MULTI-LAYER QUALITY-AWARE (MULQA) CLOUD FRAMEWORK

by

Arash MORATTAB

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT FOR A MASTER'S DEGREE
WITH THESIS IN SOFTWARE ENGINEERING
M.A.Sc.

MONTREAL, "JUNE 9, 2017"

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mr. Mohamed Cheriet, Memorandum Supervisor
Department of Automated Manufacturing Engineering at École de technologie supérieure

Mr. Chamseddine Talhi, President of the Board of Examiners
Department of Software and IT Engineering at École de technologie supérieure

Mr. Abdelouahed Gherbi, Member of the Jury
Department of Software and IT Engineering at École de technologie supérieure

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON "MAY 17 2017"

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. Mohamed Cheriet for the continuous support of my Master study and related research, for his patience, motivation, and immense guidance. He was always available and helped me all the time in this two year journey.

Besides my advisor, I would like to thank Prof. Kim Khoa Nguyen, our team lead in Synchromedia lab for giving me insights and guidance during my research and reviewing this document. Also I want to thank my friend, Dr. Shahin Vakilinia for sharing his valuable knowledge and helping me significantly to improve this research technically.

Next, I would like to thank the rest of my thesis committee: Prof. Chamseddine Talhi and Prof. Abdelouahed Gherbi, for their insightful comments and encouragement, helping me improve my research from various perspectives.

Last but not the least, I would like to thank my first teachers in life, my parents for supporting me since childhood and allowing me to realize my own potential.

MULQA: UN CADRE DE NUAGE DE QUALITÉ À PLUSIEURS COUCHES

Arash MORATTAB

RÉSUMÉ

Pendant les dernières années, des solutions infonuagiques dans le domaine TI ont augmenté significativement à cause du changement de l'industrie IoT, des réseaux à haute vitesse et notamment les avantages émergents d'informatique en nuage. Cependant, ceci présente plusieurs défis techniques comme l'optimisation de l'infrastructure pour des applications hétérogènes particulièrement celles qui sont sensibles à la qualité, et de fournir simultanément des attributs de qualité différents. Dans cette recherche, nous proposons MULQA, une plateforme logicielle autonome qui contrôle et estime les métriques de qualité dans les couches physique, d'infrastructure, de la plate-forme et logiciels d'un système de logiciel libre nuagique et assure la qualité de la métrique ciblée en déclenchant des actions appropriées. MULQA est une nouvelle approche fournissant différents niveaux de qualité dans toutes les couches du nuage.

Dans ce mémoire, nous décrivons la conception de MULQA où le module d'analyse, prédit la violation de la métrique de qualité et ces prédictions seront utilisées pour créer des événements pour l'automate fini de la plate-forme de planification. Ce mécanisme de contrôle consiste en des états Normal, Alarme et Transition. L'état Alarme est utilisé pour préparer le nuage pour l'état Transition, tandis que l'état Transition empêche les violations et ramène le système à l'état Normal. Étant une plateforme modulaire MULQA fournit des fonctionnalités génériques et les modules qui peuvent être personnalisés par des programmes d'utilisateur, qui peut être utilisé pour tester des algorithmes proposés pour les modules Moniteur, Analyser, Planificateur et Exécuteur. MULQA est conçu pour surmonter les défis dans la mise en oeuvre d'un système de couplage mou qui peut être facilement distribué et personnalisé par une API. En outre, la plateforme est compatible avec l'architecture Openstack et peut surveiller et contrôler les composants que ce intergiciel de nuage n'a pas d'accès.

Le cas d'étude présenté dans ce mémoire est une application Web à trois niveaux qui est déployée avec Openstack. Les résultats expérimentaux des tests qui se concentrent sur la performance QA (Quality Attribute) montrent que MULQA peut augmenter le taux de réussite de requêtes de 32%, 69% et 94% pour le nombre de requêtes concurrentielles de 200, 500 et 1000. De plus, le débit a été améliorée de cinq fois, avec un faible impact sur l'utilisation de CPU.

Mots clés: Informatique en Nuage, Gestion de la Qualité, QoS, Système Autonome, Openstack

MULTI-LAYER QUALITY-AWARE (MULQA) CLOUD FRAMEWORK

Arash MORATTAB

ABSTRACT

In the past few years, the popularity of cloud-based solutions in the IT domain has been increased significantly as the consequence of the industry shift towards IoT, super-fast computer networks and notably the benefits of emerged cloud computing. However, this leads to many technical challenges such as optimizing the infrastructure for heterogeneous applications especially the quality sensitive types, and issues toward addressing different quality attributes simultaneously. In this research, we propose MULQA, an autonomic framework that monitors and estimates the quality metrics in physical, infrastructure, platform and software layers of an open source cloud system, and ensures the quality of the targeted metrics by triggering appropriate actions. MULQA is a novel approach providing such framework which targets different quality metrics in all layers of the cloud.

During this thesis, we describe MULQA framework where the analyze module, predicts the violation status of the quality metrics and this predicted information will be used to create events for the finite state machine of the planning platform. This control mechanism consists of Normal, Warning and Transition states. Warning state is used to prepare the cloud for the transition state, while transition state prevents the violations and brings back the system to the normal state. Being a modular framework, MULQA provides generic functionalities and modules that can be selectively changed by additional user-written code, which can be used to test proposed algorithms for Monitor, Analyze, Plan and Execute modules. MULQA framework is built to overcome the challenges in providing a loosely coupled system which can be easily distributed and customized through an API. Furthermore, this framework is compatible with Openstack architecture and is able to monitor and control the components that the cloud middleware doesn't have access to.

The use-case in this thesis, is a three-tier Web application which is deployed with Openstack. Experimental results of the tests which focus on the performance QA, show that MULQA can increase the success rate of requests sent by 32%, 69% and 94% for request concurrency numbers of 200, 500 and 1000 in order. Moreover, throughput has been improved five times with low impact on the CPU utilization.

Keywords: Cloud Computing, Quality Management, QoS Aware, Autonomic System, Openstack

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ABREVIATIONS

ACCS	Autonomic Cloud Computing System
AE	Autonomic Element
AM	Autonomic Manager
API	Application Programming Interface
AWS	Amazon Web Services
CLI	Command-Line Interface
CMMI	Capability Maturity Model Integration
CMS	Content Management System
CPU	Central Processing Unit
DVFS	Dynamic Voltage and Frequency Scaling
E2E	End to End
ETS	École de Technologie Supérieure
FSM	Finite State Machine
HPC	High Performance Computing
IaaS	Infrastructure as a Service
ISO	International Organization for Standardization
IT	Information Technology
KVM	Kernel-based Virtual Machine
LAMP	Linux Apache MySQL PHP

LBaaS	Load-Balancer as a Service
MULQA	Multi-Layer Quality-Aware
NIST	National Institute of Standards and Technology
OS	Operating System
PaaS	Platform as a Service
QA	Quality Attribute
QoS	Quality of Service
RAID	Redundant Array of Independent Disks
RAM	Random-Access Memory
RDBMS	Relational Database Management System
ROI	Return On Investment
SaaS	Software as a Service
SDN	Software Defined Networking
SLA	Service-Level Agreement
SMI	Service Measurement Index
SOA	Service Oriented Architecture
SSD	Solid-State Drives
VM	Virtual Machine
VMM	Virtual Machine Monitor
YAML	YAML Ain't Markup Language

LISTE OF SYMBOLS AND UNITS OF MEASUREMENTS

b	Bit
cr	Concurrent requests
GB	Giga-Byte
mips	Millions of instructions per second
mps	Messages processed per second
rps	Requests processed per second
s	Second
tps	Transactions per second

CHAPTER 1

INTRODUCTION

Context and Motivation

Nowadays popularity of the cloud computing model in the IT domain plays a significant role in application design by the developers, and enterprise operation by IT managers. These considerations in planning and operation, help to bring the numerous benefits of cloud computing to the users' side. According to National Institute of Standards and Technology (NIST) in Mell and Grance (2011), "it enables ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." The mentioned characteristics and advantages of cloud computing idea are entirely consistent with the goals of upcoming IT trends and hence, make it one of the most important supporting technologies. Regarding the levels of service that can be delivered by cloud systems, we can consider three categories: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). However, providing these cloud services that ensure user's dynamic quality requirements and avoid Service Level Agreement (SLA) violations, is a big challenge in cloud computing. To improve the quality of the cloud applications, one could be mistaken by addressing this issue in isolation and without considering it in all layers of the cloud used by the application. Another possible mistake is improving one quality attribute and neglecting the others and the required trade-off between them. For example, to increase the application availability we may need to run more servers which increases the energy consumption. Besides, most of the cloud-based applications are heterogeneous (meaning that they incorporate components with different technologies or from different vendors across various levels), so interoperability should be considered in the provided quality management solution. Moreover, some of these apps like e-commerce web applications are quality-sensitive, and failure to respond or insecure transactions, can impact customer satis-

faction and sales. Furthermore, targeting the quality of just customers or just providers and neglecting the other system stakeholders is another common mistake in improving quality.

In this thesis we propose an autonomous cloud framework named MULQA after Multi-Layer Quality-Aware, to improve the quality of cloud systems. Quality attributes targeted by MULQA can be comprised of performance, availability, reliability, cost, energy-efficiency, etc. Also, quality metrics for a targeted quality attribute can be defined, modeled, monitored, predicted and controlled in all layers of the cloud, including physical, infrastructure, platform and software layer.

In this research, the quality awareness of the cloud systems is addressed, a framework to improve the quality management is built and a test scenario of the scheme in web applications is implemented. The key contributions of this thesis are twofold:

- Propose a framework for handling the quality definition, monitoring, and control for open source cloud computing systems. The proposed solution is named MULQA which is a Multi-Layer Quality-Aware cloud system.
- A real testbed in the framework is designed, and a real scenario of a web application (Wordpress) is implemented to evaluate the performance of the proposed quality management system. In the implementation, OpenStack, the most popular private cloud middleware, is used. Results show that using MULQA, can improve the quality of the use-case scenario significantly.

Problem Statement

As mentioned earlier, a diverse set of softwares can coexist in a cloud system which makes quality management difficult. These softwares may include operating systems like Ubuntu, Redhat Enterprise, and Windows, cloud middlewares like Openstack and Cloudstack, runtimes and compilers, databases, IoT Apps, web servers, monitoring and logging systems, High-Performance Computing (HPC) applications, etc. Heterogeneity in applications leads to di-

vergence in different quality metrics. In the following examples for instance, when a banking application user wants to transfer money to another account, the security of the system has a higher importance than the system's response time. So, security metrics are defined and targeted in the quality management. Secondly, when a user subscribes to an HD video channel, he may expect the video to be smooth and high quality for a higher price. And thirdly, when the cloud user is a PHP developer who runs applications on a LAMP (Linux Apache MySQL PHP) platform, the quality of handling 100 Threads/sec (i.e. a performance quality) on the platform is preferred, rather than increasing energy efficiency.

The current aforementioned quality management systems are dedicated to specific service models, mostly IaaS and SaaS. For instance, in IaaS management, CPU utilization or VM consolidation is improved to decrease the energy consumption and operational costs. Improving different quality attributes with such systems leads to multiple quality management systems at various layers of a cloud deployment, which may bring interoperability issues and cause inefficiency in monitoring and control mechanisms. In other words, management in cloud systems is not integrated to cover different QAs and all layers.

Addressing quality attributes in isolation and without considering it in all the service levels used by the application is wrong, and this research claims that to have full control on the quality there should be a system which considers quality in all layers of the cloud system including physical, infrastructure, platform and software.

Furthermore, our testbed's cloud middleware, OpenStack, at the time of writing this thesis does not have any component to guarantee and control the quality of the cloud. Ceilometer, which is the telemetry project of OpenStack collects data on the utilization of the physical and virtual resources comprising deployed clouds. It also persists these data for subsequent retrieval and analysis, and trigger actions when defined criteria are met. But Ceilometer is only a limited monitoring system which just targets some of the metrics in the physical and infrastructure layers, which is far from a complete solution to cover various QAs in different layers which is automated.

To improve the quality of a cloud system, some common actions are load balancing, VM migration, resource throttling, VM (instance) group scaling and "Dynamic Voltage and Frequency Scaling" (DVFS). The challenge is that triggering such actions may cause quality violations of other quality metrics in different layers of the cloud system. Also, designing a customizable and modular system which considers different actions regarding the state of the cloud is needed. In this thesis, we consider triggering these actions with consideration of all quality metrics in different layers. We claim that taking actions applying this extra quality information in different cloud layers results in higher quality cloud systems.

Research Questions

To conduct this research and find a solution to the mentioned problem statements, we need to answer the following research questions:

- Q1: Which quality attributes are important in cloud systems and how are they defined?
- Q2: What are the quality management challenges in cloud computing domain and how can the previous works be improved?
- Q3: How can we design an autonomic modular framework to integrate quality management in all layers and target different QAs?
- Q4: How to implement, deploy and evaluate the designed framework in a practical production use-case?
- Q5: How does our tested framework perform and how much improvement is made to the quality?

Objectives

To deal with the challenges associated with the research problems mentioned in Section 1, the following objectives are delineated:

- Obj1: Propose a quality model with notations and mathematical models for cloud systems and design an autonomic Multi-Layer Quality-Aware system to monitor, model, predict and control the different quality attributes in cloud systems.
- Obj2: Build a modular and customizable framework for the designed solution in Obj1, and design and implement a cloud-based three-tier web application that can be used to evaluate the proposed system design on a multi-node OpenStack testbed.

Thesis Organization

In this chapter, we explained the context of the problem domain and the motivations behind proposing MULQA. Next, we clarified the problem statements and asked the research questions followed by mentioning the objectives of this research; Finally in this section, the structure and organization of this thesis is described as the following.

Chapter 2 presents relevant state-of-the-art research and surveys important quality attributes in the cloud computing domain. This chapter starts with an introduction to the context and terms used in this thesis, varying from cloud characteristics and models to Openstack and autonomic cloud computing system. Next, quality with the focus on cloud systems is discussed, and challenges of quality management in cloud computing are surveyed. This chapter follows with QAs of cloud systems and their metrics, and finally reviews similar works related to MULQA.

In Chapter 3, MULQA, our proposed solution for making cloud systems quality-aware, will be described and explained in detail. This chapter first illustrates our model's notations for layers, quality attributes and metrics of the cloud. Next, it explains the finite state machine of the MULQA and the quality control mechanism. After it gives a general view of our model and then explains the multi-layer concept in our quality model. Then it describes our proposed approach for the SLA negotiation using fine-grained quality metrics.

Chapter 4 explains how MULQA has been implemented in a cloud system with OpenStack installed middleware and pictures the deployment architectures and describes the implemented

modules. Moreover, in this chapter the designed test scenario (which is a three-tiered Web application) for validating our solution will be described and the results of the experiments and the related discussion are provided.

Finally, the last chapter of this thesis concludes this document and gives the future proposals to improve this research.

CHAPTER 2

STATE OF THE ART

2.1 Introduction

To answer the research questions posed in Section 1, relevant researches are surveyed in this chapter. In the other words, quality has been surveyed and analyzed delving in the cloud computing domain.

This chapter first explains our domain, cloud computing, and its characteristics. It covers both service models and layers in cloud computing. Furthermore some other terms used in this research are explained, to help the reader understand the concepts discussed in the rest of this document.

Next, quality with the focus on cloud systems is discussed, and challenges of quality management in cloud computing are surveyed. This chapter follows with common QAs of cloud systems and their metrics, and challenges in formulating these metrics. Finally, this chapter briefly reviews the works similar to MULQA.

2.2 Context and Terms

2.2.1 Cloud characteristics and models

Based on National Institute of Standards and Technology (NIST) definition of cloud model, it is composed of five essential characteristics, three service models, and four deployment models. For sake of simplicity in this document, sometimes we refer to "cloud computing" shortly as "Cloud".

Essential characteristics of the cloud are: on-demand self-service, broad network access, resource pooling, rapid elasticity and measured service. On-demand self-service means that a

cloud consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider. Broad network access feature implies that cloud users can access capabilities over the network through standard mechanisms that promote use by heterogeneous thin or thick client platforms. In the cloud model, computing resources of the provider are pooled to serve multiple consumers using a multi-tenant model. This is done by assigning and reassigning different physical and virtual resources dynamically according to consumer demand. Rapid elasticity means Cloud is capable of provision and release the resources, to scale rapidly outward and inward commensurate with demand. The key to creating a pool of resources is to provide an abstraction mechanism so that a logical address can be mapped to a physical resource. Virtualization refers to the act of creating a virtual version of something, including virtual computer hardware platforms, computer network resources, operating systems and storage devices. Services in cloud are typically measured in pay-per-use basis. Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service. Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service. According to Bittman (2009) virtualization is a key enabler of the key attributes of cloud computing mentioned as: service-based, scalable and elastic, shared services, metered usage and Internet delivery.

When enterprise architects and network planners want to plan cloud computing deployments, they need to be able to identify the expectations for control and management, based on the type of cloud and its level categorization. One categorization can be based on kind of the service which cloud provides, such as Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). Another classification can be based on the models for cloud computing deployment, which relate to strategies for extending virtualization outside of the data center into the cloud. Deployment models include: public cloud, private Cloud, hybrid cloud and community cloud.

Each cloud service model provides a level of abstraction that diminishes the efforts required by the service consumer to build and deploy systems. Unlike cloud, in a traditional on-premises data center, the IT team has to build and manage everything. Figure 2.1 shows these models and the layers of the cloud which is managed by cloud consumers.

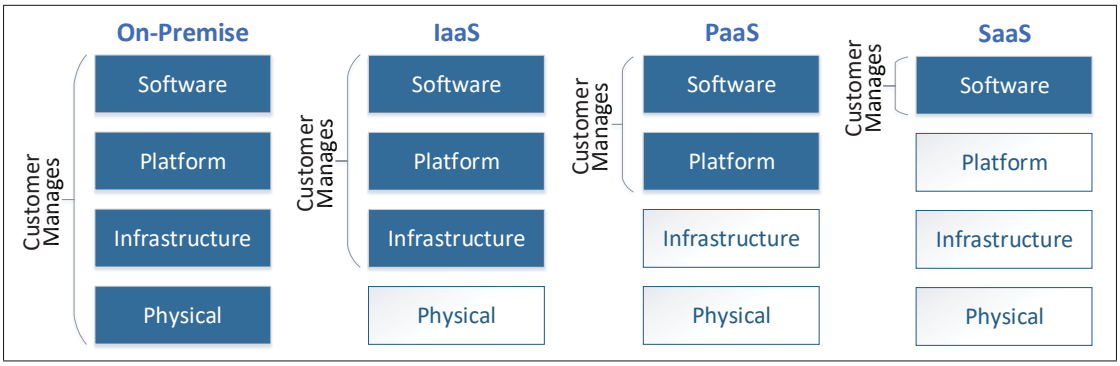


Figure 2.1 Cloud layers and the customer responsibility in the service models of the cloud

Note that layers in MULQA are an abstraction for the level of components which exist in the cloud system and they are different from the kind of service levels obtained from cloud systems such as SaaS, PaaS and IaaS.

In IaaS, the capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The most mature and widely used public IaaS cloud service provider is Amazon Web Services (AWS) (Kavis (2014)). On the other hand, for providing IaaS in private clouds, currently Openstack is the well-known solution as mentioned in Jiang (2015).

PaaS service model provides the capability to the consumer to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. Some of the services that can be found

in most mature PaaS solutions are: database, logging, monitoring, security, caching, search, E-mail.

SaaS provides the capability to the consumer to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. In this model, the underlying cloud infrastructure is not controlled by the consumer.

2.2.2 Service-Level Agreement (SLA)

As defined in Hausman *et al.* (2013), a Service-Level Agreement (SLA) is an official commitment that prevails between a service provider and the customer and summarizes the expected level of service a customer can expect to receive from a service provider, the metrics used to measure said service, and the roles and responsibilities of both the service provider and the customer.

According to Baset (2012), a typical SLA of a cloud provider has the following components: service guarantee, service guarantee time period, service guarantee granularity, service guarantee exclusions, service credit and Service violation measurement and reporting.

2.2.3 Hypervisor

As stated in Marinescu (2013), a hypervisor , also known as Virtual Machine Monitor (VMM) is the software that securely partitions the resources of a computer system into one or more virtual machines. An operating system that runs under the control of a hypervisor rather than directly on the hardware is called guest operating system. Unlike the hypervisor which runs in kernel mode, a guest OS runs in user mode.

VMMs allow several operating systems to run concurrently on a single hardware platform; Meanwhile, VMMs enforce isolation among these systems, which improves security. A hypervisor controls how the guest operating system uses the hardware resources. The events

occurring in one VM do not affect any other VM running under the same VMM. According to Marinescu (2013), at the same time, the VMM enables:

- Multiple services to share the same platform.
- The movement of a server from one platform to another, the so-called live migration.
- System modification while maintaining backward compatibility with the original system.

Hypervisor also monitors system performance and takes corrective action to avoid performance degradation.

According to Sosinsky (2010), there are two major types of hypervisor: Type I and Type II. A hypervisor running on bare metal is a type-1 VM or native VM. Examples of type-1 virtual machine monitors are LynxSecure, RTS Hypervisor, Oracle VM, Sun xVM Server, Virtual-Logix VLX, VMware ESX and ESXi, and Wind River VxWorks. On the other hand, some hypervisors are installed over an operating system and are referred to as type-2 or hosted VM. Examples of type-2 virtual machine monitors are containers, KVM, Microsoft Hyper V, Parallels Desktop for Mac, Wind River Simics, VMWare Fusion, Virtual Server 2005 R2, Xen, Windows Virtual PC, and VMware Workstation 6.0 and Server, among others.

2.2.4 Openstack

OpenStack is an open source project that provides IaaS capabilities for those consumers who want to avoid vendor lock-in and want the control to build their own IaaS capabilities in-house, which is referred to as a private cloud.

Beside Openstack, there are some other open source cloud platforms (aka cloud middlewares) like CloudStack, Eucalyptus, Nimbus, and OpenNebula. But regarding to Jain *et al.* (2014), OpenStack is the fastest growing free open source software in the cloud community. Openstack is now a global success and is developed and supported by thousands of people around the globe; backed by leading players in the cloud space today.

OpenStack official website defines it as a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface.

Many hypervisors are supported under the OpenStack framework, including XenServer/XCP, KVM, QEMU, LXC, ESXi, Hyper-V, BareMetal and others. Throughout this thesis, the Kernel-based Virtual Machine (KVM) will be used. KVM has been part of the Linux kernel since the 2.6.20 release in early 2007, and it's fully supported by OpenStack.

OpenStack has a modular architecture with various code names for its components. Core components of Openstack are: Nova, Keystone, Cinder, Swift, Neutron and Glance. Table 2.1 lists several of OpenStack's components. There are many more projects in various stages of development, but these are the foundational components of OpenStack.

Table 2.1 Openstack main projects

Project	**Code Name**	**Description**
Compute	Nova	Manages VM resources, including CPU, memory, disk, and network interfaces.
Networking	Neutron	Provides resources used by the VM network interface, including IP addressing, routing, and software-defined networking (SDN).
Object Storage	Swift	Provides object-level storage, accessible via a RESTful API.
Block Storage	Cinder	Provides block-level (traditional disk) storage to VMs.
Identity	Keystone	Manages role-based access control (RBAC) for OpenStack components. Provides authorization services.
Image Service	Glance	Manages VM disk images. Provides image delivery to VMs and snapshot (backup) services.
Dashboard	Horizon	Provides a web-based GUI for working with OpenStack.
Telemetry	Ceilometer	Provides collection for metering and monitoring OpenStack components.
Orchestration	Heat	Provides template-based cloud application orchestration for OpenStack environments.

2.2.5 Autonomic Cloud Computing System (ACCS)

Self- or Autonomic Cloud Computing Systems (ACCS), are kind of cloud systems that the services provided through them, are able to self-manage themselves as per their environment's needs without the involvement of humans.

Autonomic systems based on quality parameters are inspired by biological systems like Autonomic Nervous System that can handle situations like uncertainty, heterogeneity, dynamism and faults easily. ACCSs sense, monitor, and react based on the situations, such as self-healing, self-protecting, self-configuring, and self-optimizing.

According to Singh and Chana (2016), ACCSs are based on Computing *et al.* (2006), an IBM's autonomic model, which considers four steps of the autonomic system (Monitor, Analyze, Plan, and Execute) in a control loop, two interfaces (sensors and effectors) for environmental interaction, and one database (knowledge base) to store rules, as shown in Figure 2.2.

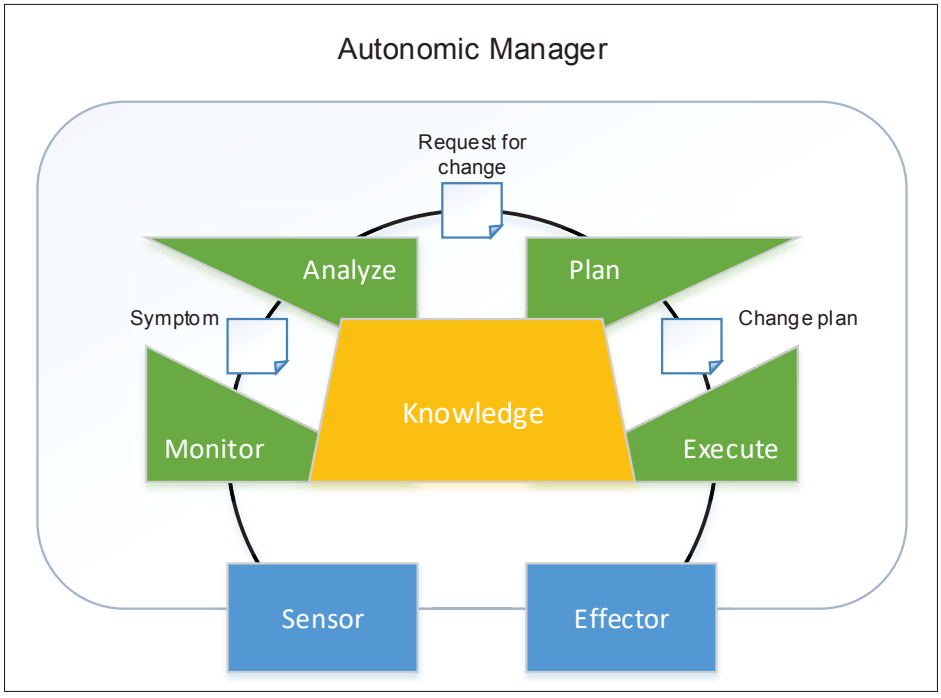


Figure 2.2 Architecture of an autonomic system

This figure illustrates Autonomic Elements (AEs) and Autonomic Manager (AM) in ACCSs and the workflow of an autonomous system; AM interacts with the environment through Sensors and Effectors interfaces to manage the system intelligently. Actions take place based on the input received from sensors and rules defined in a knowledge base. The administrator configures the AM based on alerts and actions.

2.3 Quality and Cloud Computing

2.3.1 Quality definition and views

As mentioned in Jones and Bonsignour (2012), quality can be defined in different ways, including in the context of software engineering. There are definitions of quality as given by national and international standards, however there is no 'ideal' definition of quality. In fact, there can be different viewpoints of quality: conformance viewpoint, human viewpoint and negative viewpoint.

The conformance viewpoint of quality as is in ISO (2011) is independent of the subject and relatively abstract: The totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs. In human viewpoint quality as defined in ISO (2010), quality definition is specific to a collection of subjects and relatively concrete: The ability of a product, service, system, component, or process to meet customer or user needs, expectations, or requirements. In the negative viewpoint, as stated in Emam (2005), the quality of a software system can be expressed in terms defect density, which is number of defects discovered per module size (Rico (2004)). In other words, the higher the defect density, the lower the quality.

In Garvin (1984) which is one of the earliest approaches towards perceptions of quality, five views of quality are given, as elaborated in Table 2.2. These views are not necessarily exclusive. For example, an economics-based view constrains the transcendental-based view of quality. Therefore, any initiatives for quality assurance or evaluation need to end if the cost exceeds the benefit.

Table 2.2 Different views of quality and their understanding

Type of View	Understanding of Quality
Transcendental-Based	Quality is Perfective
Product-Based View	Quality is Measurable
Manufacturing-Based View	Quality is Conformance
Economics-Based View	Quality is Benefit for Cost
User-Based View	Quality is Satisfaction

In addition, each of these views of qualities have their own issues. For example, in product-based view, the main issue is that of quantifiability. There are quality attributes, especially pertaining to human-machine interaction, which may not be quantifiable. For example, it is difficult to quantify comfortableness or satisfaction. In an economics-based view of quality, the notion of quality is benefit for cost. The main issue in this view is that of complacency. If sales of a software system are meeting an organization's target, then initiatives towards improving quality may subside.

2.3.2 Quality model

Following the definition of quality, in ISO (2007), quality model is as a defined set of characteristics, and of relationships between them, which provides a framework for specifying quality requirements and evaluating quality. Also Deissenboeck *et al.* (2009) has defined quality model as a model with the objective to describe, assess and/or predict quality. Regarding to Heston and Phifer (2011) some of the reasons for formulating a quality model are: awareness, motivation, consistency, repeatability and communication.

According to Ardagna *et al.* (2014), quality modeling discipline in cloud computing can be categorized to control theory based, machine learning based or operations research based (i.e. optimization, game theory, bio-inspired algorithms). Each of these approaches has its own usages and advantages. As mentioned in Padala *et al.* (2007), the advantage of control theory is guaranteeing the stability of the system upon workload changes by modeling the transient behavior and adjusting system configurations within a transitory period. Machine learning tech-

niques, rather, by utilizing learning mechanisms, can capture the behavior of the system with no explicit performance or traffic model. Furthermore, this approach needs minimal built-in system knowledge. However, according to Kephart *et al.* (2007), training sessions sometimes can take several hours and also for evolving workloads retraining is required. The goal of operations research approaches is optimizing the degree of user satisfaction which are expressed in terms of user-level quality metrics. Ordinarily, these approaches comprise of a performance model embedded within an optimization program, which is solved either locally, globally, or heuristically.

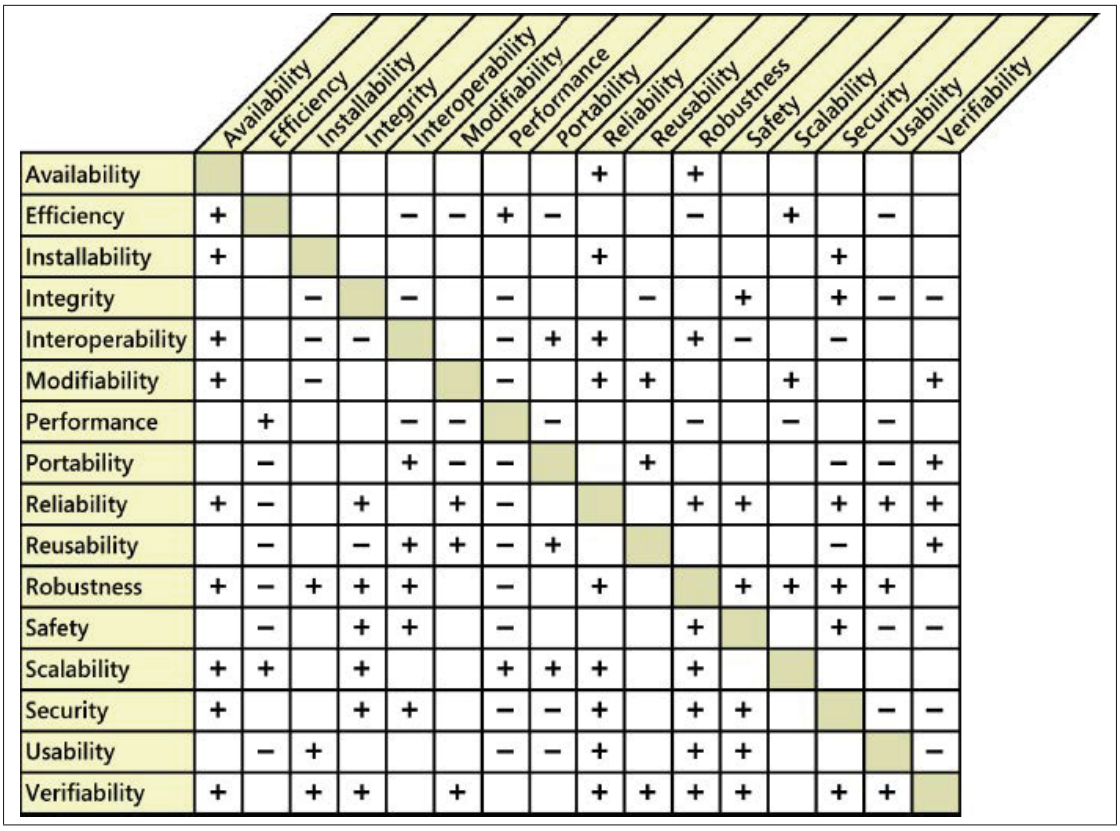
The main application of quality models is optimal decision-making for cloud system management. According to Ardagna *et al.* (2014), these decision problem areas include capacity allocation, load balancing, admission control, pricing, resource bidding, and provider-side energy management.

The notion of quality is decomposed into a number of Quality Attributes (QA) which can each be decomposed further. For example, maintainability, security, or usability, can be such a quality attribute. The quality attributes are a kind of concerns. As discussed in Cesare and Xiang (2012) a ‘standard’, closed-form, list of relevant quality attributes applicable to every system doesn’t exist. The relevancy of a quality attribute fluctuates with respect to the type of system. For example in section 1.2.3 of Suryn (2013), the most relevant quality attributes for "Network management systems" are specified as fault tolerance, interoperability, and operability; In the other hand for "Telecommunication systems", the most relevant quality attributes are functionality, reliability, usability, and efficiency; And for "Decision support systems", the most relevant quality attributes are accuracy, analyzability, and suitability.

2.3.3 Relationships between quality attributes

As mentioned in Chapter 14 of Wiegers and Beatty (2013), the quality attributes are not necessarily mutually exclusive. Indeed, they can affect each other in one of the following manner: positive (+), negative (-), or neutral (). Positive (+) relationship means that changing one at-

tribute affects the other positively. On the other side, Negative (-) relationship means that changing one attribute affects the other one negatively. And Neutral () relationship means that the attributes are independent of each other. These relationships among some quality attributes are shown in Figure 2.3.



	Availability	Efficiency	Installability	Integrity	Interoperability	Modifiability	Performance	Portability	Reliability	Reusability	Robustness	Safety	Scalability	Security	Usability	Verifiability
Availability								+		+						
Efficiency	+				−	−	+	−			−		+		−	
Installability	+								+					+		
Integrity			−		−		−			−		+		+	−	−
Interoperability	+		−	−			−	+	+		+	−		−		
Modifiability	+		−				−		+	+			+			+
Performance		+			−	−		−			−		−		−	
Portability		−			+	−	−			+				−	−	+
Reliability	+	−		+		+	−				+	+		+	+	+
Reusability		−		−	+	+	−	+						−		+
Robustness	+	−	+	+	+		−		+			+	+	+	+	
Safety		−		+	+		−				+			+	−	−
Scalability	+	+		+			+	+	+		+					
Security	+			+	+		−	−	+		+	+			−	−
Usability		−	+				−	−	+		+	+				−
Verifiability	+		+	+		+			+	+	+	+		+	+	

Figure 2.3 A concrete matrix of relationships between quality attributes.
Taken from: Wiegers and Beatty (2013)

As is shown in Figure 2.3, the main diagonal of the matrix would contain all empty spaces. The quality attribute matrix is not symmetric. For instance, efforts towards increasing security may decrease usability. However, the opposite is not necessarily the case.

As mentioned in Henningsson and Wohlin (2002), there are various results of the dependencies between quality attributes:

- Prioritization of quality requirements: The perceptions of quality by each stakeholder may be different than the others. Therefore, the importance of quality attributes may be different for each stakeholder. In result, the quality attributes are prioritized differently by different stakeholders.

- Assessment of the product: It is impractical to optimize all quality attributes simultaneously. This means some stakeholders are probably get disappointed. This can challenge or even fail the whole product, if these stakeholders are 'High' on the importance or influence scheme.

2.4 Quality Challenges in Cloud Computing

The root of the challenges in figuring a quality model has significant variations in context. Formulating a quality model generally presents some challenges rooted in the domain, stakeholders and artifacts. In theory, many quality models, including those in current international standards, aim to be general and abstract enough to be applicable for all sorts of systems. However, in practice, as discussed in Moody (2005) and Ruhe and Wohlin (2014), there are a number of challenges in applying these models sustainably. These challenges include: universality (Ruhe and Wohlin (2014)), need for tailoring and cost-effectiveness (Jagannathan *et al.* (2005))

In addition to these challenges related to formulating and using quality models mentioned previously, there exist some other cloud computing domain-specific challenges in quality topic. Many years before the advent of cloud computing, QoS and quality in computer systems have been studied and discussed. But quality analysis, prediction, and assurance in cloud platforms has got significantly complex due to performance heterogeneity and resource isolation mechanisms. According to Petcu *et al.* (2013), this is prompting several researchers to investigate automated quality management methods that can leverage the high programmability of hardware and software resources in the cloud.

Despite the fact that the cloud has enormously simplified the resource provisioning process, it represents several new challenges in the quality management. QoS in cloud context usually indicates the levels of performance, reliability, and availability offered by an application and by the platform or infrastructure that hosts it. QoS is crucial for both cloud providers, who need to find the right tradeoffs between QoS levels and operational costs, and for cloud users, who expect providers to deliver the advertised quality attributes. However, as mentioned in Ardagna *et al.* (2012), SLAs by specifying quality targets and economical penalties for SLA violations, increase the complexity of finding optimal tradeoff.

However, Quality as a general term in software engineering, can include the common QoS concepts, as well as cost and some other quality attributes such as energy-efficiency. On the other hand, instead of quality, QoS is used generally in network-specific context. Hence, this thesis mostly mentions Quality rather than QoS to generalize the quality concept.

According to Abdelmaboud *et al.* (2015), surveys on quality challenges in cloud computing can be classified in two categories: (1) Quality-aware software engineering challenges, (2) Quality-aware resource management challenges.

2.4.1 Quality-aware software engineering challenges

Many cloud computing challenges related to software engineering was discussed in Vázquez-Poletti *et al.* (2013). These challenges include: elasticity and provisioning of QoS for cloud application deployments, the lack of application management and the lack of approaches to cloud deployment optimization services with various quality metrics such as performance and cost.

Moreover, Yau and An (2011) has discussed the importance of combining the cloud computing and services paradigms and how a software engineering framework can help service providers to combine these paradigms. They contended that to address the challenges such as QoS management and security, more research was needed on software engineering for cloud computing. In addition, they discussed the main challenges and issues in application develop-

ment using service-oriented software engineering, such as service reliability and availability, confidentiality and integrity, and quality monitoring. As an example, in quality monitoring, managing different quality requirements is difficult, because in the cloud there exist multiple providers, and each of them needs different approaches to manage their services. Moreover different workflows required to host these different services dynamically.

2.4.2 Quality-aware resource management challenges

Challenges of cloud computing have already been surveyed by various researchers. Before Abdelmaboud *et al.* (2015), these researches reported on the QoS in cloud computing within limited scope. However, Abdelmaboud *et al.* (2015) performed a systematic mapping of QoS in cloud computing.

Foster *et al.* (2008) have compared grid and cloud computing from multiple aspects. They also discuss many challenges that will face cloud computing services in the future, such as cloud adoption, security issues, resources management, interoperability and the integration of services. Specifically, they mention the main challenges of resources management as: the monitoring of resources, the quality delivered to the users to locate or relocate the resources of applications, and the difficulty of achieving SLA requirements in terms of the cost effectiveness of systems provision. Similarly, the difference between quality in cloud computing and quality in grid computing was considered by Armstrong and Djemame (2009) which focused mostly on the performance and management of resources.

In addition, Dillon *et al.* (2010) mentioned the challenges of cloud computing in general. They looked into cloud adoption issues, for example security and the costing and charging model. Service providers should ensure the QoS (availability, reliability and performance) of the resources, because consumers of the cloud do not have control and access of the underlying cloud resources. Consequently, the issue of the SLA definition and specifications must be addressed in a suitable way that covers the consumers' expectations. Moreover, the SLA should include advanced mechanisms for user feedback. However, this study focused more specifi-

cally on interoperability issues. Similarly, Zhang *et al.* (2010) discussed the design challenges and state-of-the-art implementation of cloud computing, including server consolidation, automated service provisioning and virtual machine deployment. In this study, the challenges for service providers in achieving service-level objectives have been highlighted among issues related to automated service provisioning. For example, these SLA objectives can be quality requirements to allocate and de-allocate resources with minimum operational costs.

Finally, Buyya *et al.* (2009) gave a new vision of universal cloud exchange for commercial services and proposed the resources method of clouds from a market-oriented view. Also, they discussed the issues in cloud platforms such as the lack of negotiation between providers and users to fulfil SLAs, the limited support for resources management from a market-oriented perspective, the lack of models and limit mechanisms of the virtual machine resources allocated to meet SLAs, and the need to manage risks related to SLA violation. In addition, interoperability issues between various cloud service providers requiring interaction protocols were discussed. Furthermore, they identified the need for programming environments and tools to enable the development of cloud applications.

2.5 Quality Attributes of Cloud systems

As described in Gorton (2006), quality attribute (QA) requirements are part of an system's nonfunctional requirements, which capture the many facets of how the functional requirements of a system are achieved. All but the most trivial systems will have nonfunctional requirements that can be expressed in terms of quality attribute requirements. In the cloud computing context, quality attributes can be expressed in the SLA to ensure the quality of the desired service to the costumer.

As listed in Wikipedia (2016), system engineers have introduced more than 80 quality attributes with their definitions. But in-average among these, performance, availability, reliability, security, scalability, elasticity, interoperability, cost and energy were more important for the researchers who study in the field of cloud computing (considering Garg *et al.* (2013), Gorton

(2006), Lee *et al.* (2009), Mahdavi-Hezavehi *et al.* (2013), Nallur *et al.* (2009), Chang (2014), Sodhi and Prabhakar (2012)).

2.5.1 Performance

In the software engineering context, as described in Gorton (2006), a performance quality requirement defines a metric that expresses the amount of work an application must perform in a given time, and /or deadlines that must be met for correct operation. This QA is phenomenal for some application like avionics and robotic systems, which in those, if some output is produced a millisecond too late, undesirable things can happen. But applications needing to process hundreds, sometimes thousands and tens of thousands of transactions every second are found in many large organizations, especially in the worlds of finance, telecommunications and government.

Performance can mean different things in different contexts. As mentioned in O'Brien *et al.* (2007), in Service Oriented Architectures (SOA) generally, it is related to response time which is how long does it take to process a request, throughput which is how many requests overall can be processed per unit of time, or timeliness which indicates the ability to meet deadlines (i.e. to process a request in a deterministic and acceptable amount of time).

Response time is most often associated with the time an application takes to respond to some input. A rapid response time allows users to work more effectively. Also, it's often important to distinguish between guaranteed and average response times.

Throughput is usually measured in transactions per second (tps), messages processed per second (mps) or requests processed per second (rps). For instance, an on-line banking application might have to guarantee it can execute 1,000 tps from Internet banking customers. It's important to specify clearly in the throughput requirement that if it means peak throughput or the average throughput over a given period of time.

2.5.2 Availability

Availability is related to an application's reliability, but it is not the same. If an application is not available for use when needed, then it's unlikely to be fulfilling its functional requirements. Availability has been defined as the degree to which a system or component is operational and accessible when it is needed. Users are turned off if the service is not available due to frequent over-loading. Availability can be measured as a percentage of the total system downtime over a predefined period. it will be affected by system errors, infrastructure problems, malicious attacks, and system load.

Usually a failed service will stay unavailabile till the failure is detected and restarting the failed component is performed. Consequently, to have a system with high availability feature, minimizing the single points of failure and automatic failure detection and recovery are suggested. This is why recoverability quality attribute is closely related to availability.

In addition, replicating components is a tried and tested strategy for high availability. When a replicated component fails, the application can continue executing using replicas that are still functioning. This may lead to degraded performance while the failed component is down, but availability is not compromised.

2.5.3 Reliability

Reliability is the ability of the system to remain operating over time. According to O'Brien *et al.* (2007), two important aspects of reliability in SOA are the reliability of message passing between services, and the reliability of services. Failures in applications cause them to be unavailable. Failures impact on an application's reliability, which is usually measured by the probability that a system will not fail to perform its intended functions over a specified time interval as mentioned in Microsoft (2009).

Services are often made available over a network with possibly unreliable communication channels. Consequently, messages may fail to get delivered or will deliver in the wrong or-

der or will deliver more than once. However currently, messaging middlewares, like IBM WebSphere MQ and RabbitMQ (used in Openstack), support mechanisms to prevent these reliability problems. When different products with different messaging middlewares need to communicate, interoperability issue may cause reliability issue. Usually the SOA platform, not the service developer, is responsible for providing reliability.

Service reliability is the correct service operation and not either failing or reporting any failure to the service user. As O'Brien *et al.* (2007) discusses, the main challenge is managing the transactional context in order to preserve data integrity during failures and concurrent access.

2.5.4 Security

Security is the capability of a system to prevent malicious or accidental actions outside of the designed usage, and to prevent disclosure or loss of information. A secure system aims to protect assets and prevent unauthorized modification of information.

The most common security-related requirements for an application are:

- Authentication: Applications can verify the identity of their users and other applications with which they communicate.
- Authorization: Authenticated users and applications have defined access rights to the resources of the system. For example, some users may have read-only access to the application's data, while others have read–write.
- Encryption: The messages sent to/from the application are encrypted.
- Integrity: This ensures the contents of a message are not altered in transit.
- Nonrepudiation: The sender of a message has proof of delivery and the receiver is assured of the sender's identity. This means neither can subsequently refute their participation in the message exchange.

As discussed in Jansen (2010), several security metrics have been proposed in academia and industry such as vulnerability density, relative vulnerability, attack surface, severity-to-complex and security scoring vector.

According to Torkura *et al.* (2015), currently in cloud computing, Service Level Agreements (SLAs) used to guarantee security and privacy. As mentioned in Luna *et al.* (2011), unfortunately due to the Cloud's special characteristics there are just a few efforts aimed at using a framework or common set of objectives and quantitative security metrics for the Cloud. Briefly, as Luna *et al.* (2011) mentions, security metrics haven't gained attention in cloud security.

2.5.5 Scalability and Elasticity

Scalability is ability of a system to either handle increases in load without impact on the performance of the system, or the ability to be readily enlarged. Enlargement of the system may be increase either in the request load, simultaneous connections or data size.

In the perfect situation and without additional resources to serve the application, as the load increases, application throughput should remain constant, and response time per request should increase only linearly. As an example if an architecture for a server application designed to support 100 rps at peak load, with an average 1s response time, by growing request load by 10 times, throughput should remain 100 rps and response time per request should turn to 10s. On the other hand, a scalable system will permit additional resources to be deployed to increase throughput and decrease response time. This additional capacity may be deployed in two different ways, one by adding more resources to the machine the applications runs on (scale up), the other from distributing the application on multiple machines (scale out).

According to Gorton (2006), in reality, as load increases, application throughput decreases and response time increases exponentially. These happen because: First, the increased load causes increased contention for resources such as CPU and memory by the processes and threads in the server architecture. Second, each request consumes some additional resource (buffer space,

locks, and so on) in the application, and eventually this resource becomes exhausted and limits scalability.

Karacali and Tracey (2016) has evaluated the cloud scalability by examining how performance varies as a function of topology size and number of flows.

Scalability and elasticity are similar concepts and they are often confused. But they are different in some aspects. Scalability, unlike elasticity does not consider temporal aspects of how fast, how often, and what granularity scaling actions can be performed. In other words, scalability is a static property, and it is a time-free notion. However, elasticity is a dynamic property, which should consider how fast and how well a system will scale on-demand without interruption at runtime.

According to Karacali and Tracey (2016) which surveyed elasticity in cloud computing, elasticity is a complex problem which involves many aspects. Because many cloud providers have different architecture, elasticity metrics don't have standard metrics, which makes evaluating elasticity become difficult.

2.5.6 Interoperability

According to Microsoft (2009), interoperability is the ability of a system or different systems to operate successfully by communicating and exchanging information with other external systems written and run by external parties. An interoperable system makes it easier to exchange and reuse information internally as well as externally.

Communication protocols, interfaces, and data formats are the key considerations for interoperability. Standardization is also an important aspect to be considered when designing an interoperable system.

There are few researches dedicated to interoperability issues in the cloud and as far as the author of this thesis knows there is no metrics found to measure this quality. The Topology and Orchestration Specification for Cloud Applications (TOSCA) is a recent standard that has

focused on standardizing the way cloud applications are structured and managed to favor interoperability (OASIS (2013)).

2.5.7 Cost

Although cost is often not defined as a quality attribute in the software engineering literature, but the first question that arises in the mind of organizations before switching to cloud computing and choosing their Cloud provider is whether it is cost-effective or not. Therefore, cost is clearly one of the vital attributes for IT and the business. As mentioned in Garg *et al.* (2013), cost tends to be the single most quantifiable metric today and is one of the attributes in Service Measurement Index (SMI) which provides a holistic view of QoS needed by the customers for selecting a Cloud service provider.

Jallow (2016) has introduced cost as a metric in order to compare the quality of different cloud services provided by Amazon AWS and Google Cloud Platform.

According to the economical-view of quality in Section 2.3.1 (i.e. the notion of quality is benefit for cost), cost can be identified as a quality attribute in cloud context due to its importance to both cloud providers and cloud customers.

2.5.8 Energy-efficiency

The technology shift toward cloud computing introduced the growth of large-scale data centers world-wide; each of them contains thousands of nodes. These data centers consume more electrical energy which increase the operating costs as well as carbon dioxide (CO2) emissions to the environment. The growth of energy consumption is a real critical problem, for instance in 2015 the share of data centers from the total energy in Switzerland and US was 2.8% and 2.0% in order (Bertoldi (2014) and Cima *et al.* (2015)). Improving the efficiency of energy consumption in data centers is an effective endeavor towards increasing the sustainability in the future smart cities.

There are many research works which target energy efficiency in cloud-based data centers, mainly focusing on using host's CPU (Physical layer) efficiently, and many researches consider performance constraints of the VM (Infrastructure layer). Some other propose solutions with "VM to Physical host mapping" algorithms and DVFS and power switching strategies. Authors couldn't find any energy efficient solution in the literature that considers quality in Platform or Software layers.

2.6 Similar Works

The idea of considering quality in different cloud layers has been investigated in cloud monitoring domain in Trihinas *et al.* (2014), Montes *et al.* (2013), Mdhaffar *et al.* (2013) and Bruneo *et al.* (2015). One of the most similar works to MULQA in this domain is Bruneo *et al.* (2015), which presents a 3-D cloud monitoring framework called Ceiloesper. Three dimensions of this monitoring are cloud metrics, applications and physical machines. Bruneo *et al.* (2015) claims that all previous monitoring solutions fail to provide frameworks which have multi-layer monitoring and data stream analysis, and in the same time perform actions in different layers of the cloud. Ceilosper extends the OpenStack Ceilometer project and adds Complex Event Processing (CEP) engines to support the real time analysis of the collected data. Also, they have tested their solution on a scenario including high loads and low loads based on the Wordpress application. The layers of the cloud in this research are identified as: physical, virtualization, application architecture and application business logic. Although this research focuses on the monitoring and real-time analyze, rather than a autonomic quality management framework, it is similar to MULQA by monitoring from different layers of the cloud and triggering actions. In addition, Bruneo *et al.* (2015) uses Openstack as the cloud middleware, but their proposed architecture needs installations of extra modules per VM, per Node and per application on the cloud system, which decreases the scalability and interoperability of the architecture. However, MULQA by proposing a modular and customizable framework (based on Computing *et al.* (2006)) which controls upper layers of the cloud by leveraging Ansible. This minimizes

additional installations on the cloud, and provides extra freedom and security through standard SSH authentications.

Many studies also have been done on multi-layer frameworks for service computing. Among those, Sırbu and Babaoglu (2014) proposed a framework to characterize and identify the cloud system according to the analysis of datasets on four different metrics, namely power consumption, temperature, platform events and workload, which is similar to our platform regarding the selection of various metrics from different service layers. They also performed a further analysis on the correlation between the metrics from multiple layers in which some of them are highly correlated (i.e. power consumption and temperature), while some of them like platform events and workload are not correlated at all. However, in Sırbu and Babaoglu (2014), authors are not focused on optimum decision making and is based on the IBM BlueGene platform which is not suitable for this purpose. Taking fault prediction as the objective, Dudko *et al.* (2012) and Liang *et al.* (2007) strived to classify the state of IBM BlueGene cloud services through the machine learning algorithms and neural networks applied on the log of the low-level hardware failure and high-level kernel logs. To this end, they also used window based time-series, similar to our approach, to predict the errors in the system. However, in their work, a platform is not proposed to analyze the data.

In Jain *et al.* (2016), authors also performed multi-dimensional analysis of different performance metrics from various service layers and applied linear regression to predict the congestion and latency and manage other QoS in the network. Accordingly, the correlations among different performance metrics are evaluated. However, no decision making especially for unexpected and uncorrelated events are provided in their work. They also mention a need for out proposed platform to make decisions for uncorrelated events.

As mentioned earlier, none of the above literature has focused on the decision-making. One of the best platforms for decision-making is Daleel (Samreen *et al.* (2016)), a multi-criteria decision making to select the best type of the instance and the best time to create the instance according to applying regression technique on the dataset of the user requirements in different

service layers. However, in Samreen *et al.* (2016), the framework is just dived into the selection procedure neglecting the other actions such as VM migration and resizing that is addressed in our proposed framework. Moreover, their analysis is just limited to Amazon instances. In Hasan *et al.* (2012), taking into consideration metrics from different multiple layers, the authors proposed an Integrated and Autonomic Cloud Resource Scaler (IACRS) that extend the decision making analysis via the full automation of the cloud scaling. Similar to our proposed approach, policies in IACRS are defined according to the trigger events which have to be selected wisely. Hence, IACRS is focused on the selection of the metric triggers and their attributed thresholds in an effective and optimal manner. The other similarity between our proposed framework and IACRS is that they both are integrated with Openstack. However, their decision-making policies are constrained to the IAAS auto scaling and does not have anything to do with the other service layers.

Additionally, multi-layer quality management in cloud systems has been targeted by many researches in the energy-efficiency domain. Most of them focus on using host's CPU (Physical layer) efficiently and some consider performance constraints of the VM (Infrastructure layer). So the proposed solutions are usually VM to physical host mapping algorithms, DVFS and power switching strategies. Authors couldn't find any energy efficient solution references that considers quality in Platform or Software layers. Nathuji *et al.* (2007) proposes a hierarchical power management system. At the local level, the system coordinates and leverages power management policies of guest VMs at each physical machine; While global policies are in charge of managing physical nodes and have knowledge about rack or blade level characteristics and requirements. This work considers quality at the Physical layer (including some data center equipment) and Infrastructure layer. pMapper proposed by Verma *et al.* (2008) considers the energy-efficiency problem as continuous optimization and converts it to a bin packing problem. Their proposed system contains Migration Manager, Arbitrator, Performance Manager and Power Manager. Migration Manager issues commands for live migration of VMs. Arbitrator makes decisions about placements of VMs and which VMs to migrate. Performance Manager monitors applications behavior and resizes VMs according to current resource re-

quirements and the SLA. Power Manager adjusts hardware power states and applies DVFS. In pMapper, application SLA means their needed Infrastructure resources which is different than Software layer quality explained this thesis. So, only Physical and Infrastructure quality metrics are covered in this research. Kusic *et al.* (2009) investigates the behavior of each application running on the cloud using simulation-based learning. Using Kalman filter, future system states over a prediction horizon has been estimated by applying limited look-ahead control. In this paper, quality in Software layer has been considered indirectly using a learning system. Song *et al.* (2009) investigates the scheduling problem in three levels by introducing application, local and global level schedulers. Authors develop the RAINBOW framework which assigns priority to group of VMs which run a specific application and considers the Software layer QAs indirectly and limited. This model doesn't consider the case in which different applications running on a VM have different quality parameters. The GreenCloud project by Buyya *et al.* (2010) has a QoS-aware energy-efficient provisioning for Cloud resources. This research proposes real-time scheduling of VMs in cloud data centers and applying DVFS in order to minimize the energy consumption and deadline constraints of the applications. This solution considers the application quality as a condition on the hosted VM. For example in their simulation SLA violation occurs when a VM cannot get amount of MIPS that are requested.

In this section we briefly reviewed similar works to MULQA. Compared to the studied literature, MULQA is different in terms of introducing an autonomic, modular and customizable quality management framework to consider cloud QAs independently and in all layer of the cloud simultaneously. This idea complies the heterogeneity and scalability of the cloud systems at the same time. Also we have modeled, implemented and tested our proposed system with the same characteristics.

CHAPTER 3

MULQA SYSTEM DESIGN

3.1 Introduction

In this chapter, the architecture of MULQA, our proposed solution for making cloud systems quality-aware, is described and explained in detail.

According to state of the art (Section 2.4.1), some of the challenges in today's quality management in cloud systems are: the lack of application management and the lack of approaches to cloud deployment optimization services with various quality metrics such as performance and cost. Also in quality monitoring, managing different quality requirements is difficult, because in the cloud there exist multiple providers, and each of them needs distinct approaches to control their services.

This chapter first gives a general view of the quality model and the multi-layer concept. Then, it illustrates the model notations for layers, quality attributes, and metrics of the cloud. Later, it describes our modeling approach for the quality metrics, and it explains the finite state machine of the MULQA and the quality control mechanism.

3.2 Parameters and Notations

To describe the system design specifications and operations, some notations and symbols are presented. This section explains the notations for layers, quality attributes and metrics.

Figure 3.1 shows the hierarchy of these notations. The cloud controlled by MULQA consists of different layers (l_i) and each layer has its own quality attributes (e.g. q_j^i), and finally each quality attribute has some metrics (e.g. $m_k^{i,j}$).

In addition, all the notations introduced in this chapter are summarized in Table 3.1. Some of these notations will be explained later in this chapter.

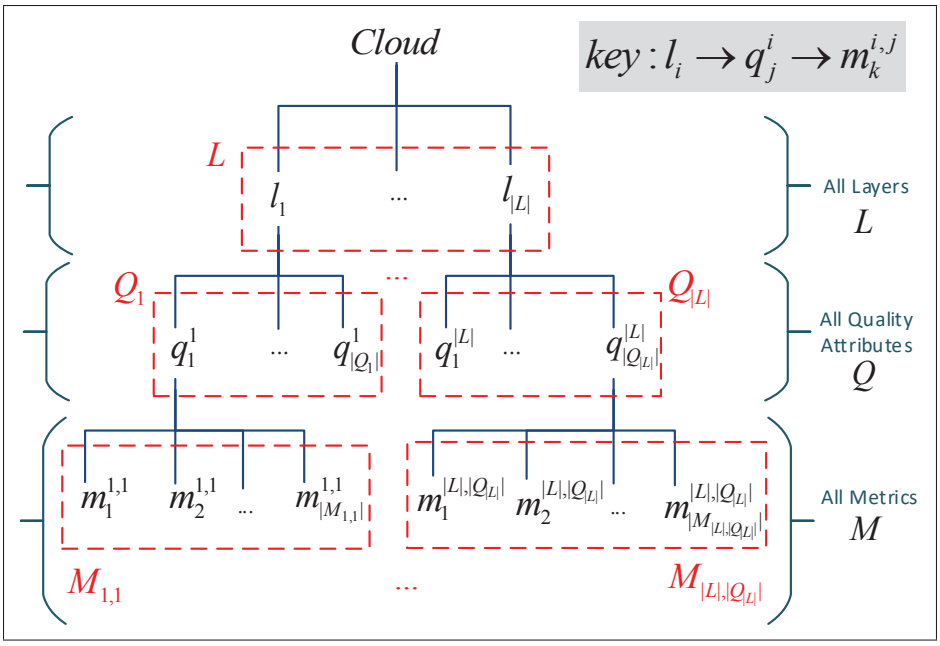


Figure 3.1 MULQA notations hierarchical model

3.2.1 Layers

MULQA is designed in order to be able to monitor metrics, and perform actions, in different layers of the cloud system autonomously, in order to improve different quality attributes. These targeted layers together create the set L which consists of some l_i. We have:

$$l_i \in L$$

For example, l_i can be $physical$, $infrastructure$, $platform$ or $software$. The cloud can be layered more fine-grained, to attain better separation of concerns. As an example, $\{presentation, application, business, data\}$ can be used instead of $software$ layer. Furthermore, quality manager may decide to add extra layers, such as $datacenter$ below $physical$ layer in order to quality control the data center facilities such as racks, cooling systems, power systems and security systems used in the data center.

Table 3.1 Summary of notations used in MULQA design

Notation	Description
l_i	ith layer of the cloud
q^i_j	jth QA of ith layer
$m^{i,j}_k$	kth metric for the jth QA of the ith layer
L	set of all layers of the cloud
Q_i	set of all QAs of the ith layer
$M_{i,j}$	set of all metrics for the jth QA of the ith layer
M	set of all metrics for all QAs in all layers
$T^{i,j,k}_{g,min}$	global minimum threshold for $m^{i,j}_k$ metric
$T^{i,j,k}_{g,max}$	global maximum threshold for $m^{i,j}_k$ metric
$T^{i,j,k}_{t,min}$	transition minimum threshold for $m^{i,j}_k$ metric
$T^{i,j,k}_{t,max}$	transition maximum threshold for $m^{i,j}_k$ metric
$T^{i,j,k}_{w,min}$	warning minimum threshold for $m^{i,j}_k$ metric
$T^{i,j,k}_{w,max}$	warning maximum threshold for $m^{i,j}_k$ metric
$T^{i,j,k}_{n,min}$	normal minimum threshold for $m^{i,j}_k$ metric
$T^{i,j,k}_{n,max}$	normal maximum threshold for $m^{i,j}_k$ metric
$I^{i,j,k}_g$	global interval for global thresholds of $m^{i,j}_k$ metric
$I^{i,j,k}_t$	transition interval for transition thresholds of $m^{i,j}_k$ metric
$I^{i,j,k}_w$	warning interval for warning thresholds of $m^{i,j}_k$ metric
$I^{i,j,k}_n$	normal interval for normal thresholds of $m^{i,j}_k$ metric
$\psi^{i,j}_k$	status of $m^{i,j}_k$ metric, compared to its thresholds
A	set of all actions available to execute by MULQA
a_n	nth action member of set A

Something we need to clarify here is that these layers are an abstraction for the level of components which exist in the cloud system and they are different from the kind of service levels obtained from cloud systems mentioned in Section 2.2.1 (e.g. SaaS, PaaS, IaaS).

3.2.2 Quality attributes

We target each quality attributes in distinct layers through q^i_j which denotes jth QA of the ith layer. As depicted in Figure 3.1, set of Q_i, includes all values of q^i_j in the specific layer of l_i. We have:

$$q_j^i \in Q_i \Rightarrow l_i \in L$$

While the q_j^i notation has been used in mathematical statements and definitions, to increase readability we have introduced the (i, j) equivalent tuple to be used in explanations.

As an examples:

$$Q_1 = \{q_1^1\} \sim \{(physical, cost)\}$$

$$Q_3 = \{q_1^3\} \sim \{(platform, reliability)\}$$

$$Q_4 = \{q_1^4, q_2^4, q_3^4\} \sim \{(software, reliability), (software, security), (software, performance)\}$$

Symbol of $\sim$ shows the equivalence relationship. Quality attributes set in each layer can include: performance, availability, reliability, scalability, cost, energy-efficiency or other quality manager custom defined QAs.

3.2.3 Metrics

In MULQA, each quality attribute of a targeted layer may have different metrics to measure and control. These metrics have been defined as $m_k^{i,j}$ which denotes kth metric from the jth QA of the ith layer. Additionally, as illustrated in Figure 3.1, $M_{i,j}$ is the set which includes all values of $m_k^{i,j}$ in the specific QA of q_j^i (which are for layer l_i). We have:

$$m_k^{i,j} \in M_{i,j} \Rightarrow (q_j^i \in Q_i) \wedge (l_i \in L)$$

Furthermore, all metrics of the layer l_i will make the set of M_i, and the global set of all metrics of the cloud will be denoted by the set of M. It is convenient that:

$$\bigcup_{q_j^i \in Q_i} M_{i,j} = M_i$$

$$\bigcup_{l_i \in L} M_i = M$$

Likewise, to increase readability we have introduced the (i, j, k) tuple to be used in practice, instead of $m_k^{i,j}$. As some examples:

$$M_{4,3} = \{m_1^{4,3}, m_2^{4,3}, m_3^{4,3}\} \sim \{(software, performance, responsetime), (software, performance, receivedreqs), (software, performance, throughput)\}$$

$$M_{3,3} = \{m_2^{3,3}\} \sim \{(infrastructure, performance, cpuusageVM2)\}$$

$$M_{1,2} = \{m_1^{1,2}\} \sim \{(physical, performance, cpuusageNode1)\}$$

3.3 Finite State Machine model of MULQA

In this part, the controlling mechanism of MULQA is expressed as a Finite State Machine (FSM) which is illustrated in Figure 3.2 with three main states: Normal, Warning and Transition state. The initial state of this FSM is Normal, and an Error state is considered to be used when MULQA can't handle violations. Penalties in the SLA, and SLA renegotiation, can be mediated for the error state, however, these issues are not covered in this thesis.

In all the main states (normal, warning and transition), an $updateState()$ function is running periodically, to check the violation (and warning) conditions (provided by Analyze module shown in Figure 3.4) of metrics, and return one of the *nState*, *wState* or *tState* events for the FSM. Also an *eState* can be returned if a transition takes longer than a specific time.

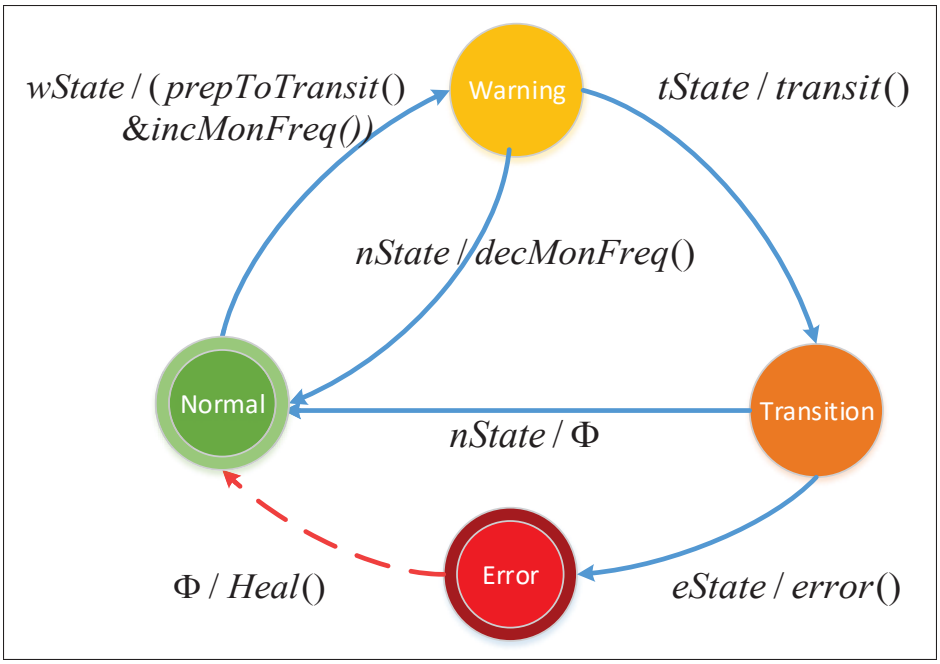


Figure 3.2 MULQA finite state machine

In the normal state, the quality metrics in different layers are periodically fetched by the Monitor module. After gathering the measured data, in the Analyze module, the quality metrics in different layers are compared with the associated thresholds. If a metric's value passes its attributed threshold, the violations or warnings are reported to the Plan module. Once the *updateState*() in this module, generates *wState* by evaluating violations and warnings, the system will be entered to the warning state and runs *incMonFreq*() and *prepToTransit*(). *incMonFreq*() is called to proactively monitor the system by increasing the monitoring frequency. Also entering warning state, executes *prepToTransit*() function to investigate the required operations and resources to return/compensate the values of quality metrics. The frequency for mining the information increases in order to monitor the information more precisely. For each metric, two kinds of thresholds for over-utilized and under-utilized situations are considered. If the *updateState*(), generates *tState*, actions will be executed through *transit*() function and the system will be transited back to the Normal state. All resources are provided and preparations are managed previously in the warning state (through running *prepToTransit*() previously) in order to run the *transit*() smoothly. Transition state should be done as short as

possible to not harm the quality parameters. For instance, in Transition state, actions like VM resizing, VM group scale-up, VM group scale-down and VM migration can be performed. If these actions take longer than a specific time, *eState* event will be triggered which takes the system to the Error state. However, if these actions succeed, the system will be back to the normal state by generating *nState* event.

Furthermore, if the system is currently in the Warning state and the $updateState()$ returns *nState*, $decMonFreq()$ will run and system will return back to Normal state. $decMonFreq()$ reverts the monitoring frequency by decreasing it to the normal state value.

Note that, the self-healing aspect of autonomous systems is not in the scope of this thesis, however it can be implemented as shown in Figure 3.2 in the Error state with the red arrow transition.

3.3.1 Metric thresholds

For each of the metrics provided by the system, consumer specifies intervals which satisfies her desired quality for that metric. MULQA defines four interval sets for each metric: Normal, Warning, Transition (or Violation) and Global. In this thesis, we have identified each interval set simply with two thresholds: $I = (T_{min}, T_{max})$. However, the general form could be a set which is not necessarily continuous.

Figure 3.3 illustrates these intervals. In this figure, the x-axis is for the metric values. All possible values for a metric such as $m_k^{i,j}$, are between its global maximum and minimum, which is $I_g^{i,j,k} = [T_{g,min}^{i,j,k}, T_{g,max}^{i,j,k}]$ interval, while the provider only provides the values in $I_t^{i,j,k} = [T_{t,min}^{i,j,k}, T_{t,max}^{i,j,k}]$ and the consumer expects the metric's value to be in $I_w^{i,j,k} = [T_{w,min}^{i,j,k}, T_{w,max}^{i,j,k}]$ which is the healthy interval for the metric as well. However, the consumer can ask for a strict condition on metric's value to be in $I_n^{i,j,k} = [T_{n,min}^{i,j,k}, T_{n,max}^{i,j,k}]$ too. $I_n^{i,j,k}$ also can be set by the quality manager or by default as a subset (i.e. 90%) of the $I_w^{i,j,k}$ interval to specify the warning state borders.

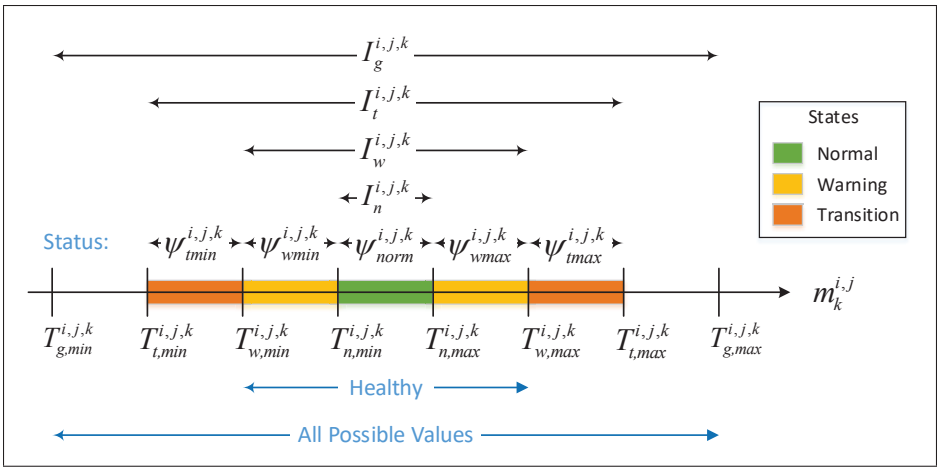


Figure 3.3 MULQA metric intervals and thresholds and their relationship with system states

Consumer specifies normal and warning intervals, and provider specifies the transition interval. Normal interval should be a subset of the warning interval. When a monitored metric's value exits the normal interval, it passes a normal threshold (upper or lower) which takes the system to the warning state. Next, the metric can either exit the warning interval (which takes system to the transition state) or go back to normal interval (which takes back the system to the normal state).

So for the normal and warning intervals of metric $m_k^{i,j}$, we have:

$$I_n^{i,j,k} = (T_{n,min}^{i,j,k}, T_{n,max}^{i,j,k})$$

$$I_w^{i,j,k} = (T_{w,min}^{i,j,k}, T_{w,max}^{i,j,k})$$

$$I_t^{i,j,k} = (T_{t,min}^{i,j,k}, T_{t,max}^{i,j,k})$$

$$I_g^{i,j,k} = (T_{g,min}^{i,j,k}, T_{g,max}^{i,j,k})$$

$$I_t^{i,j,k} \subseteq I_w^{i,j,k} \subseteq I_t^{i,j,k} \subseteq I_g^{i,j,k}$$

Note that $T^{i,j,k}$ depends on the metric $m_k^{i,j}$ and is defined based on the the quality sensitivity and SLA for that specific metric. For instance, in application layer, MySQL connection alarms (from 10/min to 20/min) , web-page error(from 0.01% to %1) or response time (from 50 msec to 1sec) can be different for different applications (Cima *et al.* (2015)).

3.3.2 Metric status, quality warning and violations

As explained in Section 3.3.1, if a quality metric passes the $T_{w,max}^{i,j,k}$ or $T_{w,min}^{i,j,k}$ thresholds, the quality constraints are most likely violated. We define metrics quality status set ψ, which includes statuses of each metric. Each member of ψ, presents the status of a metric such as $m_k^{i.j}$ compared to its thresholds, which can be: $\psi_{tmin}^{i,j,k}$ (for low value violation), $\psi_{tmax}^{i,j,k}$ (for high value violation), $\psi_{wmin}^{i,j,k}$ (for low value warning), $\psi_{wmax}^{i,j,k}$ (for high value warning) and $\psi_{norm}^{i,j,k}$ (for normal).

These values are added to ψ set, if the probability of being in a section shown in Figure 3.3 (next to *status*) becomes higher than a benchmark presented by ε for a specific metric. As a result, for the violation status, we introduce the following logical statements:

$$if\ (Pr(T_{w,max}^{i,j,k} < m_k^{i,j} < T_{t,max}^{i,j,k}) > \varepsilon) \Rightarrow \psi.Add(\psi_{tmax}^{i,j,k}) \tag{3.1}$$

$$if\ (Pr(T_{t,min}^{i,j,k} < m_k^{i,j} < T_{w,min}^{i,j,k}) > \varepsilon) \Rightarrow \psi.Add(\psi_{tmin}^{i,j,k}) \tag{3.2}$$

For warning statuses, *wMin* and *wMax*, similar statements with are evaluated:

$$if\ (Pr(T_{n,max}^{i,j,k} < m_k^{i,j} < T_{w,max}^{i,j,k}) > \varepsilon) \Rightarrow \psi.Add(\psi_{wmax}^{i,j,k}) \tag{3.3}$$

$$if\ (Pr(T_{w,min}^{i,j,k} < m_k^{i,j} < T_{n,min}^{i,j,k}) > \varepsilon) \Rightarrow \psi.Add(\psi_{wmin}^{i,j,k}) \tag{3.4}$$

And finally, for the normal status:

$$if\ (Pr(T_{n,min}^{i,j,k} < m_k^{i,j} < T_{n,max}^{i,j,k}) > \varepsilon) \Rightarrow \psi.Add(\psi_{norm}^{i,j,k}) \tag{3.5}$$

Clearly, ψ depends on the metrics in all layers including physical and virtual resources, platforms and applications.

3.3.3 Control

As discussed earlier, in order to control a quality violation, first system needs to analyze the metric values to predict the status of the metrics.

The *Analyze* function by evaluating Equations 3.1 to 3.5 for the predicted metric values, returns the statuses for metrics. This function can be explained as:

$$Analyze : M \rightarrow \psi$$

Note that MULQA is open to different kind of decision making mechanisms. Mechanisms such as machine learning or control theory mentioned in Section 2.3.2.

After analyze, to control the violation, system needs to run $prepToTransit()$ while moving to the warning state, to plan for the the violation situation. This planing could be for instance, allocate (for over-utilized case) or de-allocate (for under-utilized case) a resource. Next, we run $transit()$ when violation is about to happen to run the planned situation in order to return to the normal state while preventing the violation.

What we plan, in order to prevent the violation of metric $m_k^{i,j}$, would be a planning for a set of actions which is relevant to $m_k^{i,j}$. We define actions set A, to cover all these possible actions, for all kinds of violations. We have:

$$a_n \in A$$

For instance, in a setting these actions can be: $VmResize$, $ScaleupVmGroup$, $ScaleDownVmGroup$ and $LiveMigrateVm$.

The set A, is the answer to the "what to do?" question. On the other hand, plan function defined below is the answer to "when to do a specific subset actions of set A?":

$$Plan : \mathscr{P}(\psi) \rightarrow \mathscr{P}(A)$$

In this function, the domain is $\mathscr{P}(\psi)$, which denotes the power set of ψ, or in the other words, set of all subsets of the status set. Similarly, the codomain is the power set of actions set.

3.4 General Model

To be able to monitor all quality attributes in different layers of the cloud autonomously, MULQA is designed as illustrated in Figure 3.4. This general design improves the initial design concepts of ACCS mentioned in Section 2.2.5; However, MULQA's general design has two main differences:

- Cloud layers and layer agents are introduced which yields to fine-grained quality control and troubleshooting of the overall cloud.
- The human guidance has been introduced through an API module which is used by a quality manager to customize other modules. Furthermore, API could be used by cloud layer consumers and providers to ease and clarify the SLA negotiation by agreement on fine-grained quality metrics which is pushed to the knowledge module too.

As mentioned in Section 2.2.5, autonomic systems are favored to handle situations like uncertainty, heterogeneity, dynamism and faults easily. This is the reason for using ACCS ar-

chitecture in the MULQA design. ACCSs sense, monitor, and react based on the situations, such as self-healing, self-protecting, self-configuring, and self-optimizing. However among these self-management abilities, MULQA's first goal is to build a framework to enable researchers customize the system with their desired quality management algorithms which can be categorized to provide self-configuration and self-optimizing kinds of self-management to the targeted cloud system; However, it can be used to improve self-healing and self-protecting too.

MULQA is composed of functional and non-functional components. Functional components include API, Monitor, Analyze, Plan and Execute. Also MULQA has Knowledge as a non-functional component where all the information about the environment and the system is located. For a system component to be self-managing, it must have an automated method to collect the details it needs from the system; to analyze those details to determine if something needs to change; to create a plan, or sequence of actions, that specifies the necessary changes; and to perform those actions. When these functions can be automated, an intelligent control loop is formed. The architecture dissects the control loop into four parts that share knowledge: monitor, analyze, plan and execute. The autonomous control flow is shown with black arrows in Figure 3.4. This flow can be explained as below:

a. The monitor function provides the mechanisms that collect, aggregate, filter and report details (such as metrics and topologies) collected from a managed resource through Sensor agents and transfers this information to the next module for further analysis.

b. The analyze function provides the mechanisms that correlate and model complex situations (for example, time-series forecasting and queuing models). These mechanisms allow the autonomic manager to learn about the IT environment and help predict future situations.

c. Once data has been analyzed, the plan function provides the mechanisms that construct the actions needed to achieve goals and objectives. The planning mechanism uses policy information to guide its work.

d. The execute function provides the mechanisms that control the execution of a plan with considerations for dynamic updates.

e. Finally Effector agents are used to transfer the new policies, rules, and alerts to other nodes of the autonomic system with updated information.

In Figure 3.4 three kinds of actors are illustrated: cloud consumer, cloud provider and quality manager. MULQA offers a general and customizable system. Autonomic systems need assistance from humans and Quality manager is the one who is supervising the quality control procedure. This actor by using API, configures and customizes MULQA autonomous modules. Quality manager can change all functional and non-functional modules through the non autonomous flow shown with blue arrows in 3.4.

As Figure 3.4 presents, a cloud controlled by MULQA, will have two kinds of agents (shown with circles) in each layer to I) sense metrics and transfer data to the monitor module II) trigger desired actions sent from executer module. The rule of these agents is similar to Sensors and Effectors in IBM's autonomous model.

Moreover, MULQA modules can be deployed centralized or distributed, inside or outside the cloud which is quality-controlled by MULQA.

3.4.1 Discussion

Satisfaction of the cloud customers is based on how much the cloud service is provided with accordance with the SLA and advertised QoS. While a violation in one layer or a components of the cloud can affect the whole system, providing the desired End-to-End (E2E) quality level must be considered in quality management. According to Toosi *et al.* (2014), this is one of the major issues in the current solutions of the cloud. MULQA by investigating different qualities in all layers (from top layer to the bottom) of the cloud, is an effort to fix this issue.

Moreover, as mentioned in Beach *et al.* (2015), quality must be considered in the different phases of the life cycle of cloud solutions, including the component design of the applications.

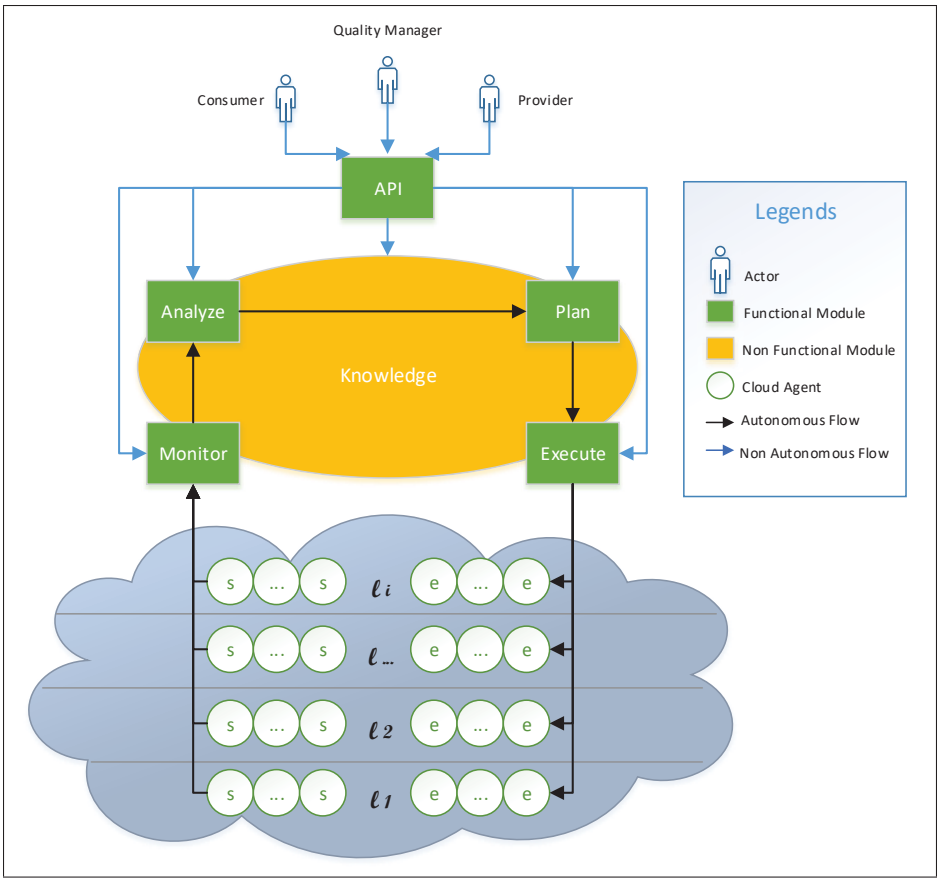


Figure 3.4 General model of MULQA

Thus, it is important to think of suitable architectural decisions (while designing a cloud application) for cloud developers who want to quality-control their system with MULQA. Complementarily, MULQA as a self-managed system is introduced to improve the E2E behavior of the global service.

In addition, stakeholders in MULQA can negotiate about SLA more clearly due to availability of fine-grained quality metrics. This will be explained in Section 3.5.

Another approach for hierarchical classification of the quality metrics could be first divide the cloud by layers, and then by components of the layers and finally metrics of those components. However, MULQA instead of components, considers quality attributes, which makes the quality negotiation more understandable, where customers can focus on their desired QA based on

their concerns, especially QAs usually affect each other negatively or positively as mentioned in Section 2.3.3. However, MULQA's approach, may be more difficult for the providers who want to add metrics for their recently added components.

3.5 SLA Negotiation

As mentioned previously in Section 2.4.2, achieving SLA requirements is difficult and challenging in cloud resource management. Regarding to the state of the art, in future models, SLA definition and specifications must be addressed in a suitable way that covers the consumers' expectations. Moreover, studies suggest that there should be fine-grained quality-based negotiation between providers and users to fulfill SLAs, and the literature doesn't include models to cover this problem effectively. As an example, there is no model to enforce limit mechanisms on the virtual machine resources to meet SLAs. Furthermore, the literature mentions that the risks related to these kinds of SLA violations need to be managed.

Figure 3.5 shows the interaction of the cloud consumer and cloud provider to negotiate SLA in autonomic clouds which has been discussed in the literature. In this model different consumers select the QAs they care for the most, and negotiate over them with providers. This model gives a general idea of SLA negotiation and it doesn't go through the details such as: QA metrics, QA of which layer or component and the user-story of the SLA negotiation. Also there is no link between SLA negotiation (non-autonomic) and the autonomic aspects of the cloud.

Stakeholders in MULQA can negotiate the SLA more clearly due to availability of fine-grained quality metrics. Figure 3.6 shows an SLA negotiation scenario for a cloud controlled by MULQA.

In this process, first the providers (through R_i^p request set for provider i) submit all the metrics of the services they are providing, as well as the I_t intervals for each metric which specifies the possible quality thresholds for the providing qualities. Next, each consumer (through R_j^c request set for consumer j) selects a subset of the provided metrics which they are interested in, as well as the I_w and I_n intervals for each selected metric. Quality manager (through R_k^q

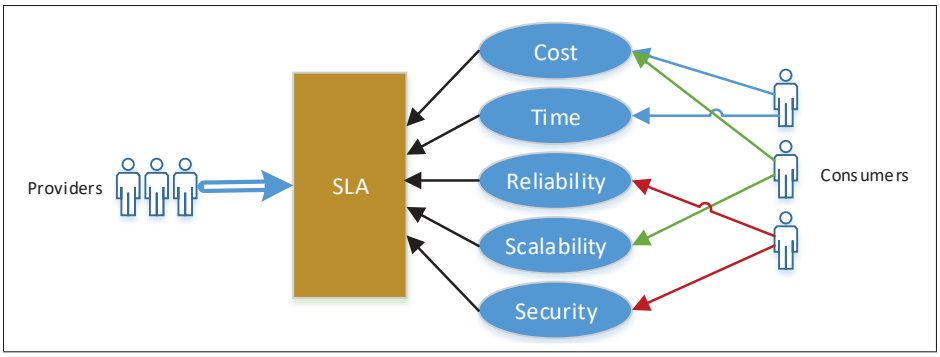


Figure 3.5 Literature model for quality-aware SLA in autonomous systems
Taken from Singh and Chana (2016)

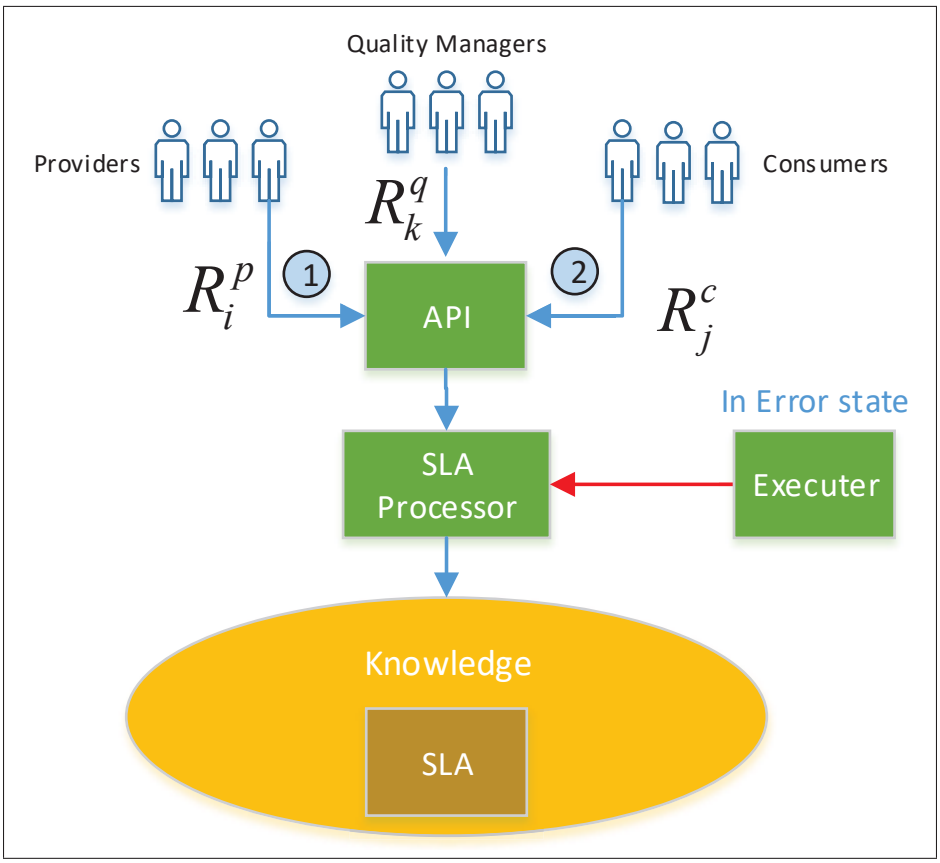


Figure 3.6 MULQA model for quality-aware SLA negotiation

request set for manager k) can set the SLA management parameters (e.g. default percentage of $I_n^{i,j,k}/I_w^{i,j,k}$ when the consumer only provides the $I_w^{i,j,j}$). All these requests enter the system from

API module and they pass the "SLA Processor" module in order to manage the conflicts, and be verified. For instance, a simple verification could be checking if $I_w^{i,j,k} \subseteq I_t^{i,j,k}$ for all metrics like $m_k^{i,j}$. All the SLA inputs from system stakeholders will be stored in the knowledge module, in order to be used by other modules (such as analyze and plan) in the future.

Moreover, desired consumer QA metrics can be prioritized by a weight parameter which can be added to R_k^c for any selected metric of any consumer. Also, in the case of Error state in the FSM mentioned in 3.3, penalties and SLA renegotiation process can be managed through a call from the Executer module to the SLA processor module.

CHAPTER 4

MULQA IN ACTION

4.1 Introduction

In the previous chapter, the general design and architecture of MULQA was explained through different diagrams and notations. This chapter dives into the details of MULQA implementation and the decisions made to fit the use-case and tests. This chapter explains how MULQA can be used in practice, in a cloud system with OpenStack installed middleware, and pictures the deployment architectures and describes the implemented modules.

Moreover, a three-tiered web application use-case which is designed to validate our solution, has been described. Also, the results of the experiments and the related discussion is provided.

To test out MULQA, first a distributed cloud-native application use-case have been designed to be able to run on top of the Openstack infrastructure. The use-case application for this thesis is a three-tiered web application which consists of a web presentation tier, an application tier, and a persistent database tier. Second, the Openstack infrastructure has been designed and deployed in order to provide the needed requirements of the use-case application. Then, the use-case application has been deployed on the Openstack.

Moreover, MULQA will be installed on the system in order to provide the autonomous multi-layer quality-aware cloud functionality. Finally, the sample test scenarios are designed and run in order to show the MULQA capabilities and picture the results. Figure 4.1 shows the steps taken in order to do experiments in this thesis.

Since our use-case is a web-application, the selected meters and actions in all layers of the cloud will be relevant to this use-case.

Web application use-case is chosen, because it is the most prevalent application in today's business with more than 1.1 billion websites online today as reported by Netcraft (2017). Web apps

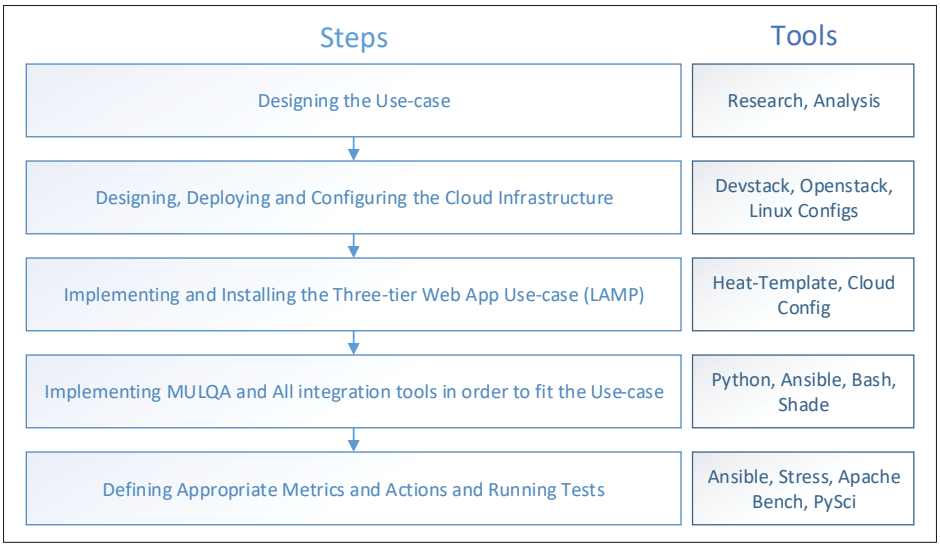


Figure 4.1 Steps to test MULQA in action and the used tools

are client-server applications in which the client usually runs in a web browser. Web applications include common websites for web-mails, social media, e-commerce, online banking and more. Another reason for selecting this test-case is, because they are typically characterized by IT resource requirements that fluctuate with usage, predictably or unpredictably and failure to respond to either can impact customer satisfaction and sales. So, quality assurance of the IT systems is crucial to these applications. Chosen web stack to run the use-case, is a LAMP stack which consists of Linux, Apache, MySQL, and PHP. LAMP is considered by many as the platform of choice for development and deployment of high performance web applications.

In this thesis, Openstack is chosen as the cloud middleware due to its open source code and its popularity and growth in today's cloud market. Openstack offers comprehensive platform for all IT applications, offering agility and cost-effectiveness by controlling large pools of compute, storage, and networking resources throughout datacenters, all managed through a dashboard or API. Also Openstack offers monitoring and orchestration of the cloud resources which have been used by MULQA in order to perform its tasks.

This chapter is divided into three main sections: Implementation, Deployment, Experimental Results. In the first section, the simplified implementation of MULQA modules is described.

The designs explained previously in Section 3 are implemented in more details to fit the use-case and tests. Next, the infrastructure architecture in Openstack has been explained, and following, the deployment of the use-case is described in detail. Finally, this chapter talks about the tests performed and system settings, and illustrates the results.

According to the use-case and the available resources, layers set in MULQA implementation and deployment in this thesis is defined as:

$$L = \{l_1, l_2, l_3, l_4, l_5\} \sim \{Physical, Openstack, Instance, Platform, Software\}$$

Note that, in this implementation, $\{Openstack, Instance\}$ is used instead of $Infrastructure$ layer to target Openstack components and VMs separately.

Figure 4.2 shows these layers with some of their deployed components in this thesis. $c1\_2server$ and $c1\_3server$ are physical blade servers which Openstack has been deployed on. vm_web1 and vm_web2 are two instances of a web_server scaling group deployed in Openstack. Also $apache$ refers to Apache web-server deployed on some instances of the Openstack. $Wordpress$ has been used as the main software to be accessed through $apache$. WordPress is a free and open-source content management system (CMS) based on PHP and MySQL and according to W3Techs (2017), it was used by more than 27% of the top 10 million websites as of December 2016.

4.2 MULQA Implementation

In the implementation phase of MULQA, different programming and scripting languages such as Python, Bash and YAML are used which are listed in Figure 4.1. Also monitor and execute modules use Ansible playbooks to communicate with both Openstack nodes (computes, controllers and etc.) and instances (VMs). Ansible is an open-source automation engine that automates cloud provisioning, configuration management, and application deployment. Once installed on a control node, Ansible, which has a secure agentless architecture, connects to a managed node through the default OpenSSH connection type. Using Ansible for communicat-

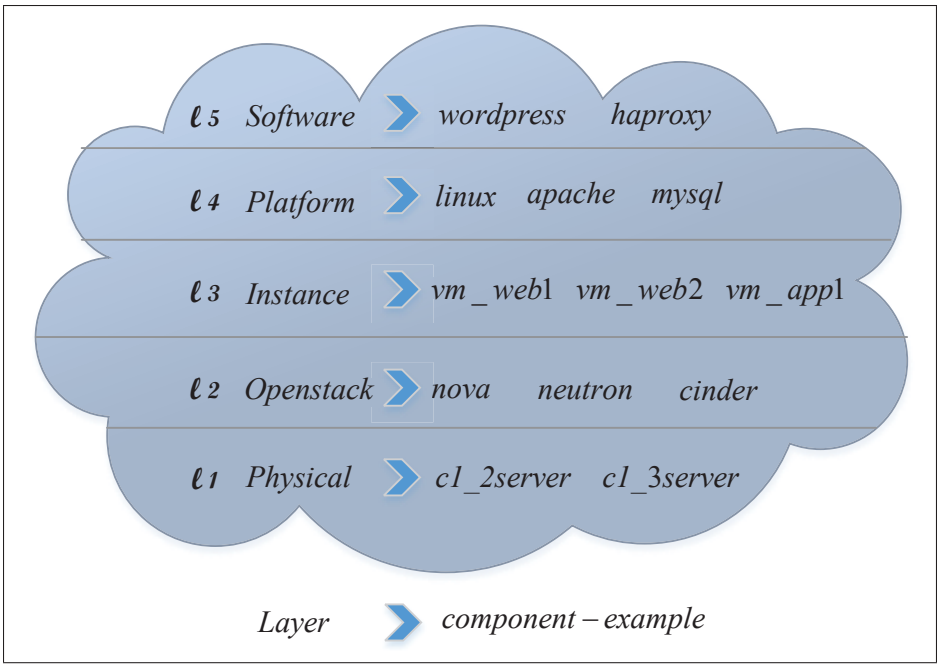


Figure 4.2 Layers defined in MULQA implementation and examples of deployed components in each layer

ing with nodes and instances, eliminates the need for having agents and listeners or any extra package installation on these elements of the cloud. Moreover, having more than 700 modules (Ansible (2017a)), Ansible eliminates coding from scratch for monitoring and control tasks. Installing Ansible tower is recommended for this implementation which enables playbooks to be triggered by requesting a URL. This can be used directly in alarm action trigger of a stack deployment in Openstack.

Python is used mainly for implementing the main MULQA system and its modules. Some of imported libraries in the Python codes are shade, mysqldb, and Openstack clients. Shade (Openstack (2017c)) is a simple client library for interacting with OpenStack clouds which eases and decreases the code amount for regular cloud commands, rather than using REST APIs and parsing the data.

YAML is used to build the Heat templates in order to make the use-case stack (i.e. three-tier web application). Also, Cloud config (Cloud-init (2017)) is used to configure the Cloud-init scripts on stack instances' first run.

In addition, devstack local.conf and local.sh configs for both controller and compute nodes are provided and some useful tools have been developed to ease and clarify the reproduction of the implementations and deployments.

4.2.1 Data storage

MULQA uses two kinds of databases. A time-series database is used for storing the metered data and another Mysql database for storing MULQA parameters and settings. Time-series database can be Openstack's default Gnocchi or other third-party DBs such as InfluxDB. In Monitor, Analyze, Plan and Execute modules, associated algorithms are stored in files. These algorithms can be changed through the API module by the quality manager.

In order to test the design of MULQA, described in Chapter 3, the system workflow and its modules can be implemented as shown in Figure 4.3. In this figure the control loop is shown with numbered arrows.

4.2.2 API and SLA modules

API module is connected to Monitor, Analyze, Plan and Execute modules as well as the MULQA DB. Cloud providers, cloud consumers and quality managers access MULQA through this API. The suggestion for this API is an Openstack style CLI and a Horizon plugin. This functionality of MULQA has not been implemented in this work. Quality manager will be capable to change the algorithms in the other modules and stakeholders will be able to input their desired metrics and their associated thresholds. Additionally, for the SLA negotiation and enforcement, MULQA can be integrated with Congress project of Openstack as explained in Giannetti and Owens (2016). These SLAs can be defined as a set of complex policies in

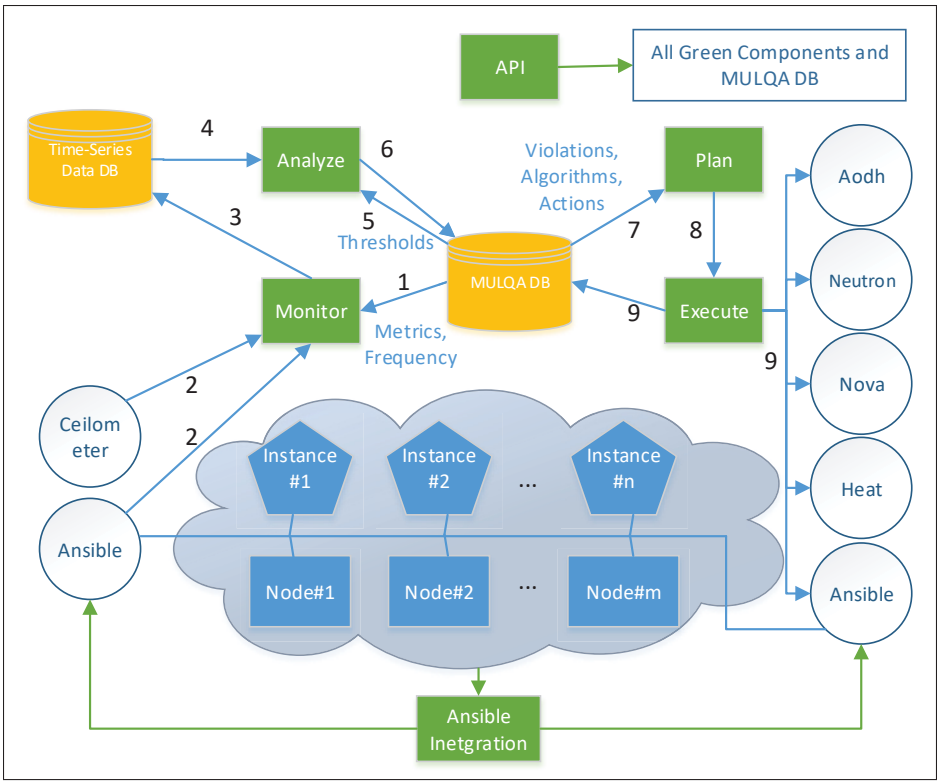


Figure 4.3 MULQA implementation modules and the control loop workflow

Congress, that identify conditions to be met in the infrastructure. The SLA implementation is not in the scope of this thesis.

4.2.3 Ansible integration module

Note that, Ansible before running a playbook, needs to be fed with the list of hosts (a.k.a. inventory). As the instances and nodes of the cloud are dynamically changing, we can't use the default static hosts file. One solution could be using a pull model for playbook execution, which makes the system more dynamic and scalable. However, this approach needs Ansible to be installed on all nodes and instances of the cloud and it increases the complexity of the management of the playbook execution. The proposed and implemented solution in this thesis is an application which performs the dynamic inventory generation continuously. This program has been developed in Python and fetches both the current Openstack nodes (through

nova – manage CLI parsing) and Openstack instances (through Shade calls) and then categorizes them by their host type or instance type (i.e. [openstack-compute], [openstack-scheduler], [instances-web], [instances-db]).

4.2.4 Monitor

The monitor function provides the mechanisms that collect, aggregate, filter and report details (such as metrics and topologies) collected from a managed resource through Sensor agents and transfers this information to the next module for further analysis.

To monitor all layers of the cloud, this thesis has used Openstack's Telemetry projects and Ansible. Openstack's Telemetry project has been divided into smaller projects such as Aodh (an alarming service), Ceilometer (a data collection service), Gnocchi (a time-series database and resource indexing service) and Panko (an event, metadata indexing service). Figure 4.4 shows how these projects are connected and the high-level architecture of Telemetry in Openstack.

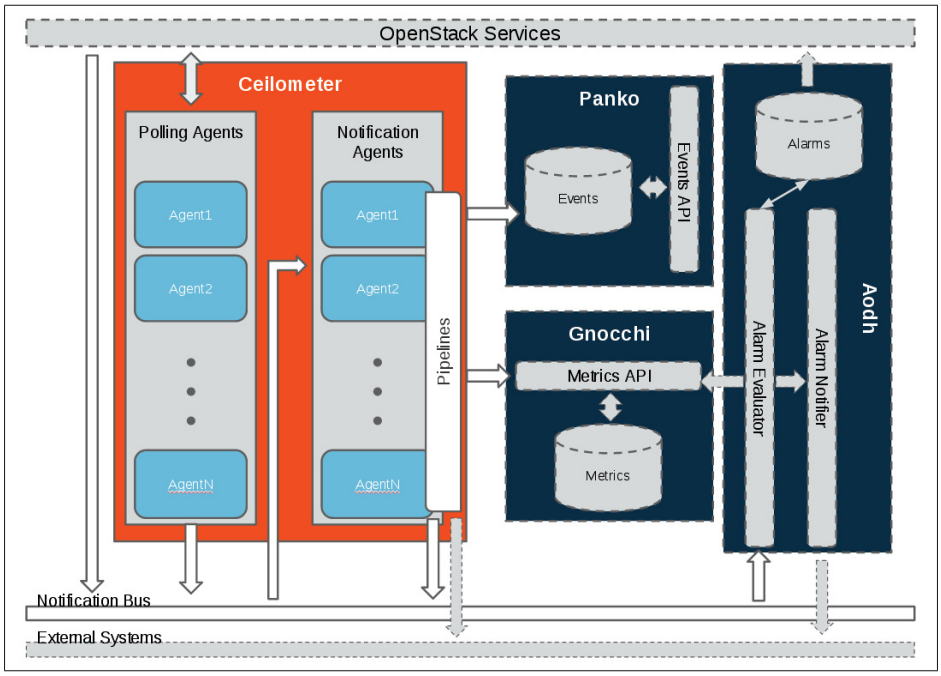


Figure 4.4 High-level architecture of Openstack Telemetry

As illustrated in Figure 4.3, Monitoring module first fetches the list of metrics to collect and the frequency for the collection, from the database. Next, this modules collects data from two main sources. Ceilometer has been used mostly to collect the metrics related to the lower layers of the cloud (e.g. Physical, Openstack and Instance layers). Metrics such as: CPU, RAM, Network and IO utilization of Openstack nodes and instances, object storage and block storage metrics and SDN and Load balancer metrics. List of all these metrics can be found in Openstack (2017d).

On the other side, by using Ansible calls, data collection from upper layers (e.g. Platform and Software) is performed. For example, data collection of metrics related to Wordpress, Apache webserver, MySQL and Linux services can be performed by running Ansible playbooks.

Frequency of the data collection for the Ceilometer metrics is configured in the pipeline.yaml file next to the interval value for each metric. For the other metrics which their value will be fetched in a loop through a playbook call, the running interval for each Ansible playbook determines the monitoring frequency. Note that, to utilize other monitoring tools, extra local sensors can be installed and configured on the instances and nodes through Ansible too. For example monitoring agents of systems such as Nagios, Zabbix or Sensu can be easily integrated with the MULQA system. Monitoring modules of Ansible are listed in Ansible (2017b).

After collecting the data, Monitor module stores them in a time-series database which can be Openstack's Gnocchi or other similar databases to be used in the Analyze phase.

4.2.5 Analyze

As mentioned before, the analyze function is to provide the mechanisms that correlate and model complex situations to allow the autonomic manager to learn about the IT environment and help predict future situations.

This module fetches the metering data from the time-series DB and then predicts the future values for the desired metrics and compares them to the thresholds fetched from the MULQA

DB and updates the violations and states data in the MULQA DB which is used in the Plan phase.

Though MULQA is open to different estimation methods to predict the quality metrics variables, in this section a simple estimator is introduced to trace the various quality metrics.

Assuming $m_k^{i,j}$ as the variable metric value. The estimated value of the $m_k^{i,j}$ can be obtained by an Exponential Weighted Moving Average (EWMA) estimator (Guo and CHEN (2002)) in which the most recent rational subgroup mean and the current value of the metric will be used iteratively over the time according to the following Equation:

$$\mathbb{E}m_k^{i,j}(t+1) = \alpha_k^{i,j} m_k^{i,j}(t) + (1-\alpha_k^{i,j})\mathbb{E}m_k^{i,j}(t)$$

Where the parameter $\alpha_k^{i,j}$ is, the weight given to the most recent historical data and $\alpha_k^{i,j} \in (0,1)$. $\alpha_k^{i,j}$ selection is a matter of experience and does depend on the dynamicity of the metric so that more dynamic the metric lower the value of α should be selected. However, tracking the variable in a short time is enough to select the best weight coefficient.

Note that, since scale and noise of the system increases, these quality metrics become more complicated and harder to be tracked from the software layer to the physical layer. Under these circumstances, more advanced estimation methods are recommended.

4.2.6 Plan

As depicted in Figure 4.3, once data has been analyzed, the plan function by fetching the state of the metrics and violations from MULQA DB, provides the mechanisms that construct the actions needed to achieve goals and objectives. This module should be designed to find the best set of actions and send it to the execute module in order to control the system. To plan efficiently, this module will fetch the available actions and their effects from the MULQA DB.

In the implementation of this thesis, Plan module is a simple mapping algorithm which is based on the experience.

However, MULQA system can be easily used to implement planing algorithms based on different approaches like control theory, machine learning or operational research and other optimization algorithms.

4.2.7 Execute

Due to diversity of coverage in Monitor, Analyze and Plan modules on all layers and different QAs, the Execute module should have the similar control action capabilities. This tends to design a module which is able to trigger actions in different components of the cloud, such as: Openstack modules including Nova, Neutron, Heat, Aodh and Openstack instances through Ansible calls. Execute module in MULQA is brainless, and as Figure 4.3 depicts, it just runs the action set provided by Plan module and sends appropriate signal to the other modules which are described in the following.

Aodh and Panko components from the Telemetry project of Openstack can be called to make an alarm or a notification in response to a specific state change of the cloud or violation (warning or transition). A common usage of calling Aodh and Panko can be a response to a hypervisor failure, which triggers evacuation alarm for an Openstack node. If a hypervisor needs to be taken down for maintenance, the source hypervisor should be emptied by moving instances to other target hypervisors. Rebuilding instances is required when something goes horribly wrong. The instance is booted from a new disk, but preserves its configuration including the IP address.

Also some of the other actions can be plugged (with a REST API call) to the action trigger attribute of an alarm. For instance scale-up URL can be attached to the CPU alarm of a scaling group.

Neutron can be called in order to control the Openstack network including: adding, removing or configuring the routers, load balancers or other network components, changing the topology or even change the SDN layer.

Moreover, Nova can be called to perform an action (generated by a command from the Plan module). Among common usages of Nova calls are, resizing or migrating an instance. When an application runs into resource limits and does not scale out (for example in autoscaling group of instances), scaling up can be considered. With Openstack enabled cloud, its possible to change the memory, CPU and storage on the servers. Someone can start with a smaller set of resources and then order additional memory or CPU for the physical servers or move to larger systems as the need arises. There is, however, a limit to how much resources can be scaled up and it may be cost prohibitive. Most of these resize operations require downtime. When you resize the primary disk on a dedicated server, the cloud provider may reinstall the operating system. With the proper partitions, OpenStack allows the resizing of ephemeral storage without loss of data and will move the VM to another hypervisor in the process. *allow_resize_to_same_host* parameter can be set to allow resizing on the same host. If supported, hot-add RAM (Kernel (2016)) and hot-plug CPU (Raj (2016)) functionalities can be performed too, so even there will be no need to shut down the virtual machine or application.

Another action to execute using Nova calls is migration. Migrating VMs between cloud nodes may not be straightforward; However, Openstack allows seamless migration between virtual and physical environments. For instance, if someone run into resource limits that are I/O bound, she can move to more powerful dedicated servers with Solid-State Drives (SSDs) and RAID. In OpenStack, migration provides a scheme to move instances from one OpenStack compute node to another and it is useful for redistributing the load among the available hypervisors. There are two types of migration: Live (or Hot) and Non-live (or Cold). Non-live is where the instances will be shut down for the move to another hypervisor. While, live migration is where the instance will be kept running. Live migration offers extreme versatility but may result in degraded performance during the migration. Among live migration types (shared storage based and block migration), block migration is incompatible with read-only devices such as

CD-ROMs and Configuration Drive. Since the ephemeral drives are copied over the network in block migration, migrations of instances with heavy I/O loads may never complete if the drives are writing faster than the data can be copied over the network.

Heat is called when we want to create, remove or modify a resource in a stack. In the use-case of this thesis for example, the three-tiered web application stack can be modified in response to an observation in the layers of our cloud. A common usage for the heat calls is performing horizontal scaling in a scaling group of a stack. Scaling out (horizontal scaling) is often cheaper, easier to run fault-tolerance and easy to upgrade, compared to Scaling up (vertical scaling or resize). On the other hand some cons of this method are: more licensing fees, bigger footprint in the data center, higher utility cost and possible need for more networking equipment (switches/routers).

Ansible will be called to perform any action which can't be handled through other components described above. These actions usually include the ones controlling below Openstack layer (like DVFS in the in the Physical layer) and above instance layer (Software and Platform layers). Ansible can perform any possible action which can be run though SSH, in all instances and nodes of the Openstack. These actions can include adjustment of OS in Openstack nodes and instances, database actions, Apache configuration and even Wordpress configuration and updates. One approach to save energy and operational costs is performing Dynamic voltage and frequency scaling (DVFS) in the physical node. This approach enables the operating system to scale the CPU frequency up or down in order to save power. CPUs can be dynamically disabled and re-enabled on a Linux system.

Connection to almost all components of the cloud, creates a lot of freedom in actions, which provides tremendous control capabilities on the MULQA cloud.

Also this module can access the MULQA DB to update some values such as monitoring_-frequency, which is used by other MULQA components.

4.3 Deployment

In this section, first the proposed and deployed Openstack architecture for the infrastructure has been discussed. Next, the use-case application which has been deployed on top of the Openstack has been explained.

In this project, first we deployed an Openstack controller node and then a compute node using Devstack on two physical blade servers. Next the required images and keys and other configurations are installed and then, we launched the use-case stack to be deployed on the Openstack layer. Later, MULQA will be installed in order to control the cloud.

4.3.1 Openstack

Openstack has been used as the cloud middleware for MULQA. The suggested Openstack deployment for the three-tiered web use-case in production setup is composed of more than five Openstack nodes and more network considerations to insure the quality, and not being a bottleneck for the upper layers. However, due to the limited facilities available to this project, the setup for the tests of this thesis is composed of two openstack nodes with multi-node setup. Openstack deployment architecture should be based on the chosen hardware (for compute, storage and network) and the requirements for the cloud. For example based on Openstack (2017a), an online classified advertising company who wants to run web applications consisting of Tomcat, Nginx and MariaDB in a private cloud may need:

- Between 120 and 140 installations of Nginx and Tomcat, each with 2 vCPUs and 4 GB of RAM
- A three-node MariaDB and Galera cluster, each with 4 vCPUs and 8 GB RAM

On a typical 1U server using dual-socket hex-core Intel CPUs with hyperthreading, and assuming 2:1 CPU overcommit ratio, this would require 8 OpenStack compute nodes. The general architecture for this deployment is illustrated in Figure 4.5.

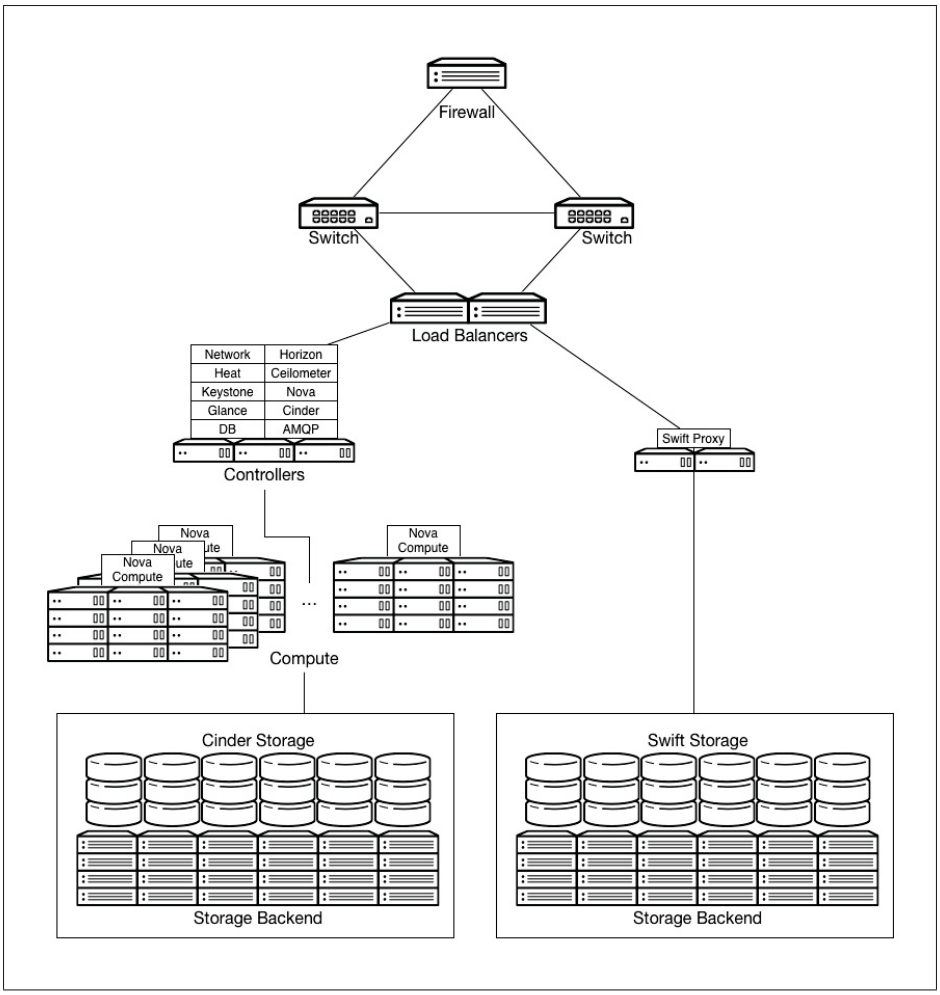


Figure 4.5 Proposed architecture for a general purpose Openstack deployment to run a webserver application
Taken from Openstack (2017a)

However in this thesis tests, our setup is composed of only two Openstack nodes which one acts as both controller and compute (with MULQA components) and the other one is a compute node. Figure 4.6 shows the Openstack deployment in this setup.

Each Physical Openstack node is a Cisco UCS B200 M3 Blade Server System with two Intel Xeon Processor E5-2660 v2 CPUs and 8x16GB DDR3 (M393B2G70DB0-CMA) RAM. Each of the Xeon CPUs have 10 Cores and 25MB of L2 Cache and working by default in 2.2GHz frequency. Also each server has 500GB SCSCI 1500rpm HDD for storage. All servers run Ubuntu 14.04 with Linux kernel 3.16, KVM hypervisor QEMU 2.0.0. Openstack version used

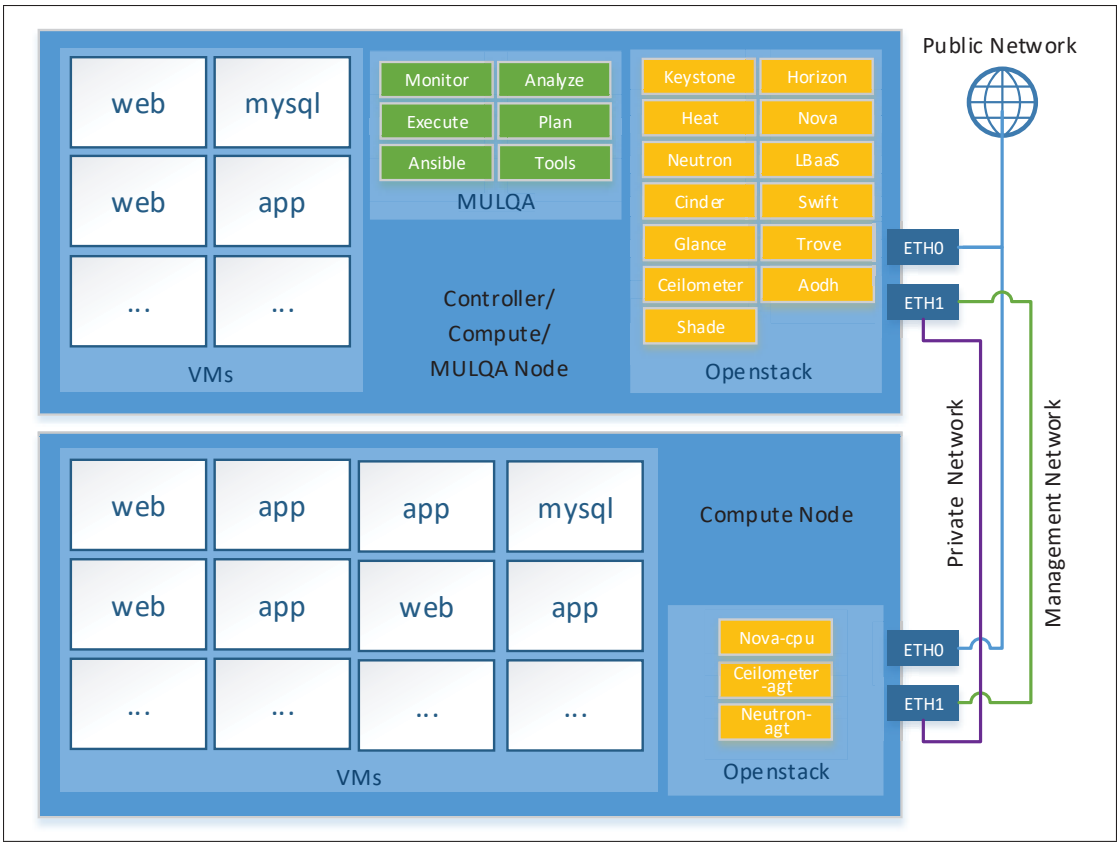


Figure 4.6 A two-node Openstack deployment for MULQA

in this setup is Mitaka. Devstack has been used to to quickly bring up the OpenStack environment with the multi-node setting.

Each compute node has two physical network interfaces. In this setup we have three main networks: Public, Private and Management. The public network is to expose instances (VMs) on floating IPs to the rest of the world and also, make OpenStack services APIs public. This network is a single class C network from the cloud owner's public network range and is isolated from private networks and management network.

The private network is connected to the compute nodes; all the bridges on the compute nodes are connected to this network. This is where instances exchange their fixed IP traffic. If VlanManager is in use, this network is further segmented into isolated VLANs, one per project existing in the cloud. Each VLAN contains an IP network dedicated to this project and connects

virtual instances that belong to this project. If a FlatDHCP scheme is used, instances from different projects all share the same VLAN and IP space.

The management network is used to exchange internal data between components of the Openstack cluster, as well as MULQA. This network must be isolated from private and public networks for security reasons. This network is a single class C network from a private IP address range (not globally routed).

As shown in Figure 4.6, VMs (instances) related to the use-case stack are distributed between compute nodes. Our web stack is composed of three kinds of instances: Web, App and MySQL. These instances will be explained in detail in Section 4.3.2. The Controller node runs most of the Openstack services which are relavant to our use-case. On the other hand, the Compute node only runs nova-cpu, neutron-agent and ceilometer-agent and essential openstack services like rabbitmq.

While in Figure 4.6, the internal connections between components are not shown, Figure 4.7 shows some of these connections between Openstack components used in the use-case scenario. VM in this figure relates to each of Openstack instances like: Web, App and MySQL. Horizon, Keystone and Ceilometer are connected to all other openstack components (colored yellow).

4.3.2 Three-tier web application

The web application use-case which is a LAMP stack, is deployed in three-tiers:

- Web presentation tier: cluster of web servers that will be used to render either static or dynamically generated content for the web browser.
- Application tier: cluster of application servers that will be used to process content and business logic.
- Database tier: cluster of database servers that store data persistently.

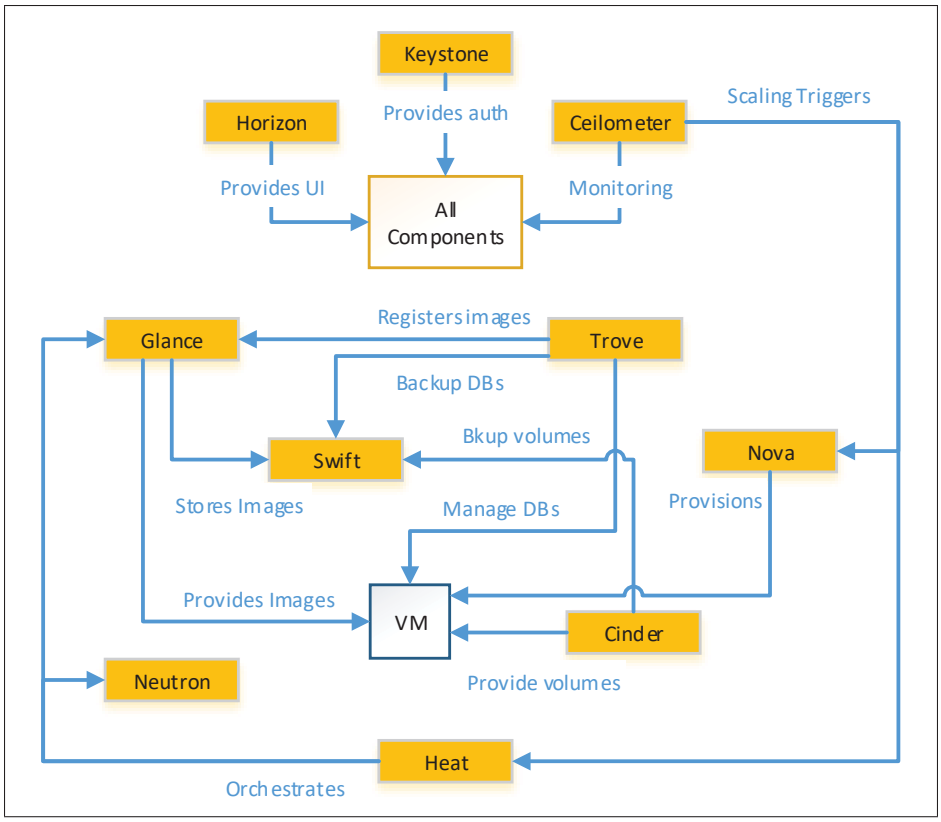


Figure 4.7 Openstack component connections and its workflow in the three-tiered web app use-case

The layered architecture for this use-case is illustrated in Figure 4.8. In this architecture, the end-user who wants to connect to the Wordpress application, sends requests to a load balancer's Virtual IP (VIP) and accesses the Web tier, then the traffic will be directed to the App tier cluster through another load balancer and then the App tier, whenever it needs, will access the Database tier. Each tier in Figure 4.8 consists of some instances in a scaling group. On each instance of the Web tier, Apache webserver with mod_proxy is installed and App tier instances will have Apache webserver, PHP, MySQL Client and WordPress. Also, the database server is running MySQL. Figure 4.9 shows the homepage of this use-case while accessed via web browser.

Spawned instances of Web, App and Database, can be configured in different ways. These instances can be boot from pre-installed and configured images or be configured on-the-fly.

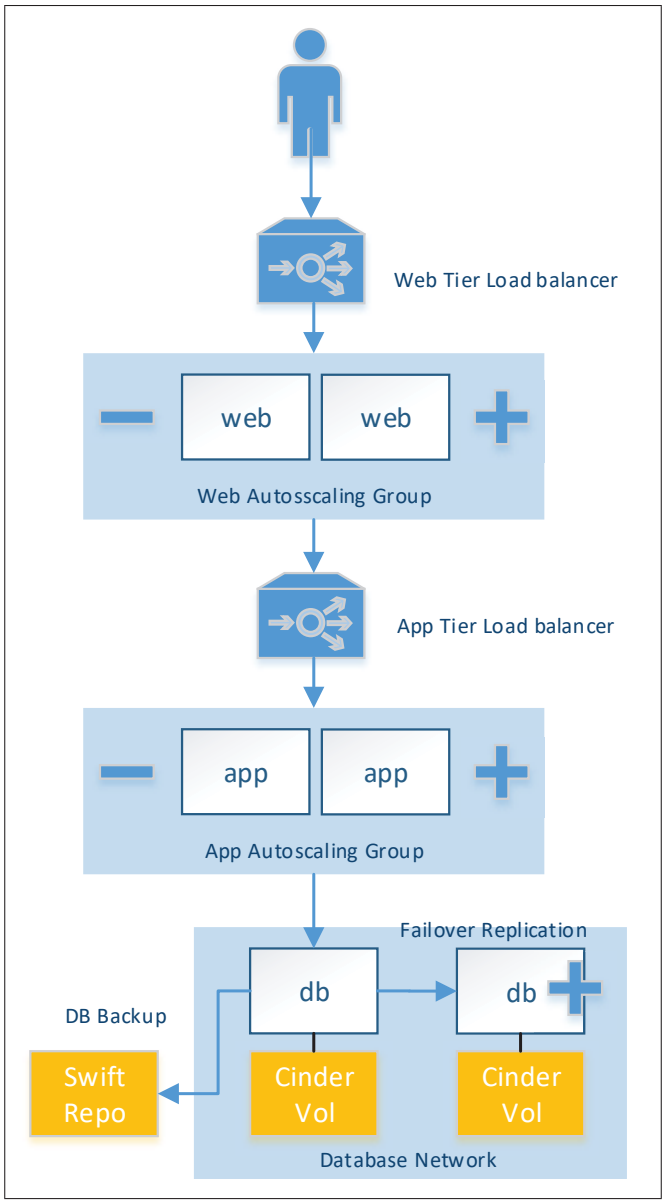


Figure 4.8 Layered architecture of the three-tiered web application use-case

Pre-configured images will go functional faster, but they need to be patched, updated and licensed time to time. On the other hand, on-the-fly configuration of instances is more flexible, and dynamic, but the instance will get ready slower. This thesis has used Cloud-config and cloud-init to configure and install all needed packages in each instance.

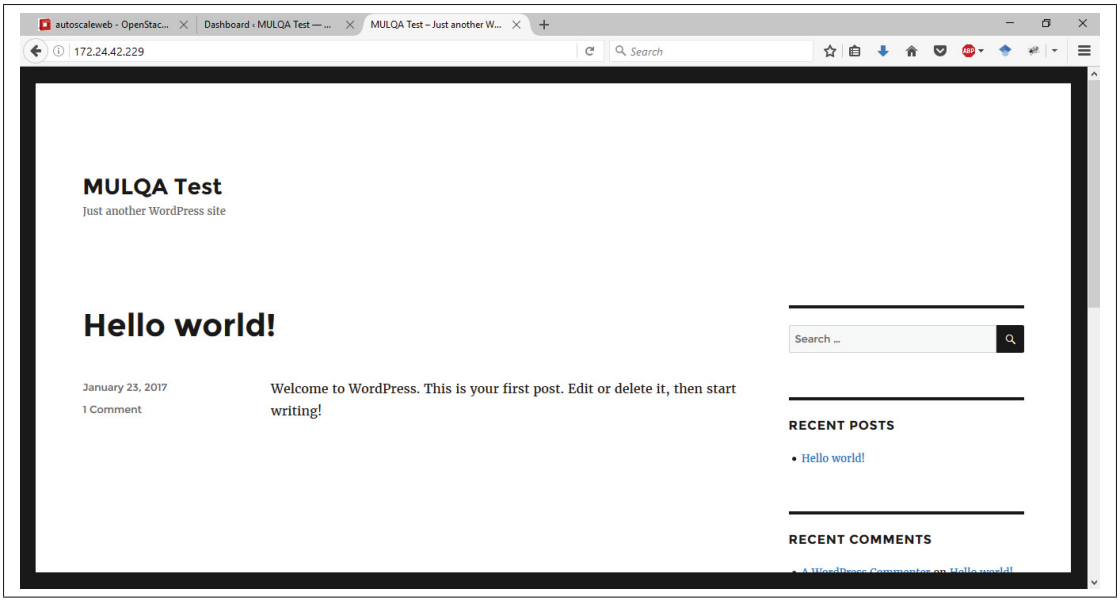


Figure 4.9 Wordpress homepage of the use-case in a web browser

Auto-scaling (using Ceilometer) on each scaling group is desirable to automatically respond to unexpected traffic spikes (by scaling up) and resume to normal operation when the load decreases (by scaling down). Two load balancers are required to equally distribute load. The first load balancer distributes the web traffic at the presentation tier. A separate load balancer is required to distribute the load among the application servers. The database tier uses a master/slave RDBMS configuration. Data is kept in persistent block storage and backed-up periodically. For security reasons, using security groups, a set of firewall rules are enforced at each tier.

The whole use-case including its instances for each tier, networks, routers, security groups and etc. are deployed as a Heat stack on top of our Openstack deployment mentioned in Section 4.3.1.

The Heat template for this use-case, uses a nested structure, with a primary YAML file, which uses four nested files. The templates were tested using Mitaka release of OpenStack, and Ubuntu server 14.04 (Trusty).

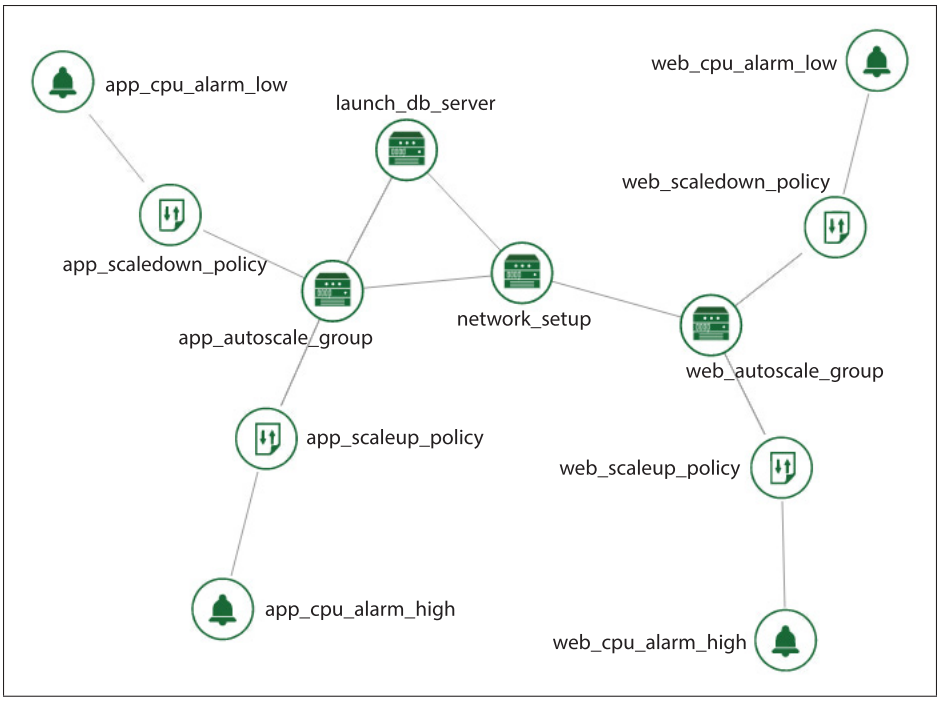


Figure 4.10 Topology of the deployed use-case Heat stack

In this thesis, the use-case deployment uses LBaaSv1 due to its compatibility and stability. However, the newer LBaaSv2 can be used with Octavia project and newer Openstack releases which has been explained in Box (2016). By default, there is round robin, least connections, or random policies for the load balancing method in Openstack. Round robin has been used in our deployment. Moreover, haproxy has been used as the load balancer provider, and the health monitor for the server pools uses TCP checks.

In this deployment the network is configured to filter unnecessary traffic at different tiers. Neutron is used to create multiple subnets, one for each tier: a web subnet, an application subnet, and a data subnet. Neutron routers are created to route traffic between the subnets. Figure 4.11 shows the network topology for the use-case deployment.

As illustrated in Figures 4.7 and 4.8, the use-case stores the data in block and object storages. Cinder volumes are persistent block storage devices that act like physical external hard drives which can be mounted and attached to an instance. In MULQA use-case, a Cinder volume is attached to the Database VM to increase the data persistency in the database tier. In this

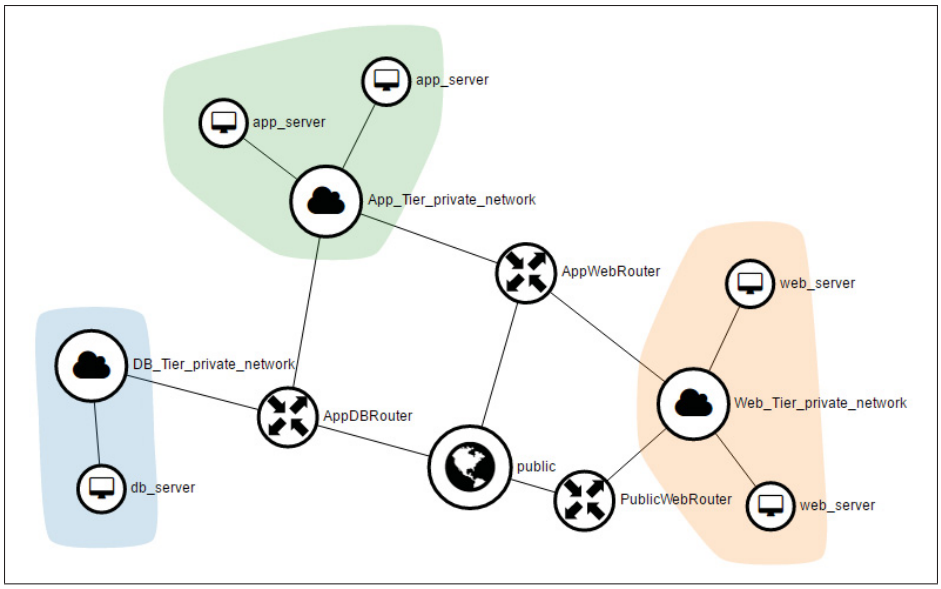


Figure 4.11 Network topology of the deployed use-case stack

architecture, when a Database VM failure happens, a new VM can be created and the Cinder volume can be re-attached to the new VM.

In addition, Swift is used to provide object storage with highly availability, eventually-consistency and distributed characteristics. Unlike Cinder blocks, which need to be connected to an instance, object storage is independent of the instances and are accessed through REST service calls. In our use-case, the object storage is used for storing static files like images and videos used by the web application and also for the database backups.

4.4 Experimental Results

To show a sample utilization of MULQA, an experiment is designed which is focused on the performance quality attribute. The goal of this test is to show that without a multi-layer system like MULQA for quality control (both sensing and actions), a common cloud system with Openstack is not be able to fully guarantee the performance. However, MULQA is able to easily solve the issue by sensing the metrics in one layer and changing the system in another (or same) layer.

Note that designing a test with mix of all QAs and layers at the same time, and discuss the results, due to its complexity and size, is beyond the scope of this thesis.

4.4.1 Test scenario

The test scenario is utilizing the three-tier web use-case deployment illustrated in Figure 4.8. In the test, the user sends HTTP requests to the public IP (first load balancer's VIP) of the stack and in the same time, all layers of the cloud are monitored.

The HTTP requests are sent using Apache Bench (a tool for benchmarking an HTTP server). According to Apache (2016), this tool, especially shows how many requests per second an Apache web server installation is capable of serving. Using Apache Bench, the streams of 5000 HTTP requests are sent with different concurrent request numbers. After sending each stream, there is a 20s halt (sleep) period without requests. These requests are sent to our three-tiered Heat stack. Figure 4.12 illustrates these requests over time. Concurrency numbers are in order: 10, 20, 30, 40, 50, 100, 200, 500, 1000. The test script used here, can be found in Appendix I. In this test, the metrics are monitored in all layer with the normal state frequency of 0.2Hz.

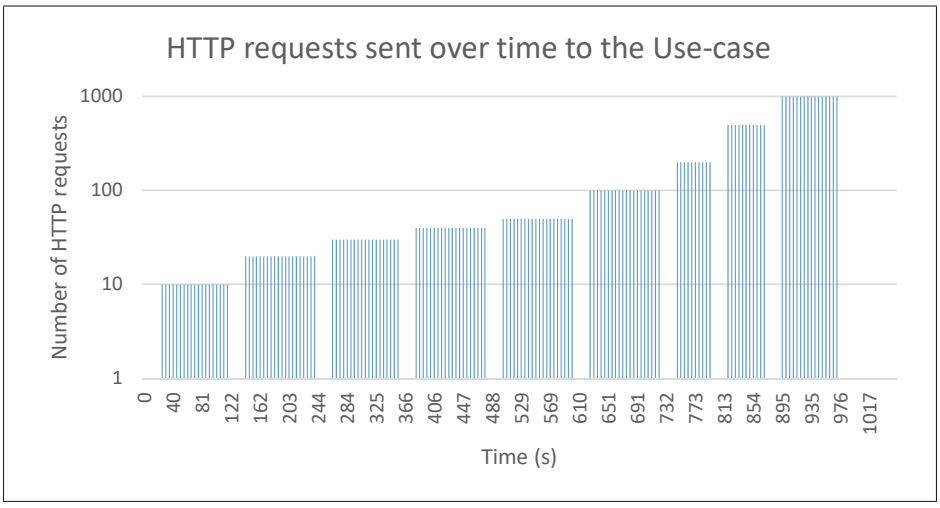


Figure 4.12 Chart for HTTP requests sent to the use-case stack over time

Figure 4.13 shows the available VMs (instances) and their placement on the nodes, while this test was running before and after controlling by MULQA. As this figure shows, there is five instances running in the beginning: $web1$, $web2$, $app1$, $app2$ and db. These instances are located on two physical nodes: $c1-2$ and $c1-3$ which their specifications are described in Section 4.3.1. $c1-3$ is the controller/compute/MULQA node, and $c1-2$ is the compute node.

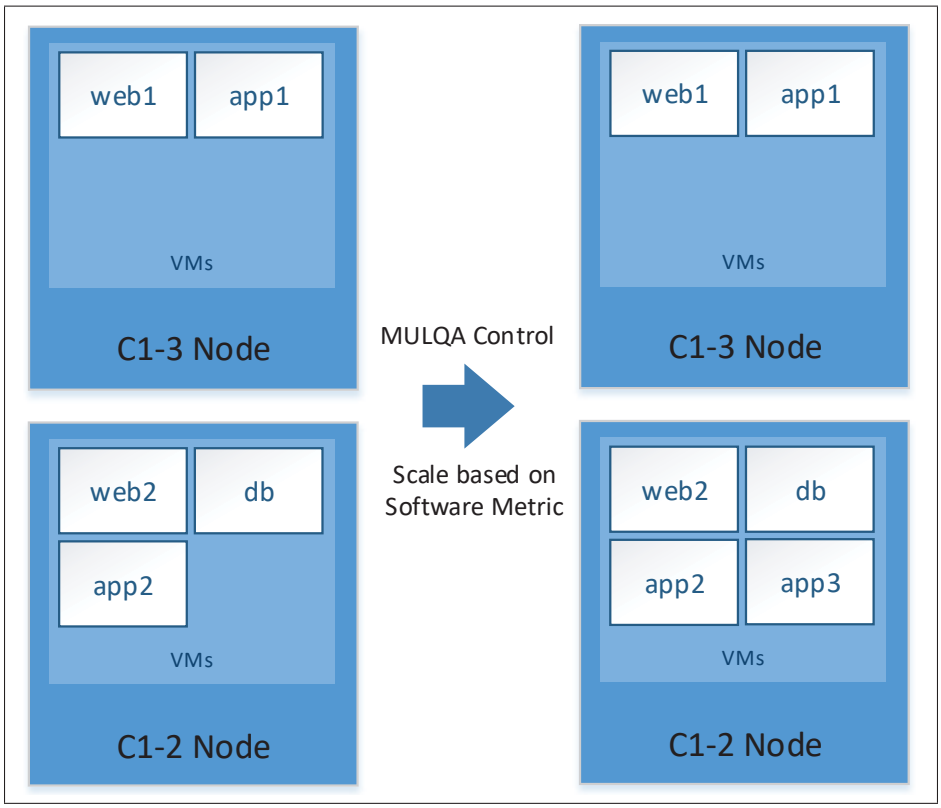


Figure 4.13 Use-case VM placement on the physical nodes during the test

4.4.2 Problem

Table 4.1 shows the results of the test on the use-case. This table shows number of successful requests (# S Reqs), success rate (S Rate), average response time (Avg Resp), maximum response time (Max Resp) and time taken for each stream (T Taken) for the associated request streams with the request concurrency number (Req Con #). As this table illustrates, running the

test on the web stack without MULQA control, has request failure on high concurrent requests (200 and more) and as this number increases, the success rate decreases. This test totally took 1017 seconds (including 20s sleep periods).

Table 4.1 Apache Bench results summary for Web stack without MULQA

Req Con #	# S Reqs	S Rate (%)	Avg Resp (ms)	Max Resp (ms)	T Taken (s)
10	5000	100	192.651	715	111.07
20	5000	100	390.132	1409	97.533
30	5000	100	595.365	2082	99.228
40	5000	100	791.815	3012	98.977
50	5000	100	1000.678	3951	100.068
100	5000	100	2047.614	8254	102.381
200	3789	75.78	3177.992	14031	79.45
500	2967	59.34	7548.273	32921	75.483
1000	2581	51.62	14651.42	66235	73.257

In this section, some of the monitored performance metrics are illustrated. By investigating these charts, we try to find a solution to fix the problem using MULQA. Note that, in this section MULQA control is not enabled and MULQA monitored data is used in the charts.

Figures 4.14 to 4.18 show the monitored data for this test. Also, Table 4.2 lists the illustrated metrics in these figures and their associated layers. In this section, to make the comparison of requests easier, Apache mod_proxy requests in the Web tier and Mysql requests in the database tier are considered in the Software layer, however, in some references they are part of the Platform layer.

As shown in the left-hand side of Figure 4.13, at the beginning, we have two instances of *web*, a *db* instance and two instances of *app* in our stack. Figure 4.14 illustrates the software requests received by these instances. In the diagrams in this section (results), we just show the results of the *web*1, *web*2, *app*1 and *db* VMs. As we know, because *web*1 and *web*2 are behind the load balancer, the number of requests received by each of them should be almost half of the requests sent to the stack (shown in Figure 4.12). But, due to the packet storm tests, when the

Table 4.2 Monitored metrics discussed and their associated layers

Layer	**Metrics**
Software	Number of web, app and db requests received
Platform	Number of threads on the operating system
Instance	CPU utilization for VM
Instance	RAM utilization for VM
Physical	CPU utilization for Node

connections to the software ports are established, it will take some amount of time for the server to close them, and this increases the counted number of received requests in the monitoring. This figure also shows that number of web tier requests is almost balanced for *web*1 and *web*2 and the number of requests in *db* instance is lower than the instances in other tiers.

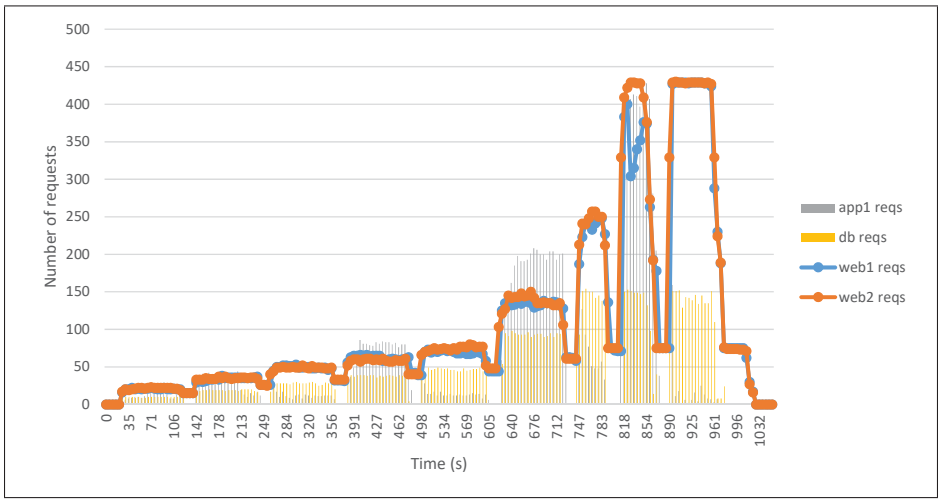


Figure 4.14 Chart for requests received in the software layer on VMs over time while MULQA is not controlling

Figure 4.15 shows the CPU utilization for the VMs, which is an instance layer metric. If we look into each VM's chart solely, it shows rises in CPU utilization almost similarly, in the stream periods, regardless of the concurrency number. From this chart, we can conclude that the most processing stress in the tiers, is on the App tier which runs PHP and Wordpress. So in order to solve the request failure problem, this tier should be targeted.

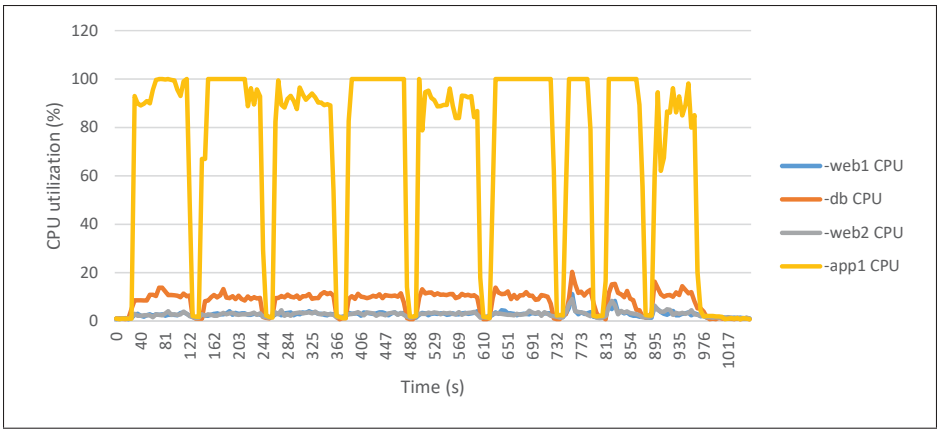


Figure 4.15 Chart for VMs CPU utilization over time

Figure 4.16 illustrates the RAM utilization for VMs which is an instance layer metric. This figure shows that, streams of requests don't have a major effect on the RAM utilization of the VMs, except some rises for $app1$ after we start streams of 100 and 500 concurrent requests.

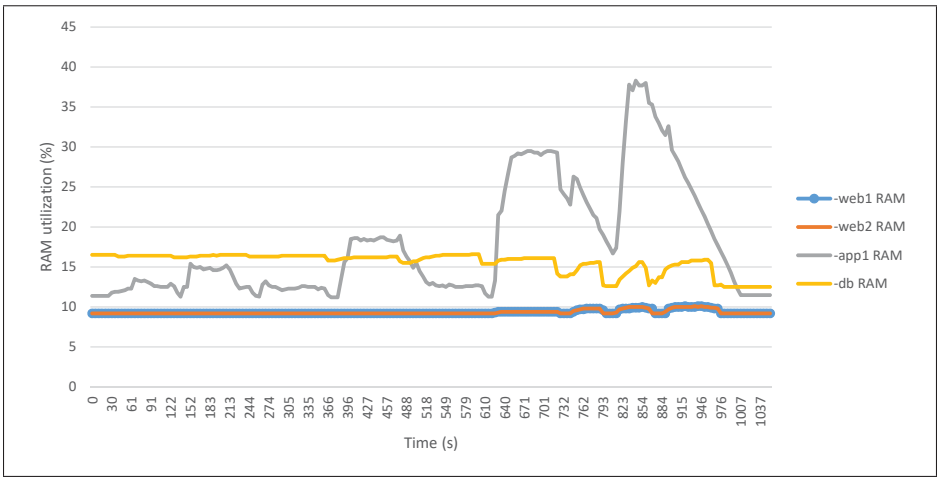


Figure 4.16 Chart for VMs RAM utilization over time

Figure 4.17 shows number of running threads on the Linux operating system of the VMs. This metric is a platform layer metric. This figure shows that, number of threads increases clearly for $app1$, for some streams, but there is no significant trace of the streams that cause request failure (200 and more concurrent requests).

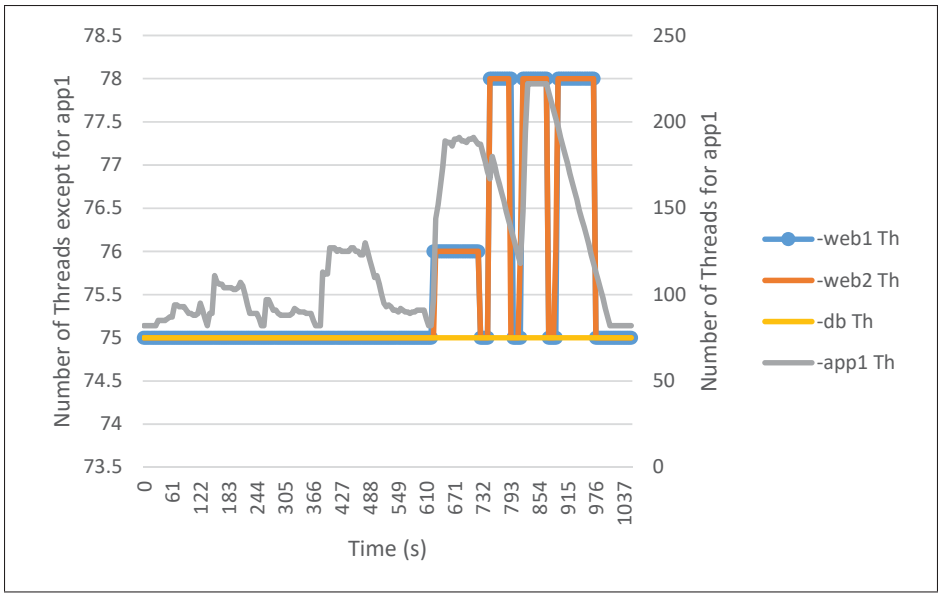


Figure 4.17 Chart for number of operating system threads for VMs over time

Figure 4.18 illustrates the CPU utilization for the physical nodes: $c1-3$ and $c1-2$. As shown in Figure 4.13, two instances ($web1$ and $app1$) are running on $c1-3$ node, which is an Openstack controller node too. So, when the node is not receiving requests, the CPU utilization is about 6% and streams of requests to the web stack, increases both node's CPU utilization. Also, three instances ($web2$, $app2$ and db) are located on $c1-2$, and the CPU metric for this instance has higher jumps on the request streams.

As previously shown in Table 4.1, there are failed requests when concurrent request number is equal or more than 200; However, other streams are served successfully. If we define throughput as the number of concurrent requests that system can handle, throughput for this test is less than 200. The deployed Heat stack for the three-tiered web was not able to detect the failure in the software layer in order to scale the web tier effectively or trigger other actions in order to serve high requests. The scaling alarm, in this test is triggered when the average CPU utilization over 10 minutes (i.e. a common trigger metric in Openstack autoscaling applications) of the scaling group VMs passes 50%. As illustrated in Figure 4.15, this metric for the *web* scaling group is not passing this threshold, so the scaling didn't happen for this tier. Note that,

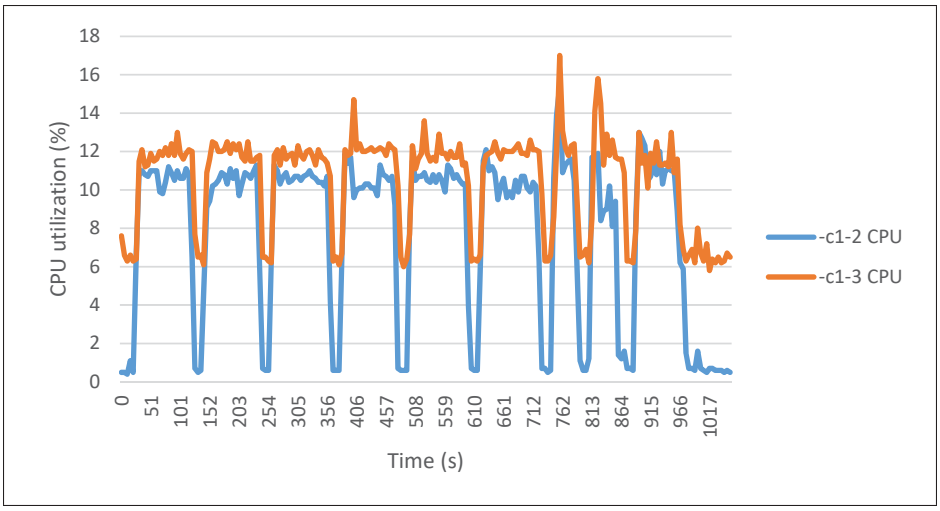


Figure 4.18 Chart for CPU utilization for physical nodes over time while MULQA is not controlling

in this chart, there is no specific threshold to be used to distinguish the failure states (when we have 200 or more requests) and normal states. As a result, changing the alarm threshold in the use-case stack to another number rather than 50% does not solve the problem. Furthermore, autoscaling in App tier is not efficient, because having concurrent requests as low as 10, the CPU utilizations reaches 100%. Investigating Figures 4.16 and 4.18 leads to the same conclusion. So, normal Ceilometer metrics which are performing in Physical, Openstack and Instance layers can't be used as thresholds for the autoscaling in order to solve the request failure problem. Number of threads for $app1$ metric, which is a platform layer metric monitored by Ansible, may be used to detect the failure periods. But, using the software request numbers is the best option to predict the failure, because due to availability of other unrelated processes in the operating system, number of threads may increase regardless of the failure situation.

4.4.3 Solution using MULQA

Following the discussion in the previous section, it was concluded that predicting the metrics monitored in Physical, Instance and Platform layers, can't detect the request failure. However, by predicting the received requests in the Software layer, the failure can be predicted. For

example, an appropriate action can be triggered when the number of received requests in $app1$ reaches 150.

This metric is monitored by an Ansible call and uses a simple netstat command which checks for the established connections to the software's port (80 in our use-case). MULQA uses this metric from Software layer to predict the failure. Next, it prevents the failure by triggering an autoscale signal in the App tier (stressed tier) of the use-case stack. In this section, we discuss the results of the similar tests, on a the same cloud deployment, with MULQA enabled. All the configured parameter in this deployment of MULQA are listed in Appendix II. When the system is about to fail, MULQA adds a new app instance (e.g. $app3$) which helps to prevent the failure. This VM is shown in right hand side of the Figure 4.13.

In our deployment, it took 25s to create and configure a new app instance. This period can be decreased by using a pre-installed image, instead of running a cloud-config script on the boot of the instance. The first autoscale is supposed to be triggered at about 640s, while we have set "the number of received requests in $app1$ is more than 150" as the plan to do so. The allocation and installation of the VM will happen in the warning state, while the transition will happen on higher number of received requests, which is 200.

Table 4.3 shows the results for the test on the use-case, while MULQA is fully operating. This results show that, by using MULQA, the success rate will reach 100% even for high request concurrency numbers. MULQA improved this metric by 32%, 69% and 94% for 200, 500 and 1000 "Req Con #" in order. Being able to handle 1000 concurrent requests, means the throughput is at least 1000, which compared to "without MULQA" case (with less than 200cr throughput), shows more than 400% improvement.

As shown in right hand side of Figure 4.13, by checking Openstack logs, it was found that $app3$ is added to $c1-2$ and got ready successfully, and the requests which caused failure previously, are handled. Furthermore, as the load increases to 500 and 1000, more instances are added to the App tier. Figure 4.19 shows CPU utilization for $c1-2$ physical node in both scenarios: before and after MULQA control enabled. This figure also shows the average value of this

Table 4.3 Apache Bench results summary for Web stack with MULQA

Req Con #	# S Reqs	S Rate (%)	Avg Resp (ms)	Max Resp (ms)	T Taken (s)
10	5000	100	191.631	721	96.325
20	5000	100	390.145	1409	99.222
30	5000	100	594.372	2065	99.726
40	5000	100	789.789	3039	99.036
50	5000	100	1001.707	3944	101.368
100	5000	100	2047.59	8217	103.276
200	5000	100	2000.441	14079	50.011
500	5000	100	5641.267	45070	56.413
1000	5000	100	17487.48	68005	87.437

metric, in both scenarios. Average CPU utilization for $c1-2$ in enabled MULQA, is greater than before MULQA control scenario by 0.77%. As higher CPU utilization, causes higher energy consumption, we can conclude that using MULQA slightly increases energy consumption which is a trade-off for gaining better performance. In addition, CPU and RAM utilization of the MULQA processes (with the settings described in Appendix II) were measured separately and results show that running MULQA processes will add 0.06% CPU utilization and 2576KB RAM utilization overhead.

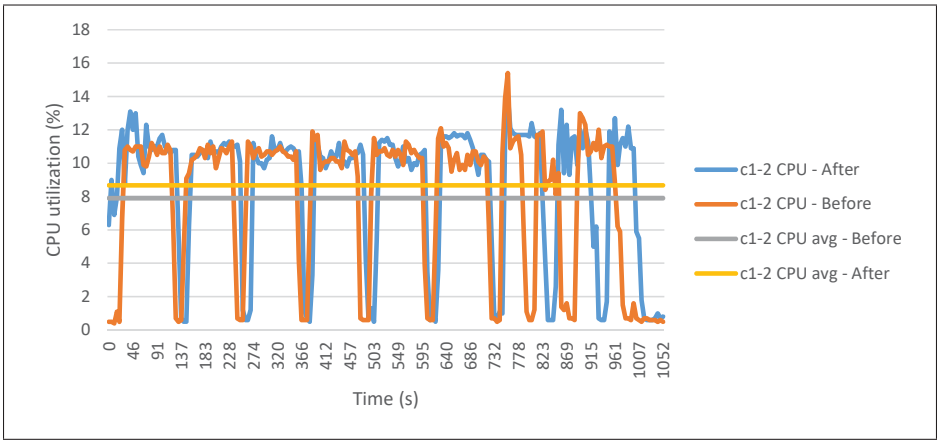


Figure 4.19 Chart for CPU utilization of Node c1-2 over time before and after MULQA control

Also in Figure 4.20, number of received requests by $app1$, while MULQA is controlling the quality (performance in this case) is compared to the previous scenario's data (before MULQA control). This figure shows that, $app1$ is receiving more requests to handle, while the concurrency number is more than 200. Probably the reason for having request failures previously was a crash in $app1$ (and probably $app2$ as well) server on high requests.

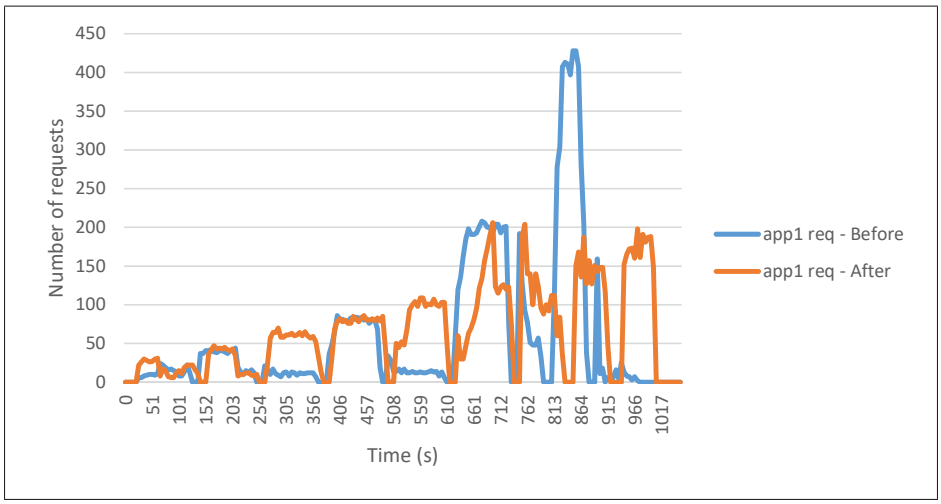


Figure 4.20 Chart for number of received requests by app1 over time before and after MULQA control

This was a simple example in improving the results of a concurrency test for a real use-case application using MULQA. Note that, this system can be used to handle the complex situations by using more advanced Analyze and Plan modules.

4.4.4 Discussion

In this chapter, we used MULQA to improve the performance quality of a practical cloud-based use-case. The parameters of MULQA in the tests, were configured to monitor, analyze and plan for the performance metrics in all layers of the cloud. Although the tests were focused on a single QA, MULQA is able to target multiple QAs simultaneously. Table 4.4 gives an idea about how MULQA can be used to cover a diverse mix of QAs and layers in both sensing and

actions. For instance, first row of this table can be explained as the following: we know there is a direct relationship between CPU frequency of a node, and its energy consumption. So, we can configure $(Physical, Energy, CPUUtilNodeN)$ metric to be monitored, and controlled through running DVFS actions like $IncreaseNodeNCPUFreq$ and $DecreaseNodeNCPUFreq$. The metric in this case is a *Physical* layer metric and an *Energy* QA metric. This metric is sensed by Ceilometer and its attributed action (which is a *Physical* layer action) is executed by Ansible.

Table 4.4 Sample multi-QA setting explanation for MULQA

Metric	**Monitor Description**	**Action Decription**	**Action Layer**
(Physical, Energy, CPUUtilNodeN)	Check CPU utilization of node N using Ceilometer and calculate energy consumption	Run DVFS using Ansible to increase or decrease the CPU frequency in Node N	Physical
(Openstack, Scalability, #InsBy#InsQuota)	Check the number of available instances divided by the quota number of instances	Send warning alarms using Adoh to add more nodes in the Physical layer	Openstack
(Instance, Performance, CPUUtilVMGrX)	Check average CPU utilization of the scaling group X	Scale-up and Scale-down scaling group X using Heat	Openstack
(Platform, Reliability, ApaAndDbHealth)	Check Apache servers and Db servers health	Reset the failed service using Ansible	Platform
(Software, Security, WordpressVer)	Check version of Wordpress and its plugins	Update the outdated version using Ansible	Software

Note that, MULQA is not an integration system which serves as a universal hub for connecting different heterogeneous components and technologies for monitoring, analyzing, planning and execution. However, quality managers can feed the framework with their desired algorithms in Python programming language format for each module. By doing this, quality managers will benefit from a complete system which maintains high quality. This system is made of loosely coupled modules which can be easily distributed and scaled to multiple nodes. The interoperability issue which MULQA is trying to resolve is related to the heterogeneous components that

are inside the cloud controlled by MULQA. MULQA is able to sense and control these components in different layers, through standard protocols. On the other hand, sometimes using the built-in monitoring and control APIs of the cloud systems or other third-party monitoring platforms may ease the installation and management process and increase the efficiency. For example in this thesis we have used Ceilometer module of Openstack to monitor some components of the cloud. MULQA is able to easily integrate with a lot of these systems too (which are supported by Ansible). However, having multiple monitoring softwares can create new challenges such as difficulty in modification and maintenance, decreasing the interoperability and security of the cloud, and increasing the cost.

CONCLUSION AND RECOMMENDATIONS

In this thesis, an autonomic framework named MULQA was proposed. MULQA is a Multi-Layer Quality-Aware system to improve the quality of cloud systems. Quality attributes targeted by MULQA could comprise of performance, availability, reliability, cost, energy-efficiency, etc. Also, quality metrics for a targeted quality attribute can be defined, modeled, monitored, predicted and controlled in all layers of the cloud, including physical, infrastructure, platform and software layer.

By enabling autonomic fine-grained quality management in all layers, MULQA unlike previous systems in this field, was able to bring full quality control to the cloud, including to the software layer. Being a modular framework, MULQA provides generic functionalities and modules that can be selectively changed by additional user-written code, which can be used by future quality management researchers to test their proposed algorithms for Monitor, Analyze, Plan and Execute modules. Furthermore, the modular design of MULQA can be easily scaled and deployed in a distributed fashion to fit the requirements.

Modules of MULQA include: Monitor, Analyze, Plan and Execute. First Monitor module collects, aggregates, filters and reports quality metrics from managed resources through Sensor agents and transfers this information to the next module for further analysis. Then, Analyze module is used to predict the performance metrics and their statuses. If the probability of predicted metrics violates the performance metrics threshold, depending on the SLA and QoS sensitivity of the applications, attributed events will be generated. These events will trigger the state changes in the finite state machine used by Plan module. This module prepares the cloud for the possible violation and when the quality is about to violate, it triggers appropriate actions (through Execute module), again in all layers of the cloud, to brings back the cloud to the normal state.

This framework also provides a practical use-case scenario to evaluate the performance of the quality management system. The implemented use-case of MULQA was a three-tier Web application which runs Wordpress, one of the most common applications used on the Internet. This use-case is implemented with Heat project of OpenStack. Openstack was selected to be the cloud middleware, and it was deployed in a multi-node architecture. We tested our implementation of MULQA to show that MULQA is able to improve the performance of the use-case by monitoring different quality metrics in all layers of our cloud. Experimental results of the tests show that MULQA could increase the success rate of requests to the use-case by 32%, 69% and 94% for request concurrency numbers of 200, 500 and 1000 in order. Moreover, throughput was been improved by 400%, while having low impact on the energy consumption.

Throughout this thesis, We were able to:

- Identify the relevant quality attributes in cloud systems and their metrics and challenges in managing each of them by surveying the existing literature. These QAs are performance, availability, reliability, security, scalability and elasticity, interoperability, cost, and energy-efficiency.
- Review the state-of-the-art in the area of quality in cloud computing and find the challenges of quality management in the cloud systems. These problems are categorized to "software engineering based" and "resource management based". MULQA aims to solve the challenges rooted in interoperability, scalability, multi-QA and multi-layer, and autonomicity of these systems. Also, it proposes the integration of SLA negotiation system with the fine-grained quality management system to solve some issues in SLA management.
- Introduce a quality model with notations and mathematical models for cloud systems and design an autonomic Multi-Layer Quality-Aware system to monitor, model, predict and control the different quality attributes in cloud systems. Also create an FSM for the control mechanism which consists of Normal, Warning, Transition and Error states.

- Build a modular and customizable framework for the designed solution, and develop a cloud-based three-tier Web application that can be used to evaluate the proposed system design on a multi-node Openstack testbed.

- Run tests on the use-case and Configuring MULQA to improve the performance QA of a system. Experimental results of the tests show that implemented MULQA can increase the success rate of requests to the use-case by 32%, 69% and 94% for request concurrency numbers of 200, 500 and 1000 in order. Moreover, throughput has been improved by 400%, while having a low impact on the energy consumption.

The main challenges targeted by MULQA and the solutions provided by this thesis are listed in the following:

- *Challenge:* Current quality management systems are dedicated to specific service models, mostly IaaS and SaaS. Improving different quality attributes with such systems leads to multiple quality management systems at various layers of a cloud deployment, which may bring interoperability issues and cause inefficiency in monitoring and control mechanisms. *Solution:* MULQA design is a unified quality control system for all layers of the cloud and multiple quality attributes with customizable and loosely coupled modules.

- *Challenge:* Quality analysis, prediction, and assurance in cloud platforms has become significantly complex due to performance heterogeneity and resource isolation mechanisms. *Solution:* As Petcu *et al.* (2013) indicates, MULQA by using automated quality management methods (which is based on ACCS), can leverage the high programmability of hardware and software resources in the cloud and facilitate solving this challenge.

- *Challenge:* SLAs by specifying quality targets and economical penalties for SLA violations, increase the complexity of finding optimal tradeoff.

Solution: MULQA proposes the integration of the SLA negotiation system with the fine-grained quality management system to solve some of these issues in SLA management. However, the implementation and test of the proposed idea for this section was not done.

- *Challenge:* Lack of application management and the lack of approaches to cloud deployment optimization services with various quality metrics such as performance and cost.
  Solution: MULQA by covering different QAs including performance, cost and even custom-defined QAs, builds a quality management framework. This framework has been implemented and tested in practical use case scenario.

- *Challenge:* Proposed literature architectures need installations of extra modules per VM, per Node and per application on the cloud system, which decreases the scalability and interoperability of the architecture.
  Solution: By leveraging standard communication protocols like SSH and HTTP and Ansible calls for sensing and executing commands on the endpoints, MULQA overcomes this challenge.

Briefly, we believe that using MULQA alongside today's heterogeneous cloud systems, and adoption of its approach towards quality, will facilitate the quality management process and can improve the quality of complex clouds more.

Future Research Direction

Despite contributions of the current thesis in building a framework for quality management of the cloud systems, there are a number of open research challenges that need to be addressed to further advance the area. Such are listed below:

- Being an open framework, all modules of MULQA are open to more advanced mechanisms for monitoring, data analysis, prediction, filtering and decision making.

- Finding proper mechanisms to specify the thresholds automatically and efficiently
- Evaluating and testing the system with other settings and in other applications
- Develop and implement the SLA negotiation modules
- Design and implement the auto-healing mechanism

APPENDIX I

MULQA TEST SCRIPT USED IN THIS THESIS

```
1  #!/usr/bin/env bash
2  echo 'sleep for 20s...'
3  sleep 20
4  echo 'stream with 10 cr# starts:'
5  ab -n 5000 -c 10 http://172.24.42.229/
6
7  echo 'sleep for 20s...'
8  sleep 20
9  echo 'stream with 20 cr# starts:'
10 ab -n 5000 -c 20 http://172.24.42.229/
11
12 echo 'sleep for 20s...'
13 sleep 20
14 echo 'stream with 30 cr# starts:'
15 ab -n 5000 -c 30 http://172.24.42.229/
16
17 echo 'sleep for 20s...'
18 sleep 20
19 echo 'stream with 40 cr# starts:'
20 ab -n 5000 -c 40 http://172.24.42.229/
21
22 echo 'sleep for 20s...'
23 sleep 20
24 echo 'stream with 50 cr# starts:'
25 ab -n 5000 -c 50 http://172.24.42.229/
26
```

```
27 echo 'sleep for 20s...'
28 sleep 20
29 echo 'stream with 100 cr# starts:'
30 ab -n 5000 -c 100 http://172.24.42.229/
31
32 echo 'sleep for 20s...'
33 sleep 20
34 echo 'stream with 200 cr# starts:'
35 ab -n 5000 -c 200 http://172.24.42.229/
36
37 echo 'sleep for 20s...'
38 sleep 20
39 echo 'stream with 500 cr# starts:'
40 ab -n 5000 -c 500 http://172.24.42.229/
41
42 echo 'sleep for 20s...'
43 sleep 20
44 echo 'stream with 1000 cr# starts:'
45 ab -n 5000 -c 1000 http://172.24.42.229/
```

APPENDIX II

MULQA PARAMETERS SET IN THE TESTS

$$L = \{l_1, l_2, l_3, l_4, l_5\} \sim \{Physical, Openstack, Instance, Platform, Software\}$$

$$Q_1 = \{q_1^1\} \sim \{(Physical, Performance)\}$$
$$Q_2 = \Phi$$
$$Q_3 = \{q_1^3\} \sim \{(Instance, Performance)\}$$
$$Q_4 = \{q_1^4\} \sim \{(Platform, Performance)\}$$
$$Q_5 = \{q_1^5\} \sim \{(Software, Performance)\}$$

$$M_{1,1} = \{m_1^{1,1}, m_2^{1,1}\} \sim \{(Physical, Performance, CPUUtilNodeC12),$$
$$(Physical, Performance, CPUUtilNodeC13)\}$$

$$M_{3,1} = \{m_1^{3,1}, m_2^{3,1}, m_3^{3,1}, m_4^{3,1}, m_5^{3,1}, m_6^{3,1}, m_7^{3,1}, m_8^{3,1}\} \sim$$
$$\{(Instance, Performance, CPUUtilVmWeb1),$$
$$(Instance, Performance, CPUUtilVmWeb2),$$
$$(Instance, Performance, CPUUtilVmApp1),$$
$$(Instance, Performance, CPUUtilVmDb),$$
$$(Instance, Performance, RAMUtilVmWeb1),$$
$$(Instance, Performance, RAMUtilVmWeb2),$$
$$(Instance, Performance, RAMUtilVmApp1),$$
$$(Instance, Performance, RAMUtilVmDb)\}$$

$$M_{4,1} = \{m_1^{4,1}, m_2^{4,1}, m_3^{4,1}, m_4^{4,1}\} \sim \{(Platform, Performance, NumThreadWeb1),$$
$$(Platform, Performance, NumThreadWeb2),$$
$$(Platform, Performance, NumThreadApp1),$$
$$(Platform, Performance, NumThreadDb)\}$$

$$M_{5,1} = \{m_1^{5,1}, m_2^{5,1}, m_3^{5,1}, m_4^{5,1}\} \sim \{(Software, Performance, NumRecReqWeb1),$$
$$(Software, Performance, NumRecReqWeb2),$$
$$(Software, Performance, NumRecReqApp1),$$
$$(Software, Performance, NumRecReqDb)\}$$
$$M = M_{1,1} \bigcup M_{3,1} \bigcup M_{4,1} \bigcup M_{5,1}$$

$$A = \{a_1, a_2\} \sim \{ScaleupVmgroupApp, ScaledownVmgroupApp\}$$

$$T_{n,min}^{5,1,3} = 0, \quad T_{n,max}^{5,1,3} = 150$$
$$T_{w,min}^{5,1,3} = 0, \quad T_{w,max}^{5,1,3} = 200$$

$$\alpha_k^{i,j} = 0.6$$

$$Plan = \{(\{\psi_{wmax}^{5,1,3}\}, \{a_1\})$$
$$(\{\psi_{wmin}^{5,1,3}\}, \{a_2\})\}$$

BIBLIOGRAPHY

Abdelmaboud, A., D. N. Jawawi, I. Ghani, A. Elsafi, and B. Kitchenham. 2015. "Quality of service approaches in cloud computing: A systematic mapping study". *Journal of Systems and Software*, vol. 101, p. 159–179.

Ansible. 2017a. "List of Ansible Modules". <http://docs.ansible.com/ansible/list_of_all_modules.html>.

Ansible. 2017b. "Monitoring Modules of Ansible". <http://docs.ansible.com/ansible/list_of_monitoring_modules.html>.

Apache. 2016. "Apache HTTP server benchmarking tool". <https://httpd.apache.org/docs/2.4/programs/ab.html>.

Ardagna, D., B. Panicucci, M. Trubian, and L. Zhang. 2012. "Energy-aware autonomic resource allocation in multitier virtualized environments". *IEEE Transactions on Services Computing*, vol. 5, n° 1, p. 2–19.

Ardagna, D., G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang. 2014. "Quality-of-service in cloud computing: modeling techniques and their applications". *Journal of Internet Services and Applications*, vol. 5, n° 1, p. 1.

Armstrong, D. and K. Djemame. 2009. "Towards quality of service in the cloud". In *Proc. of the 25th UK Performance Engineering Workshop*.

Baset, S. A. 2012. "Cloud SLAs: present and future". *ACM SIGOPS Operating Systems Review*, vol. 46, n° 2, p. 57–66.

Beach, T., O. Rana, Y. Rezgui, and M. Parashar. 2015. "Governance model for cloud computing in building information management". *IEEE Transactions on Services Computing*, vol. 8, n° 2, p. 314–327.

Bellavista, P., G. Carella, L. Foschini, T. Magedanz, F. Schreiner, and K. Campowsky. 2012. "QoS-aware elastic cloud brokering for IMS infrastructures". In *Computers and Communications (ISCC), 2012 IEEE Symposium on*. p. 000157–000160. IEEE.

Bertoldi, P. 2014. "A Market Transformation Programme for Improving Energy Efficiency in Data Centres".

Bittman, T. J. 2009. "Server virtualization: One path that leads to cloud computing". *Gartner Technology Research, Gartner RAS Core Research Note G*, vol. 171730, p. 2009.

Box, B. 2016. "Using Heat for autoscaling". <http://ibm-blue-box-help.github.io/help-documentation/heat/autoscaling-with-heat/>.

Bruneo, D., F. Longo, and C. C. Marquezan. 2015. "A framework for the 3-D cloud monitoring based on data stream generation and analysis". In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. p. 62–70. IEEE.

Buyya, R., S. Pandey, and C. Vecchiola. 2009. "Cloudbus toolkit for market-oriented cloud computing". In *IEEE International Conference on Cloud Computing*. p. 24–44. Springer.

Buyya, R., A. Beloglazov, and J. Abawajy. 2010. "Energy-efficient management of data center resources for cloud computing: a vision, architectural elements, and open challenges". *arXiv preprint arXiv:1006.0308*.

Calheiros, R. N., E. Masoumi, R. Ranjan, and R. Buyya. 2015. "Workload prediction using ARIMA model and its impact on cloud applications' QoS". *Cloud Computing, IEEE Transactions on*, vol. 3, n° 4, p. 449–458.

Cardosa, M., M. R. Korupolu, and A. Singh. 2009. "Shares and utilities based power consolidation in virtualized server environments". In *Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on*. p. 327–334. IEEE.

Cesare, S. and Y. Xiang, 2012. *Software similarity and classification*.

Chang, A. 2014. "7 Critical Cloud Service Attributes". <http://www.networkcomputing.com/cloud-infrastructure/7-critical-cloud-service-attributes/1852724555>.

Cima, V., B. Grazioli, S. Murphy, and T. M. Bohnert. 2015. "Adding energy efficiency to Openstack". In *Sustainable Internet and ICT for Sustainability (SustainIT), 2015*. p. 1–8. IEEE.

Cloud-init. 2017. "Cloud config in Cloud-init". <http://cloudinit.readthedocs.io/en/latest/topics/examples.html>.

Community, O. 2016. "OpenStack Administrator Guide". http://docs.openstack.org/admin-guide/index.html. [Online; accessed 29-August-2016].

Computing, A. et al. 2006. "An architectural blueprint for autonomic computing". *IBM White Paper*, vol. 31.

Deissenboeck, F., E. Juergens, K. Lochmann, and S. Wagner. 2009. "Software quality models: Purposes, usage scenarios and requirements". In *Software Quality, 2009. WOSQ'09. ICSE Workshop on*. p. 9–14. IEEE.

Dillon, T., C. Wu, and E. Chang. 2010. "Cloud computing: issues and challenges". In *2010 24th IEEE international conference on advanced information networking and applications*. p. 27–33. Ieee.

Dromey, R. G. 1995. "A model for software product quality". *IEEE Transactions on Software Engineering*, vol. 21, n° 2, p. 146–162.

Dudko, R., A. Sharma, and J. Tedesco. 2012. "Effective failure prediction in hadoop clusters". *University of Idaho White Paper*, p. 1–8.

Emam, K. E. 2005. "The ROI from software quality".

Foster, I., Y. Zhao, I. Raicu, and S. Lu. 2008. "Cloud computing and grid computing 360-degree compared". In *2008 Grid Computing Environments Workshop*. p. 1–10. Ieee.

Garg, S. K., S. Versteeg, and R. Buyya. 2013. "A framework for ranking of cloud computing services". *Future Generation Computer Systems*, vol. 29, n° 4, p. 1012–1023.

Garvin, D. A. 1984. "What Does “hltoduct Quality” Really Mean?". *Sloan management review*, p. 25.

Giannetti, F. and K. Owens. 2016. "Enforcing Application SLAs with Congress and Monasca". <https://www.openstack.org/videos/video/enforcing-application-slas-with-congress-and-monasca>.

Gmach, D., J. Rolia, L. Cherkasova, and A. Kemper. 2009. "Resource pool management: Reactive versus proactive or let’s be friends". *Computer Networks*, vol. 53, n° 17, p. 2905–2922.

Gorton, I., 2006. *Essential software architecture*.

Guo, R.-S. and J.-J. CHEN. 2002. "An EWMA-based process mean estimator with dynamic tuning capability". *IIE Transactions*, vol. 34, n° 6, p. 573–582.

Hamilton, J. D., 1994. *Time series analysis*, volume 2.

Hasan, M. Z., E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi. 2012. "Integrated and autonomic cloud resource scaling". In *Network Operations and Management Symposium (NOMS), 2012 IEEE*. p. 1327–1334. IEEE.

Hausman, K. K., S. L. Cook, and T. Sampaio, 2013. *Cloud Essentials: CompTIA Authorized Courseware for Exam CLO-001*.

Henningsson, K. and C. Wohlin. 2002. "Understanding the relations between software quality attributes-a survey approach". In *Proceedings 12th International Conference for Software Quality*. Citeseer.

Heston, K. M. and W. Phifer. 2011. "The multiple quality models paradox: how much ‘best practice’is just enough?". *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, n° 8, p. 517–531.

Hida, T., 1980. *Brownian motion*.

Islam, S., S. Venugopal, and A. Liu. 2015. "Evaluating the impact of fine-scale burstiness on cloud elasticity". In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. p. 250–261. ACM.

ISO, I. 2007. "IEC 25030 Software and system engineering–Software product Quality Requirements and Evaluation (SQuaRE)–Quality requirements". *International Organization for Standarization*.

ISO, I. 2010. *Iec/ieee 24765: 2010 systems and software engineering-vocabulary*. Technical report.

ISO, I. 2011. "IEC 25010: 2011,"". *Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models*.

Jagannathan, S., S. Bhattacharya, and K. Matawie. 2005. "Value Based Quality Engineering". *TickIT International*, , p. 3–9.

Jain, S., M. Khandelwal, A. Katkar, and J. Nygate. 2016. "Applying big data technologies to manage QoS in an SDN". In *Network and Service Management (CNSM), 2016 12th International Conference on*. p. 302–306. IEEE.

Jain, S., R. Kumar, and S. K. J. Anamika. 2014. "A Comparative Study for Cloud Computing Platform on Open Source Software". *ABHIYANTRIKI: An International Journal of Engineering & Technology (AIJET)*, vol. 1, n° 2, p. 28–35.

Jallow, A. 2016. "CLOUD-METRIC: A Cost Effective Application Development Framework for Cloud Infrastructures".

Jansen, W., 2010. *Directions in security metrics research*.

Jiang, Q. 2015. "Open Source IaaS Community Analysis". http://www.qyjohn.net/wp-content/uploads/2015/04/CY15-Q1-IaaS-Community-Analysis.pdf. [Online; accessed 16-August-2016].

Jones, C. and O. Bonsignour. 2012. "The economics of software quality B. Goodwin, ed".

Karacali, B. and J. M. Tracey. 2016. "Experiences evaluating OpenStack network data plane performance and scalability". In *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*. p. 901–906. IEEE.

Kavis, M. J., 2014. *Architecting the cloud: Design decisions for cloud computing service models (SaaS, PaaS, AND IaaS)*.

Kephart, J. O., H. Chan, R. Das, D. W. Levine, G. Tesauro, F. L. Rawson III, and C. Lefurgy. 2007. "Coordinating Multiple Autonomic Managers to Achieve Specified Power-Performance Tradeoffs.". In *ICAC*. p. 24.

Kernel, L. 2016. "Memory Hotplug". <https://www.kernel.org/doc/Documentation/memory-hotplug.txt>.

Khosravi, A., S. K. Garg, and R. Buyya. 2013. Energy and carbon-efficient placement of virtual machines in distributed cloud data centers. *Euro-Par 2013 Parallel Processing*, p. 317–328. Springer.

Kumar, S., V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan. 2009. "vManage: loosely coupled platform and virtualization management in data centers". In *Proceedings of the 6th international conference on Autonomic computing*. p. 127–136. ACM.

Kusic, D., J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang. 2009. "Power and performance management of virtualized computing environments via lookahead control". *Cluster computing*, vol. 12, n° 1, p. 1–15.

Lee, J. Y., J. W. Lee, S. D. Kim, et al. 2009. "A quality model for evaluating software-as-a-service in cloud computing". In *Software Engineering Research, Management and Applications, 2009. SERA'09. 7th ACIS International Conference on*. p. 261–266. IEEE.

Liang, Y., Y. Zhang, H. Xiong, and R. Sahoo. 2007. "Failure prediction in ibm bluegene/l event logs". In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*. p. 583–588. IEEE.

LLC, M. 2016. "Cisco Visual Networking Index: Global - 2019 Forecast Highlights". <http://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/vni-forecast.html>.

Luna, J., H. Ghani, D. Germanus, and N. Suri. 2011. "A security metrics framework for the cloud". In *Security and Cryptography (SECRYPT), 2011 Proceedings of the International Conference on*. p. 245–250. IEEE.

Mahdavi-Hezavehi, S., M. Galster, and P. Avgeriou. 2013. "Variability in quality attributes of service-based software systems: A systematic literature review". *Information and Software Technology*, vol. 55, n° 2, p. 320–343.

Marinescu, D. C., 2013. *Cloud computing: theory and practice*.

Marston, S., Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi. 2011. "Cloud computing—The business perspective". *Decision support systems*, vol. 51, n° 1, p. 176–189.

Mdhaffar, A., R. B. Halima, M. Jmaiel, and B. Freisleben. 2013. "A dynamic complex event processing architecture for cloud monitoring and analysis". In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*. p. 270–275. IEEE.

Mell, P. and T. Grance. 2011. "The NIST definition of cloud computing".

Microsoft. 2009. "Microsoft Application Architecture Guide 2.0 - Quality Attributes". <https://msdn.microsoft.com/en-us/library/ee658094.aspx>.

Montes, J., A. Sánchez, B. Memishi, M. S. Pérez, and G. Antoniu. 2013. "GMonE: A complete approach to cloud monitoring". *Future Generation Computer Systems*, vol. 29, n° 8, p. 2026–2040.

Moody, D. L. 2005. "Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions". *Data & Knowledge Engineering*, vol. 55, n° 3, p. 243–276.

Nallur, V., R. Bahsoon, and X. Yao. 2009. "Self-optimizing architecture for ensuring quality attributes in the cloud". In *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*. p. 281–284. IEEE.

Nathuji, R., C. Isci, and E. Gorbatov. 2007. "Exploiting platform heterogeneity for power efficient data centers". In *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*. p. 5–5. IEEE.

Netcraft. 2017. "Total number of Websites". <http://www.internetlivestats.com/total-number-of-websites/>.

OASIS. 2013. "Topology and Orchestration Specification for Cloud Applications Version 1.0". <http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html>.

O'Brien, L., P. Merson, and L. Bass. 2007. "Quality attributes for service-oriented architectures". In *Proceedings of the international Workshop on Systems Development in SOA Environments*. p. 3. IEEE Computer Society.

Openstack. 2017a. "Prescriptive example for General Purpose Openstack Deployment". <http://docs.openstack.org/arch-design/generalpurpose-prescriptive-example.html>.

Openstack. 2017b. "Openstack Sample Configurations". <https://www.openstack.org/software/sample-configs>.

Openstack. 2017c. "Shade's documentation". <http://docs.openstack.org/infra/shade/>.

Openstack. 2017d. "Telemetry Measurements". <http://docs.openstack.org/admin-guide/telemetry-measurements.html>.

Padala, P., K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. 2007. "Adaptive control of virtualized resources in utility computing environments". In *ACM SIGOPS Operating Systems Review*. p. 289–302. ACM.

Petcu, D., G. Macariu, S. Panica, and C. Crăciun. 2013. "Portable cloud applications—from theory to practice". *Future Generation Computer Systems*, vol. 29, n° 6, p. 1417–1430.

Raghavendra, R., P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. 2008. "No power struggles: Coordinated multi-level power management for the data center". In *ACM SIGARCH Computer Architecture News*. p. 48–59. ACM.

Raj, A. 2016. "CPU hotplug Support in Linux(tm) Kernel". <https://www.kernel.org/doc/Documentation/cpu-hotplug.txt>.

Rico, D. F., 2004. *ROI of software process improvement: Metrics for project managers and software engineers*.

Ruhe, G. and C. Wohlin, 2014. *Software Project Management in a Changing World*.

Samreen, F., Y. Elkhatib, M. Rowe, and G. S. Blair. 2016. "Daleel: Simplifying cloud instance selection using machine learning". In *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*. p. 557–563. IEEE.

Singh, S. and I. Chana. 2016. "QoS-aware autonomic resource management in cloud computing: a systematic review". *ACM Computing Surveys (CSUR)*, vol. 48, n° 3, p. 42.

Sırbu, A. and O. Babaoglu. 2014. "Towards a systematic analysis of cluster computing log data: the case of IBM BlueGene/Q".

Sodhi, B. and T. Prabhakar. 2012. "Examining the Impact of Platform Properties on Quality Attributes". *arXiv preprint arXiv:1205.4626*.

Song, Y., H. Wang, Y. Li, B. Feng, and Y. Sun. 2009. "Multi-tiered on-demand resource scheduling for VM-based data center". In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. p. 148–155. IEEE Computer Society.

Sosinsky, B., 2010. *Cloud computing bible*, volume 762.

Stillwell, M., D. Schanzenbach, F. Vivien, and H. Casanova. 2009. "Resource allocation using virtual clusters". In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. p. 260–267. IEEE Computer Society.

Suryn, W., 2013. *Software Quality Engineering: A Practitioner's Approach*.

Toosi, A. N., R. N. Calheiros, and R. Buyya. 2014. "Interconnected cloud computing environments: Challenges, taxonomy, and survey". *ACM Computing Surveys (CSUR)*, vol. 47, n° 1, p. 7.

Torkura, K. A., F. Cheng, and C. Meinel. 2015. "Application of quantitative security metrics in cloud computing". In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*. p. 256–262. IEEE.

Trihinas, D., G. Pallis, and M. D. Dikaiakos. 2014. "Jcatascopia: Monitoring elastically adaptive applications in the cloud". In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. p. 226–235. IEEE.

Vakilinia, S., M. M. Ali, and D. Qiu. 2015. "Modeling of the resource allocation in cloud computing centers". *Computer Networks*, vol. 91, p. 453–470.

Vakilinia, S., B. Heidarpour, and M. Cheriet. 2016a. "Energy Efficient Resource Allocation in Cloud Computing Environments". *IEEE Access*.

Vakilinia, S., X. Zhang, and D. Qiu. 2016b. "Analysis and optimization of big-data stream processing". In *Global Communications Conference (GLOBECOM), 2016 IEEE*. p. 1–6. IEEE.

Vázquez-Poletti, J. L., R. Moreno-Vozmediano, R. S. Montero, E. Huedo, and I. M. Llorente. 2013. "Solidifying the foundations of the cloud for the next generation Software Engineering". *Journal of Systems and Software*, vol. 86, n° 9, p. 2321–2326.

Verma, A., P. Ahuja, and A. Neogi. 2008. pmapper: power and migration cost aware application placement in virtualized systems. *Middleware 2008*, p. 243–264. Springer.

W3Techs. 2017. "Usage Statistics and Market Share of Content Management Systems for Websites". <https://w3techs.com/technologies/overview/content_management/all/>.

Wiegers, K. and J. Beatty, 2013. *Software requirements*.

Wikipedia. 2016. "List of system quality attributes". <https://en.wikipedia.org/wiki/List_of_system_quality_attributes>.

Yau, S. S. and H. G. An. 2011. "Software engineering meets services and cloud computing". *Computer*, vol. 44, n° 10, p. 47–53.

Zhang, Q., L. Cheng, and R. Boutaba. 2010. "Cloud computing: state-of-the-art and research challenges". *Journal of internet services and applications*, vol. 1, n° 1, p. 7–18.

Zhang, Q., M. F. Zhani, R. Boutaba, and J. L. Hellerstein. 2014. "Dynamic heterogeneity-aware resource provisioning in the cloud". *Cloud Computing, IEEE Transactions on*, vol. 2, n° 1, p. 14–28.