**IMT Institute for Advanced Studies**

Lucca, Italy

# Supporting Autonomic Management of Clouds: Service-Level-Agreement, Cloud Monitoring and Similarity Learning

PhD Program in Computer Science and Engineering

XXVII Cycle

**By**

**Rafael Brundo Uriarte**

**March, 2015**

# Contents

# List of Figures

# List of Tables

# Acknowledgements

When starting this thesis, I asked myself whether to use "I" or "we" pronouns; the answer came straight to my mind. No one does anything alone. In my case, this work would never be accomplished without the invaluable support and dedication of many.

I have no idea how to express my gratitude to the people, who were part of this journey. First, I want to thank Francesco Tiezzi for his endless patience and for the uncountable hours discussing problems and solutions, which were essential for this work. Rocco De Nicola for his guidance, his availability and for being always open to new ideas. Also, to Sotirios Tsaftaris, who has contributed immensely, not only to the technical aspect of the thesis but also for my (ongoing) maturation process as a researcher. I really appreciate their efforts and patience to teach and advice a stubborn person like me. Professor Carlos Westphall (and his students in LRG), which, even from Brazil, always made everything possible to help. Finally, I am grateful to my dissertation committee members, Carlos Canal, Michelle Sibilla and Mirco Tribastone for their time, interest, encouragements and helpful comments.

I gratefully acknowledge the funding sources that made my Doctorate work possible. I was funded by the Italian Government, by the Cassa di Risparmio di Lucca Foundation, by the EU project ASCENS and by the CINA project.

Words cannot express how grateful I am to my wife, my friends and my family. They support me every time I needed, even sometimes only with a smile or a small joke which would cheer me up. Specially, to Yesim for her help with the thesis and mainly for being my beloved and inspiration. Through her love, caring, patience and her (sometimes unjustified) unwavering belief in me, I was able to complete this journey. Also, special thanks to my friends, in particular to my lifelong friend Gustavo, who is always there when necessary. My time in Lucca was made enjoyable not only by this wonderful city but also by its inhabitants and friends in IMT, thank you Sahizer, Oznur, Omar, Michelle, Ajay, Alberto, Carlo, Francesca, Massimo, Elio and Bernacchi.

Finally, I would like to dedicate this thesis to my mother, father and sister. It is incredible that the distance does not diminish their importance. To them, I dedicate this thesis.

You all make me feel the luckiest person in the world. Thank you.

# List of Publications

1. R. B. Uriarte, S. Tsaftaris and F. Tiezzi. Service Clustering for Autonomic Clouds Using Random Forest. In *Proc. of the 15th IEEE/ACM CCGrid* [In Press], 2015.
2. R.B. Uriarte, F. Tiezzi, R. De Nicola, SLAC: A Formal Service-Level-Agreement Language for Cloud Computing. In *IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC 2014)*, 2014.
3. R.B. Uriarte, C.B. Westphall, Panoptes: A monitoring architecture and framework for supporting autonomic Clouds, In *Proc. of the 16th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2014.
4. R.B. Uriarte, S.A. Chaves, C.B. Westphall, Towards an Architecture for Monitoring Private Clouds. In *IEEE Communications Magazine*, 49, pages 130-137, 2011.
5. R. Willrich, L. H. Vicente, A. C. Prudłncio, V. S. N. Alves, R. B Uriarte, F. B. Teixeira. Estabelecimento de Sessoes SIP com Garantias de QoS e sua Aplicaçao em um Dominio DiffServ. In *Proc. of the 29th Simpsio Brasileiro de Redes de Computadores*, 2011.
6. R. B. Uriarte, S. Chaves, C. Westphall, P. Vitti. Projeto e Implantaçao de um Arcabouço para o Monitoramento de Nuvens Privadas. In *Proc. of 37th Conferencia Latinoamericana de Informatica*, 2011.
7. S. A. de Chaves, R. B. Uriarte, C. B. Westphall. Implantando e monitorando uma nuvem privada. In *Proc. of the 8th Workshop em Clouds, Grids e Aplicaçoes*, 2010.
8. R. Willrich, L. H. Vicente, R. B. Uriarte, A. C. Prudencio, J. J. C. Invocaçao Dinamica de Serviços com QoS em Sessoes Multimidia SIP. In *Proc. of the 8th Int. Information and Telecommunication Technologies Symposium*, 2009.

# Abstract

Cloud computing has grown rapidly during the past few years and has become a fundamental paradigm in the Information Technology (IT) area. Clouds enable dynamic, scalable and rapid provision of services through a computer network, usually the Internet. However, managing and optimising clouds and their services in the presence of dynamism and heterogeneity is one of the major challenges faced by industry and academia. A prominent solution is resorting to self-management as fostered by autonomic computing.

Self-management requires knowledge about the system and the environment to enact the self-* properties. Nevertheless, the characteristics of cloud, such as large-scale and dynamism, hinder the knowledge discovery process. Moreover, cloud systems abstract the complexity of the infrastructure underlying the provided services to their customers, which obfuscates several details of the provided services and, thus, obstructs the effectiveness of autonomic managers.

While a large body of work has been devoted to decision-making and autonomic management in the cloud domain, there is still a lack of adequate solutions for the provision of knowledge to these processes.

In view of the lack of comprehensive solutions for the provision of knowledge to the autonomic management of clouds, we propose a theoretical and practical framework which addresses three major aspects of this process: (i) the definition of services' provision through the specification of a formal language to define Service-Level-Agreements for the cloud domain; (ii) the collection and processing of information through an extensible knowledge discovery architecture to monitor autonomic clouds with support to the knowledge discovery process; and (iii) the knowledge discovery through a machine learning methodology to calculate the similarity among services, which can be employed for different purposes, e.g. service scheduling and anomalous behaviour detection. Finally, in a case study, we integrate the proposed solutions and show the benefits of this integration in a hybrid cloud test-bed.

# Chapter 1

# Introduction

*Cloud computing* has grown rapidly during the past few years and has become a key paradigm in the Information Technology (IT) area. The IT analysis firm IDC confirms the importance of cloud computing with a market analysis. It predicted that by the end of 2014 the spending on cloud computing will reach over 100 billion dollars, with a remarkable growth of 25%, while the total IT expenditures will grow only around 5% [Gen13].

The popularity of cloud is due to the fact that it enables dynamic, scalable and rapid provision of services through a computer network, usually the Internet. Moreover, cloud systems abstract the complexity of the infrastructure underlying the provided services to their customers. However, due to these layers and its features along with its distributed and large-scale nature, managing clouds is considerably more complex than traditional data centres [NDM09].

A prominent approach to cope with this complexity is *Autonomic Computing* [Hor01], which aims at equipping such systems with capabilities to autonomously adapt their behaviour and/or structure according to dynamic operating conditions. To effectively achieve such self-management, the system entities in charge of enacting autonomic strategies, the so-called autonomic managers, require knowledge about the system. Management is performed through the well-known MAPE-K (Monitoring,

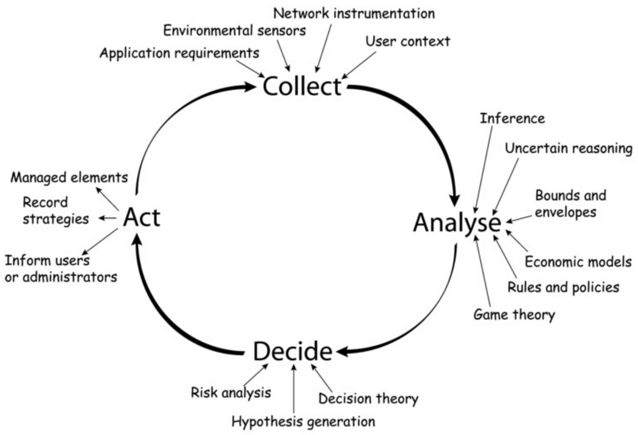

**Figure 1:** Autonomic functions.

Analysis, Planning, Execute based on Knowledge) [IBM05] control loop. The analyses and decisions of this control loop rely on knowledge and the status of the system, which are perceived through *sensors*. Figure 1 [DDF06] depicts the major functions of autonomic managers and some solutions that support these functions.

Differently from the standard autonomic approach, autonomic clouds emphasise the management of services since clouds are service-oriented. As a result, the application of the autonomic concepts in the cloud domain must also consider the objectives of the services being provided.

In light of the characteristics of the domain, different types of knowledge are required by the decision-making of autonomic managers. In particular, three types of knowledge are common to all implementations of autonomic clouds: high-level *policies*, which describe the objectives of the system and are commonly defined by the cloud administrator; the *definition of the cloud services' provision* and their objectives; and the *status of the cloud*. Moreover, depending on the capacity and on the methods employed by decision-making process, the autonomic managers need *specific types of knowledge* about the cloud or services.

The overall aim of this thesis is to design a theoretical and practical framework tailored for the cloud domain for the definition of services, monitoring information gathering and the generation of knowledge. In what follows, we describe the types of knowledge addressed by this thesis. Nevertheless, it should be noted that among the types of knowledge required by the autonomic managers, we do not cover the specification of high-level policies since they rarely change and the policies defined for the cloud domain have little or no difference to other domains.

The first type of knowledge concerns the definition of services and their objectives. The use of Service-Level-Agreements (SLAs) is a possible solution for the provision of such definitions to the autonomic managers. SLAs are the formalization of the characteristics and objectives of the services and are employed to regulate the service. In autonomic clouds, the SLAs serve as a guide for the decision-making process of the autonomic managers, i.e. they provide performance goals of the service. However, SLAs must enable the definition of cloud services with their specificities and, in particular, they have to be both human-readable, in order to facilitate the task of specifying and maintaining them, and machine-readable, to be used a as source of knowledge by autonomic managers.

The second type of knowledge refers to the status of the system. To generate this knowledge, a monitoring system needs the ability to correlate signals and facts, potentially expressed as event messages, to determine what is occurring in the cloud environment [Ste02]. Combinations and correlations of facts and signals are used to generate higher abstractions of the system, which are provided to autonomic managers in order to empower their decision-making process.

The third category refers to the types of knowledge required by specific implementations of autonomic managers. These types of knowledge vary considerably according to the implementation and scope of the autonomic managers and can assume many forms; for example, the risk of violation of a SLA, a model to find a robust similarity measure among services or the prediction that the computing load of a cloud will be reduced in the next hours. Due to its generality and broad scope, we define and analyse the process of discovering knowledge in autonomic clouds and focus on a specific type of knowledge which can be used in different scenarios: the

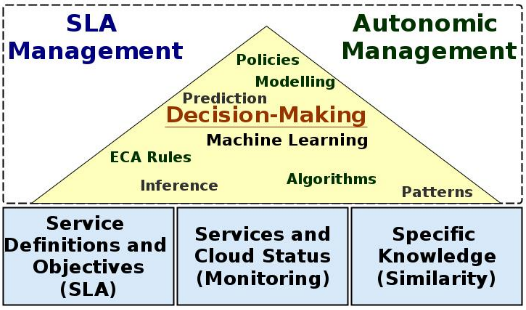

**Figure 2:** Foundations of the decision-making in the autonomic management which are addressed in this thesis.

similarity among services.

Figure 2 depicts the decision-making process of the autonomic managers and the main pillars that feed it with knowledge from the cloud.

To illustrate the importance of the knowledge provision in the autonomic management, we list the requirements of a common objective of cloud autonomic managers: the enforcement SLA terms. Most models that predict possible violations on the SLA require: (i) the SLA objectives, (ii) monitoring information of the service, (iii) the model to generate this knowledge and (iv) the timely delivery of this knowledge. Only with this knowledge, the autonomic managers can pro-actively adapt the system and avoid such violations.

However, the characteristics of cloud computing, such as large-scale, dynamism and the measures used to improve security in the domain, hinder the knowledge discovery processes. Moreover, the autonomic management of clouds must manage different levels of abstraction of the system components, such as services, clusters and external providers, which also complicate these processes. Therefore, these difficulties should be considered in the development of solutions for the domain.

Nevertheless, to the best of our knowledge, no work deals at once with various aspects of the knowledge discovery process in the autonomic cloud domain. Although several works provide a high-level account of the autonomic components and of the MAPE-K loop, also for the

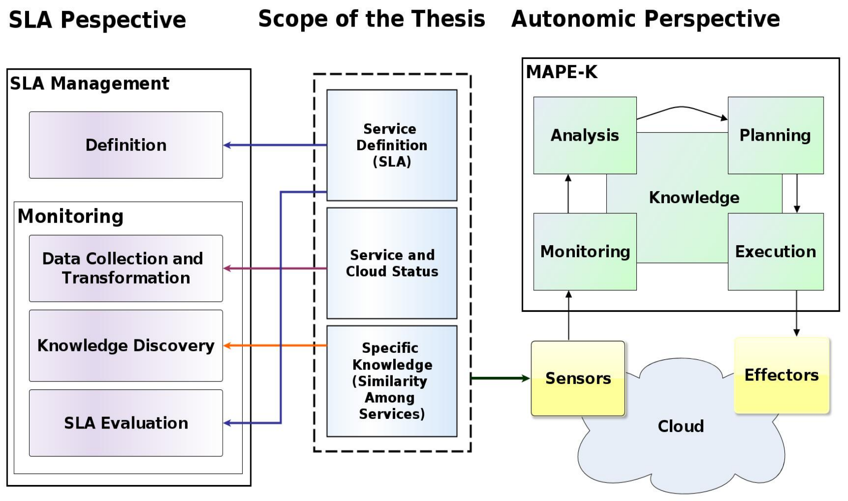

**Figure 3:** The relation of the thesis with the autonomic and SLA management perspectives.

cloud domain, they focus on the decision-making and assume that the knowledge for the autonomic decisions is available[1].

Similarly to autonomic management, the foundations of the SLA management are based on three pillars. These pillars are the same as the ones of the autonomic computing but their scope is limited to the definition, knowledge and status only of the services of the cloud. In fact, these requirements are a subset of the knowledge that is necessary for the autonomic management; therefore, we also consider the works regarding the SLA management and present the proposed solutions from this standpoint as well. However, also from the SLA perspective, to the best of our knowledge, no work focuses on the provision of knowledge for its management. Figure 3 relates the types of knowledge addressed by this thesis with the autonomic and the SLA perspectives.

Although big strides have been made in recent years in the field of

[1]Many works discuss solutions for a single aspect of the provision of knowledge for the autonomic management. These works will be analysed in the chapter where we specifically address the particular aspect.

autonomic clouds and SLA management, there remain open questions on the foundations of these two processes: how the knowledge is provided to managers and how multiple sources of knowledge are integrated. Therefore, to address this gap in the literature, we analyse the domain and design Polus, which is a theoretical and practical framework to integrate multiple sources of knowledge provisions. In particular, we focus on the definition of services, on the monitoring of the domain, on the knowledge discovery process and on the discovery of similarity among services, which are the foundations of the decision-making of autonomic clouds.

In addressing the lack of comprehensive solutions for the provision of knowledge for the autonomic management of cloud systems, we provide the necessary means to define services, collect information and transform it into knowledge, which enables the accurate decision-making and, consequently, the enactment of the self-management properties in autonomic clouds.

## 1.1 Research Questions and Objectives

Based on the knowledge required by the decision-making process of autonomic clouds, we specify the main research questions addressed by the thesis as follows:

**Research Question 1**

*How to describe services and their objectives in the cloud domain?*

Clouds have a unique set of characteristics, some of which are inherited by the services that are provided by this paradigm. These characteristics make the description and objectives of services difficult to grasp in SLAs. In particular, SLAs provide to autonomic managers knowledge, such as the aims of the services, the penalties in case of violation (which can help the autonomic manager to evaluate when it is convenient to violate a SLA) and priorities. Thus, a language for the definition of SLAs, which is able to capture the characteristics of the domain, is a requirement for the autonomic and the SLA management. This question is addressed in Chapter 3.

**Research Question 2**

*What is data, information, knowledge and wisdom in the autonomic cloud domain?*

Knowledge is the foundation of the decision-making process of autonomic clouds. Yet, knowledge has different definitions according to the context. Therefore, robust definitions of the concepts and levels of the knowledge hierarchy are essential to analyse and build a conceptual framework to support autonomic clouds with the necessary knowledge. This question is addressed in Chapter 4.

**Research Question 3**

*How to collect and transform the enormous amount of operational data into useful knowledge without overloading the autonomic cloud?*

Collecting and processing data in the autonomic cloud domain is a major challenge due to its dynamism, scale and elasticity. Moreover, a cloud has many abstraction levels, which can be independent of physical or logical location (e.g. service, node, cluster). Therefore, designing a non-intrusive monitoring system, which provides the status of the system and services in a timely manner, is important. This question is addressed in Chapter 4.

**Research Question 4**

*How to produce a robust measure of similarity for services in the domain and how can this knowledge be used?*

Different knowledge types are essential for the decision-making process. Similarity among services is a type of knowledge which can be used for various aims, such as anomalous behaviour detection, application profiling and scheduling. However, clouds are heterogeneous, dynamic and large-scale, which hinder this task. Therefore, new techniques are necessary to discover a robust measure of similarity among services in the domain. This question is addressed in Chapter 5.

**Research Question 5**

*How to integrate different sources of knowledge and feed the autonomic managers?*

The knowledge that is employed to manage autonomic clouds is commonly generated from different sources. Managing requests and the provision of such knowledge might require a complex interaction between the autonomic managers and these sources. Therefore, integrating these components to feed the autonomic manager is a necessity. This question is addressed in Chapter 6.

## 1.2 Scientific Contributions

The overall contribution of this thesis to the state-of-the-art in the autonomic cloud management is the design and implementation of a theoretical framework (named Polus) and software tools to define services, collect data from the system and transform this data into knowledge.

Below, we the list the major contributions of this thesis in different areas[2] . Although the proposed solutions are independent and can be exploited separately, they can also be integrated as demonstrated in Chapter 6.

- **SLAC: A Language for the Definition of Service-Level-Agreement for Clouds** - A SLA definition language tailored for clouds and it includes the vocabulary for the specification of SLA in this domain. Other contributions on the area are:
  - The analysis of the requirements for a SLA definition language for the domain;
  - A survey considering existing solutions;
  - The implementation of a software framework which supports this language.

[2]The software implementation of the proposed solutions are free and open source. They are available on:
`http://sysma.imtlucca.it/tools/slac/`
`http://code.google.com/p/unsupervised-randomforest/`

The details of this language are presented in Chapter 3 and has been published in [UTD14].

- **Knowledge Discovery Process** - The definition of the knowledge discovery process in the autonomic cloud domain. These definitions are presented in Section 4.1.

- **Panoptes: A monitoring System for Autonomic Clouds** - The design of an architecture of a multi-agent monitoring system, which considers the knowledge discovery processes and can be integrated with autonomic systems. Other contributions in this area are:

  – The implementation of this monitoring solution;

  – The support for the conversions of SLAC terms into low-level monitoring metrics.

  The details of this solution are presented in Chapter 4 and has been published in [UW14].

- **Service Similarity Discovery** - The development of a methodology based on machine learning techniques to discover similarities among services considering the specificities of the domain. This methodology can be used for, e.g. application profiling and service scheduling. Below we list other contributions of the chapter:

  – The implementation of this methodology;

  – The development of a novel scheduler using the concept of similarity.

  The methodology is discussed in Chapter 5 and has been published in [UTT15].

- **Integration of the Sources of Knowledge** - The integration of the solutions proposed in this thesis which form Polus, which is a framework to provide knowledge to autonomic manager. Moreover, we develop an autonomic cloud use case which provides an overview of the use of Polus with an autonomic scheduler for services in a

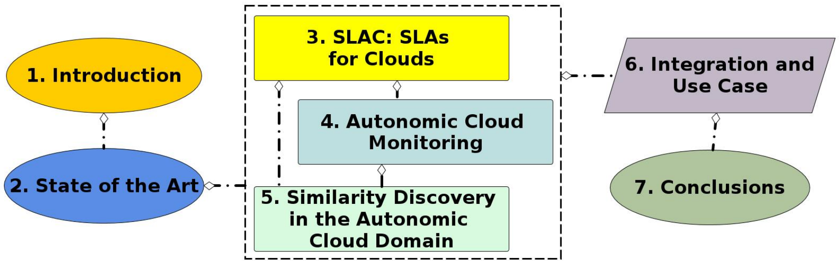
1. Introduction
2. State of the Art
3. SLAC: SLAs for Clouds
4. Autonomic Cloud Monitoring
5. Similarity Discovery in the Autonomic Cloud Domain
6. Integration and Use Case
7. Conclusions

**Figure 4:** Thesis structure and the dependencies among chapters.

cloud test-bed based on the OpenNebula [Ope14] solution. The integration and the use case are presented in Chapter 6.

## 1.3 Thesis Organization and Use Case

The thesis is divided into seven chapters as illustrated in Figure 4. The core of the thesis concerns the foundations of the autonomic management, i.e. Chapters 3, 4 and 5, as represented in Figure 4 by a dotted rectangle. The arrows among chapters (or among the core of the thesis and other chapters) represent their dependencies. However, these dependencies do not indicate a compulsory reading and are only a guide to illustrate the relations among them. In the following, we detail the chapters.

- **Chapter 2 - Autonomic Clouds and SLAs: The State of the Art** - Several concepts and paradigms are essential to understand the autonomic cloud domain and the solutions proposed in this thesis. Chapter 2 provides these ground concepts, discusses the works with a similar scope and overviews the most important SLA languages and the monitoring properties in the cloud domain.

- **Chapter 3 - Service-Level-Agreements in the Cloud Domain** - In Chapter 3, we discuss the requirements of the domain for the definition and formalisation of services and propose SLAC, a language which we developed to fulfil these requirements. In particular, we describe its syntax, the formalisation of the SLA evaluation semantics and introduce an extension to support the business aspects. Moreover, we describe a software framework developed to support SLAC and discuss a use case employing this implementation.

- **Chapter 4 - Autonomic Clouds Monitoring** - This chapter presents the knowledge discovery process in the domain. Considering this process, we devise a monitoring solution for autonomic clouds, which focuses on the integration of the monitoring system with autonomic management and on monitoring properties that are particularly important in the autonomic cloud domain.

- **Chapter 5 - Similarity Learning in Autonomic Clouds** - We propose a methodology to provide a robust similarity measure based on the definition of services or on the monitoring data. Similarity knowledge can be used in different areas, such as optimisation of resources, service scheduling and anomalous behaviour.

- **Chapter 6 - Polus: Integration and Use of SLAC, Panoptes and Similarity Learning** - This chapter discusses the integration of the solutions proposed in the other chapters. Moreover, we provide a real-world use case which demonstrates the benefits of our approach. Despite having the scope restricted to the definition of SLAs and the provision knowledge to the autonomic management, i.e. not covering the management of these systems, we implement service scheduler algorithms to illustrate the potential benefits of our solutions.

- **Chapter 7 - Conclusion** - In this chapter, we summarise the findings of this study, relate the research questions with the contributions and describe possible directions for future work.

We conclude this section by presenting a scenario which will be used throughout the thesis to illustrate the presented concepts. It is based on a real-world cloud implementation employing the OpenNebula platform. We set up this platform in a cloud test-bed in IMT Lucca as an academic cloud.

The scenario concerns the provision of computational resources to the students of the institute and is used for research purposes, employing the Infrastructure-as-a-Service model. Two standard versions of the service are available: (i) a single and powerful virtual machine (VM); and (ii) a cluster of smaller VMs. The first service aims at supporting centralized research applications, while the second one targets applications which require distributed environments.

As an extension of this scenario, we employ a third party cloud to form a *hybrid* cloud and overcome the resource limitations of our OpenNebula test-bed. Thus, the IMT cloud offers to the researchers a third kind of service with more powerful machines, outsourcing its provision to an external cloud.

# Chapter 2

# Autonomic Clouds and SLAs: The State of the Art

This chapter presents the ground concepts and an overview of the fundamental paradigms that constitute the basis of this thesis. In particular, it describes the cloud computing domain, demonstrates the importance of autonomic computing for clouds, explains the essential role of the SLA in the context of autonomic systems and of cloud management, and discusses the monitoring system properties in the cloud domain.

The overall contributions of this chapter consist of:

- A description of the cloud paradigm with its taxonomy, major challenges and benefits, which is essential to understand the problem addressed by this thesis and provides the basis of the design choices for the solutions presented in other chapters;

- A comprehensive analysis of the autonomic computing vision and the use of its main concepts, applied to the cloud domain;

- An introduction to fundamental concepts of SLAs and their relation with the management of complex systems, such as clouds. SLAs are the core of this thesis, serving as the starting point of all the proposed solutions in various areas that this thesis addresses;

Notably, this chapter focuses on the works related to the thesis as a whole. The works on specific aspects of the knowledge provision are discussed in their respective chapters.

## 2.1 Cloud Computing

Cloud computing is a new paradigm that uses principles, techniques and technologies from different fields [CWW11]. The idea behind the cloud is not new. In fact, Professor John McCarthy said at MIT's centennial celebration in 1961 that "...computing may someday be organized as a public utility just as the telephone system is a public utility... The computer utility could become the basis of a new and important industry" [Gar99].

Despite this early definition, it took almost four decades to develop the ground technologies which make cloud viable. Among the most significant evolutions there are: the growth of Internet and the consolidation of the virtualization paradigm.

Nevertheless, several definitions of cloud computing exist. Among them, the most cited and widely accepted definition of clouds is provided by the United States National Institute of Standards and Technology (NIST) in [MG09]:

> Cloud computing is a model for enabling 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. This cloud model promotes availability and is composed of five essential characteristics, three service models, and four deployment models.

### 2.1.1 Essential Characteristics

The five essential characteristics that define cloud computing cited by NIST are presented below:

- On-demand self-service - A consumer can require the provision of computing capabilities (such as, storage and server time) without human interaction with the provider;

- Broad network access - The services are delivered over a network (e.g. Internet) and used by various clients and platforms;

- Resource Pooling - The cloud providers "pool" computing resources together in order to serve multiple consumers using the virtualization and multi-tenancy models. Virtualization refers to the partitioning of a resource into multiple virtual resources, which, e.g. enables isolation of resources, leverages security and provides hardware-independence. Multi-tenancy refers to a single software that provides multiple independent instances of the service to different consumers. The information on the location of the data centres or the details on the management of this pool of resources are transparent for the consumers, which rely on a high-level abstraction of these details (when available);

- Rapid Elasticity - Cloud providers should offer virtually infinite resources, always matching the consumers' needs. Moreover, these resources have to be rapidly, elastically and possibly automatically scale in and out;

- Measured Service - Providers should autonomously manage and optimize resources usage. This usage should be monitored, controlled and reported to assure transparency for the provider and consumer of the service.

Both consumers and providers can benefit from this paradigm. Providers can reduce operational costs through economy of scale, since they serve several consumers with a range of standard services. Cloud consumers can, instead, (i) have low upfront cost, which reduces the required investments to start a new business or to migrate to the cloud; (ii) can opt for the pay-as-you-go model, i.e. consumers pay only for used resources; (iii) have elasticity and agility for their service and rapidly in and out scaling according to their needs; and (iv) focus on their business instead of focusing on managing IT infrastructure. These benefits help consumers to attain reductions on IT-related costs, which attracted a great deal of attention from industry and academy.

Despite the popularity of cloud computing, a number of challenges and risks are inherent to this model. We describe some of the most important:

- *Security, trust and privacy* are the biggest barriers for the adoption of cloud computing [RCL09, CZ12, ZZX$^+$10] as consumers' data is out of their control. For example, the lack of standard procedures, regulations and guarantees contributes to the fear to execute sensitive operations and to store sensitive data in clouds.

- *Legal and regulatory issues* need attention as the services can be anywhere in the world and the location of the resources might determine the laws which regulate that service. For instance, apart from exceptional cases, transfers of personal data outside of European Economic Area is prohibited.

- Another important challenge is the *vendor lock-in*. Providers might require applications and data in non-standard formats, which may not be portable to other providers or in-house deployment unless a costly conversion process is carried out. Consequently, this lock-in might prevent the consumers from switching provider, even in cases, such as when the provider does not meet the consumers' requirements any more or in cases in which the price of the service is raised.

### 2.1.2 Deployment Models

The cloud infrastructure is not necessarily deployed by a third party that provides services to consumers. Figure 5 presents the four major deployment models used in the cloud domain, which are:

- In *private* clouds the infrastructure that enables the cloud is dedicated to a single organisation, being hosted and managed by a

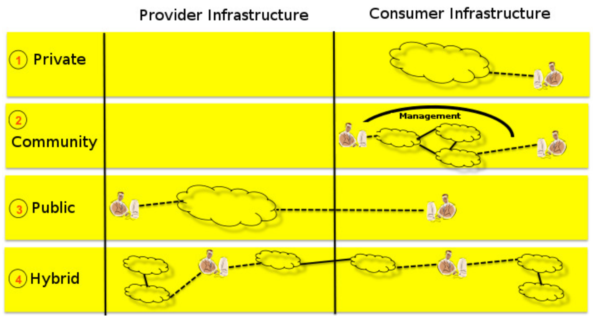

**Figure 5:** The major cloud deployment models.

third-party or by the consumer himself. This model provides greater control and security, since it is accessible by a single organization and, normally, is deployed behind the organization's firewall. Frequently, private clouds are used in large organizations to benefit from the cloud characteristics and, at the same time, to keep sensitive data under the organization's premisses;

- The goal of a *community* cloud is to share the infrastructure of the involved partners to build a multi-tenant cloud for specific purposes. Intuitively, it is a cloud formed by resources of all the participants, making these resources available on-demand to the participants. This model provides an additional level of privacy and security due to internal policies, which are approved by the involved partners and which do not permit access to the general public;

- In *public* clouds, services are provided to the general public over a network, usually the Internet. The infrastructure is operated by the provider and service offers are, commonly, fixed. The main advantages of this model is the reduction of complexity, up-front costs and the attractive service cost due to the economy of scale afforded by cloud providers;

- The resources of a private cloud can also be combined with resources of a public cloud to form a *Hybrid* cloud. In this model, commonly, private clouds execute sensitive tasks and are employed to store sensitive data, while the public cloud is used for non-sensitive operations as they are less secure but cost efficient and scalable.

Though not a deployment model, another relevant concept in the field is the one of *federated clouds*. Federated cloud is an organizational model to transparently (to the consumer) integrate multiple clouds. Rochwerger et al. [RBL$^+$09] argue that infrastructure providers could take advantage of their aggregated capabilities to provide seemingly infinite computing utility only through federation and interoperability .

Just as clouds enable users to cope with unexpected demand loads, a federated cloud enables single clouds to cope with unforeseen variations of demand [GVK12]. However, they require cross-site agreements of cooperation regarding the provision of capacity to their clients [EL09]. Other important requirements for federated clouds are: standard interfaces, interoperability and solutions for integrated management.

### 2.1.3 Service Models

Regardless of the deployment model, cloud providers deliver computational resources as services. In this context, “service” has a broad meaning and, virtually, everything can be provided as an on-line, on-demand service [TNL10]. Such generic service delivery model is called Everything/Anything as a Service (XaaS) and the most well-known instances of this model are [MG09]:

- Software-as-a-Service (SaaS): It is a software distribution model delivered over a network, in which consumers use applications hosted by a cloud provider. Sometimes referred to as on-demand software, this model frees the consumers from the management of the cloud infrastructure, platform and software. However, commonly, the services made available by a provider are restricted to few applications with little customization capabilities. From the provider’s perspective, the applications are executed in multi-tenant virtual environments, which provide scalability and elasticity.

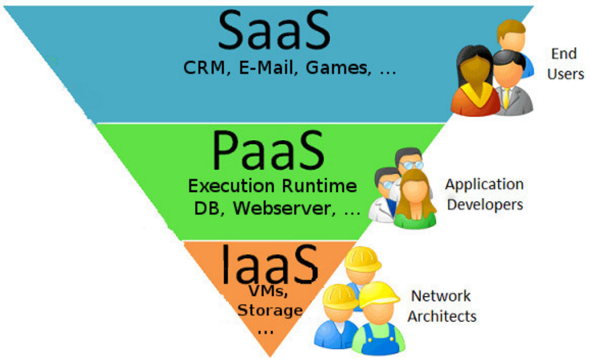

**Figure 6:** Cloud delivery models, examples of applications and their target consumers.

- Platform-as-a-Service (PaaS): This model delivers operating system, application development tools and a framework for application's deployment to consumers over a network without requiring the installation or download of extra tools. Thus, consumers deploy their own applications (home-grown or acquired) into the cloud infrastructure without the cost and complexity of acquiring and managing the underlying layers.
- Infrastructure-as-a-Service (IaaS): It provides storage, network, processing and other infrastructure resources to consumers. In these resources, consumers can deploy arbitrary software, ranging from applications to operating systems. This model offers more control to consumers but, as a trade-off, requires also the management and operation of the software components of the stack. For instance, when a consumer requests a VM from a provider, he is the responsible to operate, install and configure the operating system, platform to support the deployment of the software and the software configuration.

Figure 6[1] illustrates the hierarchy of the service models, their commonly provided services and their typical target users. The width of

[1]Figure based on:
`http://blog.samisa.org/2011/07/cloud-computing-explained.html`

the layers in the figure is related to the target audience: SaaS is easier to deploy and manage, thus has a higher number of potential users. PaaS targets application developers and experts that are able to deploy services. Finally, IaaS requires planning and installation of the whole software stack, which is executed by system administrators and network architects.

These models do not restrict the concept of service to a single resource or provider; therefore, a service can be composed of several other services and regulated by the same agreement, which, from the perspective of the consumer, is a single service.

### 2.1.4 Roles

In [LTM+11], the authors identified the major actors who carry out unique and specific roles in the cloud activities. The provision of cloud services can include more than one actor with the same role and more than one role for a single actor. These roles are:

- *Cloud Consumer* is an entity that uses services from providers;
- *Cloud Provider* is an organization or entity that makes a single or multiple services available to interested parties. Examples of well-known commercial cloud providers are Amazon, Rackspace and Google;
- *Cloud Carrier* is a third party involved in the provision of the service that provides connectivity and transports the services from the providers to consumers. Many carriers are telecommunication companies such as Telecom Italia or Telefonica;
- *Cloud Auditor* is an independent party that assesses the service provided in term of, e.g. security level, performance, SLA;
- *Cloud Broker* is an entity that negotiates relationships between providers and consumers, and manages the service use and delivery. Due to its importance in our work, we detail this role in the next section.

### 2.1.5 Cloud Broker

The increasing interest in cloud computing led to the development and creation of many service offers from multiple vendors. With the advent of these offers and many options for similar services, the role of the broker has also grown in importance. The broker is an entity that intermediates the interaction between one or multiple providers and the consumers. It can be of four types in the cloud domain [LTM$^+$11]:

- Matching - The broker searches for services compatible with the needs of consumers and returns a list of providers that are able to meet the consumers' requirements;
- Intermediation - The broker enhances a given service, offering value-added services (e.g. identity management, performance reporting, extra security mechanisms);
- Aggregation - Multiple services are combined into one or more new services. The broker provides the integration of the services (e.g. data and identity). The information about which atomic services are used to fulfil the consumers request are transparent to the consumer;
- Arbitrage - This type is similar to the aggregation brokerage. The main difference is that the broker has the flexibility to choose the providers of services during the provision, i.e. the providers are not fixed.

The business model for the brokerage may vary and the brokers can gain their share in several ways, for instance: (i) reducing the costs of the service through economy of scale and then selling the service to end users including its profit. For example, a provider offers 20% discount to consumers that use more than 1000 hours per month. A broker can purchase these hours and re-sell it for the same price as the provider, making a maximum of 20% profit; (ii) charging extra for value-added services or for combining different services; (iii) charging for each operation (e.g. a list of the IaaS providers available); or (iv) matching consumers to a specific provider, who gives commission for indicating his services.

## 2.2 Autonomic Computing

One of the biggest challenges in the system management is the growing complexity of computer systems and new paradigms [IBM05]. Cloud computing is an instance of complex systems, in which its characteristics, such as the dynamism, large-scale and heterogeneity, emphasise the complexity of the management process. Moreover, as previously discussed, the automation of the management is one of the essential characteristics of the cloud.

A prominent approach to cope with this complexity is Autonomic Computing [Hor01], which aims at equipping computer systems with capabilities to autonomously adapt their behaviour and/or structure according to dynamic operating conditions.

In order to effectively achieve self-management, a system needs the following properties:

- Self-Configuration - Installation and configuration of complex systems is an expensive and a fallible process. The self-configuration is executed according to high-level policies. For example, if a new node is added to a cloud infrastructure, an autonomic cloud must install and configure the necessary software stack to integrate this node in the cloud and add agents into this node to collect monitoring information;

- Self-Optimization - Provides the capacity of the system in tuning and adjusting policies and configurations to maximize system's performance and resources;

- Self-Healing - Assures the effective and automatic recovery when failures are detected;

- Self-Protecting - This characteristic enables the system to defend itself from malicious and accidental attacks. For instance, when a user tries to delete an important file of the operating system, the system forbids the operation;

- Self-Awareness - Computing systems are composed of a collection of "smaller" systems, such as single computers or processors, depending on the level of abstraction. Therefore, the system has to know itself, i.e. it requires the detailed knowledge of its components and of itself as a whole. For example, a cloud should retrieve data of single nodes but also process the data and extract information, such as the cloud usage and the total number of users;

- Context-Awareness - An autonomic system should collect information about important aspects of the environment and the interaction with other systems in order to optimize and react to changes [SILM07];

- Openness - Autonomic systems can not exist in hermetic environments. While independent in its self-management, it must function in heterogeneous environments and implement open standards;

- Anticipatory - The system must manage itself pro-actively, anticipating, when possible, the needs and behaviours of the system itself and of the context [PH05].

Autonomic systems use concepts of multi-agent systems in order to manage complex systems. An autonomic element is composed of a managed element and an autonomic manager. *Managed elements* can be any computing entity that requires self-management. The system entities (or agents) in charge of enacting autonomic strategies, the *autonomic managers*, rely on the MAPE-K loop to guarantee the above properties of the system. Figure 7[2] shows these phases, which are described below:

- Monitoring - Manages sensors and gauges that capture the properties (either physical or virtual) from the target system and the context. These properties represent an abstraction of the current state of the system, which are, then, stored in the knowledge database and sent to the analysis phase.

[2]Figure borrowed from
`http://www.ibm.com/developerworks/autonomic/library/ac-edge6/`

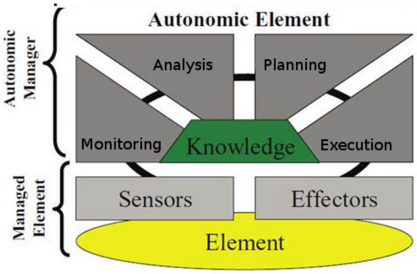

**Figure 7:** MAPE-K loop of the autonomic managers.

- Analysis - The information and knowledge produced in the monitoring phase are processed, generating high-level knowledge. Afterwards, the policies of the system and the produced knowledge are analysed and diagnostics of specific components and of the whole system are produced.
- Planning - The agent selects appropriate plans to adapt the system and maintain the autonomic properties according to the results of the analysis executed in the previous phase and the evaluation of the implications of using a specific strategy on the system.
- Execution - It defines the strategy to execute the selected plan during the previous phase. It includes also the choice of the actuators (the agent that will apply the defined actions).
- Knowledge - All the phases of the MAPE loop depend on knowledge. In the knowledge database, the status of the system is stored along with its polices, historical information and various types of knowledge defined by experts.

Moreover, this loop communicates with the managed element through sensors and effectors. The former collects information and knowledge from the managed element while the latter enables the autonomic manager to perform adaptations to the managed resource.

In light of these properties, a system can be defined as autonomic when agents manage the system's components (managed elements) using

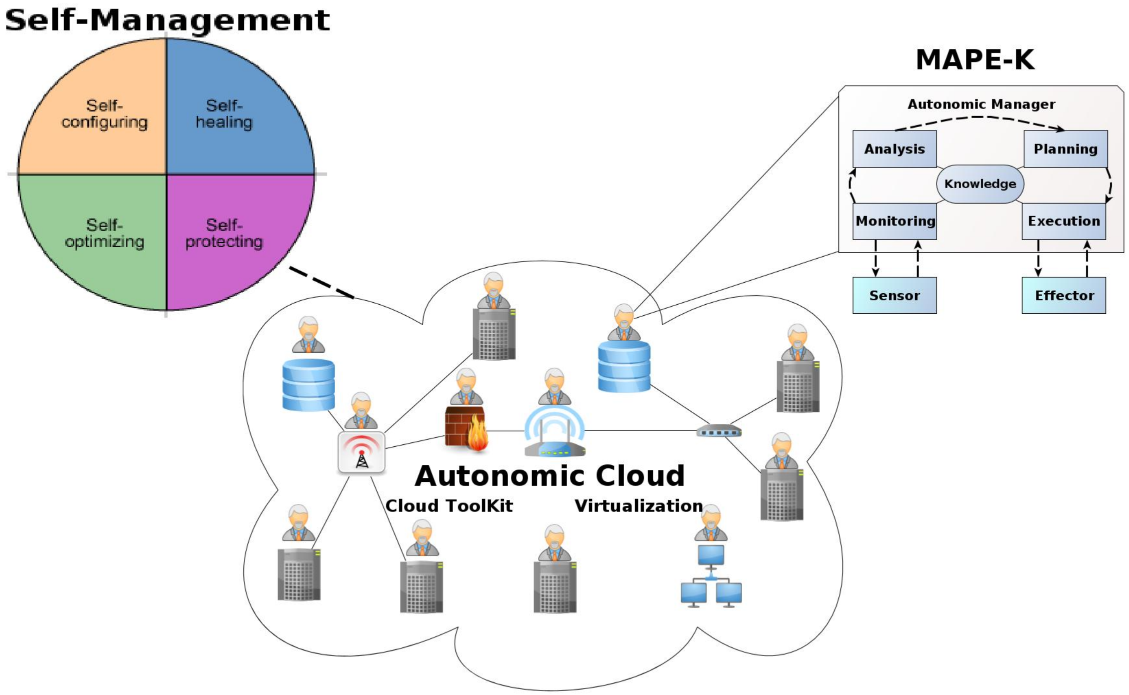

**Figure 8:** Example of an autonomic cloud; the agents employ the MAPE-K loop and communicate among themselves to enact the self-management properties.

the MAPE-K loop and interact with other agents to generate the local and the global view of the system, thus enforcing the self-* properties. Figure 8 illustrates a possible scenario for an autonomic cloud (IaaS), with multiple nodes (servers, routers, storage) and the autonomic agents.

## 2.3 Autonomic Clouds

The management of cloud computing is a crucial and complex process which needs to be automated [BCL12]. The key characteristics of cloud computing, such as dynamism, scalability, heterogeneity and the virtualization layer make clouds considerably more complex than legacy distributed systems [NDM09, BCL12], thus hindering the management and integration of its resources.

Autonomic clouds exhibit the ability to implement the self-* properties in accordance with the policies of the system and the SLAs of the provided services; however, achieving overall autonomic behaviour is still an open challenge [PH05]. In this section, we discuss solutions developed to enact the application of the properties of autonomic computing to clouds.

Buyya, Calheiros and Li [BCL12] propose an architecture for the SaaS delivery model considering multiple aspects of autonomic systems. In particular, they designed an application scheduler that considers aspects, such as energy efficiency, dynamic resources and QoS parameters. Moreover, the architecture has a mechanism that implements an algorithm for detection of DDos attacks. However, these aspects partially provide only the self-protection and the self-optimization properties and at the software level (SaaS), i.e. other autonomic properties and layers are not addressed. Similarly, most of the works in the literature apply autonomic computing concepts to the management of specific tasks in the domain, e.g. to service scheduling [RCV11, LC08, SRJ10, SSPC14, QKP$^+$09], to multi-cloud frameworks [KRJ11] and to security [WL12, KAC12, MWZ$^+$13, WWZ$^+$13].

An exception is the work of Solomon et al [SILI10] that describes a software architecture to enable autonomic clouds, which are divided into eight loosely coupled components (each of the self-* properties needs to implement these components): sensors, filters, coordinator, model, estimator, decision maker, actuator and adaptor. This division is based on the authors' previous work [SI07], which was devised for self-optimization in a real-time system. However, as most works in the area, see e.g. [Bra09, MBEB11], the authors give only a high-level account of the architecture due to the heterogeneity of the domain; leave the fine-grained details unspecified; and they neither implement nor validate this architecture. Moreover, the architecture is not specific for clouds and, hence, does not reflect important characteristics of the domain, such as, virutalization, elasticity, heterogeneity.

In light of these limitations and of the broad scope of autonomic solutions for clouds, we focus on the following areas of the management of autonomic clouds: (i) the service definition using SLAs; (ii) data collection and integration with the autonomic system; and (iii) the knowledge discovery process, which transforms operational data into knowledge to feed the autonomic managers and improve the decision-making.

## 2.4 Service-Level-Agreements

A SLA is the formalization of the performance goals and the description of a service, being an essential component of legal contracts between parties (in most cases being the contract itself). It typically identifies the parties involved in the business processes and specifies the minimum expectations and obligations among them [BCL+04]. In particular, it contains the obligations, permissions, quality of service, scope, objectives and the conditions under which a service is provided. This contract can either be informal, which is valid only between parties, or legally binding.

The reasons for using SLAs are manifold. They can range from process optimization and decision support, to the definition of measurable and legally enforceable contracts [PSG06]. It can be expressed in several forms but large-scale and dynamic systems, such as cloud, require machine-readable and non-ambiguous contracts to verify the compliance of the provided service and to apply the duly penalties in case of violation. Table 1 exemplifies the terms of a SLA.

**Table 1:** Example of SLA metrics

| **SLA Metric** | **Objective** |
|---|---|
| Response Time | $< 5$ ms |
| Availability | $> 99$ % |
| Price | $< 0.4$ EUR / Hour |

### 2.4.1 Life Cycle

The SLA has no widely accepted life cycle. In this subsection, we discuss some well-known life cycles in light of the cloud computing domain, highlighting their aspects and differences.

SLAs are not employed exclusively in clouds. They are used to other types of services, such as software development and network provision. Therefore, many definitions of SLA life cycles, mainly related to software development, include pre-installation phases, e.g. design and development of the software. In the cloud domain, these pre-installation phases are not considered in the SLA as they affect exclusively the provider. From the consumer perspective, the service already exists (services are not developed for single consumers) or the consumer is responsible for the development and deployment of his own software (e.g PaaS model), which also is not included in the SLA. Hence, life cycles that include or focus on these pre-installation phases are not considered in this work.

Specifically for web services, Ron and Aliko [RA01] divided the SLA life cycle into three major phases: (i) creation phase, in which the consumers search for a compatible provider, negotiate the service and define a SLA; (ii) operation phase, in which the agreed service is provided and monitored; and (iii) removal phase, in which the SLA is terminated and service is removed from the resources.

A more detailed life cycle was presented in [WB10]. The author divided it into six phases:

1. Discovery - Consumers and providers search and match parties that are able to fulfil their requirements. In environments, such as clouds, a large collection of various services is available. Moreover,

offers are dynamic (a provider can change, remove or add offers) and, therefore, finding the most adequate service is challenging in the area;

2. Definition - The parties negotiate and formalize the terms of the agreement (e.g. QoS parameters);
3. Establishment - The parties commit themselves to the agreement and the provider deploys the service. The first three phases of this definition correspond to the first phase in Ron and Aliko's life cycle;
4. Monitoring - The compliance of the service performance with the specification of the SLA is verified. This phase plays an essential role in the SLA life cycle as it is necessary to detect whether a SLA is violated or achieved;
5. Termination - The services and the associate configurations are removed from the providers with the termination of the SLA. A SLA can terminate due to a violation, its expiration or by an agreement between the parties;
6. Enforcement of Penalties - If the SLA was terminated due to a violation, the corresponding penalties are invoked.

The presented SLA life cycles define the sequence of changes of the SLA from the perspective of all parties involved in the contract. However, in the work of Keller and Ludwig [KL03], the authors define a SLA management life cycle from the perspective of the provider of the services. The main difference to the previous models is the inclusion of service management definitions, such as corrective actions which are not specified in the SLA but are employed by the provider to avoid violations. The life cycle is defined as follows:

- Negotiation and Establishment - Consumers retrieve the SLA offers (or request a service), the parties negotiate, and the SLA is established. The result of this phase is a single document, which comprises the service's description, characteristics and the obligations of the involved parties;

- Deployment - It validates the SLA and distributes the obligations and permissions to all parties. Moreover, the service is deployed on the resources and included in the SLA management framework;
- Measurement and Reporting - The information about the system and its behaviour is retrieved. Then, such information is evaluated and, if a threshold is met, the involved parties are notified;
- Corrective Management Actions - Once a SLA has been violated or the system detects that the SLA could be eventually violated, it plans corrective management actions to avoid these violations. It is worth noticing that the SLA is non-modifiable and, consequently, the actions taken to avoid violations affect exclusively the provider's system;
- Termination - The SLA terminates when a breach on the SLA occurs, on mutual agreement or when expired.

A similar approach is defined in the scope of the SLA@SOI project [WBYT11]. Figure 9 [WBYT11] shows the phases of this life cycle. The *design and development* of the service is executed by the provider. In the *Service offering* phase providers and brokers define SLA specifications, e.g. templates. When a consumer finds a service compatible with his needs, the parties *negotiate*. The SLA is the result of this negotiation and, according to this agreement, the provider prepares the resources for the service (*provisioning*). Then, the service is instantiated and the *operations*, such as monitoring and adjustments to enforce the SLA, are executed. Finally, when the SLA is terminated, the service is *decommissioned*.

The management of the service being provided is the focus of the life cycles proposed by [WBYT11, KL03]. However, in this thesis, we opt to take into account the life cycle proposed in [WB10]. This approach abstracts the management of the service from the life cycle, which does not concern the consumer of the service. Moreover, the independence from a single party's standpoint enables auditors to supervise the provision of the service.

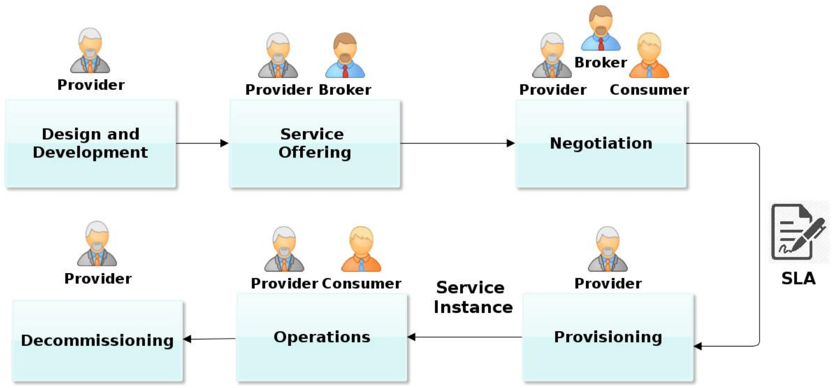

**Figure 9:** SLA life cycle.

### 2.4.2 Specification Languages

Most providers in the cloud domain make available only a description in natural language of the general terms and conditions of their services. Not providing machine-readable or formal semantics for SLAs has many drawbacks. For instance, it creates ambiguity on the SLA interpretation and makes the automation of the SLA's life cycle impracticable.

Nevertheless, several languages to specify and to automatize the SLA life cycle were proposed [WB10]. In Appendix A we summarise the most important SLA definition languages and, in this section, we present the most important SLA definition languages from the cloud domain standpoint, highlighting their strengths and weaknesses. These languages and the ones included in Appendix A are the base for Chapter 3, in which we further analyse them in light of the requirements of SLA in the cloud domain.

#### SLA*

The SLA* [KTK10] language is part of the SLA@SOI project, which aims at providing predictability, dependability and automation in all phases of the SLA life cycle.

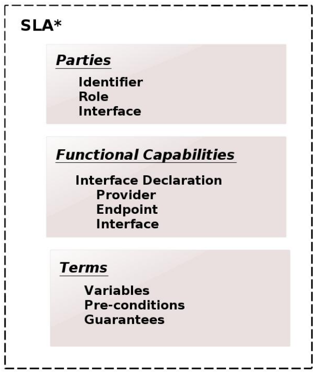

**Figure 10:** Structure of SLAs defined in the SLA* language.

SLA* is inspired by WS-Agreement and WSLA and, in contrast to the described languages that support only web services, aims at supporting services in general, e.g. medical services. To achieve this aim, the authors specified an abstract constraint language which can be formally defined by plugging-in domain specific vocabulary.

Agreements in SLA* comprise: the involved parties, the definition of services in terms of functional interfaces and agreement terms. The agreement terms include: (i) variables which are either a "convenience" to be used in place of an expression (shorthand label) or a "customisable" which expresses "options" (e.g. *<4 and <10*); (ii) pre-conditions that define the cases in which the terms are effective (e.g. week days, business hours); and (iii) guarantees that describe states that a party is obliged to guarantee (for example, a SLO) or an action should be taken. This structure is depicted in Figure 10.

The benefits of the language are: it supports any kind of service; it is extensible; it is expressive; has a framework which covers all phases of the SLA life cycle and; was tested in different domains. Nevertheless, the SLA* specification lacks precise semantics due to its multi-domain approach and the support to brokerage. Moreover, it requires the development of specific vocabulary for each domain.

**SLAng**

The first version of SLAng is presented in [LSE03]. However, in [Ske07], Skene, one of the authors of the original paper, claims that this language was highly imprecise and open to interpretation. Hence, Skene decided to continue the development of SLAng to addresses these issues. Therefore, in this work, we review the SLAng developed by Skene, i.e. the improved version of the language (we refer to his doctoral thesis [Ske07] for the full specification of the language).

SLang specification is presented as a combination of an EMOF [Obj04b] structure, OCL [Obj03] constraints and natural language commentary. A SLA defined in SLAng is the instantiation of the EMOF abstract model, which can be concretely instantiate in several ways, for example, using Human-Usable Textual Notation (HUTN) [Obj04a] or XML Metadata Interchange (XMI) [Obj14] (it can also include comments in natural language to facilitate the understanding). The OCL constraints are used to refine the model and define, to some degree, the semantics of the SLA.

To illustrate the OCL use in SLAng, Listing 2.1 presents the specification of the total down time for an operation of a service. The constraint selects and sums all non-scheduled events in which an operation failed or which the latency is higher than the specified maximum latency.

**Listing 2.1:** Extract of a SLA specified in OCL.

```
--Total downtime observed for the operation
let totalDowntime(o : Operation) : double
o.serviceUsage -> select(u (u.failed or u.duration >
maximumLatency) AND schedule -> exists(s |
s.applies(u.date)) ) -> collect(u | downtime(u.date,
o)) -> iterate( p : double, sumP : double | sumP + p)
```

As depicted in Figure 11, the EMOF model consists of:

- *Administration Clauses*, which define the responsibilities of parties in the SLA administration. This administration sets constraints to define how the SLA is administrate and which party is in charge

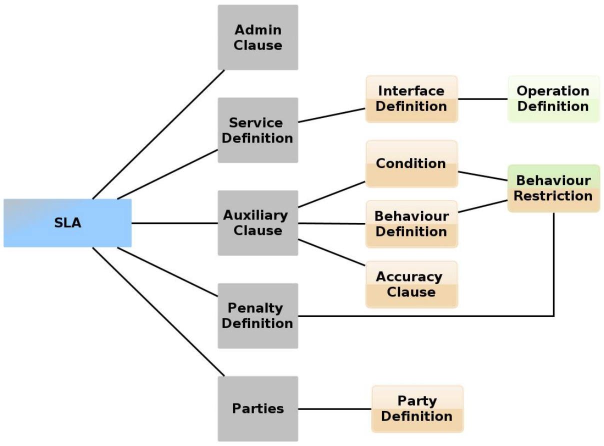

**Figure 11:** Structure of SLAs defined in the SLAng language.

of this administration, for example, they express who can submit evidences of SLA violations;

- Service's *Interface Definitions*, including the operations available for this service;
- *Auxiliary Clauses*, which are abstract constructs composed of: *Conditions* to associate the behaviour of the service to a constraint; *Behaviour Definition*; and *Accuracy Clause* which establishes the rules to assess the accuracy of service measurements. Then, these measurements are employed to verify violations and apply penalties. Also, conditions, behaviours, penalties and parties are used to create constraints on the service behaviour (e.g. availability), named *Behaviour Restrictions*;
- *Penalty Definitions*, which defines the actions that should be enforced in case of violation;
- *Parties Description*, which describe the involved parties.

In contrast to the previous languages, SLAng is domain-specific, devised for Application-Service Provision (ASP). Its main strengths are: low ambiguity due to the correspondence between elements in an abstract service model and events in real world [LF06]; emphasis on compatibility, monitorability and constrained service behaviour; and domain-specific vocabulary for IT Services.

The main limitation of SLAng is the complexity to: fully understand its specification; create SLAs using this specification; and extend the language. Its limitation is due to: the combination of techniques as OCL and EMOF, which require technical expertise to use [Ske07]; its expressibility; and its formal nature. Moreover, considering the heterogeneity of the IT services domain, the language requires an extensive analysis effort by experts and the definition of extensions of similar size to SLAng core language itself [Ske07] to be deployed in real-world cases. These efforts and complexity lead not only to difficulties to users but also to high costs for its adoption.

### CSLA

Cloud Service Level Agreement[3] [Kou13, SBK+13] is a specification language devised for the cloud domain.

Its structure is similar to WS-Agreement and is presented in Figure 12. Validity describes the initial and expiration dates for the SLA. The parties are defined in the Parties Section of the agreement while the template is used to define the service, the associated constraints, the guarantees related to these constraints, the billing scheme and the termination conditions.

A novelty of the language is, in addition to the traditional fixed price billing model, the possibility to use the pay-as-you-go model. Moreover, CSLA introduces the concept of fuzziness and confidence. The former establishes an error margin for a metric in the agreement. The latter defines the minimum ratio of the enforcements that the metric values do not exceed the threshold, permitting the remaining measures to exceed the threshold but not the fuzziness threshold. For example, the threshold for the response time of a service is 3 seconds, the fuzziness value is 0.5

[3]The language was presented in a short paper [KL12] but it is not available on-line.

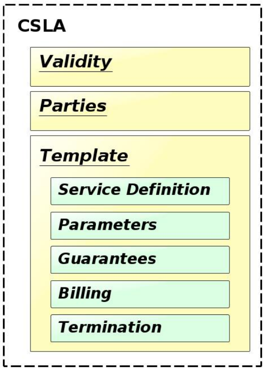

**Figure 12:** Structure of SLAs defined in the CSLA language.

and the confidence is 90%. In every 100 requests, minimum 90 need to have values between 0 and 3 and maximum 10 can be between 3 and 3.5 without violating the SLA.

As drawbacks, the language is neither formally defined, nor supports parties with important roles (e.g. the broker), nor comprehends other dynamic aspects of the cloud.

### Overview

In the present subsection, we describe the machine-readable solutions to define SLAs for services. Apart from SLA*, which enables the specification of SLA for electronic services, other analysed languages target web services or a specific subgroup of this area.

Major challenges for the adoption of abstract languages for SLA specification (e.g. SLA*) are the creation of the vocabulary for a domain (e.g. metrics) [LF06], and to assure that all parties share and understand the definitions in that vocabulary.

SLAng and CSLA are domain-specific and this problem has a lower impact. Nevertheless, SLAng is rather complex and requires experts to understand the specification and adapt it to each use case. CSLA, instead, provides neither the formalism for the SLA specification, nor captures important characteristics of the domain, e.g. broker support.

In light of these considerations and of the cloud requirements (investigated in Chapter 3), we propose a SLA specification language for cloud computing in Chapter 3.

### 2.4.3 SLA Management Solutions

Several projects propose different degrees of SLA-aware management of resources. Follows the analyses of the most relevant ones. SLA@SOI [SLA14] focus on service-oriented architectures, including non-IT services (e.g. medical services). From the life cycle standpoint, the project emphasises the SLA prediction and risk analysis. However, they also proposes a SLA definition language and a monitoring architecture, solutions which are detailed and analysed in Chapter 3 and Chapter 4. Yet, the target platform of SLA@SOI is not cloud computing; therefore, they do not account important particularities of the domain, such as data incongruence, the amount of operational data and virtualization.

Cloud-TM [Clo15] focuses in data centric cloud applications and does not consider other cloud models. BonFIRE [Bon15] project neither considers the management of heterogeneous resources and applications, nor the autonomic components of clouds. Due to the broad scope and complexity of the SLA life cycle, most works in the literature propose solutions which cover only a specific phase of SLA life cycle. The works related to this thesis which cover only a specific aspect of the life cycle will be analysed in their respective chapter.

The SLA management is fundamental for the autonomic management of cloud. Therefore, although we focus in autonomic clouds, the solutions here proposed cover also some important phases of the life cycle. In particular, we address the SLA definition, monitoring and the knowledge discovery for the evaluation and enforcement of SLAs.

## 2.5 System Monitoring

The information collection process is essential to analyse the status of the system and of the service; thus, it is necessary to verify compliance of SLAs. Moreover, this processes play an important role in the information and knowledge generation in clouds and autonomic systems. Therefore, in this section, we list the basic concepts of the system monitoring area, including the properties of the system, which are used in Chapter 4 for the proposed monitoring solution.

Monitoring systems continuously collect information, verify the state of its components and of the system as a whole. It provides the data, information and knowledge, which are necessary to measure, assess and manage the hardware and software infrastructure. Moreover, these measures are also used by auditors and users to verify the performance and correctness of a system, enabling, for instance, the SLA conformance check.

There are various approaches to collect information from a system. The most common are the *active* and *passive* monitoring. Active monitoring simulates the usage of the system, monitors this simulation and collects the monitored data. Passive monitoring does not generate extra load to the system since it retrieves and analyses the data available in the system, typically generated by its users. For instance, to test whether a route of a network is available, using passive monitoring would require the interception or a copy of a package to be analysed and to assess whether it follows the target route. On the other hand, using the same example, with the active approach, the monitoring system would generate and send a package which uses this route and test whether it has arrived.

To provide the status of the system to the decision-makers the monitoring system has to maintain some properties. Considering our context and in accordance with the definitions of Aceto et al [ABdDP13], we selected twelve properties. For the sake of simplicity we divided these properties into two big categories: the ones related only to the monitoring system itself, named *System*, and the ones associated to the collected data/information, named *Data*. The *System* category was further divided

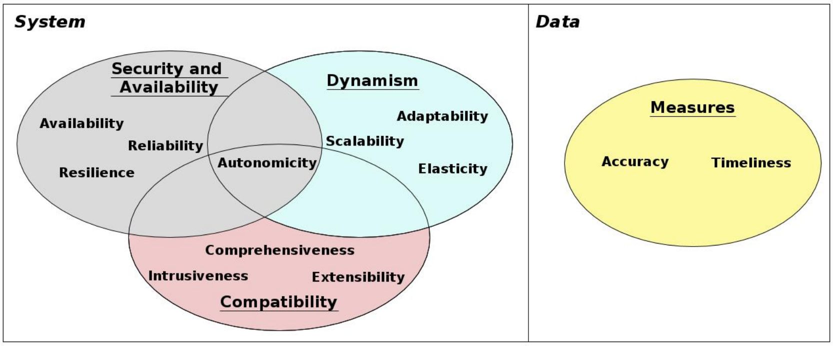

**Figure 13:** Classification of monitoring systems properties by type.

according to their areas: (i) Security and Availability; (ii) Dynamism; and (iii) Compatibility. The only property belonging to more than one group is *Autonomicity*, which is related to all groups under the System category. Figure 13 depicts this classification, which is discussed below.

### 2.5.1 System

*Autonomicity* is related to autonomic computing, which is detailed in Section 2.2. In particular, a monitoring system has this property if it implements all the self-* properties and, therefore, manages itself.

***Dynamism***: *Adaptability* is the capacity of the monitoring system to adapt to the host system load to avoid degradation of the performance of the system's aim activities due to the monitoring system's workload (non-invasiveness). This workload consists of the operational functions, such as collecting, processing, integrating, storing and transmitting information. To be non-invasive in dynamic environments, the monitoring process must tune and relocate its components according to the policies of the system, reacting or operating pro-actively. Moreover, the monitoring system not only has to adapt itself but also has to support drastic changes in its topology and organization. This property is called *elasticity*. Finally, the *scalability* property is related to the capacity of supporting a large number of probes and, at the same time, being non-invasive [CGM10].

***Security and Availability*:** A system is said *resilient* if the monitoring process is still functional to critical activities even after a high number of component's fail; the system is *reliable* if it can perform its functions under specific conditions for a period of time; and is *available* in case it responds to the requests whenever required (according to the system's specification).

***Compatibility*:** Monitoring components should support different types of resources (both virtualized and physical), types of data and multi-tenancy [HD10]. The property that considers this support is named *comprehensiveness*, while *extensibility* enables easy extensions of such support (e.g. through plug-ins). Finally, *intrusiveness* measures the significance of the modifications required in the system to integrate with the monitoring system.

### 2.5.2 Data

This category is linked to the way that the data is collected and delivered to the interested party.

Monitoring data is fundamental to assist the system to achieve its goals. However, the data should reach the system on time for an appropriate response (e.g. to replace a disk in case of failure). The property that enables the monitoring system to provide such information on time, in case of need, is named *Timeliness*.

*Accuracy* is also a desired property and is achieved when the monitoring system provides the information as close as possible to the real value in the configured abstraction of the system. The accuracy is essential for the management and decision making of the system.

## 2.6 Knowledge Discovery Process

Knowledge is one of the foundations of the decision-making process of autonomic clouds. Therefore, in this section, we review the literature that defines the knowledge hierarchy.

The origin and nature of knowledge are investigated by Epistemology, which is a branch of philosophy. However, the definition of knowledge is a matter of ongoing debate and may assume different forms according to the context. Therefore, after summarising some of the common interpretations of knowledge, we propose a definition from the autonomic cloud domain perspective and analyse the process of discovering knowledge into operational data.

A well known definition of knowledge was created by Wright in [Wri29]. It emphasizes the fact that knowledge is domain specific and that it must be based on solid foundations:

> "Knowledge signifies things known. Where there are no things known, there is no knowledge. Where there are no things to be known, there can be no knowledge. We have observed that every science, that is, every branch of knowledge, is compounded of certain facts, of which our sensations furnish the evidence. Where no such evidence is supplied, we are without data; we are without first premises; and when, without these, we attempt to build up a science, we do as those who raise edifices without foundations. And what do such builders construct? Castles in the air."

In light of this statement, it can be claimed that we need to understand the difference between simple facts and knowledge to discover knowledge. A well-established framework which attempts to capture the relation between "what we see" and "what we know" is the *"DIKW"* [Zel87] [4] hierarchy, which divides the knowledge discovery process into *data, information, knowledge* and *wisdom*.

*Data*, in this context, represents the observable evidences, symbols and stimuli of physical states, acquired from inspection of the world [Sch11]. Typically, data denotes confusing and disconnected facts, which restricts their use.

The *information* has a descriptive nature and is able to answer questions, such as "who", "what", "how many" and "when". Intuitively, information can be defined as meaningful and useful data [BKC00].

[4] Its origin is not clear, but seems that it first appeared in the work of Zeleny.

*Knowledge* is a set of expectations or rules, which provides a clear understanding of aggregate information. It represents the recognition of patterns in the information together with experience and interpretation. It answers questions, such as "how" and "why". Another commonly cited definition for knowledge is the one of Devenport and Prusack [DP98]:

> "Knowledge is a fluid mix of framed experience, values, contextual information, expert insight and grounded intuition that provides an environment and framework for evaluating and incorporating new experiences and information."

Finally, *wisdom* is knowledge that, when in a framework, can be employed to generate new knowledge or to take decisions. It provides the ability to judge based on knowledge and is composed of proved knowledge, heuristics and justifications.

The knowledge discovery process selects, aggregates, filters and uses learning and other models to produce knowledge from data. The transformation of operational data into knowledge for autonomic clouds is detailed in Chapter 4, while in Chapter 5 we propose a methodology to discovery knowledge.

In autonomic cloud domain, knowledge plays a major role. The MAPE-K loop depends on knowledge to feed the decision-making system and enact the self-* properties. However, the scale of clouds, the incongruence of the data and the characteristics of autonomic clouds, such as dynamism, virtualization and the measures to improve the security of the data, which obfuscate information, hinder the knowledge discovery process.

Several techniques were devised to handle a single or a subset of these characteristics. For instance, machine learning techniques, such as [AHWY04, GRS00], which can cope only with the scale of the clouds, while [SLS$^+$09, GMR04] focus on handling its dynamism, updating the models on-line to adapt to new services. However, to the best of our knowledge, the knowledge discovery process has not been analysed in the context of autonomic clouds, considering all its characteristics; and therefore, there remains a need for such analysis and for the development of a solution able to cope with these characteristics.

Furthermore, several solutions require knowledge about the environment to generate new knowledge for specific applications. For instance, to identify services with anomalous behaviour, a solution needs the knowledge of the similarity among services. However, most solutions in the cloud domain implicitly assume the: homogeneity of the resources and services; preparation and normalisation of the data; or good representation of the relations of data features. However, these assumptions are not valid for autonomic clouds.

To address this gap in the literature, later in Chapter 5 we define the requirements of the domain and propose a methodology to generate a specific type of knowledge, which can be applied in different context: the similarity among services.

## 2.7 Summary

We provide an overview of the main paradigms related to autonomic clouds and on works with a similar scope.

In the first part of the chapter, we have defined the most important aspects of cloud computing. Then, we described autonomic computing, which is employed to address the complexity of clouds. Next, we analysed the existing solutions for the management of our target domain, i.e. autonomic clouds. Due to the broad scope of the domain and the complexity of its management, few works cover all models and levels of the domain. Commonly they discuss only a high-level account of an architecture to the management of autonomic clouds and leave the fine-grained details unspecified.

As clouds are service-oriented; hence, we also presented the service management perspective and describe the existing proposals for the SLA life cycle. Considering the broad scope of the life cycle, there is a lack of fine-grained specification of its phases, mainly from the knowledge collection and discovery standpoint.

Additionally, we discussed the approaches and concepts which provide the foundations for understanding the works specifically related to the types of knowledge covered in this thesis (definition of SLA, information collection and knowledge discovery).

Regarding the definition of SLAs, we described the existing works, which have significant gaps, such as lack of: support for brokerage, formal specifications and mechanisms to support the dynamism of clouds.

Related to the information collection, we presented the monitoring system properties required by the cloud domain, which will later assist the analysis of the existing solutions and to define the scope of our solution in the area.

Finally, we presented definition for knowledge, information and data, and discussed the knowledge discovery process. Moreover, we described the works related to the knowledge discovery process in the domain.

Despite the fact that many works design solution for the management of autonomic clouds, they assume that the knowledge is already available. Therefore, to address this gap in the literature, this thesis focuses on the generation and provision of this knowledge for the autonomic management of clouds.

# Chapter 3

# SLAC: A Language for the Definition of SLAs for Clouds

The cloud computing paradigm provides to consumers elastic and on-demand services over the Internet. Hence, an important aspect of the clouds regards the management of services, which requires the definition of the deployed services to feed the decision-making process (e.g. to activate the monitoring of these services, to evaluate their performance and to adapt the cloud to provide the promised quality of service).

With the recent commercial growth of the cloud paradigm, several new service offers emerged. However, the vast majority of providers offer only simple textual description of the terms and conditions of their services. This approach has many drawbacks; for instance, ambiguity and unfeasibility of the automation of the SLA's evaluation, of the search for services and of the negotiation of contract terms. The importance of a machine readable agreement is highlighted also by the need of formal guarantees that the delivered services are compliant with the agreed terms since cloud users may outsource their core business functions onto the cloud [DWC10].

Several languages to specify machine readable SLAs and to automatize their evaluation and negotiation were proposed [WB10]. Yet, we argue that these languages are not able to cope with the set of distinctive characteristics of clouds, such as multi-party agreements, deployment models and the growing importance of the broker role.

Considering this gap in the SLA definition languages, we propose SLAC [UTD14], which is a language to define SLAs specifically devised to the cloud computing domain. Our approach supports the broker role and multi-party agreements. Moreover, SLA is easy to use and provides vocabulary for clouds.

In order to present this language, in this chapter, we: (i) define the requirements of the domain and compare the existing languages to SLAC; (ii) describe its syntax and semantics; (iii) discuss business extensions; (iv) detail the implementation of a framework that supports this language; and (iv) examine a use case employing this framework.

## 3.1 Support to Cloud Requirements in the Existing SLA Definition Languages

Cloud computing services have a distinctive set of characteristics, as detail in Chapter 2. In this section, we analyse the requirements of SLAs in the domain and their impact on the contracts which define services. Moreover, we evaluate the existing languages for the definitions of SLA based on their support for these requirements and examine the differences among these languages and our proposed language.

### 3.1.1 SLA Requirements in the Cloud Domain

In order to support all deployment models and express real-world agreements, a SLA language in the cloud domain must support the definition of *multiple parties*. This feature is not restricted only to the roles present in cloud computing (detailed in Section 2.1.4) but needs also to support multiple roles for a single party and multiple parties with the same role. Among these roles, the *broker* requires special attention due to its importance in the domain. Therefore, the language must support different types of brokers, which are described in Section 2.1.5.

A SLA definition language should consider that many consumers and, possibly, other parties are not domain experts; therefore, the language for the definition of SLA should be *easy* to understand and tools should be available to facilitate this process. Moreover, an important characteristics in a SLA definition language is the extensions necessary to *support the domain* and the availability of *domain specific vocabulary*, i.e. if a language needs to be adapted to the context. This adaptation to the context needs experts and development time, which has an impact on the cost of the solution. Therefore, a SLA language for the domain, not only needs to be simple for all the parties but also needs to support the domain.

Cloud computing has a key role in industry and academia. Many offers of cloud services are available and their number is expanding. Moreover, clouds considers multiple billing models (e.g. pay-per-use) in its definition. Therefore, a SLA definition language for the domain should support its main *business aspects*, including such billing models.

Considering that, in cloud, virtually anything can be provided as a service, the *service delivery models* in cloud computing are many. The most important models, i.e., IaaS, PaaS and SaaS (we refer to Section 2.1.2 for further details), should be supported in the SLAs.

Ambiguity in SLAs (and contracts in general) is a major challenge since a SLA is the formalisation of the guarantees of a service. Hence, a SLA language must have *formal definitions* (semantics) to avoid disputes over the interpretation of contracts.

Finally, a SLA definition language requires *tools* for evaluating and processing SLAs to support the language in the management of autonomic clouds.

### 3.1.2 Comparison Between SLAC and the Existing Languages

In view of such requirements, we carried out a survey on the existing languages and verified their support for these requirements (please, see Section 2.4.2 and to Appendix A). The comparison among the existing

**Table 2:** Comparison of the SLA definition languages according to the domain requirements.

| | Features | WSOL | WSLA | SLAng | WSA | SLA* | CSLA | SLAC |
|---|---|---|---|---|---|---|---|---|
| *General* | Cloud Domain | - | - | - | - | - | ■ | ■ |
| | Cloud Service Models | - | - | - | - | - | ■ | □ |
| | Multi-Party | - | - | - | - | - | □ | ■ |
| | Broker Support | - | - | - | - | - | - | ■ |
| | Ease of Use | □ | □ | - | □ | □ | □ | ■ |
| *Business* | Business Metrics | □ | □ | □ | □ | □ | □ | ■ |
| | Price schemes | □ | □ | - | □ | □ | □ | ■ |
| *Formal* | Syntax | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| | Semantics | - | - | □ | - | - | - | ■ |
| | Verification | - | - | □ | - | - | - | ■ |
| *Tools* | Evaluation | ■ | ■ | ■ | ■ | ■ | □ | ■ |
| | Free and Open-Source | ■ | □ | ■ | ■ | ■ | ■ | ■ |

SLA definitions is summarised in Table 2.

For the sake of simplicity and standardization, we classify the languages in three levels: the ■ symbol represents a feature covered in the language, □ stands for a partially covered feature and - represents a not covered feature. The criteria used for their classification in each characteristics is detailed as follows:

- *Cloud Domain* - This criterion defines when a language was devised for the cloud domain. Languages for a broader domain (e.g. IT Services) are not considered to cover the criterion;

- *Cloud Delivery Models* - Fully supported when a language covers IaaS, PaaS and SaaS, and partially supported when only one of these models is covered;

- *Multi-party* - Fully supported when the language enables the use of: (i) all roles defined in Section 2.1.5; (ii) multiple-parties with the same role; and (iii) multiple roles for the same party. Partially supported when at least one of the characteristics is present;

- *Broker support* - To support the broker, a SLA definition language should enable the specification of: (i) the parties involved in each term, which define the parties in charge of providing and consuming the service; (ii) offers and requests, which enable the creation of specification of services needed by consumers and of services available by providers; (iii) the role of the broker in the party's definition; and (iv) the actions and metrics which enable the parties to state who is the responsible for the service, who consumes it, who pays for it and the parties involved in each action. When at least two of these characteristics are supported we determine that the language partially fulfils this criterion;

- *Ease of Use* - This criterion is considered fully supported when a language is ready to be used in the domain (without or with minimal extensions) and, at the same time, it is easy to understand, also for human actors. If a language has only one of these characteristics it is considered as partially supported;

- *Business Metrics and Actions* - Fully supported when the language provides metrics related to the business aspects, such as performance indicators (KPIs) and enables the use of actions which express common business behaviours, for example the payment of penalties. It is partially supported if at least one of them is available in the language;

- *Price schemes* - The language should consider flat and variable pricing schemes and the main models of the domain, which are fixed pricing, bilateral agreement, exchange, auction, posted price and tender (detailed later in Section 3.3.1). A language offers partial support when it includes at least one type of variable pricing;

- *Formal Syntax* - Supported only when a formal definition of the syntax, e.g using BNF, is available;

- *Formal Semantics* - Fully supported if a formal definition of the SLA evaluation process is available. By formal we intend the use of any recognized formal tool (e.g. operational or denotational semantics) to avoid ambiguity on its interpretation. It is partially supported if possibly ambiguous tools are used in its definition;

- *Formal Verification* - Fully supported when the formal verification of agreement exists before execution time. We consider as partially supported if only the syntax is verified;

- *Evaluation Tools* - A language fully supports it when an implementation of a tool which evaluates the SLAs against the services' monitoring information is available. If only design time verification is available this characteristic is only partially supported;

- *Open Source and Freely Available* - Defines how the tools that support the language are made available to the users. The main options of software distribution are: free and commercial. Moreover, independently of the distribution model the tools can be open source or closed source. Thus, it is categorized as fully supported if a language is open source and free, and partially supported if a language has one of these characteristics.

Although we specify the criteria for the comparison of the languages, their classification is subjective since they are not quantitative. However, this comparison provides the base to understand to which extent the existing languages support the domain.

The results of the comparison among the language show that the existing languages do not support some of the main requirements of the domain. Among the major gaps for the definition of SLAs, there is the lack of support for multi-parties, the broker, the vocabulary of the domain and the formal specification of the SLA evaluation. In Section 2.4.2 we overview the general features of the languages that support most of the characteristics of clouds, i.e. SLA*, SLAng and CSLA. In this section, we analyse these approaches in light of the results of this comparison.

SLA* is a language-independent model for the specification of SLA. It supports the definition of SLAs in different domains and, at the same time, specifies, to a certain degree, the fine-grained level of details of the contract. However, the model does not support multi-party agreements, does not provide the formal semantics of the language and requires the definition of the vocabulary for the cloud domain.

SLAng, instead, is domain specific (IT services) and provides a formalism inherited from the tools used to specify the language. However, SLAng only considers two parties in the SLA, i.e. the consumer and the provider, which limits its use in the cloud domain (e.g., SLAng cannot be applied to scenarios involving the community cloud model or brokers). Moreover, the specification of SLAs using SLAng is rather complex, and requires the full comprehension of the model and technologies used in the specification of the language [Ske07]. Also, SLAng needs extensions (e.g. definition of the vocabulary) to be employed in the domain, which, as shown in [Ske07], requires extensive analysis efforts by experts and these extensions are of size that is similar to SLAng core language itself.

Finally, CSLA is a language devised for clouds. This SLA language covers all three cloud delivery models and proposes the definition of SLAs using a mechanism to support fuzziness in the numeric specification of metrics. However, this approach does not consider multi-party agreements, the deployment models (e.g. community cloud) or the prominent role of broker in the domain. Moreover, the support of CSLA for the delivery models is restricted to a few metrics and terms, which also requires extensions for real-world deployment.

In summary, our work differs considerably from the ones mentioned in the comparison table: it emphasizes the formal aspects of SLA; considers the particularities of the cloud computing domain (e.g. multi-party and brokerage); specifies the semantics of the SLA conformance verification; supports some of the most important business aspects of the SLA; and provides the base for dynamism in SLAs.

## 3.2 SLAC: Service-Level-Agreement for Clouds

In this section, we present the core of the SLAC language from the technical perspective. This core-language provides the basic ingredients that enable the description of a simple SLA for the cloud computing domain. Moreover, we discuss the semantics of the SLA consistency check and the SLA conformance verification with respect to the monitoring information of a deployed service.

### 3.2.1 SLAC Syntax

In this section, we present the syntax of SLAC, which is inspired by WS-Agreement and shares many features with the definitions and structure of this language.

The main elements of a SLA are: the description of the contract, the specification of terms and the definition of the guarantees for these terms. Differently from most of the existing SLA definition languages, SLAC does not differentiate service description terms and quality requirements.

Table 3 shows the formal definition of the syntax of the core language, which is defined in the Extended Backus Naur Form (EBNF). In this notation, the *italic* denotes non-terminal symbols, while `teletype` denote terminal ones. As usual, | indicates choice, ? after a symbol (or a group of symbols, when parenthesis are used) indicates an optional object, + requires at least one of the marked objects and * represents zero or more occurrences of the selected objects. A sequence of objects represented by + requires commas between each pair of instances of these objects, which were omitted in the syntax for the sake of simplicity (e.g. $Role^+$ stands for $role_1$,...,$role_n$). Also, some objects are prefixed by a related keyword that is omitted whenever the object is missing. Thus, e.g. the SLA (`id:` *Id* ... `terms:` *Term* `guarantees:`) would be written as (`id:` *Id* ... `terms:` *Term*), i.e. if no guarantee is specified then the keyword `guarantees:` is also omitted.

The non-terminal symbols $Id$, $Date$, $PartyName$ and $GroupName$ are implementation specific, hence, their details are intentionally left unspecified in the syntax of Table 3.

**Table 3:** Syntax of the SLAC language.

| | | |
|---|---|---|
| *SLA* | ::= | `id:` *Id* `parties:` *PartyDef* *PartyDef*$^+$ *Expiration* `term groups:` *Group*$^*$ `terms:` *Term*$^+$ `guarantees:` *Guarantee*$^*$ |
| *Expiration* | ::= | `valid from:` *Date* `expiration date:` *Date* |
| *PartyDef* | ::= | *PartyName*$^?$ `roles:` *Role*$^+$ |
| *Role* | ::= | `consumer` \| `provider` \| `carrier` \| `auditor` \| `broker` |
| *Group* | ::= | *GroupName* `:` *Term*$^+$ |
| *Term* | ::= | *Party* `->` *Party*$^+$ `:` *Metric* \| `[`*Expr*`,` *Expr*`]` `of` *GroupName* |
| *Party* | ::= | *Role* \| *PartyName* |
| *Metric* | ::= | *NumericMetric* `not`$^?$ `in` *Interval* *Unit* \| *BooleanMetric* `is` *Boolean* |
| | \| | *ListMetric* `has not`$^?$ {*ListElement*$^+$} `or` {*ListElement*$^+$}$^*$ |
| *NumericMetric* | ::= | `cCPU` \| `RT_delay` \| `response_time` \| `RAM` \| `availability` \| `jitter` \| ... |
| *Interval* | ::= | `]`*Expr*, *Expr*`[` \| `]`*Expr*, *Expr*`]` \| `[`*Expr*, *Expr*`[` \| `[`*Expr*, *Expr*`]` |
| *Expr* | ::= | *Literal* \| `infty` \| *NumericMetric*`(`*Parameter*`)` \| *GroupName* \| *Expr* *Operator* *Expr* |
| *Parameter* | ::= | `min` \| `max` |
| *Operator* | ::= | `+` \| `-` \| `*` \| `/` \| `>` \| `>=` \| `<` \| `<=` \| `and` \| `or` |
| *Unit* | ::= | `gb` \| `mb/s` \| `ms/min` \| `minute` \| `seconds` \| `ms` \| `month` \| ... |
| *BooleanMetric* | ::= | `back_up` \| `replication` \| `data_encryption` \| ... |
| *ListMetric* | ::= | `operating_systems` \| `jurisdiction` \| `hypervisor` \| ... |
| *ListElement* | ::= | `occi` \| `ec2` \| `kvm` \| `xen` \| ... |
| *Guarantee* | ::= | `on` *Event* `of` (*Party* `=>` *Party*$^+$ `:`)$^?$ *GuaranteeMetric* `:` *ConditionAction* |
| *GuaranteeMetric* | ::= | (*GroupName* `:`)$^*$ *NumericMetrics* \| (*GroupName* `:`)$^*$ *ListMetrics* |
| | \| | (*GroupName* `:`)$^*$ *BooleanMetric* \| (*GroupName* `:`)$^*$ *GroupName* \| `any` |
| *Event* | ::= | `violation` |
| *ConditionAction* | ::= | (`if` *Expr* `then` *Action*$^+$)$^+$ (`else` *Action*$^+$)$^?$ \| *Action*$^+$ |
| *Action* | ::= | (*Party* `=>` *Party*$^+$ `:`)$^?$ *ManagementAction* |
| *ManagementAction* | ::= | `notify` \| `renegotiate` |

The *description* of a *SLA* comprises a unique identification code (*Id*), the definition of at least two parties and the specification of the period of validity of the SLA through its *Expiration* definition. A party is constituted of an optional *PartyName* and one or more *Role*s. The definition of multiple roles for a single party enables the support of scenarios, such as community clouds, in which a provider can be also a consumer. Furthermore, the definition of only roles, instead of a specific party, enables the creation of templates, both for the definition of offers and for the definition of requests.

The *terms* of the agreement express the characteristics of the service along with their respective expected values. Each SLA requires the definition of at least one term, which can be either a *Metric* or a *Group* of terms. We illustrate the definition of a term in two figures (abstracting some details for the sake of comprehension). Terms composed of a metric are represented in Figure 14, while terms composed of the instantiation of a group in Figure 15. In this figure, a white arrow represents choice, a line with a black diamond head stands for a contained element and a line with a white diamond head represents an existing contained element.

In the SLAC language the parties involved should be defined in each term, i.e. the party responsible to fulfil the term (a single party) and the consumers of the service (one or more). This explicit definition contributes to support multi-party agreements, to reduce ambiguity for the definition of the monitoring responsibilities and to leverage the role of the broker in the agreements. Moreover, it can be used for improving the security aspects of the agreement, such as the integration with the authorization control; for instance, only parties involved in the term have access to it.

A *metric* can be of three types: *(i) NumericMetric*, which is constrained by open or closed *Intervals* of values (that can be defined explicitly in the SLA or inferred from the evaluation of an expression) and a particular *Unit* (e.g. milliseconds, gigabytes); *(ii) BooleanMetric*, which can assume `true` or `false` values; and *(iii) ListMetric*, whose values are in a list.

The metrics and the way to measure them are pre-defined in the language in light of the requirements of the cloud domain. As discussed in Section 5.6, they can be extended according to the needs of the involved

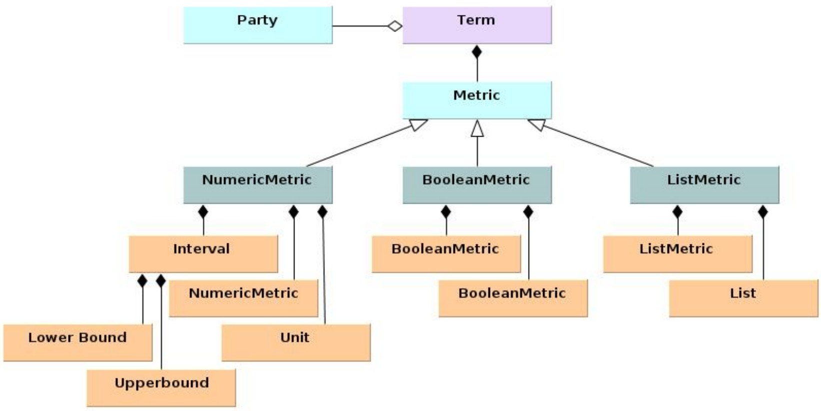

**Figure 14:** The definition of a term which defines a metric in the SLAC language.

parties. We have also pre-defined a set of valid items for each list metric available in the language; this helps avoid ambiguity or spelling errors in their specification. We refer the interested reader to [UTN14] for a complete account of metric definitions.

Expressions (*Expr*) have two main uses: the specification of interval values in numeric metrics and the specifications of conditions in guarantees. An expression can be a *Literal*, a *NumericMetric* with a parameter indicating if its upper or lower interval should be used, infinity (`infty`) that does not set a lower or upper constraint in the metric, or the composition of sub-expressions by means of mathematical *Operator*s. For example, the numeric metric `RAM in [(2 + 4 * cCPU(min) ), 20]` sets the minimum amount of memory RAM in the context as 4 times the lower bound of the *cloud CPU unit* required in the SLA plus 2. Notably, group instantiations cannot use the infinity value in the interval definitions.

Another feature of the language is the specification of granularities for terms using groups. A group of terms (*Group*) is identified by a name (unique in the contract) and is composed of one or more terms. Groups enable the re-use of the same term in different contexts. For example, in the use case presented in Chapter 1, let us suppose that a consumer needs

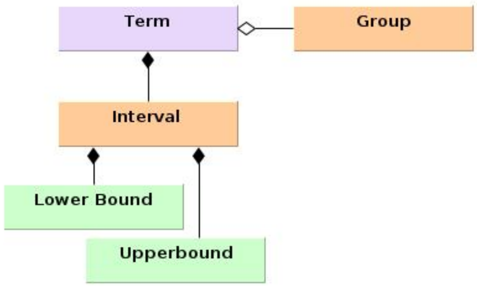

**Figure 15:** The definition of a term which instantiate a group in the SLAC language.

a centralized and two distributed VMs. In this case, the characteristics of each VM could be defined in a group, for example: `Centralized_VM` is a group defining 99% of availability, 4 cores and 16 GB of RAM for a machine, while `Small_VM` is a group specifying an availability of only 90%, 1 core and 1 GB of RAM. In this case, the two VMs have the same metrics but these are only valid in the context of the group.

Additionally, groups can include the instantiation of other groups. Continuing with the example above, to define a cluster that is constituted of the two less powerful VMs previously defined and specifying the maximum round-trip delay to the server as 0.6 milliseconds, the SLA could have a third group, named `Cluster`, which instantiate two `Small_VM` and the `RT_delay`. However, recursive definitions are not allowed, that is, a group cannot (directly or indirectly) refer to itself.

Notably, to use a group in a SLA definition, it is not sufficient to define it, as illustrated above, but it is also necessary to instantiate the group by specifying the number of instances. For example, previously we defined the group `Small_VM` that specified the characteristics of a type of VM; to actually deploy 2 instances of this group in the SLA, the term `[2, 2] of Small_VM` must be specified in the Terms section.

The concept of groups enables the use of the same term in multiple levels. When a group or the terms section employs a term that is also used in an instantiated group (the same involved parties and metric), only outer term definition is valid. For example, let us suppose that in the terms section a response time metric is defined and 2 `Small_VM`, which also include a term to define the response time, are instantiated. In this scenario, the definition of response time within the `Small_VM` is not taken into account, i.e. the response time in the terms section defines the response time of the service. When two groups at the same level are instantiated, the term definitions of both groups are valid only within the group. For example, if a `Small_VM` and a `Centralized_VM` are instantiated in the terms section of a SLA (without any response time specification) the response time will be different for each type of VM.

The final section of the SLA, the *Guarantees*, is optional in the agreement. Yet, it can play a significant role in the contract as it ensures that the terms of the agreement will be enforced or, in case of violation, it defines the actions that will be taken. Specifically, a guarantee refers to a term defined in *Terms* section of the agreement, i.e. a single term (i.e. *NumericMetric, ListMetric, BooleanMetric*), an instantiation of a group (*GroupName*) or to any term (using the reserved keyword `any`).

When an event occurs (e.g. a violation), the specified conditions are tested (defined by an *Expr*) and the execution of one or multiple *Action*s is requested (*ConditionAction*). This mechanism enables the definition of flexible conditions which can range from simple user defined thresholds to the percentage that each metric was violated. Notably, actions may require the specification of the involved parties according to the type of action.

In Table 4 we present a basic example of a SLA written in SLAC. In the example we define:

- Two parties, the $IMT$ as provider and $Rafael$ as consumer;
- The date in which the agreement starts to be valid and its expiration date;
- A group named $Tiny\_VM$, which represents a VM with exactly 1 cCPU and 1 GB of memory;
- A single term with the instantiation of 1 $Tiny\_VM$.

**Table 4:** Example of a simple SLAC SLA.

```
SLA
id: 123
parties:
  IMT
    role: provider
  Rafael
    role: consumer
valid from: 12/02/2012
expiration date: 13/03/2015
term groups:
    Tiny_VM:
      IMT → Rafael:cCpu in [1,1] #
      IMT → Rafael:RAM in [1,1] gb
terms:
    [1,1]of Tiny_VM
```

### 3.2.2 SLAC Semantics for the Evaluation of SLAs

The semantics of the evaluation of a SLA is formulated as a Constraint Satisfaction Problem (CSP) that verifies: (i) at negotiation-time, whether the terms composing the agreement are consistent; and (ii) at enforcement-time, whether the characteristics of the service are within the specified values.

More specifically, the formal semantics of SLAC is given in a denotational style. Denotational semantics [NN07] is based on the recognition that the elements of the syntax are symbolic realizations of abstract mathematical objects or functions. Although originally developed to describe the behaviour of imperative programming languages, the denotational approach is also appropriate for declarative languages (as required for SLAs) due to its capacity to translate the static and dynamic elements of the domain [Ske07]. This use of the semantics provides the formalism that is needed to express the meaning of the elements of a SLA definition language. The semantics is defined in Table 5.

The semantics of a $SLA$ is a function $[\![SLA]\!]$ that sends a pair composed of a set of group definitions and a constraint representing the semantics of $SLA$'s terms. This pair constitutes the CSP associated to the agreement, that will be solved by means of a standard constraint solver, in our implementation, the Z3 solver [MB08] as shown in Section 3.4.

In a $SLA$, a group is defined by an identifier and a set of terms. The semantics of a $Group$ is defined by a function $[\![Group]\!]^D$ that is parameterised by the set $D$ of all group definitions previously determined by the $SLA$ translation. Intuitively, every time a groups is translated, it is included in $D$, which is passed as parameter for the translation of the next group. This parameter is used in order to enable the instantiation of other groups in the current group. Due to its simplicity and readability, this approach was chosen instead of pre-parsing techniques; however, it implies that only ordered instantiation is permitted, i.e. within each group, only groups defined previously in the SLA (i.e. belonging to $D$) can be instantiated.

The semantics of a collection of terms $Term^+$ is defined by the logical conjunction of the constraints corresponding to each term. These constraints are generated by function $[\![Term]\!]_g^D$, which takes as parameters the set $D$ of group definitions and the parameter $g$, which is a group name, which is used to identify the context of the term and to generate the appropriate constraint identifier for term translation. Notably, the value $\emptyset$ for parameter $g$ is used for evaluating terms specified in the terms section of the agreement. The group definitions are initially added only to the set of definitions in the first field of the $SLA$ pair, i.e. the $Group$, which serve as parameter to the $Terms$ of the SLA. Therefore, a group is effectively considered in the $SLA$ only if instantiated in the terms section or used by a group, which is instantiated in the terms section.

Terms, used both by groups and in the terms section of the SLA, can be of two types: metric instantiations $Party -> Party^+ : Metric$ and group instantiations $\texttt{[}Expr\texttt{,}Expr\texttt{]}\ \texttt{of}\ GroupName$.

In case of metric instantiation, the parties are used as a parameter for the evaluation of the metric to avoid ambiguity (the same metric can be used in the same context for different). Notably, we opt for a compact notation to define metric and group names (e.g. $NM$ stands for $NumericMetric$) in the term translation.

**Table 5:** Semantics of the SLAC language.

$$
\begin{array}{rcl}
\llbracket SLA \rrbracket & = & \left\langle \llbracket Group^{*} \rrbracket,\ \llbracket Term^{+} \rrbracket^{\llbracket Group^{*} \rrbracket}_{\emptyset} \right\rangle \\
\llbracket Group_1 \ldots Group_n \rrbracket & = & \left\{ \underbrace{\llbracket Group_1 \rrbracket^{\emptyset}}_{D_1}, \underbrace{\llbracket Group_2 \rrbracket^{\emptyset \cup D_1}}_{D_2}, \underbrace{\llbracket Group_3 \rrbracket^{D_1 \cup D_2}}_{D_3}, \ldots, \llbracket Group_n \rrbracket^{D_1 \cup \ldots \cup D_{n-1}} \right\} \\
\llbracket GroupName \colon Term^{+} \rrbracket^{D} & = & GroupName \overset{\text{def}}{=} \llbracket Term^{+} \rrbracket^{D}_{GroupName} \\
\llbracket Term_1 \ \ldots\ Term_n \rrbracket^{D}_{g} & = & \llbracket Term_1 \rrbracket^{D}_{g} \ \wedge\ \ldots\ \wedge\ \llbracket Term_n \rrbracket^{D}_{g} \\
\llbracket Party -> Party^{+} : Metric \rrbracket_{g} & = & \llbracket Metric \rrbracket_{g,\ (Party \rightarrow \llbracket Party^{+} \rrbracket)} \\
\llbracket Party_1 \ \ldots\ Party_n \rrbracket & = & Party_1 \texttt{,}\ \ldots\ \texttt{,}\ Party_n \\
\llbracket NM \ \texttt{in [} Expr_1, Expr_2 \texttt{]}\ Unit \rrbracket_{g,p} & = & con(\llbracket Expr_1 \rrbracket, \llbracket Unit \rrbracket) \leq \llbracket NM \rrbracket_{g,p} \leq con(\llbracket Expr_2 \rrbracket, \llbracket Unit \rrbracket) \\
\llbracket NM \ \texttt{not in [} Expr_1, Expr_2 \texttt{]}\ Unit \rrbracket_{g,p} & = & (con(\llbracket Expr_1 \rrbracket, \llbracket Unit \rrbracket) > \llbracket NM \rrbracket_{g,p}) \vee (\llbracket NM \rrbracket_{g,p} > con(\llbracket Expr_2 \rrbracket, \llbracket Unit \rrbracket)) \\
\llbracket BM \ \texttt{is}\ Boolean \rrbracket_{g,p} & = & \llbracket BM \rrbracket_{g,p} == Boolean \\
\llbracket LM \ \texttt{has}\ \{LE^{+}\}\ (or\ \{LE^{+}\}_{i})_{i \in I} \rrbracket_{g,p} & = & \{LE^{+}\} \subseteq \llbracket LM \rrbracket_{g,p} \vee \bigvee_{i \in I} \{LE^{+}\}_{i} \subseteq \llbracket LM \rrbracket_{g,p} \\
\llbracket [Expr_1, Expr_2]\ of\ GN \rrbracket_{g}^{\{GN \overset{\text{def}}{=} c\} \cup D'} & = & (\llbracket Expr_1 \rrbracket \ \leq\ \llbracket GN \rrbracket_{g} \ \leq\ \llbracket Expr_2 \rrbracket) \ \wedge\ \bigwedge_{0 < n \leq \llbracket Expr_2 \rrbracket} c \downarrow_{g,n} \\
\llbracket NM \rrbracket_{g,p} & = & g : p : NM \\
\llbracket BM \rrbracket_{g,p} & = & g : p : BM \\
\llbracket LM \rrbracket_{g,p} & = & g : p : LM \\
\llbracket GN \rrbracket_{g} & = & g : GN
\end{array}
$$

Eight definitions of *numeric metrics* are possible, differing only for the type of interval (e.g open and closed, closed and closed, etc.) and for the presence of the `not` operator. We have reported in Table 5 only the two definitions for the closed interval; the other definitions follow the same idea and, hence, has been omitted. Essentially, a numeric metric is translated into a numeric constraint. This translation follows these steps: (i) the expressions are evaluated as numeric values; (ii) these values, together with the specified *unit*, are converted by the $con$ function into values in the standard unit of that metric; (iii) the name of the constraint variable is composed by using the group name, the involved parties and the name.

*Boolean metrics* are the simplest case, which the value of the defined metric has to be equal to the one specified in the SLA.

A *list metric* is translated into a constraint that checks whether all elements of at least one of the lists $\{LE^{+}\}$ are contained in the set of values of the specified metric. For example, in our use case let us suppose that a consumer has three different software products and each needs a different cloud interface to work. This consumer can use the software product that requires the EC2 interface for a task or he can use the other two software together to complete the same task, i.e. he needs both OCCI and UCI interfaces for the second option. Therefore, the consumers can specify the interface as $interface\ has\ \{OCCI,\ UCI\}\ or\ \{EC2\}$, which verifies whether the provider supports EC2, *or* both OCCI *and* UCI.

Finally, the group instantiation corresponds to the conjunction of a constraint concerning the number of instances being run on the system and a collection of constraints regarding the specification of each instances. In particular, the constraint corresponding to each instance $n$ (with $n \in \mathbb{N}$) of the group is obtained from the constraint $c$ of the group definition by applying the function $c \downarrow_{g,n}$; it replaces each constraint variable $v$ occurring in $c$ by $g : v : n$. Intuitively, this function specifies exactly to which instance the metric refers to and, thus, avoids ambiguity on the evaluation of the metrics. Notably, we add a number of instance constraints corresponding to the maximum number of defined instances. For example, if two instances of a group are the upper bound of the defined interval, the constraint of that group is added twice (with the variables properly renamed).

The drawback of using the maximum number of instances for the creation of constraints is that the semantics related to groups might generate a large number of constraints, in particular if many groups refer to other groups. However, considering that SLAs are usually of limited size, the computational capacity and the efficiency of constraint solvers, this does not represent a problem in practice.

We have seen so far how a $SLA$ is translated into a CSP. This CSP can be used directly for checking the consistency of the terms within the $SLA$, at design time. Similarly, at run-time, data representing the status of the system is collected by the monitoring system of the cloud and, then, translated to a CSP. This translation gives a high-degree of flexibility; for example, in case the monitoring measures are not exact or consist of multiple values from a specific period, they can be specified as intervals. The SLA and monitoring CSPs are then combined for the evaluation at enforcement time.

After this evaluation of the SLA, the guarantees specified in the SLA are also evaluated. In particular, when a metric of the SLA is violated (*Event*), a *Condition* might be verified and the corresponding list of *Action*s is sent to the manager responsible for executing them. For instance, in case of a violation of a response time (*Event*), if this is higher than 10 ms (*Condition*) the provider has to notify the consumer (*Action*).

### 3.2.3 SLAs in SLAC: An Example

To illustrate the developed semantics, Table 6 shows an excerpt of a SLA and the corresponding constraints generated using the semantics presented in the previous section. This example is related to the use case presented in Chapter 1 and employs all types of metrics and three groups.

In this SLA, it is specified that $IMT$ must provide a $Cluster$ to the consumer ($Rafael$), support a list of interfaces and that the service must have replication. The service named $Cluster$ is a group of 2 VMs with minimum 1 and maximum 2 CPUs and 1 Gigabyte of RAM. Moreover, the

**Table 6:** Semantics at work on the academic cloud case study.

| **SLA** | **Constraints:** |
|---|---|
| term groups: | #SLA Terms |
| Small_VM: | $1 \leq$ Cluster $\leq 1 \wedge$ |
| IMT → Rafael:cCpu in [1,2] # | imt,rafael:replication == $True \wedge$ |
| IMT → Rafael:RAM in [1,1] gb | $(\{OCCI, UCI\} \subseteq$ imt,rafael:interface $\vee \{EC2\}) \subseteq$ imt,rafael:interface$) \wedge$ |
| Centralized_VM: | |
| IMT → Rafael:cCpu in [2,4] # | #Constraints of the Cluster group |
| IMT → Rafael:RAM in [8,16] gb | $0.0 \leq$ Cluster:imt,rafael:RT_delay:0 $\leq 0.6 \wedge$ |
| Cluster: | $2 \leq$ Cluster:Small_VM:0 $\leq 2 \wedge$ |
| IMT → Rafael:RT_delay in [0.0,0.6] ms | |
| [2,2] of Small_VM | #Constraints of Small_VM, instantiated in #Cluster Group |
| terms: | $1 \leq$ Cluster:Small_VM:imt,rafael:cCpu:0:0 $\leq 2 \wedge$ |
| [1,1] of Cluster | $1 \leq$ Cluster:Small_VM:imt,rafael:RAM:0:0 $\leq 1 \wedge$ |
| IMT → Rafael:interface has {OCCI, UCI} or {EC2} | $1 \leq$ Cluster:Small_VM:imt,rafael:cCpu:1:0 $\leq 2 \wedge$ |
| IMT → Rafael:replication is True | $1 \leq$ Cluster:Small_VM:imt,rafael:RAM:1:0 $\leq 1$ |

response time between this machines must be lower that 0.6 milliseconds.

The generated constraints include the instantiated metrics, the duly renamed metrics in the `Cluster` group and the ones in the `Small_VM` group added twice as the `Cluster` group makes use of maximum 2 of this group. The `Centralized_VM` group is never used but is part of the example to demonstrate that SLAs can contain groups which are not instantiated. This feature is used for several reasons, e.g offers and legal aspects.

## 3.3 Extensions

In this section, we discuss two extensions for the SLAC core-language. The first discuss the support for business aspects while the second adds support to the PaaS model.

### 3.3.1 Business Aspects

The cloud market has grown considerably in the past few years. Yet, in the SLA specification field, few works deal with the business aspects of the provided services. In fact, the main existing SLA languages, partially cover at most partially these aspects [KK07]. This gap represents a significant barrier for the adoption of such languages since they require non-standard and non-trivial extensions to support important characteristics of the SLA, such as pricing models or metrics required for the management of the service.

To analyse and design the support for the business aspects in SLAC, we adopt the scheme of Karanke and Kirn [KK07] that divides the SLA into three phases from the business standpoint: *(i) Information Phase*, in which the details about the services, consumers and providers are browsed and collected; *(ii) Agreement Phase*, in which the participants negotiate and define the terms and the pricing model; and *(iii)* the *Settlement Phase*, which is related to the evaluation and enforcement of the SLA.

In this section, we define the requirements of these three phases and describe the mechanisms used in the SLAC language to fulfil these needs.

To this end, we produce an extension of the SLAC language devised for the business aspects of the SLA.

The information phase requires formalization of the desired or offered services. Also, a repository or a protocol to search for compatible offers or requests is necessary. The core language of SLAC natively supports the definition of offers and requests of services through the specification of parties by means of their roles, the use of groups without instantiation (which enable the formalization of different options for the services available) and intervals for numeric metrics (for example, a VM with memory between 4 and 8 GB). Such features enable the definition of *templates* of services that, when accepted, fulfil the missing data generating a SLA. However, the repository for offers and requests is part of the definition and implementation of the negotiation among parties, which is out of the scope of this thesis.

The agreement phase encompasses the negotiation and the support for different pricing models in the agreement. As previously stated, the negotiation is not part of the scope of this thesis. However, the pricing models impact on the definition of the service and, thus, should be addressed by the SLA definition languages. These models are classified as flat or variable pricing. *Flat* pricing indicates that the price of a service is the same for the whole duration of the agreement. Considering that dynamic markets - as cloud computing - vary considerably, the price might change in the period the SLA is active. Therefore, a *variable* pricing model is often used to allow fluctuations and changes in the price during the agreement. This variation occurs in fixed intervals and is commonly used in the pay-per-use model.

To illustrate the differences between the flat and variable models we exemplify with a commercial solution that employs both models: the Amazon Cloud Service[Ama15]. It provides on-demand VM instances, in which customers pay a pre-defined price for each hour of use. In this case the price is the same for the duration of the service, i.e. flat pricing model. However, Amazon also provides a service named Spot Instances that employs an auction scheme using the variable price model to offer multiple resources. The price of the services fluctuates according to the customers offers[1]. Hence, a consumer makes an offer and, when his bid exceeds the current price of the requested service, the service is provided to this consumer. Then, if the current service price becomes higher than the consumer's bid, the service is interrupted and resumed when it becomes lower again. In this way, the consumer always pays the bid price or less for the specified service, but he might not have the service continuously available.

These two models, flat and variable, can use different pricing schemes. Table 7 lists these schemes (based on [BAGS02]). In this table, they are marked in the respective columns if the model is available in that scheme and the requirements to support the variable model are described in the last column. Below, we provide a brief account on these schemes:

- *Fixed Price* - In this schema, the service price is not subject to bargaining. It uses the variable mode when the validity of the price is shorter than the expiration time, otherwise it uses the flat model. If the model is variable, the new price after the expiration date can be set by a party involved in the agreement or an external party, e.g. a regulatory agency;

- *Bilateral Agreement* - The only difference between this schema and fixed price is that the price is subject to bargaining, i.e. the involved parties must agree on the price. The same assumption is valid if the contract uses the variable pricing. However, if the parties do not agree on the price after the price expiration the agreement is terminated;

- *Exchange* - The parties involved in the SLA exchange resources without involving monetary terms. The SLA must define the resources or services offered by each party. It also supports the variable model and, after the "price" expiration, the participants must agree on the resources to be exchanged or the agreement is terminated;

[1]Although Amazon claims that the price is market driven (demand vs idle capacity), the algorithm is not public. Agmong et al [ABYST13] showed that this is not completely true, since (dynamic) reserve prices and other factors are used to control the prices.

- *Auction* - In this model, an offer or a request of service is proposed and the interested parties bid for it. Several types of auction exist. For example, in the English auction, the bids are visible to anyone, consumers can make multiple bids and the highest offer wins the auction. This scheme also supports the variable model and on the expiration of the price, a new auction occurs. If the involved parties do not win in this auction, the contract is deactivated till the next auction (defined by the price expiration date);
- *Posted Price* - The consumer searches in a directory for services compatible with his needs and chooses the best offer. It is not variable during the contract unless, after the choice of the provider, other schema is agreed;
- *Tender* - A request for a service is defined and interested providers can offer their services at a price. Typically, the combination of most suitable service (respecting the minimal defined in the request) and lowest price wins the tender. This schema does not support the variable model.

The SLAC language needs extension of some terms to support these schemes. To enable the specification of SLA using the flat model, the language should enable the specification of standard service offers and requests. The support of the variable model requires the specification of a dynamic function in the SLA to retrieve information (the current price) and, in some cases, to specify a price expiration date (or frequency), which enables the modification of the price after its expiration without terminating the SLA.

Finally, the settlement phase regards the enforcement, accounting and billing of the SLA. It encompasses the definition of the billing period (that, in turn, depends on the scheme), new metrics and business related actions. The metrics for the business aspects also include Key Performance Indicators (KPIs), such as the response time of the provider's support service (`support_RT` or support response time) and the Mean Time To Repair failures on the system (`MTTR`). Business related actions enable the parties, for example, to `reserve` resources for future instantiation, to

**Table 7:** Pricing schemes and their support in the flat and variable models.

| Pricing scheme | Flat | Var. | Flat Support | Variable Support |
|---|---|---|---|---|
| *Fixed pricing* | ✓ | ✓ | Offer, Request | Price Expiration, Current Price |
| *Bilateral Agreement* | ✓ | ✓ | Offer, Request | Price Expiration |
| *Exchange* | ✓ | ✓ | Offer, Request | Price Expiration |
| *Auction* | ✓ | ✓ | Offer, Request | Price Expiration, Current Price |
| *Posted Price* | ✓ | | Offer, Request | |
| *Tender* | ✓ | | Offer, Request | |

offer financial `credits` for service use, to provide additional services without costs to a party (`bonus`), and to define payments (`pay`) to the involved partners. For the complete list of them, we refer the reader to [UTN14].

Table 8 summarizes the modifications on the syntax of the core language to support the business aspects discussed above. In this table, the symbol ::= stands for a new definition, while + = adds the elements to the core language's definition that shares the same name.

These aspects are extensions of the core language, hence, all additions to the $SLA$ are optional and can be used in both at top level on the $SLA$ specification (in this case, they are valid for the whole agreement) and within the definition of a $Group$. The possibility of specifying the pricing model and the billing for groups gives the flexibility for defining independent business aspects for multi-party agreements.

The main novel feature with respect to the syntax in Table 3 is the possibility of using a function. Indeed, the SLA needs to retrieve external information to support the pricing schemes, in particular the current price of the service. This is achieved through the function `from`, which takes as parameter the $Address$ (e.g. an URL) from which the information is retrieved.

The pricing model and schemes are particularly useful for searching compatible services and for negotiation. Concerning the evaluation of SLAs with pricing models defined, when the flat model is used, no specific

Table 8: Syntax of the business aspects for the SLAC language.

| | | |
|---|---|---|
| *SLA* | += | *PricingBilling* |
| *Groups* | += | *Party* `->` *Party*`:pricing and billing:`*PricingBilling* |
| *PricingBilling* | += | `pricing model:` *Model* `pricing scheme:` *Scheme* *Pricing* `billing:` $Billing^?$ |
| *Model* | ::= | `flat` \| `variable` |
| *Scheme* | ::= | `fixed_pricing` \| `exchange` \| ... |
| *Pricing* | ::= | *PricingExpi* `current price: from(`*Address*`)`$^?$ |
| *PricingExpi* | ::= | `pricing expiration date:` $Date^?$ |
| | \| | `pricing expiration frequency:` $Frequency^?$ |
| *Billing* | ::= | `pre-paid till:` *ExpirationDate* \| `yearly` \| `...` |
| *Actions* | += | `pay` *Interval Unit* \| `reserve:` *Expr Unit* `of` $(Term^+)$ |
| | \| | `credit` *Interval Unit* \| `bonus:` *Expr Unit* `of` $(Term^+)$ |
| *NumericMetric* | += | `offer` \| `support_RT` \| `MTTR` \| `...` |
| *ListMetric* | += | `currency` \| `support_type` \| `...` |

check at run-time is required. Instead, the variable model requires the retrieval of the current price and time. Then, the SLA is evaluated according to the pricing scheme. The steps of this evaluation are pre-defined for each scheme and are performed separately from the CSP evaluation. To illustrate this evaluation, in Figure 16 we show the pseudo-code employed in the evaluation of SLAs which use the auction pricing model. Intuitively, the provision of the service is interrupted when, after the expiration of the price, the current price of the service (offer of other clients or a minimum price for the auction) exceeds the consumer's offer.

Though on-demand self-service is one of the key characteristics of the cloud paradigm [MG09], it does not imply immediate payments. Thus, different billing models are available in SLAC, which can be pre-paid or post-paid. The agreement can be specified as post-paid by defining the billing frequency (e.g., `monthly`) and as pre-paid by not specifying any price nor billing model in the SLA or specifying an *ExpirationDate*, for instance `pre-paid till: 10/05/2015`.

```
auction_variable_model(expiration_dt,offer,address):
        if expiration_dt <= current_date:
                expiration_dt = calculate_exp_dt()
                current_price = from(address)
                if offer < current_price:
                        return interrupt_service
                else:
                        return continue_service
```

**Figure 16:** Evaluation of a variable auction pricing scheme.

### 3.3.2 Variable Auction Example in SLAC

To illustrate the use of the business aspects of the language, we define an example of a SLA for the Amazon Spot Instances service, a commercial solution previously described.

Table 9 shows a SLAC specification of this scenario which uses the variable pricing model and the auction pricing scheme. In particular, the consumer bids for a large VM offered by Amazon. The metric `offer` indicates the bid of the consumers (i.e. the price that the consumer is willing to pay for that service) and the function `from` is used to retrieve the final price. The price expiration frequency is set according to the frequency the provider updates the current price. If the consumer's bid wins the auction and the service is provided (or continue to be provided) to him, the `offer` metric is transformed into the `price` metric. Otherwise, the service is interrupted till the next expiration date, when a new auction takes place.

### 3.3.3 PaaS Extension

The core of the SLAC language focuses on the IaaS level. In addition to this model, we extend the language to support also PaaS, which provides two major types of services to the consumers: (i) a development and deployment platform, mainly focused on web services; or (ii) a platform for the execution of applications, commonly used for scientific purposes (e.g. weather forecast, calculations).

The modifications required to support such scenarios in the SLAC language are the specification of metrics, such as load balancing, time

**Table 9:** Example of the use of SLAC to describe a service using the variable model and the auction scheme.

```
SLA
...
term groups:
 XLarge_VM:
  Amazon→ consumer:cCpu in [8,8] #
  Amazon→ consumer:RAM in [30,30] gb
  broker→ Amazon:offer in [0.56, 0.56] per hour
  Amazon→ broker: pricing and billing:
   pricing model: variable
   pricing scheme: auction
   pricing expiration frequency: 1 min
   current price: from(http://spoinstance)
   billing: hourly
terms:
 [3,3] of XLarge_VM
```

to deploy, types of database, programming languages, and support to metrics which enable the specification of an open value (textual values).

Table 10 shows the additions on the syntax of the SLAC language to support this model. Apart from the definition of new metrics, the main novelty of this extension is the inclusion of the $DescriptionTerm$ in the existing types of terms. These terms are composed of an $OpenMetric$ and a $value$ (which, in most cases, are implemented as a String). These metrics are also pre-defined in SLAC.

The description terms do not impact on the formal semantics of the language since they are not considered during the evaluation of the SLA. However, they are necessary for the description of the service and for its deployment. For instance, in the execution of scientific applications, the users commonly need to send arguments or parameters to the application, or a script must be executed before the application itself.

Notice that the type of service provided (IaaS, Paas or SaaS) is not explicitly defined in the SLA since some metrics are specific for a single software model. Therefore, the model is inferred by the SLA manager according to the metrics.

**Table 10:** Syntax of the extensions to support PaaS in SLAC.

| | | |
|---|---|---|
| $Term$ | += | $DescriptionTerm^?$ |
| $ListMetrics$ | += | `database_type` \| `programming_language` \| ... |
| $DescriptionTerm$ | = | $OpenMetric$ `Value` |
| $OpenMetric$ | = | `command` \| `pre_script` \| `app_name` |

**Table 11:** Example of a PaaS SLA.

```
SLA
...
term groups:
 AppGear:
  IMT→ consumer:RAM in [512,512] mb
  IMT→ consumer:storage in [1,1] gb
  IMT→ consumer:programming_language has {python}
 DBGear:
  IMT→ consumer:RAM in [512,512] mb
  IMT→ consumer:storage in [1,1] gb
  IMT→ consumer:database_type has {redis}
terms:
 [2,2] of AppGear
 [1,1] of DBGear
```

Table 11 shows an example of a simple SLA using the PaaS model. In this example, the parties agreed on the provision of two types of platform. One with support to the developed of applications in Python and another that includes a database, in this case the Redis [Red14] database.

## 3.4 Software Tool

We designed a framework to support the definition and deployment of services using the SLAC language. This management framework is divided into two main components: the Service Scheduler and the SLA Evaluator. The former is context specific and was designed to exemplify and test the integration of the evaluator in real-world scenarios. The latter is context independent and can be integrated in the solutions for any type of cloud problem.

### 3.4.1 Service Scheduler

The service scheduler relies on external systems to deploy and monitor services. Intuitively, it just coordinates the request, deployment and evaluation of the SLA.

It works as follows. The scheduler receives and processes requests from the customers after the negotiation phase. In the first step, the scheduler sends the SLA to the parser, which converts the SLA into constraints and sends them to a consistency checker to verify its consistency.

After the consistency check, the scheduler requests to the cloud platform to start the new services. If the request is accepted, the scheduler is responsible to set up the monitoring system to collect the information concerning the metrics related to the SLA in the system. From this point onward, it repeats the following phases till the end of the agreement: it receives the monitoring data, sends it to the SLA evaluator and reports the results to the interested parties.

### 3.4.2 SLA Evaluator

The SLA evaluator parses the SLA and the monitoring information, and evaluates the SLA according to this information.

Figure 17 illustrates the SLAC evaluator. The SLA evaluator receives the SLA written in SLAC language, parses it and generates a set of constraints corresponding to the specification along with the service definition. Then, the consistency of these constraints is checked and the results are sent to the interested parties. Next, the constraints are stored (organized by the ID of the SLA) and the evaluator yields the service definition to the sender of the agreement.

Then, when new monitoring information is received, it is transformed into a set of constraints and, together with the constraints generated from the SLA, are passed to a constraint solver that verifies their satisfiability.

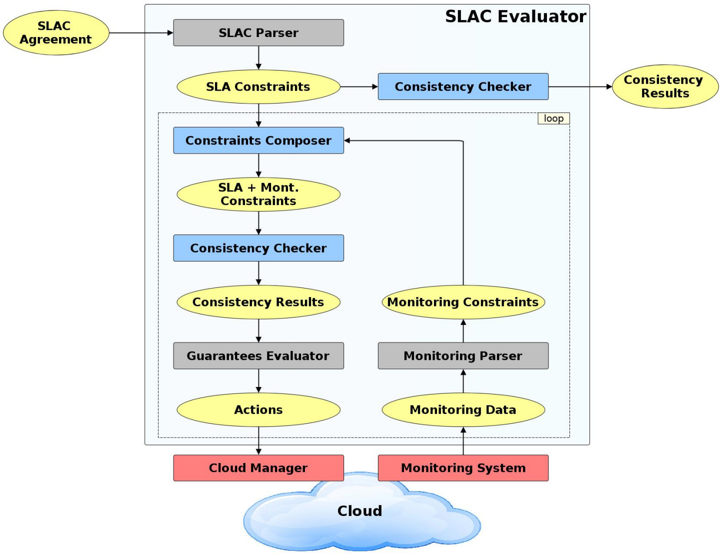

**Figure 17:** SLA Evaluator Framework.

In case of non-satisfiability, which means violation of a metric, the SLA guarantees are evaluated and the due actions will be returned to the sender to be enacted.

Notably, not all monitoring data may be required for the evaluation of the constraints. This feature enables the evaluation of the SLA even with partial observation of the system. For instance, in case one of the monitoring components fails, even if the data of a metric is not collected, the satisfiability of the SLA can be tested with the available information.

### 3.4.3 Implementation of the SLAC Framework

The SLAC framework[2] was implemented using the Python programming language and it focuses on the core of the SLAC language.

The *scheduler* is to the monitoring system, which collects the necessary

[2]The SLAC Management Framework is a free, open-source software; it can be downloaded from:
`http://code.google.com/p/slac-language/`

information to evaluate the SLA. For the deployment of services, the scheduler was integrated with the OpenNebula cloud platform.

The *SLA Evaluator* parses the SLA with the Simpleparse library [Fle15], by relying on the EBNF grammar reported in Table 3. The constraints, in turn, are handled by the Evaluator using the Z3 solver [MB08].

We also developed an editor to define SLAs in the SLAC language based on a plug-in for the Eclipse platform. This editor provides features, such as syntax highlighting and completion to assist the SLA definition. Since several agreements in the cloud domain are still manually defined by the parties, this tool promotes the adoption of the SLAC language, mainly in scenarios where non-experts are involved.

### 3.4.4 Testing the SLAC Framework

In this section, we present the results of a series of tests on the academic cloud case study described in Chapter 1. These tests aim at showing expressiveness of the SLAC language and at illustrating the practical benefits of using the management framework.

Since negotiation protocols are not part of this work, in the proposed scenario, the researchers (consumers) select the offered IaaS cloud services according to a list of pre-defined SLAs. These SLAs are specified in SLAC and enforced with its framework. The cloud system, resulting from the integration of our framework with OpenNebula, when prompted by a request, automatically deploys the virtual machines, configures the monitoring system and periodically evaluates the SLAs.

To compare the impact of different SLAs on the system, we collected information of five hours of system use by the researchers. This information was then used to create a *dataset* which was employed in all tests. The creation of a dataset is crucial for guaranteeing an appropriate comparison as it enables us to repeat the tests with different SLAs and the same input information.

In the tests, we use a definition of a SLA and the generated dataset in order to simulate the behaviour of the service, i.e. we send the monitoring information collected to create the dataset as monitoring data and describe

**Table 12:** Guarantee specification used in the test.

**guarantees:**
```
  on violation of any:
   if cluster < 1
    renegotiate
   else
    bonus:  1 hour of ([1,1] of Cluster)
```

the behaviour of the SLAC framework.

The first test concerns with the SLA presented in Table 6, which instantiates a service without guarantees. Due to their absence, the users as well as the system administrator receive no information concerning the service provision, i.e. they are not even notified if a SLA is violated.

In the second test, we extend the SLA to include guarantees for the quality of service and business aspects. In particular, as shown in the excerpt of the SLA reported in Table 12, if the SLA is violated, the user is guaranteed to receive a compensation. Indeed, in this scenario, in which the service is not paid (as the provider is an academic institution), the provider offers extra credits to the consumer for future use of the service (i.e. one hour of bonus in case of cluster service).

In the second test, using the same dataset employed for the first test, the SLA was violated six times. To depict the behaviour of the services, Figure 18 shows the `RT_delay` (with 0.6 ms as upper bound) of a cluster group running during the test. Analysing the notifications and the enforcement information of the framework, we found out that the main reasons for the violations in the SLAs were the slow network and the lack of proactive management by the cloud system. The insights from the SLA enhanced with guarantees allow both users and the cloud administrator to be notified of the violations and, thus, to take the appropriate actions in order to deal with the issue. Actually, guarantees pave the way for the development of self-managing cloud systems, as well as for the definition of dynamic SLAs (i.e. contracts that enforce different conditions according to the evolution of the system).

**Table 13:** Example based on the extended use case, in which part of the service is outsourced.

```
SLA
...
term groups:
...
 XLarge_VM_SS:
  Amazon→ consumer:cCpu in [4,4] #
  Amazon→ consumer:RAM in [16,16] #
  broker→ Amazon:price in [0.32, 0.32] hour
  Amazon→ broker: pricing and billing:
   pricing model: flat
   pricing scheme: fixed_price
   billing: hourly
terms:
 [2,2] of XLarge_VM_SS
 Amazon → consumer:interface has EC2
```

Finally, we describe a possible extension of the case study that aims at demonstrating the expressiveness of SLAC. In fact, taking advantage of the compatibility of the OpenNebula tool with the Amazon EC2, we can integrate the IMT cloud with the Amazon EC2 service. Using SLAC, this scenario can be expressed in different ways depending on the business processes of the hosting institution. Table 13 shows possible definition of a SLA for this scenario. Notably, in this SLA, IMT is at the same time a provider and a broker, i.e. it offers the local resources and is also the responsible for the payment of the resources used in the public cloud.

### An Overview on the Frameworks Performance

Clouds are elastic and commonly serve a large number of consumers. Therefore, we analysed the scalability and performance of the SLA management framework implementation. To this end, we defined a SLA in SLAC with 20 metrics (including groups), a SLA with 100 metrics, a test set of monitoring information and evaluated the SLAs considering this information. The experiments were carried out using a 2.8 GHz i7 processor.

The framework evaluates in average 35 SLAs with 20 metrics per second, while with 100 metrics this number is reduced to 12. These results suggest that the tool can cope with clouds with a considerable number of consumers. However, it may not cope with the requirements of larger clouds.

Nevertheless, it should be stressed that our implementation is a proof-of-concepts developed to test its feasibility and the applicability of the language in the domain. Therefore, more efficient and distributed implementations can be envisaged to cope with the requirements of larger clouds.

### 3.4.5 Discussion

Now that the technicalities of our approach have been introduced, we can discuss its main features, our design choices and illustrate possible uses and extensions of the functionalities provided by the developed framework.

An important decision while defining a SLA specification language is the appropriate level of granularity and the trade-off between expressiveness and specificity [DWC10]. Most of the related works aim at creating generic models and specifications, which require the creation of extensions of the language, i.e. the domain specific vocabulary. However, these extensions are costly and require expertise. In clouds, a wide range of services is available and the domain has an uncommon set of characteristics, which are difficult to grasp in a high-level SLA specification language (e.g. dynamism, multi-party agreements). In view of these particularities, we opt to define SLAC as a domain specific language for cloud computing, thus capturing the most important aspects of the domain and, at the same time, defining the low level of detail (including, e.g. metrics of the cloud domain). As a result, we provide a ready-to-use language for the definition of SLA for clouds.

However, if the required metrics are not available, providers and consumers can extend the metrics of the language according to their needs, i.e. the language and the architecture of the framework permit both the addition and redefinition of metrics. Extending the language might create incompatibility between SLAs using non-standard metrics and SLAs written in the standard language. To alleviate the impact of such extensions, the user can configure a translation function of the new metric using one or more built-in metrics.

The main features of the language are the support for brokerage and multi-party. The broker takes an important part in the SLA since it has a key role in the domain. We enable the use of the broker in the SLA with the definition of groups, specification of the involved parties on each term and action, the inclusion of all roles of the domain in the SLA and the possibility to define multiple roles for the same party. The support to multiple party, in turn, enables the users of the language to define SLA of all deployment models and to express real-world scenarios in the SLA. To enable this feature, we define groups, which set the scope for metrics; definition of the parties in each metric to explicitly express the involved parties; support of all roles involved in the SLA in the cloud; and the support for multiple roles for the same party. Notably, all terms and business aspects can be contextualised in groups, which entails a high-degree of flexibility and expressibility.

The core of SLAC focuses on IaaS but we have also specified an extension to support PaaS. However, since most requirements of SaaS are also taken into account, we do not envisage any major issue in extending SLAC to cover it. The main extensions needed to support this model regards the generality of the types of metrics available in the SaaS model. We plan to specifically address the support of SaaS in future works, considering the concrete and ready-to-use nature of SLAC, i.e. avoiding high-level abstractions which may become a source of ambiguity, and which require lengthy and complex extensions.

Although negotiation is out of the scope of this thesis, the SLAC language and its business extension already took into account the needs of this process. For instance, the concept of groups allows the providers to specify all services available in their offers without actually instantiating them in the SLA. The definition of roles also leverages the negotiation

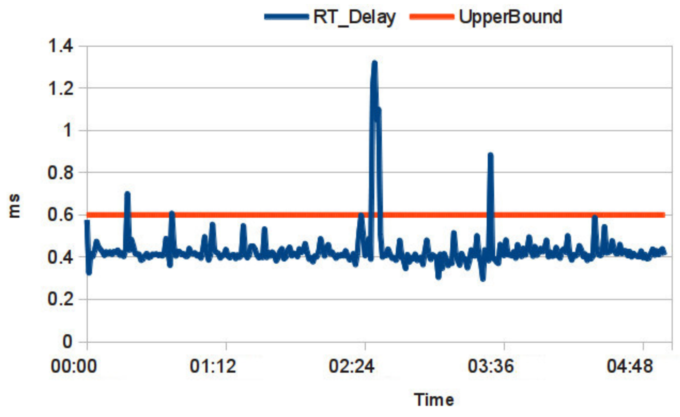

**Figure 18:** Delay of the communication of two VMs of a cluster group.

process, as offers and requests can refer to the other involved parties without specifying them.

Despite the fact that several works regarding the service negotiation include priorities for the metrics, we opted for not including them in the SLAC language. For instance, in [GVB11] the authors define a framework in which the metrics are paired with a weight, which are then ranked according to this weight. Afterwards, this ranking is used to search for compatible services. However, we believe that the importance of each metric is implicitly established in the `guarantees` section which can be used also for negotiation. Intuitively, in an offer or request, as more important is a metric the higher the associated penalty; therefore, the SLAC does not implement such a feature.

Finally, security is a priority in the domain. Nevertheless, currently, the language and the framework have no mechanism to check the veracity of the monitoring information related to the specified metrics (*monitorability* of the SLA). When the framework receives a request, it extracts the specification of the service to be deployed and configures the monitoring system to provide the monitoring data necessary to test the conformance of the SLA. Consequently, the data that is made available by the parties might not correspond to the collected data. Two possible solutions are following: include a party that audits the information (it should be a trusted party); or integrate the participant parties with the framework, thus allowing the framework to collect the data directly on the infrastructure. The latter approach requires access to the infrastructure of the parties, an independent management framework (not associated with the provider), and the development of non-invasive and platform-independent monitoring solutions.

## 3.5 Summary

In this chapter, we covered the first pillar of the autonomic management in the autonomic cloud domain: the definition of services. In autonomic cloud, where the provision of services is a priority, the service definition provides the high-level objectives of each service, which are then used as a guide for their management and for the cloud itself.

In view of this need, in this chapter we proposed SLAC, a language for the definition of SLAs tailored for the cloud domain. In particular, we specified the syntax of the core of the language as well as the semantics of the conformance check of SLAs, which is defined in terms of constraint satisfaction problems. Moreover, we implemented the core language in a framework and deployed it in an academic cloud case study, which was used to illustrate the applicability of the approach.

The other existing solutions do not cover important aspects of the domain, such as brokerage, multi-party agreement, specification of multiple composed services and the vocabulary for clouds. The SLAC language successfully addresses this gap and, additionally, provides support to significant business aspects of the domain, such as pricing models, billing and business actions. Thus, we enable the use of the language in real-world autonomic cloud implementations.

# Chapter 4

# Panoptes: An Architecture for Monitoring Autonomic Clouds

In the previous chapter, we proposed a language to define services in clouds. This definition is one of the main pillars of the self-management of autonomic clouds. However, to enact the self-management, autonomic managers need to know the status of the services being provided in the cloud, of the autonomic cloud itself, and of the environment.

This provision of this knowledge to autonomic managers is another pillar of the autonomic management and should be available in real-time. These characteristics and the specificities of autonomic clouds, such as heterogeneity, scalability and dynamism, represent significant challenges for the collection and processing of data.

In this chapter, we argue that these processes should be carried out by a monitoring system devised for autonomic clouds and, therefore, they should be external to the autonomic managers. In the case of autonomic clouds, this monitoring system must provide the knowledge about the state of the system and the environment, considering different abstractions used in the system (e.g. services, virtual machines, cluster).

In light of these needs, in this chapter, we focus on the definition of the knowledge discovery process. In particular, we define data, information, knowledge and wisdom in the autonomic cloud domain, analyse the autonomic cloud requirements for such tasks and, considering also the monitoring properties presented in Chapter 2, we design a multi-agent architecture to collect and process the collected data. This architecture is devised to be integrated with the autonomic system and to take into account the services defined using the SLAC language. Finally, we validate this framework through its integration with a simple self-protection framework and with experiments that test its scalability and invasiveness.

## 4.1 From Data to Knowledge In the Autonomic Cloud Domain

In Section 2.6, we define the meaning of knowledge, information and data based on the DIKW hierarchy. In this section, we define these concepts considering the specificities of autonomic cloud.

*Data* represents sequences of observations collected in the system. Also known as operational data, it consists of local measurements, such as *CPU usage*, which are of little utility to the decision-making process if neither specifically requested, nor processed to fulfil the needs of managers.

*Information* can be defined as data used in the decision-making process (useful data). The requested data is selected in the information layer (filtering process) or transformed into information through aggregation, filtering and simple inferences. Nevertheless, the real value of information depends on its accessibility, reliability and timeliness.

*Knowledge* is the specific interpretation of (aggregated) information, which feeds the decision-making. This interpretation is discovered through models, inference, machine learning and analyses of historical information. Also, we borrow a concept from epistemology which defines two types of knowledge: (i) procedural knowledge, which is used to model and produce more knowledge; and (ii) propositional knowledge, which is the knowledge produced from these models.

*Wisdom* is closely related to the decision-making process and the definition of the heuristics to discover new knowledge in the autonomic cloud domain. In contrast to knowledge which produces new knowledge based on information, the discovery of new knowledge from wisdom is based exclusively on existing knowledge.

Much research has been conducted to explore the nuances of the levels in the DIKW hierarchy [Sch11]. To clarify these nuances in the domain, we propose two scenarios in which we relate these definitions with elements of the domain: the application of autonomic computing to the management of the infrastructure of a cloud; and the autonomic service management in clouds based on enforcement of SLA.

In the first scenario, the data, which are the measures of the use of the resources (CPU load, memory usage, network bandwidth), is collected from all resources. Information, in this context, has different abstractions depending on the type of resources that the target autonomic manager coordinates. For example, the autonomic manager of a cloud cluster might require the information of the average load of the cluster in the last hour. To discover this information, the data is filtered (selecting the data from resources only of that cluster) and aggregated, forming the information. Finally, the same autonomic manager predicts the future load of the cluster, thus creating knowledge. To create this propositional knowledge, the managers need to transform historical information and the current measures into a prediction through a model (procedural knowledge).

In the second scenario, concerning the SLA, the data is collected from the (physical and virtual) resources, as in the previous case. Then, for the evaluation of simple metrics of the SLA (e.g. CPU load), the data is filtered and sent to evaluation, thus becoming information. In the case of compound metrics (e.g. response time), the data is filtered, aggregated and contextualised to create the information, which is used for the evaluation of these metrics. For example, to test the availability metric, it is necessary to apply a function on the previous availability information and the current state of the system. Finally, let us suppose that the managers need to know how similar two services are. This knowledge can be created from the information about the services, which are then aggregated and used as input to an inference model (created by a machine learning algorithm) to generate the measure of similarity among the services. In this case, the model which defines how to calculate the similarity among services is a procedural knowledge, while the similarity among the services is propositional knowledge.

## 4.2 Role of the Monitoring System and Domain Requirements

The autonomic cloud domain presents significant differences to traditional data centres, such as heterogeneity, elasticity, virtualization, large-scale and dynamism. These characteristics hinder its management; hence, solutions to collect and process data to discover knowledge and assist the autonomic decision-making are required.

The monitoring system is the entity in charge of capturing the state of the system. However, the monitoring task can be seen from multiple standpoints. In the autonomic cloud domain, the most important perspectives are the data hierarchy, autonomic computing and service management, and we describe them below.

From the *data hierarchy* standpoint, the scope of the monitoring system is restricted to the transformation of operational data into knowledge. Therefore, these systems neither generate nor use *wisdom* as the latter is employed to assist the decision-making process, which is not part of monitoring. Indeed, monitoring systems can be autonomic and, in such cases, they also execute the decision-making process task. However, in these cases the decision-making and the wisdom are managed by an independent component of the system which, might require the knowledge discovery to take decisions but is not directly related to the monitoring task.

In *autonomic computing*, the autonomic managers use sensors, which are software or hardware components, to collect data from the environment. In this regard, there exist two main approaches: the autonomic managers collect facts from the environment themselves or they rely on other applications, such as a monitoring system, which feeds the control loop of these managers with knowledge [HM08].

The autonomic systems must focus on the decision-making process, which adapts the system to enact the self-* properties. Therefore, the use of an external monitoring system is preferred to remove the complexity of such process from the autonomic system. However, an external monitoring solution does not replace the monitoring phase of the autonomic control loop which must still, e.g. verify the status of the autonomic agents and evaluate the knowledge received through sensors.

Nevertheless, the employment of this approach requires the integration of the monitoring system with the autonomic managers, which request the necessary knowledge to the monitoring system and receive it through sensors. Therefore, the monitoring component must be capable of providing fine-grained data as well as a general overview of the state of the system.

Finally, we also consider the *service management perspective* as clouds are distributed systems that are employed to provide services to their consumers. From this perspective, the system that is in charge of collecting the status of the service is a part of the monitoring phase of the SLA life cycle, which retrieves the status of each metric for the SLA compliance verification. Moreover, the monitoring system should also report the status of the provided services to the consumers, according to the configurations in the SLA.

In light of these perspectives and multiple functions of the monitoring systems in this domain, such as feeding the autonomic system, reporting to consumers and having multiple abstractions of the domain (services, nodes, clusters), we argue that the monitoring system must be an independent entity which enables the discovery of knowledge in the domain. With this vision in mind, we designed and implemented a monitoring system to cope with the requirements of autonomic clouds.

## 4.3 Related Works

Few other works address specifically the monitoring of clouds, and none considers also the needs of autonomic systems. In [CUW11], a monitoring architecture and a framework, named PCMONS, for private clouds are defined. However, its main drawback is limited scalability due to its centralized approach.

In [ABdDP13], an extensive survey on the cloud monitoring area is presented, including many cloud deployment models. The authors discuss the properties of monitoring systems, present the current commercial, open source platforms and cloud services for cloud monitoring, and confront these solutions with the cloud monitoring properties. Furthermore, they identify open issues, challenges and future directions within the field. Among the listed open issues, they highlight resilience as a fundamental property but none of the existing solutions focus on it. Timeliness was also highlighted and has been only implicitly addressed by works that focus on other properties, such as scalability and adaptability, but is only explicitly considered in the work of Wang et al. [WST+11]. In view of these gaps, Aceto et al. state that the particularities of cloud computing require new monitoring techniques and tools specifically devised for cloud computing.

Many monitoring solutions for distributed systems are available. However, commonly they assume homogeneity of monitored objects and are not scalable due to their centralised architectures [EBMD10]. Below, we discuss some works which do not have these restrictions.

Monalytics [KST+10] is devised for the monitoring of large-scale systems and shares many design choices with the solution proposed in Section 4.4, such as the Publish/Subscribe (Pub/Sub) communication model and data analyses close to the source. In [WST+11], the authors emphasise the analytical perspective of monitoring solutions through timeliness and different granularities. They make use of a hybrid topology, which can dynamically adapt to different paradigms. However, both proposals do not address the monitoring properties discussed in this work, and they focus on generic data centres, while we address the particularities of clouds and autonomic system.

In [CGCT10], the authors define Lattice, an open source monitoring framework, to enable the monitoring of federated clouds. However, all parties involved in the cloud must deploy this framework, which is hardly feasible in commercial clouds. Furthermore, this solution is not scalable since the filtering and processing of data occurs only after the data of whole cloud is collected.

Finally, in [EBMD10], the authors focus on the equivalence between SLA metrics and low-level monitoring metrics. In our work, the conversion of SLAC metrics into low-level monitoring metrics employs an approach based on [EBMD10]. However, to collect data from the system, they employed the Ganglia [Gan14] which focuses on distributed systems. Therefore, they neither consider the particularities of clouds, nor integrate the monitoring and autonomic systems, nor tackle specifically any monitoring system property.

To our knowledge, apart from the previously analysed works, no other monitoring architecture has considered the requirements of autonomic systems and cloud computing. Moreover, only few novel works have applied the concepts of multi-agent systems and autonomic computing in the monitoring area [SILI10, Ant10] and none specifically addresses clouds.

## 4.4 Panoptes Architecture

The existing monitoring solutions cannot cope with the needs of autonomic clouds, such as strong integration of the monitoring system with the autonomic managers and cloud dynamism. Therefore, in this section we present Panoptes[1] [UW14], a multi-agent monitoring architecture tailored for autonomic clouds.

In Chapter 2, we describe the properties of monitoring systems related to clouds. The scope of properties is broad and range from scalability to resilience; therefore, we focus on the most relevant properties of the domain and the properties which had not been implemented before. These properties are: (i) scalability, due to the large-scale of clouds; (ii) timeliness, since the real value of the knowledge for the autonomic managers depends on its timing; (iii) resilience, as the nodes of the cloud are loosely

[1]The framework was named after Argus Panoptes, a giant from Greek mythology with one hundred eyes. In the legend, he was considered an effective sentinel because he never closed all his eyes. Even while sleeping half of his eyes were kept open.

coupled and the monitoring system is essential for the decision-making; (iv) adaptability, since the autonomic clouds should focus on services; and (v) extensibility, to support the heterogeneity of the resources and services of the clouds.

Autonomic clouds commonly have four high-level management *roles*: cloud, cluster, nodes and storage controllers [CUW11]. Storage controllers are similar to node controllers from the monitoring perspective. However, for other components of the cloud, e.g. routers, switches and UPSs, no specific management role exists. Yet, in autonomic clouds, also these components need to be monitored.

Considering the management hierarchy and these particularities, we define the roles of the agents in the monitoring systems as four:

- *Cloud Agents* do not collect data on the resources but receive coarse-grained information/knowledge from other agents. At this level, complex operations, such as data mining algorithms and analytical tools are employed. As cloud have a service-oriented nature and these agents have a broad view of the system, they are the responsible to manage the monitoring of services since they can be executed in multiple locations at the same time.

- Clusters are typically deployed in geographically restricted areas; therefore, *Cluster Agents* generate information and knowledge and, at the same time, filter the collected data to reduce network traffic and to avoid unnecessary processing at the cloud level.

- *Monitoring Agents* filter, aggregate and contextualize fine-grained data. In this architecture, they are employed in storages and cloud nodes.

- *Probe Agents* target devices with low processing power to comprehend also other components apart from servers and storages, which commonly have low processing power. This role is designed to consume as less resources as possible and, consequently, it does not process the collected data.

Figure 19 illustrates the roles of the agents from the *knowledge discovery* standpoint (DIKW hierarchy). Probe agents collect data and send it to the monitoring agents. Monitoring agents collect data or receive from the probe agents and generate information through filtering and aggregation of this data. Cluster agents transform information into knowledge and generate more knowledge from existing knowledge. Finally, cloud agents produce knowledge and do not collect any type of data.

The agents of all roles are deployed in the cloud resources. Monitoring, cluster and cloud agents are typically placed in cloud nodes and probe agents are intended for resources with low processing, such as routers and VMs. These agents are executed in the resources according to the structure of the cloud and the needs of the autonomic managers, and each resource can have multiple agents with different roles. To decide about the placement of the agents, the structure of the cloud resources should be explicitly configured or the monitoring system should be integrated with a cloud provision solution that make this information available.

The main advantage of such a monitoring structure is its scalability since it organizes the agents according to the cloud. This architecture also prevents the dissemination of unnecessary information through the cloud due to different abstraction levels of agents, whereas it still provides flexibility for *in-situ* analysis.

Nevertheless, the monitoring system should prevent the performance deterioration of the target system (the autonomic cloud) caused by the monitoring tasks. To prevent this deterioration, the system should adapt itself. To this end, each request of monitoring data, information or knowledge at any level requires the specification of its priority. As a result, agents can adapt the number of active tasks (e.g. deactivate low-priorities tasks) according to its usage on the resource. Another valid strategy to prevent the invasive behaviour is to change role or to move to another resource. For instance, let us suppose that a cluster agent requires a high CPU load algorithm to generate knowledge and the node where it is being executed is overloaded. In this case, the cluster agent finds a more powerful node, migrates and activates a probe agent only to collect data in its place. These features are configured using simple Event-Condition-

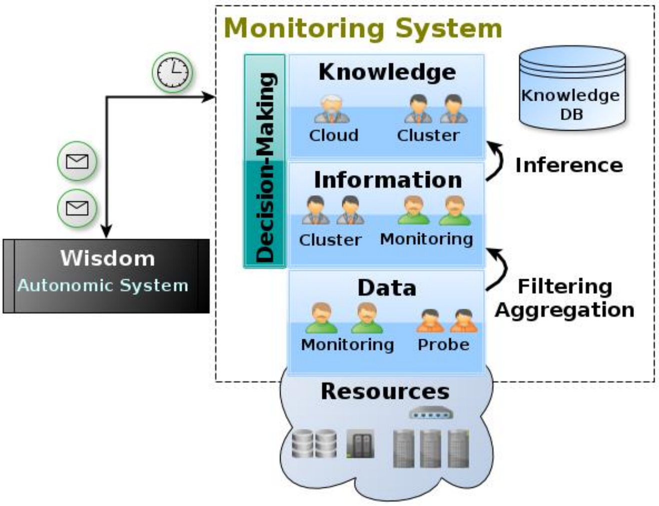

**Figure 19:** Relation between the roles of agents in the monitoring system and the DIKW hierarchy.

Actions rules, e.g. defining the threshold to the CPU usage which would result in the migration of the agent.

The communication among agents is primarily performed by topic (classes of messages). They register in topics they are interested to (e.g. cluster01, probeagents, nodeIP). Then, agents publish their messages to a specific topic and these messages are delivered only to agents subscribed to this topic. This model, named Publish/Subscribe, enables any number of publishers to communicate with any number of subscribers asynchronously via event channels on particular topics, thereby preventing broadcasting. We opt for this model due to its high-scalability, flexibility and performance [FHAJ+01].

The agents must also have an independent communication server; therefore, they can also communicate with each other by sending direct messages. This type of communication among agents serves two purposes: the transmission of critical messages, which are sent directly to the requester skipping the other phases of the knowledge discovery processes to reduce latency; and the improvement of availability, resilience and reliability of the monitoring system, in case of failures in the Publish/Subscribe architecture. To carry out this direct communication, the agents need to be updated about their respective addresses of upper-layer agents, which keep a list of the agents in lower-layers and a list of the topics that these agents are subscribed.

The requests of the interested party and their integration with the monitoring system is performed through the agents' interface that enables the on-line configuration of the monitoring system. This interface is used to create, remove and modify modules of the system (monitoring tasks), which consist of the definition of what should be monitored and how to discover knowledge with the collected data.

The results of the configured modules are sent directly to the interested party by the agent which produced the result. Notably, considering that the monitoring system is independent from the autonomic system, the produced knowledge is sent to the interested party as this knowledge can not be directly added to the knowledge database of the autonomic managers due to security reasons.

Transforming operational data into useful knowledge and accurately capturing the state of the system are challenging in large-scale paradigms due to the amount of operational data generated in the domain [VHKA10]. As a result of the large-scale and of the amount of operational data generated in autonomic clouds, the autonomic agents should rely on techniques to process data on-line. Therefore, the monitoring system does not have a knowledge database. However, the discovery of knowledge might require local storage of data, information or knowledge for operations, such as aggregation and inference.

Considering this need of temporary storage to perform such operations, we propose a simple database model based on associative arrays (also known as dictionary data structures). Intuitively, they are a collection of $(key, value)$ pairs where $keys$ are unique in the database. This model is known as key/value model and we have chosen it since it has no strict structure, it is horizontally scalable (particularly suitable for clouds) and it does not require the set of properties which guarantee that the transactions in databases are processed reliably (known as ACID, Atomicity, Consistency, Isolation, Durability).

Panoptes architecture was designed to be modular and easily modifiable. Therefore, its components can be replaced according to the needs of the systems (for example, the manager can replace the communication server or role of the agents). This modular design motivates the adoption of this architecture for research purposes (e.g. when a single part of the monitoring is the subject of a study) and in the industry.

The main features of the Panoptes architecture are summarised in Table 14. With this architecture, we address the properties as follows:

- *scalability* with multiple levels of agents and on-line data filtering, aggregation and inference close to the source;
- *adaptability* with activation and deactivation of modules according to their priorities and change of roles;
- *resilience* with the multi-agent architecture and direct communication among agents;
- *timeliness* with definition of priorities and the possibility to skip data processing;
- *extensibility* with the architecture itself, the creation of scripts, the on-line addition and removal of modules, and the use of monitoring templates.

Moreover, the integration between Panoptes and the autonomic system is leveraged by the urgency mechanism, the on-line configuration of the monitoring modules, the adaptation capabilities of Panoptes and the centralisation of the collection and processing of multiple data abstractions in a single system.

## 4.5 From SLA Metrics to Monitoring Modules

SLAs provide a high-level account of the description of services, which commonly does not have a direct correspondence to the low-level metrics

**Table 14:** Summary of the main characteristics of the Panoptes Architecture.

| | *Type* | *Description / Example* | *Characteristic* |
|---|---|---|---|
| ***Roles*** | Cloud Agent | Produces only knowledge | High-level view of the system |
| | Cluster Agent | Produces information and knowledge | Keeps the data and information local |
| | Monitoring Agent | Collects and processes data | Low-level view of the system |
| | Probe Agent | Only collects data | Low resource consumption |
| ***Know. Disco.*** | Filtering | Selects only the data required to discover knowledge | Reduces the amount of data to be processed |
| | Aggregation | Groups data or information | High-Level view of the data |
| | Inference | Produces knowledge through, e.g. machine learning and data mining | Requires resources and complex models |
| ***Adaptation*** | Priorities | Deactivates or activates modules | Adapts the load of the monitoring system |
| | Change of Role | Agents change their roles according to the load of the monitoring tasks | Enables the self-organization of the agents |
| ***Communication*** | Publish/Subscribe | Communication based on topics | Efficient multicast communication |
| | Direct Messages | Based on topics but requires the addresses of the agents | Point-to-point communication |
| | Knowledge Delivery | Agents can send the processed knowledge directly to the requesters | Distributed approach |
| | Critical Messages | Skips the data processing and sends the message directly to the interested party | Reduces latency |

**Table 15:** Example of the equivalence between a SLA term and low-level metrics.

| SLA Metric | Mapping | Required Monitoring Data |
|---|---|---|
| Availability | $\frac{Uptime}{(Uptime+Downtime)}$ | The invocation of the service through the Internet to test whether the services are available, possibly from multiple locations. Then, it requires the aggregation of the results to calculate the up and down times. |
| Boot Time | $availabilityDate - startingDate$ | The exact date and time of the request (or approval) of the service and the date and time this service is available. |

collected by the monitoring system. Therefore, there is the need to find this correspondence in order to accurately measure whether a SLA has been enforced. Table 15 illustrates this concept with two SLA metrics and the low-level metrics necessary to calculate them.

In the case of the SLAC language, we propose an approach based on [EBMD10], which provides mapping rules for each metric. The advantage of direct mapping between SLA metrics and monitoring modules (as they are called in the Panoptes framework) is the simplicity of the approach which facilitates the automation of the process. Moreover, the SLAC language already provides a description of the metrics for the domain which are used to form the pre-defined rules.

The mapping rules are stored in a database and organised by context. Therefore, a mapping request requires also the information about context of the SLA or metric, i.e. the name of the metric, the type of service and the platform in use. In particular, in each mapping request must be specified: the cloud platform (e.g. OpenNebula, Eucalyptus and generic), the location and IP of the resource in which the service is deployed (e.g VM X, cluster Y), the service model (e.g. IaaS and PaaS) and the operating system (e.g. Linux and Windows). Consequently, a module that can

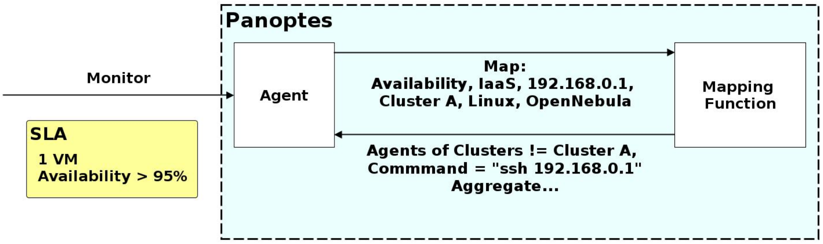
Panoptes
Monitor
Agent
Map:
Availability, IaaS, 192.168.0.1,
Cluster A, Linux, OpenNebula
Mapping Function
SLA
1 VM
Availability > 95%
Agents of Clusters != Cluster A,
Commmand = "ssh 192.168.0.1"
Aggregate...

**Figure 20:** Mapping of a SLA term to a (simplified) module of Panoptes architecture.

be directly employed by the monitoring agents is returned. Figure 20 illustrates the mapping process with a request and the result for that specific platform. In this example, an agent requests the mapping of the *availability* metric and sends the parameters necessary for the mapping of this metrics. In this specific case, the resulting module defines that this monitoring component runs in agents which are not in cluster A (*Agents of CLusters != Cluster A*) and defines the command as a *SSH* request to test whether the VM is accessible.

## 4.6 Implementation

In order to demonstrate the feasibility of the architecture, we developed a monitoring framework in Python. The reasons for opting for this programming language are its simplicity and readability, which motivate the development of extensions and scripts for the framework.

The agents of the framework are generic entities that assume roles according to the needs of the system and can change during their life cycle. They collect data from log files, using the SNMP protocol or executing specific commands that are configured by the manager. The adaptability is implemented considering the physical resources employed to run the agents (RAM and CPU usage). The agents monitor their resource usage and compare it with the defined policies. In case of violation of the policies, the agents can change role or reduce the number of modules (deactivate the low-priority ones).

The database chosen to support the data processing function and the SLA metric mapping is Redis (REmote DIctionary Server [Red14]), which is an open source project for Key/Value storage. Moreover, we also use the implementation of the Publish/Subscribe that is included in the Redis database for the communication among agents. Finally, for the direct communication, the agents have embedded their own communication server, which is developed using TCP network sockets.

The specification of modules defines the granularity, metrics, host addresses and regular expression for matching patterns of critical messages. To support the development of monitoring scripts, the framework provides a basic API for the Python with functions such as *aggregate*, *send* and *urgent message*. This support to scripts in Python enables the managers to personalise the knowledge discovery process.

Finally, we integrate the framework with the OpenNebula [Ope14] solution through the automatic injection of probe agents in its VMs.

## 4.7 Experimental Evaluation

In order to evaluate the Panoptes monitoring implementation, we present a use case which proposes the integration of Panoptes with a self-protection framework. Moreover, we evaluate the monitoring properties, such as scalability and timeliness through analytical experiments.

### 4.7.1 Use Case

We developed a simple multi-agent framework that is devised for the self-protection of clouds in order to validate the concepts and to offer an insight into the integration between an autonomic system and Panoptes.

#### Design and Implementation of the Self-Protection Framework

The framework is a multi-agent system that allows the definition of modules for self-protection. It provides the essential infrastructure to execute the MAPE-K control loop.

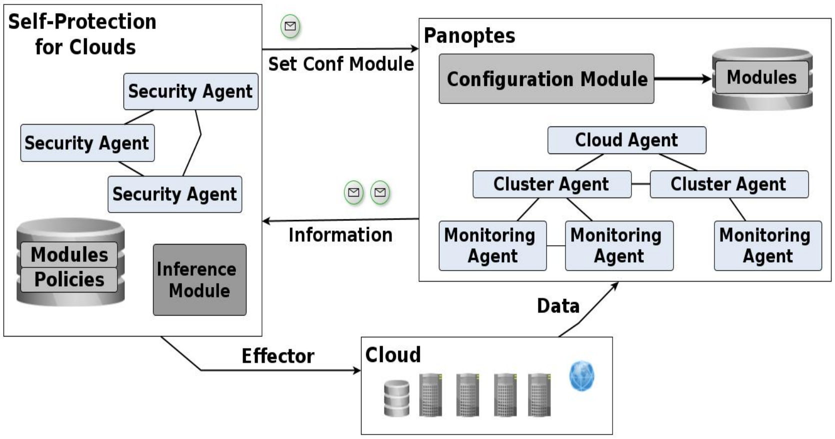

**Figure 21:** Integration between Panoptes and the self-protection system.

The self-protection functions are defined in the form of modules. These modules guide the MAPE-K loop for the decision-making. Follows the description of this loop. The *Monitoring phase* receives information from Panoptes, and collects information on the updates the system policies and the other agents of the framework. The *analyses phase* relies on a utility function defined for each module. This function calculates the security risk of each module. In case of high-risk (defined by high-level policies) an action needs to be taken. The *Planning phase* chooses which action will be taken to avoid the risk. In particular, the available (pre-defined) plans are evaluated using an Event-Condition-Action rules database. Finally, the chosen plan is executed in *Executing phase* by the agents of the system.

Figure 21 depicts the interaction among the cloud, Panoptes and the self-protection system. The self-protection system adds new modules or configures the modules of Panoptes in an on-line fashion, and executes plans on the cloud carrying out corrective and preventive actions.

### Use Cases: The benefits of integrating Panoptes and the Self-Protection Framework

In order to understand the implications of the integration of the monitoring system and an autonomic system in real-world clouds, we propose the integration between Panoptes and the self-protection framework in the academic cloud presented in Chapter 1. In this cloud, the students can employ VMs for research purposes and access them through the Secure SHell (SSH) interface. To provide these VMs, we employed the OpenNebula tool in six physical machines divided into two clusters.

**Listing 4.1:** Configuration of Panoptes for the use case.

```
Host = VMs
MonitoringType = Log
        Reference = /var/log/sshd
Frequency = newEvent
Filtering = Monitoring
        Pattern = failed password
        Filter = (dateTime)(IP)
Aggregation = Cluster
        Maximum Aggregation = 5
Priority = medium
Critical = (Ip) != (10.0.0.*)
Destination = localhost:100
```

In this scenario, we developed a module to protect such interfaces from password guessing attacks. The protection framework was configured with a utility function that analyses the failed login attempts considering historical information and, in case there is a high probability of an attack, it blocks the connections from the suspect IPs.

Listing 4.1 presents a summary of the configuration of the module defined for the use case. The *host* keyword defines the monitored resources, which are, in this case, all the VMs running in the cloud. The *log* verifies the file of the SSH server and checks it on every modification (*frequency*). Filtering and aggregation can be performed in different levels (monitoring, cluster, cloud); therefore, it is defined in their configurations.

In the filtering, only events that match the *pattern* are filtered and only the filtered part of the events is used for the processing of the collected data. In this case, only the IP, date and time of the occurrence are processed (which is specified using regular expressions but was omitted here for the sake of simplicity). The *aggregation* enables the accumulation of a number of events before sending it to upper-level agents or the final destination. The priority is used by the adaptation component of the Panoptes. The keyword *critical* denotes the pattern of messages that are urgent, in this case, the IP addresses that are not from the local network. Finally, the interface address of the interested party is set by the *destination* keyword. Most of the configurations support more than a single option, for example, multiple critical patterns can be added according to the needs of the system (as occurs in our use case).

The monitoring system was manually configured to provide the results to the self-protection framework. It aggregates the information of the login attempts from the cloud, cluster, monitoring and VM perspectives, and sends this information to the framework. However, the module developed for this framework updates the configuration of Panoptes on-the-fly. In particular, when the risk of invasion is higher than 50%, the framework configures Panoptes to mark the modules as high-priority. Thus, for all new events related to the suspect IPs, it skips the data processing to reduce the latency of the process.

For the tests, we developed a script that randomly selects an origin IP from a list and attacks a VM. It was executed in 30 minutes periods in a cloud with 50 monitoring agents divided into two clusters, each with a single cluster agent and only a cloud agent in the system.

Table 16 presents the significant reduction in the number of message processed by the self-protection system from the monitoring system in comparison to the number of messages collected directly by the self-protection agents in the same case. Panoptes reduces 87% of the number of messages processed by the autonomic manager and 84% of the volume of the messages. In the same test, we also counted the number of messages exchanged by the monitoring system which amounted to 148130. This indicates that the total number of messages, i.e. the ones exchanged in

**Table 16:** Messages exchanged in the described scenario.

| | Number of Messages | Volume of Messages |
|---|---|---|
| **Panoptes** | *160450* | *19.0 MB* |
| **Autonomic Agents** | *1160229* | *117.6 MB* |

the monitoring system related to the module plus the ones sent to the autonomic agents, was reduced by 73%.

In addition to this significant improvement, all the complexity of collecting, filtering, aggregating and inferring the data is handled by the monitoring system, which is devised for the domain. Therefore, the autonomic system can focus on the self-management process, which includes the analyses, decision-making, and pro-active and corrective actions.

### 4.7.2 Experiments of the Scalability and Invasiveness of Panoptes

In this section, we propose analytical tests focused on the scalability and invasiveness of the solution. Therefore, the experiments evaluate the resource usage of the solution and the number of messages processed per second.

More specifically, the first experiment evaluates the CPU usage of a probe agent and of a monitoring agent both executed on an Intel Core2 Duo T7500 2.00GHz, while the second experiment compares the timeliness of the normal messages with the critical ones.

The chart in the left side of Figure 22 demonstrates the average CPU usage of a monitoring agent (in blue or dark grey colour when on black and white) and the usage of a probe agent (in orange or dark grey colour when on black and white). Both agents use less than 5% of CPU till 200 events per second, and less than 3% with a normal load (less than 100 events/s). These results suggest that our solution scales well and that the probe agents consume less resources since in the test they used less than 2% of CPU even with a high number of new events.

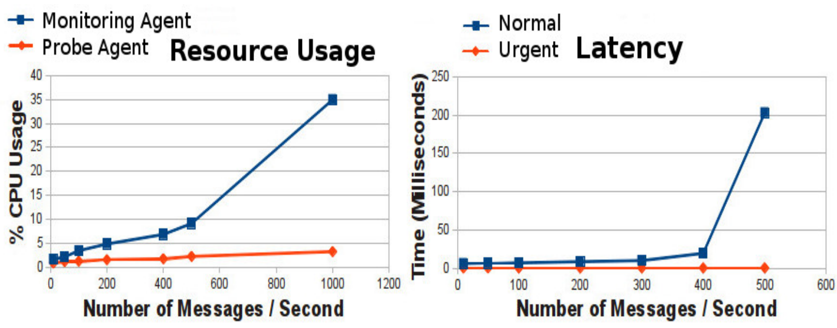

**Figure 22:** Tests of scalability and timeliness of Panoptes.

The second chart depicts the average latency of messages, calculated from the detection of the event until the end of the processing by the cloud agent. In particular, the monitoring agent notices the event, processes the message, sends it to the cluster agent which, in turn, also processes the message and sends it to the cloud agent. All the agents are in the same machine.

The blue line represents messages marked as not critical and the orange line indicates urgent messages that are sent directly to the autonomic managers by the monitoring agent. The drastic increase of latency, between 400 and 500 messages per second regarding normal messages is due to the hardware limitations of the host. Additionally, the results clearly show the usefulness of the urgency mechanism that presents almost constant and low latency compared to normal messages, even with large number of messages per second.

The results demonstrate the scalability and low invasiveness of our monitoring solution. Moreover, our adaptive approach provides flexibility to the agents to change role, migrate or alter the quantity of active metrics, to prevent message-processing bottlenecks and high latency.

## 4.8 Summary

In this chapter, we define the knowledge discovery process from multiple perspectives, specify the role of the monitoring system in autonomic clouds and devise an architecture to effectively monitor autonomic clouds, including the infrastructure and the services provided to consumers. This architecture focuses on the integration with the autonomic managers and on the following monitoring properties: scalability, adaptability, resilience, timeliness and extensibility. Nevertheless, the main requirement of such an architecture is its configuration, which needs intelligent autonomic managers to take advantage of all its potential.

Based on this architecture, we implemented a framework and demonstrated its benefits through the integration with a self-protection framework and through the analysis of its scalability and timeliness.

The experiments and the features of Panoptes demonstrate the benefits of our architecture in the domain. It drastically reduces the number of messages processed by the autonomic manager, enables the integration of the monitoring system with the autonomic system, externalises the complexity of monitoring and data processing, reduces the total number of messages processed by both systems and considers the characteristics of autonomic clouds.

# Chapter 5

# Similarity Learning in Autonomic Clouds

In the previous chapter, we described the knowledge discovery process and presented the second pillar of the self-management of autonomic clouds, i.e. a monitoring and data processing system. In this chapter, we employ this discovery process to generate a specific type of knowledge from monitoring data or SLA definitions in the cloud domain, namely the similarity among services as this knowledge is flexible and can be used in various scenarios.

The characteristics of a service consist of several types of data (called *features* in the field of machine learning), such as CPU load, memory and service duration, which form a one dimensional vector (the *observation*) describing this service. To transform data into useful knowledge, we assume no prior knowledge about services or their relations, as the information available is restricted in cloud environments. Nevertheless, the multi-dimensional correlations are difficult to extract from raw data and performance features, due to the heterogeneity of resources and services in clouds. Hence, we employ machine learning techniques to obtain such knowledge from data describing the service, in order to measure the similarity of services and to cluster[1] them using this measure. The clusters and the measure of similarity can be used in all phases of the MAPE-K loop for different purposes. For example, for optimisation of resources and anomalous behaviour detection.

First, we formulate the problem, then discuss the requirements of the clustering techniques in the autonomic domain and present our solution. Next, to illustrate the use of the proposed solution, we carry out experiments and describe a use case. Finally, we present the related works and the summary of the chapter.

## 5.1 Problem Formulation

Figure 23 describes the typical architecture of autonomic management in cloud environments and clarifies in which modules (highlighted in grey) our approach provides a novel contribution. Customers interact with the cloud system through an interface to request the execution of custom services, which are then deployed in the cloud resources, e.g. in Virtual Machines (VMs). The monitoring system uses sensors to collect raw data from the managed resources used by services. Such data are then elaborated in order to produce knowledge. The elaboration of the data follows the knowledge hierarchy defined in Chapter 4, i.e. the raw data is processed by the monitoring system, generating information which is then used as input for the inference. This inference discovers knowledge in the information and is the scope of our solution. Continuing the MAPE-K loop, the manager uses the produced knowledge in the decision-making process, which yields a plan of possible adjustments to be applied on the cloud resources and services via the actuators.

The abstraction layer created by cloud computing obfuscates several details of the provided services, which, in turn, hinders the effectiveness of autonomic managers. Moreover, the abstraction provided by clouds restricts the amount of knowledge available to autonomic managers, and

[1]Differently from previous chapters where cluster meant a group of machines (computer cluster), in this chapter, the term cluster follows the jargon used in machine learning and means groups of observations.

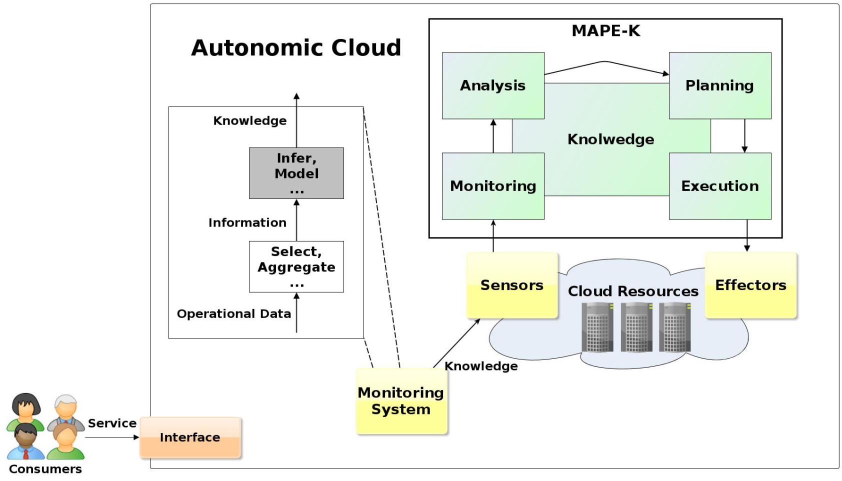

**Figure 23:** Management architecture of autonomic clouds.

consequently limits their range of actions.

Data-driven approaches to discover knowledge, without human knowledge and intervention, can assist the operation of autonomic managers. Therefore, we employ machine learning techniques to obtain knowledge from data describing the service. In particular, we provide a measure of similarity among services and cluster them using this measure. Machine learning techniques, such as *clustering*, also generate knowledge consisting of groups (clusters) of services with similar resource usage patterns. These measures of similarity and clusters can be used in all phases of the MAPE-K loop for different purposes, such as optimisation of resources, service scheduling and anomalous behaviour detection.

We illustrate the use of such knowledge with a motivating scenario. A provider enables its consumers to deploy any application in the cloud. Due to security concerns, in this scenario, the autonomic manager relies exclusively on the service description and quality of services defined in the SLA and on the monitoring information of the service. Let us assume that the autonomic manager notices that service A, which had only one available CPU, violated its SLA (for example, the completion time). Then, a new incoming service B is clustered in the same group with service A (or presents a high similarity). Instead of assigning only one CPU for service B, the autonomic manager assigns two CPUs to avoid violations of its SLA, as occurred with service A.

A critical aspect that complicates this approach is that the information provided by the monitoring contains both *categorical* (e.g. virtual machine instance type) and *continuous* (e.g. CPU load) types of data. Current approaches address this problem in a heuristic fashion: they either use only one data type, which reduces distinguishability, or construct combinations of data types by human expert intervention. Both do not cope well with the dynamism of autonomic cloud: when new types of services are introduced they may not be distinguishable or a human intervention would be necessary again.

In this chapter, to provide a *truly autonomic and effective management of services in clouds*, we propose the use of the *Random Forest* (RF) algorithm [Bre01] to learn similarities among services by using *all* data types. We learn from services already deployed in the cloud system and provide the extracted similarities to a clustering algorithm. In particular, for the sake of efficiency and meeting the dynamism requirement of autonomic clouds, our methodology consists of two steps: (1) *off-line clustering*, to learn similarities and obtain the clusters; and (2) *on-line prediction*, to predict to which of the computed clusters an incoming new service belongs.

While on-line clustering algorithms exist, they are computationally demanding. In our context, this is mainly due to the need to update the clustering as soon as a new service arrives prior to predicting its cluster. Our solution, instead, skips this updating phase, but retains and leverages in the best way possible all knowledge of the off-line step without increasing computational demand.

More specifically, the main contributions of this chapter are: (i) the analysis of the specificities of the autonomic cloud domain and the definition of the requirements of a clustering approach for cloud services; (ii) an off-line approach that relies on the RF algorithm to learn the similarities between all observed services, essentially a matrix, which is then provided

**Table 17:** Correspondence between autonomic cloud characteristics and the requirements for clustering algorithms.

| Characteristics | Requirements |
|---|---|
| Security, Heterogeneity, Dynamism | Mixed Types of Features |
| Large-Scale, Dynamism | On-line Prediction |
| Large-Scale, Multi-Agent | Loosely-Coupled Parallelism |
| Heterogeneity | Large Number of Features |
| Security, Heterogeneity, Dynamism, Virtualization | Similarity Learning |

to an off-the-shelf clustering algorithm to identify clusters; (iii) a cluster parsing to reduce the size of the matrix; which is then used by (iv) the on-line prediction to reduce computational requirements; (v) performance and accuracy analysis of the proposed methods using real-world datasets; and (vi) a use case employing the proposed on-line solution on a novel scheduling algorithm implemented in a cloud test-bed.

## 5.2 Requirements for Clustering Techniques in the Autonomic Cloud Domain

Designing or adapting machine learning algorithms to the autonomic domain is challenging [PH05]. Moreover, the cloud domain has a unique set of characteristics, which hinders the clustering task.

Table 17 presents the characteristics of the domain and the requirements for service clustering related to them. We describe below the most relevant characteristics of the domain and their impact on service clustering.

Data *security* is one of the biggest barriers for cloud adoption. Concerning the knowledge generation, Pearson and Azzedine [PB10] state that even embedded functionality for tracking and profiling the behaviour of individual services in clouds brings potential risks for privacy. Commonly, approaches to improve security are based on data cryptography and control of cross-layer transmission of information. To process the data converted with these security measures, a clustering algorithm needs to support different types of features (e.g. discrete, continuous, symbolic). Moreover, as these techniques obfuscate the features of the data, they hinder the manual combination of data types. Therefore, a data-driven similarity learning approach is required.

To offer seemly infinite pool of resources, cloud providers deploy *large-scale* clouds with distributed resources. The massive operational data generated in these environments might require a considerable amount of resources to be processed. Therefore, the knowledge discovery process should not be invasive, i.e. should not impact on the performance of the cloud services provision. Accordingly, a clustering algorithm should run in parallel to cope with the large quantity of services in acceptable time (low overhead), to divide its computational load and to operate close to the data sources, thus reducing the impact on single resources and avoiding unnecessary network traffic.

Clouds are inherently *dynamic* from the providers' perspective. New resources are constantly added and removed from the infrastructure and the types of services and the requested resources vary over time (also due to the pay-per-use business model employed in the clouds). Considering the number of services in the domain (large-scale clouds), the number of clustering requests and their inconstant arrival rate, it is impracticable to re-cluster all observations on each request. Hence, on-line prediction for new observations is a requirement for clustering algorithms. Moreover, this dynamism is enabled also by the loosely coupled nature of the cloud infrastructure; therefore, the parallelisation of the clustering algorithms should also be loosely coupled.

Cloud systems contain a *virtualization* layer. A potential risk that this layer brings to the domain is the fine-control over the monitoring of resources [FZRL08], hindering the management of such systems. In light of this loose control and of the uncertainty added by virtualization, the data is heterogeneous and also often incongruent [CFR13]. These characteristics pose significant challenges towards manual combination of data types.

Virtually everything can be provided as a service in the cloud domain. Due to such *heterogeneity*, some types of services might require monitoring data types that may not be easily converted to continuous numerical data types, which are commonly accepted by the clustering algorithm. This issue could be alleviated by human expert intervention and pre-processing, such as discretisation, normalisation and standardisation. However, these techniques are hindered by security restrictions, virtualization and the variety of services in the cloud domain. To overcome these limitations and, most of all, to avoid manual expert intervention, mixed types of features should be handled by the clustering algorithm.

Moreover, due to the heterogeneity and complexity of the available services (e.g. services with 100 features), the clustering algorithm should process them in an acceptable time and should not be invasive on the system. A possible solution is the use of techniques which reduce the number of features of the data (e.g. [CBQF10]). However, reducing the number of features also implies in loss of information which reduces the quality of the generated knowledge. Therefore, a large number of features needs to be supported by the clustering algorithm.

Autonomic computing employs the *multi-agent* architecture to enact the self-* properties. A clustering algorithm can benefit from this arrangement by parallelizing its workload.

## 5.3 Autonomic Management of Clouds Using Clustering Techniques and Similarity Learning

To achieve a meaningful measure of similarity among services in the context of autonomic clouds, we use clustering methods to learn similarities and identify usage patterns. This renders our proposed methodology versatile to deal with a wide range of application scenarios and enables the adaptation of the clusters to evolving services, which is required by the dynamic context of the domain. From the range of available clustering algorithms, we seek those that: (i) can handle mixed data types (continuous and categorical) without expert intervention; (ii) are fast both in the training and prediction phases; and (iii) offer superior performance. In the following, we first provide some background on clustering algorithms and how they are related to the above mentioned requirements, and then we proceed in defining our methodology for learning similarities based on the Random Forest algorithm and for clustering services using such obtained similarities.

Notably, the content of this section may result sometimes too technical for a reader non-familiar with concepts from the machine learning field. However, such readers can easily understand the overall approach, without going into detail of its technicalities. Moreover, this reader can appreciate the experiments carried out to demonstrate the effectiveness of the proposed solution. On the other hand, the reported technical details can be appreciated by the reader interested on a more precise idea of the functioning of the approach.

### 5.3.1 Background: Clustering as Unsupervised Machine Learning

Machine learning techniques are categorized as supervised and unsupervised. The *supervised* approach infers a function from a labelled training set, i.e. a group of examples containing observations whose class membership are known. Classes are the "group" of the observations; for instance, "small" and "large" for virtual machines. Finally, the function inferred on the basis of the training set is then used to determine the class of non-labelled observations. Evidently, the reliance on a training set, its size and labelling quality directly affect the performance of the supervised technique. This dependency hinders the adoption of supervised learning techniques in the context of autonomic clouds, as manually labelling services is particularly difficult to perform due to the domain heterogeneity, dynamism and security aspects.

On the other hand, *unsupervised learning* does not require labelled training data and is used to find structures and patterns in data.

Clustering solutions are classified in batch methods, which require all data available a priori, and on-line methods, which process the data sequentially as they are received. For an extensive review on them we refer to [XW05] and, specifically on on-line clustering, to [QPGS12, GZK05].

Few clustering solutions handle mixed types of data (e.g. [AD07, YHC04, YZ06]). HClustream [YZ06], an extension of the Clustream algorithm [AHWY03], supports mixed data types. However, it cannot handle cases where a large number of features are used (even with 10 dimensions the algorithm presents poor results) [AHWY04]. In fact, the majority of the existing on-line clustering algorithms, which handle mixed data types, share the same limitation.

When the features are purely numerical or categorical, several clustering solutions exist (e.g. [AHWY04, GRS00]). However, this implies that some information (i.e. one of the data types) will not be utilized appropriately. Using only categorical or continuous data types while possible, may lead to clusterings that cannot distinguish between services and thus may provide inferior performance. Another approach would be to hand craft new data types that combine categorical and continuous data types. This requires full understanding of the dataset, the domain and the relationships among data types. This heuristic approach is problem-specific and determines the quality of the clustering results. Moreover, devising such heuristic solutions in the autonomic cloud domain is even more complex, considering that clouds are dynamic[2], heterogeneous, use virtualization and have strong security requirements.

Another approach to deal with mixed data types is to devise data-driven solutions that can learn similarities among the observations (we refer to [YJ06] for a detailed review on them). Some of these solutions require information a priori about the data (known as supervised similarity learning), which is not available in our context. On the other hand, manifold learning approaches do not need such a priori information but are computationally intensive and do not scale well.

Thus, in this chapter, we propose a combination of a similarity learning step to discover a proper measure of similarity among observations and a

[2]In our context, this would imply building new heuristics every time the autonomic manager faces a new service type.

clustering algorithm to group the observations according to this measure of similarity. In light of the domain requirements, as the means to obtain such notion of similarity, we adopt the Random Forest algorithm.

### 5.3.2 Proposed Methodology: Service Clustering with Random Forest in the Autonomic Cloud Domain

The Random Forest algorithm relies on an ensemble of independent decision trees and was initially developed for regression and classification. It has *a training and a prediction step*. In its training step, RF uses bootstrapping aggregation (i.e. re-sampling from the dataset) and random selection of features to train $T$ decision trees (where $T$ is a number defined by the user). In the prediction step, the observations are parsed through all $T$ trees and the classes of the observations are defined aggregating the decision of each tree. For details on the classification and regression algorithms, we direct the reader to [Bre01]. The main characteristics of RF are:

- It can handle mixed features in the same dataset;
- Due to feature selection, it effectively handles data with a large number of features;
- It is one of the most accurate learning algorithms [CNM06];
- It is efficient and scales well [CNM06];
- The algorithm is easily parallelizable;
- Generated forests can be saved for future use (in our case for on-line prediction).

What is particularly relevant to our purpose is that this algorithm generates an intrinsic similarity measure. Intuitively, the principle used is the following: the more times two observations end up on the same leaf, the more similar they should be.

In [BA03], Breiman and Cutler proposed an unsupervised version of RF. Intuitively, the algorithm works as follows: (i) the training dataset (original data) is labelled as class one; (ii) the same number of synthetic observations are generated by sampling at random from the univariate distributions of the original data (synthetic data); (iii) the synthetic data is labelled as class two; (iv) the trees are trained with the original and synthetic data; and (v) the original data is parsed through the trees, which yield the references of the leaves in which the observations ended up.

More formally, the similarity between two observations $x_m,\ x_n$ ($m, n$ are the indices of the observations) is calculated as follows. Each observation is parsed through all $T$ trees of the forest; the leaves in which the observations end up are annotated as $l_m^i$ and $l_n^i$ respectively, where $i$ is the index of the tree. Let $I$ represent an indicator function, which yields 1 if two observations end in the same leaf in that tree and 0 otherwise. Thus, the similarity between two observations is defined as:

$$S(x_m, x_n) = \frac{1}{T}\sum_{i=1}^{T} I(l_m^i = l_n^i) \tag{5.1}$$

The similarity of all pairs of observations is calculated, which results in an $N \times N$ matrix, named $SIM$, where $N$ is the number of observations. The *dissimilarity matrix* (which is generated from the similarity matrix by applying $DISSIM_{nm} = \sqrt{1 - SIM_{nm}}$) is symmetric, positive and lies in the interval [0,1]. This matrix requires a considerable amount of fast memory when dealing with large datasets. To address this issue, Breiman proposed the use of the references of the leaves in which the observations ended up in each tree, generating a $N \times T$ matrix (where $N$ is the number of observations and $T$ the number of trees, where usually $N >> T$). Therefore, the forest can be build in parallel and the system can generate the dissimilarity matrix when necessary.

To cluster the observations, the dissimilarity matrix is used as input to a compatible clustering algorithm, for example, the PAM clustering algorithm [KR90]. Otherwise, the dissimilarity matrix can be transformed into points in the Euclidean space to be used as input to other clustering algorithms, e.g. the standardized version of K-means [Llo82]. The disadvantages of this extra step is the computational cost and the time necessary to perform the transformation operation.

Due to the scale of autonomic clouds and the possible high arrival rate of new observations, the domain requires very low prediction time. The unsupervised RF algorithm (successfully used in [AHT+03, SSB+05, SH06, BA03]) needs to re-execute the whole clustering process for each new observation, which is impracticable in the domain because of the high overhead of this process. The alternative is to use online RF algorithms, which learn similarities and cluster observations in an instantaneous fashion without requiring all data a priori. Unfortunately, the most known adaptations of this approach ([Has08, SLS+09, ASMC07, HH07], and even the most recent one [LRT14]) are computationally demanding and cannot make a fast prediction[3]. Finally, RF has been used for similarity learning in [XJXC12] but this solution requires labelled data, which is not available in autonomic clouds.

Therefore, we propose a novel on-line prediction algorithm based on RF, to fulfil the requirements of the domain.

### 5.3.3 Proposed Methodology: On-Line Prediction with RF

To enable fast prediction with an implementation that has minimal memory footprint, we propose a novel on-line prediction solution tailored to fulfil the requirements of autonomic clouds (summarized in Table 17). This solution takes advantage of the design of the clustering algorithm and pre-processes the trees in order to permit a fast and low memory implementation.

The outcome of the RF similarity learning is the $N \times T$ matrix, where $N$ is the number of observations and $T$ is the number of trees. As $N$ grows, this matrix may grow significantly and have a large memory footprint.

[3]Notably, we use a batch mode RF implementation for training and, thus, we need all the observations a priori. We find this to be an adequate solution since the training happens in parallel and when the system has available resources. However, as the amount of monitoring data increases, off-line training can be demanding. We could adapt an on-line RF algorithm for the training phase and still use the on-line prediction algorithm we propose. However, adapting such algorithms is not trivial, as they create intermediate leaves on the trees, which are split when a minimum gain is reached. This approach is incompatible with unsupervised learning as: (i) it creates pruned trees with maximum depth; and (ii) the observations in intermediate leaves should be re-parsed on every new split and the observations re-clustered.

We propose a solution which, instead, requires an $M \times T$ matrix, where $M$ is the number of clusters. Since $M << N$ (typically $M \leq 20$), this matrix has a very small memory footprint.

Our solution, termed **RF+PAM**, combines the strengths of similarity learning of RF with the computational benefits of Partitioning Around Medoids (PAM) [KR90] and is divided in off-line training and on-line prediction. The training phase consists of the following steps: (i) the forest is built using the training set, which is composed of the original and synthetic data (as described in the previous section); (ii) the original data is parsed through the resulting forest, which yields the dissimilarity and the $N \times T$ matrices; (iii) this matrix is given as an input to the PAM clustering algorithm, which yields the $M$ medoids for the dataset, i.e. the observation of each cluster which maximise the inter-cluster dissimilarity; and (iv) only the results of the medoids are selected from the $N \times T$ matrix, enabling us to store only the forest and this smaller $M \times T$ matrix, which consists of the references to the leaves where the medoids ended up in each tree.

In the prediction phase of RF+PAM, the new observation is parsed through the forest. Then, the $M \times T$ matrix and the results of the previous step, i.e. the leaves in which the observation ended up in each tree, are used to calculate the dissimilarity of the new observation with respect to each medoid. Finally, the new observation is assigned to the cluster whose medoid has the least dissimilarity to the new observation. Intuitively, a new observation is assigned to the cluster of the medoid which this observation ended up in the same leaf most times, considering all trees, i.e. the most similar medoid. Figure 24 illustrates the training and the prediction steps.

Since we separate training from prediction, and our training happens off-line, naturally we would expect to retrain the forest at some point. The retraining requires the definition of a mechanism to recognize when a forest should be rebuilt. However, this mechanism is problem-specific and depends on the available resources and accuracy requirements. In our context, we propose a simple but effective threshold: a user-defined ratio between the number of new observations and the total number of observations used to train the forest.

The benefits of RF+PAM are several: (i) it can be trained fast and in parallel; (ii) it handles, in a data-driven fashion, mixed data types; and (iii) it can provide predictions in a rapid and efficient manner. In Section 5.4, we will demonstrate the accuracy and effectiveness of our approach by comparing it with clustering based approaches that have been used in the context of job management, but adapted to the problem of service clustering. Since these methodologies rely mostly on the K-means clustering algorithm, to isolate and quantify the exact benefit of similarity learning, we also considered a version of our RF based approach, termed **RF+K-means**, which utilizes the K-means algorithm for clustering services and a similarity measure obtained by RF. Note that we do not necessarily advocate the use of RF+K-means, but we explained it below for completeness and for the purpose of providing a fair comparison with methodologies in the literature. We also use it as a way to showcase the superiority of relying on PAM in the context of autonomic service management.

The training phase of RF+K-means uses the same initial steps of RF+PAM to obtain the dissimilarity matrix. However, it needs an extra step before clustering. Since the standardized version of K-means uses the Euclidean distance to cluster observations, the dissimilarity matrix is first transformed into a set of points in the Euclidean space using the Multidimensional Scaler (MDS) algorithm [CC94]. Thus, the distances between the observations are approximately equal to their dissimilarity. Next, the observations are clustered using K-means, which returns the cluster assignments of the observations. The outcomes of this phase which need to be stored are the $N \times T$ matrix, the forest and the clustering assignments.

The on-line prediction phase of RF+K-means is composed of the following steps: (i) parse the new observations through the trees; (ii) calculate the dissimilarity between the new observations and *all original data* using the $N \times T$ matrix of the original data and the result of step (i), which consists of the references of the leaves in which the new observa-

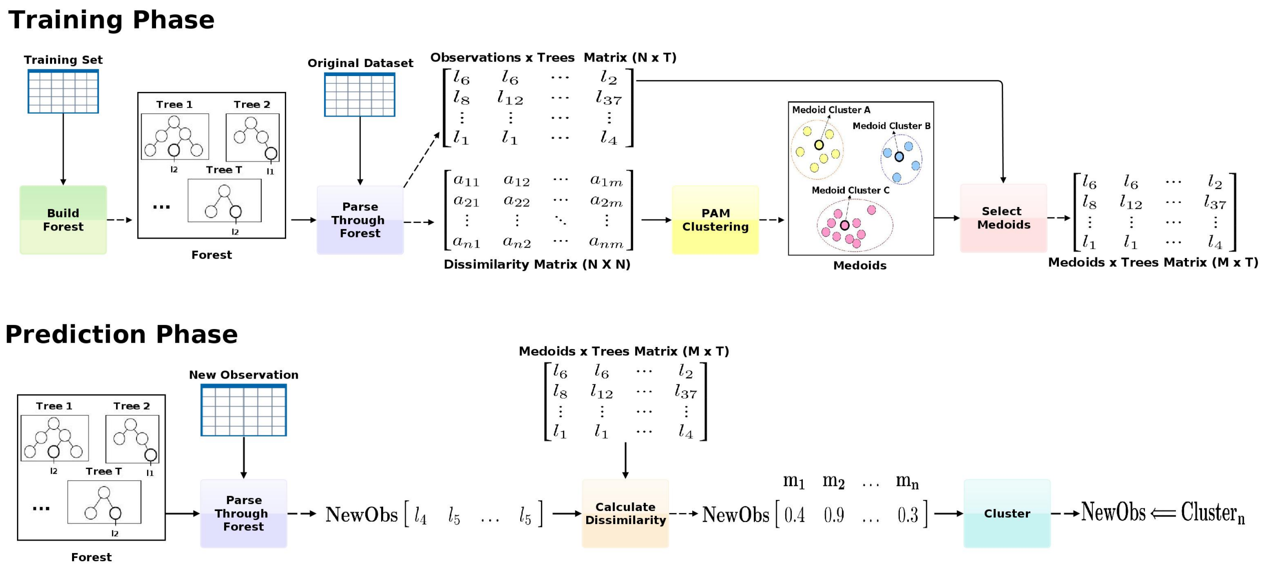

**Figure 24:** Training and prediction phases of the developed RF+PAM.

tions ended up[4]; and (iii) assign each new observation to the cluster with the least average dissimilarity between the new observation and all the observations in that cluster.

Although the differences between RF+K-means and RF+PAM are subtle, the impact is significant. RF+PAM is faster and has lower memory requirements as it uses the $M \times T$ matrix, which is much smaller than the $N \times T$ matrix used by RF+K-means. Moreover, our RF+K-means requires the MDS step, which can be computationally demanding[5].

## 5.4 Experiments

To understand the implications of the solutions described in the previous section, we carry out several experiments. These experiments are purposely designed to: (i) demonstrate the importance of similarity learning and appreciate the clustering quality compared to other methodologies using the same dataset; (ii) validate the quality of on-line prediction through the comparison of a setting with all data available a priori and a setting with less data; and (iii) present a use case to demonstrate the applicability of our solution in the domain.

For datasets, we use the first 12 hours of a publicly available dataset released by Google [RWH11] and a dataset from a grid.

Specifically, the Google dataset contains traces from one of Google's production clouds with approximately 12500 servers. The data consists of monitoring data of services[6] in 5 minutes intervals. To illustrate the content of the dataset, we list some of the available features: CPU and memory usage, number of tasks, assigned memory, unmapped page cache memory, disk I/O time, local disk space, task's requirements and priority. The complete list of the dimensions can be found in [RWH11].

[4]This solution calculates the dissimilarity between each new observation and the original data, producing only a dissimilarity row for each observation.

[5]In some cases, the use of MDS is beneficial from a flexibility standpoint, since it can open the road to the wide range of clustering algorithms that require a Euclidean distance for clustering.

[6]Notably, the traces of the dataset use the terms "jobs" and "tasks". Since this notion of job is fully compliant with our concept of service, in the rest of the paper we will only use the term "service".

The second dataset contains the traces of a grid of the Dutch Universities Research Testbed (DAS-2) [Sur15] with approximately 200 nodes. This dataset consists of the requests of resources to run services and has over 1 million observations. Among the features available in the dataset there are: Average CPU Time, Required Time, User ID, Executable ID and Service Structure.

### 5.4.1 Implementation

In order to validate the methodology described in this chapter, we implemented the RF+PAM methodology in an open-source multi-agent framework written in Python[7].

This implementation is flexible and supports different needs. It enables the clustering of a dataset; the training and storage of a forest for on-line prediction; the prediction of the cluster of new observations; the calculation of the dissimilarity matrix; and the calculation of the dissimilarity among two services.

To access these functions, we provide three interfaces: a standalone implementation which can be used locally; a library with these functions to be integrated in the development of other tools; and a distributed version, which enables the parallel training of a forest.

The architecture of the implementation is a multi-agent framework with support to distributed training where each agent has three independent components. Figure 25 illustrates the architecture of the agents and a training request.

The top component provides an interface for the agent's interaction. All agents, when initiated, register in a reference manager (an agent), which coordinates the distributed training autonomously. When a training request is received, the manager requests to each registered agent the training of a specific number of trees. When the registered agents finish building the trees, they sent these trees to the manager, who merges the results of all agents and execute the functions required by the request

[7]The source code is available in `http://code.google.com/p/unsupervised-randomforest/`.

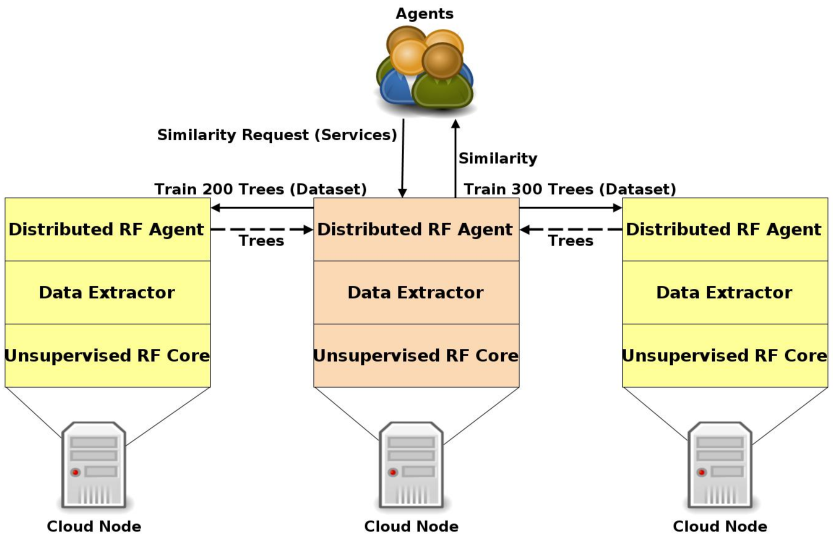

**Figure 25:** Example of a distributed set-up of RF+PAM.

(e.g. predict new observations using that forest). Notably in our implementation, the whole process is managed by a central agent and the only action required by the cloud manager (human or autonomic) is starting the agents in the resources.

The second component of the hierarchy receives the dataset from the top layer in the distributed version or reads the dataset from a file in the standalone version. This layer prepares the data and creates the synthetic class required by the unsupervised RF (for details see Section 5.3.2).

Finally, the base of the hierarchy, namely the RF core, performs the operations related to the RF algorithm. Intuitively, it builds the forest, predicts observations and calculates the dissimilarity matrix.

For the experiments, all three versions of the RF+PAM implementation were used: the standalone to demonstrate the importance of RF; the library to illustrate the on-line prediction quality; and the distributed version for the cloud use case.

### 5.4.2 Demonstrating the Importance of RF Based Similarity Learning

In this section, we evaluate the use of RF for unsupervised similarity learning in the autonomic cloud domain in an off-line setting, i.e. all observations are available for the training of the forest. In particular, we compare the clustering quality of our solution with two methodologies that used the Google's cloud dataset. Since these methodologies (**Mt1** and **Mt2**) use K-means, for a fair comparison and to illustrate the importance of similarity learning, we use here RF+K-means.

Mt1 [MHCD10] is divided into four steps: (i) selection and preparation of the features; (ii) application of the off-the-shelf K-means clustering algorithm to construct preliminary classes; (iii) definition of the break points for the qualitative coordinates based on the results of the second; phase and (iv) merging of close adjacent clusters.

While applying Mt1 in the Google dataset, the authors selected the *CPU* and *memory* features, transformed into normalised per hour values, and the *duration* was normalised and converted into seconds. In the second step, they heuristically defined 18 classes that represent the combination of: *Small*, *Medium*, *Large* for *CPU* and *Memory*, and *Small* and *Large* for *Duration*, and clustered the data points using K-means. In the third step, they employed these definitions and the clustering results to define the break points to separate the observations and, in the fourth step, they merged adjacent classes ending up with 8 clusters. Evidently, Mt1 cannot be deployed as a general solution for autonomic clouds given the necessary man-made interventions. However, since it uses the same dataset, it was considered here for comparison.

Mt2 [CG10] is defined as follows: (i) selection of the continuous (numerical) features; (ii) creation of new features based on the existing ones (even if redundant); (iii) normalisation of the data and (iv) clustering the data using K-means. Also Mt2 defined the number of clusters as 8. It is clear that, in Mt2, the categorical values are ignored and that the careful selection of the features is critical; this deviates from the approach proposed in this chapter, which aims at offering a robust and flexible solution that can accommodate many different settings.

Both methodologies employ K-means for clustering. Therefore, for a fair comparison and to demonstrate the gain from defining a dissimilarity matrix (i.e. learning the similarity between observations), we use as clustering algorithm K-means rather than PAM. Hence, we used the dissimilarity matrix, generated by the unsupervised RF similarity learning, as the input for the MDS algorithm, and the resulting Euclidean points as input for K-means clustering.

For all experiments, we defined the number of clusters as 8 (as did Mt1 and Mt2). We considered two variants of the original dataset, dropping certain features in each case: *Dataset 1* prepared for Mt1 (see the methodology definition); and *Dataset 2* which contains only all continuous features of the original dataset (i.e. categorical ones are excluded), which is used by Mt2. Then, we applied our methodology based on RF to both datasets to compare its cluster quality with the other two methodologies.

**Clustering quality measures:** Notably, unlike supervised classification where several measures to evaluate performance exist, clustering has no widely accepted measure. For Mt1, the authors used the Coefficient of Variation (CV), i.e. the ratio of the standard deviation to the mean. However, since each data dimension has a different CV, this requires an unwieldy multi-dimensional comparison with large dimensions, the interpretation of which is far from straightforward [CG10]. Therefore, in alignment with approaches in the clustering literature, here we report some of the most popular indicators for the comparison of clustering results. *Connectivity* indicates the degree of connectedness of the clusters. The measure has a value between 0 and $\infty$, with 0 being the best. *Dunn index* is the ratio of the shortest distance between data points in different clusters by the biggest intra-cluster distance (a high Dunn index is desirable). *Silhouette* measures the degree of confidence in the assignment of an observation; better clustering has values near 1, while bad clustering -1 (in the literature some works point out that over 0.75 is the best class for an observation). These indicators (and others) are analysed in [HKK05], which recommends the silhouette measure for the evaluation of noisy datasets.

**Table 18:** Comparison of RF+K-means with Mt1 [MHCD10] and Mt2 [CG10] on two differently processed versions of the Google dataset.

| | **Dataset 1** | | **Dataset 2** | |
|---|---|---|---|---|
| | **Mt1** | **RF+K-means** | **Mt2** | **RF+K-means** |
| **Connectivity** | 53.33 | 33.42 | 32.26 | 25.89 |
| **Dunn Index** | 0.01 | 0.08 | 0.06 | 0.15 |
| **Silhouette** | 0.67 | 0.98 | 0.89 | 0.99 |

Table 18 summarises the results of the experiments on the methodologies detailed above. These results show that RF+K-means performed significantly better on both datasets, considering any of the evaluation criteria. Similarity learning here outperforms the other approaches, leading to better defined clusters, even when projected to the Euclidean space with MDS. These results also demonstrate that our approach works well in the considered application domain. We should also note that, for a fair comparison, only the continuous features of the datasets were used, although our RF solution is able to handle also categorical features.

### 5.4.3 Evaluating the RF Based On-line Prediction

To assess the performance of the on-line prediction of RF+PAM, we conducted experiments to verify the agreement between two set-ups of the algorithm: a *benchmark* set-up, where all the data are available for training/prediction, and another set-up, with only a subset available for training and the remaining set used for testing. We use the set-up with all the data to obtain a ground truth cluster assignment, since all information is available and we cannot expect the algorithm (with less data to train) to perform better than that. We evaluate the on-line prediction by measuring whether unseen observations (not included in the training set) ended up in the same cluster as assigned by the benchmark set-up. Thus, accuracy in this context is measured with rand index as the agreement in the cluster assignment.

In the experiment, we first use all observations and obtain the cluster assignments for the benchmark set-up, which we accept as the ground truth. We proceed carrying out a K-Fold *cross-validation* strategy to evaluate the agreement. K-Fold cross-validation divides the dataset into $K$ partitions. It reserves one partition for testing and uses the other $K-1$ for training the trees and learning the similarities and clusters. We execute the following steps $K$ times, every time using a different $K$-th partition:

1. Train a forest using the data in the $K-1$ partitions and obtain cluster assignments;

2. Predict the cluster assignment of the observations belonging to the $K$-th partition using the on-line RF methodologies;

3. Compute the Adjusted Rand Index (see below for details) between the results of steps 2 and the ground truth of the benchmarks set-ups.

To illustrate the power of PAM, we compare the results of the above process, using RF+PAM and RF+K-means. A measure of quality for comparison of clustering methodologies is the Adjusted Rand Index (ARI), which quantifies the agreement of the clusters produced by each methodology. The maximum value, 1, indicates that two results are identical (complete agreement); value 0 indicates that the results are equivalent to random; the minimum value, -1, indicates completely different results (for more details, we refer to [HA85]).

Table 19 presents the results of the experiments considering both Google and DAS-2 datasets. The results are averaged over all K-Folds and presented along with the standard deviation (reported within parenthesis).

RF+PAM performs significantly better in the tests. This difference is due to the reliance of the K-means version on MDS to lower the dimensions and construct a Euclidean distance. Since many features are used, the dimensionality reduction step and embedding the observations in linear space (from unfolding the higher dimensional manifold), achieved with MDS, lead to poorer separability of the clusters.

**Table 19:** Clustering agreement results.

| | Google Dataset | | DAS-2 Dataset | |
|---|---|---|---|---|
| **K** | **RF+PAM** | **RF+K-means** | **RF+PAM** | **RF+K-means** |
| **100** | 0.81 (0.32) | 0.50 (0.37) | 0.70 (0.23) | 0.52 (0.21) |
| **50** | 0.75 (0.19) | 0.45 (0.19) | 0.68 (0.17) | 0.54 (0.18) |
| **20** | 0.73 (0.09) | 0.43 (0.11) | 0.67 (0.11) | 0.47 (0.08) |
| **10** | 0.70 (0.06) | 0.43 (0.13) | 0.63 (0.09) | 0.44 (0.09) |
| **5** | 0.69 (0.05) | 0.42 (0.06) | 0.61 (0.07) | 0.41 (0.01) |

Notably, these datasets are examples of real-world monitoring data from the cloud domain and are not (manually) prepared (e.g. transformation or removal of features). When comparing the results of the two datasets, we see clear improvements with high dimensional data (Google's dataset). It indicates that RF is able - without heuristic or manual expert intervention to prepare the dataset - to benefit from the additional information contained in the features to obtain clustering (through similarity learning) and can, dynamically, adapt to scenarios where the relation among features change.

### 5.4.4 Cloud Use Case

To demonstrate the applicability of the on-line RF+PAM methodology in the domain, we propose a scheduling algorithm based on the similarity between services. Intuitively, the scheduler assigns an incoming service to the node executing the most dissimilar services, thus avoiding race conditions for the node's resources. For each node, the scheduler averages the dissimilarity between the new service and the services running in that node, then it assigns the service to the node with highest average dissimilarity.

The scheduling steps are detailed in Algorithm 1. The scheduler receives the new service and the list of nodes as parameters, which also contains the list of the services running in each node. Then, it clusters

**Algorithm 1** Calculate the dissimilarity between a new service and the services running in the nodes of the cloud.

```
1: procedure CALCULATE_DISSIMILARITY(new_Service, node_list):
2:     new_Service.cluster ← CLUSTER_SERVICE(new_Service.SLA)
3:     for node in node_list do
4:         node_dissimilarity ← 0
5:         for service in node do
6:             dissimilarity_servs ← dissimilarity(new_Service, service.cluster)
7:             node_dissimilarity ← node_dissimilarity + dissimilarity_servs
8:         if node_dissimilarity > 0 then #Average Dissimilarity
9:             node_dissimilarity ← node_dissimilarity/len(node.services)
10:        else #No Services in the node, best case
11:            node_dissimilarity ← 1.1
12:        nodes_dissimilarity.append([node, node_dissimilarity])
13:    ASSIGN_SERVICE_TO_NODE(new_Service, nodes_dissimilarity)
```

the new service and calculates, for each node, the dissimilarity between the new service and all services running in that node. According to the RF+PAM methodology, this dissimilarity is calculated between the new service and the cluster medoids of the running services. Then, if there is at least one service running in the node, the total dissimilarity is divided by the number of services. Otherwise, since no service will compete for the same resources, the dissimilarity for the node is defined as 1.1 to prioritize it in the assignment phase (as the maximum dissimilarity is 1).

In the assignment phase, the scheduler assigns the service to the node with most dissimilar services, after verifying whether it has enough resources to run the service. Algorithm 2 illustrates this process. In particular, the nodes are sorted by their dissimilarity, i.e. the most dissimilar services have priority in the list. Then, the algorithm verifies whether a node has the resources to run these services till it finds a compatible node or it test all nodes always prioritising the nodes on the top of the list. If there is no node with available resources, the service joins a waiting list.

Finally, when a service terminates, the scheduler selects the compatible service from the queue (a waiting list) with the highest dissimilarity to the

**Algorithm 2** Assigns the new service to the node with enough resources and the most dissimilar services.

```
1: procedure ASSIGN_SERVICE_TO_NODE(new_Service, nodes_dissi):
2:     nodes_dissi ← SORT_BY_DISSIMILARITY(nodes_diss)
3:     for node in nodes_dissi do
4:         if node.available_resources > new_Service.resources then
5:             start_Service(new_Service, node)
6:             return node
7:     return 'Not enough resources, service added to the Q'
```

services running in this node (not considering the terminated one). More specifically, the algorithm tests the dissimilarity between the services in the waiting list with the services running in the node. Then, it executes the most dissimilar service, verifying first whether enough resources are available. Moreover, to avoid long waiting times on the selection, the manager can set a maximum number of services to test, which are randomly selected from the waiting list (e.g. randomly select 25 services and executes the most dissimilar).

We employed these concepts in a framework that coordinates the execution of services. In our use case, services are applications defined by a SLA, which are executed in a cloud. We carried out experiments using a cloud test bed with 9 VMs.

To assess the performance of the dissimilarity scheduling, two other scheduling algorithms were used in addition to the Dissimilarity algorithm. In the first, named *Isolated*, each service runs without any other service in the same VM, thus having all resources available for the execution of the service. The second (named *Random*) assigns the services randomly to the nodes. Moreover, the Random and Isolated algorithms also have a queue for services and, when a services finishes, a compatible service is assign to the node using the same algorithm. Notably, all three algorithms (Random, Isolated and Dissimilarity) have the resource admission control and services are assigned only to machines that have enough resources to run them.

In the experiments, services are generated randomly at the beginning of every round of tests and the same services are executed using all three described algorithms. Each service has an associated SLA, which is created along with the service based on an estimation of the resources necessary to finish the service within the completion time. The created dimensions are: CPU, RAM, requirements, disk space, completion time and network bandwidth. The services in the experiments are of different types, such as web crawling, word count, machine learning algorithms, number generation and format conversion, which are close to real-world applications [NMRV11].

To measure the performance of the three algorithms, we compare the sum of the overall runtime of the services given the same input services. In particular, we vary the number of input services (from 50 to 250) for each algorithm and sum up the runtime of the services. This procedure was repeated 10 times for every input, averaging the results.

Figure 26 presents the results of the experiments. Dissimilarity algorithm performs significantly better than the Random algorithm, reducing in 25% the total run time. The best case[8] in our experiment, the Isolated algorithm, is around 20% better than the Dissimilarity. However, in this algorithm, each service runs alone in the resource, which is impracticable in real world deployments as it would lead to low resource usage (idle resources) and long waiting times for services.

With the Dissimilarity scheduler, we illustrated the potential benefits of a metric learning and clustering algorithm in the autonomic cloud domain. Indeed, in real-world deployments, other aspects of the incoming services, such as service priority or SLA violation probability, must be considered for designing a scheduler. Yet, considering the results of the experiments, more complex schedulers can benefit from integrating dissimilarity scheduling in their solutions.

[8]The results of the Isolated algorithm have been calculated and reported in the chart to offer the reader a comparison of the performance of our approach with respect to the minimal total run time possible in this setting, i.e. the best case.

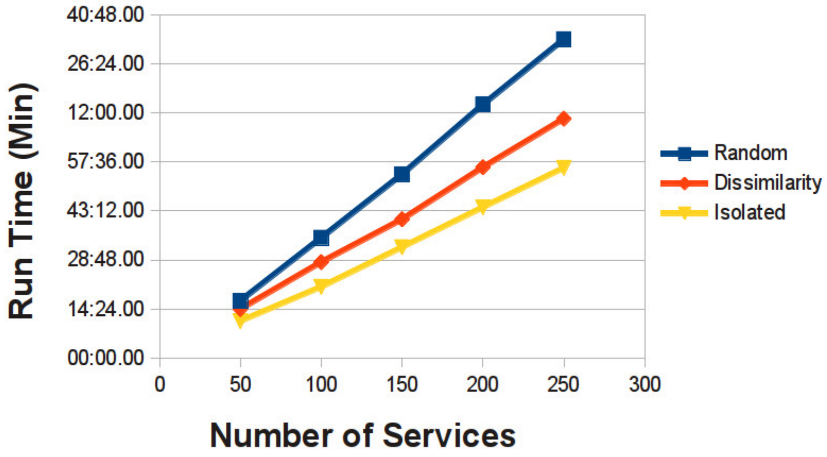

**Figure 26:** Total run time of the scheduling algorithms.

## 5.5 Related Works

In this section, we discuss the relevant literature in the cloud domain that uses a notion of similarity to support decision systems with knowledge. In the *service scheduling* field, several works, e.g. [LC08, SRJ10, SSPC14, QKP$^+$09], use a measure of similarity. However, they consider only numerical features and, as discussed in Section 5.1, the domain requires the support of different types of features. In our use case, we propose a service scheduling algorithm, which uses the knowledge on similarities among services to avoid race conditions in the cloud resources. A similar approach was presented in [NMRV11]; the authors manually combine features and employ a supervised Incremental Naive-Bayes classifier to assign a service. However, this approach depends on the hand-crafted combination of features, which is problem-specific, and on several parameters defined by the administrators.

Regarding the *application profiling* field, most approaches are problem-specific, e.g. [WSVY07, DCW$^+$11] focus only on VMs. Hence, they cannot cover the diversity of the services and the heterogeneity of clouds. The solution of Kahn et al. [KYTA12] on *workload characterisation* clusters workload patterns by their similarity. However, their similarity clustering algorithm is based on simple heuristic metrics to accommodate VMs, which does not cope with the dynamism of the autonomic cloud domain.

In the *anomalous behaviour detection* field, [MWZ$^+$13] uses a heuristic notion of similarity to cluster service requests and detect anomalous behaviours. Similarly, Wang et al. [WWZ$^+$13] propose a methodology to detect anomalies for Web applications in which the similarity among the workloads is used to detect problematic requests. However, both works do not consider different types of features.

In summary, most works in cloud, which employ a notion of similarity, implicitly assume: homogeneity on the resources and services; preparation and normalisation of the data for the clustering process; and good representation of the relations of data features. Our clustering solution, instead, does not rely on these assumptions and is not problem-specific. Thus, it can be used with any kind of service. Therefore, we advocate that our solution, or an adaptation of our approach, could significantly improve the decision-making in autonomic clouds.

## 5.6 Summary

The characteristics of autonomic clouds hinder their management and the decision-making process as they obfuscate several details of the provided services and of the infrastructure. In order to assist the autonomic managers in the decision-making, we devised a machine learning solution to learn the similarity among services since the existing solutions do not cope with the characteristics of clouds. This knowledge has a wide range of applications in the domain, both to provide the similarity knowledge to the autonomic managers and to serve as the basis for other solutions which generate new knowledge using the similarity notion, e.g. for the detection of anomalous behaviour or for application profiling.

In particular, we devise a novel clustering methodology based on Random Forest and PAM. We validate it through several experiments. The first experiment, regarding the clustering quality, shows the superiority of our solution in comparison to two other methodologies devised specifically for the used dataset, which is a real-world cloud dataset. The second experiment shows a high-agreement between set-ups where all data is available a priori, and set-ups in which only a part of the dataset is available a priori and the remaining is predicted on-line. It confirms the quality of our on-line prediction. Finally, in the last experiment we illustrate the applicability of the solution in the domain devising a novel scheduling algorithm, which uses the notion of similarity to assign incoming services to cloud resources with most dissimilar services, thus avoiding race conditions for the nodes' resources.

The results of all experiments demonstrate significant benefits of our methodology: superior performance, low memory footprint, support to mixed types of features, support to a large number of features and fast on-line prediction. Moreover, these encouraging results offer insights into the potential of approach as it not only provides a valuable knowledge to feed the autonomic managers but can also be used by other methodologies to discover new knowledge (e.g. anomalous behaviour detection).

# Chapter 6

# Polus: Integration and Use of SLAC, Panoptes and Similarity Learning

In the previous chapters, we have described the design of: (i) a domain-specific language, SLAC, which provides the support for the definition of SLAs in the cloud domain; (ii) an extensible monitoring architecture devised for the autonomic cloud domain; and (ii) a machine learning methodology to determine the similarity among services devised to fulfil the requirements of the domain. In this chapter, we discuss the integration of all these solutions and present the application of the resulting framework, named Polus[1], to a use case.

In particular, in order to support this integration, we design knowledge extraction component, which enables the discovery of knowledge from multiple sources and on-demand. The advantages of adding this artefact to Panoptes in autonomic clouds are:

- It centralises the knowledge generation into a single system;

[1]The framework was named after Polus (latin equivalent for Coeus), one of the titans in the Greek mythology. He was the titan god of intellect and represents wisdom and intelligence.

- It processes data close to the source, which reduces the number of messages exchanged in the cloud and the amount of data analysed;
- It facilitates the integration with autonomic managers as it provides a single interface for monitoring and knowledge generation.

We also describe the adoption of our solutions in a use case which employs a hybrid cloud. This cloud consists of the academic cloud of IMT and complementary resources from a public cloud. In the use case, we develop service schedulers which employ the Polus framework to decide where to allocate new services.

The results of the use case show that Polus brings many benefits for the autonomic management of clouds. Moreover, they also indicate that our architecture would be appropriate for other scenarios in cloud environments.

## 6.1 Polus: A Framework for Providing Knowledge for the Cloud Autonomic Management

In the first part of this section, we discuss an extension of Panoptes to support the knowledge generation process, which may include information that is not related to the monitoring of the cloud. Then, in the second part, we present the architecture of Polus. Moreover, we discuss the role of each component of this architecture and the interaction among them.

### 6.1.1 Knowledge Extraction Components

The knowledge generation process of Panoptes is performed exclusively through monitoring modules, which transform monitoring data into knowledge. However, the knowledge necessary for autonomic management of clouds is provided by multiple sources and is not restricted to monitoring data.

In view of this limitation, we design *Knowledge Extraction Components* (KECs) for Panoptes. These components generate knowledge from multi-

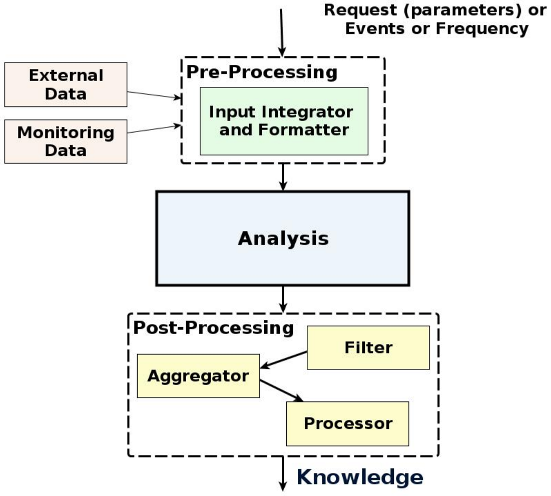

**Figure 27:** Knowledge extraction component in Panoptes.

ple sources and enable the use of different methodologies to extract this knowledge.

In Figure 27, we show the main steps performed by KECs in Panoptes. The process can be triggered by:

- a *request*, in which the requester can send parameters for the knowledge extraction analysis;
- on *events*, such as the arrival of new monitoring data (retrieved by monitoring modules of Panoptes);
- on a specific *frequency*, with defined intervals (for instance, every 20 minutes).

This process supports the following sources of data, information and knowledge: *external*, which can be read from datasets or text files; *monitoring*, i.e. the results of Panoptes monitoring modules; and *parameters* sent with the request for the execution of the component.

After the process is triggered, the multiple sources of data are integrated and formatted in the *pre-processing* phase. Next, the *analysis* of the data is performed. The analysis can use functions that are available in Panoptes or external tools, based on, e.g. machine learning and statistics techniques. Finally, in the *post-processing* phase, the results of this analysis are filtered, aggregated, processed and then delivered to the interested parties.

Furthermore, many algorithms and methodologies for knowledge extraction require a preparation phase (e.g. building the model or training the algorithm), which is executed before the knowledge extraction process. This preparation is performed through a command. For example, in the case of the RF+PAM methodology, the forest needs to be trained before the prediction phase (for details we refer to Chapter 5). Therefore, in a KEC this training is performed before use, i.e. in the preparation phase. Moreover, in the RF+PAM methodology as well as other methodologies the model used by the knowledge extraction component needs to be updated. Hence, a frequency and a specific command can be defined for updating the model.

Listing 6.1 demonstrates the configuration of a KEC, which was developed to integrate the RF+PAM methodology with Panoptes and will be used later in this chapter for the analysis of services. It calculates, on request, the dissimilarity between a new service and the service running in a node.

In the general description section of the configuration, we define: the name as Service Analyser, a textual description and how often it should be executed. Then, we define the preparation phase that specifies the command to train the algorithm. Next, we present the updating frequency and the command to perform this update. The data sources of this component are two parameters and a monitoring module, which are integrated in the pre-processing as shown in Figure 27. The first parameter has the characteristics of the service which will be used by the RF+PAM algorithm to cluster the service and to calculate the dissimilarity. The second parameter is used to select the module, which is another data source. This module retrieves the list of services running in the nodes and sends them (their IDs) as parameters to the RF+PAM methodology. The knowledge extraction (the analysis phase) is then performed through the command line. In this configuration, the results are filtered (only the numbers, i.e. the total dissimilarity), which is part of the post-processing, and, finally, sent to the specified destination. Note that the configurations which are not determined in the component take the default value, e.g. when no aggregation is defined, no aggregation is set.

**Listing 6.1:** Configuration of the Service Analyser KEC.

```
General
        Name = Service Analyser
        Description = Unsupervised Random Forest...
        Frequency = OnRequest
Prepare
        Command = RF+PAM 100, 100 --file=training.cvs
Update
        Frequency = 30 min
        Command = RF+PAM -id 12 -retrain
DataSources
        Parameter =  newService
        Parameter = nodeID
        Module = ServicesNode_$nodeID
Analysis
        Command = RF+PAM -dissimilarity -new
$newService -services ServicesNode_$nodeID
        Filter = *Numbers*
Destination = localhost:100
```

The design of KECs aims at closing the gap between the knowledge discovery process, the monitoring system and the autonomic managers. Integrating them in Panoptes combines the knowledge discovery and one of the main sources of data for this process: the operational data. Moreover, it centralises two important sources of knowledge for the autonomic manager in the cloud domain and facilitates the integration of these two systems.

Moreover, integrating this process with the monitoring system has many potential advantages. For instance, the complexity of the knowledge generation is centralised in the monitoring system, it enables the

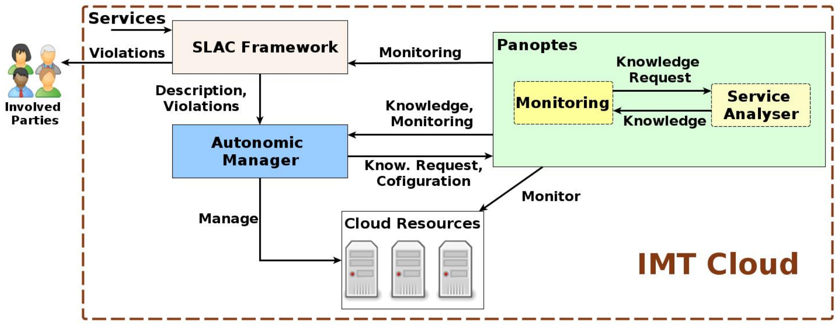

**Figure 28:** Architecture of Polus.

application of such algorithms in different data abstractions (e.g. service, node, cluster), it provides a single interface and, if the source of data is the monitoring itself, the knowledge discovery process is executed close to the source of data and the number of messages are reduced. Also, Panoptes architecture is devised to support the collection and transformation of data into knowledge in the cloud domain, which leverages this process.

## 6.2 Integration of the Proposed Solutions

In this section we present the integration of the solutions proposed in this thesis for autonomic clouds and discuss their role in the resulting framework (i.e. Polus). Figure 28 illustrates the interaction between the components of Polus, the autonomic manager and the resources.

The SLA of the services accepted by the provider are sent to the *SLAC Framework*, which parses the SLA and generates the constraints of the SLA and the definition of the service. The results of the former process are sent to the autonomic manager for the deployment of the service, while the results of the latter are used to evaluate the conformance of the service and the SLA using also the monitoring results. Moreover, it sends reports about the enforcement of the SLA to the autonomic manager and the parties involved in the agreement.

*Panoptes* plays a central role in Polus. It monitors the cloud and services, processes the operational data and delivers knowledge to the autonomic managers of the cloud and only the service monitoring information to the SLAC Framework. It also supports requests for specific knowledge. In particular, we defined KECs to support the RF+PAM methodology developed in Chapter 5 and named the component implementing this methodology as *Service Analyser*.

The *Autonomic Manager* manages the resources as services using the information and knowledge provided by Polus. It also configures modules and KECs for the provision of the knowledge necessary for the execution of its functions and for the provision of monitoring information of the services to the SLAC Framework. The latter configuration uses the automatic conversion of SLA terms into low-level metrics and the support in Panoptes to generate the monitoring results in the format supported by the SLAC Framework.

Notably, the Polus framework neither covers the autonomic management of the cloud, nor the SLA negotiation and nor the service admission.

Although Panoptes and the SLAC framework are compatible and interact with each other, we opt for not including the SLAC Framework into Panoptes. This decision was taken since the evaluation of SLA should be independent (and auditable) and as it may be performed by independent entities, such as an auditor.

## 6.3 Use Case: Scheduling in Hybrid Clouds

In this section, we present a use case employing the SLAC Framework and language, Panoptes and the Service Analyser to demonstrate the benefits of the solutions proposed in this thesis. We design a service scheduler, which uses our solutions to generate the information and knowledge for the decision-making process and decides where to allocate new services. The scheduling process is a component of the autonomic management; therefore, this use case also provide insights into the integration of our solutions with the autonomic managers.

The use case is divided into two parts:

- An extension of the use case presented in Section 5.4.4, which includes the support to SLAs. For this scenario, we performed experiments in the IMT cloud to compare an approach that schedules services randomly with the Dissimilarity scheduler developed in Chapter 5. The results show that, with the support of the knowledge generated by our solutions, the Dissimilarity scheduler can achieve significant reductions in the number of SLA violations (up to 48%).

- The second scenario uses a public cloud to mitigate the SLA violation risks. This setting, where a private cloud uses a public cloud to complement its capacity, is called *hybrid cloud*. In this cloud, before assigning a node to allocate a new service, the scheduler must decide whether to execute this new service in the local cloud or in the public cloud. To take this decision, we propose two schedulers: a Risk-Aware, which considers only the SLA violation risk; and a Cost-Aware scheduler, which also considers and compare the cost of allocating the service locally and in the public cloud. We performed experiments which demonstrate the benefits of these schedulers and, mainly, the benefits of using Polus to provide the necessary knowledge to autonomic scheduling.

The two schedulers of the second scenario differ only in the decision-making aspect. The first scheduler considers only the violation risk of the SLA and, when the risk is higher than a threshold set by the policies of the system, the service is sent to be executed in the public cloud. The second variant, considers also the financial aspects of executing the service in the public cloud. It analyses the cost of executing service locally, of executing it in the public cloud, the violation risk and penalties for SLA violation. The novelty of these approaches for scheduling in hybrid clouds is the inclusion of the risk of SLA violation in the model for the decision-making.

Figure 29 details Figure 28 by showing the elements of each component that are employed in our use case scenarios.

The autonomic manager schedules[2] new services only locally (in the

[2]In this use case, we propose solutions for the decision-making process of the autonomic scheduling of services to demonstrate the benefits of our solutions with concrete examples.

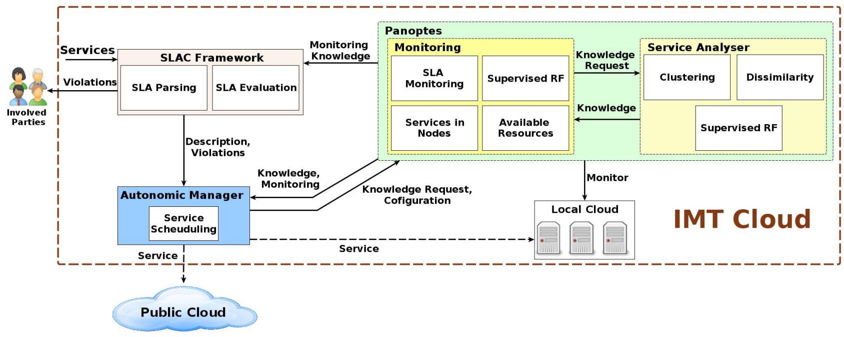

**Figure 29:** Detailed Polus architecture for the use case.

first scenario) or decides whether to allocate the service locally or in the public cloud (second scenario). The SLAC Framework *parses* and *evaluates* the SLAs which contain the service specification and requirements. Panoptes provides the status of the system (*Monitoring*) and knowledge about the service (*Service Analyser*), e.g. dissimilarity, violation risk.

### 6.3.1 Dissimilarity Scheduler and SLAs

In this first scenario, we employ the Dissimilarity scheduler, presented in Chapter 5.4.4, to automatically assign new services to cloud nodes. More specifically, we define a scheduler that receives service requests and assigns them to nodes based on the compatibility of the service's requirements and descriptions (defined in the SLA) with the SLAs of the services being executed in each node. This process requires knowledge on the dissimilarity among services and the status of the cloud, which are provided by Panoptes. Additionally, the SLAC framework is used to parse and evaluate the SLAs of the services and the SLAC language to define such SLAs.

Intuitively, the scheduler uses the dissimilarity knowledge to decide in which node to allocate the new service, and requests the autonomic manager to deploy the service in the chosen node. Then, it configures the

However, the decision-making is not the focus of this thesis, instead, the emphasis is on the provision of knowledge to base such decisions.

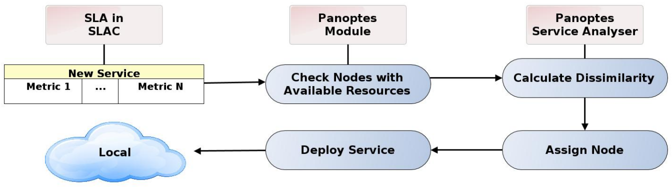

**Figure 30:** Main steps of the autonomic scheduling of a new service and the components which feed or interact with the scheduler in each step.

monitoring system to collect information about this new service and to send the monitoring information to the SLAC Framework, which periodically evaluates the SLA. Figure 30 depicts the main steps of the scheduling process which are described as follows.

The service is defined using the SLAC language. The scheduler requests to Panoptes which nodes have available resources to execute the service. With the list of nodes in the local cloud that have available resources, the scheduler requests to Panoptes the dissimilarity between the SLA of the new service and the services running in each node of this list. With the dissimilarity knowledge, the scheduler assigns the new service to the node with most dissimilar services (i.e. the node with the highest dissimilarity) and deploys the service in that node.

To assess the advantage of this methodology, we define a Random scheduler that assigns the services randomly to the nodes. In both schedulers, i.e. Dissimilarity and Random, the services are assigned only to nodes that have enough resources to execute them (resource admission control based on the SLA).

In order to evaluate the performance of our solution, we conducted experiments using 9 VMs of the IMT cloud. The services were generated and executed in this cloud to test the number of SLA violations of both schedulers, i.e. Random and Dissimilarity. For the tests, different types of services were dynamically generated, e.g. web crawling, word count, learning algorithms, number generation and format conversion, which are close to real-world applications [NMRV11]. For each service, an associated SLA is created based on an estimation of the resources necessary to complete the service within the specified completion time. This estimation was performed beforehand through the benchmarking of each type of service using different resources of the cloud. The metrics in the service definition are: CPU, RAM, Requirements, Disk Space, Completion Time and Network Bandwidth.

In real-world clouds, new services arrive in variable intervals. In our scenario, we assume that the services' arrival is a Poisson process with parameter $\lambda$, which defines the intensity of these arrivals. Intuitively, the higher the $\lambda$, the more often new services are requested, e.g. for $\lambda$ set to 0.2 a service arrives in average every 5 seconds, while for $\lambda$ set to 1 the same happens on every second. We vary the value of $\lambda$ in the experiments to analyse the performance of the cloud with different loads. On every experiment, we generate 100 services and run both algorithms to schedule these services. Finally, we repeat this procedure 5 times for every $\lambda$.

The results of the experiments are shown in Table 20 and graphically plotted in Figure 31. Using the Dissimilarity methodology, the SLA violations were reduced up to 48% and, on average, 33%. The Dissimilarity scheduler performs better when the cloud is not overloaded since it has more options to allocate services in the node with the most dissimilar ones. However, even with high arrival rates (worst case for this scheduler), our solution performs significantly better as it allocates the services that use different resources together. This approach reduces the competition for the resources of the node, thereby improving the performance of the cloud.

### 6.3.2 SLA Risk Management in Hybrid Clouds

The IMT cloud used in the previous scenario has limited amount of resources and, as seen in the results of the experiments, the performance of the cloud declines with higher arrival rates. Consequently, the number of SLA violations raises. In order to reduce this number, we extend the previous scenario to support hybrid clouds, i.e. to use a public cloud to complement the resources of the local cloud.

Since we want to show how the performance of the hybrid cloud is

**Table 20:** Average Number of SLA violations and standard deviation (in brackets) using the Random and Dissimilarity schedulers with different arrival rates ($\lambda$).

| $\lambda$ | Random | Dissimilarity | Violation Reduction |
|---|---|---|---|
| **0.2** | 12.4 (3.78) | 6.4 (2.88) | 48% |
| **0.4** | 15.8 (3.27) | 9.4 (2.07) | 40% |
| **0.6** | 24 (4.00) | 16.6 (2.51) | 30% |
| **0.8** | 31.2 (2.58) | 20.8 (3.70) | 33% |
| **1** | 32.8 (1.92) | 26.8 (3.56) | 18% |

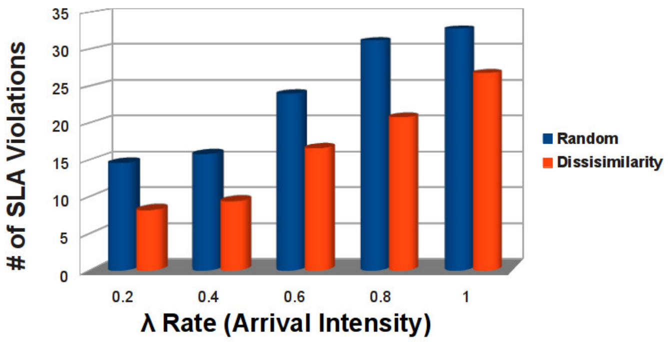

**Figure 31:** Average number of SLA violations with different arrival rates ($\lambda$).

affected by the latency and the impact of limited access to the remote infrastructure, we use a cloud offered by one of our partners in Brazil, the Federal University of Santa Catarina (UFSC), for the role of public cloud. The UFSC cloud provides IaaS. However, for the sake of simplicity, i.e. to abstract the management of the services and resources in such an environment, we simulate a public *PaaS* provider that runs services in their cloud. In this cloud: we employ 10 large VMs using the Apache CloudStack [Apa15] and execute the services using the same solutions as the ones in the local cloud (which uses SLA as formalisation of the services and its guarantees).

In this setting, we propose two scheduling solutions, which decide

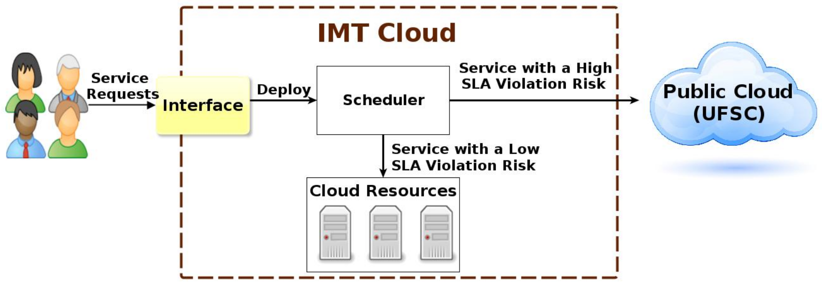

**Figure 32:** Risk-Aware scheduler scenario.

whether to execute a service in the local or in the public cloud. In the first scheduler, the decision is taken based on the SLA violation risk. To calculate this risk, we propose the use of a machine learning algorithm which considers the historical data. The scheduler compares the calculated risk with a threshold defined in the high-level polices of the system to decide where to execute the service.

However, in real-world scenarios, executing services in a public cloud might have a higher cost than the in-house cloud. Considering also the financial aspect, analysing the risks and deciding where to schedule new services (locally or in the public cloud) is a significant challenge in the area. Therefore, we also propose a variant of this scenario, where we consider the financial terms in the decision-making. In particular, we design a scheduler, named Cost-Aware, which takes into account the costs of running the new service in the public cloud, of running this service locally and the penalties in case of violation for allocating new services. Notably, we do cover the management of the resources in the public cloud, which should also manage the elasticity and the placement of new services and minimise the costs while enforcing the SLA.

### Risk-Aware Hybrid Cloud Scheduler

In hybrid clouds, the decision of where to execute the service is influenced by many aspects, e.g. the confidentiality of the data necessary to execute the service, the resources available for the execution of the service and the costs involved. In this section, we aim at minimising the number of SLA violations by sending the SLAs with high violation risk to the public cloud as depicted in Figure 32. Therefore, we specify a scheduler, named Risk-Aware, which, to decide where to allocate a new service, also considers this violation risk. Apart from the scheduler, this setting is the same as the previous scenario.

The main steps of the scheduling algorithm are shown in Figure 33. Intuitively, the scheduler requests Panopotes to produce as much knowledge as possible to base the assessment of the SLA violation risk. The steps are detailed as follows.

First, the scheduler requests to Panoptes which nodes have available resources to execute the new service. Then, it asks to Panoptes (which uses the Service Analyser component to answer this request) the group of the service, i.e. its cluster. This knowledge is used not only to calculate the dissimilarity but also as a parameter to assess the violation risk. Therefore, differently from the previous scenario, the scheduler stores this knowledge to avoid defining the cluster knowledge of the new service twice since it is used in two steps: in the dissimilarity step; and in the violation risk step. This entails the reconfiguration of the dissimilarity component of Panoptes defined in Listing 6.1. Instead of invoking the Service Analyser also for clustering, this component receives the cluster as a parameter and adds it to the command which invokes it. With the dissimilarity knowledge yielded by this component, the scheduler selects the node with the highest dissimilarity between the new service and the services running in that node.

The SLA violation risk assessment is performed using a machine learning classification algorithm. The idea behind the algorithm is to define two classes of services; in our case we classify between services which SLA was *violated* (or will be violated) and which SLA was *not violated* (or will not be violated). Then, the algorithm uses the results of finished services to learn the patterns of these classes to define whether a new service might be violated. In particular, we employ the supervised Random Forest[3] algorithm which not only classifies the new service in one

[3]Notably, the *supervised* Random Forest algorithm is different from the solution proposed

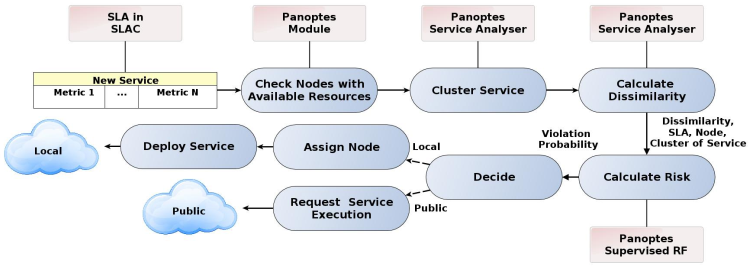

**Figure 33:** Main steps of the Risk-Aware Scheduling of new services.

of the classes (i.e. the service is likely to be violated or not violated) but also provides the probability of violation. Although on-line supervised random forest algorithm exists, for the sake of simplicity, we used the standard algorithm and re-train it in fixed intervals.

Therefore, to feed the supervised RF algorithm, we send the available information and knowledge about the service, which in our case are: the SLA; the node characteristics; the cluster of the service; and the dissimilarity value. These are provided to the supervised RF component of Panoptes, which returns the SLA violation risk. In case no node has available resources to run the service in the local cloud, taking into account that the service can be deployed in a node with similar services, we propose the assessment of the risks considering the minimum dissimilarity (the worse case, i.e. 0) and, if the risk of violation is low, the service is added to a service queue of fixed length in the local cloud. This evaluates the risk of executing the new service with the worse combination of services in a node.

The decision of where to deploy the service is based on a risk threshold defined by the policies of the system. Therefore, if the violation probability is higher than this threshold, the service is sent to the public cloud, otherwise it is run locally.

To assess the effectiveness of the risk-management scheduler, we use the scheduler developed in the previous scenario, namely the Dissimi-

Chapter 5, namely RF+PAM, which was developed for clustering while this is a classification problem (supervised).

**Table 21:** Average number of SLA violations and standard deviations (in brackets) using the Dissimilarity and Risk-Aware schedulers with different arrival rate ($\lambda$).

| $\lambda$ | Dissimilarity | Risk-Aware | Violation Reduction |
|---|---|---|---|
| **0.2** | 7.2 (3.03) | 4.4 (1.51) | 39% |
| **0.4** | 10.4 (2.96) | 5.8 (2.58) | 44% |
| **0.6** | 14.4 (2.60) | 9.2 (1.304) | 37% |
| **0.8** | 24.4 (1.94) | 17.0 (2.12) | 30% |
| **1** | 25.8 (2.58) | 17.4(4.33) | 31% |

larity scheduler, which assigns all resources to the local cloud. Then, we compare the number of SLA violations that arise when using the two schedulers.

As the previous scenario, the experiments were carried out in the same test-bed by varying the arrival rate as before and by employing a fixed threshold of 50%, i.e. if the SLA violation risk is higher than 50%, the SLA is sent to the public cloud.

Table 21 details the results and Figure 34 illustrates the number of SLA violations. This approach, backed by the knowledge generated by Panoptes, was able to reduce the SLA violation up to 44% in the best case, and on average 36%. This improvement is even more significant, considering the low number of services sent to the public cloud. In this setting, the reduction of the SLA violations was achieved sending in average only 6.4% of the services to the public cloud.

### Cost-Aware Hybrid Cloud Scheduler

The previous scenario shows the benefits of using a public cloud to execute services with high-violation risk. However, the extra resources provided by the public cloud have a cost, which should be considered by the scheduler to decide whether to send a service to the public cloud. For instance, if a service has no penalty associated with the violation of its SLA, it might be more appealing to a provider to risk the violation of this service, rather than paying for its execution in a public cloud. Therefore,

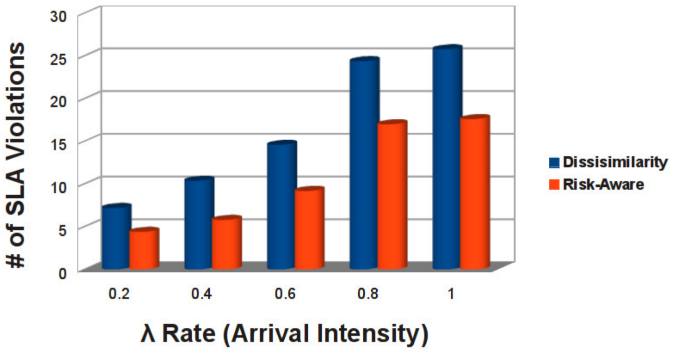

**Figure 34:** Average Number of SLA violations with different arrival rates ($\lambda$).

instead of deciding on the basis of the violation risk only, we design a more realistic scheduler, which takes into account also the penalties defined in the SLA, the cost of executing the service locally and the cost of using the public cloud to execute the service.

All scheduling steps of this scenario are the same as the previous scheduler apart from the decision function. Instead of comparing SLA violation risk with a pre-defined threshold, the scheduler takes into consideration the cost of running the service locally, of executing it in the public cloud and allocates the service in the cloud with the lowest cost.

We define a model to calculate the total cost of running a service ($s_i$) in the public cloud ($C_{s_i}^{public}$) and in the local cloud ($C_{s_i}^{local}$):

$$\begin{aligned} C_{s_i}^{public} &= Price_{s_i}^{public} \\ C_{s_i}^{local} &= Price_{s_i}^{local} + Penalty_{s_i} \times Risk_{s_i} \end{aligned} \tag{6.1}$$

In the public cloud, only the price for running the service is considered. This refers to the instance hour price (commonly a VM instance) and is used by most commercial cloud providers. On the other hand, the cost of executing the service in the local cloud ($C_{s_i}^{local}$) involves the $Penalties$, in case of SLA violations, along with the price for running the service locally ($Price_{s_i}^{local}$). This refers to the instance hour price of the local cloud.

The prices for executing the service locally and in the public cloud may vary considerably according to the context. In [CHS10], the authors carried out a study to compare the price of their local cluster and a commercial cloud solution. They calculate the equivalent price of a node of their cluster and nodes of a public cloud. To this aim, they consider the performance of the nodes, the prices associated to that cluster and its usage. However, such a comparison requires benchmarking the local resources and the public cloud resources, and information on the local cloud (e.g. upfront cost and usage of the cloud), which are not available in our context. In [MH13], a similar approach was used and in one of the benchmarked clusters the specification of the nodes is similar to our specifications; therefore, we employ the results presented in [MH13] as the instance price of the local resources, that is \$ 0.25. Moreover, the nodes of the public cloud employed in this scenario are similar to a commercial offer of the Amazon provider[4]; consequently we adopt the price for this commercial offer in our scenario, that is \$ 0.40[5].

We summarise this scenario and its assumptions as follows:

- We assign a (symbolic) price for each service, which is calculated according to the local cloud instance hour price and which considers the requested resources, the estimated completion time and a random profit margin. This price is used in our experiments to generate a value for the penalty in case of SLA violation (a percentage of the price);
- We assume that the price for executing services locally is lower than the public's price. In our scenario, the instance hour price in the local cloud is \$ 0.25, while the public cloud price is \$ 0.40;
- The price for running the service in both the local and the public cloud are estimated according to the resources required by the service, its estimated completion time and the instance hour price in the target cloud;

[4]The similar type of VM is named by the provider as c3.2xlarge.

[5]Retrieved from `http://aws.amazon.com/ec2/pricing/` on 28th December 2014.

- For the sake of simplicity, we assume that the penalty to be paid in case of a service violation is covered by the agreement between the IMT and the public cloud, when a service is allocated in the public cloud. In this case the penalty foreseen for the SLA violation of a service executed in the public cloud covers also the penalty to be paid to the consumer who requested the execution of the service in the IMT cloud;

- Although SLAC supports the specification of penalties given in different conditions, for the sake of simplicity, we define a single penalty for the SLA that covers all possible violations.

With these premises, we carried out experiments to compare all scheduling algorithms developed in this chapter: Random, which runs all services locally; Dissimilarity, which also executes all services only in the local cloud; Risk-Aware, which considers only the risk of SLA violation to decide whether to execute the service locally or in the public cloud; and Cost-Aware, which considers the costs involved in executing the service locally and in the public cloud along with the risk of SLA violation. As the previous experiments, the services were generated synthetically. The penalty in case of violation of the SLA was generated with the service and defined within the range specified in each test. This range was a percentage of an estimated service price and was increased during the tests to understand the adaptability of the scheduling algorithms.

In these experiments, the arrival rate of the services ($\lambda$) was defined as 0.6 since it was the average case in the previous experiments. We executed the four algorithms five times for each penalty range (with 100 services) and compared the unexpected expenses of executing these services. Intuitively, these expenses represent all the extra costs to a provider, i.e. the sum of penalties of the violated services and/or the sum of the difference between the cost for executing each service in public cloud and of executing it in the private cloud.

More formally, the $UnexpectedExpense$ for running a set of services $S$ is the sum of the unexpected expenses of each $s_i \in S$ as defined in Equation 6.2. The unexpected expense of a service $s_i$ ($U_{s_i}$) is calculated

**Table 22:** Comparison of the Unexpected Costs with different penalties ranges.

| | | Random | Dissimilarity | Risk-Aware | Cost-Aware |
|---|---|---|---|---|---|
| **Penalties** | **0 - 40%** | $ 3.7 | $ 2.4 | $ 2.0 | $ 1.4 |
| | **20 - 60%** | $ 6.1 | $ 3.5 | $ 2.9 | $ 2.5 |
| | **40 - 80%** | $ 8.8 | $ 6.1 | $ 5.3 | $ 3.4 |
| | **60 - 100%** | $ 9.0 | $ 6.5 | $ 5.6 | $ 3.6 |

depending on the place of execution of the service (locally or in public cloud) and whether it was violated or not. Therefore, in Equation 6.3 we show all the four cases. $Penalty$, as the name suggests, is the amount to be paid in case of violation of $s_i$, $Price^{local}_{s_i}$ is the price for running the service locally and $C^{public}_{s_i}$ is the total cost for running $s_i$ in the public cloud. Notably, $C^{public}_{s_i}$ does not change if the service is violated. This is due to our assumption that the penalty for the service is paid by the public cloud.

$$UnexpectedExpense = \sum_{s_i \in S} U_{s_i} \tag{6.2}$$

$$U_{s_i} = \begin{cases} 0, & \text{if } s_i = \neg violated \wedge local \\ Penalty, & \text{if } s_i = violated \wedge local \\ C^{public}_{s_i} - Price^{local}_{s_i}, & \text{if } s_i = \neg violated \wedge public \\ C^{public}_{s_i} - Price^{local}_{s_i}, & \text{if } s_i = violated \wedge public \end{cases} \tag{6.3}$$

The results of the tests are presented in Table 22 and in Figure 35. The unexpected expenses represent a relevant cost for the provider and a reduction in the profit margin. In the case of the Cost-Aware scheduler, these unexpected expenses were reduced by up to 62% in comparison to the Random algorithm and up to 36% in comparison to the Risk-Aware scheduler.

The results show that, with our model, the Cost-Aware algorithm adapts to different penalties of the services and significantly reduces the total unexpected costs. Although the Risk-Aware algorithm shows

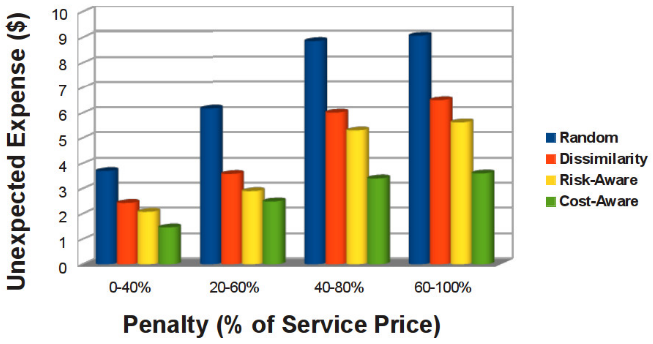

**Figure 35:** Unexpected expenses of the schedulers proposed in this thesis.

a satisfactory performance with penalties between 20-60% since these values are compatible with the maximum risk defined in the policies (50%), its performance is still 14% worse than the Cost-Aware algorithm. Overall, the Cost-Aware performed significantly better than Random, Dissimilarity and Risk-Aware algorithms in *all* tests.

## 6.4 Summary

In this chapter, we provided an overview of the role of each solution proposed in the thesis and on how they were integrated in the Polus framework, which was used to feed the autonomic managers with knowledge. In brief, the SLAC language is used to describe services; the SLAC Framework parses and evaluates SLAs; Panoptes feeds the autonomic manager with information and knowledge on the cloud and the services; and the Service Analyser, which uses the RF+PAM methodology, integrated in Panoptes, provides the similarity among services, which has multiple uses in autonomic clouds.

We also discussed the support of knowledge extraction components in Panoptes, which enable the integration of this system and knowledge generation solutions (related to the monitoring and not) and leverages the utility of these solutions. For instance, the RF+PAM methodology can be used also for finding the similarity or clustering from data not related to the monitoring and; therefore, included in Panoptes as a knowledge extraction component.

Finally, we presented a use case employing the IMT cloud and a public cloud adopted to complement the IMT cloud resources. In this hybrid cloud, we implemented service schedulers to decide where to allocate the services. The use of Panoptes in this scenario facilitated the development of autonomic tasks since the schedulers request the necessary knowledge to Panoptes and do not need to place agents in the system, implement knowledge generation algorithms, etc. It reduced considerably the complexity of developing such solutions, which enables the focus on the autonomic development. Moreover, the services were described using the SLAC language and the SLAC framework to evaluate the SLAs.

The principal objective of this use case was to demonstrate that the Polus framework was appropriate to a realistic service provisioning scenario. In this respect, Polus provides good support for the needs of the schedulers, which suggests that this framework should be appropriated for other scenarios in the autonomic cloud domain.

# Chapter 7

# Conclusions

Autonomic computing is a prominent paradigm to cope with the complexity of clouds. However, most proposals which explore the autonomic paradigm in clouds, assume that the knowledge for the decision-making is somehow available. In this thesis, we have discussed the knowledge generation process in the autonomic cloud domain and designed Polus, which is a theoretical and practical framework for the provision of this knowledge to autonomic managers. The results of the research are expected to directly contribute to: the development of autonomic clouds; the optimization of the knowledge discovery process from operational data; the adoption of SLA in the domain; and the employment of machine learning solutions in the cloud domain.

Below, we summarise the proposed solutions, reconsider the research questions posed in Chapter 1, examine the limitations of the study and discuss future research directions that emerged during this research activity.

## 7.1 Thesis Summary

The autonomic management in cloud computing is based on three main pillars: the definition of services, the monitoring of the system and the generation and exploitation of specific types of knowledge. In this thesis, we address these pillars as follows.

- In Chapter 3, we designed a language for the definition of SLAs, named SLAC, specifically devised for the cloud domain. This domain-specific language describes cloud services as well as their functional and non-functional requirements, while successfully addressing important aspects of the domain, such as brokerage, multiparty agreement, specification of multiple services, formalism and the vocabulary for clouds. Moreover, SLAC has been extended to incorporate significant business aspects of the domain, such as pricing models, billing and business actions. Finally, we designed and implemented a framework to parse and evaluate the SLAC SLAs. Clouds are managed taking into account such definitions (SLA) and the policies of the system; therefore, with SLAC, we provide knowledge about the objectives of the services that must be considered in the autonomic management;

- Chapter 4 defined and analysed the transformation of operational data into knowledge. Considering this process and the requirements of autonomic clouds, we devised an architecture to effectively monitor the infrastructure and the services provided to consumers. This architecture, termed Panoptes, focuses on the following monitoring properties: scalability, adaptability, resilience, timeliness and extensibility. It also implements mechanisms to transform data into knowledge using the filtering, aggregating and processing functions. The results indicate a considerable reduction in the amount of analysed data since Panoptes process data close to the source and logically divides the cloud according to its hierarchy;

- In autonomic clouds, the decision-making process is hindered by the obfuscation of several details of the provided services and of the infrastructure, which is caused by the characteristics of cloud, such as virtualization and dynamism. In Chapter 5, in order to assist autonomic managers in the decision-making, we devised a novel machine learning methodology to produce a flexible type of knowledge which can be used for different aims in the autonomic management of clouds. Intuitively, we produce a measure of similarity among services. This knowledge has a wide range of applications in the domain, provided directly to the autonomic managers or as the basis for other solutions which generate knowledge using the similarity notion, e.g. for the detection of anomalous behaviour or for application profiling. We validated this methodology through several experiments. These experiments demonstrate many benefits of our solution: superior performance, low memory footprint, support to mixed types of features, support to a large number of features and fast on-line prediction;

- Chapter 6 provided an overview of the role of the solutions proposed in the thesis in the autonomic cloud domain and their integration forming the Polus framework. It also proposed the support of knowledge extraction components in Panoptes, which enables the integration of this system and knowledge generation solutions. Finally, it presents a use case employing the IMT cloud and a public cloud adopted to complement the IMT cloud resources. In this hybrid cloud, we implemented service schedulers to decide where to allocate the services. With this use case, we demonstrate that Poluswas capable of providing the necessary knowledge to schedulers with different requirements, which suggests that it should be successfully used for other scenarios in the autonomic cloud domain.

## 7.2 Research Findings

In this section, we relate the findings of this thesis with the research questions posed in Chapter 1 and refer to the relevant sections for further details.

**Research Question 1**

*How to describe services and their objectives in the cloud domain?*

The definition of services in autonomic clouds guides the management of the cloud itself since they are service-oriented. As highlighted in the comparative analyses presented in Chapter 3, the existing SLA definition languages are not able to express many important aspects of cloud services. To address this question, we developed a SLA definition language, named SLAC, designed for the cloud domain. Moreover, we implemented the SLAC framework, which is able to evaluate and interpret SLAs in this language and provide the service description and the evaluation results to the autonomic managers.

**Research Question 2**

*What is data, information, knowledge and wisdom in the autonomic cloud domain?*

We address this question in Chapter 4. In particular, we defined data, information and knowledge, and discussed their specific meaning in autonomic clouds with a particular focus on the monitoring process. To this aim, we based our approach on the *Data, Information, Knowledge and Wisdom* (DIKW [Zel87]) architecture, specialised its concepts for the domain and discussed examples of data, information and knowledge. Finally, we used these concepts to design an architecture to transform operational data into knowledge and the components to generate knowledge from multiple sources.

**Research Question 3**

*How to collect and transform the enormous amount of operational data into useful knowledge without overloading the autonomic cloud?*

Autonomic clouds are dynamic, large-scale and elastic. Moreover, the resources used to provide services are loosely-coupled. To cope with these characteristics, we devised a monitoring solution, named Panoptes, which provides adaptive mechanisms to deal with different loads and to avoid interferences in the service provision. We implemented Panoptes and, with a number of experiments, we demonstrated the benefits of this architecture and its capacity to address this research question. This question was addressed in Chapter 4.

**Research Question 4**
*How to produce a robust measure of similarity for services in the domain and how can this knowledge be used?*

It is difficult to extract the similarity knowledge in the autonomic cloud domain due to its characteristics, such as the virtualization and security layers, its dynamism and the wide range of available services. Therefore, we propose a methodology, termed RF+PAM, to generate a robust measure of similarity. As a concrete example of its utility, we developed a scheduler which considers the similarity among services to allocate them. This scheduler reduces the SLA violations up to 48% and improves performance around 25%. Moreover, the concept of similarity has a wide range of applications in the domain and can be used to generate other types of knowledge. This question was addressed in Chapter 5.

**Research Question 5**
*How to integrate different sources of knowledge and feed the autonomic managers?*

This question was addressed in Chapter 6, where we discuss the role of the proposed solutions in autonomic clouds and the interaction between them in the Polusframework. To illustrate the integration of these solutions, we considered a use case where we developed schedulers with different knowledge requirements and used the proposed solutions to provide such knowledge. Therefore, we demonstrate that Polusis able to fulfil the needs of the schedulers, which suggests that it should also work in different scenarios and with different types of autonomic managers.

## 7.3 Limitations on the Study

In this section, we outline the limitations of the study:

- Overall, the aim of this thesis was to address the provision of knowledge to autonomic managers. Due to the broad scope of this task, we focused on the description and development of a framework to generate knowledge. Its architecture was designed to be modular, in order to facilitate reuse and extensions of the solutions to generate knowledge. Moreover, we implemented a methodology to discover the measure of similarity in autonomic clouds and integrated it in our framework. However, due to the wide range of types of knowledge necessary for the autonomic management, we focused on the methodology to facilitate the process and on a single type of knowledge, which provides a concrete example of the use of this methodology;

- The experiments suggest that Polus integrates and fulfils the needs of autonomic managers. However, it can be further clarified how intelligent autonomic managers need to be in order to dynamically request and change the configurations of our framework in an on-line fashion and, thus, take advantage of all the potential of Panoptes;

- The SLAC language was developed to be concrete and domain specific. In this thesis, we support the SLAC variants for IaaS and PaaS service delivery models. However, we did not include the SaaS model since it requires the development of linguistic abstractions to capture the wide range of applications available and their requirements;

- We implemented a proof-of-concept prototype of Panoptes and carried out experiments to test its benefits. We believe that it could be applied in large-scale cloud environments thanks to its architecture and adaptation mechanisms. However, further tests are required in large-scale clouds;

- Currently, the Panoptes knowledge extraction components are not optimal, e.g they do not take account of the place for processing data, which may not be the closest to the data source;

- The RF+PAM methodology was developed with the goal of separating the training and the prediction phases. Thus, the forest has to be retrained at some point. In this thesis, we proposed a threshold measure to trigger the retraining of the forest. However, we believe that the development of heuristics to recognise when to retrain the forest can improve the performance of the algorithm.

## 7.4 Future Works

As discussed in the previous section, several aspects of the domain are out of scope regarding our solutions. Some of these gaps imply open research challenges in the area. Therefore, further research efforts might consider the following:

- Although we take into account many aspects of negotiation in the SLAC language, we did not design negotiation protocols. The negotiation is an important phase of the SLA life cycle and must be fully considered in future works;

- Cloud computing is an evolving paradigm and the current definitions of SLAC might require extensions to cope with these changes. Therefore, we plan to define a mechanism to enable and regulate the definition of extensions.

- The cloud and the needs of providers and consumers change rapidly in autonomic clouds. Currently, the SLA provides only a static definition of the obligations of the parties involved; new mechanisms to support the dynamism of these environments are necessary. A potential solution for this need is the introduction of pre-defined changes in the SLA, i.e. of the conditions in which the valid terms can change. For instance, let us suppose that all consumers who adopt a new service will have, for the first 6 months of a 1 year contract, the half of the contracted maximum response time for the same price. In this situation, the SLA has a temporal constraint and has to change the response time after 6 months;

- The knowledge generation process consists of many steps and can have multiple sources. These sources and tasks can have parallel and sequential steps. Therefore, we plan to support workflows for data collection and knowledge generation, which enable the optimisation of the data retrieval and knowledge generation process;

- The RF+PAM methodology is flexible and has a wide range of applications. We plan to use this methodology as the base for other knowledge generation solutions and directly in an autonomic manager.

# Appendix A

# Overview of the Existing SLA Definition Languages

## A.1 WSOL

Web Service Offering Language (WSOL) [TPP02, TPP03, PPT03, Pat03, TPP$^+$05] is an XML based language and was developed to deal with the management gaps in the specification and standards of Web Services, such as Web Services Description Language (WSDL) and Business Process Execution Language for Web Services (WS-BPEL) [ACD$^+$03].

A SLA in the WSOL language refers to one or more WSDL files, which contain the service description; therefore, enabling the definition of management features and constraints for the service without modifying the WSDL files.

Essentially, SLA in the WSOL language are composed of five constructs:

- *Service Offering* (SO) is the central concept of the language, representing a variation of a service. In particular, it is the instantiation of a service, including pre-defined constraints and management statements. To illustrate the concept, let us suppose that an IaaS provider offers VMs to its consumers with different specifications. This provider could offer his service in two SO (class of service): the first, named Large_VM, with 16 GB of RAM and 99% of availability and; the second, named Small_VM, with the same specification but 97% of availability. Although the SO concept reduces the overhead and provides advantages in the management due to its limited number of pre-defined class of services, it hinders the customization of the service and restricts the negotiation possibilities;

- *Constraints* are divided into three categories: functional, non-functional (QoS) and access rights. Functional constraints define conditions that a correct operation invocation must satisfy. Non-functional constraints (QoS constraints) describe properties of the service, such as performance and reliability. To specify this category of constraints, WSOL requires the use of external ontologies for metrics, measurement unities and precise definitions of how the metrics are measured or calculated. Finally, access rights define when a consumer can invoke a particular operation;

- Three types of *Management Statement* schemas are specified in the WSOL language: (i) the definition of the payment models, which are: pay-per-use, in which the agreement set a price for the invocation of a single operation (pay for the quantity of invocations) and subscription-based, in which consumers pay to make use of a service for a period of time; (ii) monetary penalty, i.e. a monetary recompense in case of violation of the SLA; and (iii) management responsibility which specifies the party responsible for checking a particular constraint;

- Considering the use of multiple classes of services and their discrete variation (two SO can be the same apart from, e.g. the amount of availability of a VM), the WSOL defines *Reusability Elements*. This concept enables the creation of service templates, in which variables are defined. In the instantiation of a service, these variables are replaced with the actual values, which are sent as parameters;

- WSOL supports the specification of *Service Offering Dynamic Relationship*. Intuitively, it is the definition of alternative SO that can

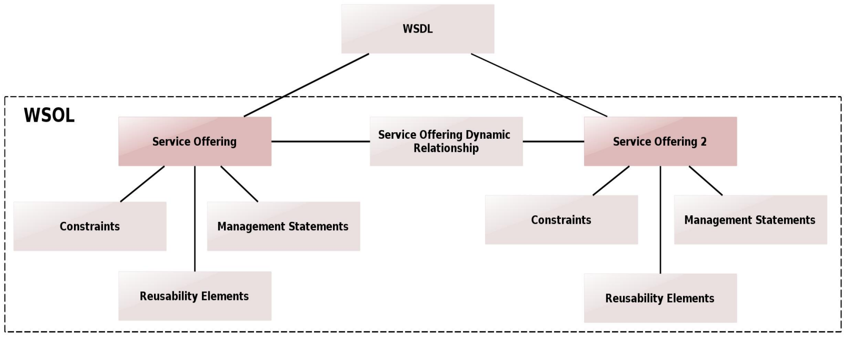

**Figure 36:** Structure of SLAs in the WSOL language.

replace deployed SO under some conditions. Such construct can be used, for instance, to replace an SO with a different SO in case of a violation of a constraint in the first.

Figure 36 depicts the relations amongst the constructs of the WSOL language. Service offerings refer to WSDL files, which are the description of the service. A service offer, in turn, comprises constraints, management statements and reusability elements. Finally, a service offering dynamic relationship refers to a deployed SO and an alternative SO.

Regarding the implementation of the language, a parser was developed in [PPT03] and a framework, named Web Service Offering Infrastructure (WSOI), was presented in [TMPE04].

In summary, the main features of WSOL language are: low overhead for negotiation and service instantiation, reuse mechanisms, support for third party monitoring and accounting, and support for the specification of SO relationships. However, the language has significant limitations. For instance, it does not support SLA customization, it is not equipped with a formal semantics, depends on the WSDL files, does not support dynamic environments and regulates only the quality of service. Thus, it does not fully capture the relationship between the parties, such as, the relationship between the carrier and the provider.

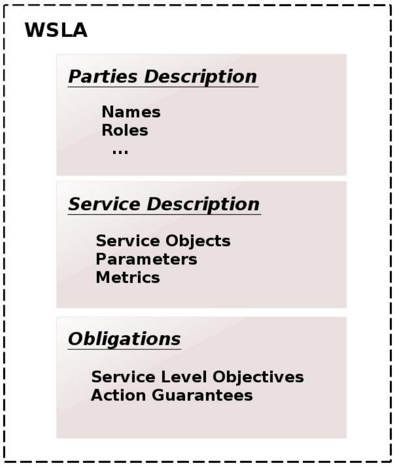

**Figure 37:** Structure of SLAs in the WSLA language.

## WSLA

Web Service Level Agreement (WSLA) [LKD+03, KL03] is also based on XML. However, in contrast to WSOL, a SLA in the WSLA language is self-contained, i.e. it does not refer to other files.

In this language, the SLA is divided into three main sections as shown in Figure 37. The first section, the *Parties Description*, contains the details of the signatory parties, including third parties.

The second section, the *Service Description*, describes the characteristics of the service (service objects, parameters and metrics). Service objects abstract SLA parameters and represent the operations of a service (e.g. instance VM). SLA parameters are properties of service objects. They are composed of a name, a type, a unit and metrics. Response time and availability are examples of these parameters. Finally, metrics refer to single or an aggregation of metrics. This aggregation can be done in two ways: using functions to define the formula of aggregation (operands); or a measurement directive, to define how a metric should be measured.

Figure 38 depicts the components of the Service Description and their relation. Service Objects have SLA parameters, which are defined by

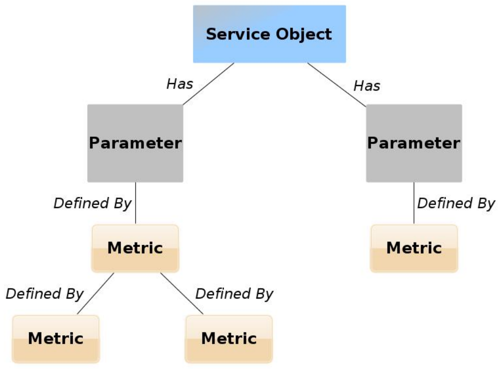

**Figure 38:** Relations of the components of the service description section in WSLA.

metrics. Finally, metrics can be composite, i.e. defined by other metrics.

The service-level-objectives (SLOs, i.e. specification of the performance of the service) and action guarantees are defined in the *obligations* section, the last section of the SLA. The SLOs represent a commitment to maintain a specific quality for the service in a period of time, constraints that may be imposed to parameters, e.g. response time $< 10$ ms. Guarantees, instead, define actions to be perform given a pre-condition [KL03].

The key features of WSLA are: its extensive documentation, the flexible metrics construct (for instance, it supports composite metrics) and its extensible. Moreover, the SLA is structured in a way that the monitoring clauses can be separated from contractual terms, which enables to involve third parties in the provision of the service (e.g. audition) without the disclosure sensible information.

The weaknesses of the approach are: it is coupled to the monitoring infrastructure (commercial solution), does not support pricing schemas, does not present formal semantics for the language [WB10], has few reusability features and is based on the XML-Schema, whose semantics are unsuitable for the constraint-oriented reasoning and optimisation [KTK10].

## A.2 WS-Agreement

WS-Agreement [ACDK04] was defined by the Open Grid Forum (OGF) and, along with WSLA, is the most well-known machine-readable SLA specification language [KTK10]. Besides the SLA, the OGF also defines a protocol for negotiation and establishment of SLA.

The structure of a SLA in WS-Agreement comprises an optional name for the agreement; the context, which describes the involved parties as well as its expiration time; and the terms which define the service and its guarantees. Figure 39 depicts this structure. Terms are composed using the *term compositor structure* construct, which provides logical operands to combine them. It enables the specification of alternative branches in the SLA, which is an important feature in the creation of offers and requests in the negotiation phase. The terms, in turn, are structured as follows:

- *Service Terms* describe the functionally delivered under the agreement. The content of this construct depends on the particular domain but is formed of *Service Description Terms*, which provide a functional description of the service. Service Description Terms are composed of: *Service References*, which define the endpoint for the service, i.e. its interface; and *Service Properties*, which express SLOs, such as response time and throughput;

- *Guarantee Terms* assure that the quality of the service is enforced or an action will be taken in case of violations. Its definition is divided into four elements:

  - *Service Scope* defines the service (or part) to which the guarantee applies;
  - The conditions under which the guarantee applies are defined by *Qualifying Conditions*;
  - *Service-Level-Objectives* express an assertion over the service attributes and external factors, such as the date;
  - *Business Value List* provides the evaluation of the guarantee in terms of rewards and penalties (in arbitrary value expressions, for instance in Euros).

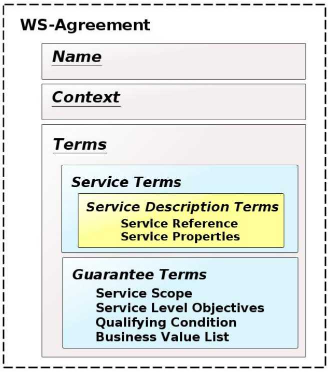

**Figure 39:** Structure of SLAs defined in the WS-Agreement language.

WS-Agreement supports third parties in the SLA (e.g., auditor or carrier) and has different implementations of the framework for monitoring and evaluation of SLA (e.g., Cremona Framework [LDK04]). Moreover, many extensions for the negotiation and management have been developed for the language.

Nevertheless, it provides only the high-level account of SLA content, thus leaving the fine-grained content unspecified. Although this approach provides flexibility to the language, it is also a source of ambiguity since the parties can have different definitions for the same term. Moreover, it requires the understanding of the ontology of the terms and constructs of the language, for example, of the metrics in the SLA, by all involved parties [LF06]. Finally, like WSLA, WS-Agreement is also coupled to XML-Schema, whose semantics are unsuitable for constraint-oriented reasoning and optimisation demands of operation research [KTK10].

## A.3 SLA*

The SLA* [KTK10] language is part of the SLA@SOI project [SLA14], which aims at providing predictability, dependability and automation in all phases of the SLA life cycle.

SLA* is inspired by WS-Agreement and WSLA and, in contrast to the

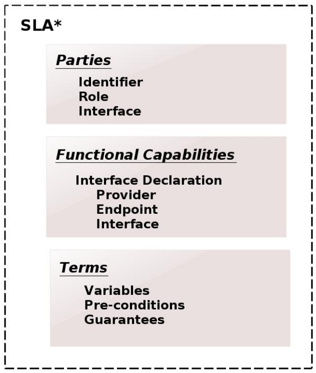

**Figure 40:** Structure of SLAs defined in the SLA* language.

described languages that support only web services, aims at supporting services in general, e.g. medical services. To achieve this aim, the authors specified an abstract constraint language which can be formally defined by plugging-in domain specific vocabulary.

Agreements in SLA* comprise: the involved parties, the definition of services in terms of functional interfaces and agreement terms. The agreement terms include: (i) variables which are either a "convenience" to be used in place of an expression (shorthand label) or a "customisable" which expresses "options" (e.g. *<4 and <10*); (ii) pre-conditions that define the cases in which the terms are effective (e.g. week days, business hours); and (iii) guarantees that describe states that a party is obliged to guarantee (for example, a SLO) or an action should be taken. This structure is depicted in Figure 40.

The benefits of the language are: it supports any kind of service; it is extensible; it is expressive; has a framework which covers all phases of the SLA life cycle and; was tested in different domains. Nevertheless, the SLA* specification lacks precise semantics due to its multi-domain approach and the support to brokerage. Moreover, it requires the development of specific vocabulary for each domain.

## A.4 SLAng

The first version of SLAng is presented in [LSE03]. However, in [Ske07], Skene, one of the authors of the original paper, claims that this language was highly imprecise and open to interpretation. Hence, Skene decided to continue the development of SLAng to addresses these issues. Therefore, in this work, we review the SLAng developed by Skene, i.e. the improved version of the language (we refer to his doctoral thesis [Ske07] for the full specification of the language).

SLang specification is presented as a combination of an EMOF [Obj04b] structure, OCL [Obj03] constraints and natural language commentary. A SLA defined in SLAng is the instantiation of the EMOF abstract model, which can be concretely instantiate in several ways, for example, using Human-Usable Textual Notation (HUTN) [Obj04a] or XML Metadata Interchange (XMI) [Obj14] (it can also include comments in natural language to facilitate the understanding). The OCL constraints are used to refine the model and define, to some degree, the semantics of the SLA.

To illustrate the OCL use in SLAng, Listing A.1 presents the specification of the total down time for an operation of a service. The constraint selects and sums all non-scheduled events in which an operation failed or which the latency is higher than the specified maximum latency.

**Listing A.1:** Extract of a SLA specified in OCL.

```
--Total downtime observed for the operation
let totalDowntime(o : Operation) : double
o.serviceUsage -> select(u (u.failed or u.duration >
maximumLatency) AND schedule -> exists(s |
s.applies(u.date)) ) -> collect(u | downtime(u.date,
o)) -> iterate( p : double, sumP : double | sumP + p)
```

As depicted in Figure 41, the EMOF model consists of:

- *Administration Clauses*, which define the responsibilities of parties in the SLA administration. This administration sets constraints to define how the SLA is administrate and which party is in charge

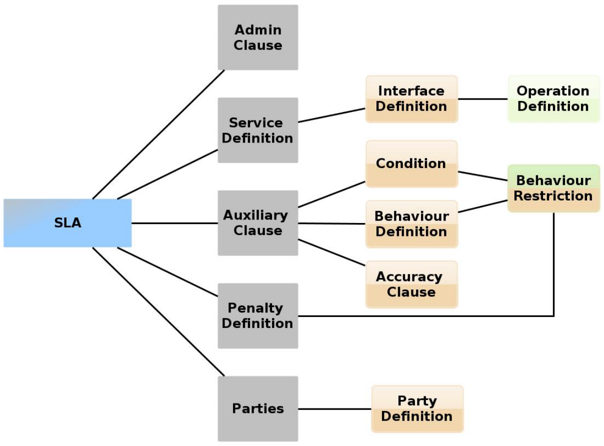

**Figure 41:** Structure of SLAs defined in the SLAng language.

of this administration, for example, they express who can submit evidences of SLA violations;

- Service's *Interface Definitions*, including the operations available for this service;
- *Auxiliary Clauses*, which are abstract constructs composed of: *Conditions* to associate the behaviour of the service to a constraint; *Behaviour Definition*; and *Accuracy Clause* which establishes the rules to assess the accuracy of service measurements. Then, these measurements are employed to verify violations and apply penalties. Also, conditions, behaviours, penalties and parties are used to create constraints on the service behaviour (e.g. availability), named *Behaviour Restrictions*;
- *Penalty Definitions*, which defines the actions that should be enforced in case of violation;
- *Parties Description*, which describe the involved parties.

In contrast to the previous languages, SLAng is domain-specific, devised for Application-Service Provision (ASP). Its main strengths are: low ambiguity due to the correspondence between elements in an abstract service model and events in real world [LF06]; emphasis on compatibility, monitorability and constrained service behaviour; and domain-specific vocabulary for IT Services.

The main limitation of SLAng is the complexity to: fully understand its specification; create SLAs using this specification; and extend the language. Its limitation is due to: the combination of techniques as OCL and EMOF, which require technical expertise to use [Ske07]; its expressibility; and its formal nature. Moreover, considering the heterogeneity of the IT services domain, the language requires an extensive analysis effort by experts and the definition of extensions of similar size to SLAng core language itself [Ske07] to be deployed in real-world cases. These efforts and complexity lead not only to difficulties to users but also to high costs for its adoption.

## A.5 CSLA

Cloud Service Level Agreement[1] [Kou13, SBK+13] is a specification language devised for the cloud domain.

Its structure is similar to WS-Agreement and is presented in Figure 42. Validity describes the initial and expiration dates for the SLA. The parties are defined in the Parties Section of the agreement while the template is used to define the service, the associated constraints, the guarantees related to these constraints, the billing scheme and the termination conditions.

A novelty of the language is, in addition to the traditional fixed price billing model, the possibility to use the pay-as-you-go model. Moreover, CSLA introduces the concept of fuzziness and confidence. The former establishes an error margin for a metric in the agreement. The latter defines the minimum ratio of the enforcements that the metric values do not exceed the threshold, permitting the remaining measures to exceed the threshold but not the fuzziness threshold. For example, the threshold for the response time of a service is 3 seconds, the fuzziness value is 0.5

[1]The language was presented in a short paper [KL12] but it is not available on-line.

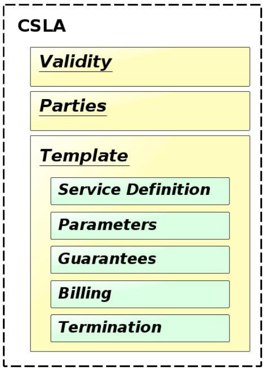

**Figure 42:** Structure of SLAs defined in the CSLA language.

and the confidence is 90%. In every 100 requests, minimum 90 need to have values between 0 and 3 and maximum 10 can be between 3 and 3.5 without violating the SLA.

As drawbacks, the language is neither formally defined, nor supports parties with important roles (e.g. the broker), nor comprehends other dynamic aspects of the cloud.

## A.6 Overview of the Languages

In the present subsection, we describe the machine-readable solutions to define SLAs for services. Apart from SLA*, which enables the specification of SLA for electronic services, all analysed languages target web services or a specific subgroup of this area.

WSOL, WSLA and WS-Agreement are compatible with the main standards but unsuitable for the constraint-oriented reasoning optimisation demands for the decision making [KTK10]. Moreover, the structure of agreements written in most languages is similar and rely on XML. The exceptions are SLA* which provides a higher abstraction of service but also has its implementation using XML, and SLAng, which has a completely different approach, as it is domain-specific and uses EMOF and OCL.

Despite the similarities, all specification languages have unique characteristics. For example, WSOL is less expressive than WSLA, WS-Agreement, SLAng and SLA* but provides the concept of service offerings. Consequently, it is easy to implement in different scenarios and provides low overhead in the negotiation and deployment processes. However, WSOL is neither flexible, nor fully equivalent to the other SLAs definition languages, as it capture only the QoS aspects of the agreement.

Major challenges for the adoption of abstract languages for SLA specification (e.g. SLA*) are the creation of the vocabulary for a domain (e.g. metrics) [LF06], and to assure that all parties share and understand the definitions in that vocabulary.

SLAng and CSLA are domain-specific and this problem has a lower impact. Nevertheless, SLAng is rather complex and requires experts to understand the specification and adapt it to each use case. CSLA, instead, provides neither the formalism for the SLA specification, nor captures important characteristics of the domain, e.g. broker support.

In light of these considerations and of the cloud requirements (investigated in Chapter 3), we propose a SLA specification language for cloud computing in Chapter 3.

# References

[ABdDP13] Giuseppe Aceto, Alessio Botta, Walter de Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9), June 2013. 38, 87

[ABYST13] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. Deconstructing Amazon EC2 Spot Instance Pricing. *ACM Transactions on Economics and Computation*, 1(3):1–20, September 2013. 66

[ACD+03] T. Andrews, F. Curbera, H. Dholakik, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services Version 1.1. Technical report, IBM, 2003. 163

[ACDK04] A Andrieux, K Czajkowski, A Dan, and K Keahey. Web services agreement specification (WS-Agreement). *Global Grid Forum*, 2004. 168

[AD07] Amir Ahmad and Lipika Dey. A k-mean clustering algorithm for mixed numeric and categorical data. *Data & Knowledge Engineering*, 63(2):503–527, November 2007. 112

[AHT+03] Elena Allen, Steve Horvath, Frances Tong, Peter Kraft, Elizabeth Spiteri, Arthur D Riggs, and York Marahrens. High concentrations of long interspersed nuclear element sequence distinguish monoallelically expressed genes. In *Proc. of the National Academy of Sciences of the United States of America*, pages 9940–5, August 2003. 115

[AHWY03] Charu Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *Proc. of the 29th VLDB*, pages 81–92, 2003. 112

[AHWY04] Charu Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for projected clustering of high dimensional data streams. In *Proc. of the 30th VLDB*, volume 30, pages 852–863, 2004. 42, 112

[Ama15] Amazon. Amazon Web Services. http://aws.amazon.com/, 2015. 65

[Ant10] Gabriel Antoniu. Autonomic Cloud Storage: Challenges at Stake. *Proc. of the International Conference on Complex Intelligent and Software Intensive Systems*, pages 481–481, 2010. 88

[Apa15] Apache. Apache CloudStack. http://cloudstack.apache.org/, 2015. 144

[ASMC07] Hanady Abdulsalam, David B. Skillicorn, Patrick Martin, and O N Canada. Streaming random forests. In *Proc. of the 11th IDEAS*, pages 643–651, 2007. 115

[BA03] Leo Breiman and Adele Cutler. Random forests Manual V4, 2003. 113, 115

[BAGS02] Rajkumar Buyya, David Abramson, Jonathan Giddy, and Heinz Stockinger. Economic models for resource management and scheduling in Grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1507–1542, November 2002. 66

[BCL$^+$04] M. J. Buco, R. N. Chang, L. Z. Luan, C. Ward, J. L. Wolf, and P. S. Yu. Utility computing SLA management based upon business objectives. *IBM Systems Journal*, 43(1):159–178, 2004. 27

[BCL12] Rajkumar Buyya, RN Calheiros, and Xiaorong Li. Autonomic cloud computing: Open challenges and architectural elements. In *Proc. of 3rd International Conference on Emerging Applications of Information Technology (EAIT)*, pages 3–10, 2012. 26

[BKC00] Paul E. Bierly III, Eric H. Kessler, and Edward W Christensen. Organizational learning, knowledge and wisdom. *Journal of organizational change management*, 13(6):595–618, 2000. 41

[Bon15] Bonfire. Bonfire Project. http://www.bonfire-projectt.eu/, 2015. 37

[Bra09] Ivona Brandic. Towards Self-Manageable Cloud Services. *2009 33rd Annual IEEE International Computer Software and Applications Conference*, pages 128–133, 2009. 27

[Bre01] L Breiman. Random forests. *Machine learning*, 45:5–32, 2001. 107, 113

[CBQF10] Jong Youl Choi, Seung-Hee Bae, Xiaohong Qiu, and Geoffrey Fox. High Performance Dimension Reduction and Visualization for Large High-Dimensional Data Analysis. In *Proc. of the 10th IEEE/ACM CCGrid*, pages 331–340. IEEE, 2010. 110

[CC94] TF Cox and MAA Cox. *Multidimensional scaling*. Chapman & Hall, London;UK, 1994. 117

[CFR13] Alfredo Cuzzocrea, Giancarlo Fortino, and Omer Rana. Managing Data and Processes in Cloud-Enabled Large-Scale Sensor Networks: State-of-the-Art and Future Research Directions. In *Proc. of the 13th IEEE/ACM CCGrid*, pages 583–588. IEEE, May 2013. 109

[CG10] Yanpei Chen and AS Ganapathi. Analysis and lessons from a publicly available google cluster trace. *ECS Department, University of California, Berkeley, Tech Rep.*, 2010. 122, 123, 124

[CGCT10] Stuart Clayman, Alex Galis, Clovis Chapman, and Giovanni Toffetti. Monitoring Service Clouds in the Future Internet. In *Future Internet Assembly*, pages 115–126, 2010. 87

[CGM10] Stuart Clayman, Alex Galis, and Lefteris Mamatas. Monitoring virtual networks with Lattice. In *Proc. of the 12nd IEEE/IFIP NOMS Wksps*, pages 239–246. IEEE, 2010. 39

[CHS10] Adam G. Carlyle, Stephen L. Harrell, and Preston M. Smith. Cost-Effective HPC: The Community or the Cloud? In *Proc. of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 169–176. Ieee, November 2010. 150

[Clo15] CloudTM. CloudTM Project. http://www.cloudtm.eu, 2015. 37

[CNM06] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proc. of the 23rd ICML*, pages 161–168, New York; USA, 2006. ACM Press. 113

[CUW11] Shirlei Aparecida Chaves, Rafael Brundo Uriarte, and Carlos Becker Westphall. Towards an Architecture for Monitoring Private Clouds. *IEEE Communications Magazine*, 49(December):130–137, 2011. 86, 89

[CWW11] Shirlei Aparecida De Chaves, Carlos Becker Westphall, and Carla Becker Westphall. Customer Security Concerns in Cloud Computing. In *Proc. of the 10th International Conference on Networks*, pages 7–11, 2011. 14

[CZ12] Deyan Chen and Hong Zhao. Data Security and Privacy Protection Issues in Cloud Computing. In *Proc. of the 1st International Conference on Computer Science and Electronics Engineering*, pages 647–651. IEEE, March 2012. 16

[DCW$^{+}$11] Anh Vu Do, Junliang Chen, Chen Wang, Young Choon Lee, Albert Y. Zomaya, and Bing Bing Zhou. Profiling Applications for Virtual Machine Placement in Clouds. *Proc. of the 4th IEEE Cloud*, pages 660–667, July 2011. 130

[DDF06] Simon Dobson, Spyros Denazis, and A Fernández. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2):223–259, 2006. 2

[DP98] TH Davenport and Lawrence Prusak. *Working Knowledge : How Organizations Manage What They Know*. Harvard Business School Press, 1998. 42

[DWC10] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud Computing: Issues and Challenges. *Proc. of the 24th IEEE AINA*, pages 27–33, 2010. 45, 78

[EBMD10] Vincent C. Emeakaroha, Ivona Brandic, Michael Maurer, and Schahram Dustdar. Low level Metrics to High level SLAs - LoM2HiS framework: Bridging the gap between monitored metrics and SLA parameters in cloud environments. In *Proc. of the 8th HPCS*, pages 48–54. IEEE, June 2010. 87, 88, 95

[EL09] Erik Elmroth and Lars Larsson. Interfaces for Placement, Migration, and Monitoring of Virtual Machines in Federated Clouds. In *Proc. of the 8th GCC*, pages 253–260. Ieee, August 2009. 18

[FHAJ$^{+}$01] F. Fabret, Hans-Arno, Jacobsen, F. Llirbat, J. Pereira, and K. Ross. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proc. of the 41st ACM SIGMOD*, volume 30, pages 115–126, June 2001. 91

[Fle15] Mike Fletcher. Simple Parser. http://simpleparse.sourceforge.net/, 2015. 75

[FZRL08] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *Proc. of the 4th Grid Computing Environments Workshop*, pages 1–10. IEEE, 2008. 109

[Gan14] Ganglia. Ganglia. http://ganglia.sourceforge.net/, 2014. 88

[Gar99] S Garfinkel. *Architects of the information society: 35 years of the Laboratory for Computer Science at MIT*. Mit Press, 1999. 14

[Gen13] Frank Gens. IDC Predictions 2014: Battles for Dominance and Survival on the 3rd Platform. Technical report, International Data Corporation, 2013. 1

[GMR04] J Gama, Pedro Medas, and R Rocha. Forest trees for on-line data. In *Proceedings of the 19th ACM Symposium on Applied Computing.*, pages 632–636, 2004. 42

[GRS00] S Guha, R Rastogi, and K Shim. ROCK: A robust clustering algorithm for categorical attributes. *Information systems*, 25(5):345–366, 2000. 42, 112

[GVB11] Saurabh Kumar Garg, Steve Versteeg, and Rajkumar Buyya. SMI-Cloud: A Framework for Comparing and Ranking Cloud Services. In *Proc. of the 4th IEEE UCC*, pages 210–218. IEEE, December 2011. 80

[GVK12] ER Gomes, QB Vo, and Ryszard Kowalczyk. Pure exchange markets for resource sharing in federated clouds. *Concurrency and Computation: Practice and Experience*, pages 977–991, 2012. 18

[GZK05] MM Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: a review. *ACM Sigmod Record*, 34(2):18–26, 2005. 112

[HA85] Lawrence Hubert and P Arabie. Comparing partitions. *Journal of classification*, 218:193–218, 1985. 125

[Has08] Osman Hassab Elgawi. Online random forests based on CorrFS and CorrBE. In *Proc. of IEEE Computer Vision and Pattern Recognition Workshops*, pages 1–7. Ieee, June 2008. 115

[HD10] Peer Hasselmeyer and Nico D'Heureuse. Towards holistic multi-tenant monitoring for virtual data centers. In *IEEE/IFIP Network Operations and Management Symposium Workshops*, pages 350–356. Ieee, 2010. 40

[HH07] Osman Hassab Elgawi and Osamu Hasegawa. Online incremental random forests. In *Proc. of the 1st International Conference on Machine Vision*, pages 102–106. Ieee, December 2007. 115

[HKK05] Julia Handl, Joshua Knowles, and Douglas B Kell. Computational cluster validation in post-genomic data analysis. *Bioinformatics (Oxford, England)*, 21(15):3201–12, August 2005. 123

[HM08] Markus Huebscher and Julie Mccann. A survey of autonomic computing-degrees, models, and applications. *ACM Computing Surveys*, 40(3):1–28, 2008. 85

[Hor01] P Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology, 2001. 1, 22

[IBM05] IBM White Paper. An architectural blueprint for autonomic computing. *Quality*, 36(June):34, 2005. 2, 22

[KAC12] Kenichi Kourai, Takeshi Azumi, and Shigeru Chiba. A Self-Protection Mechanism against Stepping-Stone Attacks for IaaS Clouds. In *11*, pages 539–546. IEEE, September 2012. 26

[KK07] P Karaenke and Stefan Kirn. Service level agreements: An evaluation from a business application perspective. In *Proc. of 5th eChallenges*, 2007. 64

[KL03] A Keller and H Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 2003. 29, 30, 166, 167

[KL12] Yousri Kouki and Thomas Ledoux. CSLA: a Language for improving Cloud SLA Management. In *Proc. of the 2nd International Conference on Cloud Computing and Services Science*, 2012. 35, 173

[Kou13] Yousri Kouki. *Approche dirigée par les contrats de niveaux de service pour la gestion de lélasticité du "nuage"*. PhD thesis, L'Université Nantes Angers Le Mans, 2013. 35, 173

[KR90] Leonard Kaufman and PJ Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. John Wiley and Sons, New York, NY, wiley seri edition, 1990. 114, 116

[KRJ11] Hyunjoo Kim, Ivan Rodero, and Shantenu Jha. Autonomic Management of Application Workflows on Hybrid Computing Infrastructure. *Scientific Programming*, 19:1–23, 2011. 26

[KST$^+$10] Mahendra Kutare, Karsten Schwan, Vanish Talwar, Greg Eisenhauer, Matthew Wolf, and Chengwei Wang. Monalytics: Online Monitoring and Analytics for Managing Large Scale Data Centers. In *Proc. of the 7th ICAC*, pages 141–150. ACM Press, 2010. 87

[KTK10] K.T. Kearney, F. Torelli, and C. Kotsokalis. SLA*: An abstract syntax for Service Level Agreements. In *Proc. of the 11th IEEE/ACM International Conference on Grid Computing*, 2010. 31, 167, 168, 169, 174

[KYTA12] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *Proc. of the 14th IEEE/IFIP NOMS*. IEEE, 2012. 131

[LC08] Bin Lu and Juan Chen. Grid resource scheduling based on fuzzy similarity measures. In *Proc. of the 2nd IEEE Cybernetics and Intelligent Systems*, pages 940–944, 2008. 26, 130

[LDK04] H Ludwig, A Dan, and R Kearney. Cremona: an architecture and library for creation and monitoring of WS-agreents. In *Proc. of the 2nd international conference on Service Oriented Computing*, 2004. 169

[LF06] André Ludwig and Bogdan Franczyk. SLA Lifecycle Management in Services Grid-Requirements and Current Efforts Analysis. In *Proc. of the 3rd Grid Service Engineering and Management*, pages 219–233, 2006. 35, 36, 169, 173, 175

[LKD$^+$03] H Ludwig, A Keller, A Dan, RP King, and R Franck. Web service level agreement (WSLA) language specification. *IBM Corporation*, 2003. 166

[Llo82] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, March 1982. 114

[LRT14] Balaji Lakshminarayanan, DM Roy, and YW Teh. Mondrian forests: Efficient online random forests. *ArXiv e-prints. Preprint arXiv: 1406.2673*, 2014. 115

[LSE03] DD Davide Lamanna, James Skene, and Wolfgang Emmerich. SLAng: A language for service level agreements. In *Proc. of the 9th IEEE Workshop on Future Trends in Distributed Computing Systems*, volume 34069, 2003. 33, 171

[LTM$^+$11] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badge, and Dawn Leaf. NIST cloud computing reference architecture. Technical report, National Institute of Standards and Technology, 2011. 20, 21

[MB08] Leonardo De Moura and Nikolaj Bjorner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proc. of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, 2008. 59, 75

[MBEB11] Michael Maurer, I. Breskovic, V. C. Emeakaroha, and I. Brandic. Revealing the MAPE loop for the autonomic management of cloud infrastructures. In *Proc. of the 6th IEEE Symposium on Computers and Communications*, pages 147–152, 2011. 27

[MG09] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical Report 6, National Institute of Standards and Technology (NIST), 2009. 14, 18, 69

[MH13] A Marathe and Rachel Harris. A comparative study of high-performance computing on the cloud. In *Proc. of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, pages 239–250, 2013. 150

[MHCD10] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. Towards characterizing cloud backend workloads. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):34, March 2010. 122, 124

[MWZ+13] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, and Hua Cai. Toward Fine-Grained, Unsupervised, Scalable Performance Diagnosis for Production Cloud Computing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1245–1255, June 2013. 26, 131

[NDM09] Hien Nguyen Van, Frederic Dang Tran, and Jean Marc Menaud. Autonomic virtual resource management for service hosting platforms. In *Proc. of the 1st Workshop on Software Engineering Challenges of Cloud Computing*. Ieee, 2009. 1, 26

[NMRV11] Radheshyam Nanduri, Nitesh Maheshwari, A. Reddyraja, and Vasudeva Varma. Job Aware Scheduling Algorithm for MapReduce Framework. In *Proc. of the 3rd IEEE CloudCom*, pages 724–729. Ieee, November 2011. 129, 130, 142

[NN07] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer: An Appetizer*. Springer, 2007. 58

[Obj03] Object Management Group (OMG). UML 2.0 Object Constraint Language (OCL) Final Adopted specification, 2003. 33, 171

[Obj04a] Object Management Group (OMG). Human-Usable Textual Notation (HUTN) Specification, 2004. 33, 171

[Obj04b] Object Management Group (OMG). MOF 2.0 Core Final Adopted Specification, 2004. 33, 171

[Obj14] Object Management Group (OMG). XML Metadata Interchange (XMI) Specification, 2014. 33, 171

[Ope14] OpenNebula. OpenNebula. http://opennebula.org, 2014. 10, 97

[Pat03] K Patel. *XML Grammar and Parser for the Web Service Offerings Language*. PhD thesis, Carleton University, 2003. 163

[PB10] Siani Pearson and Azzedine Benameur. Privacy, Security and Trust Issues Arising from Cloud Computing. In *Proc. of the 2nd IEEE Cloudcom*, pages 693–702. IEEE, November 2010. 108

[PH05] Manish Parashar and Salim Hariri. Autonomic computing: An overview. In *Unconventional Programming Paradigms*, pages 247–259. Springer Berlin Heidelberg, 2005. 23, 26, 108

[PPT03] K Patel, B Pagurek, and V Tosic. Improvements in WSOL grammar and premier WSOL parser. Technical report, Carleton University, 2003. 163, 165

[PSG06] Adrian Paschke and E Schnappinger-Gerull. A Categorization Scheme for SLA Metrics. *Service Oriented Electronic Commerce 80*, pages 25–40, 2006. 27

[QKP$^+$09] Andres Quiroz, Hyunjoo Kim, Manish Parashar, Nathan Gnanasambandam, and Naveen Sharma. Towards autonomic workload provisioning for enterprise Grids and clouds. In *Proc. of the 10th IEEE/ACM International Conference on Grid Computing*, pages 50–57. IEEE, October 2009. 26, 130

[QPGS12] Andres Quiroz, Manish Parashar, Nathan Gnanasambandam, and Naveen Sharma. Design and evaluation of decentralized online clustering. *ACM Transactions on Autonomous and Adaptive Systems*, 7(3):1–31, September 2012. 112

[RA01] S Ron and P Aliko. Service level agreements. *Internet Next Generation project (1999-2001)*, 2001. 28

[RBL$^+$09] B Rochwerger, David Breitgand, Eliezer Levy, Alex Galis, Kenneth Nagin, Ignacio M. Llorente, and Rubén Montero. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):1–17, 2009. 18

[RCL09] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A Taxonomy and Survey of Cloud Computing Systems. In *Proc. of the 5th International Joint Conference on INC IMS and IDC*, NCM '09, pages 44–51. Ieee, 2009. 16

[RCV11] Massimiliano Rak, Antonio Cuomo, and Umberto Villano. CHASE: An Autonomic Service Engine for Cloud Environments. In *Proc. of* *the 20th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 116–121. Ieee, June 2011. 26

[Red14] Redis. Redis Database. http://redis.io, 2014. 72, 97

[RWH11] Charles Reiss, John Wilkes, and Joseph L Hellerstein. {Google} cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, USA, November 2011. 119

[SBK+13] Damian Serrano, Sara Bouchenak, Yousri Kouki, Thomas Ledoux, Jonathan Lejeune, Julien Sopena, Luciana Arantes, and Pierre Sens. Towards QoS-Oriented SLA Guarantees for Online Cloud Services. In *Proc. of the 13th IEEE/ACM CCGrid*, pages 50–57. IEEE, May 2013. 35, 173

[Sch11] Robert P Schumaker. From Data to Wisdom : The Progression of Computational Learning in Text Mining The DIKW Framework. *Communications of the International Information Management Association*, 11(1):39–48, 2011. 41, 84

[SH06] Tao Shi and Steve Horvath. Unsupervised Learning With Random Forest Predictors. *Journal of Computational and Graphical Statistics*, 15(1):118–138, March 2006. 115

[SI07] Bogdan Solomon and Dan Ionescu. Towards a real-time reference architecture for autonomic systems. In *Proc. of the 2nd International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2007. 27

[SILI10] Bogdan Solomon, Dan Ionescu, Marin Litoiu, and Gabriel Iszlai. Designing autonomic management systems for cloud computing. In *Proc. of the International Joint Conference on Computational Cybernetics and Technical Informatics*, pages 631–636. IEEE, 2010. 26, 88

[SILM07] Bogdan Solomon, Dan Ionescu, Marin Litoiu, and Mircea Mihaescu. A real-time adaptive control of autonomic computing environments. In *Proc. of the 17th International Conference on Computer Science and Software Engineering*, pages 1–13, 2007. 23

[Ske07] James Skene. *Language support for service-level agreements for application-service provision*. Doctor of philosophy, University of London, 2007. 33, 35, 51, 58, 171, 173

[SLA14] SLA@SOI. SLA@SOI Project. http://sla-at-soi.eu/, 2014. 37, 169

[SLS+09] Amir Saffari, Christian Leistner, Jakob Santner, Martin Godec, and Horst Bischof. On-line Random Forests. In *Proc. of 12th IEEE ICCV*, pages 1393–1400. IEEE, September 2009. 42, 115

[SRJ10] Kunfang Song, Shufen Ruan, and Minghua Jiang. A Flexible Grid Task Scheduling Algorithm Based on QoS Similarity. *Journal of Convergence Information Technology*, 5(7):161–166, September 2010. 26, 130

[SSB+05] Tao Shi, David Seligson, Arie S Belldegrun, Aarno Palotie, and Steve Horvath. Tumor classification by tissue microarray profiling: random forest clustering applied to renal cell carcinoma. *Modern pathology : an official journal of the United States and Canadian Academy of Pathology, Inc*, 18(4):547–57, April 2005. 115

[SSPC14] Hongyang Sun, Patricia Stolf, Jean-Marc Pierson, and Georges Da Costa. Multi-objective Scheduling for Heterogeneous Server Systems with Machine Placement. In *Proc. of the 14th IEEE/ACM CCGrid*, pages 334–343. Ieee, May 2014. 26, 130

[Ste02] Roy Sterritt. Towards Autonomic Computing: Effective Event Management. In *Proc. of the 27th Software Engineering Workshop*, 2002. 3

[Sur15] SurfNet. The Distributed ASCI Supercomputer 2. http://www.cs.vu.nl/das2/, 2015. 120

[TMPE04] V Tosic, W Ma, B. Poprrek., and B. Esfnndiari. Web Service Offerings Infrastructure (WSOI) - A management infrastructure for XML Web services. In *Proc of the 4th NOMS*, volume 1, pages 817–830, 2004. 165

[TNL10] Stefan Tai, Jens Nimis, and Alexander Lenk. Cloud Service Engineering Categories and Subject Descriptors. In *Proc. of the 32nd ACM/IEEE ICSE*, pages 475–476, 2010. 18

[TPP02] V Tosic, B Pagurek, and K Patel. WSOL - A language for the formal specification of various constraints and classes of service for web services. In *Proc. of the 9th International Conference On Web Services*, volume 3, pages 375–381, 2002. 163

[TPP03] V Tosic, K Patel, and B Pagurek. Reusability Constructs in the Web Service Offerings Language. In *Web Services, E-Business, and the Semantic Web*, pages 105–119. Springer Berlin Heidelberg, 2003. 163

[TPP+05] V Tosic, B Pagurek, K Patel, B Esfandiari, and W Ma. Management applications of the web service offerings language (WSOL). *Information Systems*, 2005. 163

[UTD14] Rafael Brundo Uriarte, Francesco Tiezzi, and Rocco De Nicola. SLAC: A Formal Service-Level-Agreement Language for Cloud Computing. In *Proc. of the 7th IEEE/ACM UCC (In Press)*, 2014. 9, 46

[UTN14] Rafael Brundo Uriarte, Francesco Tiezzi, and Rocco De Nicola. Definition of the Metrics and Elements of the SLAC Language. Technical report, IMT Institute for Advanced Studies Lucca, Lucca, Italy, 2014. 55, 68

[UTT15] Rafael Brundo Uriarte, Sotirios Tsaftaris, and Francesco Tiezzi. Service Clustering for Autonomic Clouds Using Random Forest. In *Proc. of the 15th IEEE/ACM CCGrid [Accepted with Shepherd]*, 2015. 9

[UW14] Rafael Brundo Uriarte and Carlos Becker Westphall. Panoptes: A monitoring architecture and framework for supporting autonomic Clouds. In *Proc. of the 16th IEEE/IFIP NOMS*, Krakow, Poland, 2014. IEEE. 9, 88

[VHKA10] A. Viratanapanu, A. K. A. Hamid, Y. Kawahara, and T Asami. On demand fine grain resource monitoring system for server consolidation. *Kaleidoscope: Beyond the Internet?-Innovations for Future Networks and Services*, 2010. 92

[WB10] Linlin Wu and Rajkumar Buyya. Service Level Agreement (SLA) in Utility Computing Systems. Technical report, The University of Melbourne, 2010. 28, 30, 31, 46, 167

[WBYT11] Phili Wieder, Joe M. Butler, Ramin Yahyapour, and Wolfgang Theilmann. *Service Level Agreements for Cloud Computing*. Springer New York, New York, NY, 2011. 30

[WL12] Aurélien Wailly and Marc Lacoste. VESPA : Multi-Layered Self-Protection for Cloud Resources. In *Proc. of the 9th ACM ICAC*, pages 155–159. ACM, 2012. 26

[Wri29] Frances Wright. *Course of Popular Lectures*. Office of the Free Enquirer, 1829. 41

[WST$^+$11] Chengwei Wang, Karsten Schwan, Vanish Talwar, Greg Eisenhauer, Liting Hu, and Matthew Wolf. A flexible architecture integrating monitoring and analytics for managing large-scale data centers. *Proc. of the 8th ACM international conference on Autonomic computing*, page 141, 2011. 87

[WSVY07] Timothy Wood, PJ Shenoy, Arun Venkataramani, and MS Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proc. of the 4th NSDI*, 2007. 130

[WWZ+13] Tao Wang, Jun Wei, Wenbo Zhang, Hua Zhong, and Tao Huang. Workload-aware anomaly detection for Web applications. *Journal of Systems and Software*, March 2013. 26, 131

[XJXC12] Caiming Xiong, David Johnson, Ran Xu, and JJ Jason J. Corso. Random forests for metric learning with implicit pairwise position dependence. *Proc. of the 18th ACM SIGKDD*, page 958, 2012. 115

[XW05] Rui Xu and Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–78, May 2005. 112

[YHC04] Miin-Shen Yang, Pei-Yuan Hwang, and De-Hua Chen. Fuzzy clustering algorithms for mixed feature variables. *Fuzzy Sets and Systems*, 141(2):301–317, January 2004. 112

[YJ06] Liu Yang and R Jin. Distance metric learning: A comprehensive survey, 2006. 112

[YZ06] Chunyu Yang and Jie Zhou. HClustream: A Novel Approach for Clustering Evolving Heterogeneous Data Stream. In *Proc. of the 6th IEEE ICDMW*, pages 682–688. IEEE, 2006. 112

[Zel87] M Zeleny. Management support systems: towards integrated knowledge management. *Human systems management*, 7(1):59–70, 1987. 41, 158

[ZZX+10] Minqi Zhou, Rong Zhang, Wei Xie, Weining Qian, and Aoying Zhou. Security and Privacy in Cloud Computing: A Survey. In *Proc. of the 6th International Conference on Semantics, Knowledge and Grids*, pages 105–112. Ieee, November 2010. 16

CC
SOME RIGHTS RESERVED

Unless otherwise expressly stated, all original material of whatever nature created by Rafael Brundo Uriarte and included in this thesis, is licensed under a Creative Commons Attribution Noncommercial Share Alike 2.5 Italy License.

Check creativecommons.org/licenses/by-nc-sa/2.5/it/ for the legal code of the full license.

Ask the author about other uses.