# QoS Instrumentation to Support Cloud Computing SLA Assurance

Mustapha Ait-Idir, Nazim Agoulmine, Rafael Tolosana Calasanz, Javier Baliosian

# QoS Instrumentation to Support Cloud Computing SLA Assurance

Mustapha Ait-Idir[1], Nazim Agoulmine[1], Rafael Tolosana Calasanz[2], and Javier Baliosian[3]

[1] IBISC Lab, University of Evry Val d'Essonne, Evry Val d'Esssonne, France
[maitidir, nazim.agoulmine]@ibisc.univ-evry.fr
[2] Universidad de Zaragoza, Zaragoza, Spain
rafaelt@unizar.es
[3] Universidad de la Repblica de Uruguay, Montevideo, Uruguay
baliosian@fing.edu.uy

**Abstract.** Today, organizations and individuals are moving from proprietary servers to the cloud computing to benefit from the powerful advantages of the theoretical unlimited resources and computation power. The principle of PAYG (*Pay As You Go*) makes this new paradigm more attractive and provides a rapid growing and extension. However, using a multi-tenancy environment brings new challenges in terms of security, reliability and QoS (*Quality of Service*). Thus, to fulfill cloud customer expectations in cloudified applications, a relationship must be formalized in a signed contract (i.e. SLA or *Service Level Agreement*). The service provider monitors the application components to prevent breaches during the SLA life cycle and guarantees the enforcement of the promised QoS. Usually a service is monitored as a whole instead of considering each component in the service individually for a best intervention and an optimized scaling. In this paper, we propose a component-based application monitoring mechanism. We have adopted the OCCI (*Open Cloud Computing Interface*) framework to model our agreement and instrument a response-time metric for SLA enforcement. The goal is to prevent latency and SLA violation by identifying responsible components involved in the request chain. Our priority is to take actions to prevent the breaches or at least act for a resolution when a problem manifests.

## 1 Introduction

Based on the best of our knowledge, there is a lack of clear specifications of a clear Quality of Service (QoS) framework in cloud computing. This concept actually varies from one IT domain to another. QoS is often applied to a system, a service or a component and is defined in [22] as the overall service performance from the end user perspective. Popularity of Internet services has required the need to provide customers with more than best-effort support allowing them to choose between providers based on the level of the provided services. QoS has always been one of the key differentiation factors among service providers and this is also true in Cloud Computing (CC). CC QoS could be *Functional* (functionality and behavior) or *Non Functional* (criteria for evaluation and operation). A

QoS metric could also be qualitative (e.g. user experience) or quantitative (e.g. response-time). There is therefore a need to define what needs to be monitored and analyzed to assess whether the level of QoS is satisfactory in the signed SLA contract as well as if is respected.

### 1.1 Problem Statement

Cloud Computing QoS is a very important aspect as a customer will only experience the launching and the resuming of a service provisioning request regardless how many underlying components and/or services are actually activated to provision the customer application. In case of resources shortage, the Cloud Computing scaling mechanism may only impact the overloaded components instead of the overall application. Often, when scaling an application we scale a VM (*Virtual Machine*) regardless of which component in the application is causing a bottleneck or a latency that may add an overhead (time and cost) for resource provisioning and releasing. For this reason, in this paper, we propose to accurately identify and monitor all the components status to achieve a better decision-making based on the collected information. Always, when processing a client request, several components may interact together to handle it, consequently increasing the complexity of investigating any QoS attribute failure. The more the accuracy in monitoring, the faster the resolution of any detected problem is. By using this approach, it will be possible to improve the performance of the provided service by reducing outage, MTSO (*Mean-Time Switch Over*) and MTSR (*Mean-Time System Recovery*) duration. For instance, in case of response-time performance QoS, the *requester* component will be tightly coupled with the *requested* component. Therefore, the more the requested component fulfills the SLA, the more the requester component will process a request in time. In this paper, we propose to put an enforcement framework in place in order to monitor the response-time performance of these components using smart data monitoring and decision-making algorithm, either to prevent any latency or to intervene immediately after a detected problem. We will describe how to measure a response-time metric based on acceptable QoS values defined in the SLA and how to pinpoint potential failing components.

### 1.2 Quality of Service Concept

Most existing state of art contributions on QoS assurance have focused on two main aspects: *QoS modeling using Ontology* and *QoS Instrumentation*. QoS modeling using ontology aims to provide definition, terminology, metrics unit and all areas where QoS should be applied [4, 11, 7, 13, 6, 10, 12, 17]. QoS instrumentation is related to measurement, enforcement techniques and monitoring [20, 3, 9, 21]. Ontology is very important when defining a common understanding of SLS (*Service Level [Agreement] Specifications*) parameters. To enforce this concept, the NIST (*National Institute of Standards and Technology*) has proposed an abstract overview for metrics definitions and measurements [18]. The cloud popularity pushes service providers to offer more QoS metrics monitoring and

reporting by exposing transparently their semantics. Thus, several authors emphasized their importance for CC adoption and attractiveness.

### 1.3 Paper Structure

The remaining of the paper is organized as follows: Section 2 presents the related works mainly in the areas of QoS instrumentation and enforcement. Section 3 presents the proposed solution. The implementation and use case are presented in Section 4. Finally, Section 5 concludes this work present some important future directions.

## 2 Related Work on QoS Enforcement

Understanding the semantics of QoS metrics is very important. The QoS Ontology model gathers standardization, definition and terminology in this area while instrumentation is more focused on the metrics for monitoring and measurements. Monitoring is mandatory for QoS validation, such as verifying whether a QoS metric instance is within acceptable thresholds, as specified in the SLS of the SLA. Oftentimes QoS parameters are specified in the SLA and are monitored by the provider to assess whether SLA is violated. To achieve this objective, the monitoring and SLA management systems should collect performance data from the underlying monitored resources and map them to the target QoS parameters. SLA monitoring is one of the seven functions of SLM (*Service Level Management: creation, negotiation, provisioning, monitoring, maintenance, reporting and assessment*) even if QoS properties may differ from one service to another. User experience is itself considered as a QoS such as QoBiz (*Quality of Business*) metric. The work done in [4] provides a reasonable specification range for the QoS metrics values and describes the service operations (e.g. migration) to keep the overall QoS acceptable. In the work presented in [11], authors assessed the QoSS (*Quality of Security Service*) concept to verify whether it may improve security and system performance in a QoS-Aware distributed environment. Security is studied as a dimension of QoS and it is related to a system that was deployed in several sites. In such a case, a managed resource may have a predictable and efficient allocation and utilization. Hence, if a resource is overloaded, the provider can define priority actions such as postponing tasks or even canceling and terminating jobs. Authors in [7] proposed another approach based on the W3C (*World Wide Web Consortium*) QoS specifications for Web services to identify relevant attributes when selecting a service provider. The considered QoS attributes include: *Performance, Scalability, Reliability, Accuracy, Integrity, Robustness, Availability, Interoperability, Accessibility and Security.* Current standards in Web services such as WSDL (*Web Service Definition Language*) mostly supports the description of functional aspects of interfaces rather than QoS specification and the used terminology is somehow ambiguous when defining quality attributes. Authors in[7] have therefore proposed an interesting classification for QoS attributes and sub-attributes as depicted in the Table 1.

| Attribute | Sub Classes |
|---|---|
| Functionality | Suitability, Accuracy (or Correctness), Security , Interoperability, Compliance. |
| Reliability | Maturity, Fault Tolerance, Recovery, Compliance, Robustness, Availability, Integrity. |
| Efficiency | Time Behavior (or Performance) (Latency and Throughput), Resource Behavior, Compliance, Scalability. Accessibility. |
| Maintainability | Analyzability, Changeability or Modifyability, Stability, Testability, Compliance. |
| Portability | Adaptability, Install-ability, Co-existence, Replace-ability, Regulatory. |
| Usability | Understandability, Learn-ability, Operability, Attractiveness, Compliance, Documentation. |

Table 1: QoS Attributes Classification

In the commercial area, Appirio Cloud Metric [1] is used to monitor and manage *Saleforce.com* by delivering reports. It helps customers to identify the most important factors impacting the production environment and let them focus on the performance enhancement. This kind of reports help clients detecting QoS breaches in their services and managing them accordingly. SAS Environment Manager [21] is used to track the performance of middle-tier component during run-time. Potential bottlenecks and breaches can be identified by collecting the relevant measurements. This requires to know more precisely *when, what, how and where* to collect relevant information. DevOps also Brought the notion of rapid deployment, configuration and scaling. In this area, AWS Fargate [8] helps customers to deploy applications using containers on AWS by leveraging containers instead of VM as a compute primitive. In this case a knowledge of the application is mandatory for well sizing each container image. Kubernetes [2] is another open-source framework used to manage centralized application for automated deployment, configuration and scaling. The containers are grouped in a logical units to build an application. Hence by considering a component as an application will increase drastically the number of containers. Finally, the state of the art also shows that the concept of QoS monitoring is omnipresent, and often focuses on either the middle-tier infrastructure or the whole system (in case of cloudified application). The more the monitoring accuracy (specific component), the stronger the SLA enforcement is.

## 3 Proposed Solution & Architecture

From this state of the art analysis, it appears that just monitoring the system as a whole, rather than also monitoring the specific constituent component of a system, does not lead to accurate SLA management. We argue that it is necessary to have a more fine grained monitoring of the system. Therefore, we propose a solution that aims to provide QoS modeling and its associated algorithm SLA

breach detection and assurance. We focus mainly on the response-time QoS in the performance category [19], as it is the most visible metric for the end user.

### 3.1 Solution Architecture and Design

The proposed solution is based on components monitoring by analyzing dynamically the monitoring data produced by these components. The monitoring data should respect specific protocol and structure that will be described later in this section. The goal of the monitoring data analysis is to survey response-time QoS (date-time metric). As specified in [12], a response-time is a minimum elapsed time in a service request. It is measured by date-time metric (often in milliseconds) and it may be expressed with an average or a min-max interval. In our case, the response-time of a component is the time used by a component to process a request from the time it received it to the time it sent the response. In a request chain, we will use the term *requester* for a component $C_i$ sending a request and the term *requested* for a component $C_j$ receiving a request. The idea behind is to trigger an alert whenever the component is not capable to process the request in the specified time and therefore exhibiting response-time exceeding a predefined threshold derived from the SLA. Two types of alert can be triggered:

- **Proactive Alert**: As an alert triggered by a component and serves to mitigate potential SLA violation.
- **Emergency Alert**: Is an alert triggered by the application and requires an immediate intervention since it will eventually break the SLA terms and lead to penalties for the provider.

**Parametrization and Hypothesis:** After a few long running performed tests and benchmarks on a real deployed service, we have measured the average response-time thresholds to apply service components. In addition, we have observed that useful alerts happen after three defects of a component to respond within an acceptable threshold. Thus, if a component triggers three consecutive alerts in the same hour, it is reasonable to consider it as a component violation (not a SLA violation). We derive then the following conclusions for a customer request on a service that involve n components:

- $RT_{max} = \sum_{i=0}^{n} RT_{max}(i)$ Pessimistic response-time for n components
- $RT_{min} = \sum_{i=0}^{n} RT_{min}(i)$ Optimistic response-time for n components
- $RT_{avg} = \sum_{i=0}^{n} RT_{avg}(i)$ Average response-time for n components
- $Alert_{max}$ Number of occurrences before triggering an emergency alert set to 3. Each component $C_i$ has its maximum alert occurrence $Alert(i)_{max}$

**Response-Time Model:** A set of components are deployed and logically connected together to provide an end-to-end service. Each triggered component receives a request, performs specific computation and processing on the request and finally delivers a response to the requester component or another component in the component chain. A response-time for a request $RT_{req}$ is tightly bound with all involved components in the request chain. Thus, the final response-time $RT_{req}$ can be calculated as follows:

$$RT_{req} = \begin{cases} \sum_{i=0}^{n} T(i), & \text{n+1 involved components} \\ where, & \\ RT_{req} \leq RT_{max}, & \text{RT QoS in SLA} \\ T(i) = T_{out}(i) - T_{in}(i), & \text{RT of the } i^{th} \text{ component} \end{cases}$$

**Monitoring Data Structure:** The monitoring tool requires a specific data structure. We used JSON format [14] to represent it and REST API [23] render it as these two technologies are widely used today. The data structure is depicted in the Table 2.

| Data Type | Attribute | Description |
|---|---|---|
| Identification | message-Id | Message Id identifier across the request. |
| | component-Id | Component Id identifier across the application. |
| | instance-Id | Container instance Id identifier in the VM. |
| Measurements | time-stamps | Operating System timestamps in milliseconds. |
| | start-time | Represents a component request $T(i)_{in}$ in seconds. |
| | end-time | Represents a component response $T(i)_{out}$ in seconds. |

Table 2: Monitoring Data Structure

### 3.2 SLA Enforcement Algorithm

The algorithm we propose is based on two major events that could trigger two types of alerts. The first type of alert (*Proactive*) is used to alert about a particular component performance that may cause a SLA violation if not mitigated. The second type (*Emergency*) is for immediate intervention because it eventually causes SLA violation. Each time a performance violation is detected for a particular component, the *Alert* occurrence is incremented. When a maximum allowed number of alerts $Alert(i)_{max}$ is reached within the same hour for a particular component, a proactive alert is sent. If the $RT_{req}$ is not met and a maximum number of alerts $Alert_{max}$ is reached within the same hour then an emergency alert is triggered. Emergency alert should satisfy these conditions:

$$Emergency \begin{cases} RT_{req} > RT_{max} & \text{RT exceeds the limit.} \\ and & \\ Alert_{max} = 3 & \text{Maximum number of QoS violation during the same hour.} \end{cases}$$

A system is reset if one hour has elapsed or an emergency alert has been triggered.

### 3.3 SLA Modeling

The corresponding SLA model is specified using OCCI Service Level Agreement [16]. OCCI is used both for SLA specification as well as for modeling the monitoring tasks [15, 5]. The proposed SLA is divided into two major parts. The first part is related to services specification (e.g. Application Components) and the second part is related to the modeling of the QoS objectives or SLO (or Service Level Objectives). In our case, the SLO are related to services performances (e.g. response-time). The Figure 1 depicts the resulting Service Level Agreement model for one customer application (user service). The **QoSSLA** is

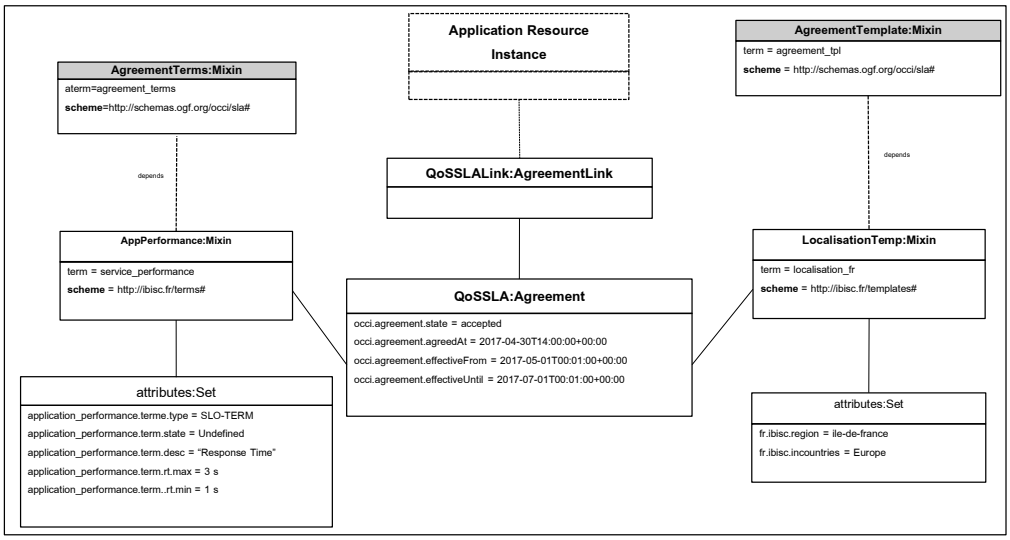


Fig. 1: Application OCCI SLA Model

the agreement related to the application resource and is linked to it using the **QoSSLALink** link. The QoS response-time is represented by the **AppPerformance** Mixin (a concept in OCCI that allows to add properties dinamically to a particular concept) with its OCCI AgreementTerms Mixin and set of attributes including the response-time limit condition. A **LocalisationTemp** Mixin is also added using OCCI AgreementTemplate Mixin with its attributes set. The second model is for the monitoring representation as depicted in the Figure 2. The **AppMonitoring** resource is linked with the monitored application using the **AppMonitoringLink** link. The **AppMonitoringData** Mixin represents the performance condition while the **AppMonitoringCollector** Mixin defines the

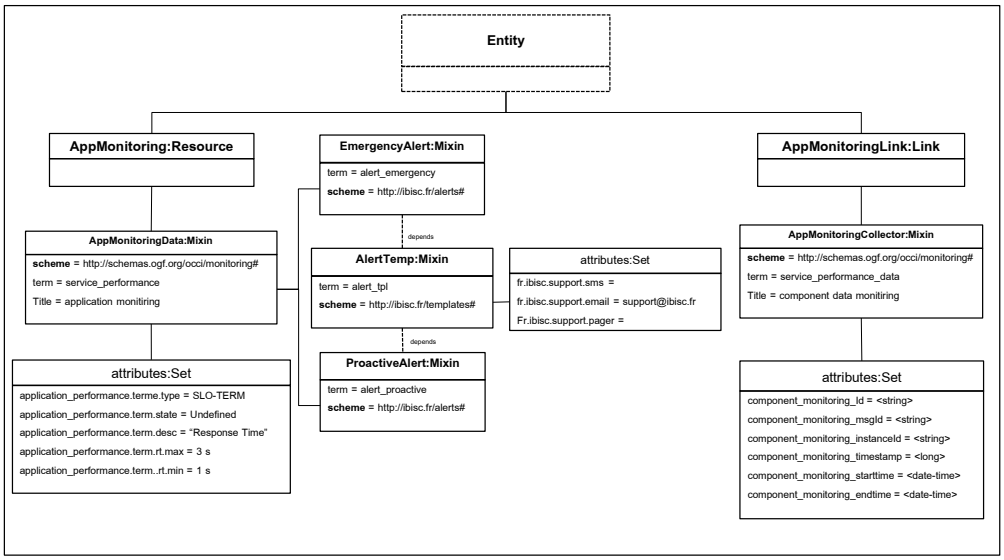


Fig. 2: Monitoring OCCI Model

collected data. A set of alert Mixin is defined (**EmergencyAlert** and **ProactiveAlert**) based on the **AlertTemp** template Mixin. Please note that we did not represent the Decision Agent and the Notification Agent as depicted in the architecture diagram from the Figure 3.

## 4 Implementation

We have implemented a proof of concept simulation. It consists of an environment hosting two VM(s) with different resources capacity: $VM_1(CPU : 1, RAM : 3GB, Disk : 30GB)$ and $VM_2(CPU : 1, RAM : 2.5GB, Disk : 20GB)$. We configured each VM with the necessary tools and software to accurately monitor the middleware and executing the application. We have installed a J2EE based application server that serves as an application container and we added a MDC (*Monitoring Data Collector*) and a DNA (*Decision and Notification Agents*). The two VM(s) are faced by a local LB (*Load Balancer*) in a local network as illustrated in Figure 3. We have specified our complex and composite application as a set of interacting components. In order to facilitate our simulation, the MDC tool collects monitoring data from each component and builds monitoring metrics. This data is then exposed through a REST API in a JSON format. When we ran the performance tests, we faced an issue with the CPU consumption that impacted the overall response-time. After running an RCA (*Root Cause Analysis*), we discovered that the issue was caused by the increasing number of REST calls. By tuning the time between two consecutive

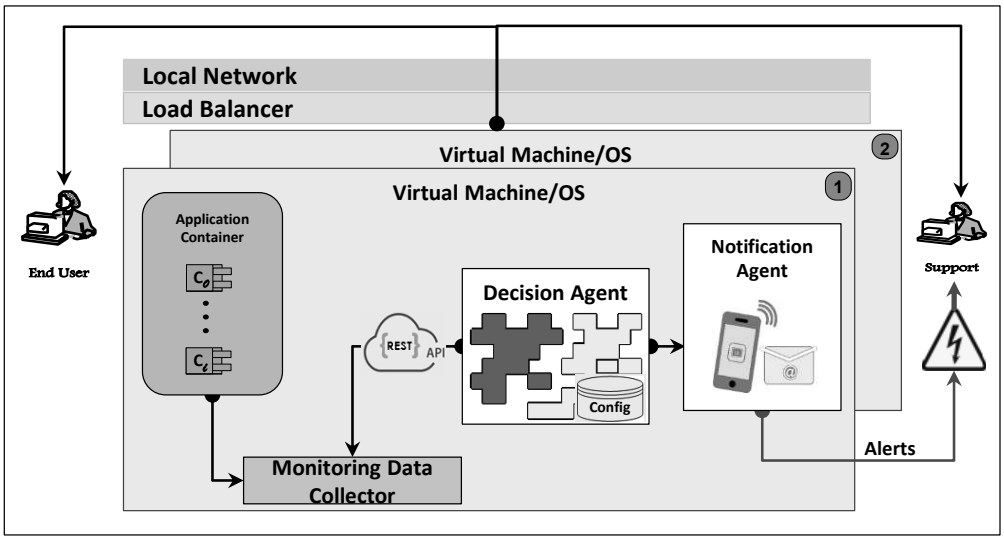


Fig. 3: Global Architecture Design

calls, we reached an acceptable compromise between the CPU usage and the number of monitoring calls as illustrated in Figure 4 and Figure 5. This issue is pinpointed in the SLAMOM project [19] where they recommended a minimum time interval between two monitoring requests in order not to overload the components (around 1 minute). After executing our simulation scenario, we received

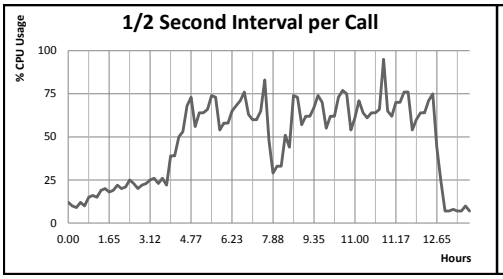


Fig. 4: CPU Usage (1)

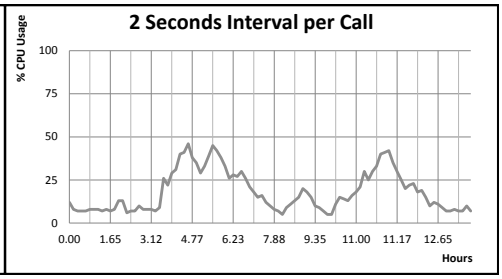


Fig. 5: CPU Usage (2)

twenty-six alerts (twenty-five proactive and one emergency). After analyzing the alerts, we discovered that the emergency alert happened after the same component caused four proactive alerts as depicted in Figure 6 and Figure 7.

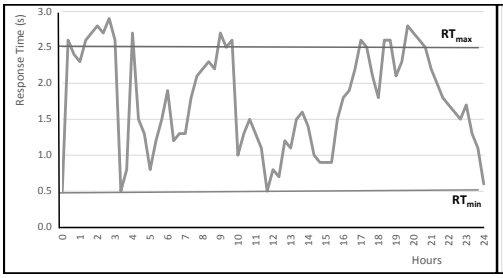


Fig. 6: Response Time Result

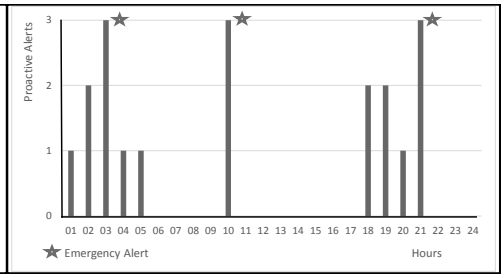


Fig. 7: Alerts Result

## 5 Conclusions and Future Works

Assuring SLA in Cloud Computing environment is very important for its adoption. However, the QoS metrics that are defined by Cloud Providers are heterogeneous, domain dependent and more seriously lacking the same semantics from one application to another. Thus, if we exclude traditional hardware monitoring and tuning, it is difficult to have common tools to manage these metrics in an efficient way. Cloud applications are used to build a set of components that are logically connected together. It is important to monitor each component individually and understand how the performance violation of each component may impact the service as a whole. It is also important to avoid performing unnecessary reconfiguration or scaling on these components and to refine the right thresholds and failure occurrence before any action. In this work, we have proposed an approach to optimize and adapt these values based on peak hour and normal hours to minimize resources usage and enhance the overall system performance. In the future, we would like to leverage this approach to other QoS, improve the provisioning process and scaling mechanism using Container (e.g. Docker) technology and associated monitoring mechanisms. In this scenario scaling containers instead of VM may permit to better optimize resource utilization by modeling a set of components as an application.

## Acknowledgments

This research is partially funded by the French Ministry of Foreign Affairs and International Development (MAEDI) in the frame of the AmSuD-VNET project (16STIC11-VNET). Also by the Industry and Innovation department of the Aragonese Government and European Social Funds (COSMOS group, ref. T93) and the Spanish Ministry of Economy (TIN2013-40809-R). Thanks to all the partners of the project who have helped with their discussions to improve the research work presented in this paper.

## References


1. Appirio. Cloud Metrics for Salesforce.com. Technical report, Appirio, Inc., 760 Market St. 11th Floor, San Francisco, CA 94102, 2012.
2. The Kubernetes Authors. Production-grade container orchestration, 2017.
3. A.K. Bardsiri and S.M. Hashemi. QoS Metrics for Cloud Computing Services Evaluation. *I.J. Intelligent Systems and Applications, MECS*, pages 27–33, December 2014. doi:10.5815/ijisa.2014.12.04.
4. Ch. Braga, F. Chalub, and A. Sztajnberg. A Formal Semantics for a Quality of Service Contract Language. *ELSEVIER, Electronic Notes in Theoretical Computer Science 203 (2009)*, pages 103–120, 2009.
5. A. Ciuffoletti. A simple generic interface for a cloud monitoring service. *CLOSER 2014 - 4th International Conference on Cloud Computing and Services Science, April 2-5 2014 (Barcelona)*, April 2014.
6. G. Dobson, R. Lock, and I. Sommerville. QoSOnt: A QoS Ontology for Service-Centric Systems. *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 80–87, September 2005. doi:10.1109/EUROMICRO.2005.49.
7. S. Hanna and A. Alawneh. An Approach of Web Service Quality Attributes Specification. *IBIMA Publishing, Communications of the IBIMA Journal (ISSN:1943-7765)*, 2010:1–13, January 2010. doi:10.5171/2010.552843.
8. R. Hunt. Introducing aws fargate run containers without managing infrastructure, November 2017.
9. A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems*, 22:931–945, June 2011. 10.1109/TPDS.2011.66.
10. C. Irvine and T. Levin. Toward a Taxonomy and Costing Method for Security Services. *Proc. ACSAC99, Phoenix AZ*, December 1999.
11. C. Irvine and T. Levin. Quality of Security Service. *NSPW - Proceedings of the workshop on New security paradigms*, pages 91–99, 2001.
12. J. McKendrick. Survey: Public Cloud Metrics Are Still Too... Cloudy. http://www.forbes.com/forbes/welcome/3db1de661194, December 2015.
13. J. Jin and K. Nahrstedt. QoS Specification Languages for Distributed Multimedia Applications: A Survey and Taxonomy. *IEEE Computer Society, IEEE MultiMedia, ISSN :1070-986X*, 11:74–87, July 2004. doi:10.1109/MMUL.2004.16.
14. JSON. The home of JSON Schema. Json-schema Organization (2017). http://json-schema.org/documentation.html, 2017.
15. M. Mohamed, Dj. Belaid, and S. Tata. Extending OCCI for autonomic management in the cloud. http://elsevier.com/locate/jss, 2016.
16. OGF. GFD228-Open Cloud Computing Interface Service Level Agreements. http://occi-wg.org/2016/10/, October 2016.
17. Opengroup. The Open Group, Building Return on Investment from Cloud Computing : Cloud Computing Key Performance Indicators and Metrics. www.opengroup.org/cloud/whitepapers/ccroi/kpis.htm, June 2016.
18. P. Pritzker and W. May. Cloud Computing Service Metrics Description. *National Institute of Standards and Technology, Special Publication 500-307*, 2015.
19. Slalom Project. SLA specification and reference model-c-D3.6. Technical report, Institute of Communication and Computer Systems and other members of the SLALOM consortium, 2016, ICCS 9, Iroon. Polytechniou Str., 157 73 Zografou, Greece, 2016.

20. M. Selva, L. Morel, K. Marquet, and S. Frenot. A QoS Monitoring System for Dataflow Programs. *ComPAS2013 : RenPar21/ SympA15/ CFSE9 - Grenoble, France*, January 2013. HAL Id: hal-00780976.
21. R. Sioss. SAS 9.4, Web Application Performance: Monitoring, Tuning, Scaling, and Troubleshooting. *Proceedings 2014, SAS Global Forum*, March 2014.
22. Wikipedia. Quality of service. https://en.wikipedia.org/wiki/Quality_of_service/, August 2016.
23. Wikipedia. Representational state transfer. https://en.wikipedia.org/wiki/Representational_state_transfer/, September 2017.