Service Level Agreements-Driven Management of Distributed Applications in Cloud Computing Environments

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Alexandru-Florian Antonescu

von Rumänien

Leiter der Arbeit:
Professor Dr. Torsten Braun
Institut für Informatik

k

Service Level Agreements-Driven Management of Distributed Applications in Cloud Computing Environments

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Alexandru-Florian Antonescu

von Rumänien

Leiter der Arbeit:
Professor Dr. Torsten Braun
Institut für Informatik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, September, 2015

Der Dekan:
Prof. Dr. Gilberto Colangelo

Advancements in cloud computing have enabled the proliferation of distributed applications, which require management and control of multiple services. However, without an efficient mechanism for scaling services in response to changing workload conditions, such as number of connected users, application performance might suffer, leading to violations of Service Level Agreements (SLA) and possible inefficient use of hardware resources. Combining dynamic application requirements with the increased use of virtualised computing resources creates a challenging resource management context for application and cloud-infrastructure owners. In such complex environments, business entities use SLAs as a means for specifying quantitative and qualitative requirements of services.

There are several challenges in running distributed enterprise applications in cloud environments, ranging from the instantiation of service VMs in the correct order using an adequate quantity of computing resources, to adapting the number of running services in response to varying external loads, such as number of users. The application owner is interested in finding the optimum amount of computing and network resources to use for ensuring that the performance requirements of all her/his applications are met. She/he is also interested in appropriately scaling the distributed services so that application performance guarantees are maintained even under dynamic workload conditions. Similarly, the infrastructure providers are interested in optimally provisioning the virtual resources onto the available physical infrastructure so that her/his operational costs are minimized, while maximizing the performance of tenants' applications.

Motivated by the complexities associated with the management and scaling of distributed applications, while satisfying multiple objectives (related to both consumers and providers of cloud resources), this thesis proposes a cloud resource management platform able to dynamically provision and coordinate the various lifecycle actions on both virtual and physical cloud resources using semantically enriched SLAs. The system focuses on dynamic sizing (scaling) of virtual infrastructures composed of virtual machines (VM) bounded application services. We describe several algorithms for adapting the number of VMs allocated to the distributed application in response to changing workload conditions, based on SLA-defined performance guarantees. We also present a framework for dynamic composition of scaling rules for distributed service, which used benchmark-generated application monitoring traces. We show how these scaling rules can be combined and included into semantic SLAs for controlling allocation of services. We also provide a detailed description of the multi-objective infrastructure resource allocation problem and various approaches to satisfying this problem. We present a resource management system based on a genetic algorithm, which performs allocation of virtual resources, while considering the optimization of multiple criteria. We prove that our approach significantly outperforms reactive VM-scaling algorithms as well as heuristic-based VM-allocation approaches.

q

Contents

Preface

The work presented within this thesis was achieved during my enrolment as external PhD Student at the Institute of Computer Science and Applied Mathematics (IAM) of the University of Bern, while I was employed as PhD Student at SAP (Schweiz) AG. During that time, I was mainly funded over the European-Union research project GEYSERS. These two organizations deserve my honest appreciation for supporting me and giving me the opportunity to pursue my research.

I express my gratitude to Prof. Dr. Torsten Braun, head of the Computer Network and Distributed Systems group (CDS), for supervising this work and for his insightful advises. Prof. Dr. Torsten Braun encouraged and motivated me to publish my results in various reputable conferences and journals. I also thank Prof. Dr. Burkhard Stiller, who was willing to be the co-examinator of this thesis.

Many thanks go to my fellow colleagues from the RVS/CDS group. Special thanks go to Matthias Thoma, André Sérgio Nobre Gomes, Dr. Almerima Jamakovic-Kapic, Islam Alyafawi, and Jonnahtan Saltarin for their good cooperation over the past years. Our group secretary Daniela Schroth further deserves my gratitude for having done an excellent job. Special thanks go to Philip Robinson and Marc Brogle from SAP for their kind support they offered me during my research.

Finally, I am deeply grateful to my wife Mara for her support and for sharing a wonderful time together. I am grateful to my father Dumitru, my sister Loredana, my family members Marcela and Virgil, Dan and Doina, Cora and Viorel for their support and encouragement. Special thanks for keeping me motivated go to my friends Lacramioara and Rainer, Carolina, Regina, Daniel and Nicoleta, Stefan and Sabine, Manfred, Kurt and last, but not the least to my friends from Baicoi, Mircea si Ioana, Irina, Corina.

Chapter 1

Introduction

1.1 Background

Recent advancements[158] in cloud computing[108] have enabled the proliferation of distributed applications[154], which require orchestration and control of multiple services running on virtualised containers inside network-connected computing environments. However, without an efficient mechanism for scaling services in response to changing workload conditions, application performance might suffer, leading to violations of Service Level Agreements (SLA) and/or inefficient use of hardware resources. By combining dynamic application requirements with the increased use of virtualised computing resources a highly challenging resource management situation is created for owners of distributed application and network and computing infrastructures.

The future network platform[59, 20] for enterprises emerges as an easy-access distributed environment providing the necessary secure infrastructure, technologies, applications, and content to deliver end-to-end business services optimised for mobility. We refer to this as the Business Web [136]. Applications developed for and using enterprise platforms are composed of globally-distributed components delivered as services. Examples include analytics[134], logistics[71], and collaborative business[40]. These services function autonomously with globally-distributed users having different workload / connection characteristics. The volumes of stored and transmitted information continue to grow to the order of exabytes[42]. This makes intelligent resource usage optimization and communications acceleration techniques mandatory, including scaling, parallelisation, and load-balancing.

In such complex environments, business entities use SLAs[125] as a means for specifying quantitative and qualitative requirements of services. Given that SLAs have legal and financial implications, proper management of SLAs is critical for modern and future application and infrastructure service providers. Leading cloud service providers of infrastructure or application services (e.g. Google[68], Amazon[8]) use SLA management for spec-

ifying and maintaining the quality of service (QoS) and availability levels to their customers. Dealing with exclusively-owned virtual machine (VM) instances deployed on shared physical infrastructures presents a greater challenge for each lifecycle phase of resource management. This is caused by (1) the multi-objective resource allocation optimization problem of having to maintain SLAs for various customers while minimising resource utilisation targets, and (2) the differentiation in SLA-requirements from different classes of VMs and users. Furthermore, the violation of SLAs results in financial penalties for the cloud-infrastructure provider, adding a direct economic dimension to the problem.

1.2 Motivation

SLAs are also important for maintaining the reputation of the infrastructure cloud provider as consumers compare SLA claims and experience when selecting providers[155]. They can then plan the performance and risks of applications more optimally by having knowledge of the infrastructure's availability and performance [124] [141]. For this reason providers will seek ways of ensuring optimal SLA satisfaction, within the constraints of their resource usage, energy consumption and budget constraints, in order to maintain business. Having insight or prior knowledge of application workloads and data volumes that will be running on their infrastructure is hence beneficial towards tuning their infrastructure for optimizing both of these objectives. This reflects the goals of Cross Stratum Optimisation (CSO) [92], such that an effective, automated SLA management architecture should be considered important in this context. CSO refers to performing optimisations across multiple layers of the cloud management system (CMS), such as orchestrating the horizontal scaling of services on the upper layers of the CMS with the optimized allocation of virtual machines hosting the services to physical infrastructure resources.

When prioritizing operational tasks and resources based on their commitments, capacities and capabilities, providers then use the information in an SLA. Providers aim to satisfy the Service Level Objectives (SLOs) of consumers without disrupting their internal operation goals including minimisation of operational costs, power consumption and legal issues. Network operators and Internet Service Providers have been using SLAs for assuring client's experience in remote access to services and inter-networking scenarios for many years [124][142]. SLAs are being used generally in Cloud Computing and Infrastructure as a Service (IaaS) for similar purposes [124][155]. The demand for rapid delivery of highly available, high performance, networked applications and infrastructure services makes automation support for SLA Management more critical.

However, the combination of CSO and SLA Management is not a straight-

forward and complimentary relationship, especially when delivering services composed of resources with different owners and concurrent consumers. First, existing and emerging application and infrastructure services are no longer single, isolated, remote servers, virtual machines (VMs) or bandwidth. Disruptions in one resource can have consequences for the overall composite service, such that state information needs to be controllably shared. There are then dependencies across operational strata (application, machine and network) and domains with autonomous administrators, lifecycles and management controllers, which can lead to a break in the autonomy property. Secondly, application topologies continue to become more complex and heterogeneous as service-oriented design becomes the established enterprise architectural pattern for building distributed applications [26]. For these reasons the interdependencies between application deployment, usage and network capabilities need to be considered during service planning and provisioning, making the coordination of SLA management more complex.

Using SLA management and data prediction for system reconfiguration enables both minimising SLA violations caused by environment's dynamics (e.g. variable workload), and triggering infrastructure reconfiguration actions for optimising resource utilisation [19]. By analysing the large amounts of monitoring information collected from distributed, cloud-deployed services, it becomes possible to (1) observe patterns in service utilisation and then (2) to use the gathered information for optimising the allocation of network and computing infrastructure resources in a cloud. By composing series of performance related patterns gathered over longer time periods, an SLA management system can use this information for elastic deployment of services, while also maintaining the SLA guarantees [82] associated with the given services.

Given the business importance of SLAs in the context of a company's reputation, customer satisfaction, and operational objectives, we can conclude that SLAs must be taken as input by any cloud management platform and in particular by those management platforms coordinating multiple distributed computing environments. SLAs are thus a critical input parameter for distributed service orchestration platforms, considering their impact on cloud resources reservation and provisioning. Each closed group of business users, charged and handled as a single entity, sharing common access to dedicated software instance (including its data, configuration and user management) is reffered as a cloud *tenant* [90].

Finding the proper resource allocation in Cloud Computing environments composed of multiple tenants with different SLA requirements implies solving a multiple objective allocation problem. This is caused by the conflicting requirements of the different cloud resource providers (infrastructure, platform and software [108]) and the cloud tenants. Usually, the providers of the cloud resources have efficiency requirements regarding minimising the total cost of ownership (TCO), as well as minimising the

penalties caused by the SLA violations. The cloud tenants have performance related SLA requirements, such as maintaining certain levels of application performance metrics (e.g. transactions per second, average response time, etc.). As the providers' requirements translate to reducing the number of infrastructure-resources, and the tenants' requirements translate to increasing the quantity of cloud-resources, this leads to the multi-objective problem of satisfying tenants' SLAs while ensuring the operational efficiency of the cloud infrastructure provider.

1.3 Problem Statement and Research Question

We focus our analysis on the difficulties associated with SLA-aware control and scaling of cloud systems hosting distributed applications composed of inter-dependent services. We further break this problem into three sub-problems, which can be defined by the following research questions:

1. "How do distributed application owners and infrastructure providers benefit from using Service Level Agreements in Cloud Computing environments?"

2. "How to control the allocation of infrastructure resources in cloud environments while maximising both tenant's Quality of Service and provider's Efficiency of Operations?"

3. "How to design a system for management (scaling) of SLA-constrained distributed cloud services?"

The solutions to the previous three questions help answer the following main research question: **How to design a cloud-infrastructure management system based on service level agreements for the SLA-compliant management of both services and cloud-infrastructure resources?**

In order to answer these questions, we considered multiple connected problems from the perspective of both enterprise application owners and cloud infrastructure providers. The efficient management of SLAs is of particular importance for Cloud Computing, where exclusively-owned Virtual Machines (VMs) are allocated resources on hosts in a shared physical infrastructure.

The application owner is interested in finding the optimum amount of computing network and storage resources for ensuring that the performance requirements of all applications are met. After the initial allocation of virtual computing resources, the application owner is also interested in appropriately scaling distributed applications so that application performance guarantees are maintained even under dynamic user-generated workload conditions.

Similarly, infrastructure providers are interested in optimally allocating the virtual resources on the physical computing and network infrastructure so that their operational costs are minimised, while maximising the tenant's application performance. In this context, we show how predicting the infrastructure utilisation peaks and dynamically optimising the distribution of virtual resources leads to improving the use of available cloud resources, lowering the penalties caused by violating the tenant's performance-SLAs during live-migration of VMs.

At the cloud-platform level, the tenants are interested in obtaining a constant application-level performance, independent of the number of external users accessing their cloud-applications. This behaviour can be obtained if the cloud-platform-manager ensures that the appropriate number of virtual resources (e.g. VMs) are allocated to the distributed applications. Optimally determining the number of necessary VMs for supporting given levels of application-performance leads to three distinct sub-problems: (1) finding the relation between the quantity of virtual resources (or size of the virtual infrastructure) allocated to distributed cloud-applications and the corresponding application-level performance metrics; (2) finding the optimal VM-scaling policy; and (3) using prediction to optimally vary the number of VMs allocated to cloud applications in order to minimise resource utilisation while meeting the agreed SLA performance guarantees.

1.4 Contributions

Overall, the key contributions of this dissertation are:

- a semantic SLA-enabled specification language and architecture for dynamically managing distributed software, together with the corresponding computing, storage and network cloud infrastructure virtual and physical resources [18] (described in Chapter 3)
- three SLA-based VM scaling algorithms that use reactive and analytic mechanisms combined with data prediction method, as well as results from applying Little's Law to SLA defined objectives ([14, 12, 15]), such as keeping the average execution time of end-to-end requests below a given maximum value (described in Chapter 4)
- a method of sizing virtual infrastructures based of SLA-defined constraints and benchmark-gathered application profiling information [16], and composing scaling rules for distributed services using prediction mechanisms and correlation-derived relationships between SLA monitoring metrics [11] (Chapter 5)
- a framework for dynamically allocating VMs to physical hosts while considering both SLA-constraints and multiple objectives optimisa-

tions, which uses a bin-oriented genetic algorithm combined with data forecasting (exponential smoothing) [19] (described in Chapter 6)

- implementation of the previously mentioned algorithms and methods for SLA-aware management of cloud infrastructure and software resources in the Service Middleware Layer (SML) component of the GEYSERS FP7 EU research project ([17], [52], [132]) (described in Chapter 7)

Proposed Solution

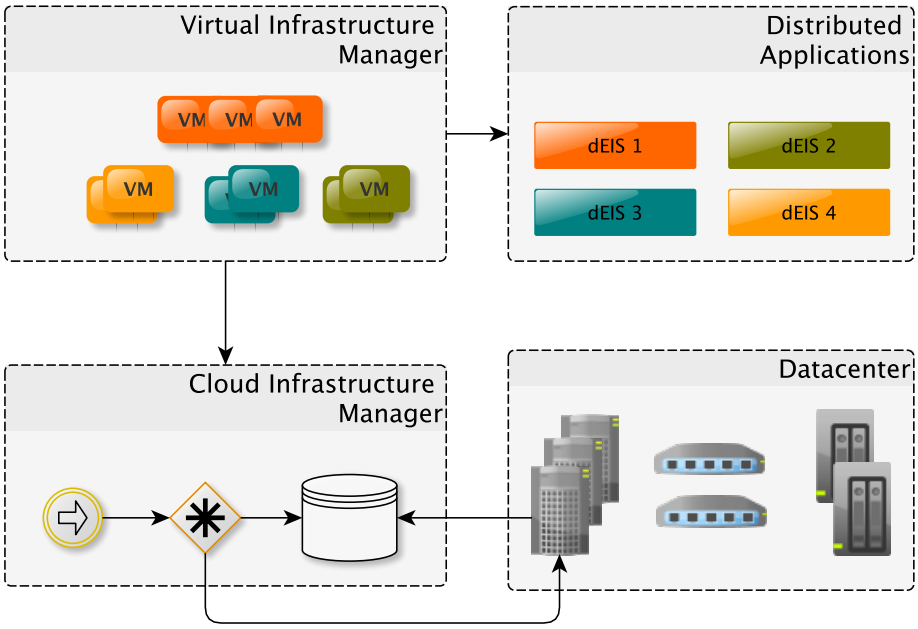


Figure 1.1: Architecture Overview

The proposed solution to the management problem identified in Section 1.3 is to use two connected sub-systems: Cloud Infrastructure Manager (CIM) and Virtual Infrastructure Manager (VIM), as shown in Figure 1.1. The VIM part of the system deals with the management of virtual resources (distributed applications), such as provisioning and scaling distributed applications belonging to the cloud tenants. The CIM subsystem deals with cloud infrastructure management, specifically allocation of virtual resources (VMs) to physical resources (e.g. computing and network) in the datacenter, monitoring and infrastructure optimization (e.g. migration).

SLA-based Virtual Infrastructure Manager

The largest part of this research is focused on the SLA-driven management of cloud distributed applications, in particular on the aspects of scaling cloud services under SLA constraints. We first described the SLA mechanisms associated with describing distributed services and applications, which we then used as input for different algorithms for scaling services and allocating them to cloud infrastructure resources. Finally these algorithms were incorporated into a SLA-based cloud management platform.

The VIM is described in chapter 4, where we focus on using the service management platform for testing different VM scaling algorithms. For this purpose, we created a statistical model of the dEIS application, which we then later used as a basis for comparing different SLA-based VM-scaling algorithms. In section 7.2 (based on our paper [12]) we describe the performance characterisation of the dEIS system and how the gathered data can be used for creating a simulation model of dEIS using the CloudSim cloud simulator. Having validated the use of the dEIS models in CloudSim, we continued by extending this work with a reactive SLA-based VM-scaling algorithm, which we described in section 4.4.1 (based on our article [14]).

The VIM leverages on the extended SLA description language presented in chapter 3 for describing the components, resources and monitoring metrics of a distributed Enterprise Information System (dEIS)[19]. The dEIS (described in section 7.1) was then used as a basis for further exploration of different mechanisms used by the Cloud-Services Management Platform, which takes SLAs as control inputs in order to scale the application-services.

The algorithms used by the VIM system are described in section 4.4 (based on our article [15]), together with a method of finding the optimal number of VMs required for ensuring that given performance constraints are met under varying workload conditions.

Also, parts of the algorithms, methods and frameworks developed during this thesis were implemented, tested and validated in the context of the European Research project GEYSERS [52], [17], [132].

Cloud Infrastructure Manager

The Cloud Infrastructure Manager (CIM) is responsible for controlling the allocation of datacenter infrastructure resources to virtual resources. In order to perform this task, the CIM first discovers and configures the physical infrastructure, followed by the provisioning of the virtual resources.

In chapter 6 (based on our article *Dynamic SLA Management with Forecasting using Multi-Objective Optimizations*) we present a bin-oriented genetic algorithm for allocating VMs to physical resources in a cloud environment, while taking into consideration multiple optimisation criteria (energy efficiency, SLA violation minimisation, user performance maximisation) for

both the cloud provider and its tenants. The requirements were described as Service Level Agreements (SLAs) using the specification language that we presented in chapter 3. We used the semantic SLA specification for both describing the topology of the distributed cloud applications, its requirements in terms of infrastructure resources, and application-level performance monitoring metrics.

The practical implementation of the CIM subsystem [17, 132], specifically the platform for allocation of virtual machines to physical infrastructure resources, have been developed and described in the context of GEYSERS EU research project [52].

1.5 Thesis Outline

In this section we present the outline of the thesis, focusing on the main research contributions in the context of cloud computing. In chapter 1 we gave a introduction into (1) the research question (optimizing allocation and scalability of cloud resources and services using objectives defined in Service Level Agreements), (2) connected problems addressed by this research (such as multi-objective optimization of allocation of infrastructure resources, optimal capacity planning for cloud services, processing of large datasets composed of service monitoring data) and (3) the problem statement (designing a SLA-based cloud management system for controlling allocation and scaling of enterprise distributed applications). Also, the summary of the contributions is presented, linked to the corresponding chapter of the thesis.

Chapter 2 describes the background and related research works organised in four sections: 2.1 Cloud Computing, 2.3 Distributed Enterprise Applications, 2.2 Service Level Agreements, and 2.4 data processing mechanisms needed for handling large datasets of service-generated monitoring information.

Chapter 3 describes the semantic model used for representing SLAs, services and cloud resources. We present the concepts of topology orchestration as the basis of unattended provisioning of distributed service deployments. We also describe the SLA model, which is the foundation for scaling and allocation algorithms described later in the thesis. We introduce the concepts of guaranteed states and actions as core SLA instruments for automated management of cloud infrastructures. We then link these two concepts to the SLA-defined application-level key-performance indicators (e.g. metrics), which enable monitoring of the state of distributed services, as well as triggering automated responses in case of performance degradation, beyond the thresholds defined in the SLAs.

Chapter 4 describes the algorithms and methods used for scaling cloud services under SLA-defined performance guarantees and dynamic workload conditions, by using the semantic SLA models described in the previous

chapter. We present the architecture of the SLA-based service scaling manager (SSM) which is responsible for the horizontal scaling of distributed services according to performance guarantees specified in SLAs. We present next a multi-step prediction algorithm using auto-regression, which we used in the scaling algorithms for predicting the state of distributed services and the parameters of the scaling triggers.

Chapter 5 introduces several methods for sizing virtual infrastructures and dynamically generating scaling rules for services, based on SLA-defined performance constraints. We investigate how SLAs can be dynamically optimized for enhancing the rules controlling the scaling-out (increasing the number of VMs allocated to distributed services) and scaling-in (reducing the number of VMs allocated distributed services) services belonging to distributed applications, in particular Enterprise Information Systems (EIS).

Chapter 6 presents a multi-objective SLA-based virtual resource allocation algorithm (e.g. the efficient allocation of virtual machines to physical hosts), which considers multiple criteria coming from both the infrastructure cloud provider and the customers using the IaaS services for running distributed applications.

Chapter 7 contains the evaluation of the previously presented algorithms and methods for service management, including descriptions of the distributed application, evaluation testbed, as well as the performed experiments and simulations.

Finally, Chapter 8 presents the summary of the challenges addressed, as well as the contributions of this research work and the outlook.

Chapter 2

Background and Related Work

2.1 Cloud Computing

"Cloud computing" is a well established computing paradigm developed as a result of the convergence of multiple virtualisation technologies, such as computing, network, storage, platform and services. It is characterised by rapid provision of scalable IT capabilities as a service (e.g. platform, software) in an on-demand and self-service manner over the network using large pools of infrastructure resources.

The term *Cloud Computing* has received over the time many definitions, most of them revolving around the outsourcing of IT activities to external third-parties business, which have large pools of available resources and are able to meet the IT-related needs in an efficient manner. This view on Cloud Computing can be found in the official definition given by the National Institute of Standards and Technology [108]:

> "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."

In a similar manner, Forrester Research [131] presents the following definition, which emphasises the network-related delivery of IT services, as well as its economical aspects:

> Cloud computing is a standardised IT capability (services, software, or infrastructure) delivered via Internet technologies in a pay-per-use, self-service way

2.1.1 Taxonomy of Cloud Computing Models

Given the broad nature of the available cloud services, they can be classified according to certain fundamental models into three main categories, as

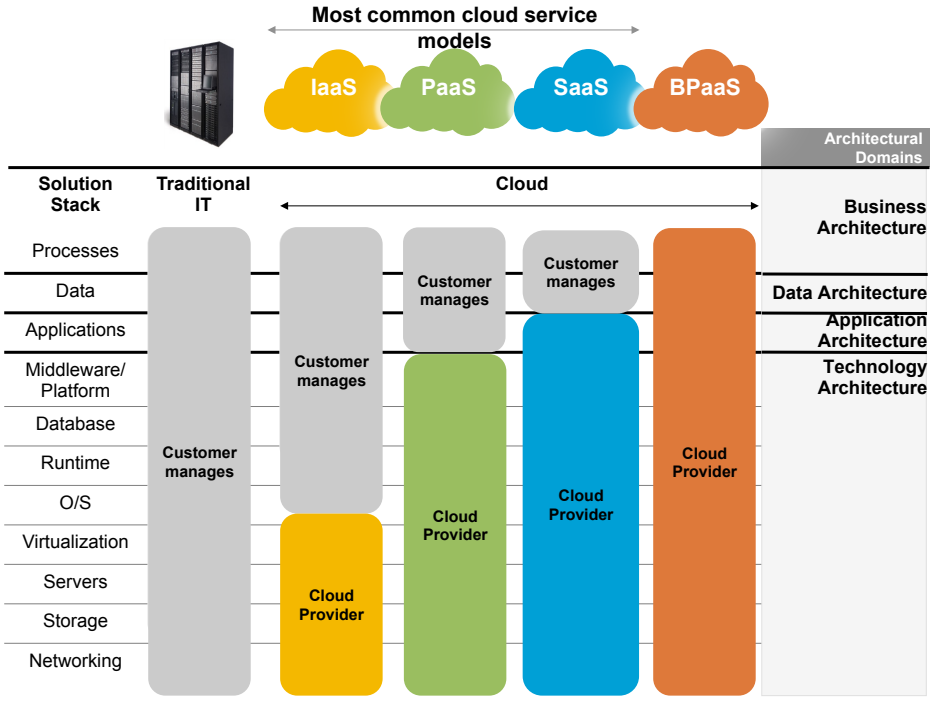


Figure 2.1: Comparison of Cloud Computing Models

shown in Figure 2.1, namely: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), Software-as-a-Service (SaaS). The fourth category is Business-Processes-as-a-Service (BPaaS) and defined according to Gartner [62] as "the delivery of business process outsourcing (BPO) services that are sourced from the cloud and constructed for multitenancy. Services are often automated, and where human process actors are required, there is no overtly dedicated labor pool per client, the pricing models are consumption-based or subscription-based commercial terms."

The three core models of Cloud Computing (IaaS, PaaS and SaaS) refer to different proportions of the managed technology-services offered by the Cloud provider, and consumed by the Cloud users.

The most basic level as defined by IETF [86] and NIST [108] is the *Infrastrucuture-as-a-Service* (IaaS), responsible for the virtualisation and abstraction of the cloud-infrastructure physical resources, namely networking, computing and storage. At this level, the cloud provider controls the provisioning of infrastructure resources, by managing their configuration so that each consumer (cloud tenant) is given its set of resources for exclusive use. The cloud user can run arbitrary software, including applications and operating systems inside virtual machines (VMs), however, with only limited control of the underlying physical resources (e.g. network firewalls), while being charged on a utility basis and by consumption. This level of abstrac-

tion is achieved by delegating control of the infrastructure resources to a hypervisor, such as KVM [5], Xen [28], VMware [153], Hype-V [110], which will host VMs belonging to the cloud tenants. Examples of IaaS providers are Amazon EC2[7], Google Compute Engine [67], Microsoft Azure Virtual Machines [112], SAP Monsoon Infrastructure Service.

The *Platform-as-a-Service* (PaaS) is a segment of cloud computing services, where customers are provided with a computing platform and a stack of solutions/ tools (typically including operating system, execution environments for different programming language (e.g. JAVA virtual machines), database, web servers, etc.) for further development of applications or services, without the complexity of provisioning hosting capabilities or the cost of managing and buying the underlying software and hardware. This reduces the necessary capital expenditures and allows a gradual increase of the costs with the size of the deployed infrastructure. An important aspect of the PaaS model is the option of having the underlying computer and storage resources scale automatically in order to match the application demands so that the cloud user does not have to allocate resources manually. Examples of PaaS cloud providers include Microsoft Azure [111], Google App Engine [66], or SAP HANA Cloud Platform [135].

In the business model using *Software-as-a-Service* (SaaS), cloud users are given access to application software and databases, which run on managed infrastructure and platforms, eliminating the need to install, run and manage on-premise software. Usually, the customer subscribes to usage contracts, and she/he is billed based on some sort of pay-per-use policies (e.g. data volume, number of transactions, etc.). The cloud applications are accessed through thin clients, such as web or mobile application interfaces.

A characteristic of the provider-managed cloud applications is that of their high scalability, ensuring that very large data volumes are handled without any need for manual scaling of the underlying software or infrastructure resources. This is achieved by using multiple virtual machines together with load balancers, which transparently distribute the work over the set of virtual machines at run-time in order to meet the changing workload demands of the cloud users.

Beyond the three cloud models presented above (IaaS, PaaS and SaaS), Forrester is predicting the rise of *Business Process as a Service* (BPaaS)[62] as part of the Business Process Management discipline[51], which is a new market beyond SaaS. It offers Human Resources [65] as a service, considering not only applications or technology, but also includes services performed by people. Above the SaaS level, it is common to find a wide range of service offerings, the so-called Anything-as-a-Service or XaaS [100].

2.1.2 Characteristics of Cloud-Computing Applications

Resource virtualisation is the main driver of Cloud Computing, and it refers to the possibility of creating virtual (rather than actual) version of resources, such as operating system, server, storage or network resources. Virtualisation considers several aspects:

- allows partitioning of physical resources into multiple logical or virtualised resources
- isolates the virtualised resources from each other, providing (some kind) logical independence from hardware
- encapsulates virtual resources, allowing saving the state of virtual resources into persistent formats (e.g. files) and enabling it to be moved to different cloud environments

Resource virtualisation enables the realisation of another critical concept for Cloud Computing, that of *multi-tenacy*. Multi-Tenancy refers to a principle in software architectures, where a single instance of the software runs on a server, serving multiple client-organisations (tenants). It contrasts with multi-instance architectures where separate software instances (or hardware systems) operate on behalf of different client organisations.

The range of enabling technologies involved in the Cloud Computing paradigm are shown in Figure 2.1 and they are: networking, storage, servers, virtualisation, Operating System (OS), application runtime, database, middleware/platform, applications, data and processes. In the next subsection we present a classification of Cloud Services based on the granularity of managed services regarding the covered cloud technologies.

Characteristics of Cloud Computing paradigm include the following elements:

- **Use of shared computing resources**, such as physical infrastructure (computing, storage, network), IT platform (databases, communication busses, distributed caches) and software. The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned according to consumer demand.
- **(Exact) location of the cloud provider** is usually not known by the cloud consumer, however (s)he may be able to specify the location at a higher level of abstraction.
- **Rapid and elastic provisioning**, where cloud users request resources in a self-service manner, and then they are quickly served with the requested service or services. Also, capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward on-demand.

- **multi-tenancy**: the cloud provider supports a large number of tenants [90] using the shared infrastructure and services, enabling centralisation of the physical infrastructure in location with lower costs (energy, real-estate, etc.) and increasing the efficiency of operations for customer systems with low utilisation levels (e.g. 10-20

- **Minimal management effort**, implying that the cloud users are given an easy and simple to use management interface for their cloud services and resources, without the need of doing complicated configuration operations

- **Services are consumed in a standardised manner through the Internet or Virtual Private Networks from the cloud provider**. Capabilities are available over the Internet and accessed through standard mechanisms that promote the use of both heterogeneous thin or thick client platforms. A broad and reliable network access is necessary, and the access to cloud services might be direct or through secured virtual networks.

- **Flexible financing models**, such as pay-per-use, subscription, auction, volume-based, and others. The underlying idea for pricing cloud services is based on a subscription model rather than buying IT assets such as licenses and/or hardware, which leads to a conversion of the costs from fixed costs into variable costs.

- **On-demand self-service**: implying that the cloud services are requested, provisioned and consumer in a dynamical manner, creating the impression that the cloud environment offers unlimited amount of resources. This allows the cloud consumers to unilaterally initiate requests of additional computing capabilities, as they require.

- **Isolation of services**: Cloud services are offered in a secure manner, where different cloud tenants (business users) share the same physical infrastructure, but do not affect the functionality and performance of each other's applications and cloud services.

- **Rating, charging and billing**: Cloud management systems automatically control and optimise resource usage by leveraging metering capabilities at some level of abstraction appropriate to the type of service. Resource usage can be monitored, controlled, scaled, and reported.

- **Standardisation of services**: Cloud computing principles are based on standardisation of cloud management services, allowing for decoupling of the implementation of the Cloud services and their interfaces.

- **Quality of service (QoS)**: Cloud Providers offer pre-defined and guaranteed levels of quality for the offered services. QoS levels are specified in

Service Level Agreements, which allow the clients to choose the cloud provider with the matching level of technical needs.

2.1.3 Cloud Computing Deployment Models

Depending on how the three main cloud computing models are implemented and deployed in various organisations, we can distinguish four deployment models according to NIST [108], namely private, public, hybrid and community.

Private Cloud

Resources inside a private cloud are operated solely by an organisation, making this model the most secure of the four presented deployment models, as resources such as servers, storage, and network are virtually available only to one organisation. A private cloud can be deployed on-premise or operated externally in the data center belonging to a cloud provider.

Public Cloud

In case of the public cloud deployment model, the infrastructure is provisioned for the open use of the general public. It may be owned, managed, and operated by a business, academic, or government organisation, or a combination of them. In general, it allows multi-tenancy.

Hybrid Cloud

A hybrid cloud can include private cloud setups, and it may be comprised of two or more cloud infrastructures that remain unique entities but are bound together by standard or proprietary technologies, thus enabling data and application portability (e.g., cloud bursting for load balancing between clouds).

Community Cloud

In the case of a community cloud[102], several organisations with similar concerns (e.g. mission, security, requirements, policy, compliance, etc.) share a cloud infrastructure, which may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on- or off-premises.

2.1.4 Cloud Management Platforms

A cloud management platform must provide a wide range of capabilities to support the needs of organizations using hybrid clouds infrastructures with potential multiple infrastructure providers.

Characteristics of Cloud Management Tools

We present in this section an overview of the requirements that a cloud management solution must meet in order to support the Cloud Computing specifications, with particular regards to offering of IaaS services. In order to objectively compare cloud management framework a common basis for requirements needs to be established. These prerequisites are extracted from the specifications of the European large-scale research project GEYSERS (Generalized Architecture for Dynamic Infrastructure Services) [52] and are presented below.

Registry enabling management of physical resources. An important requirement for a cloud management platform is to offer visibility of the available hardware resources, like servers, network-attached storage, network equipment, etc. and to support lifecycle operations on those resources.

Adaptors/ Connectors for various types of physical infrastructure resources. Because the hardware in a cloud environment is typically heterogeneous, a management solution must have different types of connectors and provide an abstracted management interface in order to perform orchestration of the different resources located in the datacenter.

Monitoring and feedback for infrastructure management actions. Monitoring of physical, virtual and software cloud-resources must be supported in order to facilitate management of infrastructure along with accounting and load balancing.

Physical resource abstraction and representation as logical resources. In order to allow on-demand allocation of virtual resources from possible multiple providers, an abstraction of the actual hardware resources must be performed.

On-demand provisioning of virtual resources. An essential characteristic of a cloud management solution is the offering of on-demand provisioning of infrastructure / platform / software resources through the use of APIs (e.g. web interfaces, remote method invocation, etc.).

Support for multiple virtualisation technologies. The management solution must be able to connect to different types of hypervisors.

Scheduling of provisioning actions. In order to enforce isolation between different provisioning actions, planning and scheduling of the requests must be supported. Also, in order to support elastic scaling of virtual resources, scheduling of provisioning actions must be supported, allowing dynamic in or out-scaling of the virtual infrastructures.

Declarative approach to resource representation and management. In order to support composition and requesting of new virtual infrastructures, it is desirable to express the composition of the infrastructure using a declarative approach, such as using different languages for resource representation (e.g. DMTF's OVF [49]).

Monitoring of single and aggregate virtual resources. In order to support

accounting and high availability, monitoring of provisioned infrastructure resources must be supported by the underlying cloud management framework.

Discovery of physical resources and their virtualisation capabilities. In order to allow dynamic modifications of the underlying hardware landscape, (automatic) registration of newly added hardware resources must be supported as well as de-registration of resources which have been removed/upgraded.

Enablement of resource discovery selection based on registered capabilities. Automatic resource discovery enables a better management of the underlying hardware resources as well as support for dynamic landscape reconfiguration (for example adding new network resources).

Registry for multiple infrastructure service providers (external resources) In case of hybrid clouds, formed by integrating both on-premise and remote (external) resources, the management framework should support this situation by allowing registration of multiple resource providers.

Resource matchmaking based on user-requests. By definition, cloud services are elastic and dynamic. These properties are enabled by supporting dynamic allocation of virtual resources. This is enabled by the management framework by allowing a flexible way of defining requirements for the provisioned resources, thus performing resource matchmaking against the available datacenter resources.

Support for virtual infrastructure construction by composing multiple virtual resources. In case of cloud providers offering more than one type of infrastructure resources, for example compute / storage and network, it is desirable to allow creation of a single virtual infrastructure containing all the provisioned resources, instead of individually managing the created virtual resources.

Management of the virtual infrastructure's lifecycle. A cloud management framework must support all the operations for the administration of virtual infrastructure's lifecycle, from provisioning, configuration and control, to secure decommissioning. This implies orchestrating more than one operation on the underlying infrastructure resources.

Support for offering multiple virtual infrastructures in parallel (multitenancy). In order to provide true cloud services, multiple parallel infrastructures must be supported for running on shared hardware infrastructures. This implies the existence of isolation features in the underlying hypervisors as well as planning and scheduling support in the cloud management framework.

Comparison of virtual resource operational status against concrete metrics or quality constraints (response time, availability, mean CPU usage). In order to ensure advanced Service Level Agreements (SLA) support, the cloud management framework must first support defining such metrics and it then must integrate automatic monitoring of virtual resources status in order to ensure enforcing of agreed SLAs.

Cloud Management Tools

We present a non-exhausting list of open-source cloud management frameworks, which can be used for the implementation of an *IaaS* cloud provider. Form the six presented management platforms, OpenNebula was selected as the one to be used in the practical experiments described in this thesis. It was selected at that time because of the stability, community support, supported APIs (XML-RPC[146], portal-based self service, command-line), interoperability with open source tools (libvirt[73], KVM[5]) as well as having most of the characteristics presented in Section 2.1.4.

AbiCloud

Abicloud [3] is an open-source cloud-infrastructure manager for the creation and integral management of Public and Private Clouds. The framework facilitates the creation, deployment and management of cloud computing infrastructures, allowing other users to create virtual data centers over the cloud infrastructure by allowing them to provision, scale and control servers, storage, networks, virtual network devices as well as applications. It is expected that in future versions it will support cloud federation and integration with public cloud providers, as well as extending virtualisation support.

Eucalyptus

Eucalyptus [140] enables creation of hybrid and private cloud deployments for enterprise data centers. Leveraging Linux and web service technologies, Eucalyptus allows customers to create computing clouds "on premise", at the same time supporting the popular AWS [7] cloud API, which allows on-premise clouds to interact with public clouds.

OpenNebula

OpenNebula [117] is an open-source project aimed at building a cloud computing tool for managing the complexity and heterogeneity of distributed data center infrastructures. It provides a complete toolkit for building IaaS private, public and hybrid clouds, by managing virtual machines, storage and network resources throughout their lifecycle. Its features and APIs are also extensible with the possibility of installing further add-ons. It supports building private clouds or clusters running different virtualisation platforms (KVM, Xen, VMware, Hyper-V). It supports external cloud resources by supporting Amazon EC2 [7] API and it also provides federation of different cloud instances in order to build a hierarchy of independent virtualisation clusters, thus enabling higher levels of scalability. From an public cloud perspective, it supports OGF OCCI [57] and Amazon EC2 interfaces, along

with offering native Ruby, Java and XML-RPC [146] APIs for creating new cloud interfaces

OpenQRM

OpenQRM[118] is an open-source data-center management platform, which enables automatic deployment of virtual infrastructures composed of appliances. It also supports multiple virtualisation technologies, such as KVM, XEN, Citrix XenServer, VMWare, lxc and OpenVZ. Similarly, it provides an API for integrating third-party tools as additional plug-ins. OpenQRM is published under GPL license. It was originally written in Java but ported to PHP. It also offers a self-service cloud portal with an integrated billing system that enables end-users to request new managed servers and application stacks on-demand.

OpenStack

OpenStack [58] is a collection of open source technology targeting massively scalable cloud infrastructure. It is composed of multiple related subprojects, such as (1) OpenStack Compute, which offers computing power through virtual machine management, (2) OpenStack Storage, which enables provisioning of redundant and scalable object storage pools, and (3) Open Stack Networking, which enables provisioning and management of software defined virtual networks. Closely related to the OpenStack Compute project is the Image Service project, which enables its self-servicing capabilities (allowing users to deploy the desired operating system and applications packaged as VMs [37]). OpenStack can be used for deploying large-scale cloud deployments for private or public clouds.

Scalr

Scalr [138] is an open source software stack for scaling web-based application-infrastructures. The framework provides a SOAP-based API for web clients, consoles and applications for connecting to scalable virtual infrastructures composed of multiple VM instances. VMs are created using a combination of the Amazon WS API and the Scalr API, by sending commands to hypervisors and VM instances for configuration purposes. The collection of virtual machine instances in a single domain is a Scalr Farm, which then can be mapped to an Amazon EC2 environment for extended VM management.

2.1.5 Allocation Strategies for Cloud Resources

One of the key benefits of using cloud computing is the scalability, translated into the ability to change the size of the virtual infrastructure (e.g. number of VMs or application instances) for accommodating variable workload. In

order to implement the scalability feature in a cloud management platform, it is important to have a reliable and flexible mechanism for allocating virtual resources (e.g. VMs) to physical resources (e.g. servers). In this subsection we present a short overview of the state of the art approaches to resource allocation in cloud environments.

Mazzucco and Dyachuk [105] propose an approach for allocating VMs to servers by considering energy efficiency aspects by controlling the number of running servers in the datacenter. However, they do not consider the case when a server could host multiple VMs. They also use a forecasting approach for estimating the arrival rate, similar to the one described in this thesis, but only use the number of running servers as the means of saving energy without considering consolidating VMs into fewer servers.

Xu and Fortes [156] describe a multi-objective resource allocation algorithm using a group oriented genetic algorithm with a fuzzy averaged fitness function, while we propose a cost based multi-objective evaluation function using forecasted utilisation levels based on historical monitoring data. The authors consider the initial allocation of virtual machines but not the costs associated with the SLA violations, nor the possibility of oversubscribing the resources based on forecasted utilisation data.

Zhang et al. [160] describe a runtime balancing system which uses statistical forecasting to determine if a VM will experience high CPU or network utilisation during either day or night period. Their proposed system uses this information to place VMs on corresponding hosts, while our approach uses a triple exponential estimation for forecasting of resource utilisation, considering also the data seasonal trends.

Caron et al. [39] propose using a string matching algorithm for forecasting resource utilisation demand in cloud environments by identifying the VMs with similar characteristics. Our approach differs by the used forecast algorithm, which considers data seasonal trends, and by using predefined values for resource utilisation of unknown VMs.

Kim et al. [87] describe a master-agent task scheduler based on the CometCloud management system, able to provision and size virtual infrastructures composed of VMs in hybrid infrastructure environments. Their system is able to operate in an autonomous manner and to recover from node failures. They do not address SLA guarantees, or how the network is treated as a managed resource.

Lu et al. [98] take another perspective on the problem of SLA-based planning by considering how to optimise decisions to outsource infrastructure demands to subcontractors. Their optimisation problem considers the costs, profitability and SLA failure risks associated with outsourcing. While we consider similar metrics for optimisation, our automation objective is not concerned with planning and outsourcing, but with initial deployment and ongoing adaptation of how infrastructure resources are used.

2.2 Service Level Agreements

A *Service Level Agreement* (SLA) is a contract between a consumer and a provider of a service regarding its usage and quality ([99] [124] [141] [143]). SLAs are important for maintaining the provider's reputation as consumers compare SLA claims and experience when selecting providers [155]. The cloud management system operated by the provider can then optimally administer the performance and risks of applications by having knowledge of the infrastructure's availability and performance [124][141]. For this reason providers will seek ways of ensuring optimal SLA satisfaction, within the constraints of their resource usage, energy consumption and budget, in order to maintain business. Having insight or prior knowledge of application workloads and data volumes that will be running on their infrastructure is hence beneficial towards tuning their infrastructure for optimising both of these objectives. This is aligned with the goals of Cross Stratum Optimisation (CSO) [92], that of considering both the (1) characteristics and state of the network resources, and (2) the application-level SLA objectives for scaling of cloud-distributed applications.

Providers aim to satisfy the Service Level Objectives (SLOs) of consumers without disrupting their internal operation goals including minimisation of operational costs, power consumption, burnout of equipment and legal issues. Network operators and Internet Service Providers have been using SLAs for assuring client's experience in remote access to services and inter-networking scenarios for many years [124][142]. SLAs are generally used in Cloud Computing (e.g. at Infrastructure as a Service (IaaS) level) for similar purposes [124][155][99]. The demand for rapid delivery of highly available, high performance, networked applications and infrastructure services makes automation support for SLA management a critical requirement. A detailed presentation of the SLA's structure is given later in this section.

However, the combination of CSO and SLA Management is not a straightforward, complimentary relationship, especially when delivering services composed of resources with different owners and concurrent consumers. First, existing and emerging application and infrastructure services are no longer single, isolated, remote servers, virtual machines (VMs) or bandwidth. Disruptions in one resource can have consequences for the overall composite service, such that state information needs to be controllably shared. There are then dependencies across operational strata (application, machine and network) and domains with autonomous administrators, life-cycles and management controllers, which can lead to a break in the autonomy property. Second, application topologies continue to become more complex and heterogeneous as service-oriented design becomes *de facto* (e.g. common, but not officially established), due to the fact that such designs became widely adopted as architectural enterprise pattern for building distributed applications. For these reasons the interdependencies between ap-

plication deployment, usage and network capabilities need to be considered during service planning and provisioning, making the coordination of SLA management more complex.

The efficient management of Service Level Agreements (SLA) is of particular importance [124][48] for Cloud Computing, where exclusively-owned Virtual Machines (VMs) are allocated resources on hosts in a shared physical infrastructure. However, a multi-objective optimization problem for resource allocation arises, where the ability to deliver advertised levels of performance and capacity availability need to be maximised, while minimising energy consumption and resource wastage.

Leading cloud service providers [52][155] (of infrastructure or application services) use SLA management for specifying and maintaining the quality of service (QoS) and availability levels to their customers. An important phase of this process is allocation of resources including initial and runtime placement optimization.

According to Marston et al. [103] SLAs play an important role also in the enterprise environment especially for mitigating risks associated with variability in availability of cloud resources. They often contain a model of guarantees and penalties, which can be used by infrastructure management systems for allocating and optimising the use of datacenter resources.

State of the Art in Modelling SLAs for Cloud Environments

We describe two language specifications for topology orchestration and management of distributed applications, the Unified Service Description Language (USDL) and Topology and Orchestration Specification for Cloud Applications (TOSCA). Each of the two languages has its advantages and disadvantages, in the context of management of distributed applications in cloud environments. We describe both and then we present a novel application specification language from the enhanced combination of the two.

2.2.1 Unified Service Description Language (USDL)

The *Unified Service Description Language (USDL)[126]* is a domain-independent service description semantic language aimed at expressing both technical and business aspects related to services. USDL enables description of capabilities and non-functional characteristics of services, allowing complex interactions to be captured. An extension of USDL, the USDL-SLA [93] enables attaching guaranteed service states and actions in SLAs to service descriptions.

The core concepts of USDL are represented in Figure 2.2, and are part of the `usdl#` RDF (Resource Description Format) [88] namespace. The USDL's vocabulary is used for describing business, operational and technical parameters of services, including information regarding pricing, legal, service

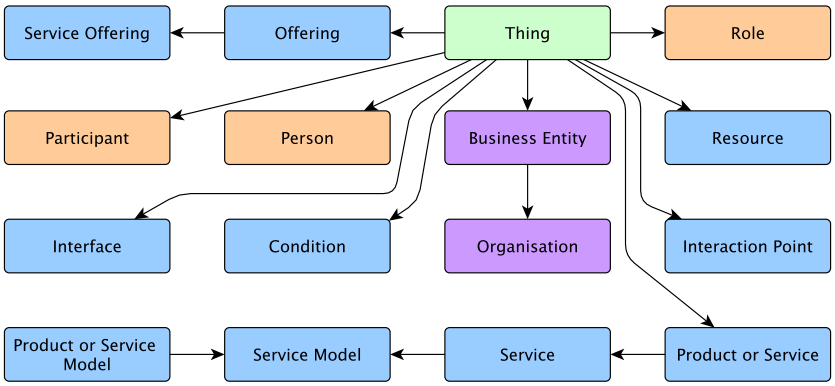


Figure 2.2: USDL-Core Model

provider, interaction methods, and service level agreements. According to the authors of USDL:

> "*The language supports the creation of structured descriptions of services covering most relevant characteristics ranging from technical aspects, to operational ones, socio-economic concerns, or even legal issues.*"

The USDL-Core is the foundation module of USDL and covers four essential aspects:

- Service descriptions
- Description of service offerings
- Business entities involved in the service delivery chain
- Interaction points allowing consumers to contract or trigger the benefits of contracted services

We briefly describe each of the four mentioned aspects in relation to the elements composing the USDL model.

The services are described by defining one or more *usdl:Service* elements, for the purpose of describing the services in a way that it can serve as interfaces between the provider and the consumer. The *usdl:Service* might specify a *usdl:ServiceModel*, which captures common characteristics of similar services (e.g. enterprise resource planning services). The technical means of accessing a *Service* is described by the *usdl:Interface* element, while the actual steps in accessing and performing operations using the service are defined by the *usdl:InteractionPoint* element.

A *Business Entity* will provide a *usdl:ServiceOffering* containing one or more *usdl:Service*s, which are offered to the public or specific customers, usually specifying a price and terms and conditions including service level agreements. The participants in a service offering are defined using *usdl:Participant* elements, each having a specified *usdl:Role*.

The following is an example of USDL description of a service offering.

```
<#Consumer> rdf:type <usdl#Role> , owl:NamedIndividual .
<#Provider> rdf:type <usdl#Role> , owl:NamedIndividual .

<#SAP> rdf:type <usdl#Participant> ,
owl:NamedIndividual ,
foaf:Organization ;
<usdl#hasRole> <#Provider> .
<#Florian_Antonescu> rdf:type <usdl#Participant> ,
owl:NamedIndividual ,
foaf:Person ;
<usdl#hasRole> <#Consumer> .
<#EIS_Load_Balancer_Service>
rdf:type <usdl#Service> ,
owl:NamedIndividual .
<#EIS_Storage_Service> rdf:type <usdl#Service> ,
owl:NamedIndividual .
<#EIS_Worker_Service> rdf:type <usdl#Service> ,
owl:NamedIndividual .
<#EIS_Service_Model> rdf:type <usdl#ServiceModel> ,
owl:NamedIndividual ;
<#has_service> <#EIS_Load_Balancer_Service> ,
<#EIS_Storage_Service> ,
<#EIS_Worker_Service> .
<#EIS_Service_Offering>
rdf:type <usdl#ServiceOffering> ,
owl:NamedIndividual ;
<usdl#hasServiceModel> <#EIS_Service_Model> ;
<usdl#hasParticipant> <#Florian_Antonescu> ,
<#SAP> .
```

The previous USDL code listing described a *Service Offering* for the EIS composite service, between two participants: *SAP* and *Florian Antonescu*. *SAP* is both a *Participant* and an *Organisation* with a *Provider Role*, while *Florian Antonescu* is a *Participant* and a *Person* with a *Consumer Role*. The EIS composite service has three sub-services: *Load Balancer*, *Worker* and *Storage*.

One of the qualities of the semantic description languages such as USDL is their extensibility. This allows enriching the original vocabulary with new terms. Such is the case of USDL-SLA[93] language, developed for representing Service Level Agreements for service. We present next its structure and usage.

Figure 2.3 depicts the main elements composing the USDL-SLA vocabulary. The central entity is the *Service Level Profile*, which is the container

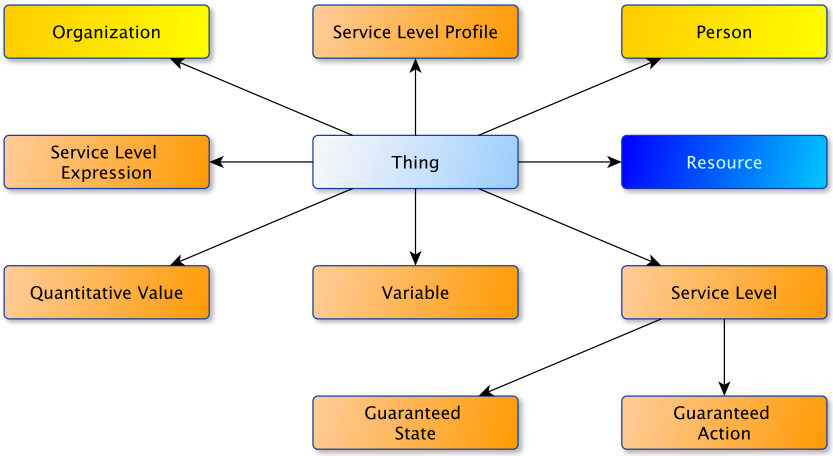


Figure 2.3: USDL-SLA Model

for the descriptions of the service levels of a service. A *Service Level Profile* contains one or more *Service Level*s, which specify single service level objectives as they characterise offered, negotiated or agreed services. Service levels are defined by the parties participating in service provisioning, delivery and consumption and express assertions that are claimed or expected to hold during these activities. There are two types of service levels: *Guaranteed State*s and *Guaranteed Action*s.

A *Guaranteed State* is a particular service level that specifies a single state that must be maintained within the lifetime of any service instance, to which the respective service level profile applies. The *Guaranteed State* is defined by a **Service Level Expression**. The other type of *Service Level* is the *Guaranteed Action*, which specifies an action that must be performed, if and only if during the lifetime of any service instance to which the respective service level profile applies, a specific precondition (*Service Level Expression*) is fulfilled. The example code listing given below shows how guaranteed states and actions are represented.

The *Service Level Expression* specifies an expression that is evaluated in the context of a service level state or action. For this purpose it may reference a set of service level attributes (constants, metrics or variable references) and define relationships between these attributes, e.g. Boolean or arithmetic operands. Typically, it resolves to a Boolean value that indicates whether a guaranteed state is met or whether the precondition to a guaranteed action is fulfilled.

```
<#Average_Users_Per_Worker_Expression>
rdf:type <usdl-sla#ServiceLevelExpression> ,
owl:NamedIndividual ;
<#has_service> <#EIS_Worker_Service> .
```

```
<#Average_Users_Per_Worker_Precondition>
rdf:type <usdl-sla#ServiceLevelExpression> ,
owl:NamedIndividual .

<#EIS_Average_Users_Per_Worker_Guaranteed_Action>
rdf:type <usdl-sla#GuaranteedAction> ,
owl:NamedIndividual ;
<usdl#hasPrecondition>
<#Average_Users_Per_Worker_Precondition> .

<#EIS_Average_Users_Per_Worker_Guaranteed_State>
rdf:type <usdl-sla#GuaranteedState> ,
owl:NamedIndividual ;
<usdl-sla#serviceLevelExpression>
<#Average_Users_Per_Worker_Expression> .

<#EIS_Worker_Service_Level_Profile>
rdf:type <usdl-sla#ServiceLevelProfile> ,
owl:NamedIndividual ;
<usdl-sla#hasServiceLevel>
<#EIS_Average_Users_Per_Worker_Guaranteed_Action> ,
<#EIS_Average_Users_Per_Worker_Guaranteed_State> .

<#EIS_Worker_Service>
<usdl-sla#hasServiceLevelProfile>
<#EIS_Worker_Service_Level_Profile> .
```

The example above contains an SLA represented in USDL-SLA, with a *Service Level Profile* for the *Worker* service, composed of two service levels: a guaranteed action and a guaranteed state. The Guaranteed State has a service level expression, which defines the average number of users per *Worker* service-instance. Similarly, the Guaranteed Action has a precondition expression, which is related to the guaranteed state. Finally, the service level profile is attached to the Worker service using the *hasServiceLevelProfile* property.

It is important to note that this example is not complete in this form, as the details regarding the evaluation of the two service level expressions are missing, and so is the information regarding the metrics used inside the mentioned expressions. In sections 3.2 and 4.2 we describe a complete semantic language for SLA representation, as well as SLA processing.

2.2.2 Topology and Orchestration Specification for Cloud Applications (TOSCA)

Topology and Orchestration Specification for Cloud Applications (TOSCA) is one of the industrially-endorsed standardisation efforts in the area of application cloud topology specification, defined by the Organization for the Advancement of Structured Information Standards (OASIS) [123].

TOSCA is used for defining both the service components of distributed applications (topology), as well as the service management interfaces (plans). Service orchestration is realised by describing the interactions between the services using workflows, called plans. TOSCA's objective is to ensure the semi-automatic creation and management of application layer services, while guaranteeing applications' portability across various cloud implementations.

The purpose of TOSCA is to enable the creation of declarative descriptions of the components belonging to distributed applications, in terms of services, infrastructure requirements, and interactions between services. It effectively combines both declarative and imperative approaches. A declarative model can be defined as describing the desired end-state, while providing the means for adjusting the state until the desired end-state is achieved. This is in contrast to the imperative approach, were the step-by-step instructions are provided for reaching the desired end-state.

TOSCA's model is composed of four top-level entities: nodes, services, relationships, and artefacts as described below. Figure 2.4 graphically depicts the logical relations between TOSCA model's components.

Nodes represent components of an application or service and its properties. Example nodes include Infrastructure (compute, network, storage), Platform (operating systems, virtual machines, database), etc. TOSCA nodes include operations as management functions for the nodes, such as deploy, start, stop, connect, etc. The nodes export their dependencies on other nodes as *requirements* and *capabilities*.

Relationships represent the logical relations between nodes, e.g. "hosted on", "connects to". They describe the valid *source* and *target* nodes that they are designed to couple, and have their own *properties* and *constraints*.

Service templates group the nodes and relationships that make up a service's topology. This enables modelling of sub-topologies, by means of *composition* of applications from one or more service templates, and *substitution* of abstract node types with service templates of the same type.

TOSCA is represented using an XML encoding, as shown below. A TOSCA container would process the XML representation as runtime, leading the execution of its management plan and to the instantiation of the distributed application.

As mentioned earlier, TOSCA relies on service management platforms for interpreting the service templates and associated management plan and for converting them into infrastructure operations. Such a cloud management

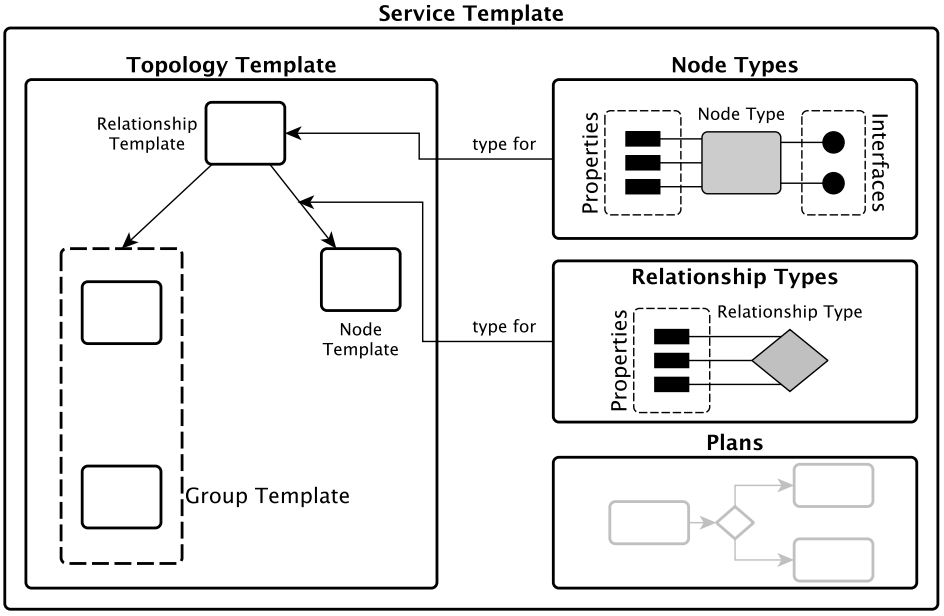


Figure 2.4: Relation between TOSCA's model's entities

```
<ServiceTempate ...>
<Extensions/>?
<Import />*
<Types/>? (<TopologyTemplate/> |
<TopologyTemplateReference/>)?
<NodeTypes/>?
<RelationshipTypes/>?
<Plans/>?
</ServiceTemplate>
```

Example 1: Skeleton of a TOSCA XML serialisation format

platform is IBM SmartCloud Orchestrator [77], which is based on OpenStack and uses TOSCA as a management input. It also uses Open Service for Lifecycle Collaboration (OSLC) [122] standard for creating, updating, retrieving, and linking to lifecycle resources.

TOSCA plans and service topologies can be created with a variety of tools, such as Vnomic's Application Landscape Designer [149], which enables provisioning of complex multi-tier enterprise application workloads from scratch or based on existing deployments onto EC2[7], VMware[153], OpenStack[29], or other cloud platforms.

Figure 2.5 presents the TOSCA description of the SugarCRM [2] customer relationship management (CRM) application. The application is composed of two main subsystems: the SugarCRM PHP application and the

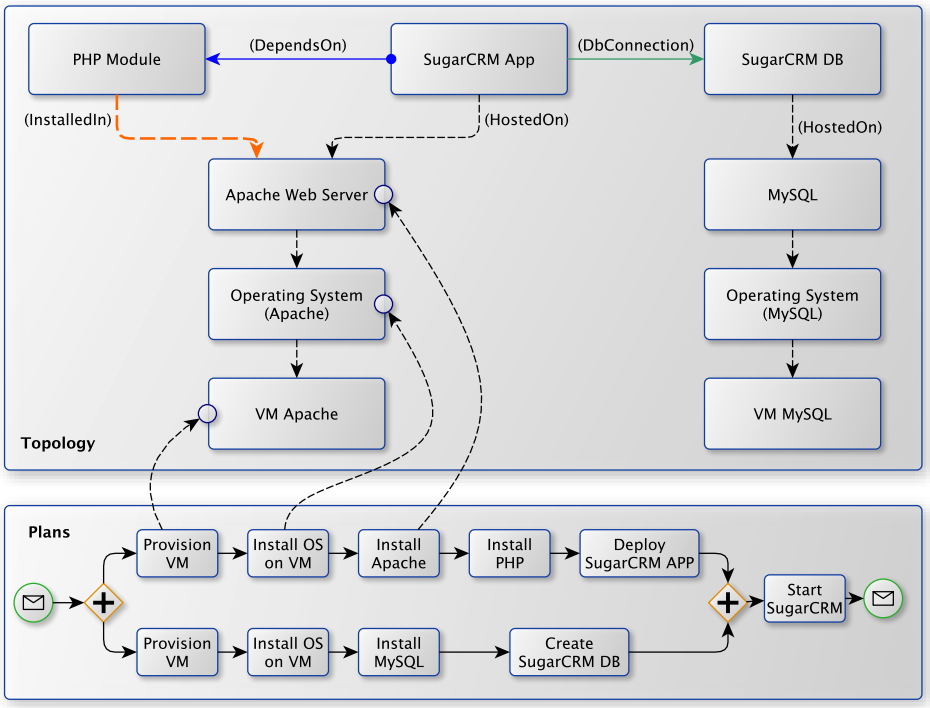


Figure 2.5: TOSCA description of SugarCMR application

MySQL [121] relational database. The elements presented in Fig. 2.5 correspond to the components described in the CSAR (Cloud Service ARchive) [137] TOSCA interoperability specification.

The TOSCA CSAR specification of SugarCRM describes the requirements for deploying and running the application. The topology specifies that the *SugarCRM* application module depends on the *PHP Module* and it is hosted on the *Apache Web Server* component, which requires an *Operating System* and a *Virtual Machine*. Similarly, the *MySQL database* component is hosted on the *MySQL* relational management database system, which runs inside the *Operating System*, which is hosted inside a *Virtual Machine*.

Corresponding to the topology components are the TOSCA plans, which specify how the various components are instantiated, deployed and started. There are two parallel plans describing the instantiation of each VM, installation of the operating system and then for the SugarCRM PHP module describing the installation of Apache Web Server, PHP and the deployment of the SugarCRM application module. For the MySQL database component, the plans describe the installation of the MySQL server application and creating the SugarCRM database schema. Once the two parallel plans have completed, the SugarCRM application can be started.

2.3 Distributed Enterprise Applications

This section discusses relevant work in the areas of distributed cloud-based applications and approaches to their management and topology orchestration.

We classify four models of distributed cloud-based applications found in literature ([71] [94] [154]) and practice.

- *Redundancy-based* models replicate functionality in different locations for the purpose of fault mitigation or load-balancing.
- *Capability-based* models arise when individual components and services of a single application have different compute and storage requirements that are not likely to be found in one location.
- *Workflow-based* models represent the necessity of scheduling individual components and services following a predefined control-flow.
- *Concurrency-based*, where there is a performance gain in job decomposition, parallel processing and result aggregation.

For each of these models there are a set of common challenges that start with the initial specification of how application components should be distributed, interconnected, monitored, maintained and eventually terminated. The challenges extend to the selection of distributed resources and the coordination of operations performed on the runtime of the service. There are currently no best practices established for these challenges nor evaluation models for solutions.

Distributed applications have special requirements as they need to be dynamically instantiated, configured, monitored and scaled. Rapid provisioning must be considered, as well as reporting, metering, and audit. Service level agreements are usually involved for guaranteeing system performance and runtime behaviour and also for creating base audits.

Cloud-Distributed Enterprise Applications [154] are component-based, distributed, scalable, and complex business applications, usually mission-critical. Commonly, they are multi-user applications handling large datasets in a parallel and/or distributed manner, and their purpose is to solve specific business problems in a robust way. Often these applications are running in managed computing environments, such as datacenters [33]. Enterprise Resource Planning (ERP)[94] applications are a type of distributed enterprise applications, which provide an integrated view of core business processes, using multiple database systems.

Topology Orchestration and Cloud Management

To our knowledge, there is currently one industrially-endorsed standardisation effort for specification of application topologies, known as the *Topology*

and Orchestration Specification for Cloud Applications (TOSCA) [115]. We hence use this as our starting point for exploring the anatomy of enterprise applications and service topologies. TOSCA is a language used for defining both the service components of distributed applications (topology), as well as the service management interfaces (plans). Service orchestration is realized by describing the interactions between services using called plans (or workflows). In this thesis we extend the TOSCA concepts by adding SLA metrics and scaling rules for both the services and the associated infrastructure (compute, storage, network) resources.

For the purpose of service specification, the *Unified Service Description Language (USDL)[126]*, is a domain-independent service description language aimed at expressing both technical and business aspects. USDL enables description of capabilities and non-functional characteristics of services, allowing complex interactions to be captured. An extension of USDL, USDL-SLA [93], enables attaching guaranteed service states and actions in SLAs to service descriptions. We extend the USDL capabilities with aspects related to network service management and more general service monitoring.

Much of the initial problem identification and definition was carried out during the *Generalised Architecture for Infrastructure Services (GEYSERS)* European research project[63], which aimed at delivering an architectural blueprint and platform for seamless and coordinated provisioning of connection-oriented networking (primarily optical) and IT (application, compute and storage) resources as infrastructure services.

Various cloud management platforms, including OpenNebula, OpenStack, CloudStack [29], are possible candidates for realising dynamic application topologies. The mentioned platforms enable on-demand virtual resource provisioning, support multiple virtualization technologies, facilitate monitoring of resource performance by exposing various metrics about the managed infrastructure and rapid resource provisioning through service catalogs that contain pre-configured virtual machines, along with their hosting requirements in terms of computing and storage. While these technical capabilities are a baseline for enabling dynamic application topologies, the related work 2.3 section shows that there is already progress beyond these basic capabilities.

Topology and Orchestration Specification for Cloud Applications OASIS Standard

Topology and Orchestration Specification for Cloud Applications (TOSCA) is one of the industrially-endorsed standardisation efforts in the area of application cloud topology specification, defined by OASIS [123].

TOSCA is used for defining both the service components of distributed applications (topology), as well as the service management interfaces (plans). Service orchestration is realised by describing the interactions between the

services using workflows, called plans. TOSCA's objective is to ensure the semi-automatic creation and management of application layer services, while guaranteeing applications' portability across various cloud implementations.

According to its designers[123],

> "the TOSCA language introduces a grammar for describing service templates by means of Topology Templates and plans. The focus is on design time aspects, i.e. the description of services to ensure their exchange. Runtime aspects are addressed by providing a container for specifying models of plans which support the management of instances of services. The language provides an extension mechanism that can be used to extend the definitions with additional vendor-specific or domain-specific information."

The purpose of TOSCA is to declaratively describe the components of distributed applications, in terms of services, infrastructure requirements, and interactions between services. It effectively combines both declarative and imperative approaches. A declarative model can be defined as describing the desired end-state, while providing the means for adjusting the state until the desired end-state is achieved. This is in contrast to the imperative approach, were the step-by-step instructions are provided for reaching the desired end-state.

Related Work on Topology Management of Cloud-Distributed Applications

Liu et. al. [97] describe a data-centric approach to cloud orchestration by modeling the resources as data structures, which are then queried and updated using transactional semantics: views, constraints, actions, stored procedures and transactions. We follow similar design principles, including the declarative approach to orchestration description and separation of logical and physical aspects. However, their novel data-centric approach introduces a new computational model for cloud management, whereas we seek to enhance the capabilities of existing management systems where possible.

Juve and Deelman [82] present a system for cloud infrastructure management with a similar approach to the one envisioned by us in Chapter 3. They only focus on the initial deployment of distributed applications and not on dynamic scaling of virtual resources and response to adaptation triggers. They also do not consider the impact and provisioning of network resources.

Malawski et al. [101] consider how to optimally provision for multiple a priori known workflow ensembles with varying deadline and budget constraints. However, they do not consider the network as a critical part of the topology orchestration and focus only on the workflow model rather than

the structural aspects of the application topology. They are also considering a fixed pool of VMs, which does not change during the workflow runtime. In contrast, we propose scaling the number of virtual machines (VMs) based on the service load defined through SLA guaranteed actions, applicable to all distribution models.

Service Level Agreement Models

In the context of SLAs, we identify three types of messages, exchanged between the provider and the consumer of the cloud services: template, objective and agreement. We provide the definition of each message type below.

Service Level Agreement Template: is a statement or advertisement from providers about their guaranteed service levels. Providers may deliver one or more SLA templates representing varying service level classes. For example, Amazon EC2 [7] offers various service levels depending on the type of selected VM. Rackspace [45] offers two services levels: Managed (for rapid, on-demand deployment and response) and Intensive (for highly-customized, proactive response on dedicated servers).

Service Level Objective: is a message from a consumer defining the type and level of service they require. For example, a cloud consumer sends a message containing a service-level objective to a cloud provider declaring the size and quality of Virtual Infrastructure (VI) required for her/his application. As an example, Amazon EC2 users can select different sizes of EC2 instances (small, large, xlarge), each representing a different SLO.

Service Level Agreement (SLA): is an agreement that the consumer is prepared to accept. When the provider also accepts to enter the contract it is represented as SLA*, which is equivalent to a signed SLA.

The SLA negotiation phase is described as follows. An SLA occurs when a service consumer (agrees) accepts specific service parameters on entering a contract with a service operator or provider. In composite IaaS the roles of provider and consumer change up and down the service delivery stack. An entity in the role consumer specifies *Service Level Objectives*, before or after the availability and capability of services and service providers are known. *Provider*s declare their service capabilities and quality guarantees in the form of an advertisement, known as a *Service Level Agreement Template*. Such templates act as the baseline for contractual agreement with customers, potentially in different classes. *Service Level Objective*s, *Template*s and *Agreement*s (SLAs) have basic requirements for structure and content and can hence be represented using the same information model (we then use the general term "SLA"):

- *Parties* include individuals, organizations and roles involved in the agreement. The roles are typically consumer/customer and provider/-

operator, but can also include a third-party broker, an intermediate actor in the SLA management process.

- *Functional description* of the service's purpose and capabilities. In the case of a technical service the functional description refers to the set of operations, methods and parameters. For example, the Web-Services Description Language (WSDL) [41] provides a standard specification for SOAP-based [72] web services. In the case of a network or connectivity service the functional description refers to path selection and bandwidth provisioning.

- *Costs* to the consumer for receiving the service. The units for costs are defined by relating financial costs (for hardware, maintenance) of the provider to utility functions (e.g. energy, real-estate) of the resources used for providing the service. For example costs can be defined per requests, per volume of storage used, per user or on a fixed-term or unlimited basis.

- *Guarantee or Quality of Service (QoS)* terms are the non-functional properties of the service. These properties include availability, performance, response time, reliability and security.

- *Recovery* terms define what types of consumer-visible incidents the service can recover from using an event-action mapping. A recovery term can also define compensation, stating what the consumer can rightfully demand from the service provider in return, should the functional or guarantee terms not be fulfilled.

SLA negotiations could describe the agreement of parameters such as response time, throughput, network bandwidth, in the context of given confidence values and prices. For example, the provider might offer a range of possible average response times (e.g. from 100 to 500ms) for a corresponding average arrival rate of requests (e.g. from 10 to 100 requests per sec.), for several confidence percentages (e.g. 95%, 97%, 99% and 99.5%). Each combination of SLA parameters corresponds to a price and it is the result of the negotiation process that a concrete value will be selected for each SLA parameter.

Figure 2.6 presents the structure of an SLA agreement, as it is defined by the TeleManagement Forum (TMF) and The Open Group standardization organizations in [143]. It is composed of three top-level elements: *Provider*, *Consumer*, and *Level*. The meeting of each element is given below.

Provider element specifies the name of provider and service being delivered (e.g. Amazon EC2). *Provider-Type* contains the role or type of service provider (e.g. infrastructure provider). *Provider-Details* specifies details including name, contact information and other properties of the entity that

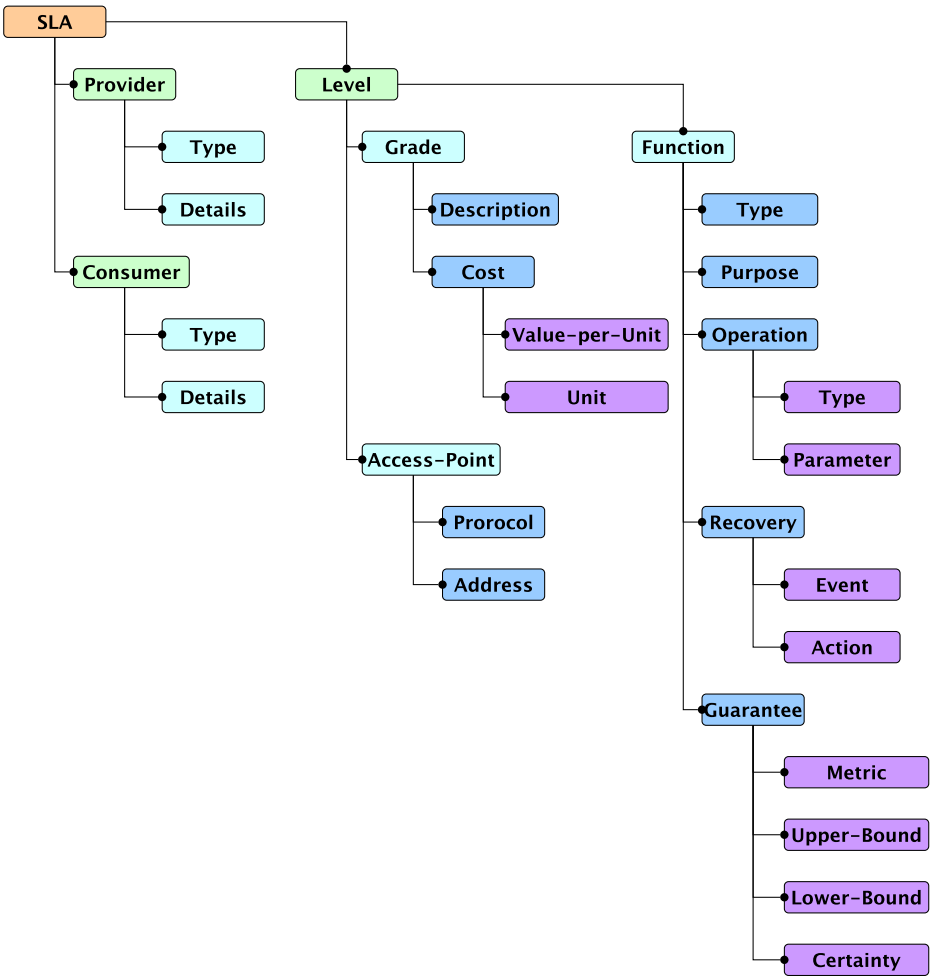


Figure 2.6: TMF SLA Model

might be necessary for legal purposes - these details are included for completeness but are not assumed to have direct consequence on the technical handling of SLAs.

The *Consumer* element specifies the name of the consumer (e.g. SAP) - SLAs are bilateral. *Consumer Type* contains the role of the consumer (e.g. Platform Operator). *Consumer Details* term holds contact details of the consumer (similar to provider).

The *Level* element provides a descriptor for distinguishing different levels of quality associated with a SLA or Template. It further contains the following elements: *Grade*, *Access Point* and *Function*.

The *Grade* element holds the grade used to categorise the SLA or Template. Examples include Gold, Silver, Bronze or Priority, Intermediate, Basic. This can however be defined for the environment in question. The

Description element is a textual description of the grade categorisation for the SLA or Template. The *Cost* element contains the cost associated with the grade of service; it also specifies a *Value-Per-Unit* element with the numeric value of the cost; and a *Unit* element with theunit of cost (e.g. Currency per month, Currency per transaction, Currency per GB).

The *Access Point* element specifies a technical access point or URL to a provider of the service at the given quality level. It also specifies a *Protocol* element with the communications protocol used for interacting with the provider (e.g. HTTPS, XML-RPC); and an *Address* element holding a unique reference to a logical or physical service access point for the service.

The *Function* element details the capabilities provided by the service at a given quality level. It further contains the following elements: *Type*, *Purpose*, *Operation*, *Guarantee* and *Recovery*. The *Type* element specifies the nature of the function (e.g. connectivity creation, storage volume creation, application's service instance creation). The *Purpose* element describes the purpose for the function (e.g. provisioning). The *Operation* element holds the list of operations that can be involved to execute the function (e.g. create, read, update, delete). Each operation further details its *Operation-Type* (e.g. the data or item returned by the operation) and *parameters* (the set of input data for the operation). The *Guarantee* element holds information about the set of guarantees at a given service level (e.g. performance, reliability, security), defined by the following terms: *Metric-per-Guarantee* (the metric used to measure the guarantee, such as response time, mean time to failure, mean time to recovery, cryptographic suite, etc.), *Upper-Bound* (the upper value for the guarantee), *Lower-Bound* (the lower value for the guarantee) and the *Certainty* (the degree of certainty the provider promises). The *Recovery* element contains a specification of actions given specific classes of incidents, detailed by their *Event*, the triggers that represent incidents (e.g. timeout, illegal access) and thier *Action* (the type of action taken if an incident or class of event occurs, such as restart, re-allocate, scale-out, etc.).

Related Work on Service Level Agreement Management Systems

There are multiple frameworks and tools, which allow management of distributed applications composed of several (interdependent) services. One example is OpenNebula AppFlow [116], which allows definition and management of applications composed of services mapped to one or multiple VMs, enabling automatic management of VMs' elasticity.

Zhang and Song [159] propose an architecture for SLA Management taking into account the five phases of SLA management process. They also consider different roles (service provider, service broker, SLA Manager, service client) and implement this via two modules (offline and runtime). The offline module handles SLA registrations and client queries, template storage. The runtime module handles monitoring, logging and controlling (detects

SLA violations, possible admission control). As their proposed architecture is single-domain oriented, its implementation is monolithic and there are no events utilized. In our proposed architecture feedback is incorporated from more than one source (e.g. different providers and clients) taken into consideration the distributed nature of the architecture.

Theilmann, Happe et al. [141] propose an architecture for hierarchical SLA Management in service oriented environments by considering the entire lifecycle of SLAs and services. Their architecture is also event-based and utilizes two types of models: SLAs for the communication within and among SLA Managers, as well with external providers and a model for describing creation of new service instances based on SLAs. Particular to this architecture is its adaptation to software service environments and not infrastructure services. Furthermore they assume exclusive control of the available resources.

There are multiple research projects exploring the use of SLAs in the context of cloud computing, especially in relation to networking services (e.g. Network-as-a-Service). These projects build on the wealth of cloud infrastructure management platforms, such as OpenNebula, OpenStack and others. We will briefly present the following projects: GEYSERS (Generalized Architecture for Dynamic Infrastructure Services), Mobile Cloud Networking (MCN), SLA@SOI and Optimis - for their pioneering role in defining frameworks for convergent management of SLAs in complex cloud computing environments.

SLA@SOI

The research project **SLA@SOI** [1] provides a major milestone for the further evolution towards a service-oriented economy, where IT-based services can be flexibly traded as economic goods, i.e. under well defined and dependable conditions and with clearly associated costs. Eventually, allowing the creation of dynamic value networks that can be flexibly instantiated, thus driving innovation and competitiveness. It provided a reference SLA management architecture for cloud environments. Also, the project contributed to the evolution of OCCI (Open Cloud Computing Interface) [57] and USDL (Universal Service Description Language) [126] cloud standards.

SLA@SOI provides 3 major benefits to the provisioning of services:

- Predictability and Dependability: The quality characteristics of service can be predicted and enforced at run-time.
- Transparent SLA management: Service level agreements (SLAs) defining the exact conditions under which services are provided/consumed can be transparently managed across the whole business and IT stack.
- Automation: The whole process of negotiating SLAs and provisioning, delivery and monitoring of services will be automated allowing for

highly dynamic and scalable service consumption.

GEYSERS

The main objective of the project **GEYSERS** ([52, 17, 21, 18]) (2010-2013) was to define, develop and validate an end-to-end network architecture, based on extending standard ones, in order to enable provisioning of virtualised networks using optical networks for cloud computing operations. The goal of the project was to create a new planning, provisioning and (ultimately) a business framework for infrastructure providers and operators. To do so, GEYSERS has defined and implemented an architecture [144], capable of provisioning Optical Network and IT resources for end-to-end service delivery, while using semantic service descriptions and SLAs for ensuring the automated provisioning and scaling of virtual infrastructures. GEYSERS proposed a vision under an evolutionary approach that follows a network-centric and bottom-up strategy.

This vision was based on partitioning the photonic network infrastructure for creating specific virtual infrastructures, composed by optical network and IT resources at the edges. Each virtual infrastructure was to be controlled by an enhanced Network Control Plane capable for the provisioning of Optical Network dynamic services coupled with IT resources.

The GEYSERS research effort created the Service Middleware Layer (SML) [52] framework for exploring the use of SLAs for automated management of distributed VM-bounded services. The SML represents a convergence layer for coordinating IT resources on behalf of various distributed applications. We shortly describe the SML, including its consumption of SLAs during the management of applications and cloud landscape.

The GEYSERS architecture [52] consists of three actors, each performing a different functional role: provider, broker and operator.

The **provider** usually owns the virtualizable physical resources (e.g. networks and servers). A provider allows access to its resources by installing the Lower Logical Infrastructure and Composition Layer (LICL) component, which provides services such as discovery of the available hardware, abstracting the access to the virtualisation controllers and mapping physical resources to logical pools of infrastructure resources.

The **broker** component interfaces between multiple providers and operators. This functionality is implemented by the Virtual Infrastructure Provider (VIP), which is managing the Upper LICL component of GEYSERS. The VIP aggregates multiple resources belonging to different providers into virtual resource pools.

The **operator** usually represents the application/user of the composed virtual resources. An operator expresses the expectations and requirements of distributed applications through SLAs. SLAs [18] may contain both (a) consumer-specified service scaling rules and (b) composed application-wide

monitoring conditions.

In a typical scenario, the Lower LICL discovers its physical resources and informs the broker accordingly. The broker maintains a list of aggregated resources per provider (e.g. total number of CPUs, memory, disks). When asking for resources, the operator receives information about the maximum resources available at each provider and the maximum available bandwidth between the providers (the possibly virtualized optical networks). The operator selects the corresponding provider(s), SLAs are agreed and the VIP manages the selected resources such that the SLA is maintained with regards to the agreed amount of virtual resources.

Mobile Cloud Networking

Mobile Cloud Networking (**MCN**)[106], 2012-2015, EU research project has the goal to "define and evaluate Europe's vision of mobile cloud computing, by enabling European Telco industry to take and sustain leadership in mobile cloud computing and thus a fundamental pillar of the Future Internet" [106].

The motivations of the Mobile Cloud Networking project are to:

- Extend the Concept of Cloud Computing beyond data centres towards the Mobile End-User by using SLAs for managing both the infrastructure resources as well as the offered services
- Provide one atomic, on-demand, elastic, and pay-as-you-go service for Mobile Network + Computing + Storage

For reaching these goals, MCN relies on a SLA management platform for(1) rating, charging and billing (RCB) purposes, (2) for inter-provider negotiation of NaaS and IaaS service offerings, and (3) for customer-to-business SLA negotiations.

OPTIMIS

Project **OPTIMIS** (Optimized Infrastructure Services) [120] (2010-2013) aimed at enabling organizations to automatically externalize services and applications to trustworthy and auditable cloud providers in the hybrid model[108]. Consequently, OPTIMIS supported and facilitated an ecosystem of providers and consumers that will benefit from the optimal operation of services and infrastructures, by covering the full lifecycle of services and their interactions. OPTIMIS provided a SLA model for multi-round negotiating and creation of agreements between providers, brokers and cloud-customers.

2.4 Data Analysis Mechanisms for SLA-Driven Management of Cloud Resources

The SLA-based management of distributed services and applications is equivalent to a multi-objective constrained problem, where various guarantees such as service response time and availability per customer have to be managed together with operational objectives such as cost, energy consumption and utilization. This leads to the requirement of using complex mechanisms for data analysis. In this section we present how the various models required for SLA management of cloud resources are calculated and which data analysis mechanisms are used.

In a distributed, multi-provider, multi-customer, cloud environment there are different categories and classes of infrastructure and application services running in parallel and contending for the same physical resources. Understanding *correlation*s between key performance indicators of dependent services is hence important in effective SLA management for understanding the relations between the distributed services and the impact on services' performance guarantees. Identifying services with similar behaviours and access patterns provides hints that they (the services) should share either the same or different resources and configurations, depending on the operational circumstances. Discovering statistical correlation between services' performance indicators (e.g. average execution time) helps better understanding the causes of performance bottlenecks (leading to SLA violations) in cloud applications [30], and thus understanding SLA violation causes. By using correlation information it also becomes possible to (1) refine service monitoring, (2) perform SLA levels health checking (compare current measured values against the maximum/minimum ones defined in the SLAs), and (3) improve scaling of cloud applications.

By applying *estimation models* on groups of performance indicators with strong statistical correlations, it becomes possible to express critical SLA parameters as function of multiple measurable service indicators (as described in Chapter 5), thus (1) describing analytical SLA dependencies and (2) allowing automated service management based on expressed dependencies between the distributed services.

As it is often the case that cloud applications exhibit predictable and repeatable patterns (hourly, daily, monthly) in their resource utilisation levels, caused by the execution of repeatable workloads (e.g. with hourly, daily, weekly patterns), a Cloud Management System (CMS) can benefit from detecting such repeatable patterns by combining this information with *prediction mechanisms* in order to estimate the near-term utilisation level of both software and physical resources, and then to optimise the allocation of resources based on the SLAs.

2.4.1 Data Prediction Mechanisms

We define the SLA Cloud Management Optimization (SLA-CMO) problem [19] [147] [18] [16] as: improving the allocation of datacenter's computing resources by dynamically changing the number of VMs allocated to cloud services, so that SLA-defined performance requirements are met under variable workload conditions. It is therefore essential to predict as accurately as possible this variable workload parameters (such as the number of users, transaction rates, rate of arrival, etc.).

Solving the SLA-CMO problem depends directly on having a reliable source of monitoring information reflecting the state (e.g. number of allocated VMs, system's throughput, requests arrival rate, response time) of the managed distributed systems. The management system will then take corrective actions for ensuring that the SLA contracts guaranteeing the system's performance are not violated. However, the actions' effects on the underlying system might be delayed, creating a time window during which the system might be in an undesirable state (e.g. under-performing). Such SLA violations can be avoided [11] if the SLA management system can timely predict the trend and variation of the critical monitoring system's parameters, allowing it to actuate (e.g. scale-out - increasing the number of VMs allocated to a distributed service) at such a time moment that the newly added virtual resources (e.g. VMs) become active just as the workload would surpass the previous capacity of the cloud system.

The prediction problem can be defined as finding the next n values of a dependent system's parameter P, using p previous values of one or more predictors as shown in Equation 2.1

$$(P_{t+1}, P_{t+2}, ..., P_{t+n}) = f(X_t, X_{t-1}, ..., X_{t-p}) \tag{2.1}$$

where t is the current time moment, X_t is the value of the predicting vector at time t, and f is a function.

The above definition is true for parameters whose values are depending on other system parameters, for example, the average execution time of a distributed application depends on the number of VM instances allocated to the service, the workload's arrival rate and the application's average occupancy (as defined by Little's law [96]).

Another class of parameters are the *independent* [139] ones, whose values are not determined by other parameters. In an experiment the *independent* variable it is changed or manipulated by the researcher, while the *dependent* variable is the response that is measured. An independent variable is the presumed cause, whereas the dependent variable is the presumed effect. An example of such system parameter is the workload arrival rate, which is determined only by factors external to the system. Such independent parameters can also be predicted, by observing certain patterns in the distribution of data and then recognising when the data flow will start following

a certain learned pattern. This can be formulated as a dependency of the current and future values of the parameter on the previous own values, as shown in Equation 2.2, where the mathematical terms are the same as the ones from Equation 2.1.

$$(P_{t+1}, P_{t+2}, ..., P_{t+n}) = f(P_t, P_{t-1}, ..., P_{t-p}) \tag{2.2}$$

Figure 2.7 shows a snapshot of such an independent parameter - the request arrival rate of a transaction processing application.

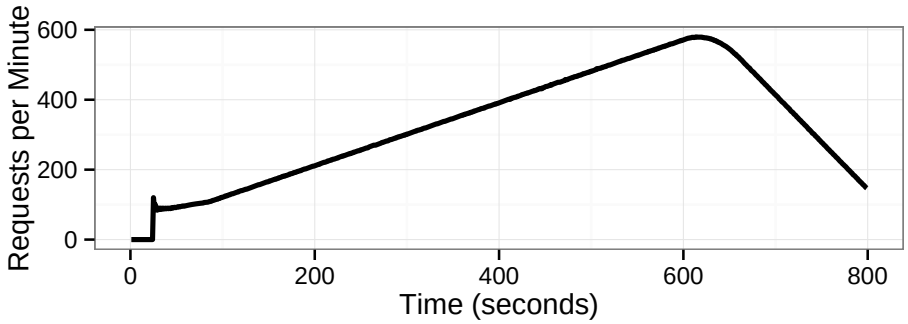


Figure 2.7: Requests Arrival Rate

When identifying repeatable patterns in temporal data series, it is important to pay attention to the statistical parameters of the data series, such as variance, mean, trend, etc. For example, a large variance in the data could hide a repeating pattern, therefore filtering the data series by applying a moving average or kernel smoothing [47] could expose the underlying trend of the data. By analysing the trend of the temporal data series, two types of variations could be observed: linear and non-linear. We shortly present in the following sections the mathematical fundamentals for linear regression, error calculation and non-linear regression.

2.4.2 Statistical Linear Models

A linear statistical model (or linear regression model) [128] represents the mathematical relation between a dependent variable Y and one or more predictor variables X, as shown in Equation 2.3, where lower case y is a single value from the time series Y. (Please note that capital letters represent the time series/vectors, while the lower-case letters represent single values of those vectors.)

$$y = x_0 + \beta x + \epsilon \tag{2.3}$$

y is the dependent value, x_0 is the intercept (free term), β is a transposed vector of scalar coefficients, X is a vector of independent variables, and ϵ is the error term [128].

If Y is a scalar variable and X is also a scalar variable, then the regression is called *simple*. If X is a vector of independent variables, the regression is called *multiple*. If both X and Y are vectors then the statistical model is called *general linear model*.

When the modelled variable is *independent* (e.g. rate of which requests are arriving at a service) and it is represented as a time series, we are dealing with *autoregression*, as shown in Equation 2.4.

$$y_t = y_0 + \beta y_{t-1} + \epsilon \tag{2.4}$$

y_t and y_{t-1} are values of variable Y at time t and $t-1$ respectively, y_0 is the model's intercept term (e.g. value ot Y at $t = 0$), ϵ is the error term, and β is the regression coefficient [128].

By assuming that the model's accuracy is *acceptable* over a time window of length p, it is possible to predict the next n values of Y by applying n times the formula in Equation 2.4, starting with Y_t. A linear model's accuracy is defined in terms of errors. In Section 4.3 we show an example of how auto-regression can be used for predicting the time series shown in Figure 2.7.

2.4.3 Statistical Non-Linear Models

Non-linear regression is a form of regression in which the mathematical relation describing the dependency between the predictors and the dependent variable is a non-linear combination of parameters, for example exponential addition. Exponential smoothing is an example of non-linear modelling, where a time series is *smoothed* by approximating its terms with computations of an exponential function.

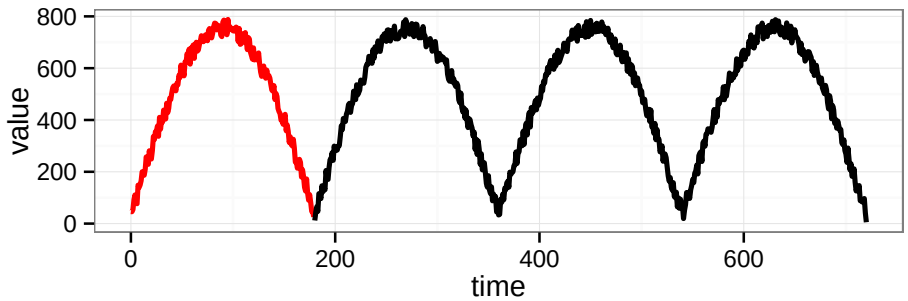


Figure 2.8: Sample Time Series with a Periodic Pattern

While non-linear models are more suitable to represent complex patterns in time series, they also introduce additional complexity as their output usually depends on correctly identifying the underlying signal's period, needed by some prediction algorithms for decomposing the signal into its seasonal,

trend and noise components. Also, some non-linear prediction algorithms, such as Holt-Winters [74] require having a time series with a length of at least two periods.

Let us present a simple example. Figure 2.8 shows a time series containing a repetitive pattern (closely resembling the more complex patterns usually found in business-critical enterprise applications), with some added noise. The first period is coloured red. The data was generated by calculating the absolute value of function *sinus* multiplied with a constant factor, as shown in Equation 2.5.

$$Y = 1 + 720 \left| sin(x\frac{\pi}{180}) \right| + \epsilon, x \in (1...720) \tag{2.5}$$

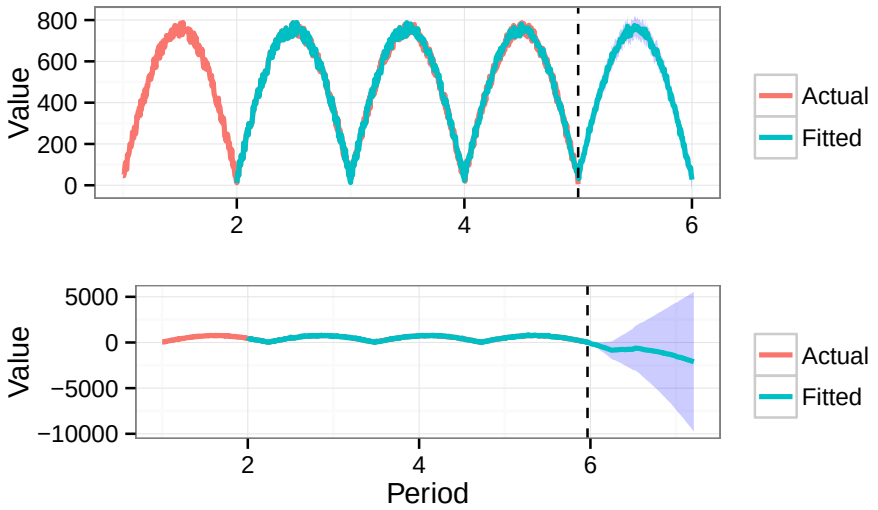


Figure 2.9: Holt-Winters Prediction. Top: Signal's Period:180. Bottom: Signal's Period: 145

There are multiple ways of extracting the period of a signal, for example by using the autocorrelation function as in [11], or by calculating the estimated spectral density of the time series using the periodogram [32] and then converting the frequency with the largest spectral density into the signal's period. Naturally, the period of Y is equal to 180 (e.g. seconds), as the function *sinus* has a period of 2π and $|sin(x)|$ has a period equal to π.

Figure 2.9 shows the prediction output for the time series given above, by applying the Holt-Winters algorithm, first with the correct period of the time series (180), and then with an approximative value for the period (145). This shows the importance of correctly identifying the signal's period before attempting to predict it, which can be difficult in practice. The lower image shows a bad prediction caused by the incorrect detection of the trend of the time series, caused by the shorter value of the period - the signal being

identified as having 6 periods instead of 5.

The non-linear prediction models are suited for forecasting time series with seasonal variations, such as hourly, daily or monthly. The predictions can then be used for optimising processes with longer time horizons [148], such as the allocation of physical computing resources [19]. For the work presented in Section 4.4 we will use a combination of linear models and error estimation, as the time horizon of the prediction is usually short, in the range of tens of seconds.

2.4.4 Errors in Statistical Models

The linear model calculates for each value x_t of the independent variable X a corresponding approximation value $\hat{X}_t$ according to equation 2.4. The difference between X_t and $\hat{X}_t$ represents the regression error. By applying the regression over a time series X, a series of error values called *residuals* are produced. When the errors are calculated between the predicted values and the actual values of the independent variable, the errors are then called *out of sample* errors. Residuals are important for calculating the accuracy of modelling, while the out of sample errors are important for calculating the prediction's accuracy.

We present two metrics for quantifying the regression errors: Root Mean Standard Deviation (or Error) and Mean Absolute Percentage Error.

The Root Mean Standard Deviation ($RMSD$) is a scale-dependent measure (dependent on the variable's maximum value) of estimation error, equal to the square root of the mean square error, as shown in Equation 2.6.

$$RMSD = \sqrt{\frac{\sum_{t=1}^{n}(\hat{Y}_t - Y_t)^2}{n}} \tag{2.6}$$

The Mean Absolute Percentage Error ($MAPE$) is calculated as the mean of the error's modulus, as shown in Equation 2.7

$$MAPE = \frac{\sum_{t=1}^{n}|\hat{Y}_t - Y_t|}{n} \tag{2.7}$$

and it expresses accuracy as percentage. We will use $MAPE$ error for calculating the regression's accuracy and for deciding whether the prediction can be used for SLA-based scaling in Section 4.4.

Correlation and Autocorrelation Mechanisms

The correlation coefficient r is defined according to Equation 2.8, where x and y are sample values, $\bar{x}$ and $\bar{y}$ are variables' average value, n is the number of monitoring samples in the considered time window, σ_x and σ_y are the standard deviation of each variable. We used the correlation coefficient

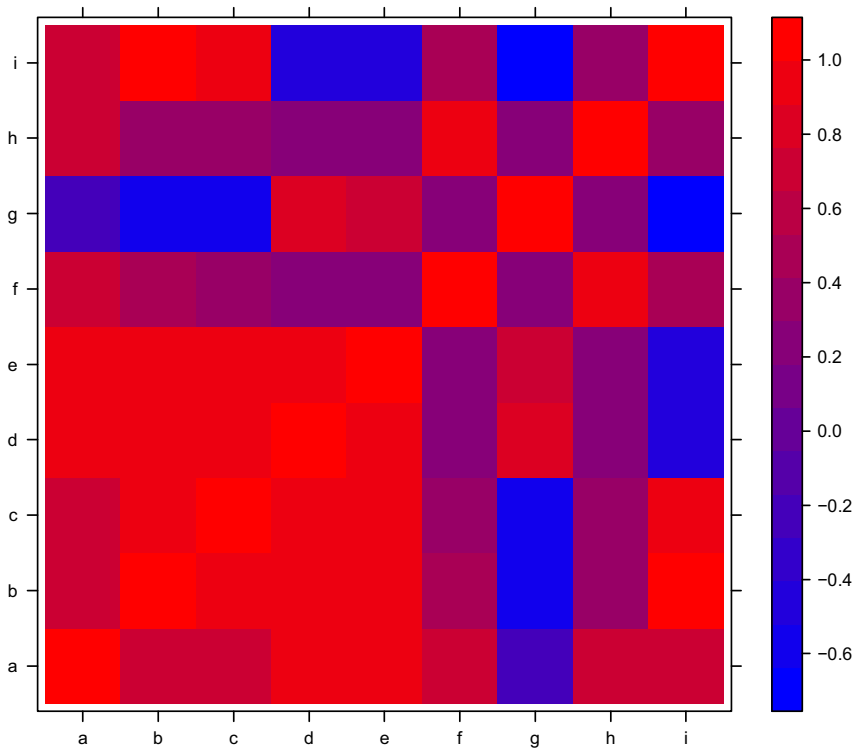


Figure 2.10: Correlogram. The axes X and Y represent the parameters whose values are correlated. Each cell represents the value of the correlation between the vectors on the X and Y axes.

for determining the relations between the different monitoring metrics of the services found under SLA control.

$$r = \frac{1}{n-1} \sum_{i=1}^{n} \left(\frac{x - \bar{x}}{\sigma_x} \right) \left(\frac{y - \bar{y}}{\sigma_y} \right) \tag{2.8}$$

When dealing with measurements from different monitoring metrics, a correlation matrix can be calculated between each time series corresponding to the service monitoring metrics. This then allows determining a set of highly correlated metrics, which can be then used for calculating different estimators for critical monitoring metrics. Figure 2.10 shows a graphical representation of such a dependency matrix between time series corresponding to the monitoring metrics of a distributed application, where the letters represent monitored service' metrics. The red squares indicate a high correlation between the time series.

When dealing with data signals experiencing repeatable patterns, it is advisable to calculate the *autocorrelation* of those signals (correlation value of signal and itself), in order to test if indeed the signal has an observable period. Figure 2.11 shows the partial autocorrelation function of a periodic

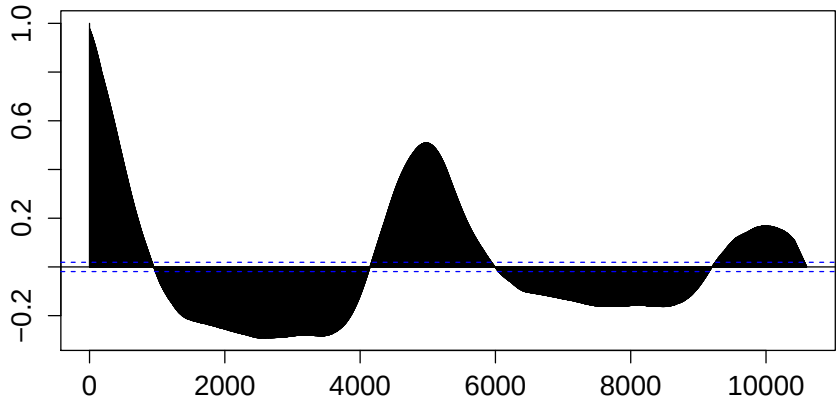


Figure 2.11: Autocorrelation of a periodic data signal

signal, which is the correlation of the signal with its delayed self. The figure expresses several local maxima, with the one corresponding to an index in the data series of approximately 5000 on the X axis, being caused by the signal's period (equal to 5000 samples).

Related Work on Prediction Algorithms

Gandhi et al. [60] propose a hybrid solution for predicting the data center resource demands using historical monitoring data. They are pro-actively predicting the load, for handling the periodic changes that characterize most cloud utilization patterns. They also propose using a reactive workload demand prediction for capturing the differences from the seasonal workload patterns. We also combine SLA-based control of distributed systems with action scheduling based on previously determined relations between application load and the number of active services, however we consider additional algorithms for processing the monitoring information such as a genetic algorithm and result from applying Little's Law.

Anh et al. [50] analyse the performance of cloud applications, which share resources with other running applications in the same physical host, by looking at the correlations between application and system performance. We use statistical correlation [47] between the time series corresponding to the performance monitoring metrics of distributed services in order to determine the set of predictors of critical SLA metrics, which can then be used for controlling services scaling.

Visan et al. [148] describe a bio-inspired prediction algorithm based on a Cascade-Correlation neural network, which uses a genetic algorithm for initialising the network's weights. The authors use their algorithm for performing both one-step and multi-step predictions of a large-scale distributed experiment, with good results. We use a group-oriented genetic algorithm for performing the allocation of VMs to hosts and a moving window linear

regression algorithm for estimating the next values of services' state parameters.

Islam et al. [79] present an approach for predicting the aggregated percentage of CPU utilisation of VMs running a distributed web benchmark, using both error correction neural networks (ECNN) and linear regression (LR). Their results suggest that although using ECNN yields better prediction results than, the need to retrain the neural network might be a disadvantage compared to the use of LR. We focus our work on using LR in the context of SLA-driven scaling of cloud services, showing how prediction can be used for mitigating the disadvantages caused by the delay in instantiating VMs.

Roy et al. [133] present a VM-allocation algorithm, which uses a second order autoregressive moving average prediction method for optimising the utility of the application over a prediction horizon. We also use the prediction of the arrival rate of users for sizing the distributed system, however, we directly use SLAs for solving the VM-scaling problem. We have proposed an algorithm based of moving window linear regression with feedback loop for improving prediction's accuracy.

2.5 CloudSim Simulator

CloudSim [91] positions itself as a simulator for both cloud applications and infrastructure. It accomplishes this by allowing modelling of hardware (VMs and network topologies) and software cloud resources. Among the modeled physical entities there are hosts, network links and datacenters, while the modeled software entities are: virtual machines (VMs), brokers and cloudlets (tasks). This is achieved by offering the mentioned entities as Java classes that can be extended according to simulation requirements. The simulator is implemented using discrete events communication, fired at a specified minimum time interval. Cloud tasks (cloudlets) are created by *brokers*, which send them to VMs for execution on the resources of the hosts forming the datacenter. Upon completion of each cloudlet's execution, its parent broker is notified.

In CloudSim, a *datacenter* is composed of (1) a collection of *hosts*, (2) *storage* devices, (3) a policy controlling the allocation of *VMs* to hosts, and (4) resource utilization costs for comsumed computing time, memory, storage and network bandwidth. Each host is defined by (1) its number of *processing element*s and computational capacity measured in Millions Instructions Per Second (MIPS), (2) RAM memory size, (3) storage size, (4) network bandwidth and *VM scheduling policy*. As VM allocation policies it supports (1) time-shared, (2) space-shared, and (3) time-shared with over-subscription. The datacenter network is given using a BRITE [107] specification.

VMs are described by their requirements in terms of (1) number of CPU cores and MIPS rating, (2) memory size, (3) network bandwidth, (4) virtual machine manager, and (5) cloudlet execution policy. There are four built-in cloudlet execution policies: (1) time-shared, (2) space-shared, (3) dynamic-workload, and (4) network space-shared. The CloudSim API allows for easy development of new cloudlet execution policies.

CloudSim models cloud tasks as *cloudlet* entities [36], defined by (1) their computing requirements given as number of processing elements, and computing task length given in MIPS, (2) network bandwidth consumption for input and output, (3) CPU utilization model, (4) memory utilization model, and (5) network bandwidth utilization model. Cloudlets are representing tasks/jobs submitted for execution to the cloud.

Cloudlets are generated by *Dacenter Brokers*, which are simulator-level representations of cloud services (distributed application services deployed in the cloud environment). Each broker controls one or more VMs and it implements a selection algorithm for choosing which VM will receive the generated cloudlets. The broker also implements the algorithm for reacting to the completion of various cloudlets it has generated.

For simulating a distributed application, one must create one or more datacenter brokers and implement the algorithms for generating cloudlets, as well as handling their completion. Also, at least one datacenter needs to be defined, including its hosts and network. In Section 7.2.2 we present how to model a distributed enterprise application using CloudSim.

Buyya et al. [36] presented an approach for simulating large scale cloud environments by using CloudSim. While they described the steps required for simulating a large number of hosts and network connections. However, they did not focus on how to model applications using CloudSim. Our work fills this gap by presenting a methodology for simulating a large-scale distributed application. We also describe how to map datasets generated by monitoring real services deployed in a distributed environment, to CloudSim models.

Garg et al. [61] extend CloudSim with more advanced network simulation capabilities. They model the datacenter network landscape by introducing switch and network packet simulation entities. The authors simulate a network application using four VMs and compare the results to the simulator, in a similar manner to our approach. We further extend CLoud Sim to support simulation of SLA-based VM-scaling algorithms.

2.6 Little's Law in Distributed Computing Environments

This section describes the theoretical foundation of Little's Law [96], as we used it for enhancing the VM-scaling algorithms presented in Section 4.4.

Little's Law applies to users-processing systems, and it is a result from

the queuing theory stating that the long term average number of users (L) in a system is equal to the product of the rate (λ) at which users arrive, and the average waiting time (W) that a user spends in the system, as expressed algebraically in Equation 2.9.

$$L = \lambda W \tag{2.9}$$

Another form of Little's Law applies to the relation between the average system's throughput (Th), mean number of users in the system (L), and the average execution time (W), as expressed by Equation 2.10.

$$W = \frac{L}{Th} \tag{2.10}$$

It is important to note that Equation 2.9 uses the arrival rate, while Equation 2.10 uses the system's throughput. The two equations are equivalent under conservation of flow conditions, when the average arrival rate (λ) is equal to the average departure rate (or throughput Th). Also, all the jobs entering the system must exit the system at a given point, so the system must report also the exceptional cases when a job fails, as long as that job was considered in the calculation of the arrival rate. Finally, the system needs to be stable [96], by occasionally having $L = 0$ (empty system).

We use Equation 2.10 in Algorithm 4.3, which uses benchmark-generated data describing the dependencies between the average arrival rate, the system's average throughput, the average number of concurrent requests executed by the distributed system and the average execution time.

2.7 Genetic Algorithms

In the section we describe the general structure of a genetic algorithm.

A genetic algorithm (GA) is a type of evolutionary algorithm used for solving search problems. They usually start from a initial solution, which is iteratively improved until the increase in the quality of the solution is no longer significant. This class of algorithms mimics the natural selection process in which a population of individuals is evolving, improving their features so that they better adapt to their natural environment.

In a genetic algorithm, a population of candidate solutions (called individuals) for an optimization (search) problem is evolved towards better solutions, as measured by a *fitness* function. Each candidate solution has a set of properties (its *chromosomes*) which can be mutated and altered (e.g. recombined). Each chromosome is composed of *genes*, which are the building blocks of the solution. In the classical problem of bin-packing, the objects are the genes, and the backpack or the bin is the chromosome.

The evolution process starts from a population of randomly generated individuals (or approximative solutions obtained using other heuristics), and

it is an iterative process. The population corresponding to each iteration is called a generation.

In each generation, the fitness of each individual is evaluated by calculating the value of the objective function corresponding to the optimization problem being solved. The individuals with a high fitness are stochastically selected from the current population, and each individual's genome is modified (recombined and possibly randomly mutated) to form a new population.

The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

A typical genetic algorithm requires:

- a genetic representation of the solution domain
- a fitness function to evaluate the solution domain

Once the genetic representation and the fitness function are defined, a GA proceeds to initialize a population of solutions and then to improve it through repetitive application of the mutation, crossover, inversion and selection operators.

The two genetic operators involved in GAs are the *mutation* and the *crossover*.

The *mutation* operates by randomly selecting a gene from one of the chromosomes forming the population, and then changing its value. As it is generally accepted that a mutation rate that is too high may lead to loss of good solutions, it is common the case that *elitism* is used for protecting the good solutions, by keeping unchanged a given percentage of the best chromosomes in each generation.

The *crossover* operator takes two parent chromosomes from the current generation and exchanges their genes, producing two new *child* chromosomes.

The quality of each solution is calculated by using a *fitness* function which measures how close the chromosomes in the current generation are to solving the problem.

The GA reaches termination when one or more of the following conditions are met:

1. a given number of generation has been reached
2. the maximum value of the fitness in each generation has reached a plateau such that successive iteration are no longer producing an increase in the quality of the solution
3. the maximum computational budget (e.g. time) has been used

Chapter 3

Topology Orchestration and Semantic SLA Modeling for Distributed Cloud-Based Applications

In order to solve the problem of SLA-driven scaling of ditributed applications, we first have to define the model that will be used for representing the distributed applications and their performance constraints. For this purpose, we describe in Section 3.1 a well-known model of topology orchestration, proposed by the Organization for the Advancement of Structured Information Standards (OASIS). After we have presented the lifecycle phases associated with topology orchestration, we continue with extending the USDL-SLA ontology (already presented in Section 2.2.1) in Section 3.2 with the needed terms for enabling automated processing of SLAs, thus facilitating the solution for unattended scaling of distributed applications using SLA-defined constraints. We introduce the semantic relations for expressing the connections between services, their monitoring metrics, and performance invariants, which are the foundation of SLA expressions. We then use this flexible ontology in Chapter 4 for describing the SLA-based Cloud Service Management Platform.

3.1 Introduction to Topology Orchestration and SLA Modeling for Cloud Environments

Cloud-distributed applications [154] are component-based, distributed, scalable, and complex business applications, usually mission-critical (e.g. the applications are the key to the success of the enterprises running them). Often these applications are running in managed computing environments, such as datacenters [33]. Therefore, the automated manner in which these distributed applications are managed, leads to the requirement of having a machine-readable representation of their components, services, infrastructure-resource requirements, monitoring metrics and actionable (executable)

management operations.

The dynamic management of distributed software and infrastructure compute, storage and network resources is referred to as dynamic application topology orchestration [10, 18], where the mapping of services to cloud resources and configuration of distributed, interconnected, interdependent application services and infrastructure resources are dynamically adjusted, according to guarantees in Service Level Agreements (SLAs) and operational constraints.

In the context of scalable distributed applications, automated management of applications plays a crucial role, as the correct provisioning of the virtual infrastructure resources (such as computing, network and storage) depends on the correct identification of the application's requirements. Similarly, at runtime, the correct interpretation of the states of the distributed applications depends on the identification of SLA guarantees and of the monitoring metrics used in the definitions of the SLA's expressions (e.g. performance guarantees). Based on this information, scaling actions can be performed, in which either the number of application instances is varied (e.g. some instances are added, or removed), or the amount of infrastructure resources is changed (e.g. number of allocated virtual CPU cores, or the amount of allocated RAM memory is changed).

The core concepts for application topology specification and orchestration of SLA management, namely, provisioning, monitoring and response are presented next.

3.1.1 Application Topology and Infrastructure Requirements

An **application topology** AT is a structural specification of how distributed **services** S are to be deployed and interconnected on any selected execution environment.

We define the application topology as a tuple $AT = (S, SS, Z, ZZ, P)$ where:

- S is the set of distributed **services** that compose the application. Each $S_i \in S$ has a unique name $S_i.\texttt{uname}$ and is associated with a set of assets $S_i.\texttt{assets} = (a_1, ..., a_n)$. An asset a_i refers to software, images, scripts or data necessary for the service to be executable. Each asset a_i declares a set of metrics $a_i.\texttt{metrics} = (m_1, .., m_n)$ with which its state can be observed. Each metric m_i has a unique name and 1 probe $m_i.\texttt{probe}$ that conforms to a standard protocol defined by the target infrastructure or platform.

- SS is a set of **service relationships** (S_i, S_j, l) where $i \neq j$, l is a label from a predefined enumeration of relationship types, and $(S_i, S_j) \subseteq S$. Relationships are used to inform the orchestration of deployment and service maintenance actions.

The set of relationship types we define for l are as follows:

- S_i `disrupts` S_j: S_i and S_j should not be collocated, otherwise the system cannot be stable.
- S_i `includes` S_j: if S_i is deployed or changed, then S_j is deployed or changed.
- S_i `interacts` S_j: S_i requires a bi-directional connection to S_j and proximity.
- S_i `mirrors` S_j: changes in metrics at S_j can be observed at S_i.
- S_i `needs` S_j: S_j must be deployed before S_i. S_i requires knowledge of changes in metrics and availability of S_j.
- S_i `observes` S_j: S_i requires connectivity to S_j for the purpose of querying or polling
- S_i `substitutes` S_j: if S_j is unreachable, S_i can provide its function and metrics.

- Z is the set of logical **regions** (each logical region contains a subset of the resources found in a physical region) used for containment of collocated services S. At deployment time each $Z_i \in Z$ is mapped to an actual, physical region, which are compute and storage resource pools at different datacenters or cloud providers. Each Z_i is a data-structure with a unique name Z_i.`uname`, internally required network Z_i.`net`, storage Z_i.`str`, memory Z_i.`mem` and processor Z_i.`cpu` capacities and capabilities. At topology specification time a mapping $Z_i \leftarrow S^i \subseteq S$ is defined to assign a subset of services to each zone.

- ZZ is the set of **inter-region relationships** $zz \in ZZ = (Z_i, Z_j, net_{i,j})$, $i \neq j$ and $net_{i,j}$ defines the initial bandwidth requirement for the connectivity between zones Z_i and Z_j.

- P defines the set of **rules** or **policies** used to manipulate the state of services in the application topology and behavior of the infrastructure towards the services.

When the application topology has been specified, its infrastructure requirements are specified per zone, without a priori selection or assignment of physical regions and resource pools.

3.1.2 Dynamic Topology Orchestration Process

Dynamic topology *orchestration* is a 5-staged process, following the specification of a valid application topology. An *application topology* is a specification of the distributed application composed of the various interconnected services, which are mapped to different cloud virtual resources (e.g. VMs).

The orchestration process is a continuous process driven by the monitored metrics, guaranteed triggers and guaranteed actions in the topology specification. The process is terminated when the agreed lifetime of the application expires or other guaranteed triggers are observed, as specified in the SLA.

1. Request Handling and Scheduling. This stage involves transformation of the application topology into multiple, service deployment requests. There is a 1-to-n mapping between services and requests, meaning that a service can execute multiple requests. Each request states the relevant service, request type, explicit target, where it should be executed, operation to be performed, set of parameters, schedule for the request to be executed and set of post-deployment information that should be provided for subsequent requests, according to its dependencies.

2. Infrastructure Preparation. This second stage determines what infrastructure resources are required, where they are located and how they should be configured, given a set of scheduled requests. The first activity of preparation focuses on the compute and storage end-points, as these need to be activated before the properties for network paths can be configured. Subsequently, reservation of inter-datacenter (region) network capacity is done, completing the configuration of paths between relevant - the GEYSERS project[63] can be referenced for more details on path-computation. The final activity in this stage is the activation of infrastructure probes according to the infrastructure metrics specified in the topology.

3. Service Deployment. This stage is the installation of application-level assets including images, binaries, scripts and application data on infrastructure resources in regions where their associated services are mapped. Application-level monitoring probes (independent applications responsible with monitoring the state of the distributed services) are then activated according to the application metrics defined in the application topology template.

4. Service Monitoring. This stage is a continuous collection of metrics from the different infrastructure and application monitoring probes. Each probe is associated with a service metric. The metric's value is sent together with the metric identifier and the unique service identifier to the monitoring handler, where the value is recorded and a window of v values stored in memory for noise rejection. For the specified window, the minimum, maximum and average values are calculated and made available for the use in the evaluation of guaranteed triggers from SLAs' expressions.

5. Response. This stage occurs when a guaranteed SLA trigger is raised and a rule exists to resolve the difference between the guaranteed trigger and the guaranteed SLA state. Both the trigger and guaranteed state are part of the SLA model, which we are detailing in Section 3.2. The resolution in the rule is an action that either returns to stage 1, creating a new request, or the invocation of a specific operation on a target infrastructure resource or service element.

3.2 Enhanced Semantic Model for SLA Representation and Cloud Topology Orchestration Modelling

A Service Level Agreement (SLA) [141] is a contract between a consumer and a provider of a service regarding its usage and quality [17]. It defines guarantees or Quality of Service (QoS) terms under which the services are provided and the ways for checking those guarantees. The SLAs might also contain guaranteed actions, which might be used for enforcing the validity of the guaranteed terms. The content of SLAs is inevitably used for concrete instantiation and configuration directives, which parametrize provisioning of resources, deployment of software and tuning of settings to enable effective operation of deployed services.

We use SLAs as the basis for specifying dynamic behaviour of application and infrastructure services, by using an extension of the USDL-SLA [93] vocabulary for describing the guaranteed states and actions (management rules), as well as the conditions required for automatic execution of the actions. The SLAs also contain enough information for determining the context in which the conditions are evaluated.

Rules for managing the dynamic application topology are specified by the owner/ architect of the topology. All rules have the following format:
`IF NOT vf(gs, gt,` $(m_1, ..., m_n)$`) THEN ga`

- `vf` $\in$ `VF` is a boolean **validation function** from the set of known functions `VF`, which causes the rule to fire when false. Standard binary and arithmetic operators are used.

- `gs` $\in$ `GS` is one **guaranteed state** from the set of states `GS` defined in the SLA, which the infrastructure is responsible for maintaining. It is the first input to the validation function `vf`. Some examples of guaranteed states include availability, capability, efficiency, stability and security. The `vf` uses this as a reference.

- `gt` $\prec \neg$ `gs` is the **guaranteed trigger** for the rule and represents a state that precedes invalidation of the guaranteed state `gs`, such that `gs` should ideally never be invalid.

- $m_1, ..., m_n \subseteq M$ is an array and subset of **metrics** `M` registered with the infrastructure. That is, the infrastructure knows how to obtain and evaluate these metrics. The array is the second input to the validation function `vf`. There are three domains within which metrics are defined:

    1. `Application.statistic` classifies metrics defined by the application owner to measure statistics for number of users, response times and request queue lengths.

2. `Application.service` classifies metrics defined by the application owner to asses the availability, location and number of application services and instances.
3. `Infrastructure.resources` classifies metrics defined by the infrastructure administrator to assess the consumption and availability of memory, storage, CPU and networking.

- `ga` $\in$ `GA` is a **guaranteed action** from the set of known actions `GA`. A `ga` is executed by the infrastructure management whenever a `vf` in the associated rule fails. The following action templates are used for defining guaranteed actions:
  - **`start`** `<service>` **`with`** `<parameters...>`: used to initiate a specific service in the application topology.
  - **`stop`** `<service>` **`with-error-level`** `<level>`: used to terminate a service with an indicator of why the termination should occur.
  - **`update`** `<service>.<metric>` **`with`** `<value>`: used to configure a service's state by assigning a value to a defined metric.
  - **`increase`** `<target>` **`by`** `<number>`: used for scaling-*out* to increase performance and reduce response times.
  - **`decrease`** `<target>` **`by`** `<number>`: used for scaling-*in* in order to save energy and costs.
  - **`redirect`** `<service>` **`from`** `<region-x>` **`to`** `<region-y>`: used for high-availability scenarios with redundancy, where a service in region-x is mirrored or can be substituted by one in region-y.

Note that the term *service* is used to describe components of the application or capabilities of the infrastructure. For example, the *computation* and *network* available to the application topology are referred to as the "computation service" and "network service" respectively.

Figure 3.1 depicts the model used for defining the SLAs. Coloured in grey are the original terms of the USDL-SLA ontology, while with green we highlighted the additional terms proposed by us. Although the USDL-SLA specification is able to represent service types and service level profiles, it does not provide a way of actually representing these SLAs (guaranteed states and actions) into a machine readable format that can be used for automated scaling management. Our enhanced model introduces the terms required for linking the services to their monitoring metrics, therefore allowing expressing the SLAs into a machine readable representation, such as MVEL [34]. More concretely, the following terms were introduced for extending the USDL-SLA model: *Service Level Expression*, *Monitoring Metric*, *Matric Type Representation*, *Constant*, *Variable*, as well as the semantic relations connecting these terms to the original entities of the USDL-SLA model.

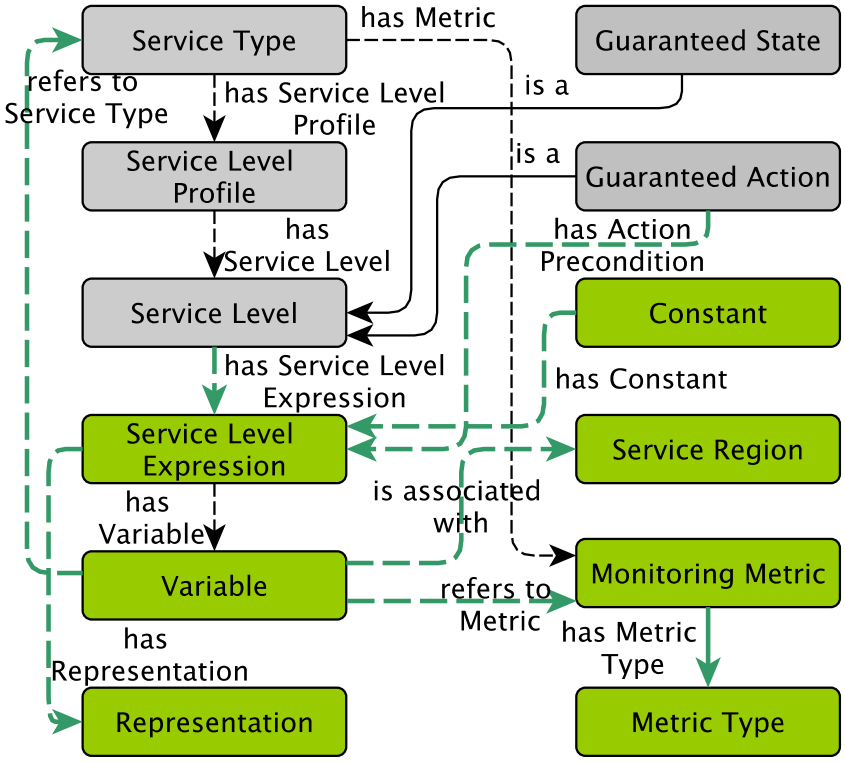


Figure 3.1: Extended USDL-SLA Ontology (own contributions are marked in green)

The SLA model contains *Service Type* entities that are used for representing the linked descriptions [126] of the services. Each *Service Type* specifies one or more *Monitoring Metrics* that are used during runtime for gathering state information about the actual service instances. The *Service Type* also specifies a *Service Level Profile* that contains one or more *Service Levels*. A *Service Level* can be either a *Guaranteed State* or *Guaranteed Action*, which are used during runtime for checking the state of the services and for performing actions on them, such as scale-up and scale-down. Each *Service Level* has a *Service Level Expression* that is used as described below.

The *Guaranteed State* service level specifies a single *Service Level Expression*, which is used for checking a service state-invariant (gs). During runtime, the expression is periodically evaluated for each service instance of the specified *Service Type* and if the guaranteed state is violated, then a log entry will be created, which can then be used for audit purposes.

The *Guaranteed Action* also has a *Service Level Expression*, which contains the actual code representation of the guaranteed action, as well as another expression, which is the condition (gt) describing the state in which

the application must be in order to trigger the specified action.

The *Service Level Expression* has a *Representation* that uses one or more *Variable*s. Each variable contains the value of a specified *Monitoring Metric*. The *Monitoring Metric* has a type, which can be application statistic, service, or infrastructure. As multiple *Service Type*s might refer to the same monitoring metric, the *variable* must also specify which *Service Type* is used for setting its value. Likewise, as the specified service might have instances running in different regions, the variable must specify to which service it refers. This is of particular importance for the service levels that have expressions evaluated in the context of one or more regions, such as requesting network bandwidth for a connection between two remote data-centers.

The following is a simplified example of an SLA written in the enhanced USDL-SLA language, with the full example being given in Annex 9.1

```
:workerSLAProfile
      rdf:type usdl-sla:ServiceLevelProfile , owl:NamedIndividual ;
      usdl-sla:hasServiceLevel :workerServiceScaleOutAction .

:workerServiceScaleOutAction rdf:type usdl-sla:GuaranteedAction ,
            owl:NamedIndividual ;
      :hasActionPrecondition :workerScaleOutCondition ;
      usdl-sla:serviceLevelExpression
            :workerServiceScaleOutActionExpression .

:workerServiceScaleOutActionExpression
      rdf:type usdl-sla:ServiceLevelExpression , owl:NamedIndividual ;
      :hasRepresentation
            """var outcome = new java.util.HashMap();
            outcome.put(\"rule-action\", \"service-scale-up\");
            outcome.put(\"service-name\", \"WorkerService\");
            return outcome;"""^^xsd:string .

:workerScaleOutCondition
      rdf:type usdl-sla:ServiceLevelExpression ,
            owl:NamedIndividual ;
      :hasExpressionPreprocessor """
var currentWorkerServicesCapacity =
workerServiceRunningOrStartingCountVariable *
workerServiceCapacityConstant;
var averageNumberOfActiveSessionsAtLoadBalancerVariable =
average(numberOfActiveSessionsAtLoadBalancerVariable);
var workersAreApproachingTheSLALimit =
(averageNumberOfActiveSessionsAtLoadBalancerVariable >=
currentWorkerServicesCapacity);
```

```
System.out.println(
\"Current user handling capacity of all Worker Services:
\" + currentWorkerServicesCapacity);
System.out.println(
\"Average number of active user sessions: \"
+ averageNumberOfActiveSessionsAtLoadBalancerVariable);
if (workersAreApproachingTheSLALimit) {
      System.out.println(\"Service Scale-UP triggered.\");
}""" ;
      :hasRepresentation
            "workersAreApproachingTheSLALimit == true" ;
      usdl-sla:hasVariable
            :numberOfActiveSessionsAtLoadBalancerVariable ;
      :hasConstant
            :workerServiceCapacityConstant ;
      usdl-sla:hasVariable
            :workerServiceRunningOrStartingCountVariable .

:numberOfActiveSessionsAtLoadBalancerVariable
      rdf:type usdl-sla:Variable , owl:NamedIndividual ;
      :refersToServiceType :LoadBalancerService ;
      :refersToMetric :numberOfActiveSessionsMetric .

:numberOfActiveSessionsMetric
   rdf:type :MonitoringMetric , owl:NamedIndividual ;
   :hasMetricType :ServiceMetric .

:workerServiceCapacityConstant
      rdf:type :Constant , owl:NamedIndividual ;
      :hasRepresentation "5" .

:workerServiceRunningOrStartingCountVariable
      rdf:type usdl-sla:Variable , owl:NamedIndividual ;
      :refersToMetric :RunningOrStartingInstancesCountMetric ;
      :refersToServiceType :WorkerService .
```

The previous example contains the definition of a SLA profile (*:workerSLAProfile*) for the *EIS Worker* service, containing one service-level, specifically a guaranteed action (*workerServiceScaleOutAction*). The guaranteed action has (1) a precondition (*workerScaleOutCondition*) defining the condition required for the execution of the guaranteed action, and (2) a service level expression (*workerServiceScaleOutActionExpression*) representing the specification of the action to be executed, given using the *hasRepresentation* property.

The service level expression *workerScaleOutCondition* specifies that in its representation (in this case, using MVEL[34] language) references two variables: *numberOfActiveSessionsAtLoadBalancerVariable* and *workerServiceRunningOrStartingCountVariable*, which are defined on lines 49, respectively, 62. Also, the service level expression contains a constant, the *workerServiceCapacityConstant*, defined in the line 58. Each of the two variables are defined in the context of a specified service (given using the property*:refersToServiceType*), by measuring the value of a monitoring metric specified by the *:refersToMetric* property. In the case of the *numberOfActiveSessionsAtLoadBalancerVariable* variable, its value is evaluated in the context of the *Load Balancer* service and its monitoring metric *:numberOfActiveSessionsMetric* (defined in line 54).

The SLA profile (full example can be found in Annex 9.1) will next be processed by the cloud management platform, in a fully automated manner, as described in Section 4.2.

Chapter 4

SLA-Based Cloud Service Management Platform

After describing the SLA specification language in the previous Chapter 3 we present in this chapter the SLA-based Cloud Service Management Platform, which uses the semantic SLA model for controlling the instantiation of services onto virtual machines, as well as the horizontal scaling of the virtual machines according to the SLA-defined application-level performance guarantees.

A typical scenario for deploying a distributed managed application on the management platform described in this chapter involves several phases as follows: (1) SLA modelling, (2) Application deployment, and (3) SLA optimisation and control of the application. Their detailed description is given below.

First, a semantic description of the distributed application is created using the SLA model described in Chapter 3. This SLA description must express all the services that are to be deployed on the cloud infrastructure, the dependencies between them, their monitoring metrics and the associated measuring probes, and the required infrastructure resources for deploying the corresponding VMs hosting the distributed services. The description of critical SLA metrics and their threshold values (e.g. end-to-end execution time lower than 1 second) is important for the automated scaling process. The SLAs must also specify guaranteed states for the distributed application as well as SLA actions to be taken when these SLA states are violated.

In the second phase, previously prepared SLAs are given as input to the SLA-based management platform, which will begin deploying the VMs according to the SLA-constraints (e.g. VM sizes, service deployment order). The system will optimize the allocation of VMs on the infrastructure resources (hosts) by applying the genetic algorithm described in Chapter 6. The algorithm determines several rounds of VM-allocations while trying to optimise a set of criteria including maximisation of energy efficiency, minimisation of SLA penalties caused by CPU and/or network over-subscription,

minimisation of SLA penalties caused by migration of VMs, and finally maximisation of provider's revenues obtained from hosting the given set of VMs.

The third phase of the deployment scenario involves optimising the SLAs with regard to the automated scaling of VMs. For this purpose, we have described in Chapter 5 two analytic approaches to composition and optimisation of SLAs based on concurrent benchmarks, using the *DynSLAOp* system. The SLA-Based Service Scaling Manager (SLA-SSM) is responsible for dynamically adjusting the number of VMs for each of the services of the distributed applications belonging to each of the cloud tenants. It accomplishes this using invariant conditions formed with SLA-terms obtained from the performance indicators of the services running in VMs. An example of an invariant condition can be: "average distributed transaction execution time is below one second". The threshold contained in the SLA invariant is then used by the SLA-SSM for determining the conditions for performing either a scale-out action [55] (creating one or more VMs), or a scale-in action (terminating one or more VMs).

We present next the information model of the SLA Service Scaling Manager, followed by the system's architecture. We then describe two mechanisms important for data processing in the context of service scaling, followed by three SLA-based service scaling algorithms.

4.1 Information Model of the SLA-Based Service Scaling Manager

The core capability for any SLA management system is access to timely information about the managed services and related resources. For the purpose of prediction, the information model must provide access to historical, current and static facts about the managed system. Historical and current facts about changing observations in the system are captured in the monitoring logs, while the static information is seen in the more structural description models. These description models are summarized below.

Distributed Applications specify a unique application name, under which multiple services are grouped. In other words an application is a collection of services in the managed system. The dEIS 7.1 is an example of a distributed application, where the worker, load-balancer, storage and customer-facing components are registered as services.

Services are a mapping of application components to infrastructure resources (e.g. VMs). When SLA management decisions are made, the registry of services is used to determine the time for executing life cycle operations. Each service is also associated with a set of metrics, which are relevant to the component or resource and can be measured using a probe attached to the component or resource.

Regions are initially logical segments of the distributed application, to which collections of related services are assigned. They are mapped to physical portions of the infrastructure that define the application's distributed runtime.

Virtual Machines are included in the information model as one class of infrastructure resource. The CPU, memory and storage capacities, as well as other OS-specific properties of each virtual machine is captured in the information model. This is important for predictive analysis and SLA management actions, as the constraints of virtual machine capabilities serve to inform decisions and limit what operations are feasible in the next control cycle.

Network Links represent infrastructure resources captured in the information model, describing the bandwidth and latency characteristics of virtual links between virtual machines.

SLAs capture the guarantees that the service provider has agreed with the service consumer. These are guaranteed states and actions that are performed if the right conditions are observed during a control loop.

Historical and current facts about the services are captured in the **Monitoring Data and System Logs**, where each record consists of a unique timestamp, service monitored and the set of metric values for the time period of monitoring. This enables time-series analysis and the identification of trends in different metrics over time. The gathered monitoring data can be classified into application-level and infrastructure level data. Example of application-level monitoring metrics are the various durations of service executions (e.g. the time it takes for the service containing the business logic to process a request). Example of infrastructure-level monitoring metrics are duration of network transfer of data, virtual machine CPU level, physical server CPU utilization.

4.2 Architecture of the SLA-Based Service Scaling Manager

This section describes a system architecture for enabling the 5-stage service-deployment process described in Section 3.1.2. The architecture is designed as a tool for application topology architects and administrators. It is a loosely-coupled, message-based architecture, such that most components are defined as types of *Handlers*, as shown in in Figure 4.1. The letter R shown above the links connecting the components stands for **R**equest and the arrow reflects the direction of the request.

Once the application topology specification bas been completed, it is submitted to the *Request Handler (RH)* via the *Request Interface*, triggering stage 1 of the orchestration process (described in 3.1.2). The states of requests, services and SLA of active topologies are continuously monitored via a corresponding *Query Interface*. The *RH* queries the *Asset Handler*

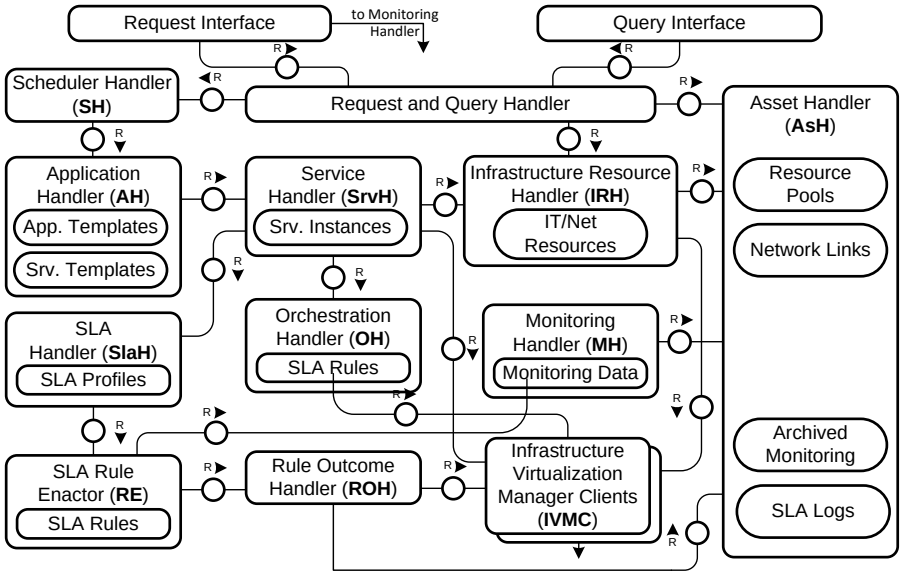


Figure 4.1: Architecture of the SLA-Based Service Scaling Manager

(AsH) for a list of available, infrastructure resource pools of computing, storage and memory resources according to the requirements of the topology specification. The application architect or administrator is responsible for initial selection of mappings between application regions in the topology specification and suggested resource pools. Once this initialization mapping is completed, the `RH` parses the topology specification into a set of deployment requests and passes these to the *Scheduler Handler (SH)*. The `SH` builds a schedule for requests and associated handler operations, allowing subsequent or batches of application topologies to be submitted.

When a topology bas been scheduled for deployment, the *Application Handler*(AH) is invoked to register the service templates and dependency mappings in the topology, indicating that it is being activated. Initially selected infrastructure resource pools are marked as `'reserved'` for the specific application topology in the *AsH*, moving to phase 2 of the management process.

This then signals the *Infrastructure Resource Handler (IRH)* to establish and set up network links between the resources pools, which could be at different, physical locations. The *IRH* has a signaling connection to *Infrastructure Virtualization Manager Clients (IVMCs)* at each site offering resource pools and connectivity. It is via these *IVMCs* that the *IRH* is able to request isolated portions of infrastructure, per application topology to be served. The network links are also registered with the *AsH* once provisioned. An example of *IVMC* is the OpenNebula adapter described in Section 5.2.1, which is used for controlling the state of the VMs hosting the application

services.

Once this underlying virtual infrastructure of distributed resource pools and network links bas been established for the application topology, the service templates and SLA profiles from the topology specification are passed to the *Service Handler (SH)*, where the actual topology orchestration process is enacted.

The *AH* processes the application instantiation request by first identifying the application topology and software templates referred in the request, as well as the resource pools indicated for hosting the services. For each service template, the *AH* requests the *Service Handler* (SrvH) the instantiation of the minimum number of instances in the mapped resource pools. The *SrvH* assigns a unique identifier to the service instance and then passes the request to the *Orchestration Handler*, which resolves the service dependencies and populates the service context, waiting if necessary for the services to be instantiated.

As the services begin operating, the individual instances begin sending monitoring information, which is received and processed at the *Monitoring Handler*.

After the successful notifications of the services' instantiations have been received, the *AH* retrieves the SLA profile for each service and then registers it at the *SLA Handler*(SlaH). The *SlaH* stores the SLA profile and then passes it to the *Rule Outcome Handler*(ROH) for processing. The *RE* extracts the *Service Level*s from the SLA profile and for each level determines its *Expression*.

For the SLA service levels of type *Guaranteed Action*, the expression is wrapped by an `if` checking the action's precondition expression, ensuring that the management system will execute the *SLA guaranteed action* only when the defined *SLA expression* is evaluated to *true*.

For the *Guaranteed State* SLA service level, the SLA expression is wrapped in a `if` triggering the creation of a *Rule Outcome* of type SLA-log when the expression evaluates to `false`. The generated code sequences are then converted to *MVEL* [34] language, which can then be dynamically evaluated. For each expression contained in the SLA guaranteed-states and SLA-triggers of the guaranteed-actions, its variables are identified, and their values are populated with data from the *Monitoring Handler*. All identified variables are then added to a *Context* (dictionary) object that will be passed to the MVEL framework for evaluation, together with the MVEL sequence. The expression's context and MVEL evaluation are wrapped in a JAVA *Runnable*[81] and executed periodically.

At each SLA monitoring cycle, the *RE* produces *Rule Outcome* objects, which are passed to the *ROH* for processing, based on the outcome's type (e.g. scale-out, scale-in, migrate, network-bandwidth-increase). Each rule outcome is a dictionary (e.g. JAVA map [80]), which has a set of keys that are used for describing the corresponding SLA action. In case of outcomes

generated from *Guaranteed Actions* SLA levels, the handler actually triggers the execution of the desired action by building a request for one of the *Infrastructure Virtualization Client*s. In case of outcomes generated from a *Guaranteed State* SLA level, the handler logs the SLA violation by archiving it in the *AH*. We present in Section 4.4 multiple VM-scaling algorithms running inside the *Rule Enactor* component.

4.3 Multi-Step Prediction using Linear Autoregression

This section describes a prediction algorithm used for analysing the monitoring data gathered by the *MH*, which is then used by the *Rule Enactor* component for further enhancing the management of the applications' scaling.

Algorithm 4.1: Streaming Linear Prediction using Autoregression

```
1  model ← NULL
2  previous.data ← empty
3  previous.time ← empty
4  win.len ← p seconds
5  prediction.length ← n seconds
6  while Data Stream is Open do
7      win.data ← buffer(win.len)
8      win.time ← time(win.data)
9      if model is NULL then
10         model ← linear regression(win.data, win.time)
11     else
12         prediction.current ← predict(model, win.time)
13         err.mape ← MAPE(win.data, prediction.current)
14         if err.mape < ϵ then
15             t ← current time
16             prediction.time ← (t + 1, ..., t + prediction.length)
17             win.predicted ← predict(model, prediction.time)
18             use win.predicted
19         else
20             model ← NULL
21             previous.data ← empty
22             previous.time ← empty
23     append win.data to previous.data
24     append win.time to previous.time
25     model ← linear regression(previous.data, previous.time)
26     model.accuracy ← accuracy(model)
```

We investigate autoregression for predicting multiple future values of

an independent variable. As underlying example we will use the time series shown in Figure 4.2 containing a window of data representing the arrival rate of requests of an ERP system. Figure 4.2 describes the process of generating a constantly increasing number of requests, followed by a slow decrease in the number of requests. As we want to use the data for predicting the future values, this means that it will be processed in a streaming fashion, as it becomes available.

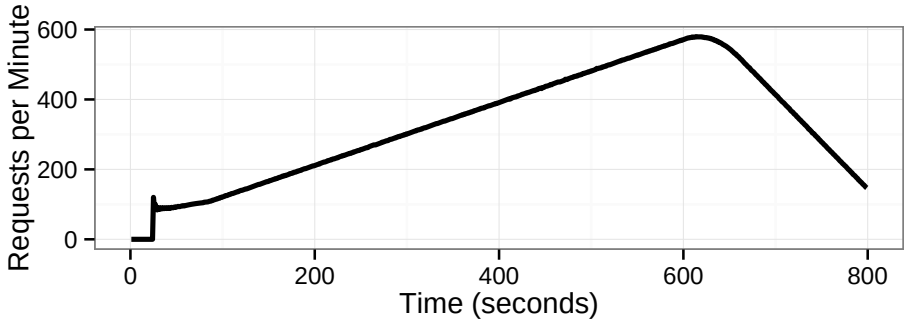


Figure 4.2: Requests Arrival Rate

Two important properties of the prediction algorithm are the following: (1) immunity to small variations (non-uniformity) in the data sampling period, and (2) auto-alignment of the regression with the actual trend of the signal. The first property ensures that the prediction produced by the algorithm will always be sampled with the same period, allowing a deterministic use of the predicted data, independent of the actual sampling period of the predicted signal. The second property ensures that the algorithm will auto-adjust to changes in the signal's trend (e.g. slope) by monitoring the prediction error and adapting the length of the window of data used for calculating the "learning" model. Next, we present the prediction algorithm.

The algorithm is evaluated repeatedly, and it uses the results from the previous run. In the first iteration, a regression model is calculated (e.g. using R's *lm* function of the *stats* package [129]) using the current window of data in line 10. Only after the *model* is calculated, can it actually be used for prediction in line 12. In the following iterations, the linear model calculated in the previous iteration will be used for predicting the current data window by applying the same regression *model* to the time moments corresponding to he sampled data (*win.time*), producing the *prediction.current* data vector in line 12. Next, the mean average percentage error (*err.mape*) is calculated from the current data window (*win.data*) and the model's prediction (*prediction.current*). The *err.mape* is actually the out-of-sample error measure of the regression model, as the error was calculated with data coming from outside the time window known by the regression model. If the error is lower than a predefined threshold ϵ (e.g. 1.5%) then (1) the model is considered valid and it is used in line 17 for predicting the next n seconds

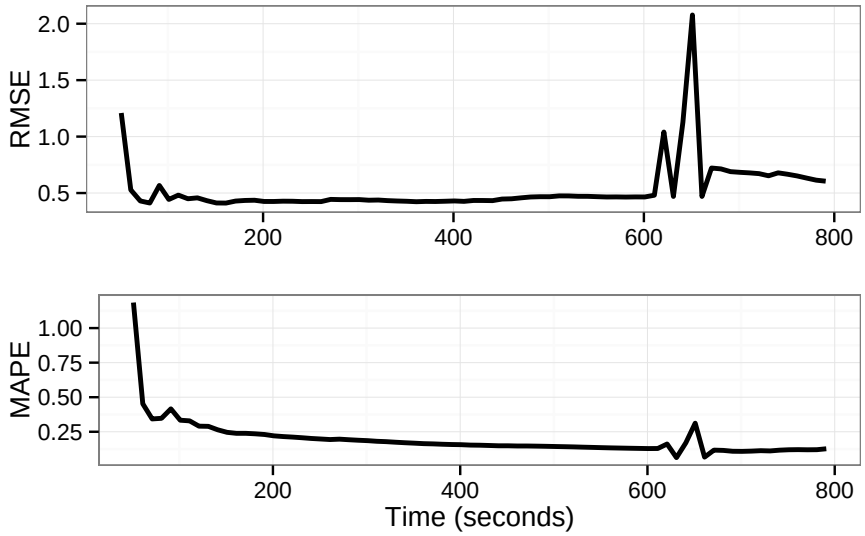


Figure 4.3: Regression Errors. Top: Root Mean Standard Error. Bottom: Mean Absolute Percentage Error

of signal's values at times $t+1, ..., t+n$, and (2) the current data window and its sampling time moments is appended to the previous data window, respectively to the previous sampling time window. If the error is larger than the specified threshold, than the *model* is dropped, together with the accumulated data, and the whole process of learning the model and using it for prediction is repeated.

By calculating the signal's prediction at times $t+1, ..., t+n$, the immunity to variations in the signal's sampling period is guaranteed, which allows the subsequent components to use the produced data forecast at any arbitrary time horizon smaller than n. Also, by accumulating the signal's data samples and calculating the regression model with an increasing length of the data window this ensures that the regression model's slope will be aligned with the signal's slope with an error of at most ϵ. As soon as the out-of-sample (defined as the error between the data prediction calculated with *in*-sample data, and the actual measured data, which has not yet been appended to the considered data frame *previous.data*) $MAPE$ error (defined in Equation 2.7, in Section 2.4.4) of the regression's model exceeds ϵ, the model and accumulated data will be dropped, and a new *model* will be calculated, thus satisfying the second property of the prediction algorithm.

The linear regression model's accuracy represented by the RMSE and MAPE errors are displayed in Figure 4.3, when applying linear regression to the data from Figure 4.2 (average arrival rates), with a regression window *win.len* of 10 seconds. As expected the graphs show two peaks because of the two changes in the underlying data's trend at time 20 and 600.

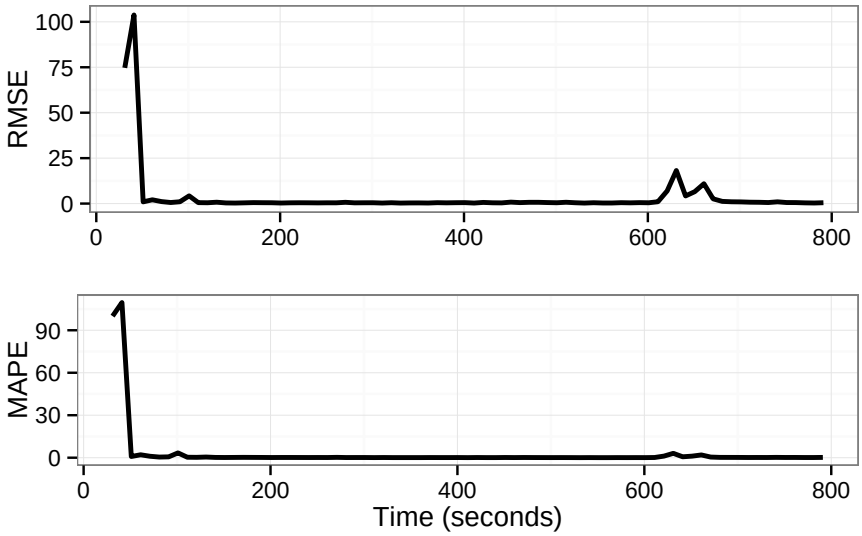


Figure 4.4: Forecast Errors. Top: Root Mean Standard Error. Bottom: Mean Absolute Percentage Error

Figure 4.4 shows the evolution of both RMSE and MAPE errors for the out-of-sample accuracy. As expected, there are two regions where the errors have large values, in the beginning, around time moment 0, and around time moment 600. This is because in the beginning of the time series the data experienced a slight instability when transitioning from 0 to 60 requests per minute. Also, at time moment 600 the trend of the data changed from increasing to decreasing, leading to the an decrease of the accuracy of the previous regression model valid only until the time moment 600.

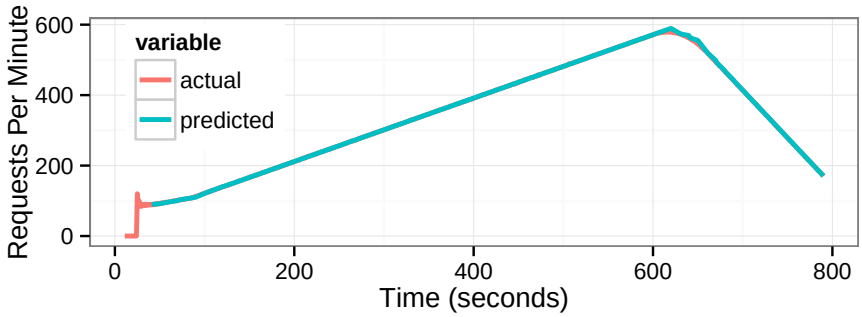


Figure 4.5: Forecast vs. Actual Data

For calculating the signal's prediction of the next n seconds (equal to 40 in our example), first a data window *prediction.time* is created (line 16) containing the time values between the current time $t + 1$ and $t + n$. The regression model is then used in line 17 for calculating the data prediction

win.prediction corresponding to time interval *prediction.time*.

The predicted data *win.prediction* is finally used in line 18. An example of using the predicted data is given in Section 4.4, where the predicted arrival rate of requests is used as input to a SLA-based VM-Scaling algorithm.

The predicted data obtained by running the algorithm is displayed along the initial data in Figure 4.5. The very good accuracy of the prediction algorithm can be observed, as the predicted data closely follows the input data. Also, it can be noticed that the predicted data starts at around time moment 70, after the data's trend stabilizes itself.

4.4 SLA-Based Service Scaling Algorithms

In this section we describe three SLA-based algorithms for scaling services. The first algorithm uses a "reactive" approach at monitoring the state of the services, by performing either a scale-out or a scale-in operation if the evaluation of the SLA-defined condition exceeds a critical threshold. The second algorithm uses results form applying Little's law. The third algorithm extends the second one by using the prediction mechanism defined in Sec. 4.3.

4.4.1 Reactive SLA-Based VM-Scaling Algorithm

The reactive service scaling Algorithm 4.2, operates by calculating the SLA ratio sr as the factor by which the average $\overline{m_{SLA}}$ over the moving time window W of the SLA metric m is approaching its maximum threshold $max_{SLA}(m)$. If sr is above a given threshold S^{UP} (e.g. 0.9) and sr is increasing from the last check then a scale-out operation is flagged.

Similarly, if sr is below a threshold S^{DOWN} (e.g. 0.6) and sr is decreasing, then a scale-in operation is flagged. Either scale-out or scale-in operations will be executed only if the number of such operations ss is below a given threshold ss^{MAX} (e.g. 2) in the last W_S seconds (e.g. 40 sec, chosen as 1.5 times the time it takes for a VM to become fully operational), for ensuring system stability by preventing (1) fast-succeeding transitory scale-in and scale-out actions, and (2) oscillations in the number of VMs.

For the scale-in operation it is notable that the VM selected for shutdown (with lowest utilization value) is not immediately terminated, but first its broker is informed about the scale-in operation for preventing new load being sent to the VM and then after a given time period T (e.g. 10 seconds), during which tasks running in the VM will get a chance to complete, the VM is finally terminated.

Algorithm 4.2: SLA Service Scaling algorithm

Data: per tenant: SLA scaling thresholds, monitoring information
Result: per tenant: scale-out, scale-in VM operations

```
while not at end of simulation do
    foreach tenant e do
        calculate average value m_SLA of SLA metric m over the
        sliding time window W
        m_SLA ← average(m(t), m(t−1), ..., m(t−W));
        calculate SLA ratio sr for metric m: sr ← m_SLA / max_SLA(m);
        evaluate scale-out condition:
        up ← (sr > s^UP) AND (sr(t) > sr(t − W_S));
        evaluate scale-in condition:
        down ← (sr < S^DOWN) AND (sr(t) < sr(t − W_S));
        calculate scaling speed ss as the number of scale-out or
        scale-in operations in the last time window W_S;
        if (up = true) AND (ss < ss^MAX) then
            create new VM;
        else if
        (down = true) AND (ss < ss^MAX) AND (count(VM) > 1)
        then
            select vm for shutdown with lowest load;
            inform VM's broker of imminent shutdown for preventing
            sending to workload to VM;
            wait for T seconds before shutting-down the VM;
    schedule next scheduling check;
```

4.4.2 λ-Based VM-Scaling Algorithm

Algorithm 4.3 describes the steps taken for sizing the number of VMs based on the current average arrival rate of requests, the current average system's throughput (Th), the maximum value for the average execution time defined by SLA, and the benchmark-obtained value of the Th. This algorithm improves the performance of Algorithm 4.2 by using a more precise way of determining the optimal workload that a VM can handle, as described next.

The algorithm receives as input the SLA containing the maximum execution time (W_{max}) across all the VM instances of the considered service. As part of the initialisation sequence, the algorithm will first search in the tuples (λ, W, L) displayed in Fig. 7.13 for the benchmark entry (e) with the average execution time ($e.W$) closest, but lower, than the SLA threshold (W_{max}). The maximum value for the throughput will be stored in Th_{max}, and if no such value exists, then the program's execution will be terminated

Algorithm 4.3: λ-Based VM-Scaling Algorithm

Data: SLA and *benchmark* table containing (L, W, Th, λ) tuples
Result: per tenant: scale-out, scale-in VM operations

```
1  W_max ← W defined in SLA and Th_max ← 0;
2  for e ∈ benchmark do
3      if e.W < W_max AND e.Th > Th_max then
4          Th_max ← e.Th;
5  if Th_max = 0 then
6      terminate execution
7  repeat every N seconds
8      if λ > 0.8 vm Th_max then
9          vm* ← ⌈ λ / (0.8 Th_max) ⌉;
10         if vm* > vm + vm+ then
11             out ← vm* − vm − vm+;
12     else if λ < 0.3 vm Th_max AND time(last scaling) < cool-down
       then
13         vm* ← ⌈ λ / (0.8 Th_max) ⌉;
14         if vm* < vm − vm− then
15             in ← vm* − vm − vm−;
16     if Th > 0.8 vm Th_max then
17         vm* ← ⌈ Th / (0.8 Th_max) ⌉;
18         if vm* > vm + vm+ + out then
19             out ← vm* − vm − vm+ AND in ← 0;
20     perform scaling
```

in line 6.

The scaling algorithm will be executed every N seconds. In line 8, the average arrival rate (λ) during the last minute will be compared to 80% of the maximum throughput (Th_{max}) multiplied with the current number of VMs (vm). If $\lambda > 0.8\, vm\, Th_{max}$ then in line 9 the number of VMs (vm^*) necessary for processing this workload will be calculated as the upper part of the division of λ by $0.8Th_{max}$. If vm^* is greater than the current number of VMs (vm) plus the current number of VMs being instantiated (vm^+), then the current scale-out scaling step (out) is calculated as the difference between the planned number of VMs (vm^*) and the total number of VMs, including the ones being instantiated ($vm + vm^+$).

Similarly, the conditions for scale-in are checked in line 12. If λ is lower than 30% of the maximum throughput of all VMs then the optimal number

of VMs (vm^*) is calculated in line 13 as upper part of the division of λ by $0.8Th_{max}$. If vm^* is lower than the current number of VMs (vm), minus the number of VMs currently being decommissioned (vm^-), than the VM scale-in step is calculated as $vm^* - vm - vm^-$. The scale-in operation is executed only if there was no scale-out or scale-in in the last *cool-down* seconds (e.g. 30).

Finally, in line 16, the average throughput (Th) during the last minute is compared to 80% of the maximum throughput across all VMs, and if it is larger, then the planned number of VMs (vm^*) is calculated as the upper part of the division of Th by $0.8Th_{max}$. If vm^* is larger than the current number of VMs (vm), plus the number of VMs being instantiated (vm^+) and the planned scale-out, then the new scale-out step is $vm^* - vm - vm^+$. This prevents that the current processing capacity of all VMs is exceeded by a rapidly increasing incoming workload.

Next, the VM manager will be informed (in line 20) about either scaling-out or scaling-in the number of VMs. In case of dEIS, the scale-in operation is coordinated with the LB service, so that no further workload is directed at the VMs selected for decommission.

4.4.3 Predictive λ-Based VM-Scaling Algorithm

In case of "reactive" scaling, the monitoring algorithm detects the scale-out or scale-in conditions and then it informs the VM manager for performing the scaling. However, as the creation of new VMs is not performed instantly, there will be a time window during which the system will be in a state in which the SLA-defined conditions might be violated.

Given the delay in instantiating VMs, it is beneficial to predict the conditions for scale-out and and to initiate the scale-out operation in advance so that the VMs are already operational at the time they will be needed.

In order to test our assumptions, we modified Algorithm 4.3 to include the prediction of the arrival rate, and to trigger scaling-out the VMs before the actual workload reaches the scale-out condition. Algorithm 4.4 lists the details of the predictive scaling algorithm. As multi-step prediction method we used Algorithm 4.1.

We initialise the algorithm in lines 2 to 4 by first determining the value for the maximum throughput (Th_{max}) (in line 1) given (1) the maximum execution time (W_{max}) defined in the SLA, and (2) the benchmark-obtained value for the Th. Also, the regression model (RM) is set to null, and the number of predicted VMs ($Pred$) is initialised with an empty set.

Next, the management loop starts, by repeating the following operations every N seconds. Let t be the current time in seconds. In line 5, the values for λ in the time window $(t, t - M)$ are retrieved and stored in the vector Λ. T is then set to the sampling time of the values in Λ.

If RM has already been calculated (is not $NULL$), then it will be used

Algorithm 4.4: Predictive λ-Based VM-Scaling Algorithm

```
   Data: SLA contract with the maximum execution time
 1 RM ← NULL;
 2 Pred ← empty;
 3 determine Th_max;
 4 repeat every N seconds
 5 |  Λ ← {λ(i) | i < t − M};
 6 |  T ← sampling(Λ);
 7 |  if RM is not NULL then
 8 |  |  MAPE = accuracy(RM, T, Λ);
 9 |  |  if MAPE < ε_max then
10 |  |  |  T* ← (t + 1, t + 2, ··· , t + N + D);
11 |  |  |  Λ* ← predict(RM, T*);
12 |  |  |  Pred[T*] ← ⌈ Λ / (0.8 Th_max) ⌉;
13 |  |  else
14 |  |  |  Pred ← empty;
15 |  |  |  drop RM;
16 |  |  end
17 |  end
18 |  RM ← regression(Λ, T);
19 |  if Pred[t+D] exists then
20 |  |  out ← max(0, Pred[t + D] − vm − vm+);
21 |  end
22 |  if out = 0 AND vm+ = 0 then
23 |  |  calculate scale-in step using another algorithm;
24 |  end
25 end
```

for predicting the arrival rates corresponding to the time moments in T, and then the out-of-sample accuracy ($MAPE$) will be calculated using the actual values from the vector Λ (composed of individual λ values). If $MAPE$ is below a threshold (ϵ_{max}), then a prediction (Λ^*) of the next $N + D$ seconds will be calculated, where D is the time necessary for a VM to be instantiated and to become operational. Next, in line 12 the number of VMs at time $(t + 1, t + 2, \cdots, t + N + D)$ is calculated using the method described in Section 4.4.2, Algorithm 4.3. If the prediction accuracy is higher than ϵ_{max}, then the predicted number of VMs is dropped, ensuring that no scaling decision is taken based on unreliable information.

Next, in line 18, the regression model RM is calculated using Λ and T, by applying the method described in Algorithm 4.1, where the current values for Λ and T are appended to the previous ones, as long as their trend

is maintained, as explained in Section 4.3.

Next, in line 19 a check is made for determining if a prediction exists for time $t + D$. By looking at the necessary number of VMs D seconds in advance, we ensure that any VM needed in the near future will actually be ready at that time. If the prediction exists, then a scale-out step is calculated as the difference between the predicted number of VMs ($Pred[t+D]$) at time $t+D$ and the total number of VMs (vm), including the ones currently being instantiated (vm^+).

If the algorithm determines that no scale-out is needed and there are no VMs currently being instantiated, then in line 23 the scale-in step will be calculated using the Algorithm 4.3.

Chapter 5

Dynamic Composition and Optimization of SLAs

In the previous Chapter 4 we have provided a solution to the research question of scaling services according to performance constraints defined in SLAs. In this chapter we are focusing on providing a solution to the question of discovering the performance limits of distributed applications constrained by the number of services/VMs, as well as dynamically composing SLA-constrained scaling rules for distributed services.

After describing in Chapter 4 the architecture and algorithms of the SLA-based service management platform, we present in this chapter the architecture and control algorithms of a system for optimising and dynamically generating SLA-bound scaling rules for controlling allocation of distributed services. For this purpose we combine data analysis mechanisms with application benchmarking using multiple VM configurations, as presented in Section 5.1.

The research question that we will answer is: how can the operator's management system select the optimum amount of resources allocated to its multi-tenant applications, so that distributed applications will offer the agreed performance levels? For answering this question we implemented the component presented in this chapter, the Dynamic SLA optimizer (*DynSLAOp*), which is able (1) to measure the performance increase of a distributed application using an increasing number of service instances, (2) to determine the correlation between the performance indicators of the services composing the distributed application and to use these correlations for calculating a service scaling model and (3) to transform these scaling models into SLA scaling rules.

5.1 Dynamic Composition of SLA Scaling Rules Based on Concurrent Benchmarks

We investigate how SLAs can be dynamically optimized for enhancing the rules controlling the scaling-out (increasing the number of VMs allocated to a distributed service) and scaling-in (reducing the number of VMs allocated a distributed service) of services belonging to distributed applications, in particular Enterprise Information Systems (EIS). We describe a mechanism for discovering the dependencies between the service metrics contained in the SLAs. First, we generate allocation profiles for the services of the distributed application and we instantiate virtual machines accordingly. An *allocation profile* maps each service of the distributed application to a given number of VMs (e.g. load balancer service uses one VM, business logic service uses four VMs, adn the storage service uses two VMs). Next, we perform a set of benchmarks typical for the application domain, where we continuously increase the number of concurrent incoming requests. Following that, we apply linear regression on the obtained time series in order to learn the dependencies between the different service metrics contained in the SLAs. Finally, we convert the obtained linear models (LMs) into SLA scaling rules, by adjusting the number of distributed service instances according to the value obtained by evaluating the LMs.

We use analysis of time series to determine a Pareto set [127] of scaling solutions that cannot be further improved with respect to *all* features/metrics. The solutions of the Pareto set can be used to create new SLA rules and to schedule SLA actions on distributed applications. We apply a heuristic algorithm for selecting a service scale-out pattern, which is ensuring that the agreed application performance level is met (the details of the algorithm are given in section 5.1.1), which is then used for controlling the number of VMs allocated to the distributed services.

The evaluation of our models is presented in Chapter 7 and contains SLA-driven service scaling experiments using a distributed application representative for the enterprise information systems. We also present how time series analysis can be used for determining dependencies between the services composing the distributed application and how these dependencies can be used for enhancing the SLA rules controlling the application scaling.

5.1.1 Dynamic SLA Optimizer (DynSLAOp) architecture

Intuitively, the new *DynSLAOp* component answers the following question: How to dynamically decide how many distributed service instances are needed in order to solve the problem of handling a larger workload within the same amount of time it takes to handle a lower workload with a lower number of service instances (e.g. so that the execution time should remain constant)? Therefore, we need to identify the relevant regression parameters

in the dependency between problem size, quality metrics and combination of workers (VMs). We then describe the metrics as functions of the number of workers and the workload.

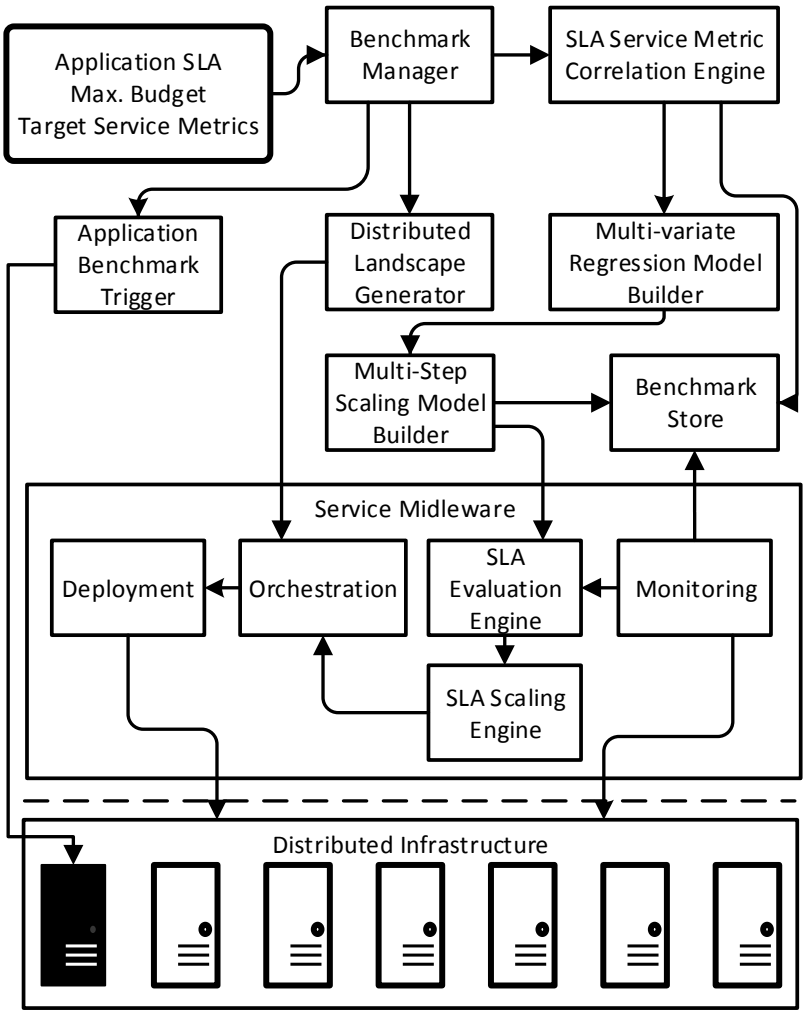


Figure 5.1: DynSLAOp System Architecture

The goal of *DynSLAOp* is to enable the service manager to dynamically adjust the number of VMs to the shifting application loads, while maintaining the contract with the user, i.e. the SLA. This is achieved by producing SLAs containing scaling rules for the application services with a variable number of instances, given (1) a semantic representation of the distributed application, (2) the maximum scale-out factor (maximum number of VMs that should be started during one scale-out operation) for each service and (3) critical SLA performance indicators. An example of managed distributed applications is described in Chapter 7, Section 7.1.

The information flow through the DynSLAOp occurs in several stages, and is represented in Figure 5.1. DynSLAOp takes as input to the **Benchmark Manager** an *SLA* with the service descriptions of a distributed application and a critical SLA performance metric along with its critical threshold. The benchmark manager communicates with the **Distributed Landscape Generator** for preparing a different number of virtual infrastructures (or landscapes) composed of different numbers of VMs allocated to each service (within the service cardinality described by the Application-SLA). Each virtual infrastructure contains a monotonically increasing number of VMs.

For each distributed service topology, corresponding VMs will be deployed on the physical infrastructure by the **Orchestration** and **Deployment** components of the SML, including the process of resolving the configuration dependencies between the VM-hosted services (e.g. IP addresses).

Once the virtual infrastructure has been provisioned, the **Application Benchmark Trigger** will be notified, which will (1) identify the VM hosting the generator of application requests, followed by (2) triggering generation of an increasing batch of concurrent user requests (creation of reports based on database queries), until the requests' average execution time, as reported by SML's monitoring component, will reach a given critical threshold.

The collected monitoring data is stored in the **Benchmark Store**. Once all the virtual infrastructures have been benchmarked, the **SLA Service Metric Correlation Engine** will begin calculating the correlation coefficients between the time series corresponding to the monitoring metrics defined in the SLA and the time series corresponding to the critical performance metric. The metrics with the correlation coefficient above a given threshold (e.g. 0.85, which according to [53] represents a "very strong" correlation) will be selected as the set of predictors for the critical SLA performance metric. The identified set of predictor metrics will then be given to the **Multi-variate Regression Model Builder**, which will calculate a linear model for estimating the critical SLA metrics, using the specified set of predictors.

Finally, the **Multi-Step Scaling Model Builder** will take the calculated linear models and will combine them with the information about the number of service instances in the corresponding virtual landscape for calculating a linear model for estimating the number of services in each landscape based on the linear estimation of the critical SLA metric (including the application workload metric - e.g. number of concurrent user requests). The final step consists in selecting a scaling path of virtual landscapes for the distributed application, by applying a heuristic described in Section 5.1.2 and then combining the linear models for the selected virtual landscapes for creating the final scaling models for each service type defined in the application description.

The following section describes the algorithms used by DynSLAOp components.

5.1.2 Dynamic Generation of SLA Scaling Rules

The generation of the SLA scaling rules is performed by three phases:

1. application benchmark data collection
2. benchmark data analysis (aggregation and monitoring metrics correlation calculation)
3. building the service scale-out model (by analysing the application scaling path and determining the service scaling model)

Application Benchmark Data Collection

The monitoring information is collected from probes running in each VM. Each probe periodically records the value of a single SLA metric and sends the gathered data at fixed intervals to the monitoring component in the DynSLAOp. The system observes via the monitoring subsystem both the values for the metrics related to the application load (e.g. the number of requests), and the values indicating the system performance (e.g. query execution time).

In order to properly stress the application performance, a benchmark data profile is created containing a snapshot of the monitoring database and the application topology. This includes all the measured values for the SLA-defined service metrics. The actual application benchmark is packaged within a VM and deployed along with the other VMs of the application. The decision of having the benchmark running on a separate VM was taken in order to ensure the independence of the DynSLAOp framework from the benchmarked application. The DynSLAOp is only aware of the API used for starting the benchmark and for checking its status for determining whether the benchmark has completed.

Benchmark Data Analysis

The monitoring data for each service instance is aggregated (e.g. averaged) into specified time intervals (e.g. 10 seconds). The data for the same service type is further aggregated, by averaging it. This is done under the assumption that the load is equally distributed between the instances of the same service type.

Once the monitoring data has been aggregated, the correlation between all the pairs of time series corresponding to the SLA service monitoring metrics will be calculated, producing a matrix called *correlation matrix*. Starting from the given critical SLA metric, a set is formed from the metrics that are highly correlated (e.g. correlation coefficient higher than 0.7). This is then repeated for each metric in the set.

The set of metrics is then added to the benchmark profile, along with (1) the maximum number of concurrent user requests determined by the benchmark and (2) the topology's configuration in terms of VMs per application service.

After the benchmark was run on all the configured application-topologies, the system enters the last phase for determining the services scaling-out model.

Building the Service Scale-Out Model

During this phase, the system will first construct an application "scaling path" and then will calculate a linear model for estimating the number of services required for achieving the selected maximum performance. The scaling path is composed of a sequence of VM allocation profiles (e.g. the number of VMs allocated to each service) able to handle an increasing number of concurrent user requests, while maintaining the defined SLA contracts, as presented in the algorithm below. A *virtual landscape* is composed of the set of VMs required for running a distributed application with an arbitrary number (n) of services, and it corresponds to an *allocation profile*.

For example, an application topology with one instance of the *database* (DB) service and one instance of the service containing the *business logic* (BL) can serve 90 concurrent requests with an average execution time of 500ms, while an application topology with two instances for each service can serve 150 concurrent requests with the same average execution time. The sequence of tuples (1 DB, 1 BL, 90), (2 DB, 2 BL, 150) is the scaling path required for keeping the average execution time under 500ms for a workload of up to 150 concurrent requests.

The optimal landscape is selected by calculating a utility cost function for each landscape and choosing the landscape with the minimum cost value. The algorithm might return multiple scaling paths of equal cost.

The maximum common set of predictor metrics is chosen from the selected of benchmark profiles. For each scaling path, using the selected collection of metrics, a larger data set is formed as follows. The aggregated values of the monitoring metrics corresponding to the first landscape in the scaling path are added to the data set. For the next scaling-landscape, the monitoring time series corresponding to the selected aggregated monitoring metrics is chosen such that the timestamps of the included samples are higher than the time when the concurrent user requests reached the maximum value achieved for the previous landscape. This creates a single data set with an increasing number of maximum supported concurrent user requests. To this data set, the number of service instances in the landscape is

added as a new time series.

Algorithm 5.1: Scaling Path Composer

```
1  ScalingPath = {L_min} where L_min is the landscape with the
   minimal number of service instances of each type
2  SI is the minimum factor by which the user-requests should be
   increased
3  VMDifference ← 1
4  repeat
5      Sel = {} is the set of selected landscapes
6      L_last ← last landscape in ScalingPath
7      UR_max ← maximum concurrent user requests(L_last)
8      for L ∈ Landscapes do
9          if VMDifference(L, L_last) < VMDifference AND
           maximum concurrent user requests(L) > (UR_max + SI) then
10             Sel ← Sel ⋃ {L}
11         Landscapes ← Landscapes − {L}
12     if IS EMPTY Sel then
13         VMDifference ← VMDifference + 1
14     else
15         select optimal landscape from Sel VMDifference ← 1
16 until NOT empty Landscapes;
```

Next, the system computes for each service type a linear model [70] for estimating the number of services in the scaling path.

Assuming that C is the critical value of the SLA target metric m^* (e.g. average execution time measured at the database service), $m_i, i \in (1..p)$ are values of p SLA predictor metrics for m^*, v_j are number of VMs associated with service $S_j, j \in (1..s)$ then

$$\begin{aligned} &lm_k(m_1, m_2, .., m_p, v_1, v_2, .., v_s) = \\ &\alpha_0 + \textstyle\sum_1^p \alpha_i m_i + \sum_1^s \beta_j v_j, j \neq k \end{aligned} \tag{5.1}$$

Equation 5.1 defines the linear model lm for estimating the number of service instances of type S_k. The actual SLA scaling rule for service S_k is then written as in Equation 5.2

$$\begin{aligned} SLA_k^{out}: &\quad if\ lm_k > v_k\ then\ \text{scale-out}(S_k) \\ SLA_k^{in}: &\quad if\ lm_k < v_k\ then\ \text{scale-in}(S_k) \end{aligned} \tag{5.2}$$

Equation 5.2 defines two SLA rules per service type S_k, which continuously monitor the estimated number of service instances lm_k and perform either a scale-out, or a scale-in if the required number of VMs for handling

current application workload is either greater or lower than the actual number of VMs v_k.

5.2 Analytic Optimization of SLAs for Scaling of Distributed Services

In the previous section we have presented a framework for automatic generation of SLA-based service scaling rules. However, SLA management is a multi-objective constraint problem, where various guarantees such as service response time and availability per customer need to be managed together with operational objectives such as cost, energy consumption and utilization, which inevitably are reflected in customer pricing.

In a distributed, multi-provider, multi-customer, cloud environment there are different categories and classes of infrastructure and application services running in parallel and contending for the same physical resources. Correlation of service dependencies is hence important in effective SLA management for understanding the relations between the distributed services and the impact on services' performance guarantees.

Identifying services with similar behaviours and access patterns provides hints that they should either share the same resources or they should use different resources and configurations, depending on the operational circumstances (e.g. depending on the desired performance, such as obtaining maximum performance under heavy load, the services should use different physical resources). For example, services communicating often together should be collocated (such as services containing the business logic of an enterprise application), while the VMs hosting the database layer of a enterprise application should better use different physical resources because of their high requirements in terms of CPU, memory, disk and network usage.

Discovering statistical correlation between service performance indicators helps better understanding the causes of performance bottlenecks in cloud applications [30], and thus understanding SLA violation causes. By using correlation information it also becomes possible to (1) refine service monitoring, (2) perform SLA levels health checking (e.g. examining if the SLA condition has been violated), and (3) improve scaling of cloud applications. By applying estimation models on groups of performance indicators with strong statistical correlations, it becomes possible to express critical SLA parameters as function of multiple measurable service indicators, thus (1) describing analytical SLA dependencies and (2) allowing automated management of services based on expressed dependencies between the distributed services.

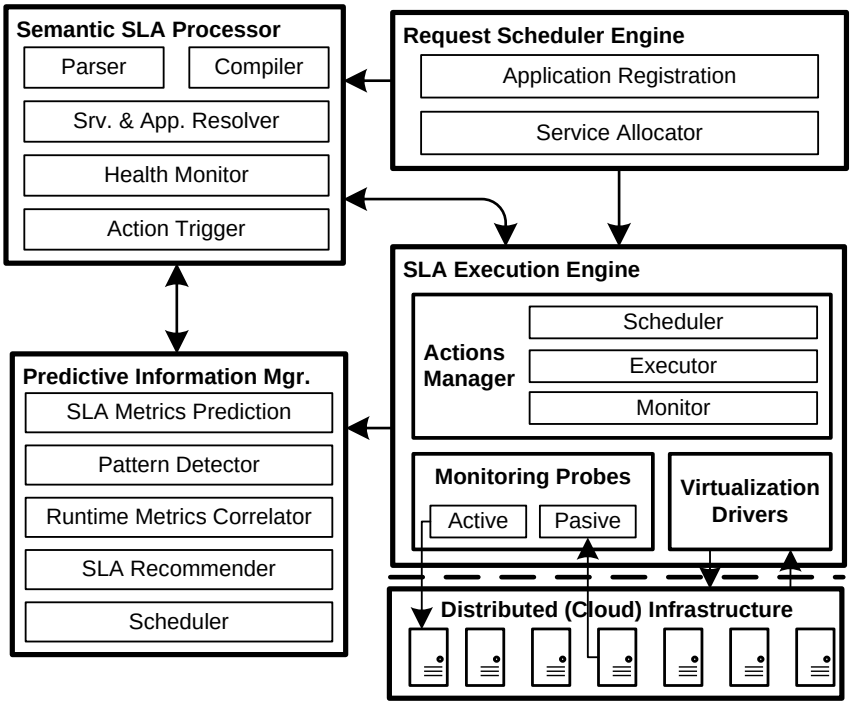


Figure 5.2: System Architecture

In order to solve the optimisation problem described above, we have envisioned a system for analysing both SLAs and the runtime state of the distributed services, and, based on this information, for optimising the resource allocation in the cloud environment. The architecture of the SLA Management and Resource Allocation Optimisation is composed of four subsystems, as shown in Figure 5.2. The detailed presentation of each subsystem is given in the following four sections. The subsystems have been designed to separate four standard lifecycle parts of SLA management, processing, scheduling, information management and execution, as well as the predictive mechanisms that we introduce. The subsystems have clear interfaces and interactions, such that these could be changed for different technologies or methods. The subsystems are described in more detail below, focusing on their SLA management relevance to support the dynamic, correlation-enhanced approach.

Semantic SLA Processor

The first component in the SLA processor is the **parser** responsible for (1) loading the SLA's RDF[88] into statements that are understandable by the **compiler** and for (2) registering the distributed application description to the information model.

The **compiler** converts the SLA-actions and SLA-guarantees into exe-

cutable Java code. Each action specified in SLAs are compiled into MVEL expression language, such that an interpreter can execute the actions.

As actions are executed, the **Service and Application Resolver** maintains a mapping between the higher-level application SLA-defined services and the VMs created from the execution of SLA actions (e.g. service scaling). It also maintains links between application services and the appropriate VM instances, such that the correct monitoring information can be extracted at runtime.

The next function of the semantic SLA processor is the **Health Monitor**, which checks values of the monitored metrics against the conditions defined in the SLA rules. This is done on a predefined control loop interval, which includes the collection of metric values from various probes. In addition, predictive states are also gathered from the *Predictive Information Manager*, discussed in Section 5.2, such that the future health of the managed services is also considered, according to the guarantees defined in the SLAs.

Finally, the **Action Trigger** executes the action associated with the violated SLA guarantee. The actual execution of the SLA-action will be performed by the *Execution Engine*, as described in Section 16.

Predictive Information Manager

This subsystem (1) predicts time series corresponding to the SLA monitoring metrics, (2) detects time series periodicity, (3) determines runtime on-demand metrics statistical correlations used for improving the existing SLAs, and (4) schedules SLA actions.

The correlation coefficient r is defined according to Equation 5.3, where x and y are sample values, $\bar{x}$ and $\bar{y}$ are variables' average value, n is the number of monitoring samples in the considered time window, σ_x and σ_y are the standard deviation of each variable. We used the correlation coefficient for determining the relations between the different monitoring metrics of the services found under SLA control.

$$r = \frac{1}{n-1} \sum_{i=1}^{n} \left(\frac{x - \bar{x}}{\sigma_x} \right) \left(\frac{y - \bar{y}}{\sigma_y} \right) \tag{5.3}$$

By starting from a set of SLA metrics specified by the application designer, the system can identify pairs of service monitoring metrics with high correlation (e.g. with $r > 0.85$), as described in the Algorithm 6. The system can then compose chains of such correlated metrics, forming the base for identifying inter-services dependencies. Such chains of monitoring metrics can then be further processed by fitting certain mathematical models (e.g. linear models) onto them, which can then be converted into SLAs. The resulting SLAs can be used for better monitoring of the cloud services and for understanding the complex relations between the various service performance indicators. Also, the models for estimating SLA service metrics

can be used for predicting critical application parameters and for avoiding those application states which would cause undesirable effects (e.g. poor application response times).

Algorithm 6 was implemented for identifying SLA correlations.

Algorithm 5.2: Correlation Chain Detection

```
1  for ST ∈ Service Types do
2      for M ∈ Metrics(ST) do
3          for Srv ∈ Services(ST) do
4              Aggregate M(t) time series using a given time window TW
5          Aggregate time series of all Services of type ST
6  for (X, Y) ∈ {Metrics × Metrics} do
7      Calculate correlation(X, Y) according to Eq. 5.3
8  Pairs = all (X, Y) pairs with correlation(X, Y) > C
9  Fit linear model LM over Pairs
10 Convert LM into SLA
```

The algorithm performs in lines *(1)-(5)* the aggregation of time series for each SLA metric and service type ST. This is because all instances of a given service type are assumed to perform identical functions and thus the application load is equally distributed among the active service instances. The time window TW over which the aggregation is performed is selected so that it is meaningful from a business point of view (e.g. one service billing period).

Lines *(6)-(7)* are concerned with calculating the statistical correlation between all the aggregated SLA monitoring time series. In line *(8)* a set of highly correlated SLA metrics is identified, given a certain correlation threshold (e.g. 0.7). In line *(9)* the identified set $Pairs$ is used for fitting a linear model [47] to estimate a critical SLA parameter.

$$\alpha_0 + \sum_{i=1}^{\|M \in Pairs\|} \alpha_i m_i(t) < thr_k \tag{5.4}$$

In step *(10)*, the linear model LM is converted into a SLA by adding a constraint (e.g. smaller than a given value) on the estimated SLA metric, as presented in Equation 5.4.
α_0 is LM's intercept term, $\|M \in Pairs\|$ is the number of SLA metrics in the set $Pairs$, $m_i(t)$ is the measured value of SLA metric m_i, α_i is the coefficient of m_i in LM, and thr_k is the user-selected maximum value for the m_K SLA metric.

The system has also an assisted mode of determining patterns in the time series of monitoring metrics. For this, the user indicates to the system a set of service monitoring metrics. The system then displays the auto-correlation and periodigram plot for the selected data, also suggesting whether the data

manifests periodicity, based on the auto-correlation information. The user can then decide whether the indicated period is correct, in which case the system would enable the use of prediction for SLA actions triggering.

The actual forecast is performed using the Holt-Winters algorithm [85], which is performing a triple exponential smoothing of the data, in combination with the *forecast* package [75] from R data analytics engine [129]. Given that the cloud-deployed applications experience daily, weekly and seasonal variance in their utilization levels, using a prediction algorithm (such as the Holt-Winters) that incorporates both seasonality and trend in its estimation improves the forecast accuracy over long periods.

Algorithm 5.3 describes the use of prediction inside the Predictive Information Manager's component. In line *(1)* the system identifies the SLA metrics marked in the SLA as expressing periodicity. The period T_i of these metrics is calculated in line *(2)* by applying statistical auto-correlation on the accumulated aggregated monitoring time series. Steps *(3)-(16)* try to apply the prediction on the suitable service levels SL (e.g. SLA service guarantees and SLA action triggers).

In lines *(4)-(8)*, the prediction of the periodic metrics in SL service level is calculated. If SL contains non-periodic SLA metrics, then SL is not further processed. In lines *10-16*, the SL's Boolean condition is evaluated over a number n of moving windows of length w (defined in SLA). n is defined as the number of time windows w fitting inside the smallest period of metrics in SL. For noise rejection purposes, the Boolean expression contained in SL is evaluated using the average value of m_j over the given time window w. If SL's expression evaluates to *true*, then the SL's action is scheduled for execution at the beginning of the i-th time window. It is up to the scheduler of the Execution Engine to filter multiple consecutive schedules of the same SL action.

In case of SLA levels triggering service scaling-up, the scheduling information can be used by the cloud resource allocator for reserving resources, also taking into consideration other factors and policies, similar to the ones presented in [19, 22]. However, exploring these scheduling aspects is beyond the scope of this thesis.

Request Scheduler Engine

This subsystem has two functions. First, it maintains **application registration** information that includes the time frames when services in each application should be available. For example, an application could be registered at time t_0 but only required to be operational at time t_n. Second, each service belonging to an application needs to be allocated resources in the managed infrastructure. The **service allocator** (1) determines required infrastructure virtual resources (e.g. cpu, memory) for each service, based on the sizing associated with each VM and (2) delays the VM instantiation

Algorithm 5.3: SLA Prediction

identify the set of periodic SLA metrics $m_i^p \in \{M^p\}$ (as indicated by application administrator in SLA)
system uses auto-correlation of $m_i^p(t)$ for identifying its period T_i
for $SL \in ServiceLevels$ *containing periodic metrics* **do**
    for $m_j \in Metrics(SL)$ **do**
        if $m_j \in M^p$ **then**
            update prediction(m_j)
        else
            predictive evaluation of SL is not possible, continue to next SL
    t is current time
    for $i \in (1..n)$ **do**
        evaluate SL with predicted data in interval the $[t + (i-1) * w, t + i * w]$
        if $expression(SL) = TRUE$ **then**
            if SL *triggers an action* **then**
                schedule action SL at time $t + (i-1) * w$
            else
                inform user about potential future SLA violation

until all service dependencies are resolved.

SLA Execution Engine

This subsystem acts as an interface and adapter to the managed cloud infrastructure. It consists of **virtualisation drivers** for controlling the capacity of virtual machines and the network available for services. This enables **monitoring probes** to be directly attached to resources and service endpoints, such that metrics can be pulled from logs or periodically executed scripts that query the virtual and host OS. The **actions manager** schedules, performs and monitors the status of direct operations on resources.

5.2.1 Implementation of the OpenNebula Virtualisation Driver

The presented management system was implemented using Java and service technologies such as OSGi [6] and CXF [23], which were selected because of (1) their wide use in enterprise software environments and (2) of their support for interoperable, standard-based, loosely coupled service architectures. We used multiple MySQL [121] databases for storing the monitoring state of the distributed application and the logging information accompanying the

SLA evaluation. All the data analytics operations were implemented in R [129], as this gave us the flexibility of writing complex but modular data processing workflows, which we could then access from Java/OSGi.

The SLAs were represented using USDL-SLA [93] and compiled at runtime into directly-evaluable MVEL [34] code. Using USDL-SLA had the advantage of representing both the distributed application service structure, related monitoring metrics and SLA conditions in an easily (machine) readable format.

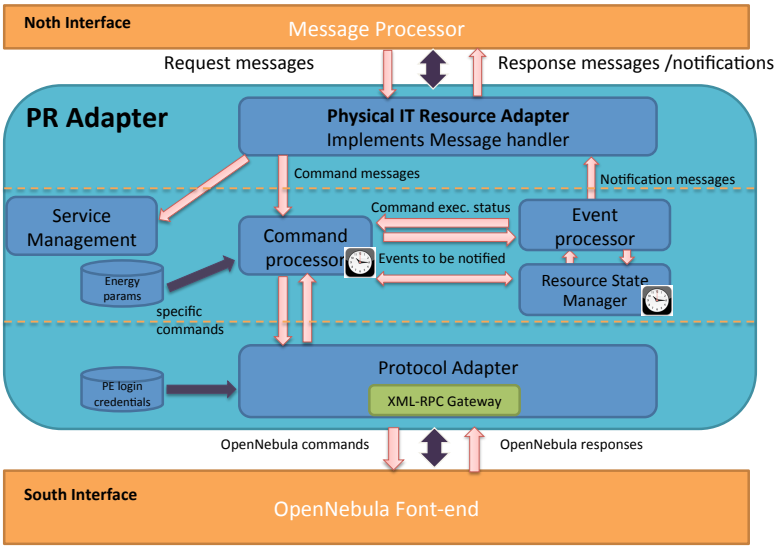


Figure 5.3: OpenNebula Execution Adapter developed in the GEYSERS project

Figure 5.3 displays the virtualisation adapter used for executing scaling actions using the OpenNebula cloud management system. The adapter implements common operations such as querying the available infrastructure resources, creating VMs, configuring VM parameters (e.g. IP address, application parameters such as server IPs), checking VMs' state, and terminating VMs. The implementation of this adapter was performed by the author of this thesis during the GEYSERS project.

Due to the asynchronous nature of executing the infrastructure operations, all commands are registered with an event processor, which is periodically checking for determining the status of the executed commands. When the execution of a command is completed, the adapter generates a notification message, which is then sent to the upper system using the north-bound interface. The actual communication protocol used for communicating with OpenNebula is XML-RPC[146].

Chapter 6

Dynamic SLA-Driven Virtual Resource Management with Forecasting Using Multi-Objective Optimizations

Having presented the architecture and algorithms of the service management platform (in Chapter 4), which uses the enhanced SLA specification described in Chapter 3, we will now show in this chapter how SLAs can be also used as input for efficient management of virtual infrastructure resources, in particular for the allocation of virtual machines to physical servers, while trying to optimize multiple objectives related to energy efficiency, costs and SLA penalties. Therefore, this chapter provides an answer to the research question, formulated in Chapter 1, regarding the optimisation of resource allocation in cloud environments. Specifically, we show in Section 6.1 how to design an algorithm for solving the resource allocation problem (e.g. the cost-efficient provisioning of virtual machines onto physical hosts), while considering multiple objectives coming from both from the infrastructure cloud provider and the customers using the IaaS services for running distributed applications. Next, in Section 6.2 we present the architecture of a system capable of implementing the previously defined algorithms in order to optimise the allocation of virtual machines in an infrastructure-cloud environment.

Cloud providers [8, 68] (of infrastructure or platform services) use SLA management for specifying the services' availability levels and for maintaining the quality of service (QoS) offered to their customers. An important phase of this process is the allocation of resources, also including initial and runtime placement optimization.

Dealing with exclusively-owned virtual machine (VM) instances deployed on a shared physical infrastructure presents a greater challenge, given the multi-objective optimisation problem (energy costs vs profits vs SLA penalties) introduced earlier (allocating VMs to hosts), as well as the differentia-

tion in demands from different classes of VMs and VM users. Furthermore, the violation of SLAs results in cash penalties for the provider, adding a direct economic dimension to the problem.

6.1 Resource-Allocation Planning and Utilisation Forecasting

A typical cloud environment consists of $h \in H$ servers each with a given amount of CPU cores c_h, main memory m_h, persistent storage s_h (usually network connected), and available network bandwidth b_h. These servers will be used for hosting one or more virtual machines, which will use a predefined amount of server resources, as determined by the service level defined (e.g. standard, gold, silver) in the SLA agreement.

6.1.1 System Model

The purpose of SLAs [83] [17] is to define the guaranteed configuration [8] of the VMs in terms of CPU, memory and network bandwidth and to also specify their hourly utilization tariff. They can also be used for defining the penalties in case of SLA violation [8, 68]. As such, the duration of the SLA violation during a month of utilization can be converted to a SLA penalty given as a percentage of the monthly bill. The penalty will be returned to the customer as a compensation for the suffered losses. This can be depicted using formula 6.1.

$$Penalty_{SLA} = \frac{a_i}{100}B,\ up \in (U_i^1, U_i^2],\ a_i \in [0, 100] \tag{6.1}$$

i is the penalty level, as shown in Table 6.1, $Penalty_{SLA}$ is the penalty cost calculated as a percentage a_i of the monthly bill's value B, when the uptime (in percent) up is between the thresholds U_i^1 and U_i^2. An example of such penalty calculation can be that 10% of the monthly bill (B) will be returned to the customer if the uptime is between 99% and 99.95%.

Penalty [%]	Lower availability limit [%]	Higher availability limit [%]
10	99	99.95
25	95	99
50	80	95
100	0	80

Table 6.1: Example of SLA penalties model

The estimated costs of violating the CPU or network SLAs is given by the time interval while the sum of estimated CPU/network utilization of each

VM exceeds the available resources of the hosts. Formula 6.2 determines the uptime value used in formula 6.1 for calculating the penalty value

$$\begin{aligned} up &= \frac{1}{T}\sum_{i=1}^{T/t_m} c_i^{cpu}\, c_i^{net}\, t_m \\ c_i^{cpu} &= \begin{cases} 1 & if\ u_i^{cpu} < U_i^{cpu} \\ 0 & otherwise \end{cases} \\ c_i^{net} &= \begin{cases} 1 & if\ u_i^{net} < U_i^{net} \\ 0 & otherwise \end{cases} \end{aligned} \tag{6.2}$$

t_m is the monitoring interval, T is the billing period (e.g. one month), u_i^{cpu} and u_i^{net} are the VM's CPU and network utilization level at the i^{th} monitoring time slot, U_i^{cpu} are the VM's host CPU and network utilization level (combined for all VMs running on the host) at the i^{th} monitoring time slot, c_i^{cpu} and c_i^{net} are the SLA compliance indicators at the i^{th} monitoring time slot.

For a given server, it is possible to model its energy consumption as a linear dependency of the CPU utilization [105]. In case of multi-core/CPU servers the average CPU utilization of all cores can be used. As a consequence of the fact that the idle power consumption is almost 60% of the one at full load [105], keeping servers in idle state or at low utilization would produce low revenues due to the fact that the server will consume almost as much energy as running with high load, but will generate only low revenues, if at all. Formula 6.3 describes the energy costs calculation, as described in [105].

$$\begin{aligned} P_i &= P_{idle} + (P_{max} - P_{idle})\,\overline{U_i^{cpu}} \\ C_{energy} &= C_{KWh} \cdot T \sum_{i=1}^{T/t_m} P_i \end{aligned} \tag{6.3}$$

where P_i is the power consumption during i^{th} monitoring time slot, P_{idle} and P_{max} are the host power consumption at idle and full load, C_{KWh} is the energy cost per KWh and C_{energy} is the total energy cost during T time interval.

In a cloud environment, usually the VMs experience low CPU utilization, with 30% average [105] and usually having daily, weekly and monthly seasonality [64]. This helps to predict the resource utilization and do a better allocation by taking into account the quantity of resources that a VM will use, in fact enabling using the virtual machine live migration as a load balancing method [150].

The actual costs of VM migration can be expressed by formula 6.4 which is the cost caused by blocking resources on the destination host during the

live migration, due to the reservation of memory before the migration. We ignored the VM downtime which is usually in terms of seconds.

$$C^{VM}_{migration} = C^{VM}_{uh} \cdot t_{migration} \tag{6.4}$$

where C^{VM}_{uh} is the cost per hour of utilization of a VM and $t_{migration}$ is the estimation of time in hours needed for performing the migration. Migration time is estimated using a linear dependency [150, 76, 114] between the amount of VM's reserved memory and CPU utilization, as expressed by formula 6.5. The formula could be extended with other factors, such as the average network bandwidth utilization, as the calculation method would remain the same: applying the superposition principle [119].

$$\begin{aligned} t_{migration} &= T^{Mem}_{idle} + \left(T^{Mem}_{f.load} - T^{Mem}_{idle}\right) \cdot \overline{u^{cpu}} \\ T^{Mem}_{idle} &= T^{M_{min}}_{idle} + \left(T^{M_{max}}_{idle} - T^{M_{min}}_{idle}\right) \cdot \alpha_{Mem} \\ T^{Mem}_{f.load} &= T^{M_{min}}_{f.load} + \left(T^{M_{max}}_{f.load} - T^{M_{min}}_{f.load}\right) \cdot \alpha_{Mem} \\ \alpha_{Mem} &= \frac{Mem}{M_{max} - M_{min}} \end{aligned} \tag{6.5}$$

where Mem is the amount of main memory reserved for the VM, $\overline{u^{cpu}}$ is the average CPU utilization of the VM at the migration time, T^{Mem}_{idle} and $T^{Mem}_{f.load}$ are the durations of the VM migration while it uses Mem amount of memory and its CPU is idling, respective, at full load. The T^{Mem}_{idle} and $T^{Mem}_{f.load}$ values can be either calculated by linear interpolation, by considering the time required for migrating a VM configured with the minimum, respective maximum amount of memory, at constant CPU utilization, either by directly measuring the live-migration time of a VM with the specified characteristics in terms of memory and average CPU utilization. The α_{Mem} is the percentage that Mem represents of the considered memory range [M_{max} , M_{min}].

We consider four objectives in our approach to allocating virtual resources: (1) *maximising* total revenues, (2) *minimising* energy costs, (3) *minimising* migration costs and also (4) *minimising* SLA penalty costs. These different objectives are linked by a aggregate objective function which evaluates the resource allocations using the function as described in Eq. 6.6.

$$\begin{aligned} F_{avg}(obj) &= w_r \sum_{i=1}^{M} \left(C^i_{uh} \cdot T^i\right) - w_e \sum_{i=1}^{H} C^i_{energy} - \\ &\quad - w_m \sum_{i=1}^{M'} C^i_{migration} - w_p \sum_{i=1}^{M''} Penalty^i_{SLA} \\ obj &= [w_r, w_e, w_m, w_p] \end{aligned} \tag{6.6}$$

obj is the evaluation objective composed of four weights: w_r for revenues, w_e for energy costs, w_m for migration costs and w_p for SLA penalty costs, M is the total number of VMs, C^i_{uh} is the per-hour utilization tariff of VM i, T^i is the utilization period of VM i during the given billing period T, and $M^{'}$ is the number of migrated VMs, $M^{''}$ is the number of VMs with SLA violations, and $Penalty^i_{SLA}$ is the SLA penalty cost of VM i. The actual composition of the objective vector *obj* is determined by the provider (e.g. he/she can chose whether to focus on energy efficiency or on maximisation of profits).

An example of a final objective could be maximization of the total profit, in which case each objective would have an equal importance, represented by giving each weight the value one (e.g. $obj = [1, 1, 1, 1]$). The variation in the weights can be also used for offering different price plans to the client, depending on the desired type of optimization. It might be the case that a provider might want to offer a 'green' service, with a strong emphasis on energy efficiency, in which case he will increase the corresponding weight of the energy costs (e.g. $obj = [.9, 1, .8, .7]$). This, however, will affect the other objectives, for example, the costs with the SLA penalties, as the allocator might select fewer physical servers for running the VMs in order to decrease the energy costs. Another possibility would be to offer a performance-oriented service, in which case, the weights corresponding to the migration costs and SLA penalties would be increased (e.g. $obj = [.9, .7, 1, 1]$), which would affect the produced allocations by using more servers and would raise the energy costs.

An efficient planning algorithm will try to find a global optimum for the allocation of VMs to hosts for a given billing period by maximizing the total revenues. Our proposed algorithm attempts to achieve this by performing a search of a local optimum with a time horizon of one billing period (e.g. one month).

6.1.2 Group-Oriented Genetic Algorithm for Multi-Objective Allocation of Virtual Machines

The problem of allocating VMs to physical hosts (while optimising the costs of the allocation in terms of energy efficiency, provider revenues, migration costs and SLA penalties) can be seen as a bin-packing or knapsack problem[104], which is known to be NP-hard, suggesting the need for a heuristic solution. Genetic algorithms [27] are a class of heuristic solutions that can make use of multi-objective evaluation functions for searching solutions in multi-dimensional spaces. a general introduction to genetic algorithms can be found in section 2.7.

Given the nature of the VM allocation problem, we selected a specialized version of genetic algorithms oriented at groups [78]. The group oriented genetic algorithm [54] operates on groups of objects, leveraging the fact that

the VMs are naturally grouped by the servers on which they are deployed and thus maintains the previously determined good properties of the groups.

The solution space of our VM allocation problem is multi-dimensional due to the multiple objectives used for evaluating them, such as the costs of SLA violation, energy consumption, VM migration and the total revenues. As the problem of allocating VMs to physical hosts is an NP-hard combinational optimization problem, it is not feasible to demonstrate the optimality of a solution. The produced solutions are Pareto-optimal [89] regarding the applied criteria. In our case, the criterion used for selecting a solution is the value of the predicted total profits of the allocation, considering the costs of the SLA violations, energy consumption and VM migrations.

Given that the average CPU utilization of a VM hosted in a data center is usually around 30% [105] it is common to overcommit the CPUs of the physical hosts with factors between 1.5 and 3 [105, 109]. For example, OpenNebula [113] allows allocating virtual CPUs to a fraction of the available physical CPUs, using support of Xen and KVM [38] hypervisors. This, however, could lead to violating the SLAs (e.g. guaranteed maximum values for service execution times) in case that the collocated VMs simultaneously experience high CPU utilization [109]. Given the fact that a server uses almost 60% [105] of the total power consumption when running idle, distributing VMs across a large number of servers results in poor server utilization levels and would hence also diminish revenues due to the high amount of energy used per VM. Our genetic algorithm allocator with load forecasting mitigates these problems for VMs with an existing monitoring history, by choosing the best current allocation that minimizes the costs of the allocation at the next time step, given that domain-specific constraints from the application are not breached.

Although it is possible to overcommit both CPU ([109, 152]) and memory ([152, 151]) in modern hypervisors, we are considering only oversubscription of the CPU, as the oversubscription of memory is usually associated with system instability [152].

Below, the basic structure of the genetic algorithm used is given.

The group oriented genetic algorithm [54] searches for solutions by generating populations of chromosomes composed of genes that belong to groups. In our implementation, a chromosome encodes an allocation of VMs by representing the groups as hosts and the genes as VMs packed onto a given group. The initial population is created by applying a heuristic algorithm, such as first-fit. Next, the population is sorted according to the multi-objective fitness function. At each step the algorithm performs two group oriented operations on the current population: crossover and mutation (described later in this section). Top 10% chromosomes from the current population are passed into the next population as *elitism* seems to improve

solution convergence [89].

Algorithm 6.1: VM Allocation Genetic Algorithm

Step 1: Generate the initial population

1. randomly allocate genes (VMs) to groups (servers) using the first-fit heuristic[4]
2. ensure the chromosome is valid with regards to VM's allocated memory by reinserting the excluded genes in the groups using the first-fit heuristic

Step 2: rank (evaluate the fitness of each chromosome) and sort the population using the fitness function defined in equation 6.6. The fitness function is evaluating how "good" each chromosome is at solving the problem. The actual function depends on the problem to be solved.
Step 3: copy E "elite" chromosomes (with highest fitness value) into the new population
Step 4a: generate an offspring (new chromosome obtained from one or more existing chromosomes) using crossover with a given probability

1. select two chromosomes from the population using fitness proportionate selection
2. generate offspring using the group-crossover operator
3. ensure the chromosome is valid (e.g. all VMs are still present in the allocation) with regards to VM's allocated memory by reinserting the excluded genes in the groups using the first-fit heuristic

Step 4b: or select the fittest of the parents
Step 5: mutate the offspring with a given probability
Step 6: rank and sort the population using the fitness function defined in equation 6.6
Step 7: If the stopping criterion is reached, terminate the search and return to the current population, else, go to *Step 3*.

The group oriented genetic algorithm [54] searches for solutions by generating populations of chromosomes composed of genes that belong to groups. In our implementation, a chromosome encodes an allocation of VMs by representing the groups as hosts and the genes as VMs packed onto a given group. The initial population is created by applying a heuristic algorithm, such as first-fit. Next, the population is sorted according to the multi-objective fitness function. At each step the algorithm performs two group oriented operations on the current population: crossover and mutation (described later in this section). Top 10% chromosomes from the current population are passed into the next population as *elitism* seems to improve

solution convergence [89].

We use two criteria for ending solution searching. First, we determine when there are no more improvements, or they are below a certain threshold, in the overall fitness value of a population. Second, we determine when finding a solution is not possible (e.g. the profits generated by the new allocation are lower than the initial allocation due to the costs of SLA violations, VM migrations and energy consumption).

Each allocated group inside each chromosome needs to have another local allocation applied for determining the allocation of physical CPU cores to the VM cores. This is achieved by running a similar genetic algorithm to Algorithm 6.1. The allocation is performed every time a group is changed (e.g. a VM is added or removed from the host).

The evaluation of each chromosome is performed by calculating the predicted energy costs for the allocation (using a linear power model[105] - defined in Eq. 6.3 - combined with forecasted CPU utilization data), revenues generated by the allocation assuming one period of utilization, predicted costs caused by CPU/network SLA violations (determined using forecasted utilization data) and costs associated with VM migrations - which are the values of the objective functions. These values will then be combined by the aggregate objective function, as described in equation 6.6.

The next step consists of applying the roulette selection [27] for identifying two possible candidates for producing the new chromosome. With a given probability either the group-crossover operator is applied for producing a new offspring, or the fittest chromosome if selected. After this, with a given probability, the mutation operator is applied to the offspring, before adding it to the new population.

After the population has been created, it will be re-evaluated (*Step 2* of Algorithm 6.1) and the process is repeated until the stop condition is encountered.

The group-oriented crossover genetic operator functions by retaining the qualities of the groups and selecting from each chromosome the corresponding group with the highest fitness value (as defined in Eq. 6.6), thereby preserving the good VM-allocations. After all groups have been processed it is possible for some VMs to remain unallocated. For these ones, a 'healing' process is applied by redistributing them according to first-fit-descending heuristic.

The mutation operator is applied by randomly removing a VM from a CPU- or network-oversubscribed host and re-allocating it according to the first-fit heuristic. A key component of the planning system is the forecasting module used by the allocator algorithm in evaluating the fitness of various VM to host distributions based on the forecasted VM request rate, CPU core and network utilization. Given the fact that both the VM request and resource utilization distributions experience daily, weekly or monthly patterns [64] we have selected the Holt-Winters algorithm [84] for performing

triple exponential smoothing of the utilization data.

The Holt-Winters algorithm performs an exponential smoothing of the data by assigning exponentially decreasing weights to the past data comprised of a period, considering also the data trend and seasonality. The seasonality refers to repeating of a data pattern after a given number of periods, called season. The trend refers to the tendency of data to either increase or decrease in the long term. In our case, the monitoring data obtained from the VM's CPU and network utilization, as well as the number or VM requests, experience seasonality with daily and monthly patterns [31].

6.2 Control System Architecture

The system functions in a control loop, as shown in Fig. 6.1. First, the requests are prepared by the *Landscape Manager* and then are passed to the *Allocator* module (1) for determining how the virtual machines should be allocated on the physical infrastructure. The *Allocator* uses historical data from the *Monitoring* module (2) and the system load forecast (3) for producing an allocation of the VMs to hosts. The actual load forecast is determined using historical monitoring data (4), using of specifically fitted forecast parameters calculated (5) by the *Forecast Parameters Optimizer* module. Next, the allocation is passed (6) to the *Multi-Objective Evaluator*, which then uses both existing monitoring data (7) and the forecasted resource utilization data (8) in order to calculate the costs involved in the allocation. After selecting the allocation with the minimal cost, the *Allocator* returns it to the *Landscape Manager* which will then register the new virtual resources with the *SLA Manager* (9) and will instruct the *Infrastructure Manager* (10) to actually provision the resources on the physical resources. The control loop is closed by the returning of the monitoring data to the *Monitoring Handler* (11), followed by informing the *SLA Manager* about the resources' state (12).

Three implementations of allocator algorithms were created in Java for the planner system: First-Fit-Descending[157], Best-Fit-Descending and the Genetic Group-oriented with Forecasting. The actual Holt-Winters forecasting is delegated to an external implementation of R[129] statistical computing software. The forecasted series are kept in a memory cache, as the prediction values are reused multiple times by the genetic allocator. The forecast is run in a progressive manner, first after on hour from the VM instantiation, then after 6h, 12h, 24h, 2d, 3d, 6d, 7d, 9d, 30d, after which the forecast is recalculated every month.

Our system makes some assumptions about characteristics of the incoming load, such as the distribution of load according to hourly and daily patterns, having one service instance per VM, and having a predictable trace of CPU and network utilization. We assume that the VM network utiliza-

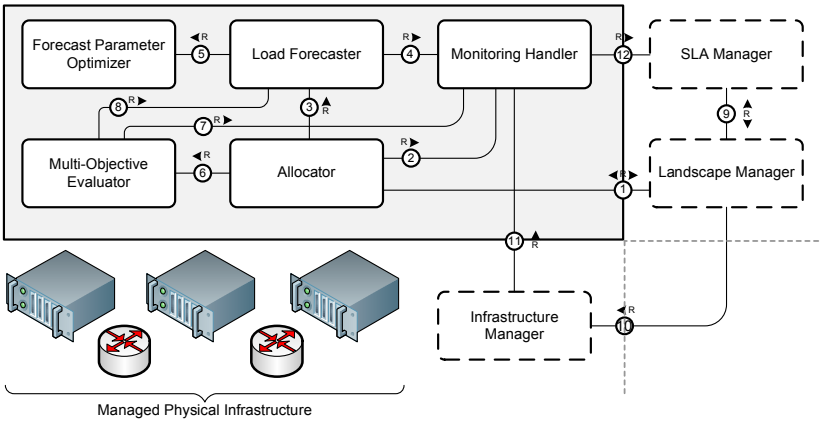


Figure 6.1: System Control Loop

tion data refers only to inter-hosts traffic but not to the intra-host traffic of the collocated VMs; and that the VM memory is always reserved all at once, while the VM's CPU cores are allocated to the physical CPU cores using the affinity mechanism [95]. We assume that the monitoring samples are collected every 5 minutes, in order to keep the generated data volume to a manageable value of approximately 9000 monitoring samples per VM for each month.

While the assumption of having just one type of service per VM seems restricting, this might be needed in an environment with automatic scaling in order to enable taking the decision on when the service should be scaled, based on previously agreed SLAs. This, however, does not prevent the existence of composite applications containing multiple services. An example of such application, with which we experimented, is an Enterprise Information System (EIS) composed of a load-balancer, a session handler, multiple workers and a storage service. We describe in Section 4.2 how the SLA-based System Landscape Orchestrator works to perform the dynamic instantiation, configuration and scaling of the services. Even if there are multiple service instances per VM, this does not change the nature of the VM-to-Host allocation problem, as the resource metrics would remain the same.

The penalty model used for calculating the costs of violating the CPU or network SLAs was already presented in Table 6.1. The penalty in percent refers to the amount of the current bill that will be paid to the customer in the next moth if the SLA availability is between the lower and the upper bounds. This implies that the target SLA availability is 99.95%.

For estimating the VM migration duration (Fig. 6.2) we used a linear model, validated against experimental data [76, 114], dependent on the amount of reserved memory and on the average CPU load of the VM. Fig-

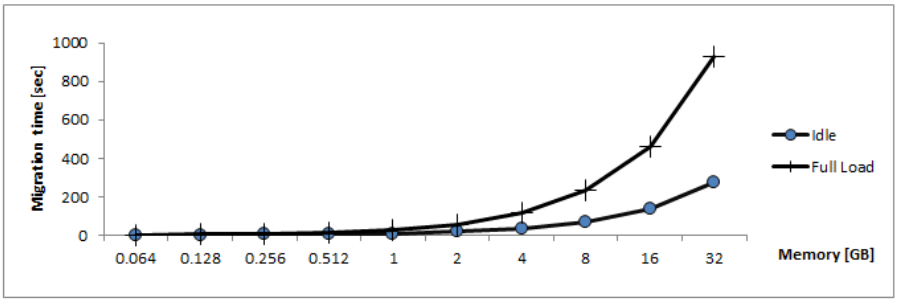


Figure 6.2: Migration time vs. memory and CPU load

ure 6.2 represents the linear dependency between the VM migration time and the amount of reserved memory for when the CPU utilization is near 0% (idle) and almost at 100% (full load). The actual migration time is determined by interpolating the time for the average CPU load using the values for the migration at idle and full CPU utilization. Also, we model the live-migration impact of the VM by increasing the load of the VM's CPU with 10% over a monitoring period, and also increasing the network bandwidth utilization with the value amount required for transferring the VM's reserved memory.

Chapter 7

Evaluation

We evaluated the work described in this thesis by performing multiple experiments with VM-scaling algorithms on several distributed testbeds, as well as several simulations based on analytic models derived from datasets gathered while monitoring a real-world distributed enterprise application.

The current evaluation focuses on assessing the benefits of using SLAs in cloud environments in management of both physical and virtual computing-infrastructures. The validation of presented methods and algorithms was performed by using a distributed enterprise application (dEIS), which we describe in Section 7.1. In Section 7.2 we present the architecture of the application under test, the performance profiling of the system, as well as the statistical analysis of the relation between the system's response time, average occupancy and the rate by which the external requests arrive at the system's input.

Having presented in Chapter 6 the application under test (dEIS), its performance characteristics, as well as the environment used for running the validation experiments, we continue in Section 7.3 with the first round of simulations meant to validate the use of SLA-based infrastructure management, as described in Chapter 3.

In Section 7.4 we describe the experimental physical testbeds, which were used for running both the static VM-provisioning and dynamic VM-scaling experiments.

In Section 7.5 we present the validation of the multi-objective virtual resource allocation algorithm introduced in Chapter 6. We describe a simulation of allocating VMs according to a periodic variable-workload based on common utilization patterns found in enterprise environments.

We split the evaluation and validation of the algorithms and methods presented in Chapter 5 into two parts, as follows. In Section 7.6 we present the experiments with VM provisioning and benchmarking onto the physical testbed for validating the concepts related to the dynamic composition of SLAs described in Section 5.1.

We continue the evaluation in Section 7.7 with a series of four experi-

ments, designed to validate the use of analytic mechanisms for SLA optimisation presented in Chapter 5.2 The first experiment was used for showing how analysis of correlations can help detect performance bottlenecks and incorrect SLA service scaling formulas. The second experiment validated the conclusion of the first experiment related to scaling policy, while the third experiment (1) validated the improved scaling policy based on the conclusions of the first two experiments and (2) confirmed the detected correlations. The fourth experiment validated the use of prediction in combination with auto-correlation.

Section 7.8 is dedicated to validating the VM scaling algorithms presented in Chapter 4, by using simulations of VM-scaling based on the monitoring traces of the dEIS application. In Section 7.8.1 we validate the simulation model described in Section 7.2.2, by comparing the measured performance metrics of the dEIS application running in the physical tested with the simulator's output. Having validated the simulation model of dEIS application, we conduct in Section 7.8.2 three simulations for validating the implementation of the reactive SLA-based scaling algorithm, which we described in Section 4.4.1. Finally, Section 7.8.3 presents the evaluation of the two VM-scaling algorithms described in Sections 4.4.2 and 4.4.3. We also compare the efficiency of the three previously presented VM-scaling algorithms. The last subsections deal with the large-scale simulations of the SLA-based VM-scaling algorithms based on an enterprise use case, concluding the validation of the work described in this thesis.

7.1 System Under Test: Distributed Enterprise Information System

For evaluating the presented approaches to SLA-driven orchestration for dynamic application topologies, we use a distributed Enterprise Information System (dEIS) with variable workload (number of users), payload sizes and classes of workloads.

A typical *dEIS* consists of the following tiers, each contributing to the SLA management problem: consumer, load balancer, business logic and storage layer. Figure 7.1 provides an overview of the overall *dEIS* topology and services. We shortly present the structure of the *dEIS* system used.

The four distributed services are interconnected by a Registry and Messaging architecture. The dEIS has the purpose of integrating benchmarks for processor and networking infrastructures. Is a distributed OSGi application with a distributed service registry for 4 service classes, where each service class has its own set of relevant, monitored metrics and control operations.

The **customer facing layer** implements the logic for generating the graphical user interface and allowing dEIS users to interact with the system. The *Consumer (CS)* VM is instantiated per client of the dEIS. It represents a

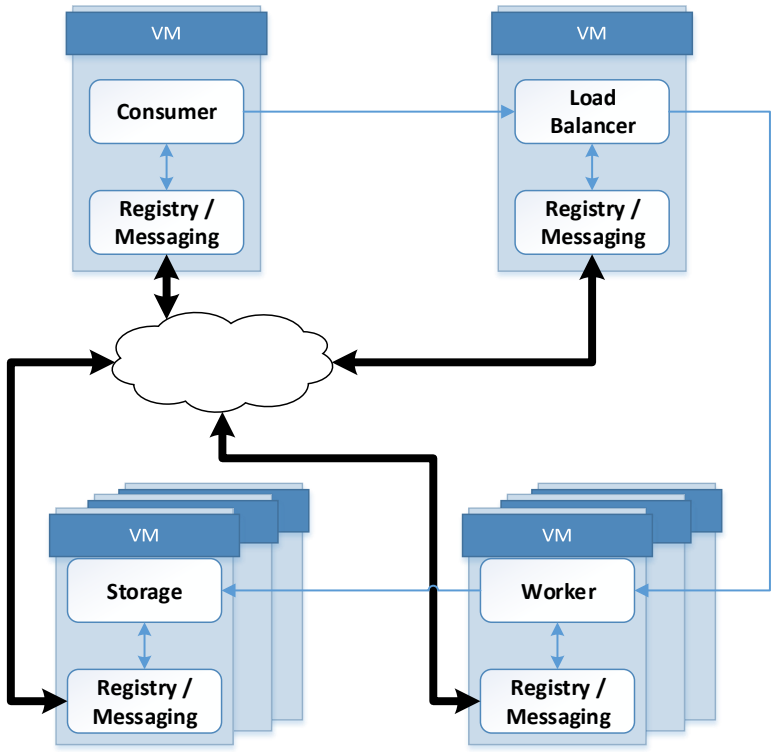


Figure 7.1: EIS Services Topology

remote thin-client deployed with proximity to the dEIS core, such that large payloads or transactions do not originate beyond the controlled network. The monitored metrics are the *number of users* and the *response-time* for requests. The number of *Consumer* instances is gradually increased and decreased using a *Load Generator*.

The **business logic** or **worker (WK)** contains components that implement the data queries, analysis algorithms, transaction execution and arithmetic processing of the enterprise application. Different types of queries, algorithms and arithmetic workloads place different demands on the available processing resources. They also impact on the end-to-end latency of user requests, such that scaling the amount of CPU and memory available to these services has a significant impact on the observed QoE and SLA compliance. There are various data-intensive benchmarks used, such as by the Transaction Processing Performance Council (TPC) [46], for creating different transaction or analytics workloads, each requiring processing of volumes of data representative of how these applications are used in everyday in business contexts.

The **storage (ST) layer** provides the interface to database resources, as

well as mechanisms for creating, reading, updating and deleting data. Given that the *dEIS* also maintains logs together with the growing application data, intelligent data management techniques such as compression are often required to avoid exceeding quotas and actual capacity. As input/output storage latency will have an impact on the end-to-end response time, such that parallelisation and redundancy are used to increase performance and availability.

The **load balancer (LB)** redirects incoming requests to appropriate workers based on algorithms, rules, and their state information. This depends on their current activities, resource consumption and priority of the request. Load balancers can also make decisions about starting and stopping workers in order to free physical processing, memory and storage resources for other activities. This enables a more efficient resource management of the underlying resources.

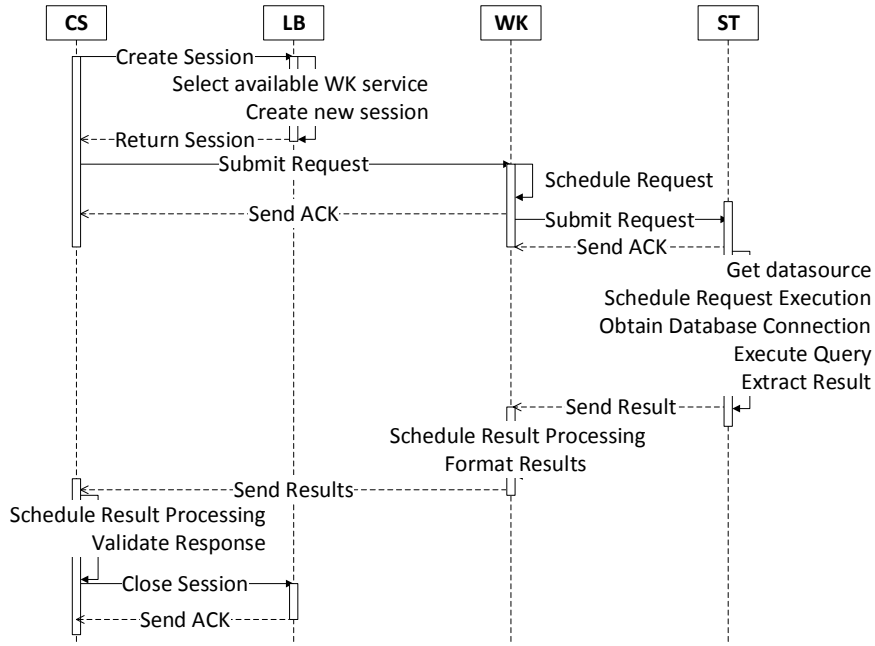


Figure 7.2: EIS Services Interactions

Fig. 7.2 presents the interactions between various dEIS services when executing a CS-initiated request. This also shows the breakdown of operations executed at each service, which serves as the foundation for building a simulation model of dEIS.

As each component, layer and service in the dEIS is distributed and autonomous, there is a need for a registration and messaging architecture that coordinates interactions between the distributed components and services. This also enables more comprehensive and consistent monitoring as each source, target and interaction can be discriminated and logged as an

individual record. The dEIS implementation used in our analysis and development uses the *Apache Zookeeper* [24] registry and synchronization service for the above-described functions.

The four *dEIS* application services were packaged into VMs, which were then deployed on a geographically distributed test-bed, described in Section 7.4. The dEIS was developed as a collection of OSGi bundles, which communicate using Distributed OSGi [25] mechanisms. Various monitoring probes were developed for each service type in order to ensure that services' state is observable from the central management system. The bundles for each service were packaged inside an OSGi container and were set for automatic start-up. The OSGi container was packaged inside an Ubuntu virtual machine, and a script was added to the VM for starting the OSGi container at boot time. The VMs were configured using a boot script for passing configuration information to the running services. The VMs were controlled using multiple OpenNebula [56] virtualisation controllers, chosen because of ability to abstract the underlying hypervisor and hardware resources.

7.2 Performance Analysis and Modelling of the Distributed Enterprise Information System

In order to simulate the dEIS application, we first recorded key performance indicators of dEIS services under constant workload. We then used these monitoring traces to create the dEIS's simulation model composed of a large number of single requests' execution measurements. For any given request we measured the round-trip times for the service calls displayed in Fig. 7.2, as well as durations of the service-local calls.

For building the application performance profile we took the following approach. We instantiated one VM for each (Consumer) CS, (Load Balancer) LB, (Worker) WK and (Storage) ST service. We initialized the concurrent *load* with 1 (active request at a time) and the benchmark time duration with 10 minutes. Every 10 milliseconds the number of active requests ar executed by the system was compared to the target $load$, and if ar was below $load$, then a number of $load - ar$ requests from CS would be generated. For each service request, the following parameters were recorded: round trip durations, service-local operation durations and VM-level performance metrics (CPU, memory, network, disk). By recording all these parameters for each distributed request, a performance benchmark profile was generated under the constant workload $load$. This whole process was then repeated for a load equal to the previous load plus 1, e.g. $load_{current} = load_{previous} + 1$.

After running the benchmark described above, the dEIS performance profiles for constant concurrent load between 1 and $max.load$ concurrent requests will be known and the performance profile will be generated with all service call round-trip times and service-local operation durations. This

profile will then be used during the simulation, as described later in this chapter.

In order to build the application's performance profile the following assumptions were made: (1) load balancing is applied at the LB service and (2) CS requests are independently executed by the dEIS services. This allows us to extend the performance profile recorded with one VM per dEIS service to any number of VMs, as the performance of individual services will be influenced only by the level of concurrent workload and not by the number of running VMs.

7.2.1 dEIS Performance Profiling

We analyze performance profiles generated for each dEIS service while the system was sequentially executing requests for 10 minutes, at a fixed concurrent workload (defined as the number of requests being processed by the system as any given time). The physical setup consisted of three servers with Intel(R) Core(TM)2 Duo CPU E8400 CPUs, 4GB of memory and 1 Gbps network interface. Figure 7.3 shows the placement of the VMs on the servers. CS and LB VMs were co-located on Server 1, while WK and ST were each located on Server 2, respectively Server 3. The fourth server was used for running validation experiments, by using two additional VMs, one VM for the WK service and one VM for the ST service.

In Fig. 7.4 we show the performance profile of the dEIS CS service (thin client). Four application metrics were recorded for the CS service: (1) round-trip time for creating a LB session and receiving a confirmation

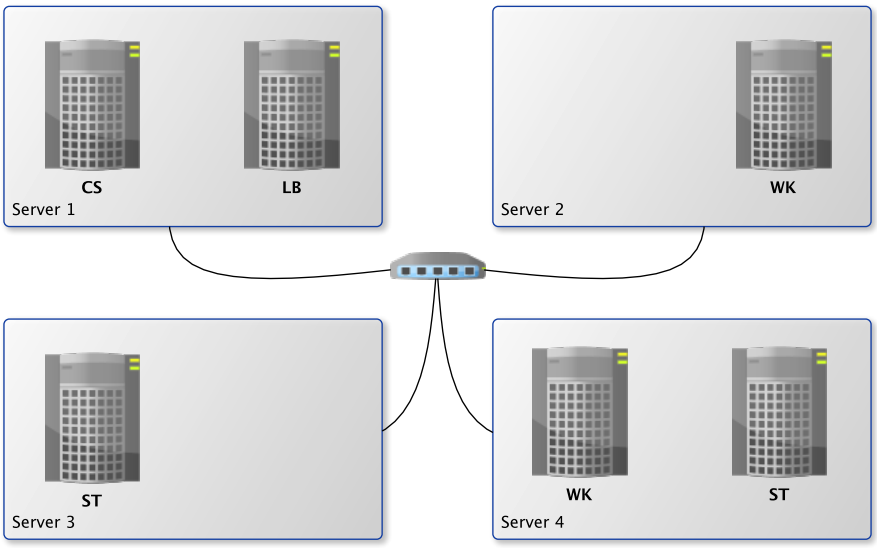


Figure 7.3: dEIS Topology (allocation of VMs to servers)

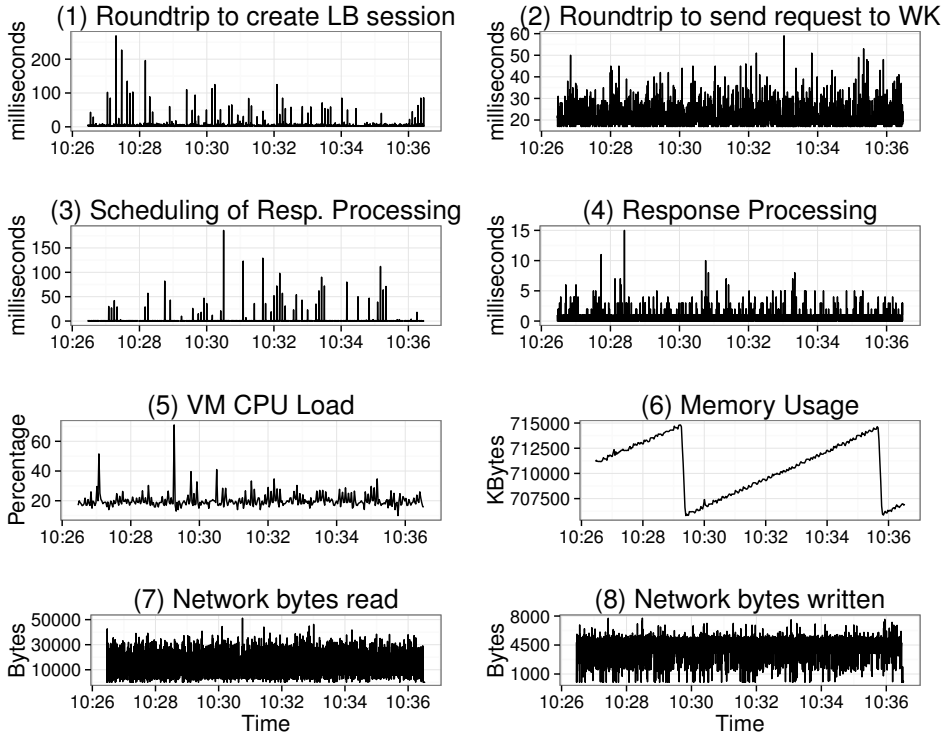


Figure 7.4: dEIS Consumer Service Performance Indicators

acknowledgement, (2) round-trip time for sending a request to the WK service, (3) the time during which the response was queued before being selected for processing, and (4) time for validating the response. For the VM performance there were also four metrics gathered: (5) VM CPU utilization percentage, (6) memory usage, (7) network bytes read, and (8) network bytes written. The network bytes read correspond to the messages received from LB (session creation confirmation, around 4KB) and WK (processing result, 10-30KB), and have a large variance. The CPU utilization has low values and is almost constant at around 20%, as expected for a thin client application.

In Fig. 7.5 we present the performance profile of the dEIS LB service. The application metrics recorded are: (1) session creation time, and (2) number of concurrent sessions. The LB VM metrics are the same as the ones gathered for CS. CPU utilization is also low at around 20%, justifying co-location of CS and LB VMs. Both network read and written bytes have a median of around 2KB and are caused by exchanging small sized monitoring and session creation/ confirmation messages.

In Fig. 7.6 we present the application performance profile of the dEIS WK service. We gather the following monitoring information: (1) request scheduling duration - time a CS-request is queued until it is selected for pro-

cessing, (2) round-trip duration of sending the WK-request to ST service, (3) ST-response queuing, (4) processing of ST-response, and (5) round-trip time for sending the response back to CS service and receiving the acknowledgment.

In Fig. 7.7 we display the WK's VM performance metrics. CPU utilization has high values because of the additional computing time required for processing the result from ST. Also, the size of the response sent to CS has a large variance (5-30KB), explaining the variance of the result processing time. Also, this rather large result size variance will influence the variance of the simulation results.

In Fig. 7.8 we present the ST's application performance metrics. The recorded metrics are: (1) time a WK-request is queued until it is selected for processing, (2) time it takes to obtain a database connection for processing the data queries, (3) database query execution time, (4) SQL result set processing time, (5) size of the result in bytes, and (6) round-trip time for sending the result back to WK.

In Fig. 7.9 we show the VM performance metrics associated with the ST service. Both CPU and memory utilization are high due to the large amounts of processing required for retrieving and processing the data from the database.

Finally, in Fig. 7.10 we present how the various durations and round-trip times stack up to compose the complete end-to-end time of each chain of requests (CS-WK-ST) and responses (ST-WK-CS). This plot represents the

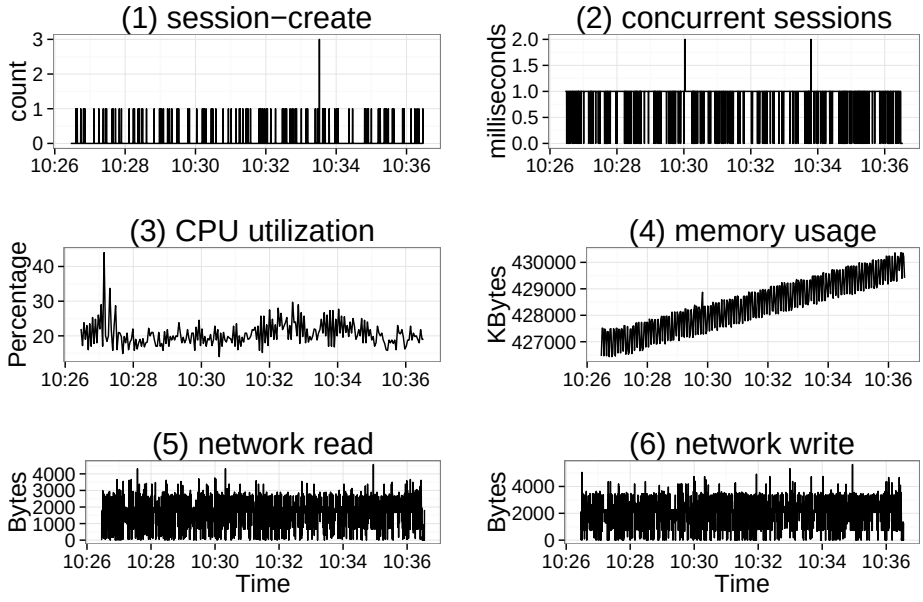


Figure 7.5: dEIS Load Balancer Service Performance Indicators

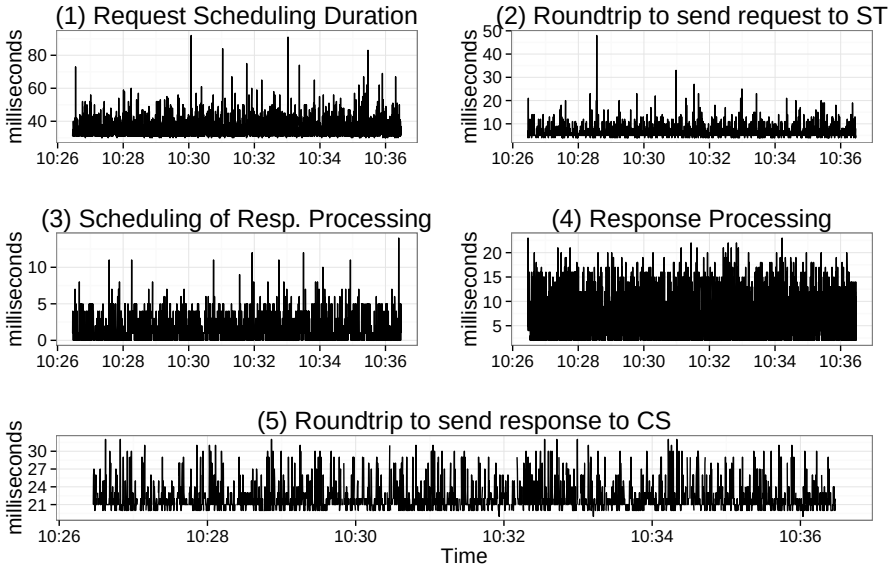


Figure 7.6: dEIS Worker Service - Application Performance Indicators

foundation of the simulation model. During the simulation, vertical "slices" (service-local durations and round-trip times) will be used for simulating CloudSim cloudlets. The probes displayed in the Fig. 7.10 are described in Table 7.1. Figure 7.10 should be interpreted as a plot of ene-to-end execution times on the CS, WK and ST services. For each request on the X axis, there

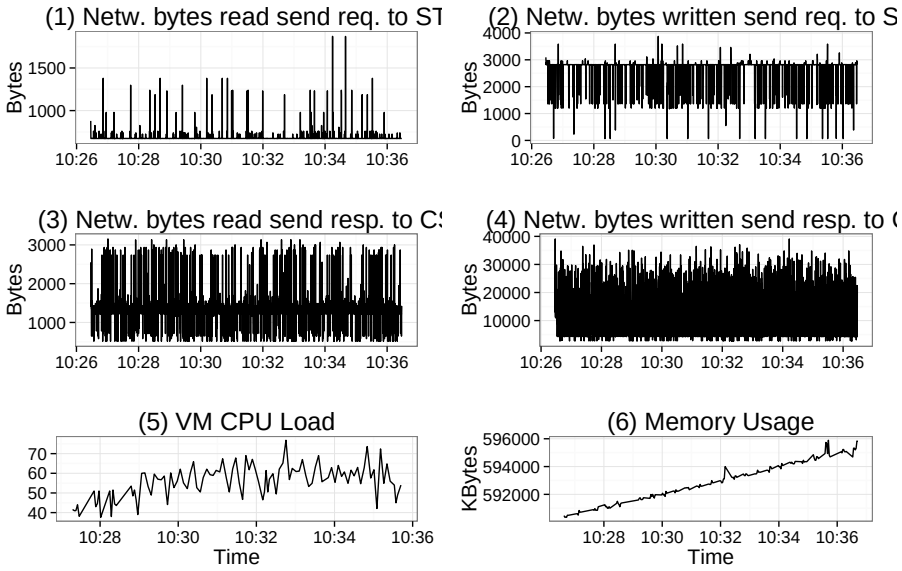


Figure 7.7: dEIS Worker Service - VM Performance Indicators

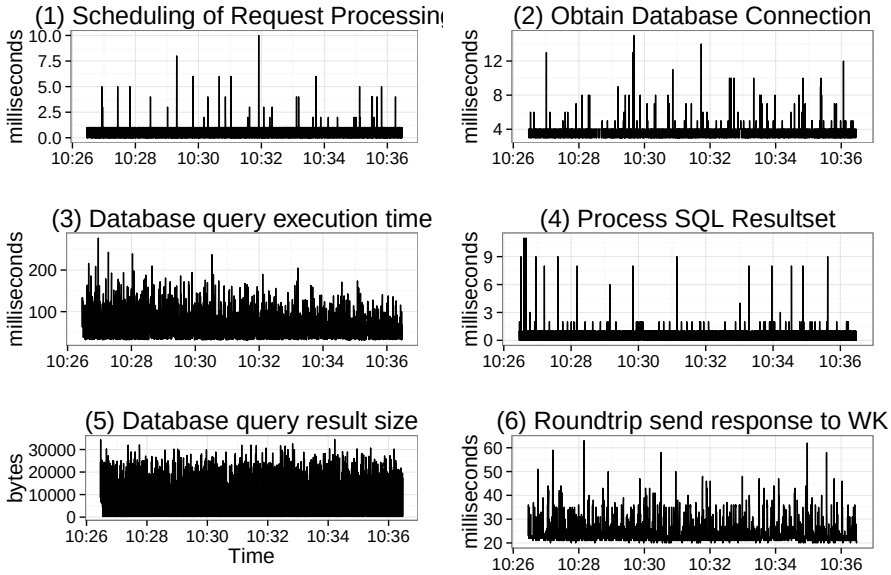


Figure 7.8: dEIS Storage Service - Application Performance Indicators (Service Related)

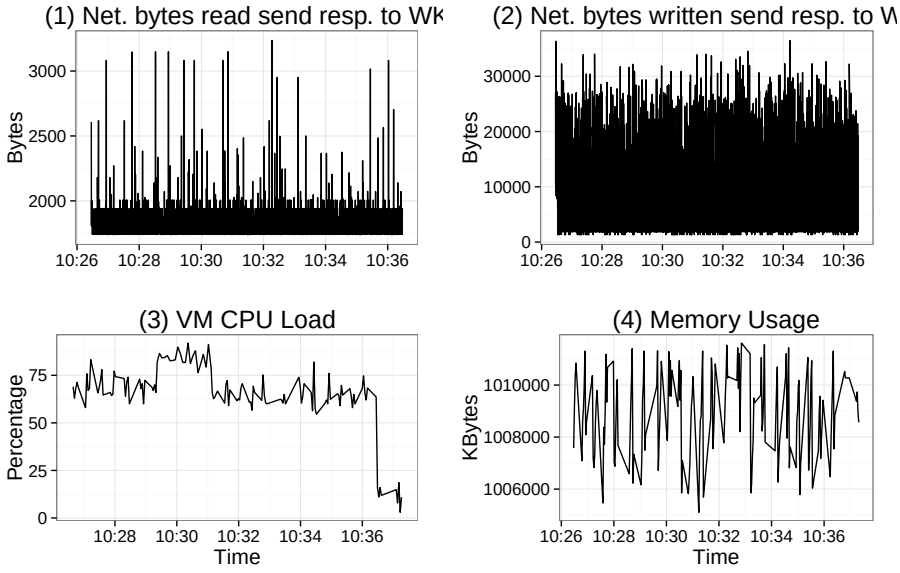


Figure 7.9: dEIS Storage Service - VM Performance Indicators (Network, CPU, Memory)

are multiple components (corresponding to the metrics given in Table 7.1) which add up to the final value of the measured end-to-end request.

Table 7.1: dEIS Monitoring Metrics (in milliseconds)

A	lb-create-session-roundtrip	B	cs-to-wk-send-request-roundtrip
C	wk-req-scheduling	D	wk-to-st-submit-request-roundtrip
E	st-get-datasource	F	st-exec-schduling
G	st-request-execution-time	H	st-result-processing
I	st-wk-send-response-roundtrip	J	wk-response-processing-scheduling
K	wk-req-execution-time	L	wk-to-cs-response
M	cs-schedule-respose-processing	N	cs-process-response

7.2.2 Modelling and Simulation

The performance-profiling traces previously gathered can be represented as shown in Equation 7.1. $Profile_{load}$ is a matrix with all measurable time values of applications' operations under concurrent *load*. $RT(S_i, S_j)$ is the round-trip time of a remote service call on service S_j from service S_i. $D_i(Op_k)$ is the time of performing service-local operation k on service S_i and $(cpu|mem|net)_{CS|LB|WK|ST}$ is the utilization level of CPU, memory and network on the VMs corresponding to CS, LB, WK and ST services.

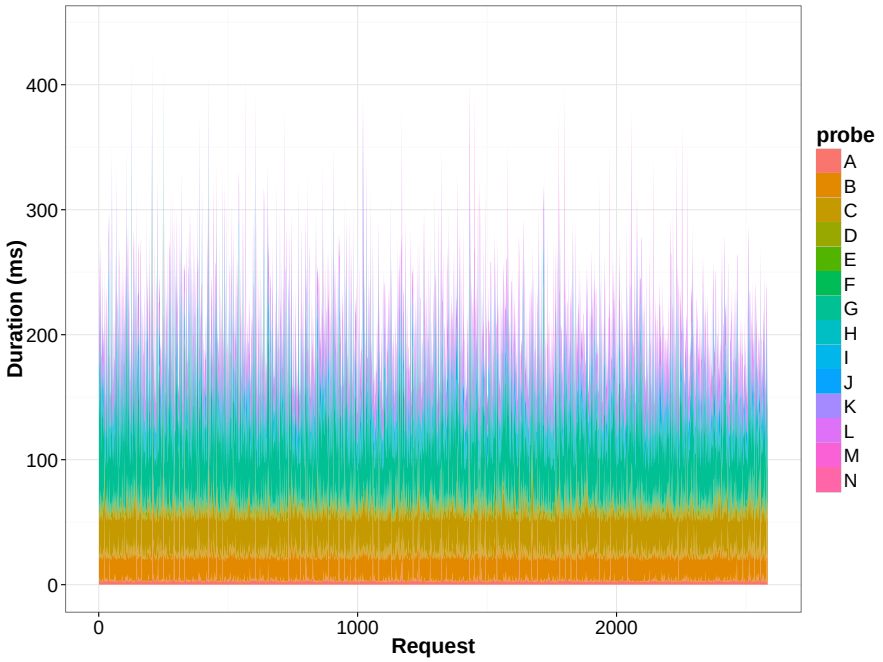


Figure 7.10: dEIS Decomposition of System Response Time

Each line of the $Profile_{load}$ matrix corresponds to a single chain of requests from CS, to WK, to ST, and then back to CS.

$$Profile_{load} = \left\langle \begin{array}{l} RT(S_i, S_j),\ D_i(Op_k), \\ (cpu|mem|net)_{CS|LB|WK|ST} \end{array} \right\rangle \tag{7.1}$$

By combining performance profiles for concurrent workload between 1 and $max.load$ end-to-end requests, we form the dEIS performance model $Profile$ as dictionary with key $load$ and corresponding values $Profile_{load}$.

In order to convert the created performance profile to CloudSim entities we transform the durations expressed in milliseconds to their equivalent MIPS rating. We also map the VM resource consumption for CPU, memory and network to CloudSim characteristics of the created cloudlets - CPU and memory *utilization model*s and network input and output payloads. The conversion of milliseconds to MIPS is done according to Equation 7.2.

$$instr = \frac{ms}{1000.0} \cdot CPU_{MIPS} \tag{7.2}$$

$instr$ is the length in instructions of an operation with a duration of ms milliseconds when executed on CPU with a MIPS rating of CPU_{MIPS}. This equation is valid only when there is only one task being executed on the CPU. If there are multiple tasks in execution on the same CPU, the tasks' length value in number of CPU instructions need to be reduced by the CPU concurrency level (number of tasks simultaneously being executed on the CPU) in order to keep the same execution deadline.

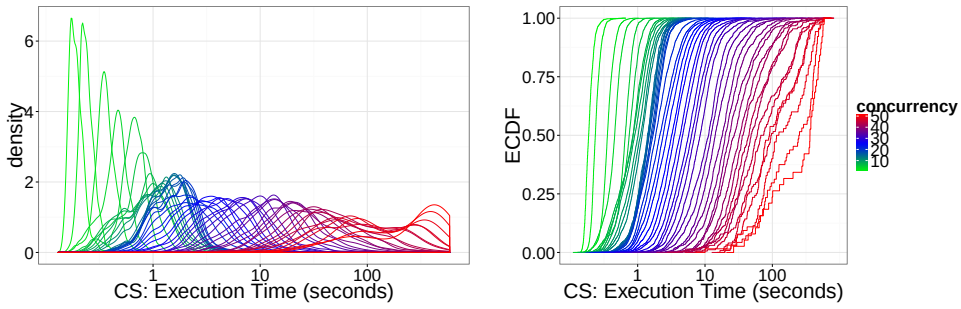
As our performance profile contains durations of operations measured under constant workload, we apply Equation 7.3 in order to convert each operation's duration in milliseconds to MIPS.

$$instr^* = \frac{instr}{cl_{new} + concurrency_{VM}} \cdot (1 - \overline{CPU_{OS}}) \tag{7.3}$$

$instr^*$ is the task's length in MIPS considering VM concurrent load, $instr$ is the value calculated by Equation 7.2, cl_{new} is the number of cloudlets newly submitted to the VM for execution, $concurrency_{VM}$ is the number of cloudlets (see Section 2.5 for a definition of the term *cloudlet*) already executed by the VM, and $\overline{CPU_{OS}}$ is the average CPU utilization caused by independent OS-level kernel tasks. This calculation is performed in each CloudSim Datacenter Broker corresponding to CS, LB, WK and ST services for each batch of cloudlets submitted for execution.

For running the simulation in CloudSim, a Datacenter Broker is created for each dEIS service and also at least one VM per dEIS service is added to CloudSim.

A *simulation scenario* is given as a list of pairs of simulation time and the corresponding expected number of concurrent CS-requests at that time. The list is then linearly-interpolated for determining the expected concurrency



(a) **(b)**

Figure 7.11: CS Service Execution Time vs. Concurrency. a) Density Plot. b) ECDF Plot

for each simulation time slot. For each simulation time step, the current number of running CS-requests cl is compared to the simulation plan values cl_{sim}, and if $cl < cl_{sim}$, then $(cl_{sim} - cl)$ CS-cloudlets are created.

Based on the expected concurrency level defined in the simulation scenario, one request trace is selected from the performance profile with the corresponding concurrency level. Next, from the performance trace, the operation duration, CPU, memory and network utilization are identified for each service-local operation and remote service calls, as shown in Fig. 7.2. These identified parameters are then used for creating CloudSim cloudlets. The chain of cloudlets is then submitted for execution, starting with the CS create-session operation. Subsequently, each cloudlet's completion will trigger the creation of a cloudlet for the next dEIS operation. Finally, when the last operation in the end-to-end request, the request execution time will be recorded.

7.2.3 Little's Law-Based Application Performance Benchmarking

In order to build a performance profile of the dEIS application we will use the results of applying Little's Law [96] to *stable* instances of the dEIS system. According to Little's Law, a *stable system* starts with load equal to 0 and finishes in a state with load also equal to 0. Intuitively, as long as the arrival rate (λ) of dEIS requests remains below the maximum processing capacity of the system, we expect the average execution time (W) to increase linearly. After a certain value of λ will be exceeded, W will begin increasing exponentially due to accumulation of jobs in the system, leading to a drop in the value of system's throughput (Th).

Mainly, we want to identify the dependency between the average execution time and the system's throughput at constant concurrency (occupancy). For this we run 50 batches of benchmark tests, where the overall system's concurrency is kept constant for 10 minutes at a value $\lambda \in (1...50)$, before

dropping back to 0. This ensures that we will get an accurate picture on (1) the distribution of execution times at all the dEIS services, and (2) the average achievable throughput corresponding to λ.

Figure 7.11 plots (on a logarithmic X scale) the execution time (W) in seconds against the concurrency (L) measured at the CS service. Figure 7.11a shows the density distribution function of the execution time, while Figure 7.11b shows its empiric cumulative distribution function (ECDF). It is easy to notice on the ECDF plot that for concurrency values above 30 all the execution times (W) are above 10 seconds. Above this concurrency level (L), any small increase in L will produce a very large increase in W.

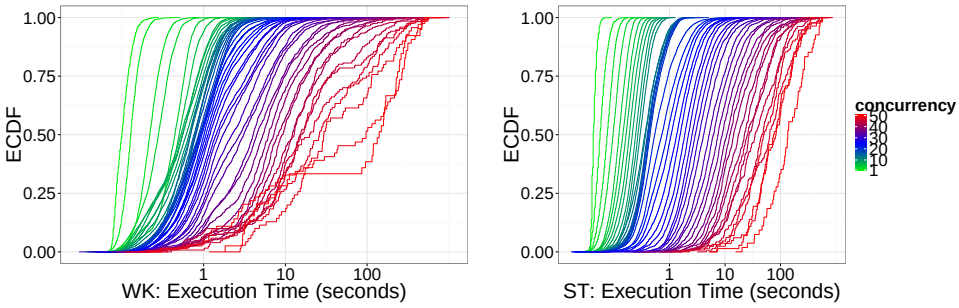


Figure 7.12: ECDF Plot of Service Execution Time vs. Concurrency. a) WK service. b) ST service.

Figure 7.12a further explains the rapid increase in the mean of the system's execution time after concurrency level 30, by plotting the ECDF distribution of W for each of the L levels, for the WK service. Similarly, Figure 7.12b shows the EDCF plot per concurrency level of W for the ST service. The plots also suggest that a management system should not allow the dEIS system to slide into a region with concurrency level (L) above 20, as the execution time will increase very fast with only slight increases in L. This corresponds to the dense green ECDF curves on the WK plot, respectively to the dense blue ECDF curves on the ST plot.

After executing the benchmark described in this section, we produce a lookup table for the WK and ST dEIS services, linking the average system's occupancy (L), average execution time (W), the average arrival rate λ, and the average throughput (Th). Later, in Section 4.4 we will present an algorithm using this lookup table as input for ensuring that the dEIS system is properly scaled so that it can handle the volume of workload directed at it.

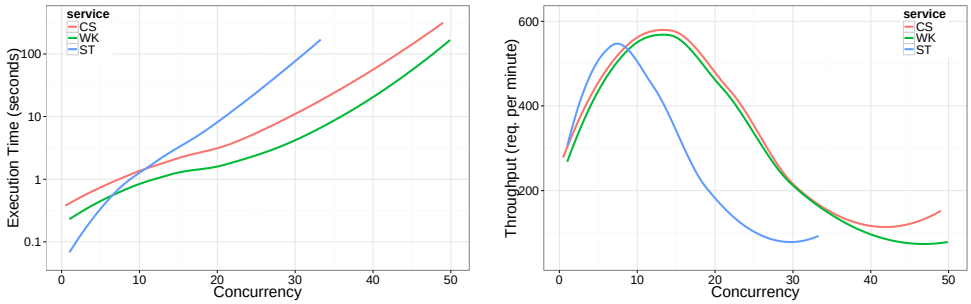
The first ten out of fifty entries from the lookup table are shown in Table 7.2, with the rest being shown in Annex 9.2. The entries in the table approximately obey Little's Law, which is due to how the average values of the W, λ, Th, and L metrics were calculated, especially the system's

WK				ST			
L	W	λ	Th	L	W	λ	Th
0.85	106.25	118.0	133.40	0.64	44.1	118.0	133.37
1.49	135.89	236.0	263.61	1.47	59.5	236.0	260.45
1.97	236.68	333.3	366.39	1.91	78.5	333.3	363.10
2.47	296.38	355.5	392.94	2.23	99.7	355.5	395.65
3.01	461.58	315.7	362.82	2.77	129.5	315.7	381.52
3.60	532.35	358.8	406.07	2.95	152.4	358.8	418.43
4.39	615.45	377.8	419.50	3.05	175.7	433.7	433.72
5.11	619.07	414.8	462.41	3.12	201.7	476.1	476.12
5.83	666.97	435.4	483.38	3.30	229.4	435.4	497.15
6.63	733.06	442.2	491.27	3.55	262.0	442.2	510.37

Table 7.2: Dependency between average concurrency (L), average execution time in milliseconds (W), average arrival rate (λ), and average throughput (Th) for WK and ST services

occupancy, defined as the number of requests being in execution as a given time. In the actual VM deployment, the metrics were averaged over a sliding time window of one minute. This, combined with the fact that the arrival rate was not constant due to the constraint of having the CS-occupancy (almost) constant, produced slight variations in the averaged values, which are not influencing the SLA-based scaling algorithms as only the values for W and Th are used as inputs.

However, it is worth observing that indeed, the execution time increases with the increase in system's occupancy. Figure 7.13a shows the relation between W and L for CS, WK and LB dEIS services, on a logarithmic scale for W. This points to the fact that W starts increasing fast after L equal to 15 measured at CS service. This rapid increase of W is explained by Figure 7.13b, which shows that the service's throughput starts dropping after a certain value of the service's occupancy (L). The change in Th's trend happens around $L = 15$ for CS and WK services, and around $L = 8$ for ST service. This delay between the occupancy's levels at ST and WK services is



(a) **(b)**

Figure 7.13: Per Service a) Average Concurrency vs. Average Execution Time. b) Average Concurrency vs. Average Throughput

due to the fact that a job is processed twice [12] by the WK service, but only once by the ST service, causing the occupancy at the ST service be lower than the occupancy at the WK service. This difference is not noticeable between CS and WK service because of the smaller processing time in the second computing round [12] at CS service.

7.3 Evaluation of Feasibility of SLA-Based Infrastructure Management

In order to evaluate the feasibility approach of performing SLA-driven orchestration for dynamic application topologies presented in Chapter 3, we used data gathered from simulating the distributed Enterprise Information System (dEIS) presented in Section 7.1 with variable user numbers, payload sizes and classes of workloads. In this section we analyse the value model and alternatives of doing SLA-driven service orchestration. We the present a comparison of various service allocation management strategies.

7.3.1 Value Model and Alternatives

The value model of the evaluation scenario is defined against two objectives: (1) *maximise* the assurance that the response time guarantees for all *Consumers* will be met and, (2) *minimise* waste of resources to spawn and maintain multiple worker *Instances*.

To remain competitive, infrastructure providers need to maintain their reputation for satisfying SLAs, while minimising their operational costs and maximising their operational efficiencies. They want to accommodate existing customers, fluctuations in demand and new customers. Given that cooling of data center equipment and connectivity costs are the highest contributors to service operation costs, an increase in efficiency has significant impact on both objectives. From these two objectives we derive three evaluation parameters discussed in the following paragraphs.

The first evaluation parameter is the **R**eaction **T**ime rt for a service, such as creating a new *EIS Instance*, is the sum of detection time dt, planning time pt and action time at required by an administrator or management system to respond to a potential incident, an undesirable state or an opportunity to enhance operations. The **average reaction time** $\bar{rt}$, expressed in equation 7.4, is the sum of detection, planning and action times for a series of adaptation triggering events $e_1, ..., e_n$, divided by the number of events n.

$$\bar{rt} = \frac{1}{n} \sum_{e=1}^{n} dt_e + pt_e + at_e \tag{7.4}$$

The second evaluation parameter is the average number of **S**LA **V**iolations $\bar{sv}$ across a list of observed service states $(s_1, ..., s_n) \subseteq S$, with cardinality

$|S|$, where S is the set of observable states and each $s \in S$ is a collection of metrics $M_s \subset M$. The set S is extracted from logs and from live monitoring probes.

$$\bar{sv} = \frac{|S - GS|}{|S|} \tag{7.5}$$

Finally, the third evaluation parameter is the **P**rovisioning **E**fficiency pe of the infrastructure considers the ratios of consumed resources R^t_{cons} and provisioned resources R^t_{prov} for the EIS application topology over a time series $t \in (t_1, ..., t_n)$. The PE is a measure of the resource provider's ability to satisfy demands, while minimising resource costs and waste. The statistical trends (PE_{mean}, PE_{max}, PE_{min} and PE_{mode}) in PE over the time series $(PE^1, ..., PE^n)$ are of interest in assessing the performance of alternatives to service management, as shown in Equation 7.6.

$$pe^t = \frac{R^t_{cons}}{R^t_{prov}} \tag{7.6}$$

There are three fundamental alternatives that can be derived, considering independent optimisation strategies for the three objectives stated in the value model. (1) **Minimise reaction time** rt**:** over provision to avoid need to monitor and adapt the initial deployment; (2) **Minimise SLA violations** sv**:** over provision to avoid violations; (3) **Maximise provisioning efficiency** pe**:** provision for mean or modal workload. Different configurations of these can lead to other alternatives but these 3 make for a comprehensive feasibility evaluation against the SLA-driven orchestration approach.

7.3.2 Comparison of Alternatives to SLA-Driven Service Management

To compare the average reaction time $\bar{rt}$ defined in Equation 7.4, we use a qualitative architecture analysis approach to estimate the relative reaction time per alternative, in comparison to an unmanaged system's response. An unmanaged system is considered as one where there is no active system in place to avoid, monitor or respond to changes in system's workload.

The results for the over-provisioned, under-provisioned and dynamic approaches were 25%, 50% and 83.3% of the maximum planned workload respectively, where the dynamic approach refers to the one described in this thesis. These results indicate that the over-provisioned alternative has the best potential for performing better than an unmanaged system with regards to the set of adaptation triggers, and the under-provisioned approach performing worst. Our dynamic approach sits in the middle. The reason that the over-provisioned approach performs best is the provision of redundancy and assumption that resource costs are irrelevant. With massive redundancy in place, peaks and in some cases failures are masked by readily available resources. However, this assumption about resource costs is not scalable

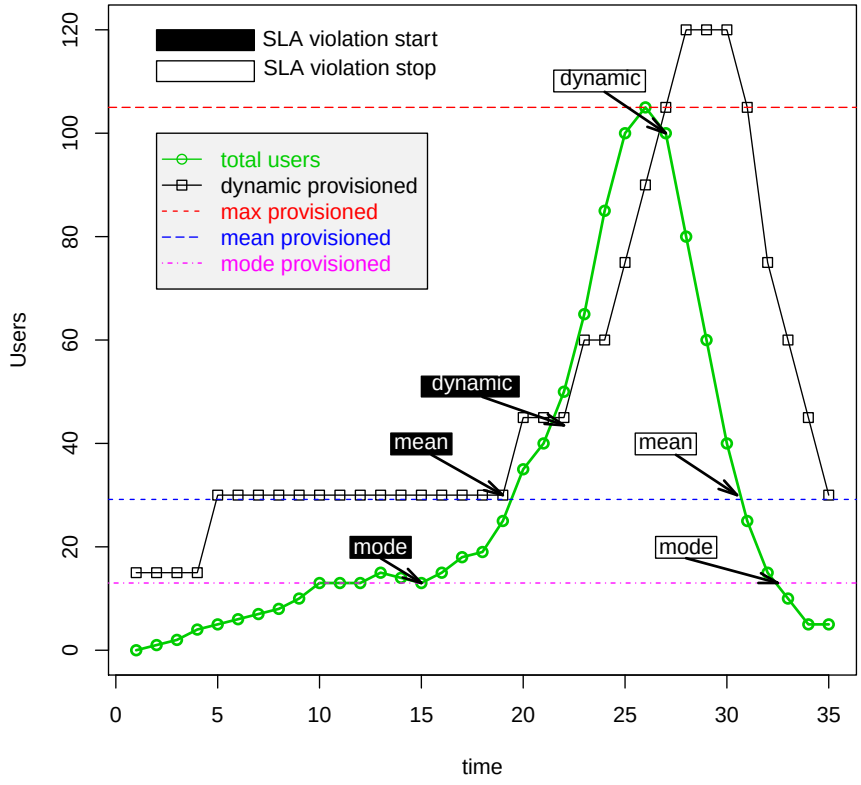


Figure 7.14: SLA Violations

or applicable to all circumstances, such that our dynamic approach has a better potential to satisfy multiple operational constraints.

In an analysis of SLA violations, using equation 7.5 and presented in Figure 7.14, the over-provisioning alternative will inevitably yield fewer SLA violations (tending to 0), as its contingency is to over-compensate with redundancy. The figure shows different values for the provisioned capacity of the system in terms of how many users it is able to handle without exceeding the maximum SLA-defined limits for the execution time.

The duration of SLA-violations in this case was the *Mode-Provisioned* strategy, as the mode tended to be low for this particular distribution of users. The duration of SLA-violation state for our dynamic approach is temporally similar to the *mean-provisioned* strategy (where the system is provisioned to handle the previously measured average or mean number of users in the SLA-defined limits of the performance) but the difference in user demand and service capability is significantly smaller throughout the duration, making the SLA violation impact less significant.

Although the *over-provisioned* (or *max-provisioned*) strategy appears to make sense from the perspective of minimising (zero-setting) reaction time and SLA violations, there is a penalty to pay for efficiency, as shown in Figure 7.15.

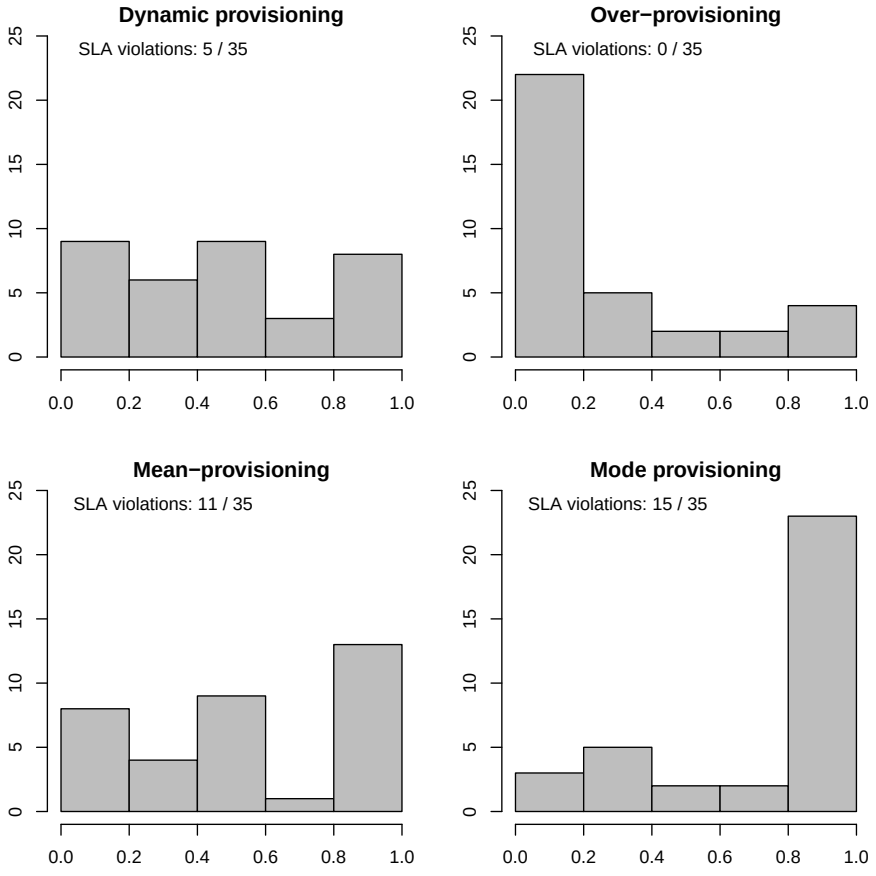


Figure 7.15: Provisioning Efficiency

The dynamic approach enables almost 20% of the operation time to be at near maximum efficiency, while the *over-provisioned* approach incurs above 20% at minimum efficiency for a fluctuating workload. The *over-provisioned* case can only be efficient if there is a constant burst of maximum anticipated users. Losses in efficiency with the dynamic approach are again transient, and can be addressed by optimising the reaction time to adaptation triggers.

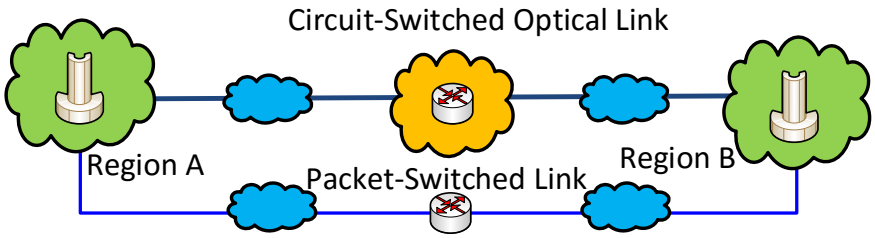


Figure 7.16: Testbed Connections

7.4 Experimental Testbed used for Running VM Scaling Experiments

The experiments performed for validating this research work were staged on two local and one distributed testbeds. The local testbeds were set up at SAP Switzerland and at University of Amsterdam (UvA), while the distributed testbed included the two mentioned local testbed connected by a circuit-switched optical network. We describe below the hardware used in each testbed.

The physical environment deployed at *SAP* consisted of three servers with Intel(R) Core(TM)2 Duo CPU E8400 CPUs, 4GB of memory and 1 Gbps network interface. CS and LB VMs were co-located on the same server, while WK and ST were each located on a separate server. A fourth server was used for running validation experiments, by using two additional VMs, one VM for the WK service and one VM for the ST service.

The *UvA* test-bed provided a single cloud-machine (Dell R810: 48cores, 128GB-RAM, 600GB storage) for on-demand computation and storage resources, which was used for running experiments with a large number of VMs.

The testbed was connected by a circuit-switched optical network (light paths) and a packet-switched network, as shown in Fig. 7.16. The optical network links had a bandwidth of 1Gbps and used Gigabit Ethernet over SDH as underlying technology. The exchange of signalling and control messages were transmitted over a permanent network connection using a packet-switched link, which was protected from communication interruptions.

The VMs created in the experiments were connected to the network environment using virtual Ethernet bridges created on each physical server using linux *brctl* [35] utility program. Each VM was connected to two virtual bridges, corresponding to the two primary networks: the circuit-switched optical link and the packet-switched link.

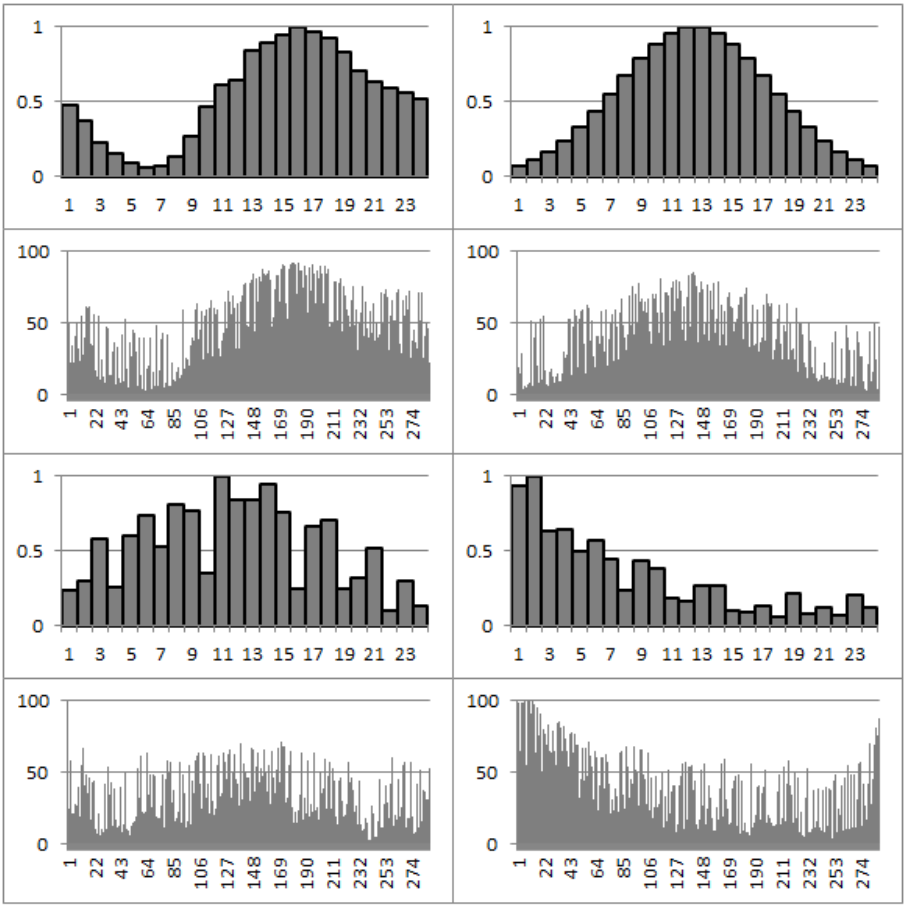


Figure 7.17: Distribution of resource utilization: above, trend per hour, below, trace per 5 minutes time slot

7.5 Evaluation of the Algorithm for Multi-Objective Cloud-Resources Allocation

We conducted a number of simulations for validating the system's characteristics with regards to the multi-criteria optimization of VM allocation described in Chapter 6. We did this by comparing the efficiency of the **G**enetic **A**lgorithm -based (GA) VM allocator with that of various known heuristics for the allocation of VMs. For this purpose we used synthetic generated VM request traces with seasonal distributions (matching Fig. 7.17) in order to load the system. For each VM we generated CPU and network traces with which we fed the monitoring and forecasting modules. We also varied the amount of noise added to the traces between 30% and 90%, in order to test the system stability.

In order to test the multi-objective evaluator, we selected for simulation the scenario in which the provider wishes to maximize his profits. Therefore

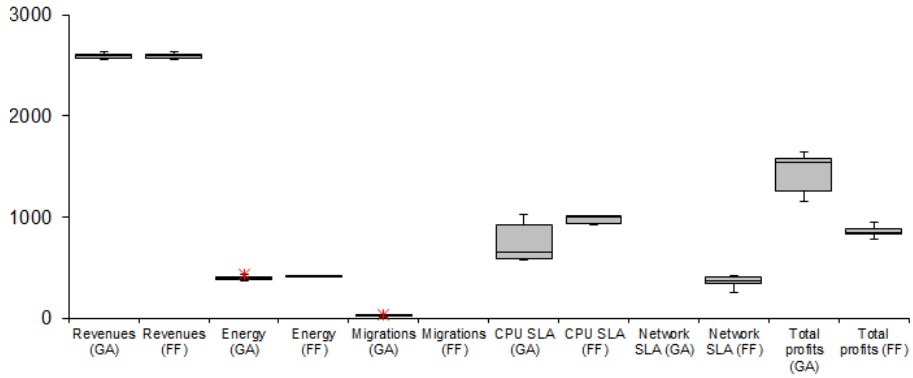


Figure 7.18: Cost and profit distribution for GA (genetic algorithm) and FF (first fit) allocators

we assigned the value of 1 to all four weights described in section 6.1.1. The weights describe the importance of each optimisation criterion (profit maximisation, energy efficiency, minimisation of SLA penalties) in the evaluation of each allocation of VMs.

We simulated a month of VM requests, including adding new VMs and removing existing ones, and we compared the SLA violations and energy efficiency of the allocations produced by the group-oriented genetic algorithm with forecasting (described in Section 6.1.2) with the first-fit and energy aware algorithms (presented in Section 6.2). The actual calculations for determining the costs of SLA violations and energy consumption were performed using the generated monitoring data and not the forecasted data.

For further testing of the algorithm's stability, we varied the genetic algorithm's parameters considering three different population sizes corresponding to one, two and four utilization weeks, four values for crossover probability (0.3, 0.5, 0.8 and 1) and the same four values for mutation probability. The results were consistent with the ones described below.

Our simulated infrastructure was composed of 10 hosts with quad-core CPUs, 16GB RAM and Gigabit networking. We varied the number of VMs between 10 and 50. The algorithm performed in average 1600 live migrations per simulation month (equivalent with two live migrations per hour) and at most one VM was selected for migration at a time.

The results are displayed in Figure 7.18, where the profits and SLA penalties caused by the live migration and over-subscription of resources are graphed using box-plots. The results show a consistent 100% reduction of network-SLA penalties together with a 30% reduction in CPU-SLA penalties, at the expense of below 1% of the revenues used for live migration. The total profit generated by using the GA allocator were in average 50% higher than the ones generated by the FF and energy-aware allocators. Also, the GA allocator distributes better the load across the infrastructure, leading to

a more uniform host utilization, lowering of the total energy consumption and reducing host wear.

7.6 Evaluation of Algorithms for Dynamic Composition of SLA Scaling Rules

We have evaluated the DynSLAOp (**Dyn**amic **SLA**s **Op**timiser system), which we described in Section 5.1, by analysing the behaviour of the dEIS application when using a DynSLAOp generated SLA for its management. For this purpose, we have run several experiments involving multiple VMs deployed in a physical environment. A increasing generated workload was directed at the dEIS application and the performance indicators of the application services were recorded. This allowed us to describe the scaling behaviour of the dEIS and to also gather large datasets which we then used for analytical determination of relations between the monitoring metrics and the system's scale.

We have checked whether the scaling sequence calculated by the dynSLAOp system provides a good compromise between the number of allocated VMs and the measured application's performance and how sensitive it is to the level of difference between the training set (benchmarks generated) and the synthetic workload. The application was tested using a different type of user load, as shown in Figure 7.19.

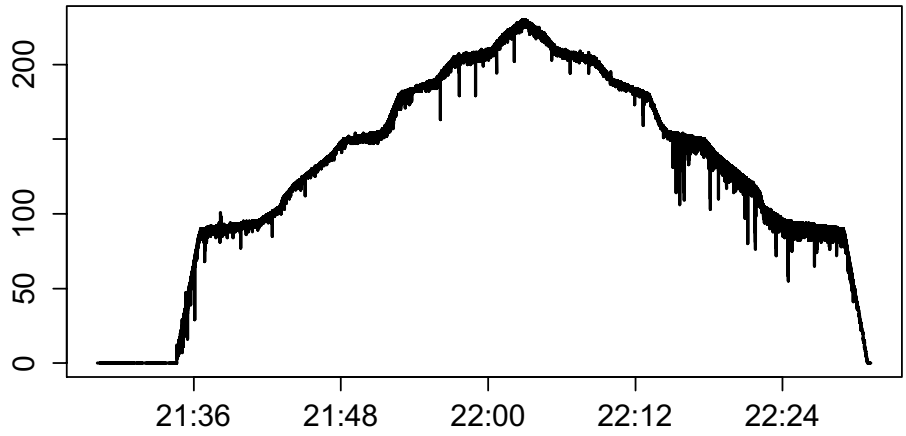


Figure 7.19: Concurrent Requests Distribution used for SLA validation

The number of concurrent requests at successive time moments was given, and the validation application client maintained a number of concurrent requests equal to the interpolated value between the two time moments. For example, if at time $t = 0$ there should be 10 concurrent requests and at time $t = 10$ there should be 20 concurrent requests, then at time $t = 3$ the validation client would maintain a number of 13 concurrent requests. The

validation client performs this check every 50ms.

While the training data has a clear increasing trend for the number of requests, the same is not true for the actual monitoring data obtained from the virtual machines. Here, the data is being sampled every x milliseconds, which can create difficulties in understanding the real state of the system, if only a individual data points are used, as the system's behaviour can only be observed during periods of at least several seconds. Additionally, the load is not perfectly balanced across the worker services, leading to the addition of 'noise' within the observed monitoring time series. In order to level-out the differences between the same service running in different VMs, monitoring data of all probes is aggregated (e.g. averaged) over predefined time intervals (e.g. one minute). This helps taking the decision whether the distributed application performs within the specified SLA contracts.

The following pipeline of experiments was executed.

- We defined the application as (1) the set of semantic service descriptions, (2) critical SLA metrics and (3) service dependencies and initialization parameters.
- We set the maximum service *scale-out* factor.
- We set the SLA range for application response time t_r, as an example of a monitored metric for Worker service.

DynSLAOp outputs the following information: (1) correlogram matrix, (2) the predicted sequence of landscapes, (3) linear regression coefficients, and (4) the final set of regression coefficients associated with this scaling sequence. Finally, the set of regression coefficients are transformed into SLA scaling rules and the generated SLA is tested as described below.

The DynSLAOp Benchmark Manager fires up the workload generator after customizing it for the agreed application. The VM containing the benchmark service is used to generate the workload according to four parameters: (1) maximum response time, (2) initial number of requests, (3) concurrent requests increment step, and (4) number of repetitions for each batch of requests. The request generator will perform the following steps:

1. send the initial number of requests in parallel
2. wait for all requests to complete their execution
3. record the requests' execution time
4. calculate the average execution time for all the requests in the batch
5. if the average execution time is lower than the maximum specified benchmark response time, then the generator will increase the number of requests and will repeat the procedure from step (1).

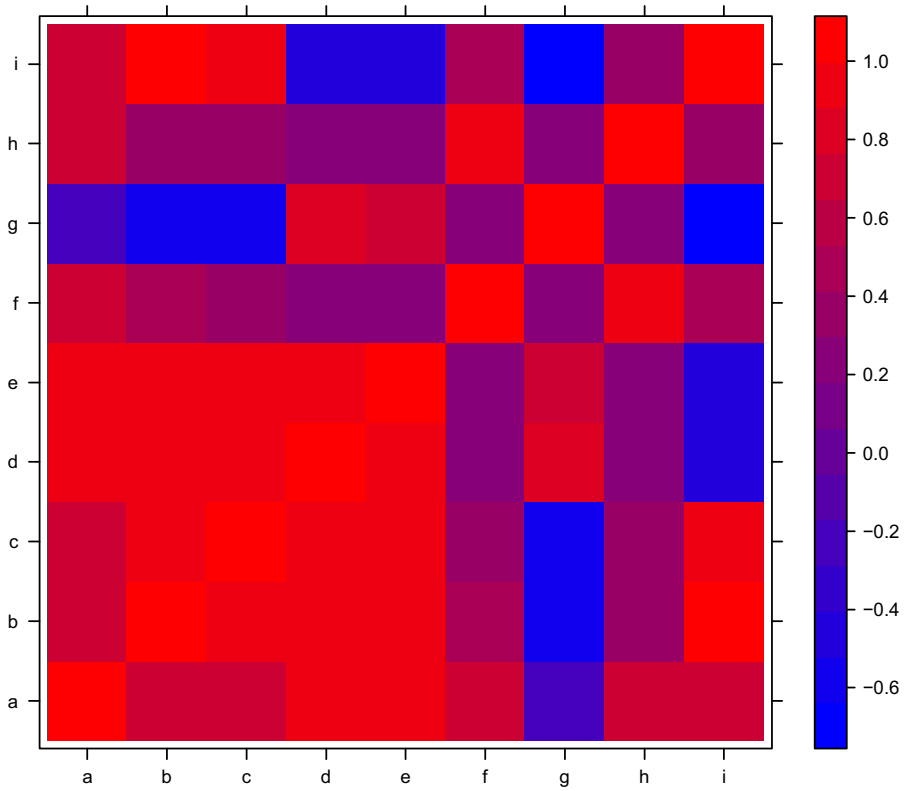


Figure 7.20: dEIS Correlogram

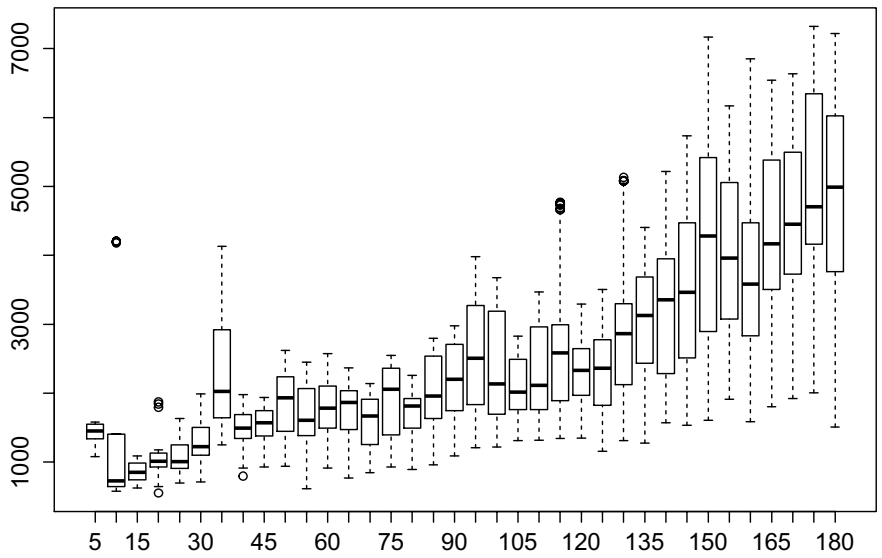


Figure 7.21: EIS Benchmark response for a landscape configuration with 4 Worker VMs and 6 Storage VMs

6. each batch of requests is repeated for the specified number of times.

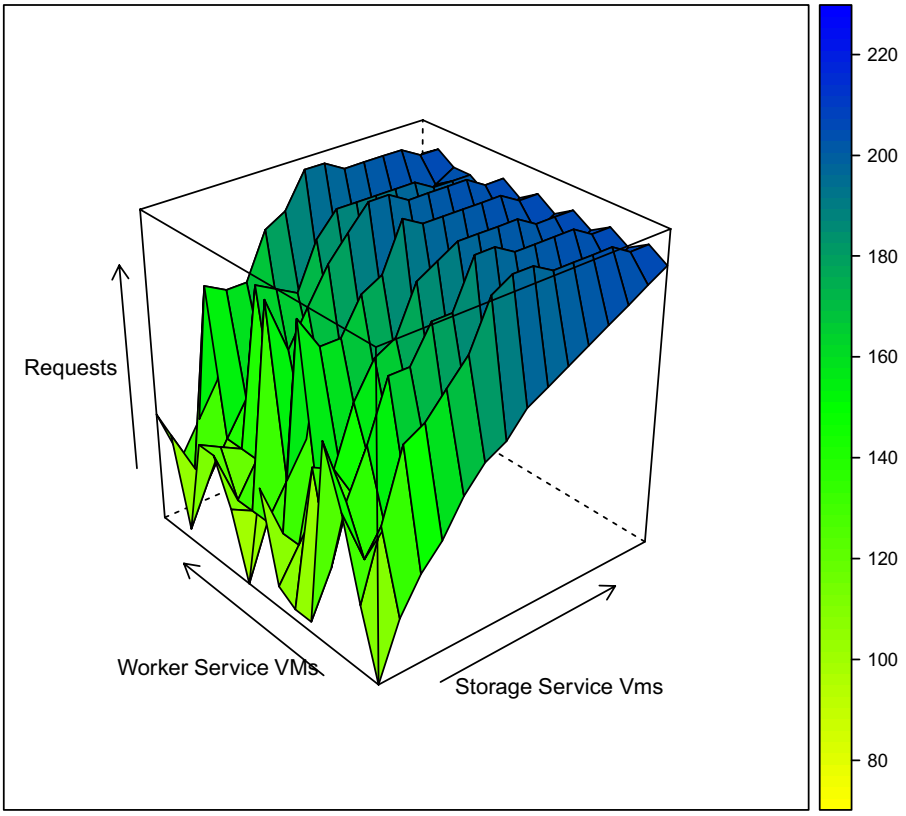


Figure 7.22: System's capacity of handling end-to-end requests vs. number of Worker and Storage VMs

The correlogram depicting the correlation between all the pairs of aggregated time series corresponding to the dEIS monitoring metrics is presented in Figure 7.20, where each metric is represented by a letter. From this graph it can be observed that metrics belonging to the Load Balancer service (a-e, e.g. the opening of a new user session at the LB service and the selection of the next suitable WK instance) are highly correlated, together with the aggregated average execution time of the Worker (g) and Storage (l), and the average number of Worker requests.

The before mentioned parameters form the actual metrics, which have been used for estimating the linear models for Worker and Storage scaling.

For each application topology configuration, the maximum concurrent load, for which the average worker execution time is below 5000ms, is determined. Figure 7.21 displays the dependency between the concurrent dEIS consumer requests and the response time measured at the worker, for a virtual landscape composed of 4 Worker VMs and 6 Storage VMs. In the Figure 7.22 it can be seen that as the number of concurrent requests executed by

the dEIS application is increased from 5 to 180 concurrent requests, the average execution time also increases linearly from approximately 500ms to 5000ms. The variance of the measured execution time also increases as the number of concurrent requests increases, due to the fact that the size of the data processed by the service for each request is not constant.

The dependency between the maximum number of concurrent user requests and the number of worker and storage service instances is displayed in Figure 7.22. From the graph can be observed that the request handling capacity of the distributed application stops increasing after a certain point. From the application source code profiling it became clear that the bottleneck was caused by the database connection management at the storage service. The authors think that replacing the current database (JDBC [9]) connection release mechanism with a pooled connection management would increase the application parallelism, as this would significantly lower the time required for opening a database connection. However, at the time we performed the experiments, there was no such straightforward implementation for MySQL [121] using OSGi [6].

Figures 7.23 and 7.24 show the values for the number of VMs as calculated using regression models for Worker, respectively Storage services. Each plot presents the number of VMs for Worker, respectively for Storage services (black line), the number of VMs calculated by using the regression model (blue line) and the filtered number of VMs resulting by applying a moving average smoothing on the regression-fitted values, combined with truncating the resulting value. In each graph, the blue line represents the model fitted values, while the black line represents the actual number of service instances determined from the scaling path.

For the application under test, the maximum request handling capacity was reached for a virtual landscape composed of 4 Worker VMs and 8 Storage VMs. After this point there was no significant increase in application processing capacity, given the requirement of executing the concurrent requests in less than 5000ms.

As previously described in Section 5.1.2, the resulting set of regression model coefficients were transformed into an SLA scaling rule, expressed as two IF statements using MVEL [34] expression language.

Figure 7.25 shows the actual application response while the SML was managing the EIS using the DynSLAOp generated SLA scaling rules. The graph of the number of concurrent requests is displayed in Figure 7.19. The actual VMs scaling for Worker and Storage services is presented in Figure 7.26, showing both the scaling-out and scaling-in.

Due to the fact that the service scale-out was triggered when the number of EIS concurrent user requests reached the maximum handling capacity of the application (average request execution time exceeded the maximum execution time), combined with the fact that VM instantiation is not an immediate action (average VM instantiation time was one minute plus around

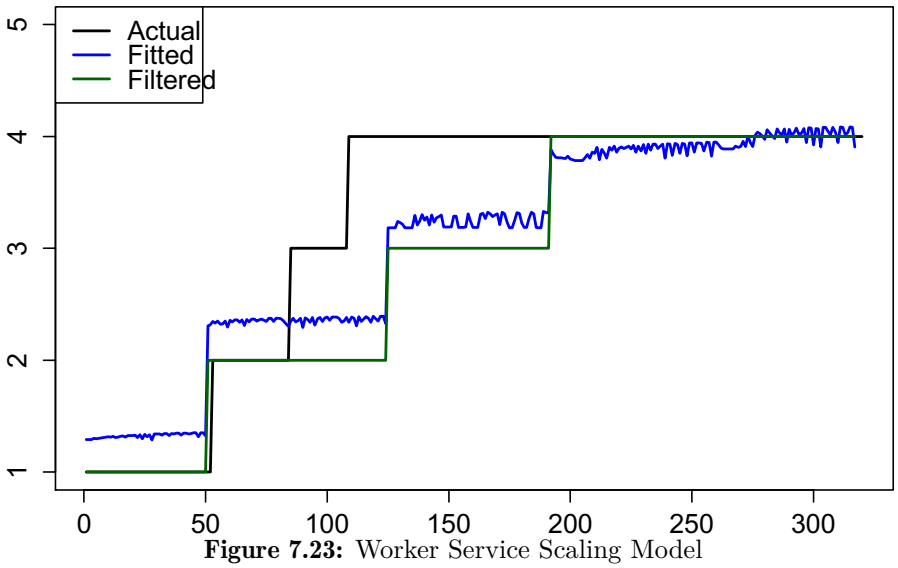


Figure 7.23: Worker Service Scaling Model

one minute for the OSGi bundles to start and to register to the distributed OSGi registry), the actual average execution time exceeded the SLA defined limit of 5 seconds (plotted as the red line in Figure 7.25), because of the delay between the VM instantiation time and the moment when the service becomes operational. This situation was repeated several times during the experiment, but only during the ramp-up phase. Towards the end of the experiment there was no SLA violation as the application had enough free capacity.

Overall, the experiment was a success, with 84% of the time enforcing the SLA, and a median execution time of 4284ms.

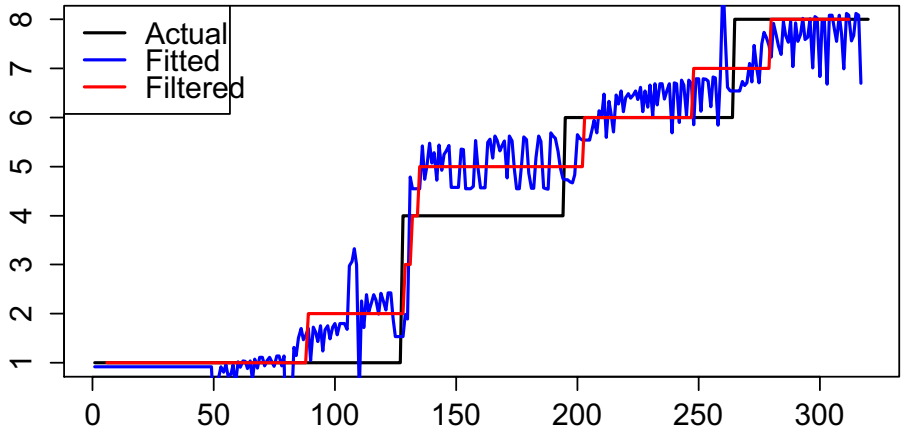


Figure 7.24: Storage Service Scaling Model

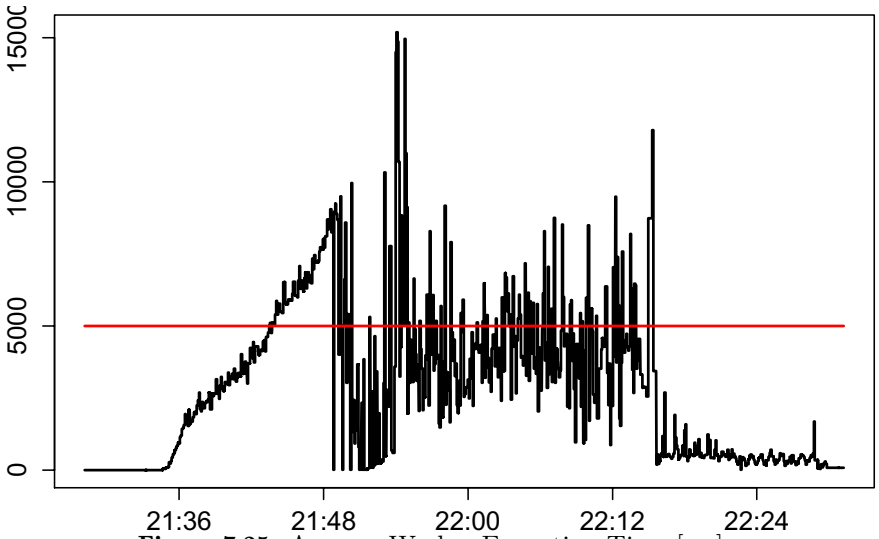


Figure 7.25: Average Worker Execution Time [ms]

7.7 Evaluation of Algorithms for Analytic Optimisation of SLAs

In order to validate the algorithms presented in Chapter 5.2 related to the analytic optimization of SLAs, we ran a number of four experiments using VMs deployed on the SAP tested, as follows. The first experiment was used for showing how correlations can help detect performance bottlenecks and incorrect service scaling policies. The second experiment validated the conclusion of first experiment related to scaling policy. The third experiment (1) validates the improved scaling policy and (2) confirms the detected correlation. The fourth experiment validates the use of prediction in combination with auto-correlation.

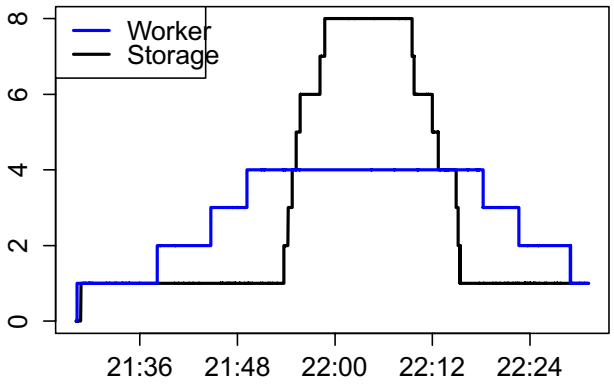


Figure 7.26: Actual Service Scaling

Experiment 1: Performance Optimisation using Correlations Detection

We initially deployed the EIS application in a single domain, with one service VM per application service (four VMs in total). The application was accompanied by a SLA containing a guaranteed state and two guaranteed actions for increasing and decreasing the number of Worker and Storage services. The SLA guaranteed state used in the experiments referred to the maximum number of concurrent user sessions, measured at the Load Balancer service with regards to the statically-defined processing capacity of the Worker service. This is further detailed below.

The specified minimum number (as defined by SLA) of application services were then instantiated in their corresponding VMs. The VMs starting, booting, active and shut-down times are displayed in Fig. 7.27. The report load generator was then started, triggering sending of multiple requests for generating reports using the data stored in the database. The number of concurrent Consumer requests has been controlled to remain in a specified region, as displayed in Figure 7.28. As can be seen in the figure, the consumer response time (1) varies with the increase in the number of concurrent requests and (2) follows the shape of both Worker and Storage services, suggesting that these SLA metrics might be influencing factors of the application performance.

Using the monitoring information received from the service probes, the system scaled the EIS application in order to maintain the SLA guaranteed state. The actual response times and system scaling responses can be observed in Figures 7.27 and 7.28.

In this first experiment, the consumer requested generating a PDF report from a variable portion of the TPCH database entries. The SLA scaling rule contained an expression for scaling the worker service when the average number of concurrent consumer requests (during a determined monitoring window) per number of active worker services exceeded 5, as shown in Equation (7.7), where LBs_i is the number of concurrent sessions at load balancer service at time i, W is the number of samples in time window of the Semantic SLA Processor, t is the sample number, $\|Worker\|$ is the number

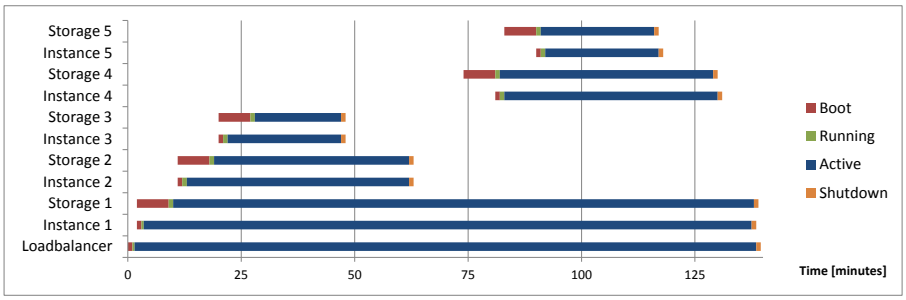


Figure 7.27: Experiment 1 - Service/VM Execution Times

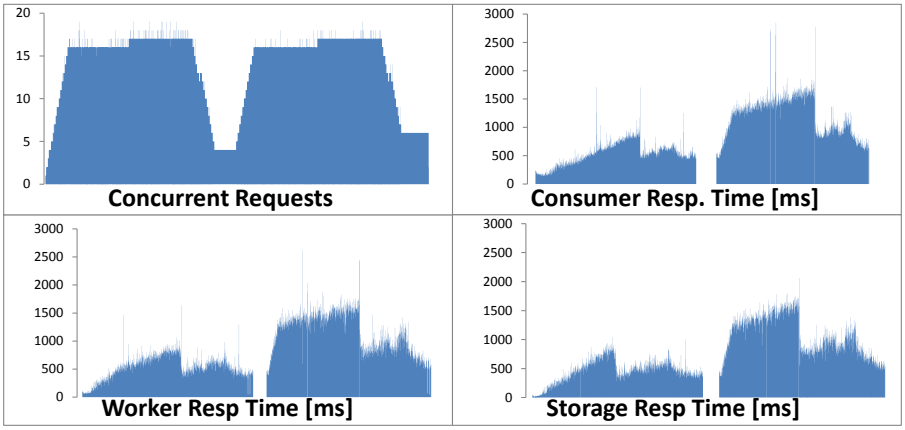
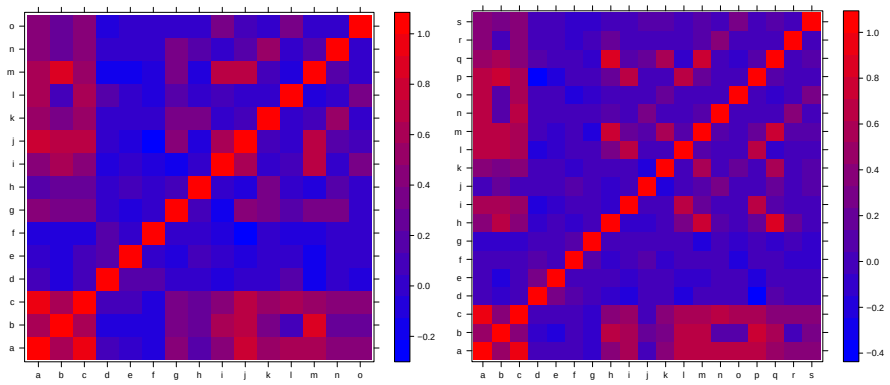


Figure 7.28: Experiment 1 - Consumer Execution Plan and Response Times



(a) **(b)**

Figure 7.29: Correlogram for the Enterprise Information System Monitoring Metrics

or active Worker/Instance services and c is a constant defining the worker service capacity, measured as the number of concurrent requests.

$$\frac{\sum_{t-W}^{t} LBs_i}{W} > \|Worker\| \cdot c \tag{7.7}$$

Since the only defined SLA guaranteed actions was the scaling of the worker/instance services, this led to a bottleneck at the Storage service. This behavior was confirmed by the correlogram of the performance metrics belonging to the EIS services, as it can be observed in Figure 7.29 and Table 7.3, where the second highest correlation coefficient (0.826) is between the Consumer and Storage services.

Table 7.3: The Metrics with the Highest Correlation

Metric A	Metric B	correlation
a (Consumer Sessions)	c (LB Sessions)	0.965
b (Consumer Rep. Time)	m (Storage 1 Rep. Time)	0.826
a (Consumer Sessions)	j (Storage 1 Req. Cnt.)	0.738
c (LB Sessions)	j (Storage 1 Req. Cnt.)	0.715

Experiment 2: Validation of Performance Correlations

For validating the assumption of high correlation between the Consumer and Storage services, we conducted a second experiment with only one Worker service and one Storage service, by disabling the application scaling.

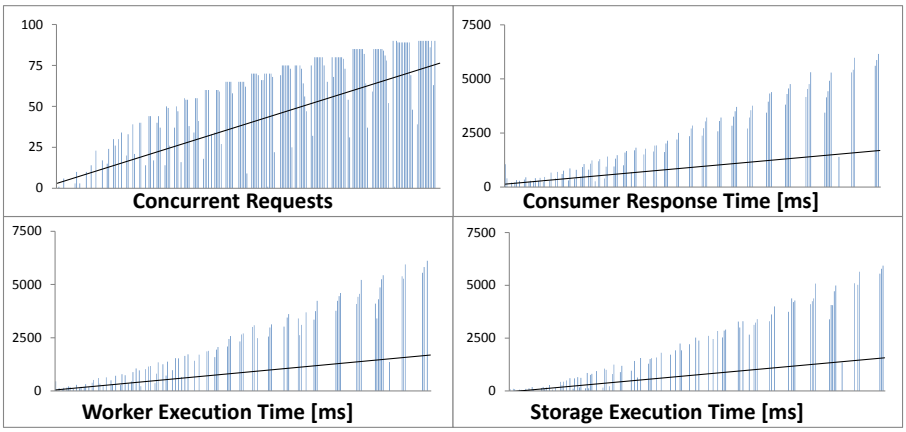


Figure 7.30: Experiment 2 - Service Execution Times

The response times displayed in Figure 7.30 confirmed that the response time increase is caused by the under-performing database service, as the response time seen at the Consumer is correlated with the query execution time at the Storage service. The same correlation was also observed in experiment 1, i.e. in Figure 7.29.

Experiment 3: Multi-Service Scaling SLA

We then prepared another experiment, in which the SLA-actions considered scaling both the report processing service (Worker) and the database service (Storage). Also, looking at the high correlation (Figure 7.29, squares corresponding to values above 0.85) between the number of concurrent sessions at the load balancer service and the response times measured at the consumer service, we decided to improve the SLA scale-action trigger to also consider the consumer average response time. In this way, the scaling would be performed only if the response time measured at the consumer would approach the limit defined in the SLA guaranteed state.

The system scaling response for this experiment is presented in Figure 7.31. Here, the running number of Worker services is the same as the number of running Storage services. The response time measured at the Consumer service decreases each time a new Storage service is started, confirming the correlation between (1) the Consumer requests' execution time, (2) the number of Storage services, and (3) the query execution time measured at the Storage service.

Experiment 4: Using Predictions and Auto-Correlation for SLA Optimisation

We also tested the ability of the system to predict the future system load. For this we enabled the prediction mechanism for the Load Balancer's active-session-count SLA metric. Figure 7.32b presents the received (noisy) monitoring data colored as black (roughly two periods) and the predicted values for the Load Balancer's active-session-count metric colored as blue, confirm-

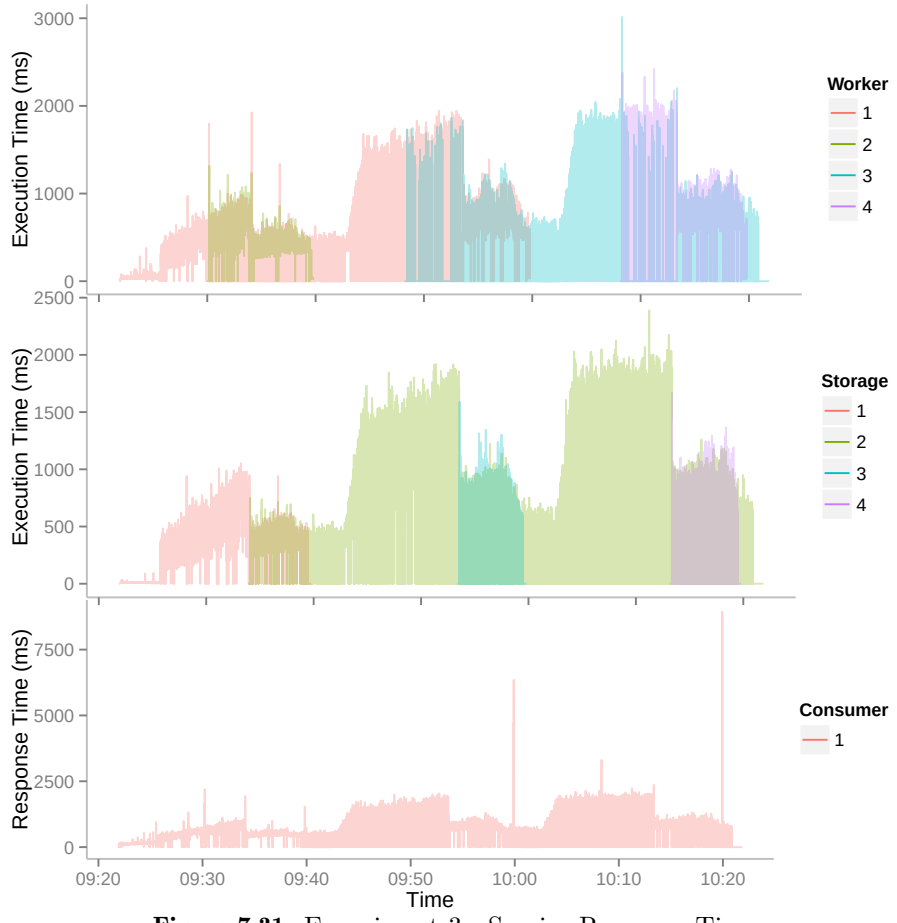
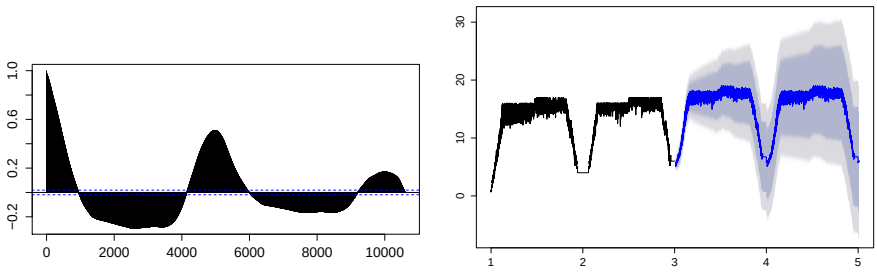


Figure 7.31: Experiment 3 - Service Response Times

ing that for certain types of application loads the prediction of monitoring metrics becomes a useful mechanism for improving the overall application performance and resource allocation. As shown in the paragraphs below, the predicted workload can be used for triggering in advance the instantiation of VMs, therefore minimising the impact of the instantiation delay on the application's processing capacity.

The data series for application load used in experiment 1 (see Figure 7.28) had a high predictability (confirmed by a high autocorrelation at a lag approaching the series period), enabling a good scheduling of service instantiation. As seen in Fig. 7.32a, the autocorrelation has a local maximum around lag (time series delay) value of 4900, indicating that this is the signal's period. The period is also confirmed by analyzing the plot from Fig. 7.32b, where two periods of the signal are plotted, along with the prediction and its confidence intervals for the next two periods.



(a) **(b)**

Figure 7.32: Experiment 4 - Consumer Concurrent Sessions Count. (a) Autocorrelation. (b) Prediction.

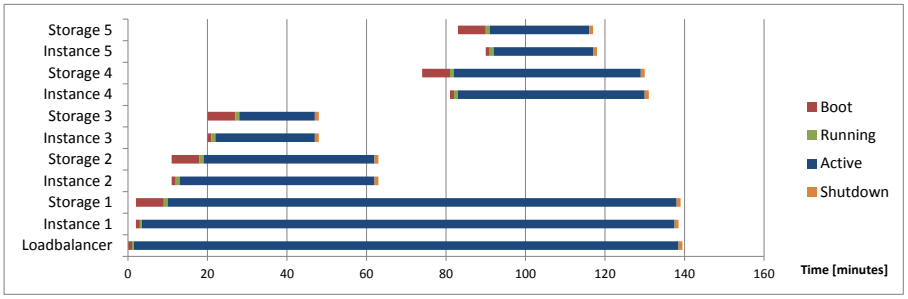


Figure 7.33: Experiment 4 - Service Start Times with Prediction

Using the predicted time when the load will increase (as calculated by the second algorithm in Section 5.2), it became possible to schedule the instantiation of the Storage service such that it was ready for processing at the time the load actually began increasing. In Fig. 7.33, the Storage services are instantiated earlier than the Worker services, so that both Storage and Worker services become active at the same time. The reason for the longer

instantiation time of the Storage service is that the VM of the Storage service is eight times larger in size (8GB) than the VM of the Worker service and so, it takes a longer time to be copied from the VM image repository.

7.8 Simulations of VM-Scaling Algorithms

Having performed multiple experiments on the physical local testbeds, we used the gathered monitoring information about the dynamic behaviour of the dEIS application for building a flexible simulation model, which we then used for validating the three VM-scaling algorithms described in Section 4.4 using several dynamic workloads.

We describe the validation of the simulation model of the dEIS application in Section 7.8.1. The next two sections 7.8.2 and 7.8.3 deal with the large-scale simulations of the SLA-based VM-scaling algorithms.

7.8.1 Validation of the dEIS Simulation Model

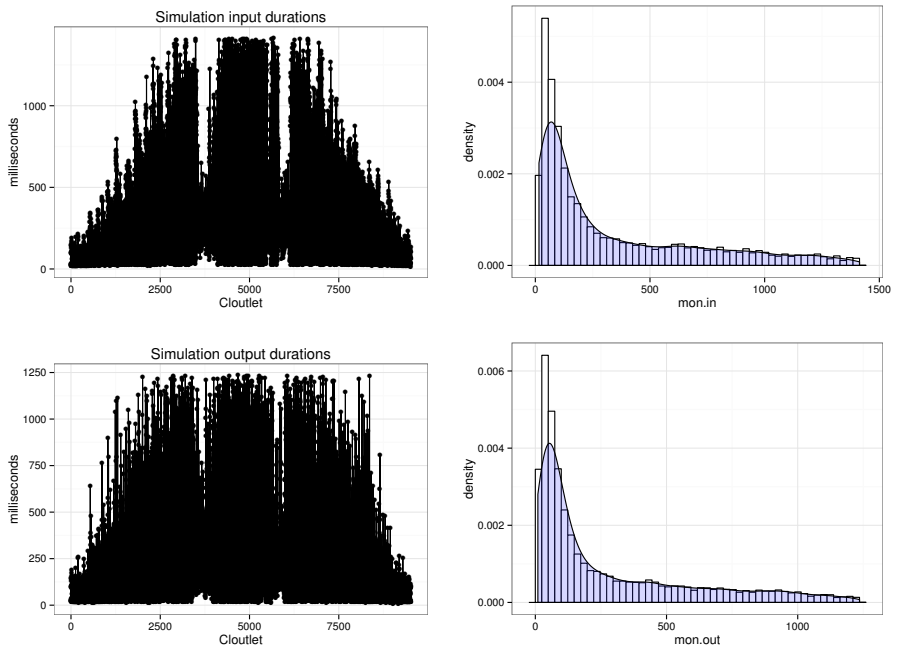
We aim to evaluate the CloudSim simulation's ability to model and reproduce the properties of the targeted distributed application. For this purpose, we used Kolmogorov-Smirnov (KS) [44] statistical test and linear dependency regression coefficient [130] for comparing the simulation and experimental results. We also evaluated the model's ability to simulate environments with different numbers of VMs and different concurrent workload. The simulations used the same concurrent workload values as the real-world scenario presented in Section 7.2.

The two-sample KS test compares cumulative distribution functions (CDF), and the test statistic is the maximum difference in CDF value, as defined by Equation 7.8,

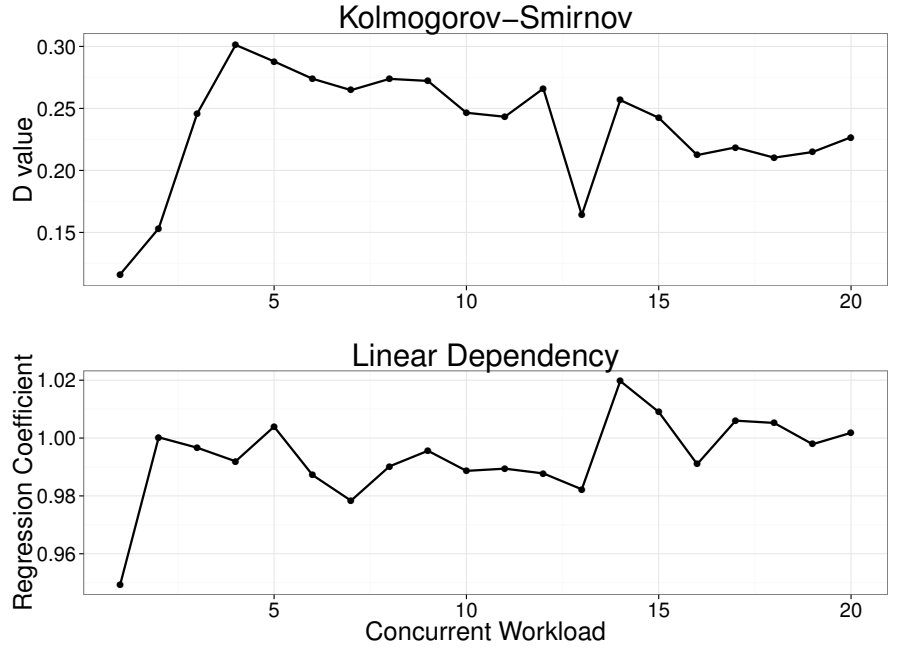
$$D_{n,n'} = \sup_x |F_{1,n}(x) - F_{2,n'}(x)| \tag{7.8}$$

where $F_{1,n}$ and $F_{2,n'}$ are the empirical distribution functions of the first and the second sample respectively.

In Fig. 7.34a we show the comparison between the simulator input values ($Profile$ performance traces) and the output produced by CloudSim, showing very similar density distributions of the execution times. The difference between the distribution of values is low, as indicated by a D value of 0.0855 of the KS test. In the top left graph we show the plot of values that were measured from an actual dEIS system running in a physical environment. The bottom left graph shows the CloudSim simulated durations using the models described in Section 7.2. As it can be seen, the two graphs are very similar, indicating that the simulation models is representative for the reproduced application behaviour. This fact is further confirmed by the



(a)



(b)

Figure 7.34: (a) Simulation input vs. output comparison. (b) EIS End-To-End Response Time (ms) - Experiment vs. Simulation comparison at constant load

graphs on the right, showing the distribution of values, which have the same shape (log-normal with a long tail) and percentiles.

We also validated the ability of the simulation model to correctly reproduce end-to-end response times under constant concurrent workload. In Fig. 7.34b we show the KS test values and linear dependency between the values gathered in experiment and simulation. The graphs show a good correlation with a value of 0.3 of KS statistic (maximum difference between CDF functions), and a linear coefficient of 1.05 between experiment and simulation results (indicating that the simulation values differ with only 5% from the experiment values).

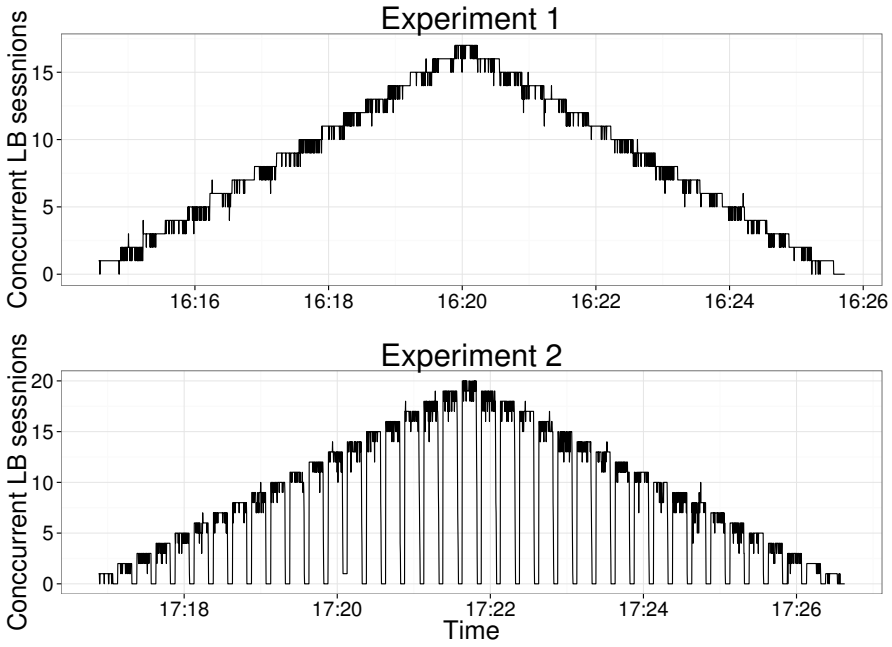


Figure 7.35: Experiments 1 and 2: Concurrent number of LB sessions

In order to test the simulation model under varying workload conditions and with a different number of VM instances for the dEIS services, we ran two additional experiments, as shown in Fig. 7.35. In the first experiment still used one VM per service. However, we varied the concurrent workload from 1 to 17-and back to 1 parallel request. In the second experiment we used 2 VMs for each of WK and ST services, using in total 4 servers, while the concurrent workload was varied from 1 to 20 parallel requests, with 5 seconds pause before switching to a new workload level. The experiments were performed with real resources and then the results were compared with the same scenario running in the simulator.

In case of experiment 1, simulation results shown in Fig. 7.36 validated the modeling of the dependency between service task durations and the corresponding number of executed CPU instructions, as the simulated end-

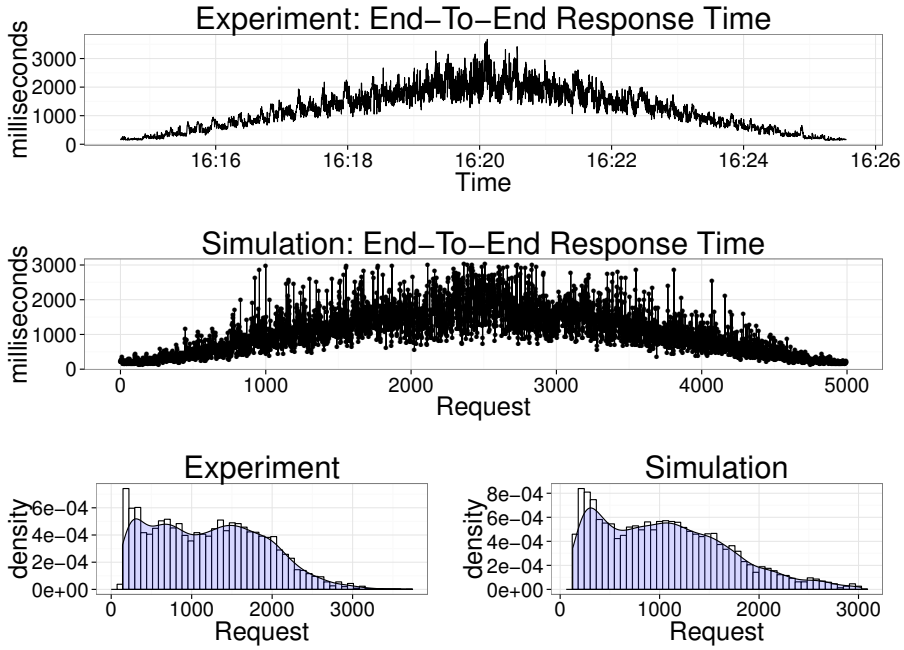


Figure 7.36: EIS End-To-End Response Time (ms) - Experiment 1 vs. simulation results comparison with 1 WK and 1 ST VMs

to-end response time closely followed the measured dEIS response times. The increased variance in simulation results was caused by the simulator's event communication overhead and by the fact that the simulator randomly chooses the requests to be simulated.

Experiment 2 validated the modeling of concurrent computing workloads distributed across multiple VMs. The simulation results closely matched the ones measured during the experiment experiment, as can be seen in Fig. 7.37.

We present in Table 7.4 descriptive statistics between the end-to-end response duration measured in the two experiments and the values produced by the simulation. In both experiments, the difference between the median and mean values is around 15%, showing a good ability of the simulation to estimate the real behaviour of the modeled distributed application.

7.8.2 Simulation of the Reactive SLA-Based VM-Scaling Algorithm

In order to evaluate the integration of the SLA Scaling Manager and Parallel Cloudlet Scheduler into CloudSim simulator we ran three different simulations testing both the handling of multiple cloud tenants and the ability to scale the number of VMs according to high-level SLAs.

For building a system able to compare the SLA-based VM-Scaling policies, we extended our dEIS CloudSim-based simulator [12, 14]. We added

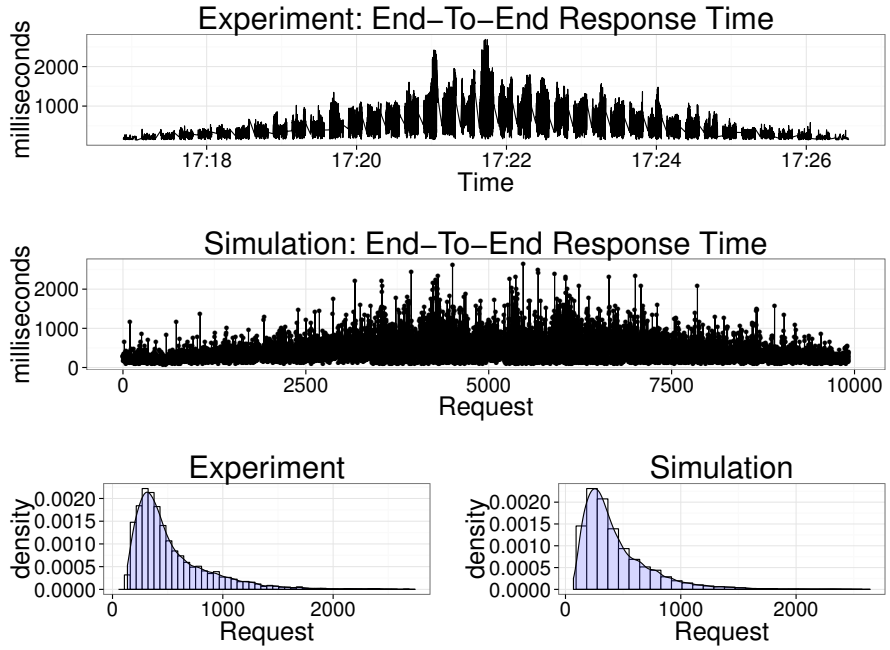


Figure 7.37: EIS End-To-End Response Time (ms) - Experiment 2 vs. simulation results comparison with 2 WK and 2 ST VMs

two new CloudSim scaling policies for the algorithms described in Sections 4.4.2 and 4.4.3, supported by two new additional monitoring metrics (arrival rate and throughput) at the CloudSim datacenter-broker level.

For integrating the multi-step prediction mechanism described in Section 4.3 we wrote a series of analytic scripts in R [129] (for calculating the linear regression model, prediction from the regression model, and prediction's accuracy), which were invoked by our CloudSim extended datacenter broker using the TCP/IP Rserve [145] library. Given that the Rserve library does not allow transferring of arbitrary complex R objects, we had to cache the regression's results in R, by allocating unique model identifiers to each pair of cloud tenant and service. In this way, CloudSim would transmit the

Table 7.4: Simulation vs Experiment End-to-End dEIS Response Time Statistics (in ms)

	Exp. 1	**Sim. 1**	**Exp. 2**	**Sim. 2**
Min.	320	141.8	69.01	135.0
1st Quantile	1730	576.0	235.40	293.0
Median	2952	2520.0	347.70	414.0
Mean	2964	3117.8	439.60	526.1
3rd Quantile	4106	4901.3	549.30	652.0
Max.	6939	12804.0	4597.00	2690.0

model identifier when a regression model was calculated, and then the same identifier would be used when CloudSim required calculating a prediction based on the regression model.

Simulation 1: Constant Workload

In this simulation we considered a single cloud tenant running a constant load of 20 concurrent transactions distributed across multiple services, with the dEIS system configured to use only one VM for each service. We used this simulation as a base for comparing the others simulations where we will introduce multi-tenancy and varying scaling conditions.

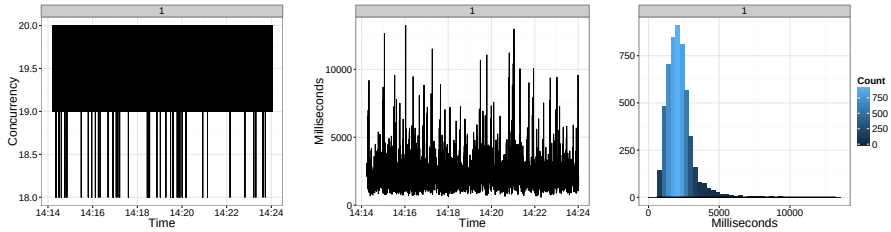


Figure 7.38: Simulation 1 (a) CS Concurrent Load (b) CS Response Time (c) Histogram of CS Response Times

In Fig. 7.38a we display the simulated load, which varied between 19 and 20 concurrent requests due to asynchronous sampling of the number of active dEIS requests. Fig. 7.38b shows the response time measured at the CS service for each distributed concurrent transaction, under the given workload. As the simulation model uses datasets from a real distributed application [13], it has a rather large variance. Fig. 7.38c displays the

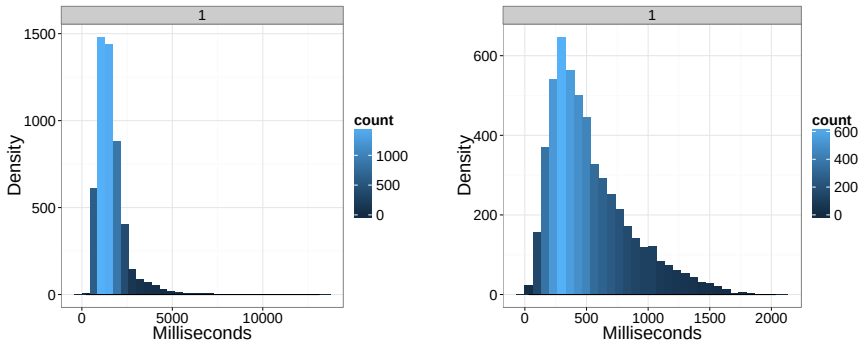
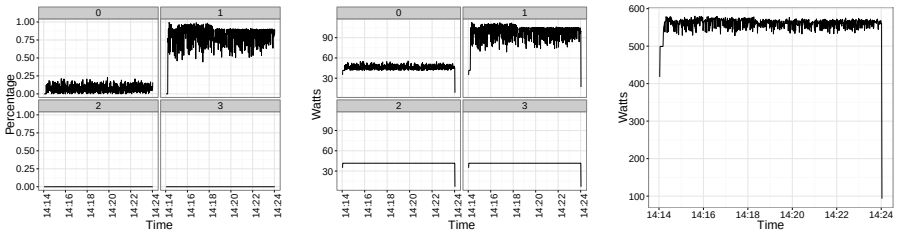


Figure 7.39: Simulation 1 (a) Distribution of WK execution times (b) Distribution of ST execution times

distribution of response times at CS service. The average CS response time was 2200ms at the considered workload level of 20 concurrent transactions per second.

In Fig. 7.39a we display the execution times at the WK service, respectively in Fig. 7.39b at the ST service. This shows the breakdown of total transaction execution time between WK and ST services, with an average WK execution time of 1652ms and a standard deviation of 1025ms, and an average of 533ms and standard deviation of 329ms for the ST service respectively.



(a) **(b)** **(c)**

Figure 7.40: Simulation 1 (a) Hosts CPU Utilization. (b) Hosts Power Consumption. (c) Total Datacenter Power Consumption

Fig. 7.40a displays the hosts' average CPU utilization as calculated from the VMs' CPU utilization, while Fig. 7.40b shows the energy consumption of the hosts, by considering a linear dependency model between the CPU utilization and host's power consumption. It is important to note the fact that hosts with no active utilization still consumed a large amount of energy.

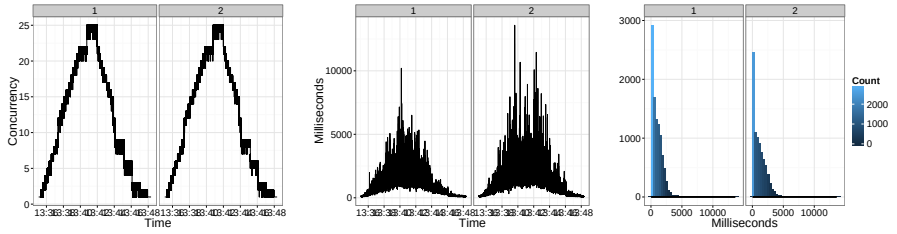
Fig. 7.40c shows the total power consumption at datacenter level, which was summed over every second, for the first simulation. The total simulated energy consumption had a value of 335.9 KJ, distributed evenly across the entire simulation duration, as a consequence of the constant workload.

This simulation re-confirms that the dEIS simulator is indeed accurate with regards to the models described in Section 7.2. Having confirmed the behaviour of the dEIS application under constant workload, we can not proceed to simulating more complex workloads with multiple cloud tenants (e.g. multiple applications). This also proves that the CloudSim-based dEIS simulator is able to simulate and validate the VM-scaling algorithms described in Section 4.4.

Simulation 2: Multi-Tenant dEIS

In the second simulation we considered two cloud tenants (client organizations), each executing a varying workload as shown in Fig. 7.41a, which increased from 1 to 20 concurrent transactions and then decreased back to 1. The first tenant (#0) executed its workload on a fixed virtual infrastruc-

ture (static number of VMs/ no scaling), while the second tenant (#1) had scaling enabled at 1000ms for WK service, and at 400ms for the ST service respectively.

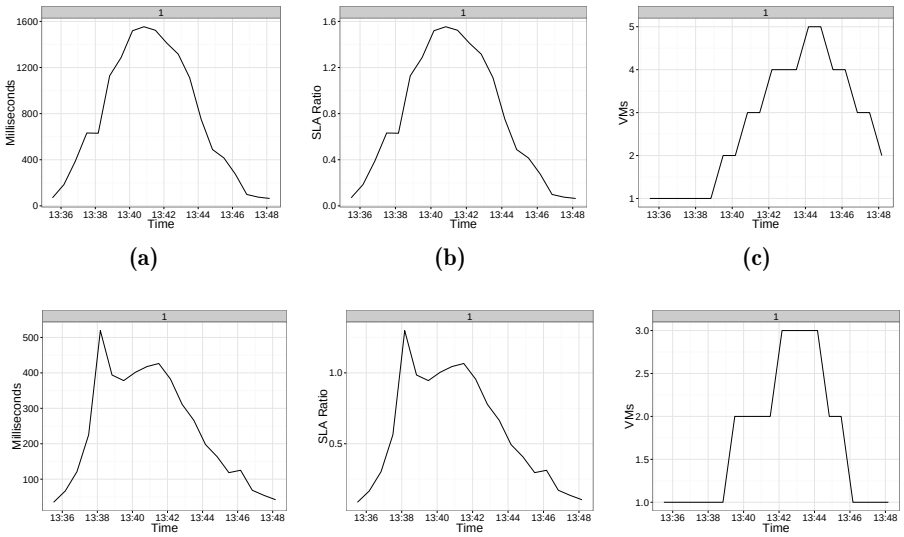


(a) **(b)** **(c)**

Figure 7.41: Simulation 2 - Per tenant, CS service (a) Concurrent Load (b) Execution Time (c) Histogram of Execution Times

As shown in Fig. 7.41b and Fig. 7.41c, tenant #1 (SLA scaling enabled) had a lower average transaction execution time of 965.8 ms, compared to tenant #0, who had an average execution time of 1144.1 ms. This shows the advantages of running the cloud workload under SLA conditions on a dynamically scaled infrastructure, compared to running it on a fixed-sized virtual infrastructure.

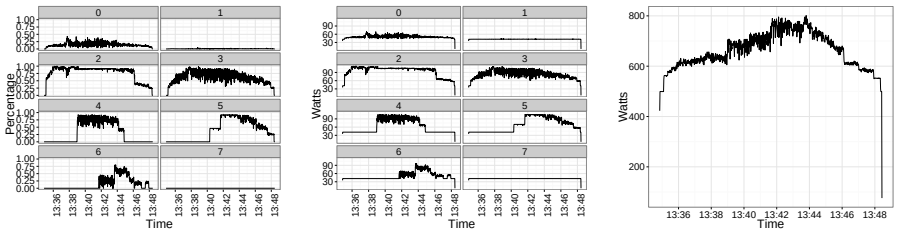
The scaling behavior for tenant #1 is described separately for WK and ST services. Fig. 7.42a shows the average execution time for the WK VMs



(a) **(b)** **(c)**

(d) **(e)** **(f)**

Figure 7.42: Simulation 2 - WK service (a) average execution (b) SLA Scaling Ratio (c) Number of VMs. ST service (d) average execution time (e) SLA Scaling Ratio (f) Number of VM



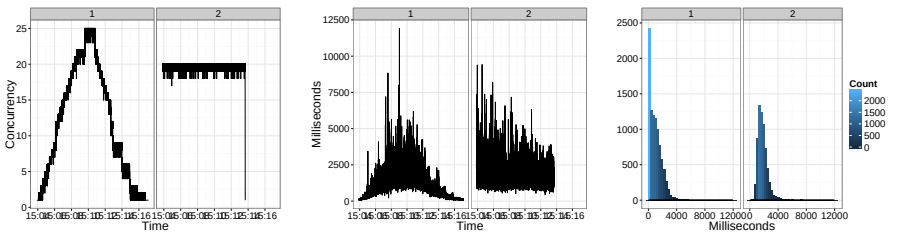
(a) **(b)** **(c)**

Figure 7.43: Simulation 2 (a) Hosts CPU Utilization. (b) Hosts Power Consumption. (c) Total Datacenter Power Consumption.

calculated over a moving time window of 40 seconds, correlated with the concurrent workload presented in Fig. 7.41a. Fig. 7.42b shows the SLA ratio between the average execution time and the SLA threshold of 1000ms. As the SLA ratio approached the SLA scaling threshold (0.9 for scale-out, respectively 0.6 for scale-in), the SSM Algorithm 4.2 varied accordingly the number of VMs, as shown in Fig. 7.42c.

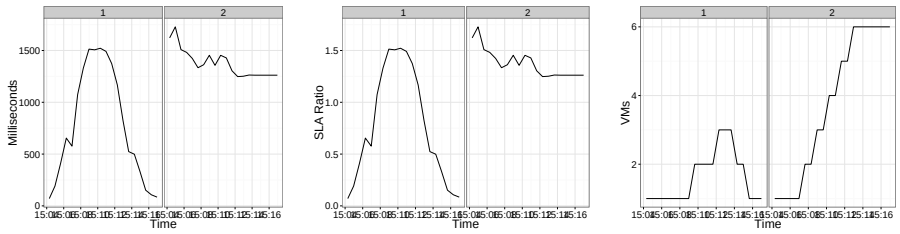
Similarly, the ST service was scaled based on the average execution time shown in Fig. 7.42d. The SLA scaling ratio for ST service is shown in Fig. 7.42e, while the actual number of ST VMs is displayed in Fig. 7.42f. The maximum number of ST VMs varied from 1 to 3 and then back to 1. It is important to note that the system did not oscillate as the SLA scaling ratio approached the scaling threshold, because of the scaling speed limitation mechanism described in Section 4.4.1.

The effect of scaling VMs on the average CPU utilization of hosts can be observed in Fig. 7.43a, while the energy consumption per host can be observed in Fig. 7.43b. The CPU utilisation for hosts 1 and 7 is constantly very low as they were not used for hosting VMs. However, their energy consumption is not negligible, as it can be seen in Figure b. Also, the CPU utilisation of the other hosts increases as more VMs are added to the system. The total datacenter's power consumption can be visualized in Fig. 7.43c



(a) **(b)** **(c)**

Figure 7.44: Simulation 3 - CS service (a) Concurrent Load (b) Response Time (c) Histogram of Response Times



(a) **(b)** **(c)**

Figure 7.45: Simulation 3 - (a) WK average execution (b) WK SLA Scaling Ratio (c) Number of VK VMs

and had a value of 538 KJ. As the VMs' utilization increases the effect on datacenter's power consumption is an increase with approx. 20%, as the idle hosts still contribute significantly to the total power consumption.

Simulation 3: Reactive SLA-Based VM Scaling with Multi-Tenancy

The third simulation consisted of two tenants, each with SLA scaling enabled. The SLA scaling thresholds were the same for both tenants, 1000ms for the WK services, respectively 400ms for ST services.

The workload executed by the first tenant was varying from 1 to 20 and back to 1 concurrent transactions, while the workload of the second tenant was constant at 20 concurrent transactions as shown in Fig. 7.44a. The execution time per request of each tenant at the CS service is displayed in Fig. 7.44b, and had an average value of 1017.4 ms for tenant 1, and 1802.2 ms for tenant 2 respectively. The measured average values are consistent with the ones obtained in simulations 1 and 2. The histogram of tenants' execution times measured at the CS service is displayed in Fig. 7.44c.

The WK's average execution time per tenant is displayed in Fig. 7.45a, while the SLA ratios are displayed in Fig. 7.45b, and the number of WK's VMs is shown in Fig. 7.45c. The simulated values are consistent with the ones produced in the previous two simulations.

This concludes the validation of the dEIS simulator as we showed that the simulator is capable of calculating complex interactions between multiple VMs and using shared hosting resources (hosts). In the next section we will use the dEIS simulator for validating the complex VM-scaling algorithms using a large-scale VM simulation.

7.8.3 Large-Scale SLA-Based Simulation of dEIS Horizontal VM-Scaling

In order to evaluate the two new SLA-based VM-Scaling algorithms previously presented in Sections 4.4.2 and 4.4.3 we implemented them in CloudSim [69], which allowed us to run multiple simulations against the dEIS distributed application.

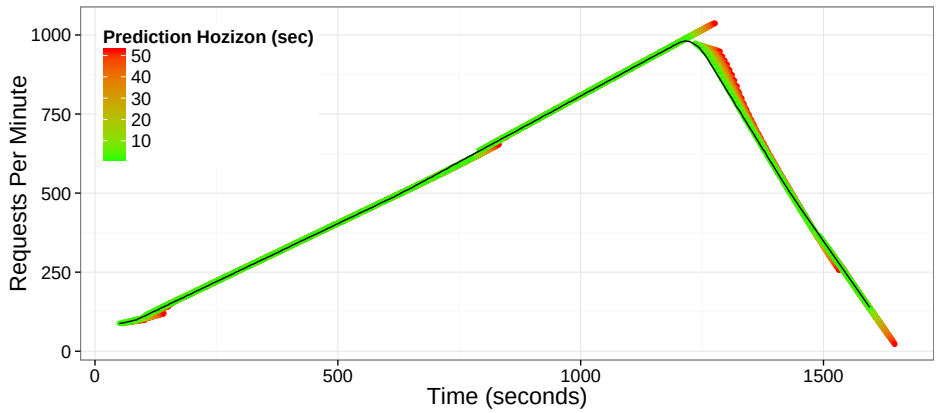


Figure 7.46: Actual Arrival Rate (black) vs. Predicted Arrival Rate

We compare the λ-based and predictive λ-based VM-Scaling algorithms using simulations based on a synthetic workload. Finally, we describe a real-world scenario where the incoming workload received by a system grew with four orders of magnitude, and then we use this workload-description to simulate and compare the three scaling algorithms described: reactive, λ-based and predictive λ-based.

Evaluation of the Predictive VM-Scaling Algorithm

In order to evaluate the predictive VM-scaling algorithm, we created a CloudSim simulation with 500 available hosts in which the workload was increased linearly from 60 requests per minute to 1000 requests per minute during 1200 seconds, and then it was linearly decreased back to 60 requests per minute during 400 seconds. We ran the simulation first with the λ VM-Scaling algorithm, and then with the predictive-λ VM-Scaling algorithm. The prediction algorithm used a forecasting window of 50 seconds. The simulated time for instantiating a VM was 18 seconds for the WK VM, respectively 23 seconds for the ST VM. The maximum error value (ϵ_{max}) for considering a prediction as valid was set to 1.5%. The VM-Scaling algorithm was executed every 5 seconds.

Figure 7.46 shows the distribution of workload across simulation's duration, measured at the WK service. The prediction horizon refers to the time distance from the time moment when the prediction is calculated to the time when the workload is forecasted. The black line represents the actual value of the workload calculated over a moving window of one minute. As it can be seen, the prediction closely follows the actual values of the workload. At simulated time 1200, when the workload's trend changes from increasing to decreasing, it can be observed that the prediction continues to increase, however, it quickly realigns itself with the new direction. This confirms that the prediction mechanism used was appropriate for predicting

this type of workload, and that incorrect predictions do not affect the algorithm's correctness as the predictions are only used for scale-out and not for scale-in.

Figure 7.47 shows the actual evolution of the number of VMs belonging to the WK service. In the simulation shown in Figure 7.47a we used the λ-based VM-Scaling algorithm, while in Figure 7.47b we used the predictive λ-based VM-Scaling algorithm. It can be seen that in Figure 7.47b the plot of the actual number of VMs is perfectly aligned with the calculated number of VMs (the two lines overlap), while in Figure 7.47a there is a noticeable gap between the calculated and the actual measured number of VMs at times 500 and 1000, caused by the delay of 18 seconds in instantiating the WK VM.

These two simulations have shown that it is advantageous to use prediction in conjunction with the λ-based VM-Scaling algorithm. The prediction-enabled VM-Scaling algorithm has the advantage of eliminating the effect of the delays when scaling-out the VMs, as VMs become operational at exactly the right moment compared to the increase in the workload responsible for triggering the scale-out.

Real-World Application Scenario

For testing the ability of the VM-scaling algorithms to dynamically increase and decrease the number of VMs allocated to distributed services, while also complying with the SLAs regarding the maximum value of the execution times, we have selected a simulation scenario based on a real-world event [43].

In order to cover a Schwingen (Swiss sport) event and for reporting real-time on the performance of sportsmen, as well as live-tracking the scores of the fights, a cloud computing infrastructure was prepared and a set of mobile applications were developed. The app combined real-time time pro-

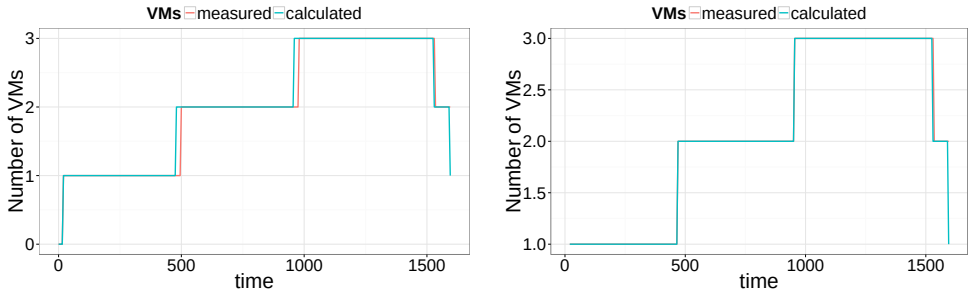


Figure 7.47: Evolution of the number of VMs of the WK service in case of applying a) λ-based scaling algorithm b) predictive λ-based scaling algorithm

cessing and analytics with a mobile platform, while running in the cloud environment.

However, due to the huge success of the event, the mobile application was downloaded 70000 times, creating a very large workload for the computing infrastructure. This, combined with sub-optimal scaling of the computing infrastructure (according to [43]), led to some very poor performance on the mobile side, with very long waiting times.

We attempted to simulate a similar load for the dEIS distributed application, by creating a workload of up to 28000 dEIS-users (each user corresponding to a CS-issued request), while at the same time keeping the combined response time for the WK and ST services below one second. Figure 7.48 shows the distribution of workload across time. The workload first increases to 500 concurrent users during 1800 seconds, then to 1000 users in 700 seconds, then it approximately doubles at every 600 seconds, until it reaches 11000 users at time moment 4800. From there on the workload increases with roughly 10000 users at every 1600 seconds, reaching 28000 users at time 6300. The workload will stay at this level for about 600 seconds, after which it begins decreasing to towards 25 users during approximately 1200 seconds. In total we simulate 8400 seconds, or 2 hours and 20 minutes.

The goal of the simulation scenario is to test the ability of the VM-scaling algorithms to maintain the execution time below the one specified in the SLA, and implicitly, to prevent the dEIS system from becoming overloaded with requests.

Comparison of VM-Scaling Algorithms

For comparing the presented VM-Scaling algorithms, we used the workload described in Section 7.8.3 together with a SLA policy defining a maximum combined response time of 1 second for the dEIS system, divided between WK and ST services with a ratio of 7:3. The resulting SLA specified a maximum execution time of 700ms for the WK service, respectively 300ms for the ST service.

The workload was simulated against the dEIS model [14] (constructed using recorded dEIS monitoring traces) [12] in the CloudSim simulator.

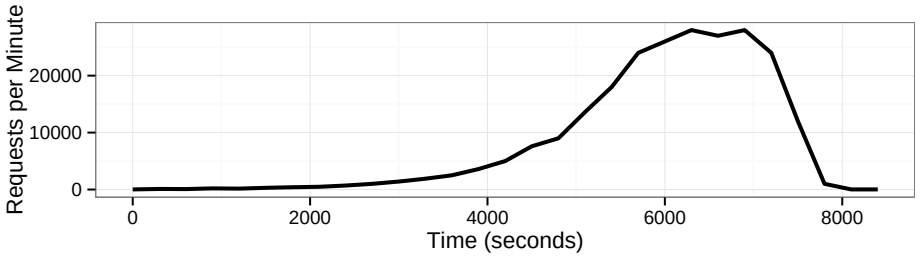
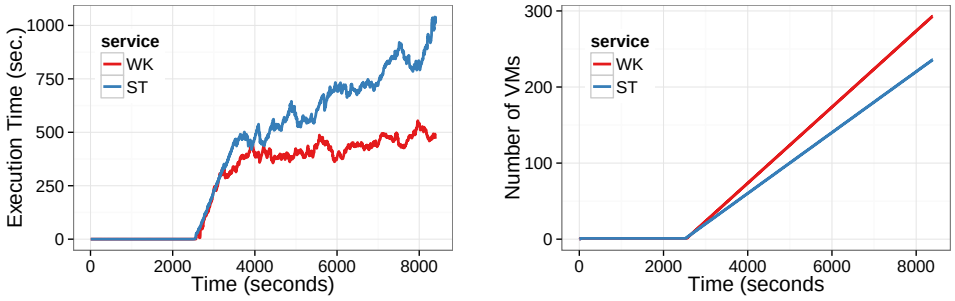


Figure 7.48: "Schwingen"-Like dEIS Workload

We set to compare the *reactive* SLA-based VM-Scaling algorithm [14] with the λ-*based* algorithm described in Section 4.4.2, and the *predictive* λ-*based* algorithm presented in Section 4.4.3. For all three algorithms we will analyse the performance of WK and ST service, by considering the distributions of (1) execution times, (2) the rate of incoming, processed, and dropped requests, and (3) the total number of VMs.

Reactive SLA-Based VM-Scaling Algorithm

We first tested the reactive SLA-based VM-Scaling algorithm, by initially simulating creating one VM for each of the dEIS application's services (CS, LB, WK and ST). After all 4 VMs were created, the workload generator was started at the CS service, which began generating requests according to the workload pattern shown in Figure 7.48.



(a) **(b)**

Figure 7.49: Simulation of the Reactive SLA-Based VM-Scaling Algorithm a) Execution time corresponding to WK and ST services. b) Number of VMs corresponding to WK and ST.

Figure 7.49a shows the average execution time (in seconds) measured at WK and ST services. At simulation time equal to 2400sec the *SLA ratio* parameter, calculated as the ratio between the average execution time (W) measured during the last 60 seconds and the SLA-defined maximum value of W, exceeded the scaling threshold of 0.8, triggering creation of a pair of WK-ST VMs, which were instantiated after 18, respectively 23 seconds, as it can be seen in Figure 7.49b.

The system's behaviour is explained by the fact that once the processing capacity of the available VMs is exceeded, the incoming workload will only delay the execution of the requests that are already being executed. This will lead to an increase for the execution time for all the requests. Combining this with the fact that the considered dEIS application does not queue requests (they begin their execution as soon as they are received), and that the workload never drops to zero during the simulation, helps explain why the execution time keeps increasing until the simulation's end.

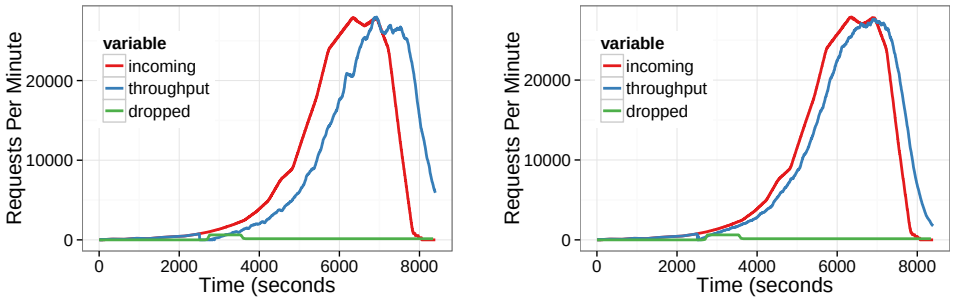


Figure 7.50: Distribution of the arrival rate (red), processing rate (blue), and dropped requests during the simulation of the Reactive SLA-Based VM-Scaling Algorithm for dEIS a) WK service. b) ST service.

Figures 7.50a and 7.50b show the number of inbound (red line), processed (blue line) and dropped (green line) requests at WK, respectively at ST services. Soon after simulation time 2400sec (where the first VM-Scaling for both WK and ST services was performed) it can be observed that the number of requests being processed by the dEIS distributed system drops below the number of inbound requests, creating an imbalance both at the WK and ST services.

The imbalance in processing of requests observed at both WK and ST services caused the accumulation of requests being processed, increasing the system's occupancy and leading to the increase in the average execution time (W), clearly visible in Figure 7.49a.

As the W metric continued to increase, the reactive SLA-based VM-Scaling algorithm continued to observe a *SLA ratio* value above the scaling threshold, triggering the continuous creation of VMs at 20 seconds intervals for the WK service, respectively 25 seconds for the ST service, as the VMs have a instantiation delay of 18 seconds (WK), respectively 23 seconds (ST), and the scaling algorithm was executed at every 5 seconds. This explains the ascending trend of the number of VMs from Figure 7.49b and the different slopes of WK and ST horizontal scaling plots.

Regarding the reason of this behaviour, it lies in the fact the dEIS distributed system was first scaled at a time when the average throughput was higher than the optimal one, as described in Section 7.2.3. This, combined with a constant increasing number of new arrivals, led to the continual accumulation of requests, and an imbalance in the dEIS system's inbound-outbound flow of requests.

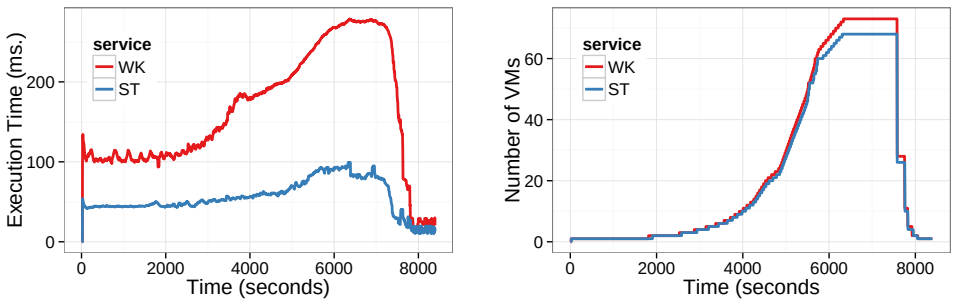
The processing imbalance remained present throughout the simulation, as it can be observed in Figures 7.50a and 7.50b, which show a difference between number of incoming and system's throughput until simulation's end. This also explains why the number of VMs in Figure 7.49b does not

drop to 1 at the end of simulation - because there were still requests being processed and the execution time was over the SLA-defined maximum value.

The defined SLAs were violated during 71% of the simulation's duration as the algorithm did not manage to keep the distributed system in a SLA-compliant state. This shows that the reactive SLA-based VM-Scaling is not suited for scaling system with fast-increasing workloads, however, we will show that the λ-based and predictive-λ-based VM-Scaling algorithms are very suited for such tasks.

λ-Based VM-Scaling Algorithm

The λ-based VM-scaling algorithm was validated with a simulation of the same workload described in Section 7.8.3, Figure 7.48, which has been also used in the previous subsection as well.



(a) **(b)**

Figure 7.51: Simulation of the λ-Based VM-Scaling Algorithm a) Execution time corresponding to WK and ST services. b) Number of VMs corresponding to WK and ST.

The simulation started with one VM per dEIS-service and the target SLAs defined a maximum execution time of 700ms for the WK service, respectively of 300ms for the ST service.

Figure 7.51a shows the average execution time measured over a moving time window of one minute, which was well below the maximum limit set by SLA. The reason why this happened in contrast to the simulation presented in Section 7.8.3 is that this algorithm considered a maximum processing capacity per VM of 435 requests per minute for the WK service, and 456 requests per minute for the ST service. These values were calculated based on the defined SLAs and the results of the benchmark described in Section 7.2.3.

Figure 7.51b shows the evolution of the number of VMs of both WK and ST throughout the simulation. As it can be seen, the first scale-out was performed earlier than in the case of the reactive VM scaling algorithm, after 1815 seconds from the simulation's start, when the average arrival rate,

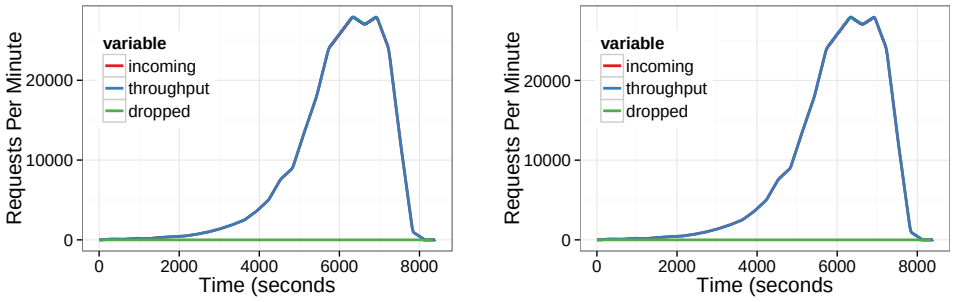


Figure 7.52: Distribution of the arrival rate (red), processing rate (blue), and dropped requests during the simulation of the λ-Based VM-Scaling Algorithm for dEIS a) WK service. b) ST service.

equal to 388 requests per minute (for WK service), exceeded the algorithm's scaling capacity threshold (80% of 435 req. per minute). During the time when the VM scaling-out was signalled and the actual time when the VMs become operational, the number of incoming requests continue to increase, however it did not exceed the services' processing capability.

The maximum number of VMs corresponding to WK service was 73, while for the ST service, the number was 68, given the slightly higher processing capacity of ST service, according to the previously presented benchmark.

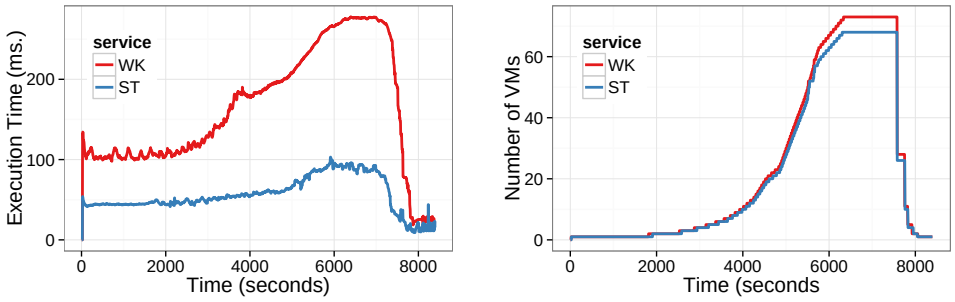
As shown in Figures 7.52a and 7.52b, the processing rate (throughput) of both WK and ST followed closely the arrival rate of requests, validating the algorithm's capacity of maintaining the distributed system in fully SLA-compliant state.

Predictive λ-Based VM-Scaling Algorithm

The predictive-λ-based VM-scaling algorithm was validated with the same workload used in the previous two simulations, described in Section 7.8.3.

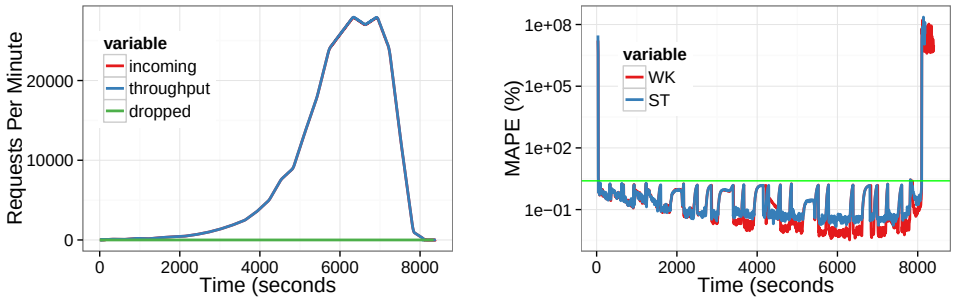
The simulation results match closely the ones from the previous simulation (Figure 7.51), given that the scaling algorithm extended the λ-based VM-scaling algorithm with prediction capabilities. The SLA compliance was 100% for both algorithms, but in the case of the the predictive λ algorithm the scaling-out of VMs was scheduled to happen so that the VMs become operational *exactly* at the time signalled by the control algorithm. This increased the system's protection to sudden increases in workload and kept the spare processing capacity constant instead of diminishing it due to the delay in VMs' instantiation.

Figure 7.53a shows that the average execution time for both the WK and ST services stayed below the SLA-defined maximum values of 700ms for the



(a) **(b)**

Figure 7.53: Simulation of the Predictive-λ-Based VM-Scaling Algorithm a) Execution time corresponding to WK and ST services. b) Number of VMs corresponding to WK and ST.



(a) **(b)**

Figure 7.54: a) Distribution of the arrival rate (red), processing rate (blue), and dropped requests during the simulation of the Predictive-λ-Based VM-Scaling Algorithm for the dEIS WK service. b) Accuracy of predicting the arrival rate at the WK service.

WK service, respectively 300ms for the ST service. Figure 7.53b shows the evolution of the number of VMs for both WK and ST services, which is very similar to the one presented in Figure 7.51b, with the difference that in the case of the predictive-λ-based VM-scaling algorithm the instantiation of VMs happened *exactly* at the moment determined by the scaling algorithm due to predicting the rate of incoming requests.

Figure 7.54a presents the evolution of processing capacity of the WK service, which was perfectly balanced with the incoming workload. Figure 7.54b shows the prediction accuracy, which remained below the threshold of 2.5% (green line) most of the time, enabling using the results of the forecasting as input for the VM-scaling algorithm.

The algorithm achieved maintaining a SLA compliance rate of 100% percent, while also creating a larger margin for the variation of the arrival rate, as the whole system was scaled faster due to the use of prediction.

Chapter 8

Conclusions and Outlook

Cloud Computing has evolved to become an enabler for delivering access to large-scale distributed applications running inside managed environments composed of network-connected computing systems. This made it possible to host Distributed Enterprise Information Systems in cloud environments, while enforcing strict performance and quality of service requirements, defined using Service Level Agreements. The first generation of cloud computing provided on-demand access to single virtual machines (VM), which could be scaled in and out on demand. The expectations of how cloud computing can be used most effectively for enterprise-scale applications have evolved, as existing enterprise applications are not single, monolithic software bundles, but are distributed systems, which can even span data centers and service providers.

In this thesis, we have investigated the use of SLAs as input for management of cloud infrastructure, with regards to providing a machine-readable representation of distributed applications, including composition relations between their services, monitoring metrics, key performance indicators, performance guarantees and automated scaling actions. We then showed how these enhanced SLAs can be used for building several algorithms and optimisation methods for management of both physical and virtual cloud-resources.

We describe in Section 8.1 the initial research questions and the corresponding solutions provided by this thesis. These contributions are then summarized in Section 8.2, and finally Section 8.3 outlines promising research topics for future work related to our contributions.

8.1 Challenges Addressed

We considered multiple connected problems from the perspective of both enterprise application owners and cloud infrastructure providers regarding the efficient management of SLAs, in the context of scaling distributed applications using virtual cloud-resources.

The first perspective is that of the application owner, who is interested

in finding the optimum amount of computing and network resources to ensure that the performance requirements of all applications are met. After the initial allocation of virtual computing resources, the application owner is also interested in appropriately scaling distributed applications so that application performance guarantees are maintained even under dynamic user-generated workload conditions.

Similarly, the infrastructure providers are interested in optimally allocating the virtual resources on the physical infrastructure resources so that its operational costs are minimised, while maximising the tenant's application performance. In this context, we show how predicting the infrastructure utilisation peaks (using both triple exponential smoothing and a streaming version of linear regression combined with feedback based on the out-of-sample error) and dynamically optimising the distribution of virtual resources (by using a VM allocator using a group-oriented genetic algorithm, combined with VM-scaling algorithms based on benchmarking and Little's Law) leads to optimising the use of available cloud resources, lowering the penalties caused by violating the tenant's performance-SLAs during the live-migration of VMs.

Therefore, we raised the following question: "**How do distributed application owners and infrastructure providers benefit from using Service Level Agreements in Cloud Computing environments?**". The answer to this question was provided in Chapter 3 where we presented an extensible, machine-readable semantic language for representing distributed applications along with their service levels agreements. Our proposed enhanced SLA specification language allows representing both (1) structural elements of distributed applications (services) along with infrastructure resources required for the proper instantiation of these services, as well as (2) the application specific performance metrics and quality of service parameters, which are then used for specifying application-specific invariants (e.g. guarantees) and actions that must be executed when the application's state changes towards violating the SLA guarantees. The validation for this solution was described in Chapter 7.3, where we looked at the benefits of using SLAs for controlling virtual resource allocation in cloud environments, compared to using different fixed allocation policies. Our results indicated that a dynamic allocation of cloud resources is clearly more valuable than both over- and under-provisioned assignment of infrastructure cloud-resources.

Having created a flexible representation language for the description of distributed applications and SLAs, we decided on exploring the resource allocation problem from the perspective of the cloud-infrastructure provider, consequently, trying to answer the following research problem: **"How to control infrastructure resource allocation in cloud environments while maximising both tenants' Quality of Experience and provider's Efficiency of Operations?"**. The answer to this question was provided in Chapter 6 where we described a resource allocation algorithm, which is capable of taking into ac-

count multiple optimization criteria, such as the impact of over-subscribing computing and network resources, energy costs, SLA penalties and generated profits. We then evaluated this algorithm in Section 7.5, showing that this approach provides more benefits to the cloud provider compared to simpler resource allocation algorithms, such as first-, worst- or best-fit heuristics.

At the cloud-platform level, the tenants are interested in obtaining a constant application-level performance, independent of the number of external users accessing their cloud-applications. This behaviour can be obtained if the cloud-platform-manager ensures that the appropriate number of virtual resources (e.g. VMs) are allocated to the distributed applications. Optimally determining the number of necessary VMs for supporting given levels of application-performance leads to three distinct sub-problems: (1) finding the relation between the quantity of virtual resources (or size of the virtual infrastructure) allocated to distributed cloud-applications and the corresponding application-level performance metrics; (2) finding the optimal VM-scaling policy; and (3) using prediction to optimally vary the number of VMs allocated to cloud applications in order to minimise resource utilisation while meeting the agreed SLA performance guarantees.

Given these objectives and the research question of **"How to design an effective system for management (scaling) of SLA-constrained distributed cloud services?"**, we described in Chapters 5 and 4 (1) several mechanisms for optimisation the scaling rules contained in SLAs by using statistical correlations between application-level monitoring metrics, and (2) multiple algorithms for scaling distributed applications using prediction and results from applying Little's Law to distributed systems, while considering the impact on the given SLA guarantees. We then described several experiments involving scaling a distributed enterprise application, as well as several large-scale simulations, which have validated the previously presented SLA-optimization methods and SLA-based VM-scaling algorithms.

8.2 Summary of Contributions

Overall, the key contributions of this dissertation are:

- (Chapter 3) a semantic SLA-enabled specification language and architecture for dynamically managing distributed software, together with the corresponding computing, storage and network cloud infrastructure virtual and physical resources [18]
- (Chapter 4) multiple SLA-based VM scaling algorithms that use reactive and analytic mechanisms combined with data prediction method, as well as results from applying Little's Law ([14, 12, 15])
- (Chapter 5) a method of sizing virtual infrastructures based of SLA-defined constraints and benchmark-gathered application profiling in-

formation [16], and composing scaling rules for distributed services using prediction mechanisms and correlation-derived relationships between SLA monitoring metrics [11]
- (Chapter 6) a framework for dynamically allocating VMs to physical resources while considering SLA-constraints and multiple objectives optimisations which uses a bin-oriented genetic algorithm combined with data forecasting (exponential smoothing) [19]

The *enhanced semantic specification language for SLA representation and topology orchestration* described in Chapter 3 is addressing the first research question (*"How do distributed application owners and infrastructure providers benefit from using Service Level Agreements in Cloud Computing environments?"*) by enabling both cloud providers and users to express their requirements in terms of available infrastructure capabilities, as well as application performance requirements and service dependencies.

The VM scaling algorithms and management system's architecture presented in Chapter Chapter 4 are providing an answer to the research question of (*"How to control the allocation of infrastructure resources in cloud environments while maximising both tenant's Quality of Service and provider's Efficiency of Operations?"*).

Finally, the third research question (*"How to design a system for management (scaling) of SLA-constrained distributed cloud services?"*) is addressed in Chapters 5 and 6. In Chapter 5 we present a method for analysing the dependencies between performance metrics of distributed application based on benchmarks-generated datasets, as well as the design of a system for analysing these dependencies and producing scaling plans, which can them be used for the management of distributed applications. Chapter 6 provides the answer to the multi-objective mapping of virtual machines to physical infrastructure resources.

We discuss next each of the main contribution points of this thesis in detail.

Enhanced Semantic Specification Language for SLA Representation and Topology Orchestration

We have presented an SLA-centric specification model, architecture and value model for dynamic application topology specification and orchestration. Our specification model is designed to simplify validation of selection, deployment and adaptation rules for application to infrastructure allocation of resources. Secondly, the specification provides application architects with an application management model that is applicable at specification and operation time, separating the concerns of specification from configuration. Subsequently, given an architecture like we proposed, such a specification increases the level of infrastructure management automation, including provisioning, deployment, monitoring, problem specification and resolution, as

an explicit expression of locality, scaling and adaptation constraints are included for each class of infrastructure resource: storage, computation and networking.

Our results show that the dynamic approach of allocating services (as opposed to having a fixed allocation) is a better choice, considering the objectives of minimising SLA violations and maximising efficiency. A qualitative estimation of the average reaction time showed advantages for the dynamic approach under basic assumptions about the readiness of over and under-provisioned strategies for response to adaptation triggers. However, there is some overhead introduced by the need to monitor more metrics with a greater frequency, and the occurrence of transient non-availabilities during adaptation of the infrastructure.

SLA-Based Reactive VM-Scaling Algorithm

We have shown how CloudSim can be used as a simulation platform for testing SLA-based infrastructure scaling policies using application performance traces recorded from a small-scale cloud deployment. We described, implemented and validated a time-shared parallel cloudlet scheduler for CloudSim, which we used for building and evaluating a SLA scaling manager for VMs, by running three simulations of varying workloads in a multi-tenant cloud environment.

We have also proposed and validated a CloudSim model for translating application-level performance profiling information to VM-level CloudSim scheduler resource utilization level. We have also identified some possible optimization points in cloud infrastructure management, regarding energy consumption of idle servers. We have shown that SLA guarantees can be used for VM scaling purposes when it is possible to convert them to SLA ratios.

SLA-Based Predictive VM Scaling Algorithms

We presented two new VM-scaling algorithms focused on distributed enterprise applications, which can be used by cloud infrastructure management systems to optimally detect the most appropriate scaling conditions using performance-models of distributed applications derived from constant-workload benchmarks. We have shown how to combine benchmark results, with Little's Law and SLAs for identifying the optimal processing capacity of cloud services, and then we used it in a SLA and arrival rate-based VM-scaling algorithm. We have extended the arrival rate-based VM scaling algorithm to consider prediction of the arrival rate, by using linear regression and multi-step forecasting.

We have evaluated a total of three VM-scaling algorithms by simulating them in a cloud simulator against trace-based performance models of a

distributed application, using a real-world application scenario involving a large variable number of users. Our results show that using predictive SLA-driven scaling algorithms in cloud management systems for guaranteeing performance invariants of distributed cloud applications improves the management efficiency of infrastructure-cloud management systems, as opposed to using only reactive SLA-based VM-scaling algorithms.

System for Dynamic Composition of SLA Scaling Rules Based on Concurrent Benchmarks

We have presented a way of dynamically optimising SLAs by generating SLA-bounded scaling models for the VM-deployed services of the distributed applications. The resulting SLA scaling rules enable a service management system to react to the variations in the number of application users, while minimizing the number of SLA violations, therefore facilitating service scale-out and service-in. We have described the analytic and benchmark mechanisms for dynamically generating the SLA scaling rules. By using heuristics for selecting the appropriate scaling-out paths for the services of distributed applications, we have shown how SLA scaling rules can be inferred and then used for controlling the runtime scale-in and scale-out of VM-bounded services. We have validated our architecture and models by performing scaling experiments with a distributed application representative for the enterprise class of information systems. We have then shown how dynamically generated SLAs can be successfully used for controlling the management of distributed services belonging to a distributed application.

Method for Analytic Optimization of SLAs for Scaling of Distributed Services

We have presented a system enabling dynamic composition, instantiation and control of distributed applications using guaranteed actions and guaranteed states contained in SLAs. We have shown how SLAs can be used for controlling the allocation and scaling behavior of services belonging to distributed applications. We have presented how predictions can be used for improving the allocation of resources, when prior monitoring information exists for the system under test. Finally, we have validated our system by conducting multiple experiments using a distributed enterprise application running in a distributed European testbed. We have also experimented with improving SLAs and application performance response using system-provided correlations between the different monitoring metrics of the services belonging to a distributed application.

Statistical Modelling of the Performance of a Distributed Enterprise Application

We have presented an approach for building a simulation model of a distributed application for concurrent workload processing, by analyzing ap-

plication's performance traces gathered in small-scale real deployments. We then showed how to integrate the simulation model into CloudSim and how to build a simulation with varying concurrent workload. We have also presented an approach for modeling concurrent computing workloads using the CloudSim simulator. Finally, we validated our models by running experiments with varying workload and with different VM scaling factors.

Our results show that it is possible to accurately model the behavior of a distributed enterprise information system using CloudSim, although extensions for supporting concurrent task processing need to be added. Also, due to variance in the execution times of distributed requests, the simulation models cannot perfectly represent the real distributed systems, however, a good-enough representation of key application-level metrics has been shown to be obtainable, by means of statistical analysis.

Multiple-Objective Infrastructure-Resource Allocation Algorithm

We described a system for supporting the planning and load distribution disciplines of SLA management while taking into consideration multiple objective optimizations and the impact of SLAs into resource provisioning and into dynamic scaling of the virtual infrastructures. We proposed a way of combining resource utilization estimation, with cost prediction and impact of the infrastructure operations for implementing the complete set of disciplines used in SLA management. We validated our model using simulation data and we were able to show that our proposed resource allocation approach significantly outperforms several standard heuristics.

8.3 Outlook

The work conducted in this thesis opens a broad range of possible directions for future research and innovation in the field of SLA-based management of distributed services. We briefly elaborate on some of the most promising topics that relate to our contributions.

Future work can hence be oriented towards a more empirical analysis of infrastructure adaptation, towards minimising the reaction time, as well as assessing the impact on management effort. We believe that the main way of minimising reaction time is through more accurate prediction, combined with long-term scheduling algorithms. Further studies can therefore be targeted at incorporating and assessing the value of predictive algorithms and methods with a longer prediction horizon in cloud computing management systems.

Another possibility of using the prediction mechanisms is to use them also for controlling the scale-down behaviour of the distributed services based on the forecasted load and the duration of the services start-up period, combined with the possibilities of scheduling infrastructure actions.

With regards to optimising SLAs using data analysis mechanisms, we mention using statistical algorithm-based analytics, for (1) detecting periodicity patterns in the SLA monitoring data, and (2) including these patterns as triggers of SLA actions.

Given our work for modelling the resource consumption of distributed applications for simulation purposes, we envision refining the models of VM resource consumption by taking into consideration the distribution of computing resources between software-tasks, the effect of background VM services and impact on other VM resources, such as disks. Also, the simulation models can be further improved by explicit consideration of a more complex state model for the CPU, including wait states and effect of I/O operations.

We can also identify a possibility of conducting further research in the field of performance characterisation of distributed systems, in particular with regards to applying the Little's Law to always-on systems, in which the workload never drops to zero. In such systems it is difficult to enforce the stability constrains requested by Little's Law and therefore a differential approach should be investigated.

Chapter 9

Annex

9.1 Complete SLA Example in Enhanced-USDL-SLA

```
@prefix : <http://sap.com/research/sla#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix sla: <http://sap.com/research/sla#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix usdl-sla: <http://www.linked-usdl.org/ns/usdl-sla#> .
@prefix usdl-core: <http://www.linked-usdl.org/ns/usdl-core#> .
@base <http://sap.com/research/sla> .

<http://sap.com/research/sla> rdf:type owl:Ontology ;
 owl:imports <http://www.linked-usdl.org/ns/usdl-core> ,
 <http://www.linked-usdl.org/ns/usdl-sla> .

#################################
#    Annotation properties
#################################

###  http://www.linked-usdl.org/ns/usdl-sla#hasServiceLevel
usdl-sla:hasServiceLevel rdf:type owl:AnnotationProperty .

###  http://www.linked-usdl.org/ns/usdl-sla#hasServiceLevelProfile
usdl-sla:hasServiceLevelProfile rdf:type owl:AnnotationProperty .

###  http://www.linked-usdl.org/ns/usdl-sla#hasVariable
usdl-sla:hasVariable rdf:type owl:AnnotationProperty .

###  http://www.linked-usdl.org/ns/usdl-sla#serviceLevelExpression
```

```
30 usdl-sla:serviceLevelExpression rdf:type owl:AnnotationProperty .
31
32 ###############################
33 #    Object Properties
34 ###############################
35
36 ###  http://sap.com/research/sla#hasActionPrecondition
37 :hasActionPrecondition
38       rdf:type owl:ObjectProperty ;
39             rdfs:domain usdl-sla:GuaranteedAction ;
40       rdfs:range usdl-sla:ServiceLevelExpression .
41
42 ###  http://sap.com/research/sla#hasConstant
43 :hasConstant
44       rdf:type owl:ObjectProperty ;
45             rdfs:range :Constant ;
46       rdfs:domain usdl-sla:ServiceLevelExpression .
47
48 ###  http://sap.com/research/sla#hasMetricType
49 :hasMetricType
50       rdf:type owl:ObjectProperty ;
51          rdfs:range :MetricType ;
52          rdfs:domain :MonitoringMetric .
53
54 ###  http://sap.com/research/sla#hasMonitoringMetric
55 :hasMonitoringMetric
56       rdf:type owl:ObjectProperty ;
57          rdfs:range :MonitoringMetric ;
58       rdfs:domain usdl-core:Service .
59
60 ###  http://sap.com/research/sla#isAssociatedWith
61 :isAssociatedWith
62       rdf:type owl:ObjectProperty ;
63          rdfs:domain usdl-sla:Variable .
64
65 ###  http://sap.com/research/sla#refersToMetric
66 :refersToMetric rdf:type owl:ObjectProperty .
67
68 ###  http://sap.com/research/sla#refersToServiceType
69 :refersToServiceType
70       rdf:type owl:ObjectProperty ;
71          rdfs:range usdl-core:Service ;
72       rdfs:domain usdl-sla:Variable .
```

```
73
74  ###################################
75  #    Data properties
76  ###################################
77
78  ###  http://sap.com/research/sla#hasEvaluationRegion
79  :hasEvaluationRegion rdf:type owl:DatatypeProperty ;
80                       rdfs:domain usdl-sla:Variable .
81
82  ###  http://sap.com/research/sla#hasExpressionPreprocessor
83  :hasExpressionPreprocessor rdf:type owl:DatatypeProperty .
84
85  ###  http://sap.com/research/sla#hasRepresentation
86  :hasRepresentation rdf:type owl:DatatypeProperty .
87
88  ###################################
89  #    Classes
90  ###################################
91
92  ###  http://sap.com/research/sla#Constant
93  :Constant rdf:type owl:Class .
94
95  ###  http://sap.com/research/sla#MetricType
96  :MetricType rdf:type owl:Class .
97
98  ###  http://sap.com/research/sla#MonitoringMetric
99  :MonitoringMetric rdf:type owl:Class .
100
101 ###  http://www.linked-usdl.org/ns/usdl-core#Service
102 usdl-core:Service rdf:type owl:Class .
103
104 ###  http://www.linked-usdl.org/ns/usdl-sla#GuaranteedAction
105 usdl-sla:GuaranteedAction rdf:type owl:Class .
106
107 ###  http://www.linked-usdl.org/ns/usdl-sla#GuaranteedState
108 usdl-sla:GuaranteedState rdf:type owl:Class .
109
110 ###  http://www.linked-usdl.org/ns/usdl-sla#ServiceLevelExpression
111 usdl-sla:ServiceLevelExpression rdf:type owl:Class .
112
113 ###  http://www.linked-usdl.org/ns/usdl-sla#ServiceLevelProfile
114 usdl-sla:ServiceLevelProfile rdf:type owl:Class .
115
```

```
116 ### http://www.linked-usdl.org/ns/usdl-sla#Variable
117 usdl-sla:Variable rdf:type owl:Class .
118
119 ### http://www.w3.org/2000/01/rdf-schema#Resource
120 rdfs:Resource rdf:type owl:Class .
121
122 ###################################
123 #    Individuals
124 ###################################
125
126 ### http://sap.com/research/sla#ApplicationMetric
127 :ApplicationMetric rdf:type :MetricType ,
128                           owl:NamedIndividual .
129
130 ### http://sap.com/research/sla#ConsumerService
131 :ConsumerService rdf:type usdl-core:Service ,
132                          owl:NamedIndividual .
133
134 ### http://sap.com/research/sla#InfrastructureMetric
135 :InfrastructureMetric rdf:type :MetricType ,
136                               owl:NamedIndividual .
137
138 ### http://sap.com/research/sla#InstancesCountMetric
139 :InstancesCountMetric rdf:type :MonitoringMetric ,
140                               owl:NamedIndividual ;
141
142                       :hasMetricType :ApplicationMetric .
143
144 ### http://sap.com/research/sla#LoadBalancerService
145 :LoadBalancerService rdf:type usdl-core:Service ,
146                              owl:NamedIndividual ;
147     usdl-sla:hasServiceLevelProfile :loadBalancerSLAProfile .
148
149 :RunningOrStartingInstancesCountMetric
150     rdf:type :MonitoringMetric , owl:NamedIndividual ;
151     :hasMetricType :ApplicationMetric .
152
153 ### http://sap.com/research/sla#ServiceMetric
154 :ServiceMetric rdf:type :MetricType ,
155                        owl:NamedIndividual .
156
157 ### http://sap.com/research/sla#StorageService
158 :StorageService rdf:type usdl-core:Service ,
```

```
159                     owl:NamedIndividual .
160
161 ###  http://sap.com/research/sla#WorkerService
162 :WorkerService rdf:type usdl-core:Service ,
163                     owl:NamedIndividual ;
164             usdl-sla:hasServiceLevelProfile :workerSLAProfile .
165
166 ###  http://sap.com/research/sla#loadBalancerSLAProfile
167 :loadBalancerSLAProfile rdf:type usdl-sla:ServiceLevelProfile ,
168                             owl:NamedIndividual ;
169     usdl-sla:hasServiceLevel
170         :workerServiceInstancesPerConnectedUsersState .
171
172 ###  http://sap.com/research/sla#loadbalancerService
173 :loadbalancerService rdf:type rdfs:Resource ,
174                         owl:NamedIndividual .
175
176 ###  http://sap.com/research/sla#networkBandwidthScaleDownAction
177 :networkBandwidthScaleDownAction
178     rdf:type usdl-sla:GuaranteedAction ,
179         owl:NamedIndividual ;
180     :hasActionPrecondition :networkScaleDownCondition .
181
182 ###  http://sap.com/research/sla#networkBandwidthScaleUpAction
183 :networkBandwidthScaleUpAction
184     rdf:type usdl-sla:GuaranteedAction ,
185         owl:NamedIndividual ;
186     :hasActionPrecondition :networkScaleUpCondition .
187
188 ###  http://sap.com/research/sla#networkScaleDownCondition
189 :networkScaleDownCondition
190     rdf:type usdl-sla:ServiceLevelExpression ,
191         owl:NamedIndividual .
192
193 ###  http://sap.com/research/sla#networkScaleUpCondition
194 :networkScaleUpCondition
195     rdf:type usdl-sla:ServiceLevelExpression ,
196         owl:NamedIndividual .
197
198 ###  http://sap.com/research/sla#numberOfActiveRequestsMetric
199 :numberOfActiveRequestsMetric rdf:type :MonitoringMetric ,
200                                 owl:NamedIndividual .
201
```

```
:numberOfActiveSessionsAtLoadBalancerVariable
    rdf:type usdl-sla:Variable , owl:NamedIndividual ;
    :refersToServiceType :LoadBalancerService ;
    :refersToMetric :numberOfActiveSessionsMetric .

### http://sap.com/research/sla#numberOfActiveSessionsMetric
:numberOfActiveSessionsMetric rdf:type :MonitoringMetric ,
                                       owl:NamedIndividual ;
                              :hasMetricType :ServiceMetric .

### http://sap.com/research/sla#workerSLAProfile
:workerSLAProfile rdf:type usdl-sla:ServiceLevelProfile ,
                           owl:NamedIndividual ;
    usdl-sla:hasServiceLevel :workerServiceScaleDownAction ,
        :workerServiceScaleUpAction .

### http://sap.com/research/sla#workerScaleDownCondition
:workerScaleDownCondition
    rdf:type usdl-sla:ServiceLevelExpression ,
        owl:NamedIndividual ;
   :hasRepresentation "scaleDownConditionValue == true"^^xsd:string ;
   :hasExpressionPreprocessor """
var averageNumberOfActiveSessionsAtLoadBalancerVariable =
        average(numberOfActiveSessionsAtLoadBalancerVariable);
var scaleDownThreshold =
    (workerServiceInstancesCountVariable - 1.0) *
    workerServiceCapacityConstant;
var scaleDownConditionValue =
    averageNumberOfActiveSessionsAtLoadBalancerVariable <
        scaleDownThreshold;
System.out.println(\"Average number of active user sessions: \" +
 averageNumberOfActiveSessionsAtLoadBalancerVariable);
System.out.println(\"Current Worker Service instances count: \" +
    workerServiceInstancesCountVariable +
    \" (scale-down threshold: \" + scaleDownThreshold + \" ) \");
if (scaleDownConditionValue) {
    System.out.println(\"Service Scale-DOWN triggered.\");
}
""" ;
    usdl-sla:hasVariable :numberOfActiveSessionsAtLoadBalancerVariable ;
    :hasConstant :workerServiceCapacityConstant ;
    usdl-sla:hasVariable :workerServiceInstancesCountVariable .

```

```
### http://sap.com/research/sla#workerScaleUpCondition
:workerScaleUpCondition
    rdf:type usdl-sla:ServiceLevelExpression ,
        owl:NamedIndividual ;
    :hasExpressionPreprocessor """
var currentWorkerServicesCapacity =
    workerServiceRunningOrStartingCountVariable *
    workerServiceCapacityConstant;
var averageNumberOfActiveSessionsAtLoadBalancerVariable =
        average(numberOfActiveSessionsAtLoadBalancerVariable);
var workersAreApproachingTheSLALimit =
    averageNumberOfActiveSessionsAtLoadBalancerVariable >=
    currentWorkerServicesCapacity;
System.out.println(\"Current user handling \"+
    \"capacity of all Worker Services: \" +
    currentWorkerServicesCapacity);
System.out.println(\"Average number of active user sessions: \" +
        averageNumberOfActiveSessionsAtLoadBalancerVariable);
if (workersAreApproachingTheSLALimit) {
    System.out.println(\"Service Scale-UP triggered.\");
} """ ;
    :hasRepresentation "workersAreApproachingTheSLALimit == true" ;
    usdl-sla:hasVariable
        :numberOfActiveSessionsAtLoadBalancerVariable ;
    :hasConstant
        :workerServiceCapacityConstant ;
    usdl-sla:hasVariable
        :workerServiceRunningOrStartingCountVariable .

:workerServiceCapacityConstant rdf:type :Constant ,
                                owl:NamedIndividual ;
                        :hasRepresentation "5" .


:workerServiceInstancesCountVariable
    rdf:type usdl-sla:Variable ,
         owl:NamedIndividual ;
    :refersToMetric :InstancesCountMetric ;
    :refersToServiceType :WorkerService .

:workerServiceInstancesPerConnectedUsersExpression
    rdf:type usdl-sla:ServiceLevelExpression ,
        owl:NamedIndividual ;
```

```
288     :hasRepresentation
289 "average(numberOfActiveSessionsAtLoadBalancerVariable) <=
290     workerServiceInstancesCountVariable *
291     workerServiceMaxCapacityConstant" ^^xsd:string ;
292     usdl-sla:hasVariable
293         :numberOfActiveSessionsAtLoadBalancerVariable ,
294         :workerServiceInstancesCountVariable ;
295     :hasConstant :workerServiceMaxCapacityConstant .
296
297 :workerServiceInstancesPerConnectedUsersState
298     rdf:type usdl-sla:GuaranteedState ,
299         owl:NamedIndividual ;
300     usdl-sla:serviceLevelExpression
301         :workerServiceInstancesPerConnectedUsersExpression .
302
303 :workerServiceMaxCapacityConstant
304     rdf:type :Constant , owl:NamedIndividual ;
305     :hasRepresentation "10" .
306
307 :workerServiceRunningOrStartingCountVariable
308     rdf:type usdl-sla:Variable , owl:NamedIndividual ;
309     :refersToMetric :RunningOrStartingInstancesCountMetric ;
310     :refersToServiceType :WorkerService .
311
312 :workerServiceScaleDownAction
313     rdf:type usdl-sla:GuaranteedAction , owl:NamedIndividual ;
314     :hasActionPrecondition :workerScaleDownCondition ;
315     usdl-sla:serviceLevelExpression
316         :workerServiceScaleDownActionExpression .
317
318
319 :workerServiceScaleDownActionExpression
320     rdf:type usdl-sla:ServiceLevelExpression ,
321         owl:NamedIndividual ;
322     :hasRepresentation
323 """var outcome = new java.util.HashMap();
324 outcome.put(\"rule-action\", \"service-scale-down\");
325 outcome.put(\"service-name\", \"WorkerService,StorageService\");
326 return outcome;""" .
327
328 ###  http://sap.com/research/sla#workerServiceScaleUpAction
329 :workerServiceScaleUpAction
330     rdf:type usdl-sla:GuaranteedAction ,
```

```
331         owl:NamedIndividual ;
332     :hasActionPrecondition :workerScaleUpCondition ;
333     usdl-sla:serviceLevelExpression
334         :workerServiceScaleUpActionExpression .
335
336 ###  http://sap.com/research/sla#workerServiceScaleUpActionExpression
337 :workerServiceScaleUpActionExpression
338     rdf:type usdl-sla:ServiceLevelExpression ,
339         owl:NamedIndividual ;
340     :hasRepresentation
341 """var outcome = new java.util.HashMap();
342 outcome.put(\"rule-action\", \"service-scale-up\");
343 outcome.put(\"service-name\", \"WorkerService\");
344 return outcome;"""^^xsd:string .
```

9.2 Little's Law Experimental Results

Table 9.1: Dependency between average concurrency (L), average execution time in milliseconds (W), average arrival rate (λ), and average throughput (Th) for the dEIS WK services

WK			
L	W	λ	Th
1.00000000000001	106.253175824763	118.0	133.402347559805
1.49209853730371	135.892854116163	236.0	263.615378438970
1.97765558945569	236.681193020889	333.3	366.391575874631
2.47256642885760	296.383069425486	355.5	392.945454399697
3.01893362528895	461.587023767713	315.7	362.823813497585
3.60030399746622	532.352964201406	358.8	406.079563427756
4.39220926885756	615.453746603090	377.8	419.509814133499
5.11854236888272	619.072416789109	414.8	462.415636632722
5.83689676700349	666.976063107783	435.4	483.388705827361
6.63212289306503	733.061835860666	442.2	491.276316774205
7.41423602440646	792.195126673492	456.7	513.412421758801
8.18564360062547	837.802534619118	470.0	527.413455784417
9.08582787863719	928.526782276457	435.4	479.495752545885
9.69589982826333	911.255316803602	470.7	524.184700910471
9.35991394532730	771.795157203942	56.2	614.143901691528
11.2358461793066	934.863415157991	526.2	573.564867732920
12.2431820442801	995.568303271454	542.1	595.962974534828
13.1560639251941	1057.30109519794	552.8	599.725248959365
14.0436654406718	1099.35802480836	564.3	621.791344674921

14.9581159089741	1145.53360019043	577.7	634.911298143772
16.0216243076700	1211.53717647379	513.2	570.962763895424
17.3502289870072	1351.96160752844	459.1	512.949261547913
18.5083315266693	1413.13576143710	434.7	486.184171872724
19.4048813543123	1560.97410727780	405.1	456.359405076374
20.8294962525027	1664.70749366665	369.4	414.070283333598
21.8984129656136	1825.53316907034	345.1	385.826988320014
23.3101195854305	2106.68267229236	308.4	352.080416772491
24.6660809658003	2247.65501647306	285.3	335.762969610098
25.8968267382830	2676.22797209435	248.3	286.564419152066
27.1659519661257	2887.84524287492	227.0	266.324822314658
28.2986701403029	3242.03163125361	206.6	243.523719304605
29.5431997657446	3740.22673873201	178.6	219.988897011040
30.8801342719387	4630.11541141157	143.8	182.738088933414
31.9198633887045	5399.11204843935	127.8	171.465091203173
33.2360679787079	6506.76222859959	105.9	148.262136451378
34.4096464604505	8103.85897928056	91.8	135.505538843127
35.6362743819602	9249.97267210008	77.8	122.885119920617
36.6048632571997	11982.7034405657	63.0	135.736468722656
37.7987565418121	13870.7820891895	52.5	196.620710165341
38.8322274047968	16225.3426282825	45.9	93.8370381111032
39.6953979042764	21662.0884702283	38.5	95.5345712000983
41.0707989926740	23237.1940615593	28.0	73.0596832933760
41.1865747289015	26307.4126039314	28.6	77.8141164818046
43.0394503546099	39817.8373147920	19.9	80.1253979768785
44.3669283097855	55174.2663698470	17.2	98.8848615886668
45.6123263888889	52395.8898960765	12.4	72.4340269418790
46.1361828774063	76838.6210102231	11.9	108.671856505841
47.6818930041152	93300.8123419405	7.0	73.3100932958845
48.8130341880342	132771.381409054	6.4	45.5205607334689
49.8968253968254	159679.602527441	3.8	73.3356519392628

Table 9.2: Dependency between average concurrency (L), average execution time in milliseconds (W), average arrival rate (λ), and average throughput (Th) for the dEIS ST services

ST			
L	W	λ	Th
1.00000000000000	44.1364749181384	118	133.379910942919
1.47848686037278	59.5419776964693	236	260.45383845421
1.91999218138395	78.5575478478216	333.3	363.105213983859
2.23688489269291	99.7188234216933	355.5	395.659723561377

2.77404817736323	129.578414279590	315.7	381.528205154811
2.95331179724445	152.419315480892	358.8	418.438897585834
3.05369194176416	175.700209885490	377.8	433.727042215522
3.12136360786964	201.765625928640	414.8	476.121573003631
3.30678154677553	229.442377456606	435.4	497.158241115614
3.55941046923965	262.060833655668	442.2	510.376770050187
3.82791140908309	291.842081880850	456.7	515.457408180452
4.07817594845672	321.984808326989	470	529.618827975436
4.32237718295087	430.762906950733	435.4	477.846609113416
4.94937434449424	448.324755806419	470.7	523.948495578588
6.22546681630632	631.583241187162	56.2	588.724671869038
5.79329962240557	477.499313113676	526.2	584.500083292696
5.98612587342152	483.959131409734	542.1	598.297421253130
6.35912827611391	496.279205682649	552.8	606.039348898504
6.66014367963151	517.013824084418	564.3	618.685902058795
6.91036188285654	537.157086242621	577.7	635.958019476017
8.04327679530483	799.712717423534	513.2	562.230461192874
8.98507215133268	1078.11516057179	459.1	501.046274544561
10.0331799474438	1301.21405895084	434.7	473.559706642170
10.8792115495752	1500.17541640450	405.1	444.762894251641
12.0443069088006	1858.00083109819	369.4	405.106900988764
12.9671203586527	2113.10564938642	345.1	384.049776315713
13.9996860621033	2534.51352073512	308.4	349.845328218091
14.9988257536138	3012.59956331101	285.3	318.987771550790
15.9430091145108	3592.23410973396	248.3	282.229125217732
16.8753272857303	4222.07029225332	227	269.201443368447
17.7441692777037	4967.42344365356	206.6	237.427120701460
18.7956419465257	6564.07507081400	178.6	212.027904562730
19.6351598671498	8163.95851777649	143.8	176.171826879828
20.6353045350298	9299.52958628097	127.8	163.260658873135
22.1026456798327	12211.6323742269	105.9	138.529122322352
22.5935675665879	13707.6073089660	91.8	123.668054636167
24.1193802991880	17530.8144341426	77.8	112.649435071334
24.1933736017069	20406.0826289865	63.0	99.9264567610490
26.2239649239649	29377.9176209859	52.5	88.2885981574206
25.8570014980271	31233.1855246229	45.9	85.1585627799066
27.7547105911330	33796.6307878469	38.5	91.2779943352144
28.8909551176793	44034.3969891253	28.0	77.75752872945490
28.3240333740334	43647.7319287173	28.6	91.5182388521269
29.6584656084656	58770.2043950417	19.9	75.9436914407777
28.4602011494253	69040.9992810433	17.2	95.5058075816268

31.2779710144928	95628.7775353784	12.4	64.2899806059830
31.1658119658120	85808.4478626768	11.9	62.8785073819733
33.2920289855072	115358.438405161	7.0	107.5525080689090
29.65486725663720	113851.123155687	6.4	86.689078530392
30.3407079646018	224803.927727526	3.8	79.8991928823325

Bibliography

[1] "SLA@SOI," http://sla-at-soi.eu/, 2012.

[2] "Sugar crm," http://www.sugarcrm.com/.

[3] Abiquo, "Abicloud," http://www.abiquo.com, ian. 2015.

[4] S. Albers and M. Mitzenmacher, "Average-case analyses of first fit and random fit bin packing." in *SODA*, vol. 98, 1998, pp. 290–299.

[5] O. V. Alliance, "Kernel based virtual machine (kvm)," http://www.linux-kvm.org, ian. 2015.

[6] O. Alliance, "OSGi-the dynamic module system for Java," *accessed, May*, vol. 25, 2009.

[7] Amazon, "Amazon elastic compute cloud (amazon ec2)," http://aws.amazon.com/ec2, 2015.

[8] Amazon Web Services, "Amazon EC2 service level agreement," http://aws.amazon.com/ec2-sla/.

[9] L. Andersen and S. Lead, "Jdbc 4.0 specification," *JSR*, vol. 221, pp. 1–126, 2006.

[10] V. Andrikopoulos, S. G. Sáez, F. Leymann, and J. Wettinger, "Optimal distribution of applications in the cloud," in *Advanced Information Systems Engineering*, ser. Lecture Notes in Computer Science, vol. 8484. Springer International Publishing, 2014, pp. 75–90.

[11] A.-F. Antonescu and T. Braun, "Improving management of distributed services using correlations and predictions in SLA-driven cloud computing systems," in *Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS)*, May 2014.

[12] ——, "Modeling and simulation of concurrent workload processing in cloud-distributed enterprise information systems," in *ACM SIGCOMM Workshop on Distributed Cloud Computing (DCC 2014)*, August 2014.

[13] ——, "Modeling and simulation of concurrent workload processing in cloud-distributed enterprise information systems," in *Proc. ACM SIGCOMM Workshop on Distributed Cloud Computing (DCC 2014)*, 2014.

[14] ——, "SLA-driven simulation of multi-tenant scalable cloud-distributed enterprise information systems," in *ACM PODC Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC)*, 2014.

[15] ——, "Simulation of SLA-based VM-scaling algorithms for cloud-distributed applications," *Future Generation Computer Systems (FGCS)*, vol. 10.1016/j.future.2015.01.015, 2015.

[16] A.-F. Antonescu, A.-M. Oprescu, *et al.*, "Dynamic optimization of SLA-based services scaling rules," in *Proc. 5th IEEE Internetional Conference on Cloud Computing Technology and Science (CloudCom)*, December 2013.

[17] A.-F. Antonescu and P. Robinson, "Towards cross stratum SLA management with the geysers architecture," in *Proc. 10th IEEE Int. Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2012.

[18] A.-F. Antonescu, P. Robinson, and T. Braun, "Dynamic topology orchestration for distributed cloud-based applications," in *Proc. 2nd IEEE Symposium on Network Cloud Computing and Applications (NCCA)*, 2012.

[19] ——, "Dynamic sla management with forecasting using multi-objective optimizations," in *Integrated Network Management, IFIP/IEEE Symposium on*, May 2013.

[20] A.-F. Antonescu, P. Robinson, and M. Thoma, "Service level management convergence for future network enterprise platforms," in *Future Network & Mobile Summit (FutureNetw)*, 2012, pp. 1–9.

[21] ——, "Service level management convergence for future network enterprise platforms," in *Future Network & Mobile Summit (FutureNetw), 2012*, 2012, pp. 1–9.

[22] A.-F. Antonescu and T. Braun, "Service level agreements-driven management of distributed applications in cloud computing environments," in *Integrated Network Management, IFIP/IEEE Symposium on*, May 2015.

[23] Apache Software Foundation, "Apache CXF," http://cxf.apache.org/, 2013.

[24] ——, "Apache Zookeeper," http://zookeeper.apache.org, 2013.

[25] ——, "Distributed OSGi," http://cxf.apache.org/distributed-osgi.html, 2013.

[26] A. Azeez, S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle, "Multi-tenant soa middleware for cloud computing," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, July 2010, pp. 458–465.

[27] T. Bäck, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford University Press, USA, 1996.

[28] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[29] C. Baun, M. Kunze, J. Nimis, and S. Tai, "Open source cloud stack," *Cloud Computing*, pp. 49–62, 2011.

[30] M. N. Bennani *et al.*, "Resource allocation for autonomic data centers using analytic performance models," in *Autonomic Computing, 2005. ICAC 2005. Proc. 2nd Int. Conf. on.* IEEE, 2005, pp. 229–240.

[31] H. L. Blog, "Host server cpu utilization in amazon ec2 cloud," http://huanliu.wordpress.com/2012/02/17/host-server-cpu-utilization-in-amazon-ec2-cloud/, 2013.

[32] P. Bloomfield, *Fourier analysis of time series: an introduction.* John Wiley & Sons, 2004.

[33] J. Bozman, "Cloud computing: The need for portability and interoperability," *IDC Analyze the Future*, 2010.

[34] M. Brock, "MVEL Expression Language," http://mvel.codehaus.org, 2013.

[35] L. B. <buytenh@gnu.org>, "brctl - ethernet bridge administration," http://linuxcommand.org/man_pages/brctl8.html, 2015.

[36] R. Buyya *et al.*, "Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities," in *Int. Conf. on High Performance Computing & Simulation, 2009. HPCS'09.* IEEE, 2009, pp. 1–11.

[37] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation computer systems*, vol. 25, no. 6, pp. 599–616, 2009.

[38] F. Camargos, G. Girard, and B. Ligneris, "Virtualization of linux servers: a comparative study," in *Proceedings of the Linux Symposium*, vol. 47, no. 06, 2008, pp. 63–76.

[39] E. Caron, F. Desprez, and A. Muresan, "Forecasting for grid and cloud computing on-demand resources based on pattern matching," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, Dec. 2010, pp. 456 –463.

[40] Q. Chen and M. Hsu, "Inter-enterprise collaborative business process management," in *Data Engineering, 2001. Proceedings. 17th International Conference on*. IEEE, 2001, pp. 253–260.

[41] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, "Web services description language (wsdl)," http://www.w3.org/TR/wsdl20/, 2007.

[42] Cisco, "The zettabyte era—trends and analysis," http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.html, June 2014.

[43] ComputerWorld, "SAP-schwing-app ein PR-GAU (engligh translation: SAP swing app a PR meltdown)," Engligh translation: http://goo.gl/Zmmjwx Original webpage: http://www.computerworld.ch/news/software/artikel/sap-schwing-app-ein-pr-gau-64170/, Sept 2013.

[44] W. Conover, "Practical nonparametric statistics," *J. Wiley., New York*, pp. 309–314, 1980.

[45] R. Corporation, "Rackspace," http://www.rackspace.com/, 2015.

[46] T. P. P. Council, "TPC-H benchmark specification," *Published at http://www.tcp.org/hspec.html*, 2008.

[47] F. E. Croxton *et al.*, *Applied general statistics.* Prentice-Hall, Inc, 1939.

[48] Y. Diao, J. L. Hellerstein, S. Parekh, R. Griffith, G. E. Kaiser, and D. Phung, "A control theory foundation for self-managing computing systems," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 12, pp. 2213–2222, 2015.

[49] D. M. T. F. (DMTF), "Open virtualization format," http://www.dmtf.org/sites/default/files/standards/documents/DSP2017_1.0.0.pdf, 2009.

[50] A. V. Do, J. Chen, C. Wang, Y. C. Lee, A. Zomaya, and B. B. Zhou, "Profiling applications for virtual machine placement in clouds," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, 2011, pp. 660–667.

[51] E. Duipmans and L. F. Pires, "Business process management in the cloud: business process as a service (bpaas)," *University of Twente*, 2012.

[52] E. Escalona *et al.*, "Geysers: A novel architecture for virtualization and co-provisioning of dynamic optical networks and it services," in *Future Network & Mobile Summit (FutureNetw)*. IEEE, 2011, pp. 1–8.

[53] J. D. Evans, *Straightforward statistics for the behavioral sciences*. Brooks/Cole, 1996.

[54] E. Falkenauer, "A hybrid grouping genetic algorithm for bin packing," *Journal of heuristics*, vol. 2, no. 1, p. 5,Äì30, 1996.

[55] M. R. Ferre, "Vmware esx server: scale up or scale out," *IBM Redpaper*, 2004.

[56] J. Fontán, T. Vázquez, L. Gonzalez, and R. S. Montero, "OpenNebula: The open source virtual machine manager for cluster computing," in *Open Source Grid and Cluster Software Conference*, 2008.

[57] O. G. Forum, "Open cloud computing interface (occi)," http://occi-wg.org, 2015.

[58] O. Foundation, "Openstack cloud software," http://www.openstack.org, ian. 2015.

[59] A. Galis and A. Gavras, *The Future Internet – Future Internet Assembly 2013: Validated Results and New Horizons*, ser. Lecture Notes in Computer Science, F. Álvarez, A. Bassi, M. Bezzi, L. Ciavaglia, F. Cleary, P. Daras, H. D. Meer, P. Demestichas, D. Raz, G. Richards, E. Salvadori, S. Sargento, H. Schaffers, J. Serrat, B. Stiller, A. F. Skarmeta, K. Tutschku, and T. Zahariadis, Eds. Elsevier, 2013, vol. 7858.

[60] A. Gandhi, Y. Chen, D. Gmach, M. Arlitt, and M. Marwah, "Minimizing data center sla violations and power consumption via hybrid resource provisioning," in *Green Computing Conference and Workshops (IGCC), International*, 2011, pp. 1–8.

[61] S. K. Garg *et al.*, "Networkcloudsim: Modelling parallel applications in cloud simulations," in *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*. IEEE, 2011, pp. 105–113.

[62] Gartner, "Business process as a service (bpaas)," http://www.gartner.com/it-glossary/business-process-as-a-service-bpaas, 2015.

[63] GEYSERS Consortium, "Generalised architecture for dynamic infrastructure services," http://www.geysers.eu, 2012.

[64] P. Gill, M. Arlitt, Z. Li, and A. Mahanti, "Youtube traffic characterization: a view from the edge," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, ser. IMC '07. New York, NY, USA: ACM, 2007, pp. 15–28. [Online]. Available: http://doi.acm.org/10.1145/1298306.1298310

[65] L. R. Gomez-Mejia, D. B. Balkin, and R. L. Cardy, *Managing human resources*. Prentice Hall, 1995.

[66] Google, "Google app engine: Platform as a service," https://cloud.google.com/appengine, 2015.

[67] ——, "Google compute engine," https://cloud.google.com/compute, 2015.

[68] Google App Engine, "Google app engine service level agreement," https://developers.google.com/appengine/sla.

[69] T. Goyal *et al.*, "Cloudsim: simulator for cloud computing infrastructure and modeling," *Procedia Engineering*, vol. 38, pp. 3566–3572, 2012.

[70] F. A. Graybill, *Theory and applications of the linear model*. Duxbury, 1976.

[71] N. P. Greis and J. D. Kasarda, "Enterprise logistics in the information era," *California Management Review*, vol. 39, pp. 55–78, 1997.

[72] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, "Simple object access protocol (soap)," http://www.w3.org/TR/soap12/, 2007.

[73] R. Hat, "libvirt: The virtualization api," http://libvirt. org, 2012.

[74] C. C. Holt, "Forecasting seasonals and trends by exponentially weighted moving averages," *International Journal of Forecasting*, vol. 20, no. 1, pp. 5–10, 2004.

[75] R. J. Hyndman and Y. Khandakar, "Automatic time series for forecasting: The forecast package for R," Monash University, Department of Econometrics and Business Statistics, Tech. Rep., 2007.

[76] IBM, "Techdocs white paper: Evaluating microsoft hyper-v live migration performance using IBM system x3650 m3 and IBM system storage DS3400," http://129.33.205.81/jct03001c/support/techdocs/atsmastr.nsf/WebIndex/WP101828, Dec. 2010.

[77] ——, "Ibm smartcloud orchestrator," http://www-03.ibm.com/software/products/en/smartcloud-orchestrator, ian. 2015.

[78] H. Iima and T. Yakawa, "A new design of genetic algorithm for bin packing," in *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, vol. 2, Dec. 2003, pp. 1044 – 1049 Vol.2.

[79] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155–162, 2012.

[80] Java Platform Standard Ed. 8, "Interface map," https://docs.oracle.com/javase/8/docs/api/java/util/Map.html, 2015.

[81] ——, "Interface runnable," https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html, 2015.

[82] G. Juve and E. Deelman, "Automating application deployment in infrastructure clouds," in *3rd International IEEE Conference on Cloud Computing Technology and Science (CloudCom)*, 2001, pp. 658–665.

[83] M. Kajko-Mattsson and C. Makridis, "Outline of an SLA management model," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, Apr. 2008, pp. 308 –310.

[84] P. S. Kalekar, "Time series forecasting using holt-winters exponential smoothing," *Kanwal Rekhi School of Information Technology*, 2004.

[85] ——, "Time series forecasting using holt-winters exponential smoothing," *Kanwal Rekhi School of Information Technology*, 2004.

[86] B. Khasnabish, J. Chu, S. Ma, Y. Meng, N. So, P. Unbehagen, M. Morrow, and M. Hasan, "Internet-draft: Cloud reference framework," https://datatracker.ietf.org/doc/draft-khasnabish-cloud-reference-framework, October 2014.

[87] H. Kim, Y. el Khamra, I. Rodero, S. Jha, and M. Parashar, "Autonomic management of application workflows on hybrid computing infrastructure," *Scientific Programming*, vol. 19, no. 2, 2011.

[88] G. Klyne and J. J. Carroll, "Resource description framework (RDF): Concepts and abstract syntax," http://www.w3.org/TR/2004/REC-rdf-concepts-20040210, 2004.

[89] A. Konak, D. W. Coit, and A. E. Smith, "Multi-objective optimization using genetic algorithms: A tutorial," *Reliability Engineering & System Safety*, vol. 91, no. 9, pp. 992–1007, Sept. 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0951832005002012

[90] R. Krebs, C. Momm, and S. Kounev, "Architectural concerns in multi-tenant saas applications," in *2nd International Conference on Cloud Computing and Services Science (CLOSER)*. SciTePress, 2012.

[91] G. Lab, "Cloud simulator cloudsim," http://code.google.com/p/cloudsim, 2014.

[92] Y. Lee, G. Bernstein, N. So, T. Y. Kim, K. Shiomoto, and O. G. de Dios, "Research proposal for cross stratum optimization (cso) between data centers and networks", ietf internet draft," http://tools.ietf.org/html/draft-lee-cross-stratum-optimization-datacenter-00, Feb. 2012.

[93] Leidig, T. and C. Momm, "USDL service level agreement," http://www.linked-usdl.org/ns/usdl-sla, April 2012.

[94] A. Leon, *Enterprise resource planning*. Tata McGraw-Hill Education, 2008.

[95] Z. Li, Y. Bai, H. Zhang, and Y. Ma, "Affinity-aware dynamic pinning scheduling for virtual machines," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec. 2010, pp. 242 –249.

[96] J. D. Little and S. C. Graves, "Little's law," in *Building Intuition: Insights From Basic Operations Management Models and Principles*. Springer, 2008, pp. 81–100.

[97] C. Liu, Y. Mao, J. Van der Merwe, and M. Fernández, "Cloud resource orchestration: A data-centric approach," in *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)*, 2011.

[98] K. Lu, T. Röblitz, and P. Chronz, "Sla-based planning for multi-domain infrastructure as a service," *Cloud Comp. and Services Science*, 2012.

[99] G. S. Machado and B. Stiller, "Investigations of an sla support system for cloud computing (slacc)," *PIK-Praxis der Informationsverarbeitung und Kommunikation*, vol. 34, no. 2, pp. 80–86, 2011.

[100] L. Magazine, "The everything as a service (xaas) model," http://www.linux-mag.com/id/7197, December 2008.

[101] M. Malawski, G. Juve, and E. Deelman, "Cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds," *24th IEEE/ACM International Conference on Supercomputing*, 2012.

[102] A. Marinos and G. Briscoe, "Community cloud computing," in *Cloud Computing*, ser. Lecture Notes in Computer Science, M. Jaatun, G. Zhao, and C. Rong, Eds. Springer Berlin Heidelberg, 2009, vol. 5931, pp. 472–484. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10665-1_43

[103] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi, "Cloud computing—the business perspective," *Decision Support Systems*, vol. 51, no. 1, pp. 176–189, 2011.

[104] S. Martello and P. Toth, *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[105] M. Mazzucco and D. Dyachuk, "Optimizing cloud providers revenues via energy efficient server allocation," *Sustainable Computing: Informatics and Systems*, vol. 2, no. 1, pp. 1 – 12, 2012.

[106] MCN Consortium, "Mobile cloud networking (mcn)," http://fp7-mcn.eu/site, ian. 2015.

[107] A. Medina *et al.*, "BRITE: an approach to universal topology generation," *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on*, 2001.

[108] P. Mell and T. Grance, "The NIST definition of cloud computing," *NIST special publication*, 2011.

[109] Microsoft, "High performance SQL server workloads on hyper-v," http://download.microsoft.com/download/D/F/8/DF89D22D-39C8-4728-A990-3BD4467891B7/HighPerformanceSQLServerWorkloadsOnHyper-V_Final.docx, May 2010.

[110] ——, "Hyper-v," http://technet.microsoft.com/en-us/windowsserver/dd448604.aspx, ian. 2015.

[111] ——, "Microsoft azure," http://azure.microsoft.com, 2015.

[112] ——, "Microsoft azure virtual machines," http://azure.microsoft.com/en-us/services/virtual-machines, 2015.

[113] D. Milojičić, I. Llorente, and R. Montero, "Opennebula: A cloud management tool," *Internet Computing, IEEE*, 2011.

[114] M. Nelson, B. H. Lim, G. Hutchins, *et al.*, "Fast transparent migration for virtual machines," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005, p. 25,Äì25.

[115] OASIS Committee Specification, "TOSCA version 1.0 2012," http://docs.oasis-open.org/tosca/tosca/v1.0/csd03/tosca-v1.0-csd03.html.

[116] OpenNebula Community, "Opennebula appflow," http://opennebula.org/documentation:archives:rel4.0:appflow, 2013.

[117] ——, "Opennebula cloud management platform," http://opennebula.org, ian. 2015.

[118] openQRM Enterprise, "openqrm," http://www.openqrm-enterprise.com, ian. 2015.

[119] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab, *Signals and systems*. Prentice Hall, 1997.

[120] OPTIMIS Consortium, "Optimis: Optimized infrastructure services," http://www.optimis-project.eu/project, ian. 2015.

[121] Oracle Corporation, "MySQL Relational Database," http://www.mysql.com, 2013.

[122] Organization for the Advancement of Structured Information Standards (OASIS), "Open service for lifecycle collaboration (olsc)," http://open-services.net.

[123] ——, "Topology and Orchestration Specification for Cloud Applications (TOSCA)," http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html, 2013.

[124] A. R. P. Patel and A. Sheth, "Service level agreement in cloud computing," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2009.

[125] M. P. Papazoglou and W.-J. Van den Heuvel, "Web services management: A survey," *Internet Computing, IEEE*, vol. 9, no. 6, pp. 58–64, 2005.

[126] C. Pedrinaci *et al.*, "Linked-USDL core," http://www.linked-usdl.org/ns/usdl-core, November 2011.

[127] V. Podinovskii and V. Nogin, "Pareto-optimal solutions of multicriteria problems," *Moscow: Sci*, 1982.

[128] M. B. Priestley, *Non-linear and non-stationary time series analysis*. Academic Press London, 1988, vol. 223.

[129] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013. [Online]. Available: http://www.R-project.org

[130] C. R. Rao *et al.*, *Linear models*. Springer, 1995.

[131] F. Research, "Cloud computing," https://www.forrester.com/Cloud-Computing, ian 2015.

[132] J. F. Riera, J. Garcia-Espin, A.-F. Antonescu, *et al.*, "Virtual infrastructures as a service enabling converged optical networks and data centres," *Optical Switching and Networking (OSN)*, pp. 197–208, 2014.

[133] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 500–507.

[134] P. Russom *et al.*, "Big data analytics," *IBM*, 2011.

[135] SAP, "Sap hana cloud platform," http://hcp.sap.com, ian. 2015.

[136] SAP Research, "The business web," http://www.sdn.sap.com/irj/scn/go/portal/prtroot/docs/library/uuid/604885bf-620d-2f10-00a6-86085e4895a3, 2011.

[137] T. I. SC, "Sugarcrm interop csar," https://www.oasis-open.org/committees/document.php?document_id=50153&wg_abbrev=tosca-interop, August 2013.

[138] Scalr, Inc., "Scalr cloud management platform," http://www.scalr.com/product/overview, ian. 2015.

[139] J. Stewart, *Calculus*. Cengage Learning, 2011.

[140] E. Systems, "Eucalyptus," https://www.eucalyptus.com/eucalyptus-cloud/iaas/architecture, ian. 2015.

[141] W. Theilmann, J. Happe, C. Kotsokalis, A. Edmonds, K. Kearney, and J. Lambea, "A reference architecture for multi-level sla management," *Journal of Internet Engineering*, vol. 4, no. 1, 2010.

[142] TMF, "Tmforum's sla management website," http://www.tmforum.org/SLAManagement/1690/home.html, 2012.

[143] TMF and The Open Group, "Sla management handbook," http://www.afutt.org/Qostic/qostic1/SLA-DI-USG-TMF-060091-SLA_TMForum.pdf, ian. 2015.

[144] A. Tzanakaki, S. Figuerola, J. A. García-Espín, D. Simeonidou, N. Ciulli, P. Robinson, J. Rodríguez, G. Landi, B. Belter, P. Vicat-Blanc, *et al.*, *The GEYSERS Concept and Major Outcomes*. Springer, 2013.

[145] S. Urbanek, "Rserve – a fast way to provide r functionality to applications," in *Proc. of the 3rd International Workshop on Distributed Statistical Computing (DSC)*, 2003.

[146] I. UserLand Software, "Xml remote procedure call (xml-rpc)," http://www.xmlrpc.com, 2015.

[147] H. N. Van, F. Tran, and J.-M. Menaud, "SLA-aware virtual resource management for cloud infrastructures," in *Computer and Information Technology, 2009. CIT '09. Ninth IEEE International Conference on*, vol. 1, Oct 2009, pp. 357–362.

[148] A. Visan, M. Istin, F. Pop, and V. Cristea, "Bio-inspired techniques for resources state prediction in large scale distributed systems," *International Journal of Distributed Systems and Technologies (IJDST)*, vol. 2, no. 3, pp. 1–18, 2011.

[149] Vnomics Corp, "Vnomic application landscape designer," http://www.vnomic.com, ian. 2015.

[150] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, "Cost of virtual machine live migration in clouds: A performance evaluation," in *Cloud Computing*, ser. Lecture Notes in Computer Science, M. Jaatun, G. Zhao, and C. Rong, Eds. Springer Berlin / Heidelberg, 2009, vol. 5931, pp. 254–265. [Online]. Available: http://www.springerlink.com/content/7052149402656584/abstract/

[151] C. A. Waldspurger, "Memory resource management in VMware ESX server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, p. 181ñ194, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844146

[152] D. Williams, H. Jamjoom, Y. H. Liu, and H. Weatherspoon, "Overdriver: Handling memory overload in an oversubscribed cloud," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2011, p. 205ñ216. [Online]. Available: http://dl.acm.org/citation.cfm?id=1952709

[153] WMware, "Wmware hypervisor," http://www.vmware.com, 2015.

[154] D. Woods, *Enterprise Services: Architecture.* O'Reilly Media, Inc., 2003.

[155] L. Wu and R. Buyya, ""service level agreement (sla) in utility computing systems"," *Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*, pp. 1–25, 2011.

[156] J. Xu and J. Fortes, "Multi-objective virtual machine placement in virtualized data center environments," in *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, Dec. 2010, pp. 179 –188.

[157] A. C. Yao, "New algorithms for bin packing," *Journal of the ACM*, vol. 27, no. 2, p. 207,Äì227, 1980.

[158] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[159] S. Zhang and M. Song, "An architecture design of life cycle based SLA management," in *The 12th International Conference on Advanced Communication Technology (ICACT)*, vol. 2, 7-10 Feb. 2010, pp. 1351–1355.

[160] Z. Zhang, H. Wang, L. Xiao, and L. Ruan, "A statistical based resource allocation scheme in cloud," in *Cloud and Service Computing (CSC), 2011 International Conference on*, Dec. 2011, pp. 266 –273.

Chapter 10

List of Publications

Refereed Papers (Journals, Conferences, Workshops)

1. **Antonescu**, Alexandru-Florian; **Braun**, Torsten (2015). "Service Level Agreements-Driven Management of Distributed Applications in Cloud Computing Environments" In IFIP/IEEE Symposium on Integrated Network Management (IM 2015), Ottawa, Canada, May 11-15 2015

2. **Antonescu**, Alexandru-Florian; **Braun**, Torsten (2015). "Simulation of SLA-Based VM-Scaling Algorithms for Cloud-Distributed Applications", In *Future Generation Computer Systems (FGCS)* Journal (impact factor: 2.639), 2015, 10.1016/j.future.2015.01.015

3. **Antonescu**, Alexandru-Florian; **Braun**, Torsten (18 August 2014). "Modeling And Simulation Of Concurrent Workload Processing In Cloud-Distributed Enterprise Information Systems". In: *ACM SIGCOMM Workshop on Distributed Cloud Computing (DCC 2014)* (pp. 11-16). ACM 10.1145/2627566.2627575

4. **Antonescu**, Alexandru-Florian; **Braun**, Torsten (15 July 2014). "SLA-Driven Simulation of Multi-Tenant Scalable Cloud-Distributed Enterprise Information Systems" (Best Paper Award). *Lecture notes in computer science*, pp. 91-102. Springer 10.1007/978-3-319-13464-2-7

5. **Antonescu**, Alexandru-Florian; **Braun**, Torsten (5 May 2014). "Improving Management of Distributed Services Using Correlations and Predictions in SLA-Driven Cloud Computing Systems". In: *IEEE/IFIP Network Operations and Management Symposium (NOMS 2014)* (pp. 1-8). IEEE 10.1109/NOMS.2014.6838320

6. Riera, Jordi Ferrer; Tzanakaki, Anna; **Antonescu**, Alexandru-Florian; Anastasopoulos, Markos; García-Espín, Joan A.; Escalona, Eduard; Peng, Shuping; Landi, Giada; Bernini, Giacomo; Belter, Bartosz; Parniewicz, Damian; Hesselbach, Xavier; Figuerola, Sergi; Simeonidou,

Dimitra (2014). "Virtual Infrastructures as a Service enabling Converged Optical Networks and Data Centres". *Optical Switching and Networking* (impact factor 0.909), 14, pp. 197-208. Elsevier 10.1016 /j.osn.2014.05.017

7. **Antonescu**, Alexandru-Florian; Oprescu, Ana-Maria; Demchenko, Yuri; de Laat, Cees; **Braun**, Torsten (2 December 2013). "Dynamic Optimization of SLA-Based Services Scaling Rules". In: *5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (pp. 282-289). Bristol, UK. 2-5 Dec. 2013. IEEE 10.1109/CloudCom.2013.44

8. Oprescu, Ana-Maria; **Antonescu**, Alexandru-Florian; Demchenko, Yuri; Laat, Cees de (2 December 2013). "ICOMF: Towards a Multi-cloud Ecosystem for Dynamic Resource Composition and Scaling". In: *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on* 1 (pp. 49-55). IEEE 10.1109/ CloudCom.2013.14

9. **Antonescu**, Alexandru-Florian; Gomes, Andre; Robinson, Philip; **Braun**, Torsten (9 June 2013). "SLA-driven predictive orchestration for distributed cloud-based mobile services". In: Kim, Dong-In; Mueller, Peter (eds.) 2013 *IEEE International Conference on Communications Workshops (ICC)* (pp. 738-743). IEEE 10.1109/ICCW.2013.6649331

10. **Antonescu**, Alexandru-Florian; Robinson, Philip; **Braun**, Torsten (27 May 2013). "Dynamic SLA management with forecasting using multi-objective optimization". In: De Turck, Filip; Diao, Yixin; Hong, Choong Seon; Medhi, Deep; Sadre, Ramin (eds.) Proceedings of the 2013 I*FIP/IEEE International Symposium on Integrated Network Management (IM 2013)* (pp. 457-463). Ghent: IFIP

11. **Antonescu**, Alexandru-Florian; Robinson, Philip; **Braun**, Torsten (2012). "SLA-Driven Dynamic Orchestration and Composition for Distributed Cloud-Based Services". In: *7th Workshop (Fachgespräch) on Cloud-based Service Platforms for the Future Internet*, Winterthur, Switzerland. KuVS Fachgespräch NGSDP

12. **Antonescu**, Alexandru-Florian; Robinson, Philip; **Braun**, Torsten (2012). "Dynamic Topology Orchestration for Distributed Cloud-Based Applications". In: *IEEE International Symposium on Network Cloud Computing and Applications (NCCA 2012)*, Imperial College, London, UK, 3.-4.12.2012 (pp. 116-123). Washington, DC: IEEE Computer Society 10.1109/NCCA.2012.14

13. Thoma, Matthias; **Antonescu**, Alexandru-Florian; Mintsi, Theano; **Braun**, Torsten (2013). "Linked Services for Enabling Interoperability

in the Sensing Enterprise". In: van Sinderen, M.; Oude Luttighuis, P.; Folmer, E.; Bosems, S. (eds.) *Enterprise Interoperability: 5th International IFIP Working Conference, IWEI 2013*, Enschede, The Netherlands, March 27-28, 2013. Proceedings. Lecture Notes in Business Information Processing: Vol. 144 (pp. 131-144). Berlin: Springer 10.1007/978-3-642-36796-0-12

14. Thoma, Matthias; **Antonescu**, Alexandru-Florian; Mintsi, Theano; **Braun**, Torsten (1 July 2013). "Linked services for M2M communication with Enterprise IT systems". In: *Wireless Communications and Mobile Computing Conference (IWCMC)*, 2013 9th International (pp. 1212-1216). IEEE 10.1109/IWCMC.2013.6583729

15. Thoma, Matthias; **Braun**, Torsten; Magerkurth, Carsten; **Antonescu**, Alexandru-Florian (5 May 2014). "Managing Things and Services with Semantics: A Survey". In: *IEEE/IFIP Network Operations and Management Symposium (NOMS 2014)* (pp. 1-5). IEEE 10.1109/ NOMS.2014.6838366

Unrefereed Papers (Technical Reports, Project Deliverables)

1. Mercier, Huges; **Braun**, Torsten; Felber, Pascal; Kropf, Peter; Kuonen, Pierre (2014). 2014 *Doctoral Workshop on Distributed Systems Bern, Switzerland: Institut für Informatik und angewandte Mathematik*

2. Mercier, Hugues; **Braun**, Torsten; Felber, Pascal; Kropf, Peter; Kuonen, Pierre (2013). *2013 Doctoral Workshop on Distributed Systems (Technical Report IAM-13-002). Bern: Institut für Informatik und angewandte Mathematik*

3. Robinson, Philip; *Antonescu*, Alexandru-Florian; et al., Whitepaper: "SLA Management for Composite Infrastructure as a Service", SAP Research, 2012

4. Robinson, Philip; *Antonescu*, Alexandru-Florian; **Braun**, Torsten: Technical Report: "Dynamic SLA Management with Forecasting using Multi-Objective Optimizations, September 5, 2012, IAM-12-002

5. A. Tzanakaki, A.-F. **Antonescu**, P. Robinson, et al., "Report on Energy Consumption Modeling," GEYSERS Internal Report.

6. A. Tzanakaki, A.-F. **Antonescu**, P. Robinson, et al., "Report on Technology Benchmarking," GEYSERS Project Deliverable.

7. J. A. Garcia-Espin, A.-F. **Antonescu**, et al., "Functional Description of the Logical Infrastructure Composition Layer (LICL)", GEYSERS Project Deliverable.

8. M. Ghijsen, A.-F. **Antonescu**, et al., "Preliminary Logical Infrastructure and Composition Layer software release", GEYSERS Project Deliverable.

9. L. Lopatowski, A.-F. **Antonescu**, et al., "Logical Infrastrucuture and Composition Layer Sub-systems release", GEYSERS Project Deliverable.

10. B. Belter, A.-F. **Antonescu**, et al., "GEYSERS Test-bed implementation", GEYSERS Project Deliverable.

Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname:	Alexandru-Florian Antonescu
Matrikelnummer:	10-129-693
Studiengang:	PhD

Bachelor ☐ Master ☐ Dissertation ◉

Titel der Arbeit:	Service Level Agreements-Driven Management of Distributed Applications in Cloud Computing Environments
LeiterIn der Arbeit:	Prof. Dr. Torsten Braun

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist. Ich gewähre hiermit Einsicht in diese Arbeit.

Winterthur 04.08.2015

Ort/Datum

Antonescu

Unterschrift