raw data. This feature enables the CELAR MS to adapt the monitoring intensity when stable phases are detected and filtering any values of little significance, minimizing computation and communication overhead respectively.

To bridge the gap between collected raw low-level monitoring metrics and service level SLA required metrics, Emeakaroha et al. [Emeakaroha 2010] propose *LoM2HiS*. *LoM2HiS* is a novel framework which is utilized for mapping low-level system metrics collected from existing monitoring tools (Ganglia), via aggregation and/or composition to high-level application specific SLA parameters (e.g. availability, response time, etc.). The CELAR MS aims at providing a subscription rule mechanism, where Application Users and internal CELAR modules, such as the Multi-Level Metrics Evaluation Module, can subscribe to aggregated metrics collected from any level of the Cloud and also compose high-level metrics from low-level metrics via a directive-based subscription language.

Table 1 presents a synopsis of the monitoring tools presented in this section of the State of Art with emphasis on their features. *Infrastructure Independent* refers to the ability of the monitoring system to be provider independent thereby deployable on any Cloud platform.*Physical*, *Virtualization* and *Application Layer* reflects the ability of the MS to collect metrics from these layers.*Elasticity* refers to the ability of the monitoring system to adapt at runtime to changes occurring to the application topology when elasticity actions are enforced and *Inject Custom Metrics to MS* refers to the ability to add at runtime custom metrics to the monitoring system without restarting the monitoring process.*Aggregation* refers to the ability of aggregating metrics for a period of time. *Adaptive Filtering* refers to the ability of adapting the filter window when filtering monitoring metric values, while *Adaptive Sampling* is the ability to adapt the intensity of the metric collecting mechanism based on previous metric values. *Subscribing to high-level metrics* refers to the ability of the MS to create high-level metrics from low-level collected metrics via subscription rules.

Monitoring System	Infrastructure Independent	Physical Layer	Virtualization Layer	Application Layer	Elasticity	Inject Custom Metrics to MS	Aggregation	Adaptive Filtering	Adaptive Sampling	Subscribe to High-Level Metrics
CloudWatch + AutoScaling			X	X	X	X	X			
Windows Azure Cloud Services + AzureWatch			X	X	X	X	X			
RackSpace Cloud Monitor			X	X		X	X			
VMware HypericHQ			X							
Ganglia	X	X								
Ganglia + sFlow	X	X	X							
Nagios	X	X								
Nagios (Carvalho et al.)	X	X	X							
Nagios + NEB2REST	X	X	X							
Zabbix	X	X								
GridICE		X								
MonALISA		X								
Lattice	X	X	X		X					
VScope	X	X	X	X		X				
Monalytics	X	X	X	X		X	X		X	
GMonE	X	X	X	X		X	X			
Meng et al.		X	X					X	X	
LoM2HiS		X	X				X			X
CELAR MS	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

Table 1: Cloud Monitoring System Comparison

2.2 Cost Evaluation

In [Chaisiri 2012] the authors introduce a mechanism for provisioning virtual machines based on the predicted cost. The approach relies on a simple presented cost evaluation mechanism, taking into account the cost of reserving, using, and requesting virtual machines on-demand. Other cost elements such as network data transfer are ignored. The authors of [Al-Kiswany 2013] present a solution for comparing the cost of data services offered by different Cloud providers, taking into account service staleness and accuracy. To estimate cost, the authors use the amount of CPU, network, and disk capacity consumed to keep the sharing at the desired staleness. This approach is applicable only to single-services, while we estimate cost for complex applications using many services with different characteristics.

A composable cost estimation framework for scientific computing applications is presented in [Truong 2010]. The authors compose cost from cost per virtual machine, per storage size, and transferred data in/out of the Cloud. Cost is evaluated for each scientific workflow activity, depending on the estimated activity execution time. Furthermore, the authors show that it is not trivial to determine some type of costs due to the complex dependency among services in the Cloud or the overhead of monitoring the cost elements (e.g., monitoring all I/O calls in an application could introduce severe performance overhead). CELAR aims at providing a multi-level cost monitoring and

evaluation solution, providing cost evaluation at different granularities, to be used by different actors with different expertise levels.

A solution for evaluating cost of transactions in Cloud application is presented in [Buell 2011]. Cost is evaluated per application transaction, and obtained by composing the cost for processing time (cost/nanosecond), storage (bytes stored), bandwidth used in data transfer in/out of the Cloud, and cost of third party services used by the transaction. The complexity of estimating cost of applications running in Cloud environments is further highlighted in [Khanafer 2013], the authors presenting a mechanism to answer the "buy or rent" Cloud services problem. A Cloud file system is used in evaluating the approach, showing how difficult is even for a small system to predict cost, as different Cloud services have very different pricing schemes, the total cost being also heavily influenced by the workload pattern.

Konstantinou et al. [Konstantinou 2013] propose COCCUS a modular system dedicated for cost-aware database query execution, adaptive query charge and optimization of Cloud data services. The audience can set their queries along with their execution preferences and budget constraints, while COCCUS adaptively determines query charge and manages secondary data structures according to various economic policies. A framework for monitoring, estimating cost and provisioning Cloud services is presented by Sharma et al. [Sharma 2011]. Unlike the majority of approaches in the field, the authors also consider run-time elasticity of Cloud application, reconfiguring the application at run-time based on cost and performance criteria. While comprehensive, this approach does not provide any high-level insight regarding cost (e.g., the cost for a data end or for a business end) or estimating cost at the virtual machine level.In CELAR we aim at providing a multi-level cost composition, from cost per individual resource (e.g., network, storage), to cost per composite component, and overall application cost. This composition enables multi-level cost analysis, such as determining which application component is the most expensive to run, and why (e.g., many I/O operations, many used virtual machines, or large network data transfer).

3 Cloud Information and Performance Monitor Layer Requirements

The role of the CELAR Monitoring System, as a key CELAR System module, is to collect, process and report monitoring metrics. Monitoring information is gathered at runtime, and includes both metrics from various levels of the underlying Cloud infrastructure as well as application performance metrics. Metrics are then processed and distributed to interested CELAR modules, such as the Decision Module in order to decide whether elastic actions should be taken into consideration or to the Multi-Level Metrics Evaluation Module to further evaluate these metrics and enrich them with cost information.

In this section of the document we present the **actors**, **use cases**, **functional** and **non-functionalrequirements** for the Cloud Information and Performance Monitor layer, in-line with the use cases presented in Deliverable D1.1 [D1.1 Section 2.3.2]. The first sub-section presents the different actors that interact with the Cloud Information and Performance Monitor layer. The second sub-section presents the Cloud Information and Performance Monitor layeruse cases, further explaining how it interacts with the previouslydefined actors. Finally, in the last sub-section a detailed description of the functional and non-functional requirements for the Cloud Information and Performance Monitor layer is provided.

3.1 Actors

The following CELAR modules and sub-modules interact with the Cloud Information and Performance Monitor layer and can be considered as actors:

IaaS Provider

Raw monitoring information is obtained by various levels of the underlying IaaSplatform such as the VM and Virtual Cluster level where metrics are gathered and reported to the MS via metric collectors that reside on the deployed VMs.

Cloud Application

Application performance metrics (i.e. latency, throughput, availability, etc.)specific to the deployed Cloud applications are also collected and reported to the MS.

c-Eclipse MS Visualization Tool

TheMS Visualization Tool is one of the main components of the c-Eclipse Application Management Platform (presented in Deliverable D2.1 [D2.1]) and it is a vital component after users submit their applications for deployment on a Cloud infrastructure.The MS Visualization Tool receives monitoring data from the CELAR Monitoring System, including resource related metrics (i.e. CPU, memory usage, etc.) and specific application performance metrics, and presents them graphically to the Application User[6]. The Application User may also request to receive monitoring data enriched with cost information or future cost estimations from the Multi-Level Metrics Evaluation Module. In this way, the Application User is able to observe his application during its lifetime and intervene, if required, by re-configuring his deployment.

[6]The Application User (defined in Deliverable D1.1 - Section 2.3) is a person that has knowledge about the application currently executed over the CELAR platform. His aim is to describe, deploy and monitor his application over CELAR

Decision Module

The Decision Module (described in Deliverable D5.1 [D5.1]) is a core module of the CELAR System. The Decision Module analyzes all the information gathered by the Monitoring System. Intelligent algorithms decide based upon the received metrics, if there is a need to elastically resize the resources provisioned to the application or if there is a need to reconfigure the application deployment. The Decision Module interacts with the MS by periodically requesting metrics, but it can also be triggered by the MS when an out-of-the-ordinary behavior is detected on the monitored metrics (i.e. a sudden spike on the user's request rate).The Decision Modulemay alsoretrieve from the Multi-Level Metrics Evaluation Modulemonitoring information enriched with cost information aggregated after the application structure. Moreover, the Decision Module can retrieve from the Multi-Level Metrics Evaluation Module estimations regarding the future values for the Cloud application's cost, quality, and resource usage, which can be utilized to improve the decision process.

CELAR Manager

The CELAR Manager, described in Deliverable D1.1 [D1.1], is considered as the communication end-point through which, the c-Eclipse platform interacts with the CELAR System. The CELAR Manager, upon receiving a user request from the c-Eclipse platform forwards the request to the interested CELAR module(s). Application Users have the ability to write their own metric collectors (Probes) to gather information specific to their application[7]. When the CELAR Manager receives through c-Eclipse a user request for deploying a new Monitoring Probe, it notifies the Monitoring System about the request in order for the new Probe to be injected in the specified virtual instances of the application. The CELAR Manager also interacts with the Multi-Level Metrics Evaluation since the Multi-Level Metrics Evaluation Module will have to retrieve the current topology of the Cloud application (i.e. the current instantiated VMs) which is stored in the CELAR DataBase, based on which the monitoring data will be structured, and analyzed in terms of cost, quality and resources. Metric composition rules for aggregating monitoring data, defined when submitting the application to CELAR, will also be retrieved from the CELAR DataBase using the CELAR Manager.

Multi-Level Metric Evaluation Module[8]

The functionality of the Cloud Information and Performance Monitor layer is further extended, when the Monitoring System is integrated with the Multi-Level Metrics Evaluation Module,to perform multi-level cost compositionby processing monitoring metrics, and enriching these metrics with cost information according to the application structure. The enriched metrics are then made available for consumption by any interested parties, such as the Decision Module or the Application Users via the c-Eclipse MS Visualization Tool. The functionality of the Multi-Level Metric Evaluation Module also includes computing estimations on the future values for the cost, quality and resource usage of applications deployed on the Cloud infrastructure.

[7]We adopt the naming Probe Developer for users that develop their own custom Monitoring Probes. For further analysis see Section 5.2

[8] The Multi-Level Metrics Evaluation Module is considered an actor for the Monitoring System Use Cases

CELAR Monitoring System[9]

The Multi-Level Metrics Evaluation Module requires fresh monitoring metrics from the CELAR Monitoring System originating from multiple levels of the Cloud infrastructure and performance metrics from the deployed applications which in turn will be aggregated according to the application structure and analyzed.

3.2 Use Cases

The following Table and Figure provide a detail overview of the Monitoring System use cases:

	Use Case Name	Actor	Description
1	Collect raw infrastructure monitoring metrics	IaaS Provider	The MS collects raw monitoring metrics from various levels of the Cloud infrastructure (VM, Virtual Cluster, Cloud level, etc.)
2	Collect application performance monitoring metrics	Cloud Application	The MS collects performance monitoring metrics from the deployed Cloud Applications
3	Distribute fresh monitoring metrics	c-Eclipse MS Visualization Tool, Decision Module, Multi-Level Metrics Evaluation Module	The MS, upon request, distributes (using either a push or pull delivery mechanism) monitoring metrics to Application Users via the c-Eclipse MS Visualization Tool, the Decision Module and the Multi-Level Metrics Evaluation Module
4	Trigger New Decision Cycle	Decision Module	The Decision Module can subscribe to receive notificationswhenever the MS detects any abnormal application behavior which is reflected in the monitored metrics
5	Deploy custom Probe	CELAR Manager	c-Eclipse users can define custom Monitoring Probes that will be injected by the CELAR Manager to the MS instances on the selected application VMs at runtime
6	Subscribe to high-level metrics	c-Eclipse MS Visualization tool, Decision Module, Multi-Level Metrics Evaluation Module	c-Eclipse users (via the MS Visualization Tool), the Decision Module and the Multi-Level Metrics Evaluation Module can subscribe to receive from the MS aggregated metrics or define new high-level metrics from pre-existing collected low-level metrics

Table 2: CELAR Monitoring System Use Cases

[9] The CELAR Monitoring System is considered an actor for the Multi-Level Metrics Evaluation Module Use Cases

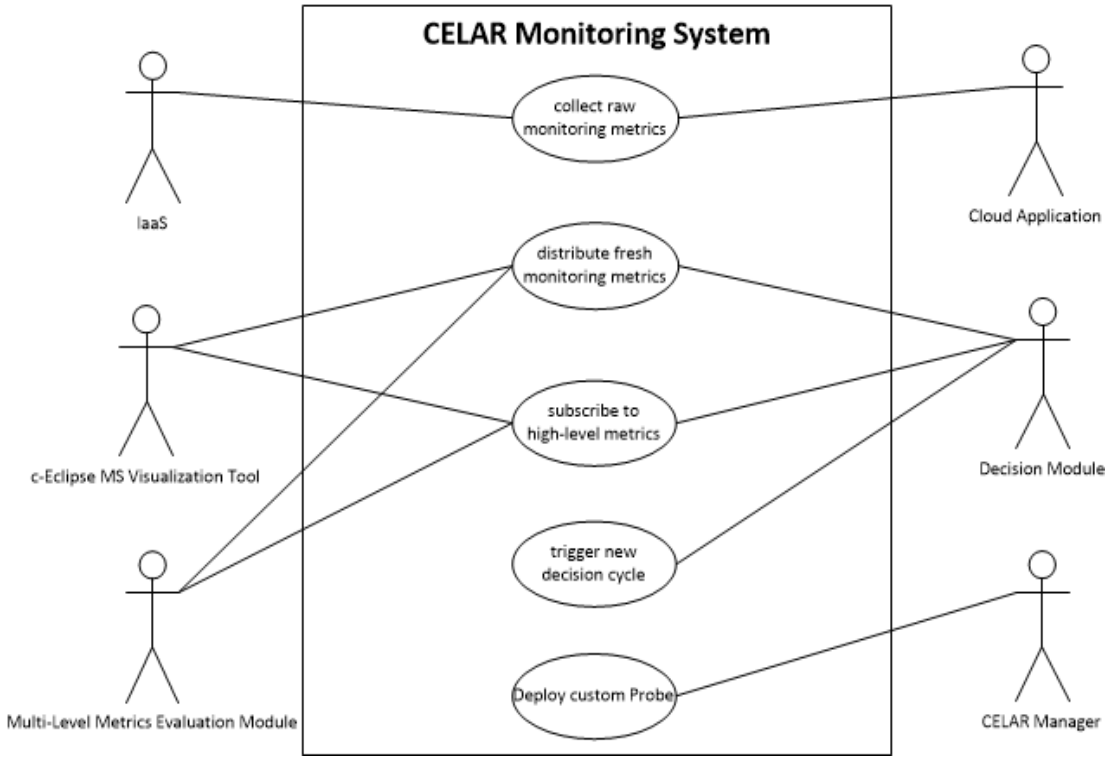

Figure 2: Monitoring System Use Case Diagram

The following Table and Figure provide a detail overview of the Multi-Level Metrics Evaluation Module use cases:

	Use Case Name	Actor	Description
1	Subscribe to monitoring metrics stream	CELAR Monitoring System	The Multi-Level Metrics Evaluation Module can subscribe to receive from the MS (i) raw monitoring metrics or (ii) aggregated metrics or (iii) define new high-level metrics from pre-existing collected low-level metrics
2	Get application topology	CELAR Manager	The Multi-Level Metrics Evaluation Module requires the application topology which is retrieved from the CELAR DataBase via an API call request to the CELAR Manager
3	Distribute cost-enriched monitoring metrics	Decision Module, c-Eclipse MS Visualization Tool	The Multi-Level Metrics Evaluation Module according to the application structure, processes monitoring data retrieved from the MS, enriching it with cost information. This information is in turn provided to the Decision Module, and/or to the c-Eclipse MS Visualization tool to be displayed to the Application User
4	Distribute cost, quality and resource	Decision Module, c-Eclipse MS	The Multi-Level Metrics Evaluation Module computes estimations on the

	estimations	Visualization Tool	future values for the cost, quality and resource usage of the Cloud application. These estimations can be used by the Decision Module in predicting the future behavior of the application and taking proactive decisions. Estimations on the predicted application behavior can also be displayed to the Application User

Table 3: Multi-Level Metrics Evaluation Module Use Cases

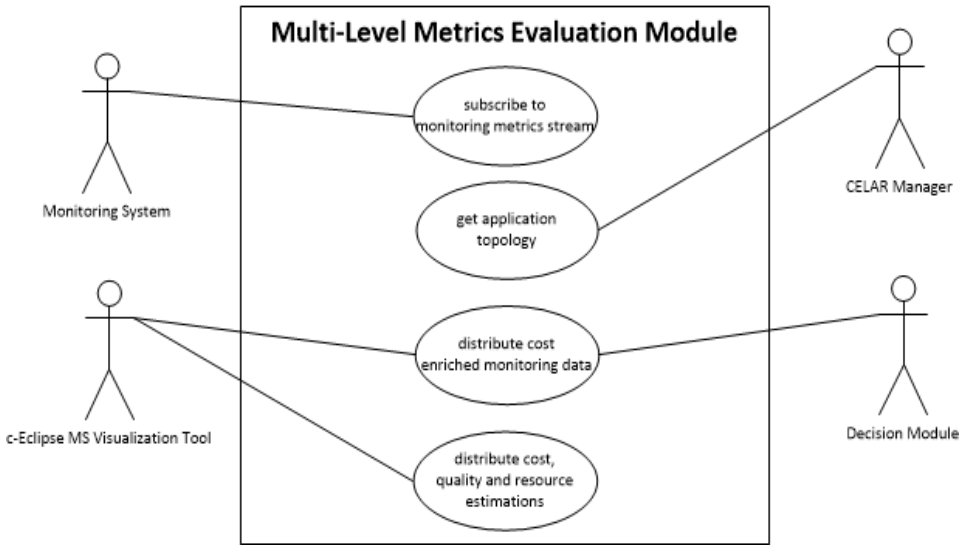

Figure 3: Multi-Level Metrics Evaluation Module Use Cases

3.3 Functional Requirements

In this section we define the functional requirements of the Cloud Information and Performance Monitor which are derived from the previously described use cases along with the general CELAR use cases as described in the Deliverable 1.1.

3.3.1 CELAR Monitoring System

Collect resource related metrics

The Monitoring System must be able to deploy on different levels of the Cloud infrastructure monitoring instances that can collect resource related raw monitoring information (i.e. CPU usage, allocated memory, etc.). The CELAR MS should adopt a hybrid monitoring approach utilizing passive collecting mechanisms of existing open source monitoring tools to collect raw monitoring data from the infrastructure components and also provide its own collecting mechanisms gathering any metrics that the existing monitoring tools cannot provide.

Collect application level metrics

In continuation to the above requirement, the MS metric collection mechanisms must also be able to provide the ability to gather monitoring information related to the performance of the users' running application(s). Monitoring metrics highly related to

the performance of a deployed application are latency, throughput, response time, availability, etc.

Deploy custom monitoring Probes

The MS needs to provide a mechanism that allows Application Developers to write and deploy their own custom monitoring probes. These probesshould be created in a way that they can be plugged directly and seamlessly into the Monitoring System to monitor user-specified application metrics that are not currently available by the MS.

Add/Remove monitoring instances at runtime

In order for CELAR to provide elasticity where the number of virtual instances deployed for the user application varies in time, monitoring instances must be easily configurable and deployed at runtime without the need of reconfiguring (or restarting) the whole Monitoring System. When a new virtual instance is added to the resources of an application, a new monitoring instance must be configured and added to this VM while the MS must also be notified of its existence. In a similar manner, the MS must also be aware when a monitoring instance has been removed due to the removal of the allocated VM from the resources of the application.

Monitoring metric deliverymechanisms

The MS needs to support both push and pull delivery models to serve the different types of metric consumers (Decision Module, Application Users, etc.) that are interested in different types of monitoring information.

- **Pull model**: interested parties can request for a specific metric by querying the Monitoring System. For example, an Application User may want to know what operating system (OS) is running on a specific virtual instance, or what is the current memory consumption of a group of instances, etc.
- **Push model**: interested parties subscribe to be notified when fresh data is made available. For instance, a simple example of this delivery mechanism is the Decision Module subscribing to receive fresh metrics whenever made available concerning the CPU usage of a specific VM or Virtual Cluster.

3.3.2 Multi-Level Metrics Evaluation Module

Perform composable cost-evaluation

The Multi-Level Metrics Evaluation Module needs to compute the instantaneous running cost at each level of the application structure[10], from the cost of running each virtual machine, to the cost of the overall application. The multi-level cost composition is crucial, as it enables the Decision Module to reason given information from multiple levels.

Convert monitoring data

To apply different cloud pricing schemes, the monitoring data might need to be converted, as pricing schemes might target different measurement units or different metric names (such as “price per data transferred outside the Cloud provider”).

[10] The Application Structure is presented in Section 4.2

Aggregate monitoring data based upon the application structure model

The Decision Module needs monitoring data aggregated according to the application structure model, in order to have a higher data understanding (e.g., a "high" cost is associated to all the VMs of a particular Application Component). As this aggregation uses information about the application structure that is not available to the Monitoring System, the Multi-Level Metrics Evaluation Module must implement this functionality.

3.4 Non-Functional Requirements

In this section we define the non-functional requirements resulting from the above functional requirements for the Cloud Information and Performance Monitor layer.

3.4.1 CELAR Monitoring System

Infrastructure Independence

The Monitoring System must be deployable and functional on any Cloud infrastructure since the CELAR System will be utilized by a number of different Cloud Providers. To cope with this, the MS must be generic in terms that monitoring instances must be deployable on different Cloud Providers and at different levels of the Cloud infrastructure. The metric collecting mechanisms should also not depend on the infrastructure where the MS is installed on.

Scalability

The Monitoring System must be scalable in order to handle a large number of metric producers on different Cloud levels while simultaneously being able to handle a large number of metric consumers. The MS should not be fragmentedby the number of running instancesor the number of monitoring probes deployed on each instance.

Non-Intrusiveness

The MS must not interfere with the system or the application(s) being monitored, and must not consume excessive resources from the virtualized instances. To address this, the MS must have a minimal run-time impact (in terms of CPU, memory and network bandwidth consumption) in order to not affect the performance of the applications running on the Cloud infrastructure.

Elasticity and Adaptability

The MS must be elastic in terms of adjusting dynamically to changes occurring to the monitoring instances in order to monitor virtual resources that are created and deducted by elasticity actions on the Cloud infrastructure. The MS must also be adaptable in terms of computation, network and storage load it imposes on user-paid resources in order to remain non-intrusive, as stated above, when elasticity actions are enforced.

Extensibility

The MS must be extensible by providing the ability to include new functionality and metrics. Thus, the MS needs to be adaptable, flexible and extensible in order to support and utilize the expanding functionality.

Reliability,Robustness and Fault-Tolerance

In a distributed large-infrastructure faults and unexpected errors are bound to present. For this reason, the MS must assure that if a monitoring instance is assumed offline or unavailable due to unexpected events, the function of the MS or the Cloud infrastructure should not be affected. Also, the metric collection should not fail or be affected by stalls or errors introduced by the source of the metrics.

Near Real-Time and Accuracy

The MS must be able to collect (near) real-time raw metrics from various levels of the Cloud infrastructure and deliver accurate processed metrics to the interested metric consumers.

3.4.2 Multi-Level Metrics Evaluation Module

Scalability

The module must be able to aggregate, analyze and provide estimations for Cloud applications ranging from small systems using few virtual machines, to massively distributed applications, running on hundreds if not more virtual machines of different types, in different Cloud availability regions or virtual machine clusters.

Robustness

For elastic applications that allocate (or de-allocate) Cloud services on demand (i.e., virtual machines, reservation schemes, network, etc.), during the execution of elasticity actions (e.g., scaling in/out), monitoring data might be inaccurate or unavailable (e.g., during software bootstrapping, a new virtual machine might appear in the application structure, but would not provide monitoring data). The Multi-Level Metrics Evaluation Module must cope with such abnormalities in input data and continue its operation.

Reusability

The Multi-Level Metrics Evaluation Module must be able to aggregate, analyze and provide estimations for heterogeneous Cloud applications having different structures and providing different monitoring data. Moreover, the module must be able to apply different cloud pricing schemes, evaluating and estimating the cost of running any cloud application in any Cloud that uses CELAR.

Flexibility

The Multi-Level Metrics Evaluation Module must have a flexible mechanism for configuration (e.g., application structure, Cloud pricing schemes), and monitoring data retrieval. This will allow the module to be integrated both within CELAR, and deployed separately over other non-CELAR monitoring and configuration systems, enlarging the applicability and usability of the Multi-Level Metrics Evaluation Module, by enabling third parties to use the CELAR features (i.e. modules) that are most relevant for their use cases.

4 Cloud Information and Performance Monitor LayerAnalysis

In this chapter we further analyze the requirements presented in the previous chapter and elaborate on the features that the components of the Cloud Information and Performance Monitor layer should incorporate. Furthermore, we provide an insight to the architecture and components of the CELAR Monitoring System. Finally, we present an analysis of the Multi-Level Metrics Evaluation Module.

4.1 Cloud Monitoring Analysis

As discussed in Section 2, Cloud Computing is evolving as a dominant business model for resource provisioning over the internet. Inevitably, monitoring applications deployed on large-scale distributed infrastructures, such as Clouds, is essential for capturing the performance and understanding the behavior of an application under various circumstances and possibly unexpected workloads. For instance, in web service scenario, clients dislike when the loading time of a web page is very long, or when a resource (i.e. video, mp3, file, etc.) downloads slowly. Organizations expect from a Cloud monitoring system to detect performance issues in order to prevent faults, security breaches or network problems and tune their application accordingly. Such, poor performance may be caused due to faulty code introduced during the development cycle or by poorly assigned resources.

Nevertheless, one can undoubtedly recognize that the task monitoring systems uptake is inherently complex. Firstly, such systems monitor heterogeneous types of information of different granularity, from low-level system metrics (e.g. network traffic, memory allocation, etc.) to high-level application specific metrics (e.g. throughput, average availability, etc.), which are collected across multiple levels (physical, virtual, application level as depicted in Figure 4) in a Cloud environment at different time intervals.

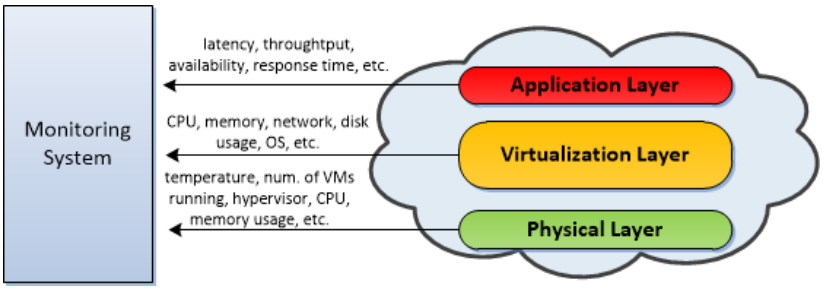

Figure 4: Monitoring Layers

Furthermore, due to the complex nature of such environments, monitoring information recipients are also heterogeneous. For instance, a particular type of information (e.g. CPU usage of a VM) can be accessed - frequently and simultaneously - by: (i) the users that deployed the application in order to manage and configure the deployment to meet their demands, (ii) by any resource provisioner or decision-making module so as to analyze the data and propose appropriate resizing actions for fine-tuning the application performance, and (iii) by the Cloud provider to assert that offered to customers SLAs are met and to perform cost evaluation and billing based on the allocated resources.

A Cloud monitoring system must run without constant human intervention, specifically in a non-intrusive and transparent manner, both to any Cloud application and toany underlying virtualized infrastructure. Finally, it needs to be highly adaptive,

thus diminishing the need for re-contextualization, each time an application and/or resource related parameter changes.

4.2 Application Structure Model

The model used to describe the structure of an application deployed on a Cloud is defined in Deliverable D5.1 [D5.1 - Section 4.1.1]. Based upon this model, the application description provided by the user via the c-Eclipse Application Description Tool [D2.1] is mapped to a meaningful model understood by CELAR internal modules. In brief, a Cloud application can be decomposed into fine-grained sub-parts (component, composite component, relationship). As depicted in Figure 5, a component represents any kind of module or unit encapsulating functionality or data. This view is generic enough to facilitate the modeling of different types of applications and systems (e.g. a web service). A group of components together, with the relationships between them, can be modeled as a composite component, with a Cloud application being composed of one or more complex components. The composite component is not physically represented at runtime; it is only a representational concept helping the internal control of theCloud application.

The provided model motivates us to enrich it with two new concepts, Virtual Machine and Virtual Cluster, representing the virtual infrastructure at runtime. A Virtual Machine concept describes a typical Cloud virtual machine (VM), and each virtual machine can run one or more application component instances. A Virtual Cluster describes locations where Virtual Machine instances can run, such as particular Clouds, availability regions, or VM clusters. Introducing the concept of Virtual Machine and Virtual Cluster, we link the Application Component, which represents a logical organization of an Application, to the underlying virtual infrastructure. This enables the Decision Module to analyze Components and Composite Components, and take adaptation actions at the virtual infrastructure level elements, e.g. by allocating (or de-allocating) different VMs in one Virtual Cluster to another, depending on the application requirements.

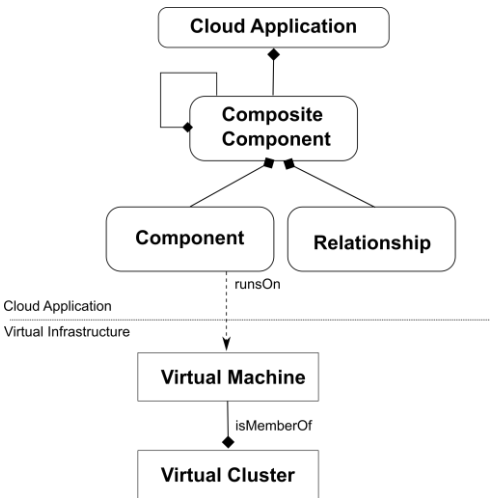

Figure 5: Abstract Application Composition Model

Each component, from the previously defined structure, may have a different architecture, operating system, workload and/or (elastic) requirements. The Cloud Information and Performance Monitor layer supports the above model offering a wide range of functionality and metrics and is adaptable to dynamic changes in the

application itself and in the underlying Cloud environment due to enforcing elasticity actions. At the end, users are only interested in service level metrics such as latency, throughput, response time, availability, accuracy and cost. Service level metrics can be easily expressed by users through directive-based languages, such as SYBL [Copil 2013], and in association with the above application model they can be mapped to low-level metrics (i.e. CPU, network, memory, etc.) gathered by the Monitoring System. The MS monitors application components that reside on multiple VMs and facilitates, in coordination with the Multi-Level Metrics Evaluation Module which has knowledge of the deployed application's topology, any component of the application model by providing a subscription mechanism to aggregate and group monitoring metrics originated by individual VMs.

4.3 CELAR Monitoring System Architecture

Depicted in Figure 6 is an abstract view of the CELAR Monitoring System's architecture. The proposed architecture follows an agent-based *producer-consumer* approach. This approach provides a scalable, real-time, Cloud Monitoring System that is utilized by CELAR to collect monitoring metrics from multiple layers of the underlying infrastructure, as well as performance metrics from deployed applications. During the metrics collection process the CELAR MS takes into consideration the rapid changes that occur due to the enforcement of elasticity actions on the application execution environment and the Cloud infrastructure in general.

Monitoring Agents, presented in Figure 6, are light-weight monitoring instances installed on virtual instances or physical nodes with the task of gathering primitive monitoring metrics from metric collectors, named Probes, residing on the same node.

Probes are independent metric collectors that gather low-level monitoring metrics from the system and performance metrics from the deployed user applications, pushing these metrics to a corresponding Agent upon availability. Instead of employing an Agent polling mechanism for new metrics, a *push* mechanism is utilized allowing thus, each Probe to handle the dissemination of a metric collection differently; by either reporting metrics periodically or upon the occurrence of a specific event.

An Agent distributes processed metrics to **Monitoring Servers** that have expressed interest in receiving metrics from the specific Agent using as a delivery mechanism a slight variation of the traditional *publish and subscribe*[Eugster 2003](pub/sub) messaging paradigm[11]. Employing a pub/sub mechanism minimizes related network communication overhead; since it avoids the need for the MS Server to constantly poll Agents for fresh metrics. A Monitoring Server processes the received monitoring metrics forming *composite metrics*, upon consumer request (subscriptions), performing aggregation and grouping metrics from various instances together.

A hierarchy of **IntermediateMonitoring Servers** that further redistribute monitoring metrics is optional, however when utilized it provides greater scalability to the topology since they offload a central Monitoring Server from continuous information processing. This hierarchical model also provides fault-tolerance by eliminating single points of failure, such as in the case of one central Server, resulting in information resilience, when Servers are reported unavailable, by allowing monitoring metrics to be retrieved by healthy, unaffected intermediate Servers.

Finally, the CELAR Monitoring System provides a web service that allows for external entities such as other CELAR modules or Application Users to access

[11] Explained in Section 5.5

monitoring information stored in MS database. Application Users can use the c-Eclipse Visualization Tool to access the web service and view in a graphical manner monitoring information.

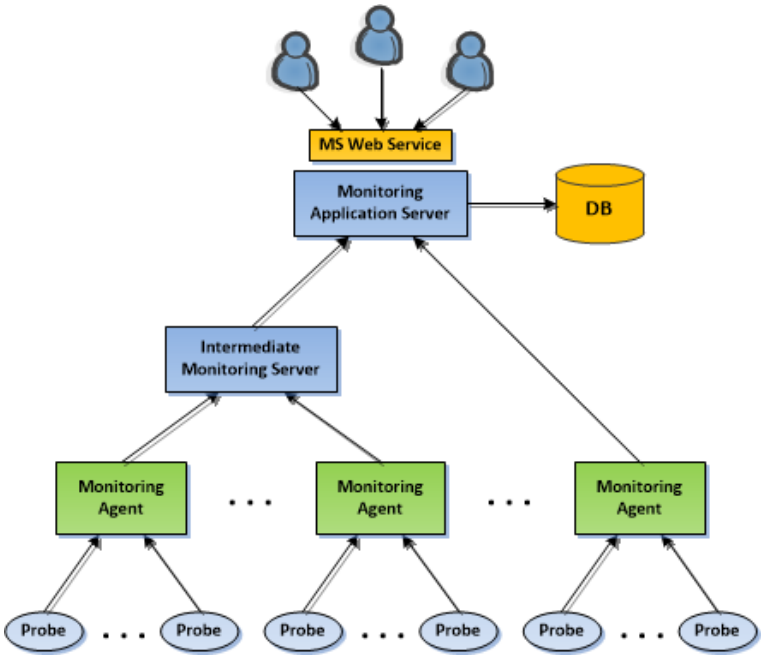

Figure 6: CELAR MS Architecture

4.4 Multi-Level Metrics Evaluation Module

As depicted in the CELAR System architecture (Figure 1), the CELAR Cloud Information and Performance Monitor layer also consists of the Multi-Level Metrics Evaluation Module. The role of the Multi-Level Metrics Evaluation Module is to perform an initial metric aggregation and analysis, aimed at providing structured and enriched monitoring metrics to the Decision Module, in order to take appropriate decisions. Moreover, the Multi-Level Metrics Evaluation Module will provide estimations about the future application cost, quality and resource usage.

4.4.1 Evaluating Cost

The most important metrics thatinfluence the cost of an application running in a Cloud environment that can be retrieved by aMonitoring System without interacting with the Cloud platform are presented inTable 4. Evaluating the cost of an application running in a Cloud environment is challenging due to the diversity and heterogeneity of pricing schemes employed by various Cloud providers (e.g.,*Provider A* may charge per I/O operation, while *Provider B* might charge only per storage size). This heterogeneity generates a gap between the monitoring metrics collected by the MS and the metrics targeted by Cloud billing schemes. For instance, a Cloud Monitoring System can be instructed to report the boot-time for a new VM, though the Cloud pricing schemecalculates *cost/hour* per VM. Another issue that arises involves having measurement units that can be used in cost composition (e.g., converting a "bytes" metric into "GigaBytes (GB)" if the storage cost is specified per GB). To address these issues, the monitoring metrics obtained from the Monitoring System must be converted into metrics understandable by Cloud billing mechanisms.

Metric Name	Description
Running Time	Cost of running a VM per hour or month
Storage Size	Cost of storage size, usually per GB
Operating System	Different cost per different OS licence
I/O Operations	Cost per number of I/O (e.g., per million)
Network Data Transfer Inbound/Outbound	Different prices depending on the data destination (e.g., expensive for external Cloud transfer, free for internal transfer)

Table 4: Cost Metrics

Another important challenge in evaluating the instant running cost of a Cloud application arises when, evaluating the cost of the applicationVM's requires information that cannot be monitored directly (Table 5). For instance, the hourly cost of running a VM in any Cloud might be determined by the VM type (and the resources it offers), and additional services such as monitoring, storage performance optimization, or the bandwidth of the attached network. All these elements cannot usually be monitored directly by a generic monitoring solution, as they are dependent on theCloud provider's billing policy.

Element Name	Description
VM Type	The Virtual Machine flavour, when only predefined flavours are available
Availabilty Zone / Region	Different regions have different pricies for running VMs (e.g., different prices for running in EU or US)
Storage Type	For instance, Amazon has a different price charging policy for different storage types (e.g.S3 storage [Amazon S3] is more expensive than EBS volumes [Amazon EBS])
Storage Backup	Cost of using a backup service
VM Optimization	Cost of having a VM type optimized for memory intensive, CPU intensive or I/O intensive jobs
Monitoring Service	Cost of using a monitoring service, usually billed either per service, or per monitored metric
Load Balancing Services	Cost of using a load balancer
VM Snapshot Storage	Cost of storing images/snapshots of virtual machines

Table 4: Main Cost Elements Dependent on Cloud Provider

Information about the topology and configuration of the running Cloud application must be used when estimating cost. The topology contains information about the type of virtual machines instantiated for its various components, and any additional used Cloud services, as this information is determined first when deploying the application, and then updated when an elasticity action is requested by the Decision Module.

Applying pricing schemes is in turn challenging, as different Cloud providers tend to have different pricing descriptions. Another challenge that must be addressed is the specification of complex pricing rules. For instance, a complex pricing rule could be "*the first 1 TB of transferred data is free, the next 1TB is offered for $5 and from there on each TB is billed for $7*", or "*the cost for adding a storage optimized VM depends on the VM type*".

We address all the above issues, from metric conversion, to specification and evaluation of complex pricing schemes by introducing a generic extensible metric composition language. The composition language is designed to support a large array of metric operations, from basic measurement unit conversion, to aggregation of multiple metrics. The structure of the language is shown in Listing 1 and uses a human readable

Backus-Naur Form (BNF) syntax employing the following set of symbols: '<>' denoting a concept type, ':=' denoting assignment, '|' denoting logical OR and '{}' denoting a repetition of zero or more times of the content inside the brackets. This grammar enables the definition of operations which apply a set of operators over a sequence of one or more operands, similar to mathematical rules. For maximum flexibility, the operands can be monitored metrics, numeric or string values. Supported operators over existing metrics are the four basic mathematical operators (+,-,*,/), and several additional ones, enabling the extraction of the average, maximum or minimum value in a metric sequence, concatenate the values of a sequence of metrics into a single one, or extracting the first or last value from a sequence of values.

```
<rule> := operation "=>" metric
<operation>:= operator "(" operand { "," operand } ")"
<operator>  := "+"|"-"|"*"|"/"|"AVG"|"MAX"|"MIN"|"CONCAT"|"FIRST|"LAST"
<operand>   := metric | number | string
```

Listing 1: Metrics Composition Language

4.4.2 Multi-Level Cost Composition

To support multi-level elasticity control of Cloud applications, the Decision Module requires multi-level cost information aggregated after the application structure (e.g., cost of an individual VM, of an application component, or overall application execution). To address this issue, the Multi-Level Metrics Evaluation Module at first, uses metric composition rules to evaluate the cost of each running VM. Secondly, the VM level information is logically structured after the application structure model (Figure 7).

To obtain the cost of a running VM, metric composition rules are used to combine the values of the monitored metrics with the Cloud pricing scheme. Afterwards, the metric composition language is utilized to define cross-layer cost composition rules. Cross-layer metric composition rules have the role of aggregating monitoring data associated to a particular level from the application structure model (e.g., VM), and associate the aggregated data to the next level (e.g. Virtual Cluster). Using composition rules, we aggregate information associated to a particular level from the application structure model, and obtain higher level information (e.g. cost per Component). Performing this step for each level provides a multi-level cost composition, describing cost for the whole Application, then for each Complex Component, for each Component, down to the cost per individual VM. Such decomposition enables the Decision Module to analyze the application cost in a top-down approach, and take appropriate adaptation actions.

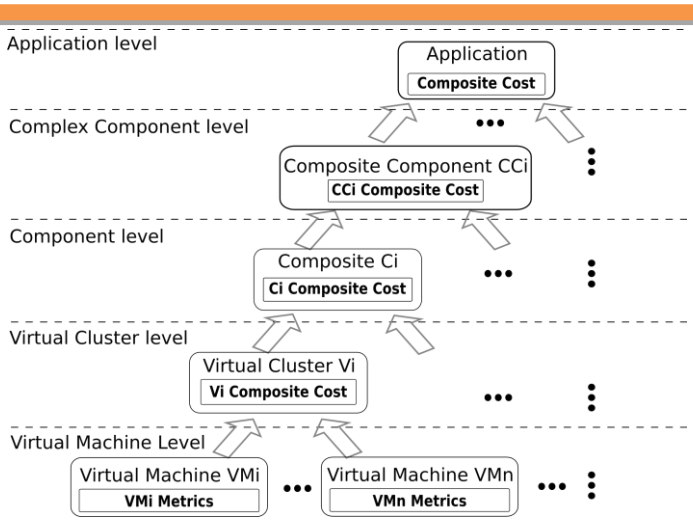

Figure 7: Multi-Level Cost Composition

Defining metric composition rules is a joint responsibility task of the CELAR Application Expert/Developer and CELAR expert[12], as they involve both metrics available for the application VMs, which might be Cloud specific, and metrics collected from the application components. In Listing 2 we describe in XML format a potential metric composition rule, where a "*memory*" metric is obtained for an Application Component, by summing the values of the "*memory*" metric of each virtual instance comprising the Application Component.

```
<CompositionRuleTargetLevel="Component">
<SourceMetric name=" memory" unit="GB/s" level="VirtualMachine"/>
<Operation type="SUM"/>
</CompositionRule>
```

Listing 2: Example of Metric Composition Rule

The same aggregation process can also be applied to structure quality and resource usage information, providing to the Decision Module a complete multi-level view over the application behavior. Providing multi-level cost information enables the Decision Module to take decisions at multiple levels, targeting individual Component instances, Complex Component instances, or the entire Application.

[12]The Application Expert/Developer and CELAR Expert actors are defined in [D1.1 - Section 2.3.1]

5 Implementation of the CELAR Monitoring System

In the previous sections we have described the requirements a Cloud Monitoring System must have and the challenges to overcome when monitoring user applications in a rapidly adapting application execution environment. Furthermore, a short description of the CELAR Monitoring System and the components that comprise it was provided. This section elaborates on implementation specifics and the features available in the current version (v1.0) of the CELAR Monitoring System.

5.1 JCatascopia

Based on the requirements specified and the complex functionality a monitoring system must facilitate when monitoring user applications deployed on a Cloud platform, we have implemented **JCatascopia**. JCatascopia is a fully automated, scalable, real-time Cloud Monitoring System, implemented in Java, which aims at supporting automated multi-grained Cloud platforms which offer elastic resource provisioning for deployed Cloud applications. JCatascopia can be utilized to collect monitoring metrics from multiple layers of the underlying infrastructure as well as performance metrics from deployed applications, and subsequently distribute them to subscribed users and platform operators. JCatascopia provides self-adaptive mechanisms to minimize the intrusiveness of its presence, and enhanced filtering to reduce communication costs for metric distribution. JCatascopia's elastic properties are further enhanced with the ability to dynamically add/remove monitoring instances located on multiple levels of the infrastructure without any human interventionor the need to restart the monitoring system. Furthermore, JCatascopia supports, via a subscription mechanism, aggregation and grouping of low-level metrics to generate high-level metrics at runtime.

The following sections provide a detailed description of the components that comprise the JCatascopia MS:

5.2 JCatascopia Probe

JCatascopia Monitoring Probes[13]are low level metric collectors utilized to gather raw metrics and generate timestamped monitoring events. A probe is a group of correlated metrics that may, but are not required to, share access to the same resources when calculated. For instance, a Memory Probe can report total, free and used memory of a running VM where these values can be retrieved on a Linux OS from different native fields available within the `/proc/meminfo`file.

For the needs of the CELAR System, Probes are utilized to collect metrics concerning resource allocation and usage from multiple levels of the Cloud infrastructure, such as at VM, Virtual Cluster and Cloud level. Probes are also used to collect application performance metrics from the application level.Probes can easily be implemented using the *JCatascopia Probe Java API* that is provided by JCatascopia. The MS can be extended to support any kind of metrics by providing the application developer the ability to write monitoring Probes for collecting any metric that may be of his interest and deploy them on running instances of the Monitoring System.

Each reported metric transmitted from a Probe to the corresponding Agent, consists of the *metric name*, *units*, *data type* (integer, double, etc.) and the *collected value*. In order for a new metric to be created by a Probe Developer, except of the metric name, units and data type a *shortdescription* and the *minimum collectingperiod* in seconds must be

[13]For simplicity JCatascopia Monitoring Probes from now on will be referred to as Probes

specified. We adopt the naming *Probe Property* to refer to a single metric and its metadata (*id, name*, *units*, *data type* and *description*) and use the naming *Probe Metric* to refer to the container (object) which contains the latest property values collected and their timestamp.

The term *intelligent* Probe, best characterizes a JCatascopia Probe, since Probes are not just scripts (e.g. Munin [Munin]) or checks (e.g. Nagios [Nagios]) which simply report values. For instance, Probes: (i) enable the storage and querying for the last metrics reported and their timestamp, (ii) perform metric validity checks (e.g. is the retrieved value of the data type specified by the Probe Developer?) and also allow Probe Developers to add their own custom filter checks (e.g. for a Probe collecting click statistics from a web server log file, don't report clicks from a specific IP), (iii) allow parameter configuration and most importantly, (iv) provide self-adaptive elastic properties such as filtering and sampling.

For a Cloud MS that monitors an elastic application environment, *self-adaptive properties* are essential features to minimize the runtime impact and consequently the cost that the MS imposes on Cloud user VMs. *Adaptive filtering* assists in reducing communication and storage overhead by not transmitting or storing continuous values of a metric with very small variances between them. *Adaptive sampling* provides the ability to automatically adjust the sampling frequency, in a similar manner, minimizing the computational overhead of constantly collecting values of a metric in short time intervals when the values are considered stable (additionally sample more frequently when values have large variances and will be of great interest to the user and Decision Module).

Probes are implemented in Java as threads and run independently from each other. If a Probe encounters a problem such as unexpected termination, the metric collection process of the other Probes' is not affected. Probes are implemented separately from JCatascopia Monitoring Agentsallowing them to be dynamically added (or removed) from an Agent at runtime, without again interfering with the monitoring process. This feature takes advantage of Java's dynamic class loader feature allowing a new Probe to be added to an Agent or to be re-configured without the need to restart the whole Monitoring System or parts of it. Furthermore, this pluggable feature grants Agents great flexibility, which is of extreme importance when monitoring applications, by allowing the number and type of Probes corresponding to an Agent to vary. An example illustrating the importance of allowing Agents to manage various Probes and not pre-defined ones, indicating why different VMs and application components require different monitoring metrics is the following: An Application User, when monitoring a VM considered as a web server, would be interested in metrics such as CPU and bandwidth utilization, uptime, average load (requests received per second), etc. whereas when monitoring a VM considered as a database node, the same user would be interested in CPU and memory utilization, disk I/O stats, throughput (count of database operations per minute such as inserts, updates, reads, writes), response time, etc.

Probes adopt a push delivery mechanism to report collected metrics, upon availability, to the corresponding Agent. This feature allows each Probe to report metrics at different time granularities and also removes the need for the local Agent to constantly poll its associated Probes for new metrics minimizing computational effort. Allowing for metrics to be gathered at different time granularities is essential to satisfy a vast number of heterogeneous types of metrics gathered from multiple levels, since low-level metrics such as CPU, memory, disk I/O, etc. are usually required to be collected in

shorter intervals (e.g. in the range of a few seconds) than high-level service metrics such as throughput, latency, availability, etc. which are usually collected in larger intervals (e.g. in the range of 100 of seconds). Optionally, an Agent can also pull metrics from a Probe upon user request, if *metricpulling* is enabled, for the interested Probe.

5.2.1 JCatascopia Probe Java API

Table 6 presents the JCatascopia Probe Java API which consists of API calls that can be utilized by Probe Developers to create and deploy their own custom Probes and can also be used by Agents to reconfigure Probes based upon remote user requests. The main features and functionality provided by the Probe API in v1.0 of JCatascopia are the following:

- *Active*/*Deactivate* Probe collecting mechanism. Also *terminate* Probe instance.
- Query for Probe metadata such as Probe *id*, *name*, *minimum collecting period*, *description*, *status*, and etc.
- *Add*/*RemoveProperties* to be monitored.
- Query for *Properties* being monitored, their *metadata*, *last reported values* and *timestamp*.
- Perform *validity checks*, checking if reported values are of the data type specified by the Probe Developer
- Turn ON/OFF *filtering*. Probe Developers can create filters by extending the JCatascopia `Filter` Class. In v1.0 of JCatascopia, a `SimpleFilter`is available, where by providing a range for a specific Probe Property, a window $[lastValue - range,\ lastValue + range]$ is determined. Values residing in the windoware discarded.
- Turn ON/OFF *adaptive sampling*. In v1.0 of JCatascopia, basic adaptive sampling is supported but it is in a primitive development phase allowing only for the sampling period to be incremented when a *stable* phase is detected but when values vary, the sampling period is set again to the minimum collecting period initially determined by the Probe Developer.

Adaptive sampling and filtering will be further pursued and enhanced in v2.0, with more complex rules to detect, in the case of adaptive sampling, stable phases and to automatically adjust the window range, in the case of adaptive filtering.

API Call	Arguments	Return Type	Description
getProbeID()	-	String	Returns Probe ID
getProbeName()	-	String	Returns Probe Name
getCollectPeriod()	-	int	Returns current Probe collection period in Seconds
setCollectPeriod()	int period	-	Set Probe collection period (in Seconds)
getProbeStatus()	-	ProbeStatus	Returns Probe current status {INACTIVE, ACTIVE, DYING}
writeToProbeLog()	Level level, String msg	-	Writes message and level of severity to log file
addProbeProperty()	int id, String name, ProbePropertyType type, String units, String desc	-	Adds a property/metric to be monitored by the Probe
removeProbeProperty()	int id	-	Remove property from being monitored
getProbeProperties()	-	Map<Integer, ProbeProperty>	Returns a map of all the properties/metrics collected by the

			Probe
getProbePropertiesAsList()	-	List<ProbeProperty>	Returns a list of all the properties collected by the Probe
getProbePropertyByID()	intpropID	ProbeProperty	Returns a Property by ID
activate()	-	-	Activate Probe collecting mechanism
deactivate()	-	-	Deactivate Probe collecting mechanism
terminate()	-	-	Shutdown Probe
getProbeMetadata()	-	Map<String, String>	Returns map containing Probe metadata
getLastMetric()	-	ProbeMetric	Returns last collected metrics
metricToJSON()	-	String	Returns last metrics collected by Probe in JSON format
getLastUpdateTime()	-	Long	Returns last metric timestamp
checkReceivedMetric()	ProbeMetric metric	-	Validates collected metrics to check if data types match. **Can be overridden** to perform custom checks
metricsPullable()	-	Boolean	Returns true if metrics can be pulled
setPullableFlag()	boolean flag	-	Sets *pullable* flag to value provided
pull()	-	-	Collects new metric values (if pullable flag is set to true) and adds them to metric queue
getDescription()	-	String	Overridden by Probe Developer to return Probe Description. **Must be provided**
collect()	-	ProbeMetric	Overridden by Probe Developer to collect defined metrics. **Must be provided**
cleanup()	-	-	**Optionally overridden** by Probe Developer to clean up loose ends before Probe terminates
turnOnAdaptiveSampling()	int min, int max	-	Turns on adaptive sampling. Probe developer specifies min and max collecting period
turnOffAdaptiveSampling()	intpropertyID	-	Turns off adaptive sampling
turnFilteringON()	intpropertyID, Filter f	-	Turns on filtering for a specific property. Values in the window of the provided filter are discarded
turnFilteringOFF()	-	-	Turns off filtering

Table 6: JCatascopia Probe Java API

5.2.2 Exemplary Probe

Probes are created using the JCatascopia Probe Java API by inheriting the Probe Class which provides the necessary abstractions for hiding the complexity of all the Probe functionality from the Developer, requiring for him to write as less code as possible. The Probe Developer is only required to override the `collect()` method, by providing the essential functionality for collecting the metrics specified, and the `getDescription()` method. Optionally, the Probe Developer may override the `cleanUp()` method, to close loose ends (e.g. close open file descriptors) just before terminating the Probe, or the `checkReceivedMetric()` method to perform custom metric checks each time a new metric is collected.

Figure 8 depicts an exemplary Memory Probe[14] that can collect Memory metrics from any Linux flavor VM. The default *minimum collecting period* is set to 30 seconds.

[14]For simplicity and to save space, the `getDescription()` method and exception handling is omitted in Figure 8

Via the API call `addProbeProperty()` we have selected to collect the following three metrics: (i) *memTotal* which is the total memory allocated by the VM, (ii) *memFree* which is the memory space currently not allocated and (iii) *memUsedPercent* which is the percentage of memory currently in use. The first two properties have an *integer* data type and report values in KiloBytes. The third property has a *double* data type and reports values as a percentage of the total memory allocated by the VM. Via the API call `turnFilteringOn()` we set a `SimpleFilter` on the *memUsedPercent* property to filter out values between ±0.5% of the previously reported value. We can also enable metric pulling for this Probe via the API call `setPullableFlag(true)`. Finally the Probe Developer by overriding the `collect()` method, specifies how these metrics will be collected. When all property values are collected, the `collect()` method must return a *ProbeMetric* object which, as stated previously, contains a mapping for each *Probe Property* to its recently collected value.

```
public class MemoryProbe extends Probe{
    private static final String PATH = "/proc/meminfo";

    public MemoryProbe(String name, int freq){
        super(name,freq);
        this.addProbeProperty(0,"memTotal",ProbePropertyType.INTEGER,"KB","Total System Memory");
        this.addProbeProperty(1,"memFree",ProbePropertyType.INTEGER,"KB","Memory Free");
        this.addProbeProperty(2, "memUsedPercent", ProbePropertyType.DOUBLE, "%", "Percentage of Memory in use");
        this.turnFilteringOn(2,SimpleFilter(0.5));
        this.setPullableFlag(true);
    }

    public MemoryProbe(){
        this("MemoryProbe",30);
    }

    public ProbeMetric collect(){
        HashMap<Integer,Object> values = new HashMap<Integer,Object>();
        BufferedReader br = new BufferedReader(new FileReader(new File(PATH)));
        String line;
        while((line = br.readLine())!=null){
            if (line.startsWith("MemTotal"))
                memTotal = Integer.parseInt((line.split("\\W+")[1]));
            else if (line.startsWith("MemFree"))
                memFree = Integer.parseInt((line.split("\\W+")[1]));
            else if (line.startsWith("Cached"))
                memCached = Integer.parseInt((line.split("\\W+")[1]));
        }
        memUsed = memTotal - memFree - memCached;
        memUsedPercent = (1.0 * memUsed) / memTotal;
        values.put(0, memTotal);
        values.put(1, memFree);
        values.put(2, memUsedPercent);
        return new ProbeMetric(values);
    }
}
```

Figure 8: Exemplary Memory Probe

After the `collect()` method returns a *ProbeMetric* object, in the background, the Probe thread parses the obtained values and any specified validity checks (if any are set by the Probe Developer) are performed. Furthermore, the values of the *Properties* that have filters enabled are checked and if the values are in the filter window range they are discarded. Finally the received Probe metric values are ready to be pushed to the corresponding Monitoring Agent. Probes add metrics to the Agents metric queue in JSON format, as depicted in Figure 9, where each message contains the Probe *id* and Probe *name* (*group*), the *timestamp* indicating when the metrics where collected and an array where each element corresponds to one *Probe Property* described by its *name*, *units*, *type* and the collected metric *value*.

```
{
  "probeID":"e42aa5783b114423a71e876e5dae10ff",
  "timestamp":"1376510743316",
  "group":"MemoryProbe",
  "metrics":[{"name":"memTotal","units":"KB","type":"INTEGER","val":"2064772"},
             {"name":"memFree","units":"KB","type":"INTEGER","val":"918192"},
             {"name":"memUsedPercent","units":"%","type":"DOUBLE","val":"29.761"}]
}
```


Figure 9:Example Metric Message of a Memory Probe

5.3 JCatascopia XProbe

The first version (v1.0) of JCatascopia offers a useful tool, named XProbe which allows for developers to instrument their applications to report metrics either periodically or upon availability to the locally deployed JCatascopia Monitoring Agent. XProbe is deployed as a system service running as a background process listening for incoming requests which report collected metric values. Upon reception of a fresh metric and its respective value, XProbe forwards the metric to the local monitoring Agent residing on the VM by pinging the *Probe Controller* of the Agent at the local port specified in the Agent configuration file (*control_port*).

This feature is useful when running experiments, benchmarking or debugging an application by removing the need for the Application User to develop custom Probes for such purposes. For example when benchmarking a distributed database system deployed on a Cloud infrastructure by producing dummy workloads using Yahoo! YCSB [YCSB] clients, these clients can be instrumented to report to JCatascopia XProbe throughput and latency metrics when made available. When forwarding metrics to the monitoring Agent via XProbe the metric *name*, *type*, *units* and the collected *value* must be provided. Optionally a metric *group* can also be provided. Figure 10 depicts the usage and an example of reporting a metric via the command line to XProbe.

```
user@ubuntuVM:~$ XProbe --help
XProbe v1.0 part of JCatascopia Monitoring Framework

Usage: XProbe <Arguments>

 Arguments

  -h, --help                       Print help menu
  -n, --name:METRIC_NAME           Name of the metric
  -u, --units:METRIC_UNITS         Units of the metric
  -t, --type:METRIC_TYPE           INTEGER|DOUBLE|STRING|CHAR|LONG|BOOLEAN
  -v, --value:VALUE                Value to report
  -g, --group:METRIC_GROUP         Optionally provide the group the metric will belong to
  -c, --config:PATH_TO_CONFIG_FILE Optionally specify path to JCatascopia Agent config file if it is not
                                   located at its default location

Example: XProbe --name:cassandra_throughput --type:DOUBLE --units:ops/s --value:6450.77 --group:Cassandra
```

Figure 10: XProbe Usage and Example

5.4 Metrics

JCatascopia takes into consideration the nature of monitoring metrics which are dynamic and vary in time. Not all metrics need to be collected at the same time; thus, each Probe has its own collecting period which can be automatically adjusted by turning on adaptive sampling. Furthermore, not all metrics need to be collected and made available continuously, but may only be collected when certain events are triggered. This situation usually arises at the application level where a process may run only a few times a day; for example monitoring resources allocated by a web stat analyzer that processes web event logs of a server only at the end of each day.

5.4.1 List of available metrics

JCatascopia can collect monitoring metrics from the Cloud platform and application level. In Table 7 we present the Probes currently offered by v1.0 of JCatascopia that can gather metrics from any Linux OS distribution (i.e. Ubuntu, CentOS, Debian, etc.) that resides on a VM or physical node. It is important to note that for the scope of the CELAR project, metrics from the physical infrastructure will be taken into consideration for the decision-making process only if they are exposed by the Cloud provider since this type of information is considered proprietary and many Cloud providers are not willing to expose it. For this reason even though the implemented Probes could be deployed on a Cloud providers' infrastructure we will utilize Probes that gather any information exposed at the physical or hypervisor level through the Cloud providers' API.

Later, in v2.0 of JCatascopia, Flexiant [Flexiant], as one of the consortium IaaS Providers, will expose such metrics through their API [FCO], allowing monitoring Probes to access such information and distribute it to the Decision Module.

Probe	Metric	Description	Unit
CPU	cpuTotal	Percentage of total CPU utilization	%
CPU	cpuUser	Percentage of CPU utilization that occurred while executing at the user level	%
CPU	cpuSystem	Percentage of CPU utilization that occurred while executing at the system level	%
CPU	cpuIdle	Percentage of time that the CPUs were idle and the system did NOT have an outstanding disk IO request	%
CPU	cpuIOwait	Percentage of time that the CPUs were idle during which the system had an outstanding disk IO request	%
Memory	memTotal	Total amount of memory	KB
Memory	memFree	Amount of available memory	KB
Memory	memCache	Amount of cached memory	KB
Memory	memSwapTotal	Total amount of swap space	KB
Memory	memSwapFree	Amount of available swap memory	KB
Memory	memUsed	Current memory utilization in KB	KB
Memory	memUsedPercent	Current memory utilization in percentage	%
Network	netPacketsIn	Packets in per second	pckt/s
Network	netPacketsOut	Packets out per second	pckt/s
Network	netBytesIn	Number of bytes in per second	B/s
Network	netBytesOut	Number of bytes out per second	B/s
Disk	diskTotal	Disk total capacity	MB
Disk	diskFree	Available disk space, aggregated over all partitions	MB
Disk	diskUsed	Current disk space used	%
DiskIO	readkbps	KB read to disk per second	KB/s
DiskIO	writekbps	KB written to disk per second	KB/s
DiskIO	iotime	Percent of disk time spent on I/O operations	%
System	os	Operating System name and release	String
System	arch	System architecture	String
System	btime	Boot time	Time
System	cpuNum	Number of CPUs	Num

Table 7: JCatascopia Metrics Currently Available at VM Level

The CELAR MS will also be enhanced with application probes in order to gather application performance metrics for widely used applications such as Apache HTTP Server [Apache Server] and Cassandra DB [Cassandra] and any other metrics needed by

the developers of the pilot applications presented in D1.1. The first version of JCatascopia currently extracts, at application level, throughput and latency metrics from Yahoo! YCSB [YCSB] clients.

5.5 JCatascopia Monitoring Agent

JCatascopia Monitoring Agents[15] reside on virtual instances or any Cloud entities (located at physical, virtual, platform or application level) intended to be monitored. Agents are the entities responsible for collecting, processing and distributing monitoring information, originating from the Probes, to JCatascopia Monitoring Server(s)[16]. An Agent can be considered as the Probe Manager for a particular VM, since it is responsible for activating/deactivating Probes, configuring accordingly their parameters and upon user request pull different metrics. By allowing monitoring Probes to be dynamically deployed at runtime, the MS becomes more flexible and extensible.

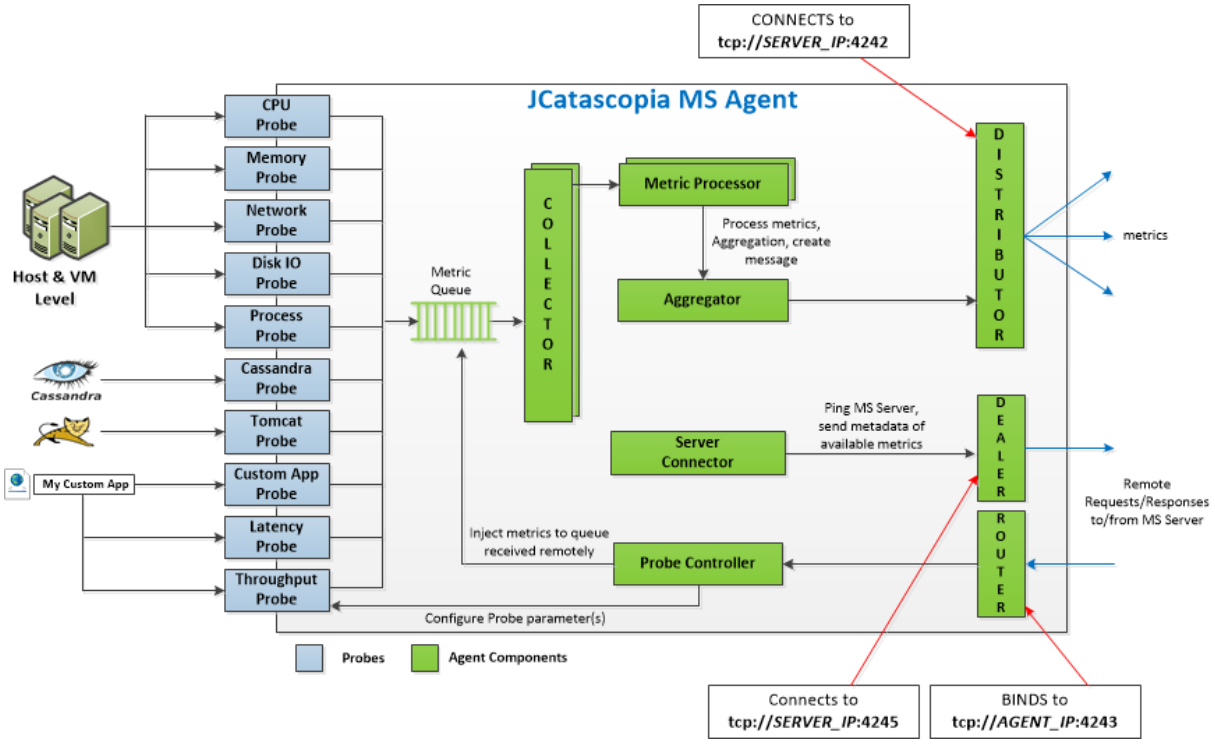

Figure 11: JCatascopia MS Agent

Figure 11 depicts the internal architecture of an Agent and its components. Initially, when a new Agent is deployed, the *Server Connector* component pings the MS Server to see if it is listening and responding to requests. If the MS Server responds, then the Server Connector transmits Agent metadata to the MS Server, such as the Agents' ID and IP address and metadata of the metrics it will be collecting.

Once the initial connect phase is over, monitoring information can now be gathered by Probes. Probes add newly collected metrics to the Agent *Metric Queue* in order to be processed. Metrics can also be added to the queue by the *Probe Controller* component that listens to requests, allowing in this manner for metrics to be added to the Agents metric queue either by another process or remotely. The Probe Controller also listens

[15]For simplicity JCatascopia Monitoring Agents from now on will be referred to as Agents
[16]For simplicity JCatascopia Monitoring Servers from now on will be referred to as Servers

for probe parameter configuration requests (e.g. configure probe collecting frequency) originating by either the MS Server or from Application Users via the MS Visualization Tool.

Metrics are de-queued by the Metric Collectors and processed by *Metric Processors* in parallel. Processing a metric refers to the task of converting a metric to a human readable format in a semi-structured manner and adding Agent metadata; thus, preparing it for distribution. The number of Collectors and Processors can be defined in a simple Agent *configuration file*[17] and depends mainly on the number of metrics and how often they are collected.

After processing the gathered metrics, these are given to the Aggregator who is responsible to withhold them from distribution until an aggregation policy is satisfied. Aggregation is an important feature of JCatascopia, aiming at minimizing the communication overhead as a result of constantly transmitting messages over the network. The aggregation policy can be configured through the aforementioned Agent configuration file and can be *time-based* (e.g. aggregate and distribute metrics every 30sec); *message size-based* (e.g. aggregate and distribute metrics if message size exceeds 2KB) or can take into consideration both (preferred case). When aggregated metrics are ready to be sent, a message is created containing all the ready metrics and is subsequently sent to the MS Server. It should be noted that an Agent can send metrics to more than one MS Servers (that act as intermediates). The distribution of metrics utilizes a push delivery mechanism to send metrics to the number of interested MS Servers, which are defined in the Agent configuration file.

5.5.1 JCatascopia Agent Message Patterns

A Monitoring Agent uses two different message patterns for communication. Briefly, the *Dealer-Router* pattern consists of a Dealer entity, which can be considered as a client that sends a request to a Router entity, which can be considered as a server, replying back with the appropriate information. An Agent provides both entities. It utilizes a Dealer entity to send remote requests to the MS Server (e.g. ping MS Server stating its existence, send metadata concerning the offered metrics, etc.) and it utilizes a Router entity to receive requests to configure a Probe (e.g. de-activate Probe, etc.) or to add metric(s) to the metric queue from another process (e.g. from XProbe).

The other message pattern that is used is a variation of the *Publish and Subscribe*pattern for the previously mentioned metric push delivery mechanism. In this pattern, initially, entities (subscribers) express interest and subscribe to an event stream, of another entity named the publisher. When events are produced, the publisher distributes them to the subscribers, if any exist, eliminating the need of the subscriber to constantly poll the publisher to check if new events are available providing greater scalability and minimizing network bandwidth consumption.

In our Cloud Monitoring paradigm, the static part of the model is the MS Server, opposed to the MS Agents which appear (and disappear) dynamically due to elasticity actions. For this reason, in contrast to the classic publish-subscribe pattern, we vary the message pattern to allow Agents, which are considered metric publishers, to initiate the subscription process by ping-ing the Server of their existence. With this variation the Server is agnostic to the network location of its Agents, eliminating the need for a directory service that contains these locations (which would be required in the classic pub/sub pattern) and allowing Agents to come and go dynamically in a flexible manner.

[17] The JCatascopia Agent configuration file is located in the JCatascopia installation folder.

5.6 JCatascopia Monitoring Server

MS Servers are the entities responsible for managing MS Agents. The basic functionality of a Server consists of receivingmonitoring metrics from Agents, processing and storing them to the *Monitoring Database* and further distributing the acquired metrics to any interested entities.

Depicted in Figure 12 is the internal architecture of a Server and its components. The*Control Listener* and *Agent Listener*components are the entities that listen for incoming Agent registration requests and newly collected metrics, respectively. When a newly deployed Agent pings the Control component (which is considered a *Router* entity) of the Server to register as a new metric stream, metadata describing the Agent and the available metrics it will collect are stored in suitable data structures (*Agent* and *Metric Map*).

After registering, an Agent distributes (publishes) metric messages to the Agent Listener component, which listens for incoming messages and enqueues them in the processing queue. Messages are dequeued from the processing queue and processed by Metric Processors in Parallel. The number of *Processors* can be defined in a simple *Server configuration file* and depends mainly on the number of messages and how often they are received. Processing messages refers to the task of parsing the message, decomposing it to grab the metrics consisting a message and updating the metric data structure. The metric data structure stores metadata and the latest value of the metrics an Agent reports.

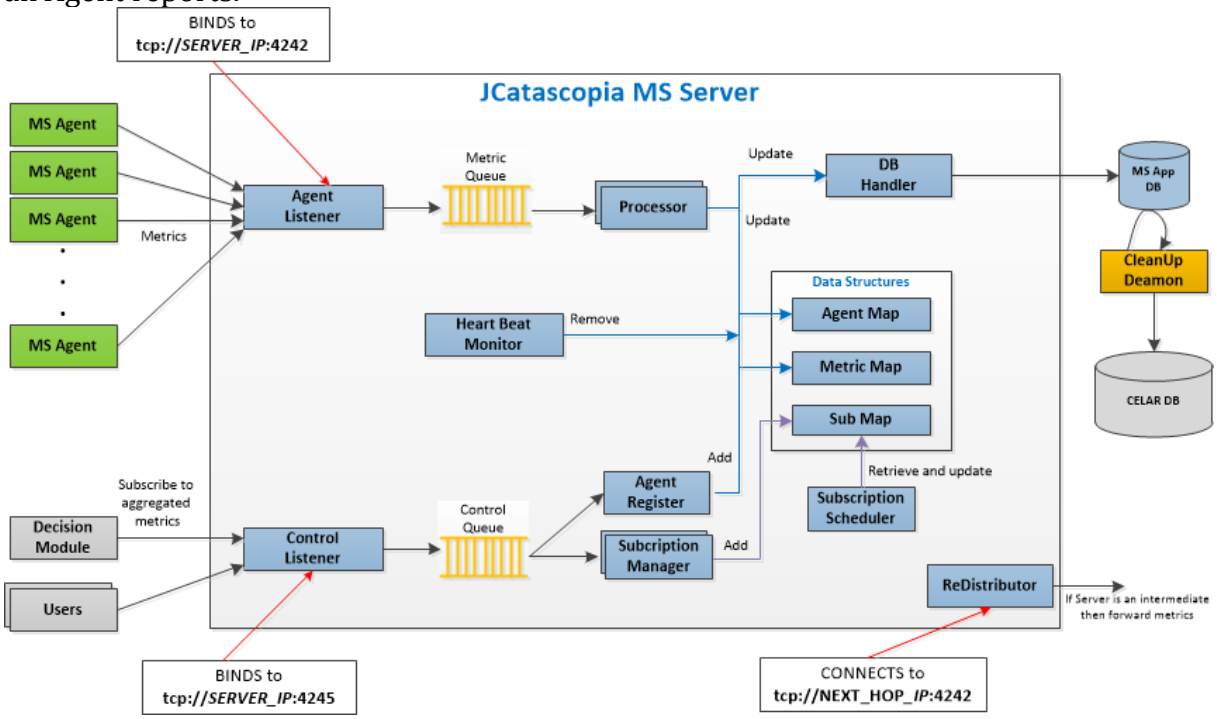

Figure 12: JCatascopia MS Server

Heartbeat monitoring[Hayashibara 2002] is used to automatically detect without the need of any external or human intervention when a monitoring Agent ceases to exist due to scaling down elasticity actions or when an Agent is considered down due to temporary network connectivity issues. The *Heartbeat Monitor* component checks periodically (time interval is defined in the Server configuration file), and reports the status (Figure 13) of the registered Agents, setting their status to DOWN when finished. If an Agent fails to contact the Server until the next interval by either sending fresh

metrics or a heartbeat (if no new metrics are collected) the status of the Agent will remain DOWN. If this occurs consecutively for three times then the Agent is considered DEAD and is removed from the Server.

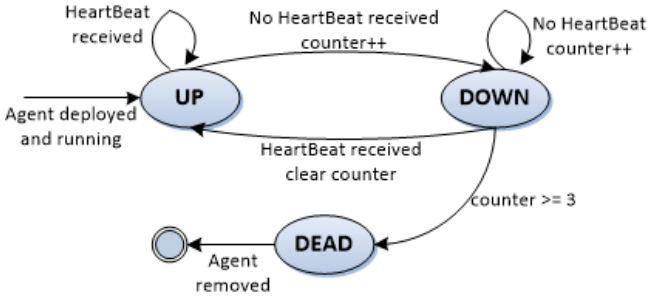

Figure 13: Agent State Diagram

If the Server acts as an intermediate metric redistributor then after processing a metric it is then ready to be distributed to the according higher-level interested Servers. If the Server is not an intermediate or if the *store in database* option is enabled then after processing a metric it is stored by the Database Handler to the database.

5.7 JCatascopia Subscription Mechanism

The Subscription Mechanism is built into the Server to allow Application Users and internal CELAR modules (e.g. Decision Module) via a RESTful API to apply aggregation filters and grouping functions upon low-level monitoring metrics gathered by individual monitoring Agents to create new high-level metrics.

The Subscription Manager, depicted in Figure 12, is the component that retrieves from the control queue and processes subscription requests, adding a new entry to the Subscription data structure and notifying the Subscription Scheduler. The Subscription Scheduler is the component that retrieves the Subscriptions from the respective data structure and updates their current value based on the minimum updating period specified by the user in the Subscription request.

A subscription rule can be considered as a *triplet* with the following main elements:

{*Filter, Members, Action*}

A *Filter* consists of the function(s) to be applied to the low-level metric(s) collected from Agents specified by their id in the *Members* list of the rule, mapping the low-level metrics to a newly created high-level metric. Figure 14 depicts the subscription rule language in BNF.

```
<SubscriptionRule> ::= <Filter>, <Members>, <Action>

<Filter> ::= <MetricName> = <Expression> | <GroupFunction>(<Expression>)
<Expression> ::= <Operand> | <Operand> <Op> <Expression>
<Operand> ::= <Number> | <MetricName> | (<Expression>)
<Op> ::= +|-|*|/
<MetricName> ::= <String>
<GroupFunction> ::= AVG|SUM|MIN|MAX

<Members> ::= MEMBERS = ({<AgentID>,} <AgentID>)
<AgentID> ::= <String>

<Action> ::= ACTION = NOTIFY(<Act>) | PERIOD(<Number>)
<Act> ::= ALL | {<Relation> <Number>,} <Relation> <Number>
<Relation> ::= <|>|=|!=|>=|<=
```

Figure 14: Subscription Rule in BNF

An exemplary *Filter* to calculate (and consequently create a new high-level metric) the average throughput from low-level metric such as read, write, insert and update operations per second of a distributed database ring/cluster is the following:

```
DBthroughput = AVG(readps+writeps+insertps+updateps)
```

When a filter is matched then the *Action* specified in the rule is enforced. JCatascopia allows users to set either *time-based* (notified periodically) or *event-based* actions (notified when event threshold is violated). An example of a time-based action, where the subscriber is notified periodically every 25 seconds is the following:

```
ACTION = PERIOD(25)
```

An example of an event-based action where the subscriber requests to be notified only if the newly created metric reports values lower that 25% or higher than 75% is the following:

```
ACTION = NOTIFY(<25,>=75)
```

Finally, a complete, yet simple, example of a subscription rule for an Application User wanting to monitor the average CPU usage of a web server cluster consisted by N individual web servers with JCatascopia Monitoring Agents deployed on them would be the following:

```
cpuTotalUsage = AVG(1 - cpuIdle)
MEMBERS = [id1,…,idN]
ACTION = NOTIFY(>=82%)
```

Listing 3: Subscription Rule Example

where, `1 - cpuIdle` is the current CPU usage of each individual member of the cluster specified in the rule and AVG is the grouping function applied to calculate the average CPU usage of the cluster.

It should be noted that the subscription mechanism does not have any rules built-in internally nor does it have the intelligence to create rules at runtime. The subscription mechanism is just the *rule applier*. Specifying rules and registering actions to be enforced is the task of Application Users and internal CELAR modules.

5.8 Message Distribution

JCatascopia embraces the ZMQ framework [ZMQ] to perform message distribution between Agents and Servers in order to register Agents to their corresponding Servers as metric streams and from there on to publish metrics to the stream.

ZMQ is a high-performance asynchronous messaging library aimed to be used in scalable distributed or concurrent applications located in large network infrastructures such as Clouds. ZMQ looks like an embeddable networking library but acts as a concurrency framework. The framework sockets provide an abstraction of asynchronous message queues and multiple messaging patterns but unlike message-oriented middleware, a ZMQ utilized system can run without a dedicated message broker, thus gaining in terms of efficiency and scalability. That is why ZMQ sockets where selected as the message distributors between the different components that comprise the JCatascopia Monitoring System. ZMQ core, libzmq, is implemented in C++ with bindings available for over 40 programming languages including C, python, Java, PHP, etc. The message distribution of JCatascopia uses JeroMQ [JeroMQ] which is a complete rewrite of the ZMQ core (libzmq v.3.2.2) in Java, instead of using the classic Java binding in coordination with libzmq (C++), thus gaining portability since JCatascopia Agents and Servers remain platform and machine architecture independent.

JCatascopia builds on the simplistic, yet powerful, ZMQ socket types (*Dealer*, *Router*, *Publisher* and *Subscriber* sockets) to create JCatascopia application level entities, to be utilized in the two implemented message distribution patterns (*Dealer-Router*, *Publisher-Subscriber*), as described in section 5.5. Thus JCatascopia is able to control the message flow between Agents and Servers, adapting, if needed, to network transmission failures by rescheduling and resending messages.

Depicted in Figure 15 is an exemplary aggregated message, in JSON format, sent from an Agent to the Monitoring Server containing metrics received by four Probes (*Memory, DiskStats, CPU*and *Network Probe*). The message contains in an array the metrics received from the Probes, in addition with Agent metadata, such as the Agent *id* and *IP address*. The aggregation policy rules specified, for this example, in the Agent configuration file are (i) *time-based* allowing for metrics to be aggregated for a time interval (`aggregator_interval=30`sec) and (ii) *message size-based* allowing metrics to be aggregated until the total message size exceeds the specified quota (`aggregator_buffer_size=2048` Bytes). Metric distribution is enabled when one of the previous policy rules are satisfied.


```
{
  "events":[{ "timestamp":"1376511531931",
              "group":"MemoryProbe",
              "metrics":[{"name":"memTotal","units":"KB","type":"INTEGER","val":"2064772"},
                         {"name":"memFree","units":"KB","type":"INTEGER","val":"702312"},
                         {"name":"memCache","units":"KB","type":"INTEGER","val":"655504"},
                         {"name":"memUsed","units":"KB","type":"INTEGER","val":"706956"},
                         {"name":"memSwapTotal","units":"KB","type":"INTEGER","val":"1046524"},
                         {"name":"memSwapFree","units":"KB","type":"INTEGER","val":"1046524"},
                         {"name":"memUsedPercent","units":"%","type":"DOUBLE","val":"34.23"}]},
            { "timestamp":"1376511531936",
              "group":"DiskStatsProbe",
              "metrics":[{"name":"readkbps","units":"KB/s","type":"DOUBLE","val":"0.0"},
                         {"name":"writekbps","units":"KB/s","type":"DOUBLE","val":"0.0"},
                         {"name":"iotime","units":"%","type":"DOUBLE","val":"0.0"}]},
            { "timestamp":"1376511531937",
              "group":"CPUProbe",
              "metrics":[{"name":"cpuTotal","units":"%","type":"DOUBLE","val":"50.0"},
                         {"name":"cpuUser","units":"%","type":"DOUBLE","val":"0.0"},
                         {"name":"cpuSystem","units":"%","type":"DOUBLE","val":"50.0"},
                         {"name":"cpuIdle","units":"%","type":"DOUBLE","val":"50.0"},
                         {"name":"cpuIOwait","units":"%","type":"DOUBLE","val":"0.0"}]},
            { "timestamp":"1376511531942",
              "group":"NetworkProbe",
              "metrics":[{"name":"netBytesIN","units":"bytes/s","type":"DOUBLE","val":"123.2"},
                         {"name":"netPacketsIN","units":"packets/s","type":"DOUBLE","val":"11.2"},
                         {"name":"netBytesOUT","units":"bytes/s","type":"DOUBLE","val":"0.0"},
                         {"name":"netPacketsOut","units":"packets/s","type":"DOUBLE","val":"0.0"}]],
  "agentID":"e42aa5783b114423a71e876e5dae10ff",
  "agentIP":"192.168.17.128"
}
```


Figure 15: Example of Aggregated Message Sent from Agent to Server

5.9 JCatascopia MS Database

The MS Server stores in its data structures only the current state of the system. The current state refers to metadata concerning the Agents currently distributing metrics to the Server and the latest value of these metrics. Consequently, there is a need for a database to store previous states of the MS and metrics in order to be later retrieved. The MS Server after processing freshly received metrics from Agents deployed on application VMs, stores these metrics locally for each application to an Application Monitoring Repository, in order to be consumed mainly by the Application User(s) (using the MS Visualization Tool of c-Eclipse) and the Decision Module via pull requests.

This allows the Decision Module to quickly retrieve requested metrics when needed, without flooding it constantly with new metrics. In addition, it reduces query latency to a minimum by not having to constantly query the central CELAR DataBase for metrics.

For at least the first version of the CELAR MS, the local Monitoring Repository is a MySQL [MySQL] relational database. The selection of a relation database for the first version was based upon providing the Decision Module with the ability to perform various types of complicated queries by allowing multiple joins on tables which cannot be facilitated by a NoSQL database. Even though a relational database provides enhanced query capabilities compared to non-relational databases it is not a suitable storage container as the size of tables grows (e.g. metric table) very quickly, increasing therefore query response time until joins can no longer be executed with the systems' current computational resources. To accommodate this problem, a *Cleanup Deamon* is used to extract old monitoring data from the local Repository, process the data by filtering values that are not needed and performing aggregation functions on larger time intervals to further reduce the size of the monitoring data and then dumping the data to the central CELAR DataBase for historical purposes. The Cleanup Deamon is activated either when the size of the local Repository exceeds a size specified in its configuration file or when a time interval expires. A NoSQL solution will be taken into consideration for the next version the CELAR Monitoring System, comparing the relational database approach against a NoSQL solution.

5.10 JCatascopia Web Service

The JCatascopia Web Service, as the name implies, is a web service that can be installed alongside a JCatascopia MS Server, allowing external entities, such as Application Users or CELAR modules (i.e. Decision Module, Multi-Level Metrics Evaluation Module, etc.) to communicate / interact with the Server. The Web Service utilizes the Jersey RESTful framework [Jersey] to provide a RESTful API with calls to obtain the status and number of monitoring Agents in the current deployment, to receive monitoring metrics from a specified (by ID) Agent for various timeframes and to add/remove subscriptions to composite metrics created by low-level metrics. Application Users access monitoring information through the c-Eclipse MS Visualization Tool that provides access to the web service through a graphical user interface used to represent metrics in a graphical manner.

5.11 JCatascopia Monitoring Visualization Tool

The current version (v1.0) of the JCatascopia Monitoring Visualization Tool provides the following features:

- **Deployment Page**: View graphically the number of running VMs, the status of each VM (i.e. up, down) and the subscriptions for the current application deployment.
- **Agent Page**: When selecting a VM, and consequently an Agent, from either the Deployment Page or from the drop-down menu, view static information concerning the VM (i.e. OS, total memory, etc.) and a list of the available monitoring metrics reported by the Agent residing on the VM. Furthermore, when selecting metrics from the provided list, monitor graphically incoming values of the requested metrics.
- **Subscription Page**: When selecting a subscription from either the Deployment Page or from the drop-down menu, view static information concerning the

selected subscription and graphicallythe reported values. VMs can be added / removed from a subscription at runtime.

Depicted in Figures 16, 17 and 18 are screenshots taken while testing JCatascopia v1.0 on Okeanos Public Cloud [okeanos]. In Figure 16, we can observe that 6 VMs are currently members of the application deployment and their IP addresses. Also we can observe that there are two subscriptions currently registered for the deployment. In Figure 17, we can observe the CPU usage of an Agent with the IP address 83.212.108.235 for the last 30 minutes. Depicted in Figure 18 is a subscription named *ClusterCPUTotal* reporting the average CPU usage for 3 VMs every 25 seconds with the following *subscription rule*[18]:

```
ClusterCPUTotal = AVG(cpuTotal)
MEMBERS = [83.212.120.155, 83.212.108.235, 83.212.105.22]
ACTION =PERIOD(25)
```

Listing 4: Cluster Total CPU Subscription Rule Example

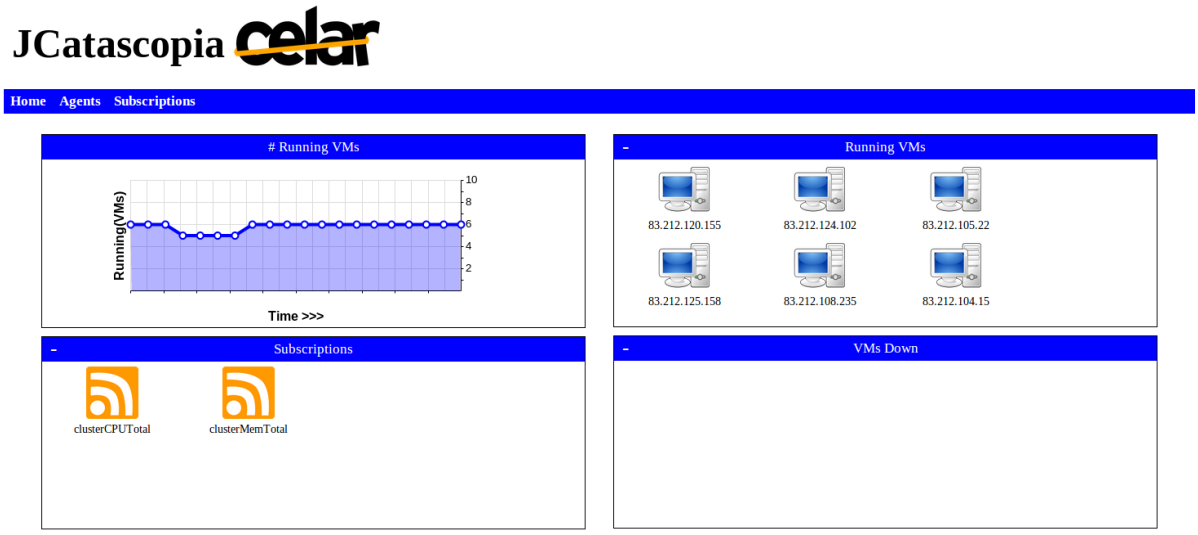

Figure 16: Monitoring Visualization Tool Deployment Page

[18]For simplicity, the elements in the Member list are shown as IP addresses but in reality the elements are Agent ID's

Figure 17: Monitoring Visualization Tool Agent Page

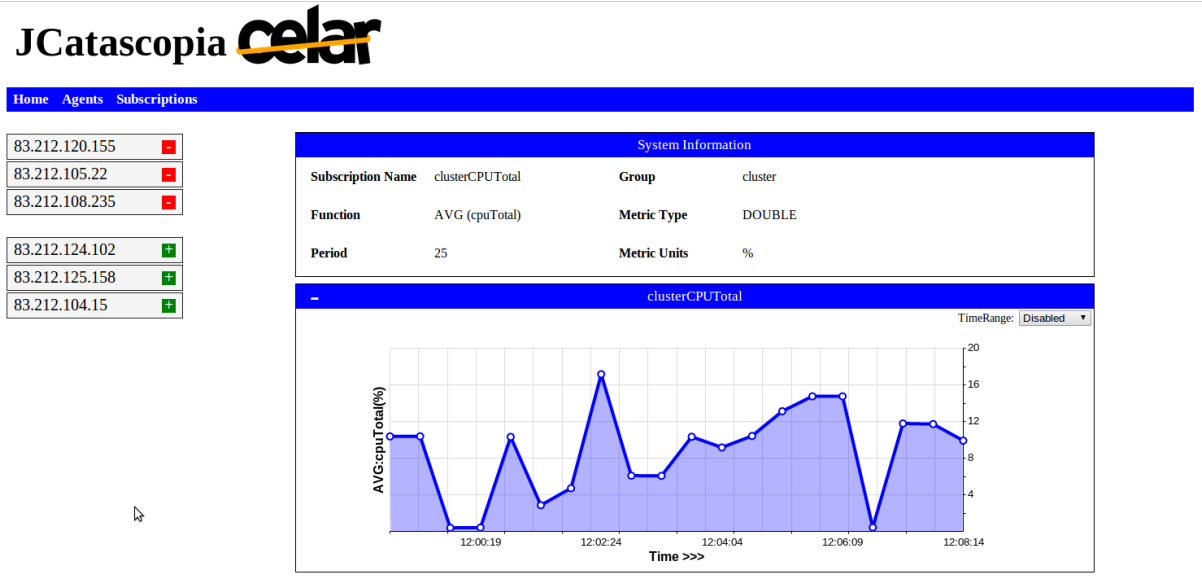

Figure 18: Monitoring Visualization Tool Subscription Page

5.12 JCatascopia Monitoring System v1.0 Code

The source code for the first version (v1.0) of JCatascopia Monitoring System can be found at GitHub under the following URL:

https://github.com/CELAR/cloud-ms

6 Implementation of the Multi-Level Metrics Evaluation Module

In this Section we provide a description of the Multi-Level Metrics Evaluation Module, focusing on the architecture and the components that comprise it. Furthermore we outline the functionality that the Multi-Level Metrics Evaluation Module will provide.

For realizing the functionality provided by the Multi-Level Metrics Evaluation Module, we deploy MELA[19], a framework for elasticity space monitoring and analysis as a service. MELA provides multi-dimensional analysis of elastic applications' behavior, classifying monitoring data in three dimensions: Cost, Quality, and Resource. Each dimension contains a set of metrics, and captures monitoring data about any monitored element (e.g., virtual machine, component or complex component) within a Cloud application that can be used for understanding the elastic behavior of that application.

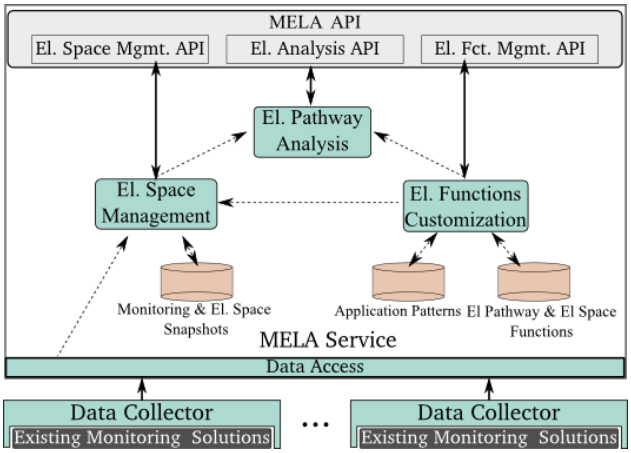

Figure 19: MELA Overview

In order to analyze the elastic behavior of a Cloud application, MELA uses the concept of *elasticity boundary*. An elasticity boundary describes the upper and lower allowed values over a set of metrics for a Cloud application. Another MELA core concept is *elasticity space*. An elasticity space is determined via an elasticity space function and captures all runtime metrics described in the user-defined elasticity boundary when a monitored element is in elastic behavior. Based on the elasticity space, MELA provides a mechanism for extracting characteristics of the elasticity space that can be used to predict its behavior, characteristics called "*elasticity pathway*". An elasticity pathway function is designed to perform a complex evaluation of the cloud service behavior, extracting characteristics that can be used to predict future behavior. One function could extract metric correlations, e.g., between throughput and cost/VM/h for a service unit, determining the influence of throughput on the cost. Another function could classify the cloud service behavior in usual and unusual combinations of metric values. The elasticity pathway function is applied to extract behavioral characteristics of monitored elements from the elasticity space, the quality of the extracted elasticity pathway being heavily influenced by the size and data in the elasticity space. Moreover, MELA allows the insertion of different elasticity pathway functions at different application structure levels (e.g., Application Component, Composite Component), providing a flexible mechanism for estimating the behavior of the application cost at multiple levels.

MELA contains a core MELA Service, and a lightweight Data Collector node (Figure 19). The Data Collector node is a customizable component that gathers from existing

[19] http://dsgwww.infosys.tuwien.ac.at/research/viecom/mela/

monitoring solutions monitoring data, associates it with the application structure (e.g., VM metrics or Application Level metrics), and sends it for processing and analysis to the MELA Service. Monitoring data is usually associated with a single level, e.g., virtual infrastructure, service topology or service unit. An important MELA feature is the linking of these levels through multi-level metric aggregation, which implies a configuration, defining for the monitored elements at each level the composition operations to be applied. This step feeds data into a repository which also contains the composition rules for building the monitoring snapshots. As the elasticity space is determined from monitoring snapshots, the El. Space Management unit also handles the monitoring snapshot construction and stores them in the Monitoring & El. Space Snapshots database. MELA exposes its functionality through REST services and Java API, providing methods for configuring MELA and analyzing elastic service behavior.

6.1 Composable Cost Evaluation using MELA

MELA implements the metric composition language presented in Section 4.4. We apply this mechanism for (i) Metrics Conversion, (ii) Cost Evaluation and (iii) Multi-Level Metric Aggregation. The metric composition mechanism is based on Composition Rules defined as a cascading sequence of operations which apply one or more operators over one or more operands. The operands can be metrics, other operations, or static values. The composition rule format is generic, enabling the definition of rules that can cover anything, from metric unit conversion, to cost policy evaluation, and multi-level metric aggregation.

```
<CompositionRuleTargetLevel="ComplexComponent" >
<SourceMetric name="dataOut" unit="GB/s"   level="Component"/>
<Operations>
<Operation name="SUM"/>
<Operation name="MUL" value=”0.12”/>
<Operation name="DIV" unit="no/s" sourceMetric="numberOfClients"
level="ComplexComponent"/>
</Operations>
<ResultingMetric name="dataCostPerClient" unit="$” type="COST"/>
</CompositionRule>
```

Listing 5: Example of MELA Metric Composition Rule

In Listing 5 we show in XML a complex multi-level MELA metric composition rule, which defines a sequence of three operations, starting from a “`dataOut`” metric, and obtaining the cost per data transfer per client for a `ComplexComponent` with a “`$`” measurement unit. The rule first defines an operation for summing the values of the “`dataOut`” metrics from all the Component children of a `ComplexComponent`. The result of the `SUM` operation is then multiplied by a `MUL` operation with the cost per GB of data transferred. The multiplication result is then divided by a `DIV` operation with the value of a “`clients`” metric retrieved from the same `ComplexComponent`.

We use this metric composition mechanism to define rules for converting metric measurement units to resemble the cloud pricing schemes. The cloud pricing schemes are in turn converted to metric composition rules, as sequence of operations that are applied over a particular metric. Finally, using more complex composition rules we aggregate monitoring data according to the application structure, from the Virtual Machine level up to the Application level.

6.2 Current MELA Prototype

The current MELA prototype is implemented in Java as a Maven [Maven] multi-module project (Figure 20). The prototype is structured in two major modules: *MELA-Core*, and *MELA-Celar*. The *MELA-Core* module contains the multi-level cost composition and monitoring data analysis functionality, which is independent from any monitoring data collection mechanism and underlying Cloud infrastructure. The MELA-Celar module contains the code needed to interact with other CELAR modules (e.g. JCatascopia), thus providing a flexible way of delivering MELA.

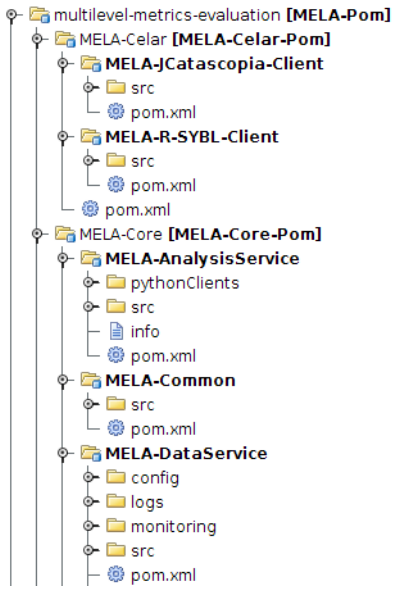

Figure 20:MELA Prototype Structure

6.2.1 MELA-Modules

In the following sub-sections we provide a short description of modules that comprise MELA and their sub-modules:

6.2.1.1 MELA-Core

MELA-Core consists of three sub-modules: *MELA-Common, MELA-DataService*, and the *MELA-AnalysisService*.

- **MELA-DataService**

  The module responsible for retrieving monitoring metrics from the specified monitoring system and, if needed, it provides a temporary metric storage space.

- **MELA-AnalysisService**

  The module responsible for retrieving data from the *MELA-DataService*.This module aggregates, analyzes, and performs multi-level evaluation on the received data, in terms of cost, quality and resource usage. The *MELA-AnalysisService* functionality is not only limited to performing data analysis since it also provides the MELA Web Graphical User Interface, and the MELA RESTful API. In the future we will consider moving the *MELA Web Graphical User Interface* in a separate Maven project, as it already relies on the MELA RESTful API to retrieve data.

- **MELA-Common**

  The module responsible forstoring the data model entities used in order for the *MELA-AnalysisService* and the *MELA-DataService*to interact. Having the common

data model as a separate project enables the model to be inserted as a dependency in any future MELA modules, increasing the architecture flexibility.

6.2.1.2 MELA-CELAR

MELA-CELARconsists of two sub-modules: the *MELA-JCatascopia-Client* and the*MELA-Client.*

- **MELA- JCatascopia-Client**
  The module utilized to interact with JCatascopia, in order to retrieve monitoring metrics. This module is embedded as a library reference in the *MELA-DataService.*
- **MELA-Client**
  A client utilized by CELAR interested modules, such as the Decision Module or the c-Eclipse Visualization Tool, in order to extract cost-enriched metrics and estimations via the MELA RESTful API. Figure 20 depicts an exemplary client (named *MELA-R-SYBL-Client*) utilized by the current implementation of the Decision Module.

6.3 Early Results

Task 4.2 (Composable Cost Evaluation) has been running from Month 4, and it has another 6 months to run, until Month 18. In these first 8 months we have focused on establishing the foundations for composable cost evaluation, and multi-level metrics analysis. In the current MELA prototype we can compose metrics collected from JCatascopia at multiple levels using our metric composition language. We use this composition mechanism to define simple cost composition models. Complex cost composition models will be developed in the next months and will be presented in v2.0.

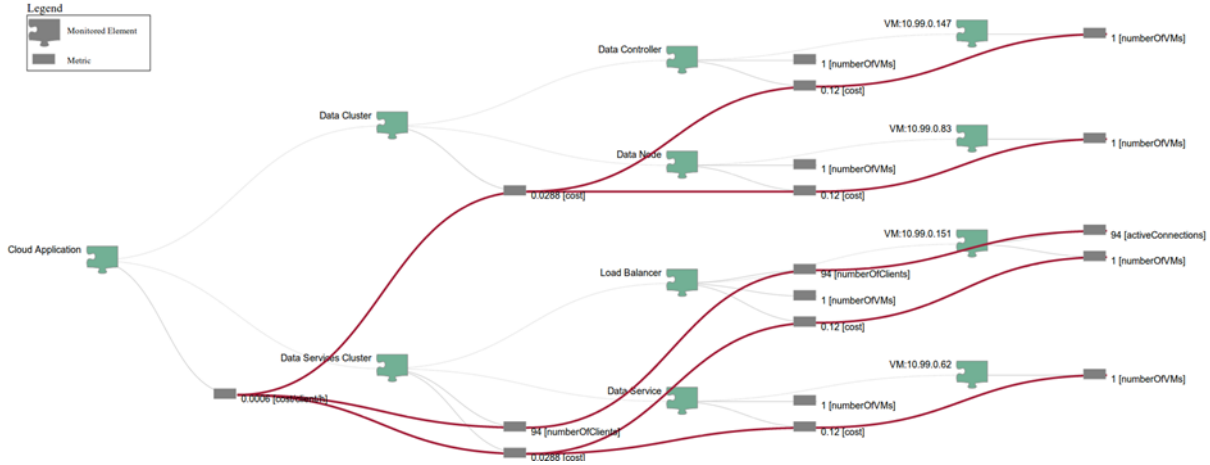

Figure 21:Multi-Level Cost Composition Example

In Figure 21 we show a snapshot of the MELA Graphical User Interface, containing an example of multi-level cost composition that can be performed with the current MELA prototype. In this case, we have a Cloud Application structured according to the Application Model presented in Section 4.2. The application has two Complex Components: *Data Cluster*, and *Data Services Cluster*. The *Data Cluster* contains in turn a *Data Controller instance*, and can have several *Data Node instances* (the Data Cluster is elastic as it supports dynamic addition and removal of Data Node instances). In turn, the Data Services Cluster complex component contains a *Load Balancer*, and several instances of *Data Service* components. The *Data Service* components access the *Data Cluster*, and the *Load Balancer* distributes client requests between them. In this scenario, cost is composed for each of the application Component instances, Complex Component

instances and for the whole Cloud Application. For the Component instances the cost is determined by the number of running Virtual Machines multiplied by 0.12 (assumed cost per VM per hour). For the Complex Component instances the cost is computed by summing the costs of their respective Component instances. For the whole Cloud Application, cost is reported as cost per client. The cost per client is obtained by dividing the sum of the cost of the Complex Components, by the number of clients served by the application obtained from the Load Balancer component. The cost composition rules used in this example are shown in Listing 6.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<MetricsCompositionRules>

<CompositionRuleTargetMonitoredElementLevel="SERVICE_UNIT">
<ResultingMetric type="RESOURCE" measurementUnit="$" name="cost"/>
<Operation value="0.12" type="MUL">
<Operation MetricSourceMonitoredElementLevel="VM" type="SUM">
<ReferenceMetric type="RESOURCE" name="numberOfVMs"/>
</Operation>
</Operation>
</CompositionRule>

<CompositionRuleTargetMonitoredElementLevel="SERVICE_TOPOLOGY">
<TargetMonitoredElementID>Data Services Cluster</TargetMonitoredElementID>
<ResultingMetric type="RESOURCE" measurementUnit="no" name="numberOfClients"/>
<Operation MetricSourceMonitoredElementLevel="SERVICE_UNIT" type="KEEP">
<ReferenceMetric type="RESOURCE" measurementUnit="no" name="numberOfClients"/>
<SourceMonitoredElementID>Load Balancer</SourceMonitoredElementID>
</Operation>
</CompositionRule>

<CompositionRuleTargetMonitoredElementLevel="SERVICE_TOPOLOGY">
<ResultingMetric type="RESOURCE" measurementUnit="$" name="cost"/>
<Operation value="0.12" type="MUL">
<Operation MetricSourceMonitoredElementLevel="SERVICE_UNIT" type="SUM">
<ReferenceMetric type="COST" name="cost"/>
</Operation>
</Operation>
</CompositionRule>

<CompositionRuleTargetMonitoredElementLevel="SERVICE">
<ResultingMetric type="RESOURCE" measurementUnit="$" name="cost/client/h"/>
<Operation MetricSourceMonitoredElementLevel="SERVICE" type="DIV">
<Operation MetricSourceMonitoredElementLevel="SERVICE_TOPOLOGY" type="SUM">
<ReferenceMetric type="RESOURCE" name="cost"/>
</Operation>
<Operation MetricSourceMonitoredElementLevel="SERVICE_TOPOLOGY" type="KEEP">
<ReferenceMetric type="RESOURCE" name="numberOfClients"/>
<SourceMonitoredElementID>Data Services Cluster</SourceMonitoredElementID>
</Operation>
</Operation>
</CompositionRule>
```

Listing 6: Multi-Level Cost Composition Rules Example

6.4 JCatascopia and MELA Integration

The role of the CELAR Monitoring System is to collect process and distribute monitoring metrics to CELAR interested modules. JCatascopia is utilized by the CELAR System to gather monitoring metrics originating from the virtualization layer. On one hand such metrics might convey resource allocation and usage of instantiated VM's, or

report the performance of applications deployed on a Cloud infrastructure. The *JCatascopia Subscription Mechanism*makes monitoring metrics available either to Application Users or CELAR interested modules such as the Decision Module or the Multi-Level Metrics Evaluation Module. By subscribing to a metric stream, entities can receive (i) raw metrics per VM, (ii) aggregated metrics or (iii) by applying subscription filters, as described in Section 5.7, higher level metrics (i.e. grouped metrics), thus facilitating the application model described in section 4.2.

Nevertheless, JCatascopia cannot be utilized directly to calculate cost evaluations and estimations since it is not aware of the application topology. Secondly, since JCatascopia is by nature Cloud provider independent, it does not have access to specific pricing schemes. Finally, it does not have any knowledge regarding to which application (or composite) components new VMs belong to, when elasticity actions are enforced.The role of MELA, the Multi-Level Metrics Evaluation Module, is to enrich monitoring metrics with cost information and to provide future cost estimations to any entities interested in such information. MELA utilizes the JCatascopia Subscription Mechanism to subscribe to metric streams in order to process monitoring metrics based on the application description, thereby providing cost evaluation and estimation. For these reasons MELA calculates cost by acquiring monitoring metrics from JCatascopia.

The Decision Module and Application Users (the latter via the c-Eclipse MS Visualization Tool), will take advantage of the MELA RESTful API to retrieve multi-level monitoring data aggregated according to the application structure. This data will be enriched with cost data and information used in the decision process. Additionally, the Decision Module can also access MELA for retrieving estimations about the future behavior of the cloud application, for each application structure level or component instance, as a way of improving the decision process with proactive decisions.

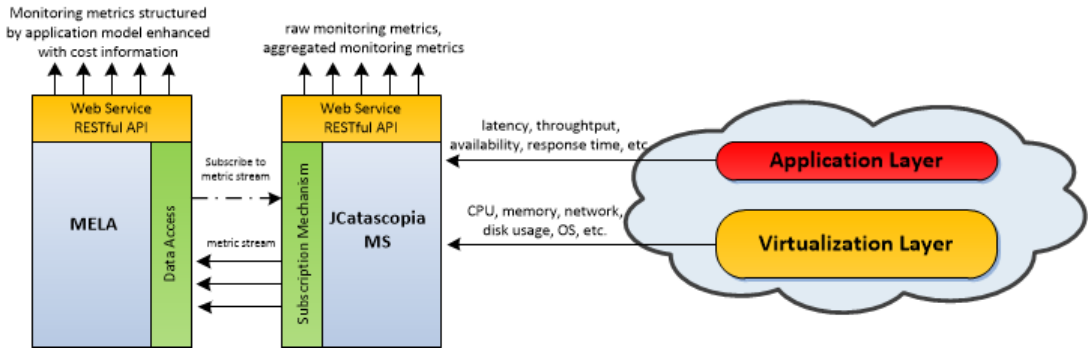

Figure 22: JCatascopia and MELA Integration

Following, is a simple example illustrating the JCatascopia and MELA interactions: suppose an Application User is interested in acquiring cost information concerning his distributed database cluster where billing is based on the amount of disk storage.Consider also that the initial deployment is comprised of 3 VMs. JCatascopia Monitoring Agents are deployed by the CELAR System on each of the application VMs and monitoring metrics are gathered and distributed to a JCatascopiaMonitoring Server dedicated for the application deployment. MELA can be instructed to retrieve monitoring metrics, process them cost-wise and report the total cost to the Application User. A subscription rule that MELA would use to subscribe to a JCatascopia metric stream is the following:

```
dbClusterDiskUsedTotal = SUM(diskUsedVM)
MEMBERS = [dbNodeID1, dbNodeID2, dbNodeID3]
ACTION = PERIOD(45)
```

Listing 7: Subscription Rule Example

where, `diskUsedVM` is the disk storage currently allocated on each database VM and by applying the grouping function SUM we can obtain the total amount of disk storage for the members in the cluster periodically (e.g. every 45 seconds). After subscribing to a metric stream, metrics will be forwarded to MELA to compute the cost by applying a metric composition rule which in turn,*multiplies the cost (0.12$) of each Gigabyte (GB) of used database storage with the value of the "*`dbClusterDiskUsedTotal`*" metric value*as depicted in Listing 8.

```
<CompositionRuleTargetLevel="ComplexComponent">
<SourceMetric name="dbClusterDiskUsedTotal" unit="GB"
level="ComplexComponent"/>
<Operations>
<Operation name="MUL" value="0.12"/>
</Operations>
<ResultingMetric name="costPerDBCluster" unit="$" type="COST"/>
</CompositionRule>
```

Listing 8: Cost Metric Composition Rule Example

When an elasticity action is enforced by the Decision Moduleresulting in a change to the application topology, MELA notifies the Monitoring System in order for the new database nodes id's to be added to the cluster subscription, therebysuccessfully updating the subscription rule and concluding a MAPE[20] loop.

6.5 MELA Code

The source code for the current MELA prototype can be found at GitHub under the following URL:

https://github.com/CELAR/multilevel-metrics-evaluation

[20] MAPE loop: Monitoring, Analysis, Planning and Execution

7 Integration with Other CELAR Modules

In this section we present the integration of the Cloud Information and Performance Monitor Layer with other CELAR modules. Depicted in Figure 23 is an updated version of the Monitoring workflow presented in D1.1 [D1.1 – Section 3.4.4]. This figure demonstrates the communication and data exchange between the components of the Cloud Information and Performance Layer and other CELAR modules after the user has successfully deployed his application on a Cloud provider.

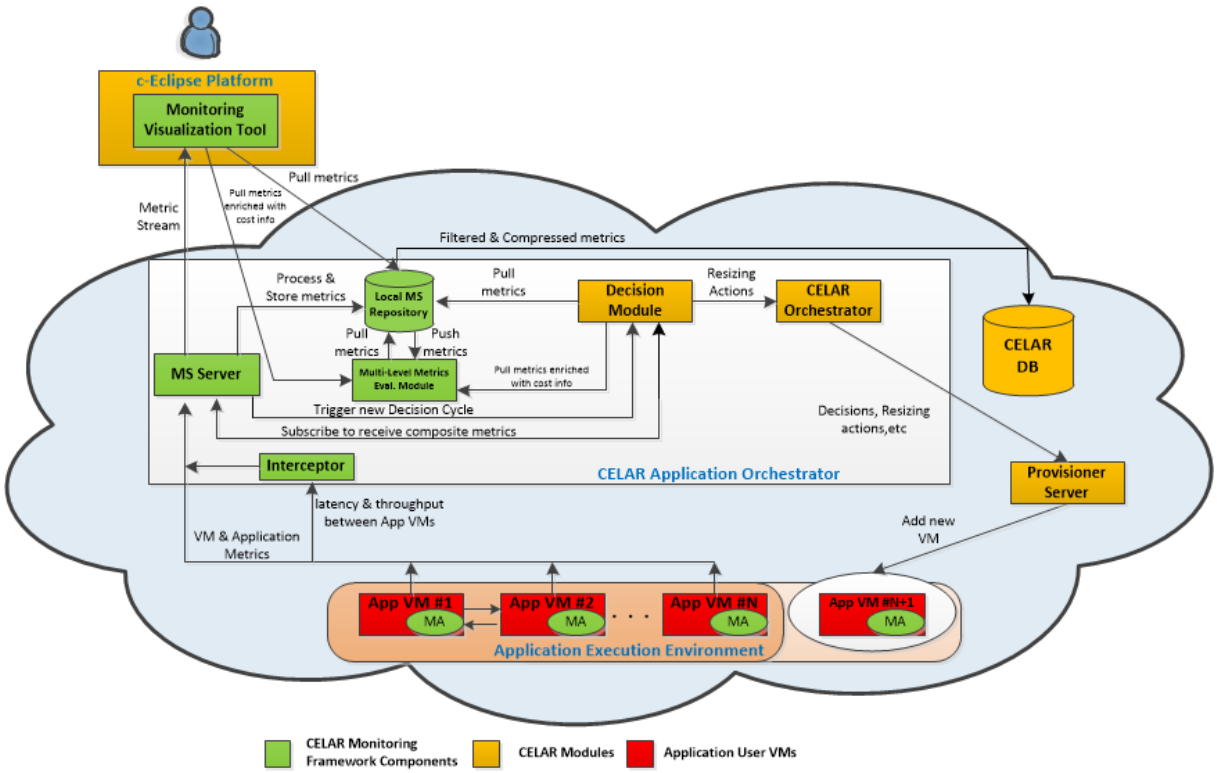

Figure 23: Cloud Information and Performance Layer Workflow Diagram

When an Application User successfully submits an application for deployment on a *CELARized* cloud provider, the CELAR Manager launches instances of various CELAR modules responsible for monitoring and resizing the current application. This takes place inside the Application Orchestrator VM[21] which is dedicated exclusively for the orchestration of the application deployment. This deployment strategy ensures both scalability (as each instance handles a single application), and application security, as information about multiple applications is not processed by the same Application Orchestration VM.

To monitor an application via the CELAR System, a JCatascopia Monitoring Server, a local Monitoring Repository and the Multi-level Metrics Evaluation Module (MELA instance) must be installed on the Application Orchestrator VM. AJCatascopia Monitoring Agent is installed on each of the application VMs of the initial deployment. Agents are non-intrusive in terms of resource consumption and run in the background as system services', thus not interfering with the application execution.

[21]A detailed description of the Application Orchestrator VM, Interceptor, CELAR Orchestrator and the Provisioner Server is given in Deliverable D1.1 [D1.1]

Through the lifecycle of a deployed application, Probes gather monitoring information from each of the application VMs, while Agents process the gathered metrics and distribute them to the corresponding Monitoring Server. It should be noted that even though not depicted in Figure 23, a hierarchy of Intermediate Servers deployed on VMs (considered as management VMs), as described in Section 4.3, can reside between Agents and the root Application Monitoring Server in order to provide scalability due to the large number of Agents. The Interceptor also sends metrics to the Server concerning latency and throughput which is calculated among the application VMs. When monitoring metrics are received by the Server, they are further processed and stored to the local Monitoring Repository.

After storing the received metrics, they are available for consumption by Application Users, the Decision Module and, of course by, the Multi-Level Metrics Evaluation Module. Both the Decision Module and Application Users may consume monitoring metrics enriched with cost information offered by the Multi-Level Metrics Evaluation Module. These modules interact with the Multi-Level Metrics Evaluation Module in a loosely coupled manner using a RESTful API.

Application Users can access monitoring metrics via the c-Eclipse Monitoring Visualization Tool, subscribing to metrics of their interest using the Monitoring System'sRESTful API. Metrics are then pushed to the Visualization Tool when made available. Users may also pull metrics values upon request, instead of subscribing to a metric stream. In a similar manner, the Decision Module can use either a push or pull metric delivery mechanism. Metrics can be pulled by the Decision Module in order to have absolute control over the time granularity of when it receives metrics, preventing metric flooding. An exemplary downside though of a pull-based mechanism is evident during the occurrence of unexpected critical events that often require immediate attention. Unfortunately, due to the large time interval between pull requests these events will be addressed too late. Because situations like this may happen frequently, the Monitoring System can be enabled to push metrics to the Decision Module, consequently triggering a new decision cycle if such critical events occur.

When the application execution environment expands or contracts, the Monitoring System follows in a similar manner revealing its elastic capabilities. Upon resizing the application deployment by adding a new application VM, as depicted in Figure 23, a new Monitoring Agent is installed on the new VM ping-ing the Server to inform of its presence. Similarly, if the action plan produced by the Decision Modules states that VMs must be removed from the application deployment, then the Server will detect the absence of the Agents residing on the deducted VMs.

8 Conclusions

In this deliverable we have provided a detailed description of the CELAR Monitoring System and the Multi-Level Metrics Evaluation Module which comprise the CELAR Cloud Information and Performance layer. We have documented the State of the Art in the field of Cloud Monitoring and Cost Evaluation, and we have given a detailed analysis of the requirements a Cloud Monitoring System and a Multi-Level Metrics Evaluation Module must have, together with the challenges to overcome when monitoring user applications in a rapidly adapting application execution environment. Furthermore, we have focusedon the design concepts, the CELAR Monitoring System's architecture, the description of the main components that comprise it and the functionality available in v1.0. Finally we have presented implementation specifics, alongside with screenshots, of the CELAR Monitoring System, describing the features and the procedures followed during the development of the each component. A detail analysis of the Multi-Level Metrics Evaluation Module and how it is integrated with the Monitoring System to enrich monitoring metrics with cost information was also provided.

Task 4.1 (Monitoring Tool) has been running from the beginning, and throughout these first 12 months we have focused on delivering the first version of the CELAR Monitoring System that encapsulates the functionality required to monitor elastically adaptive applications deployed on Cloud infrastructures. The development of the CELAR Monitoring System will be further pursued in an attempt to enhance the functionality of JCatascopia. Towards the implementation of v2.0 of JCatascopia focus will be given to the following: (i) further pursue adaptive sampling and filtering at Probe level, (ii) provide CELAR with the application Probes suitable for collecting the requested metrics from the consortium pilot applications, (iii) minimize runtime impact Agents impose on application VMs and (iv) enhance the subscription mechanism at Server level to allow for, even more, complex high-level metrics to be specified with subscriptions.

Task 4.2 (Composable Cost Evaluation) has been running from Month 4, and in these first 8 months we have focused on establishing the foundations for composable cost evaluation, and multi-level metrics analysis. A prototype implementation of the Multi-Level Metrics Evaluation Module has been provided (MELA), together with early results. In the next 6 months we will work on defining and supporting complex cost models, and finer grained multi-level metric evaluation.We will further focus on estimating and predicting cost, quality and resource usage of elastic Cloud applications at multiple levels. Different methods and algorithms for extracting behavioral characteristics of elastic cloud applications will be studied and implemented, towards estimating cost, quality and resource usage.

9 References

[Aceto 2013] Giuseppe Aceto, AlessioBotta, Walter de Donato, Antonio Pescapè, "Cloud monitoring: A survey", Computer Networks, Volume 57, Issue 9, 19 June 2013, Pages 2093-2115, ISSN 1389-1286, http://dx.doi.org/10.1016/j.comnet.2013.04.001

[Al-Kiswany 2013] Samer Al-Kiswany, HakanHacıgümüş, Ziyang Liu, and JaganSankaranarayanan. 2013. "Cost exploration of data sharings in the cloud". *In Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*. ACM, New York, NY, USA, 601-612. DOI=10.1145/2452376.2452447 http://doi.acm.org/10.1145/2452376.2452447

[Amazon EBS] Amazon Elastic Block Store, http://aws.amazon.com/ebs/

[Amazon S3] Amazon Simple Storage Service, http://aws.amazon.com/s3/

[Andreozzi 2005] Sergio Andreozzi, Natascia De Bortoli, Sergio Fantinel, Antonia Ghiselli, Gian Luca Rubini, GennaroTortone, and Maria Cristina Vistoli. 2005. "GridICE: a monitoring service for Grid systems". Future Gener.Comput. Syst. 21, 4 (April 2005), 559-571. DOI=10.1016/j.future.2004.10.005 http://dx.doi.org/10.1016/j.future.2004.10.005

[Apache Server] Apache HTTP Server, http://httpd.apache.org/

[Armbrust 2010] Michael Armbrust , Armando Fox , Rean Griffith , Anthony D. Joseph , Randy H. Katz , Andrew Konwinski , Gunho Lee , David A. Patterson , Ariel Rabkin , MateiZaharia. "Above the Clouds: A Berkeley View of Cloud Computing ".

[AutoScaling] http://aws.amazon.com/autoscaling/

[Azure] http://www.windowsazure.com/en-us/manage/services/cloud-services/how-to-monitor-a-cloud-service/

[AzureWatch] http://www.paraleap.com/azurewatch

[Buell 2011] Buell, K.; Collofello, J., "Transaction Level Economics of Cloud Applications," Services (SERVICES), 2011 IEEE World Congress on , vol., no., pp.515,518, 4-9 July 2011 DOI=10.1109/SERVICES.2011.12

[Carvalho 2011] de Carvalho, M.B.; Granville, L.Z., "Incorporating virtualization awareness in service monitoring systems," Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on , vol., no., pp.297,304, 23-27 May 2011 doi: 10.1109/INM.2011.5990704

[Cassandra] Apache Cassandra Project, http://cassandra.apache.org/

[Chaisiri 2012] SivadonChaisiri, Bu-Sung Lee, DusitNiyato, "Optimization of Resource Provisioning Cost in Cloud Computing," IEEE Transactions on Services Computing, vol. 5, no. 2, pp. 164-177, Second, 2012, DOI=10.1109/TSC.2011.7

[Clayman 2010] Clayman, S.; Galis, A.; Mamatas, L., "Monitoring virtual networks with Lattice". Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP , vol., no., pp.239,246, 19-23 April 2010
doi: 10.1109/NOMSW.2010.5486569

[CloudWatch]http://aws.amazon.com/cloudwatch/

[Copil 2013] Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, SchahramDustdar, "SYBL: An Extensible Language for Controlling Elasticity in Cloud Applications," ccgrid, pp.112-119, 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, 2013

[D1.1] User Requirements and System Architecture V1, Deliverable, CELAR Project, https://wiki.celarcloud.eu/doku.php?id=CELAR_Project:Deliverables:Deliverables_1.X:D1.1

[D2.1] Application Description Tool V1, Deliverable, CELAR Project, https://wiki.celarcloud.eu/doku.php?id=CELAR_Project:Deliverables:Deliverables_2.X:D2.1

[D5.1] Decision Process for On-demand Elasticity Report, CELAR Project, https://wiki.celarcloud.eu/doku.php?id=CELAR_Project:Deliverables:Deliverables_5.X:D5.1

[Emeakaroha 2010] Vincent C. Emeakaroha, IvonaBrandic, Michael Maurer, SchahramDustdar. Low Level Metrics to High Level SLAs - LoM2HiS framework: Bridging the gap between monitored metrics and SLA parameters in Cloud environments. The 2010 High Performance Computing and Simulation Conference (HPCS 2010), in conjunction with The 6th International Wireless Communications and Mobile Computing Conference (IWCMC 2010), Caen, France

[Eugster 2003] Patrick Th. Eugster, Pascal A. Felber, RachidGuerraoui, and Anne-Marie Kermarrec. 2003. “The many faces of publish/subscribe”. ACM Comput.Surv. 35, 2 (June 2003), 114-131. DOI=10.1145/857076.857078

[FCO] Flexiant Cloud Orchestrator, Documentation and API, http://docs.flexiant.com/display/DOCS/Introduction+to+Flexiant+Cloud+Orchestrator

[Flexiant] http://www.flexiant.com/

[Ganglia] http://ganglia.sourceforge.net/

[Hayashibara 2002] N. Hayashibara, A. Cherif, and T. Katayama."Failure Detectors for Large-Scale Distributed Systems", *In Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, Suita, Japan. IEEE Computer Society, 2002, pp.404-409

[HypericHQ] http://www.hyperic.com/

[JeroMQ] https://github.com/zeromq/jeromq

[Jersey] Jersey RESTful Web Service Java Framework, https://jersey.java.net/

[Khanafer 2013] Khanafer, Ali; Kodialam, Murali; Puttaswamy, Krishna P.N., "The constrained Ski-Rental problem and its application to online cloud cost optimization," INFOCOM, 2013 Proceedings IEEE , vol., no., pp.1492,1500, 14-19 April 2013, doi=10.1109/INFCOM.2013.6566944

[Katsaros 2011] G. Katsaros, R. Kübert, G. Gallizo, Building a service-oriented monitoring framework with REST and nagios, in: 2011 IEEE International Conference on Services Computing (SCC), 2011, pp. 426–431.

[Konstantinou 2013] IoannisKonstantinou, VerenaKantere, DimitriosTsoumakos, and NectariosKoziris. 2013. COCCUS: self-configured cost-based query services in the cloud. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13). ACM, New York, NY, USA, 1041-1044. http://doi.acm.org/10.1145/2463676.2465233

[Kutare 2011] MahendraKutare , Greg Eisenhauer , Chengwei Wang , Karsten Schwan , Vanish Talwar , Matthew Wolf, "Monalytics: online monitoring and analytics for managing large scale data centers", Proceedings of the 7th international conference on Autonomic computing, June 07-11, 2010, Washington, DC, USA doi: 10.1145/1809049.1809073

[Massie 2004] Matthew L Massie, Brent N Chun, David E Culler, “The ganglia distributed monitoring system: design, implementation, and experience”, Parallel Computing, Volume 30, Issue 7, July 2004, Pages 817-840, ISSN 0167-8191, http://dx.doi.org/10.1016/j.parco.2004.04.001.

[Maven] Apache Maven Project, http://maven.apache.org/

[Montes 2013] Jesús Montes, Alberto Sánchez, BunjaminMemishi, María S. Pérez, Gabriel Antoniu, GMonE: A complete approach to cloud monitoring, Future Generation Computer Systems, ISSN 0167-739X, http://dx.doi.org/10.1016/j.future.2013.02.011

[Munin] http://munin-monitoring.org/

[MySQL] http://www.mysql.com/

[Nagios] http://www.nagios.org/

[Newman 2003] Harvey B. Newman, IosifLegrand, Philippe Galvez, Ramiro Voicu, CatalinCirstoiu: “MonALISA : A Distributed Monitoring Service Architecture”. CoRRcs.DC/0306096 (2003)

[NIST 2011] NIST Cloud Computing Reference Architecture. September 2011, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[Okeanos] http://okeanos.grnet.gr

[RackSpace] http://www.rackspace.com/cloud/monitoring/

[Schubert 2010] L. Schubert, K. Jeffery, B.Neidecker-Lutz. "The Future of Cloud Computing".Report of the Cloud Computing Expert Working Group. European Commission, 2010, http://cordis.europa.eu/fp7/ict/ssai/docs/cloud-report-final.pdf

[Schubert 2012] L. Schubert, K. Jeffery. "Advances in Clouds", Report of the Cloud Computing Expert Working Group. European Commission, 2012, http://cordis.europa.eu/fp7/ict/ssai/docs/future-cc-2may-finalreport-experts.pdf

[sFlow] http://www.sflow.org/

[Sharma 2011] Upendra Sharma; Shenoy, P.; Sahu, S.; Shaikh, A., "A Cost-Aware Elasticity Provisioning System for the Cloud," Distributed Computing Systems (ICDCS), 2011 31st International Conference on , vol., no., pp.559,570, 20-24 June 2011, DOI=10.1109/ICDCS.2011.59

[Truong 2010] Hong-Linh Truong, SchahramDustdar, "Composable cost estimation and monitoring for computational applications in cloud computing environments", Procedia Computer Science, Volume 1, Issue 1, May 2010, Pages 2175-2184, ISSN 1877-0509, http://dx.doi.org/10.1016/j.procs.2010.04.243.

[vCenter] http://www.vmware.com/products/vcenter-server/overview.html

[VMware] http://www.vmware.com/

[vSphere] http://www.vmware.com/products/datacenter-virtualization/vsphere/overview.html

[Wang 2012] C. Wang, I. A. Rayan, G. Eisenhauer, K. Schwan, V. Talwar, M. Wolf, and C. Huneycutt. "VScope: Middleware for Troubleshooting Time-Sensitive Data Center Applications". *In Proceedings of the 13th International Middleware Conference (Middleware '12)*, 2012.

[Wee 2011] Sewook Wee, "Debunking Real-Time Pricing in Cloud Computing," Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on , vol., no., pp.585,590, 23-26 May 2011DOI= 10.1109/CCGrid.2011.38

[Xen] http://www.xenproject.org/

[YCSB] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Russell Sears. "Benchmarking Cloud serving systems with YCSB" *In the proceedings of the 1st ACM symposium on Cloud computing* (SoCC 2010). ACM, New York, NY, USA, pages 143-154. DOI=10.1145/1807128.1807152. http://dl.acm.org/citation.cfm?id=1807152

[Zabbix] http://www.zabbix.com/

[ZMQ] http://zeromq.org/