FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Reducing SLA Violations of Composite Services Deployed to the Cloud

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Mathias Hess
Matrikelnummer 0125388

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Univ.Ass. Mag. Philipp Leitner

Wien, 17.10.2011

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Reducing SLA Violations of Composite Services Deployed to the Cloud

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Mathias Hess
Registration Number 0125388

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dr. Schahram Dustdar
Assistance: Univ.Ass. Mag. Philipp Leitner

Vienna, 17.10.2011

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Mathias Hess
Ringelseegasse 17/21, 1210 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

Composite services are fundamental to the integration of heterogeneous enterprise applications, and especially to build complex distributed applications that extend across the boundaries of organizations. Their guaranteed Quality of Service (QoS) characteristics, such as availability and performance, are formulated within Service Level Agreements (SLAs), service providers and consumers agree upon. Service providers need to dimension the capacities of their infrastructures in accordance to these guarantees. Misestimating the load their services are exposed to, results in violating SLAs and facing penalties. Papers, proposing solutions to this issue, either focus on the services instead of the infrastructure, hosting a composite service, or do not consider opportunities of hosting composite services in the Cloud.

This thesis suggests deploying a composite service, together with its infrastructure, into the Cloud, so service providers take advantage of the Cloud's handling of resources. Moreover it proposes an approach to reduce the amount of SLA violations by using these capabilities. Therefore, the design and implementation of a runtime engine for composite services will be explained. The engine is able to provision itself with additional resources, and uses an event-driven approach to evaluate SLA violations. A prototype, using the open source Cloud platform Eucalyptus, was implemented as part of this thesis. Its internals and deployment within Eucalyptus will be presented in detail. The ideas presented within this thesis are applicable to Amazon Elastic Compute Cloud (EC2) as well. This is because Eucalyptus makes use of similar concepts (e.g., pre-built machine images and instance types) and provides a compatible interface. As will be shown by the results of an extensive evaluation, considering both the benefits and limitations of the proposed approach, the prototype is able to significantly reduce the amount of SLA violations, even to an extent that also outweighs potential costs.

Kurzfassung

Composite Services bilden die Grundlage, um heterogene Unternehmensanwendungen zu verbinden und im Speziellen, um komplexe verteilte Applikationen zu entwickeln, welche sich über die Grenzen von Organisationen erstrecken. Zugesicherte Qualitätseigenschaften, wie Verfügbarkeit und Leistung, werden im Rahmen von Service Level Agreements (SLAs) formuliert, die zwischen den Anbietern und Nutzern von Services vereinbart werden. Anbieter müssen die Kapazitäten ihrer Infrastrukturen nach diesen Garantien ausrichten. Schätzen sie allerdings die Auslastung ihrer angebotenen Services falsch ein, führt dies zur Verletzung von SLAs und sie haben mit Vetragsstrafen zu rechnen. Wissenschaftliche Arbeiten, die sich mit Lösungen dieser Problematik beschäftigen, legen den Schwerpunkt entweder auf die Services selbst, anstatt sich mit den Infrastrukturen zu befassen, welche die Services bereitstellen, oder sie vernachlässigen gänzlich die Möglichkeit Composite Services in der Cloud zu betreiben.

In dieser Diplomarbeit wird daher die Bereitstellung eines Composite Services, zusammen mit dessen Infrastruktur, in der Cloud vorgeschlagen. Service Anbieter können dadurch von den Vorteilen der Cloud, in Hinsicht auf deren Umgang mit Ressourcen, profitieren. Diese Diplomarbeit präsentiert daher eine Lösung zur Reduzierung von SLA Verletzungen, bei der diese Fähigkeiten der Cloud genutzt werden. Dazu werden das Konzept und die Umsetzung einer Laufzeitumgebung für Composite Services beschrieben, die dazu fähig ist, sich selbständig mit zusätzlichen Ressourcen zu versorgen, und die Entscheidung dafür anhand der Auswertung von SLAs zu treffen. Ein Prototyp, der die Open Source Cloud Platform Eucalyptus verwendet, wurde im Zuge dieser Diplomarbeit erstellt. Dieser wird schließlich, sowie seine Bereitstellung mittels Eucalyptus, umfassend beschrieben. Eucalyptus verwendet ähnliche Konzepte wie Amazon Elastic Compute Cloud (EC2), zum Beispiel Virtual Machine Images und sogenannte Instanztypen, und bietet darüber hinaus eine mit Amazon EC2 kompatible Schnittstelle an. Die in dieser Diplomarbeit präsentierten Ideen sind daher auch auf Amazon EC2 anwendbar.

Anhand der Resultate einer eingehenden Untersuchung, welche die Vorteile, aber auch die Anwendungsgrenzen der vorgestellten Lösung betrachtet hat, wird gezeigt, dass der Prototyp in der Lage ist, die Anzahl an Verletzungen eines SLAs signifikant zu reduzieren und, dass die damit verbundenen potentiellen Kosten übertroffen werden.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

In today's design of distributed systems, *Service-oriented Architectures (SOAs)* [31] without question play a fundamental role. Their building blocks are services, which, basically spoken, represent reusable software modules, providing well-defined functionalities and are accessible over a network. The primary implementing technology of SOAs are *Web services* [3]. They became a widely used and broadly accepted technology due to their high degree of standardization by the World Wide Web Consortium (W3C) [112] and their independence from concrete hardware and software platforms. Besides theses characteristics, or better still, because of theses characteristics, Web services are highly suitable for being combined and composed to provide new functionalities, which may in turn be exposed in the form of services. This process is called *service composition* and its products are *composite services*. The benefits of service composition are proven in the areas of engineering inter-organizational business processes on the one hand, and the integration and connection of existing applications within organizations, commonly known as Enterprise Application Integration (EAI) [27], on the other hand.

Web services, and composite services, are subject to legally binding contracts, formalized in so called *Service Level Agreements (SLAs)* [28], agreed upon by consumer and provider of a service and defining guaranteed characteristics in terms of the quality of a service [57]. Basically, such agreements are statically specified and published by the service provider. The customer agrees by simply consuming and paying for the corresponding service. On the contrary, both parties may individually establish SLAs, either by negotiating and contracting manually or even automatically by using software agents. Violations of a SLA by one party imply certain penalties which are specified as part of a SLA and grant compensation to the other party.

Besides SOAs, Cloud computing [56] has been one of the most observed and discussed technologies of the past years. It has been dominating not only marketing- and consulting-speech but also managed to attract attention of industrial and academic research. Above all, Cloud computing has influenced the perception of computing resources and the organization of data centers. Providing Cloud computing services to the public has become a successful business model as shown by offerings like Amazon Web Services (AWS) [7], Microsoft Azure [61] and Google AppEngine [48]. Furthermore, the availability of open source Cloud platforms such as

Eucalyptus [36], OpenNebula [107] and OpenStack [99] enable companies to turn their own data centers into private Clouds.

1.1 Motivation

In order to ensure compliance with SLAs, service providers need to dimension their resource capacities according to their guaranteed service quality. They have to establish facilities to constantly monitor their services and keeping track of their current level of SLA conformance by evaluating and checking monitored data against the objectives of SLAs [57].

The amount of resources, demanded by a service varies over time. Requests to a service, and therefore the workload it is exposed to, vary according to seasonal cycles or according to the time of the day [67]. Additionally an overall trend may increase the workload continuously over time as a result of growing popularity and demand for the service. The service provider is also at risk of falling victim to unexpected demand bursts due to external events (e.g., news events). Dimensioning resources to a service is also accompanied by the question of the magnitude of the peak workload, which results in resources being idle during periods of average workload; resources that have caused high investment costs and create ongoing maintenance and operation costs. [13]

It is impossible to keep up with such variations in demand and to always provide the exact amount of physical computing resources needed at the moment. Yet it is undesirable to violate SLAs because of lacking resources. What is needed is the ability to dynamically provision resources based on the current demand to prevent the SLAs on composite services from being violated. The paradigm of Cloud computing offers opportunities that build the foundation for accomplishing this challenge. According to [111], three aspects are new to Cloud computing which underline the capabilities of the Cloud in terms of dynamically handling resources. They are summarized by [13] as follows:

1. „*The illusion of infinite computing resources* available on demand, thereby eliminating the need for Cloud Computing users to plan far ahead for provisioning;“
2. „*The elimination of an up-front commitment* by Cloud users, thereby allowing companies to start small and increase hardware resources only when there is an increase in their needs; and“
3. „*The ability to pay for use* of computing resources on a short-term basis as needed (e.g., processors by the hour and storage by the day) and release them as needed, thereby rewarding conservation by letting machines and storage go when they are no longer useful.“

These characteristics induced further contemplation on the issue of dynamically provisioning resources to a composite service to reduce the amount of SLA violations and thus motivated the solution proposed in this thesis.

1.2 Contributions

The overall goal of this thesis is to improve SLA protection of composite services by deploying them to the Cloud and by utilizing capabilities provided by the Cloud, to handle computing resources dynamically. This idea will be implemented by designing and then prototyping a runtime engine for hosting composite services. The runtime engine, in the following referred to as the *runtime*, will be able to scale itself according to the current amount of SLA violations. It accomplishes this by fetching additional computing resources from the Cloud on demand. The open source Cloud platform Eucalyptus, which is representative for the current state of the art in Cloud technology, and shares many concepts with Amazon Elastic Compute Cloud (EC2), will be used for this purpose.

Reducing SLA violations by resource provisioning is based on the assumption that the amount of resources provided to a composite service runtime directly impacts the amount of SLA violations. The experiments carried out as part of this thesis confirm this assumption and their results will be shown later on.

The following section details the individual contributions of this thesis:

Autonomic The runtime engine will provide itself with further resources, without human interaction, by communicating with the Cloud. It will decide on its own, by monitoring itself, when it is necessary to do so. The logical separation of tasks within the runtime is based on the architecture and the loop of autonomic systems presented in [58].

Monitoring An event-based approach will be considered, enabling the runtime to monitor itself in real time. Monitoring includes tracking its own life-cycle, invocations and execution of the composite service, as well as evaluating SLA conformance. To retrospectively evaluate the behavior of the runtime, events will be persisted to a database.

Resource Provisioning An approach to scaling the runtime engine of composite services in the Cloud will be proposed, which is different from those already covered by other papers. It includes upgrading the runtime, which is hosting the composite service, without causing service degradation or unavailability. Cloud platforms, such as Eucalyptus and Amazon EC2, do not support changing the amount of computing resources provided to a system, therefore the composite service will be transparently *relocated* within the Cloud. Evaluating this approach will give an understanding on how the performance of a composite service is influenced by the amount of provided resources. Further on, will the potential of the presented solution be evaluated in terms of its ability to reduce SLA violations.

Migration The concept of *migrating* currently running instances of composite services between two remote runtime engines will be presented. Migration includes suspension and serialization on one runtime, transport over the network and resuming on the remote runtime. This concept will be used in connection with

the approach to resource provisioning to transfer them to a relocated composite service runtime.

Cloud Deployment In addition to illustrating the capabilities of Cloud platforms, such as Eucalyptus and Amazon EC2, in terms of resource provisioning, this thesis will explain in detail the deployment of the runtime to Eucalyptus. This includes extensive usage of concepts, also provided by Amazon EC2, such as *machine images*, *instance types* and *Elastic IP*. Not only will the preparation of the runtime itself be shown, but also that of services used by the runtime, as they will also be hosted in the Cloud.

Costs As part of evaluating a prototype implementation of the presented solution, the *costs* for provisioning resources will be analyzed to see whether they justify resource provisioning and under which conditions. Costs will be expressed in terms of SLA violations caused by the resource provisioning procedure.

1.3 Organization

This thesis is organized as follows:

Chapter 2 reviews current state of the art technologies, such as SOAs and Cloud computing, enabling the solution presented within this thesis.

Chapter 3 gives an overview on academic research, including Cloud capabilities, scaling composite services and process migration, related to the concepts used.

Chapter 4 provides the necessary background by explaining concrete software components and concepts used for implementation.

Chapter 5 discusses in detail the solution proposed by this thesis, as well as the implementation of the prototype, its preparation and deployment to Eucalyptus.

Chapter 6 extensively evaluates the prototype and shows its capabilities, as well as its limitations.

Finally, Chapter 7 concludes this thesis by summarizing and giving an outlook on future work regarding the proposed solutions.

CHAPTER 2

State of the Art Review

The following chapter reviews state of the art technologies that provide the basis for the solution presented within this thesis. First, the principles of SOAs and their primary implementing technology, Web services, are explained. The objects of study of this thesis, composite services and the composite service runtime engine, are based on Web service technology. Following this, Cloud computing and its characteristics, as well as Amazon EC2 and Eucalyptus, which are representative Cloud platforms, are described. Finally, the vision and principles of autonomic computing, as well as the principles of event processing are presented, providing the basis for the architectural approach of the composite service runtime.

2.1 Service-oriented Architecture (SOA)

The model of SOA [31] is based on the design paradigm of service-orientation. The underlying concept of service-orientation are services. They represent units of solution logic and are implemented as independent software programs to which the principles of service-orientation are applied. According to [33], the main strategic goal of service-orientation and the application of service-oriented architectures is to enhance the agility and cost-effectiveness of an enterprise, while reducing the overall burden of Information Technology (IT).

[32] describes the design principles of service-orientation in the following way:

Standardized Service Contracts Services in a SOA provide contracts that precisely describe their functionality in terms of interfaces, data types, data models and policies. This description provides other applications with the information on how to interact with the service.

Loose coupling The level of coupling refers to the degree of dependency between software applications. Services interact in such a way that they do not need to know the

design and implementation details, e.g., underlying platform and programming language, of each other. This characteristic is referred to as *loosely coupled*. Tightly coupled applications communicate in a RPC-style and synchronous manner, whereas message-based, asynchronous ways of communication, enable the development of loosely coupled applications [96].

Abstraction This principle aims at hiding as many details of a service as possible and therefore enabling loosely coupled relationships.

Reusability The service reusability principle aims at services not to be bound to a specific functional context or business process. The logic of a service should be designed to be robust, generic and with a focus on quality. This is comparable to the design of a commercial product and automatically increases the potential of the service to be reused.

Autonomy Services can control their environment and resources to a significant degree.

Discoverability Services are attached with meta-data that is published to a central service registry so that they can be discovered and understood.

Statelessness For services to be scalable and to increase their reliability, states are outsourced to an external component. Resource consumption of a service is decreased and the service is free to use its resources to handle requests.

Composability Services should be able to participate as effective members in solutions or business processes that represent compositions of services.

2.1.1 Web Services

A technology that is well suited and primarily used to implement SOAs, are Web services [3]. Their goal is to achieve interoperability between different software applications, regardless of their underlying platform and building framework, by providing standard means of interoperation [112]. Generally speaking, a Web service is a self-describing and self-contained software module that is accessible over a public or private network. A Web service may, for example, complete tasks, solve problems or conduct transactions as requested by an external entity like an application or user [96]. A more complete definition of a Web service is provided by the W3C as follows:

> „A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.“ [113]

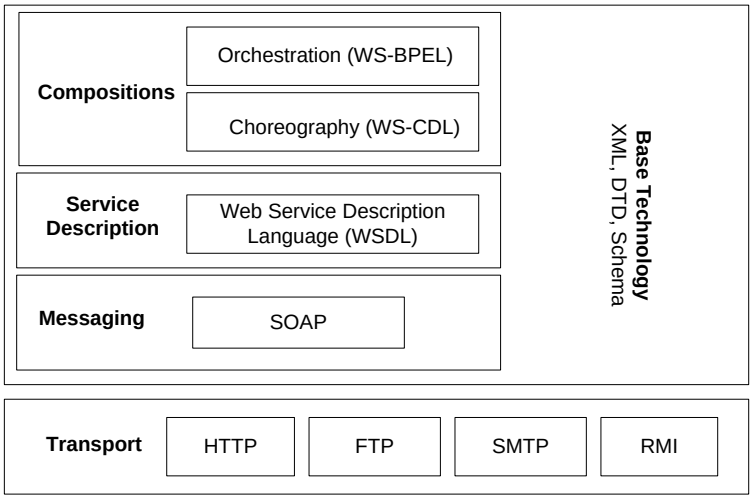


Figure 1: Web Services Architecture Stack

This definition already mentioned the fundamental technologies that are involved in Web services. Figure 1 shows these technologies, how they relate to each other and the different layers of their responsibility. Their specifications are consistently standardized and publicly available. Web Service Description Language (WSDL) [117], SOAP [116] and the purpose of Hypertext Transfer Protocol (HTTP) and eXtensible Markup Language (XML) in the context of Web services are described below.

XML and HTTP XML and HTTP are the key enabling technologies for Web services. HTTP is used by Web services at the transport level, mainly to carry SOAP data. XML offers a standard and widely accepted data format with the advantageous properties of being flexible and extensible.

WSDL WSDL is an interface definition language based on XML, used to precisely describe the way of how a Web service is accessed. A WSDL document contains an abstract definition of communication endpoints and messages as well as a separate concrete description of the supported network protocols and data encodings. The concrete, reusable specification of the network protocol and data format is called a *binding*. *Endpoint* definitions are the combinations of a binding and a network address, and are grouped together to form a *service*. The data being exchanged is described by directly referencing XML schema types. *interfaces* are abstract collections of *operations* which in turn abstractly describe the actions supported by an endpoint. WSDL provides Web services with the ability to implement the SOA design principle of standardized service contracts, which was presented above.

SOAP SOAP is a communication protocol based on XML technologies. It comprises a framework for constructing messages that are to be exchanged between dis-

tributed computing platforms over a network. Most commonly SOAP is used together with HTTP as underlying transport protocols but Simple Mail Transport Protocol (SMTP), File Transfer Protocol (FTP) and Remote Method Invocation (RMI) are supported too. According to [96], the major design principles of SOAP are simplicity and extensibility.

2.1.2 Service Compositions

Composite services are the primary object of study of this thesis. Therefore, the following explanations will illustrate the concept of composite services. Services, adhering to the SOA design principles, can be combined to *composite services*. These, in turn, are offered as high-level services or business processes that may even span different enterprises. [96] defines a business process to be an ordered set of related tasks, respectively activities, that are performed to achieve well-defined business outcomes. Specifically, a process can be defined as a sequence of steps that is initiated by an event, transforms information, materials, or commitments, and produces an output [97].

The procedure of creating composite services is called *service composition* [3]. Service composition can happen statically or dynamically [30]. Static service composition takes place during design-time, i.e., the components are chosen, linked together, compiled and deployed. In contrast, dynamic compositions are adapted to a changing environment and changing requirements during execution-time.

Transactions are needed to add guarantees to the interactions that occur in composite services. The WS-Coordination [95] protocol in combination with the WS-Transaction [12] standards provide the ability to guarantee ACID (atomicity, consistency, integrity, and durability) properties to short-duration transactions and compensation mechanisms to long-running business activities in Web services.

Web services provide software systems with the characteristic of composability. Web Services Business Process Execution Language (WSBPEL) [94], which is maintained by the Organization for the Advancement of Structured Information Standards (OASIS), is a language, based on XML technology, for specifying business processes composed of Web services. In Figure 1, showing the stack of Web service technologies, WSBPEL is located at the top most layer (*Compositions*). WSBPEL extends the service model of WSDL by providing means to describe peer-to-peer interaction between Web services. A process defined by WSBPEL is in turn exposed as a Web service using WSDL. While WSBPEL is not used within this thesis to create a composite service of Web services, as explained in Section 4.1, it is nevertheless one of the most important technologies of service composition.

In the context of Web service composition it is being differentiated between choreography and orchestration [98]. Orchestration describes interactions from the viewpoint of a participator that controls the interaction, i.e., an executable process is described. Choreography, in contrast, tracks the message sequences between the participating parties by describing interactions from the viewpoint of an independent observer. Whereas WSBPEL is an orchestration language, WS-CDL [114], also based on XML, describes collaborations of peers by choreography.

2.1.3 Service Level Agreements (SLAs)

Services and composite services are subject to SLAs [28]. [24] defines SLAs to be mandatory agreements which define mutual understandings and expectations between the provider and the consumer of a service. Specifically do they define the quality of the service offered by a provider to a customer and the circumstances under which this quality is guaranteed. According to [55], SLAs may consist of, but are not limited to, the following contents:

- The *purpose* that describes the reasons behind the creation of the SLA.
- *Parties* that are involved in the SLA and their respective roles (i.e., provider and consumer of a service).
- The *validity period* that defines the period of time (start and end) that the SLA will cover.
- The *scope* that defines the services that are covered by the agreement.
- *Restrictions* that define the necessary steps to be taken in order for the requested service levels to be provided.
- The *service-level objectives* that are agreed upon, and usually include a set of service level indicators, like availability, performance and reliability.
- *Penalties* that define what happens in case the service provider does not meet the objectives in the SLA (e.g., contract termination).
- *Optional services* may be defined that are not normally required by the user, but might be required as an exception.
- The *exclusions* that specify what is not covered in the SLA.
- *Administration* that describes the processes to meet and measure the objectives of the SLA and defines organizational authorities for overseeing them.

In service-oriented environments, where services are discovered and bound to on demand, service consumers and providers tend to automate the ways of handling non-functional service properties. Services are selected by a consumer in accordance with his non-functional requirements. Additionally, services are expected to have properties that are guaranteed by the provider. Therefore non-functional requirements are formulated in SLAs that guarantee specific quality related service properties to which both, the service providers and consumers are committed to. Those properties are referred to as Quality of Service (QoS) [70]. In contrast to traditional „paper“ SLAs that are formulated in plain natural language, SLAs in service-oriented environments are of dynamic nature [104]. They are specified by formal languages that provide the ability to be automatically processed by information systems. HP [18] demonstrated the use of a contract definition language in 1998 within its architecture for SLA management in federated environments. IBM published the WSLA framework [57], which provides automatic management functionality for SLAs in Web services, in 2004. While the contract definition language by

HP used a C-like syntax, IBM's WSLA-language was based on XML. Both approaches showed the emerging need for standardized ways of specifying SLAs and managing their (possibly short) life-cycles with minimal human interventions.

WS-Agreement

A standard way of establishing agreements between a service provider and a service consumer is WS-Agreement [92]. It represents a protocol that is based on Web services and defines XML schema for specifying agreements as well as operations for the creation, expiration and monitoring of agreement states for the management of agreement life-cycles. An extension to WS-Agreement is WS-Agreement-Negotiation [93] that specifies the way of how agreements are negotiated among two parties and that can be used to renegotiate agreements which need to be modified.

SLA Monitoring and QoS

To guarantee the conformance with already established SLAs, service providers and consumers use infrastructures to monitor the status of SLAs (i.e., to see if they are violated or fulfilled). This is done by measuring a service's QoS attributes in accordance with the objectives defined within a SLA. Some quality aspects of Web services, whose metrics may also be measured by monitoring infrastructure, are listed below [69]:

- *Availability* represents the probability that a service is available. It may also refer to the percentage of time that a service is operating [70].
- *Accessibility* represents the degree to which a Web service is capable of serving requests (e.g., a probability measure denoting the success rate or chance of a successful service instantiation at a point in time).
- *Integrity* refers to the correctness of the interaction in respect of the contract (i.e., WSDL description). Correctness of interaction can be provided by a proper usage of transactions.
- *Performance* is measured in terms of throughput and latency. Whereas *latency* refers to the elapsed time between sending a request and receiving the response, *throughput* represents the rate of Web service requests that can be processed and served in a given period of time.
- *Reliability* represents the capability of a Web service to maintain the service and its quality (e.g., the number of failures per time period) as well as the capability of assuring the ordered delivery for messages being sent and received by service providers and consumers.
- *Regulatory* describes the conformance, e.g., with law, compliance with standards, and established service level agreements.
- *Security* refers to the presence of authentication, encryption and access control mechanisms to ensure confidentiality and non-repudiation of messages or requests.

Basically there are two ways of monitoring the metrics of services. Services can be monitored from within the service (server-side) or from the outside of the service (client-side). Monitoring from the client-side usually meters just a snapshot of a QoS attribute at a discrete point in time as probes can only be taken at specific time intervals [74]. As suggested in [70], a customer may also delegate monitoring to a third-party monitoring service.

2.2 Cloud Computing

This chapter gives a brief introduction to Cloud computing and points out the characteristics and benefits of this paradigm. Amazon EC2 [6], a Cloud platform offered by Amazon, is presented to see the scope of services and functionality of representative public Cloud offering. The Eucalyptus [36] open source Cloud platform, which enables enterprises to build a Cloud from commodity infrastructure components, is discussed to complete the understanding about Cloud computing on the one hand and to introduce the Cloud platform being used within this thesis on the other hand.

When studying publications on Cloud computing it is easy to tell, what the term Cloud computing actually covers, but it turns out that it is even harder to specify what is not covered by this term. This aspect was also part of the industries' thoughts on Cloud computing a few years ago as Larry Ellison, CEO of Oracle, said at an Oracle OpenWorld conference and Andy Isherwood, vice president on software service of HP in Europe, told ZDNet UK:

> „The interesting thing about cloud computing is that we've redefined cloud computing to include everything that we already do. I can't think of anything that isn't cloud computing...I don't understand what we would do differently in the light of cloud." [41] Larry Ellison, quoted in the Wall Street Journal, September 26, 2008

> „A lot of people are jumping on the bandwagon of cloud, but I have not heard two people say the same thing about it. There are multiple definitions out there of 'the cloud'.' [15] Andy Isherwood, quoted in ZDnet News, December 11, 2008

2.2.1 Introduction

Two approaches of describing Cloud computing will be made. Firstly, a short outline of the basic concepts of Cloud computing that are mentioned by scientific literature will be presented. Secondly, a definition proposed by the authors of [110], which studied around 20 definitions of what a Cloud is, will be shown.

The basic idea of Cloud computing is that of utility computing where computing resources are consumed and provided like water, gas, electric power and telephony [22]. In terms of Cloud computing those resources are provided as services like virtual infrastructure, software applications and middleware platforms [64]. They are accessed over the public Internet or a private network like a LAN and WAN using standardized communication protocols (i.e., TCP/IP) [47]. The datacenter hardware and software that provide those services are referred to as the „Cloud". If they are provided to the public, then it is referred to as a „Public Cloud". If provided to a limited set of clients it is called a „Private Cloud" [13].

In [110] the authors encompass a definition of what can be understood by the term „Cloud“:

> „Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs.“ [110]

These definitions suggest the core concept of Cloud computing being the consumption and offering of virtualized computing resources and the very specific way of how they are made accessible to their consumers. To go into more detail, the five essential characteristics of Cloud computing as proposed by the National Institute of Standards and Technology (NIST) in [89] are presented in the following.

On-demand self-service	Customers can decide by themselves when to provision computing capabilities and when to use a certain service without the need for human interaction with the service provider.
Broad network access	Customers have access to the services and capabilities by means of a computer network and standardized mechanisms.
Resource pooling	Customers are served by the providers using multi-tenancy. Thereby the resources of a provider are pooled and dynamically assigned and reassigned according to the demand of customers. Customers do not know where their resources are exactly located but they may chose for example from a given set of geographically distributed data centers.
Rapid elasticity	Customers are able to provision new resources to their applications in a rapid and elastic way. For the customer the illusion appears of almost infinite available resources that can be purchased in any quantity at any time.
Measured Service	To provide usage based means of charging and transparency for the provider and customer, mechanisms are used to meter the usage of resources and services.

Now that Cloud computing was defined and its characteristics were shown, the scenarios of Cloud computing and its participating parties will be described.

2.2.2 Cloud Actors and Scenarios

Three stereotypes or actors (shown in Figure 2) can be identified that participate in the scenarios where Cloud computing may take place [110]. *Service Users* are accessing and consuming services (e.g., software applications). Those services are provided by the *Service Providers* that in turn use the infrastructure that is offered by *Infrastructure Providers*. Therefore it's not

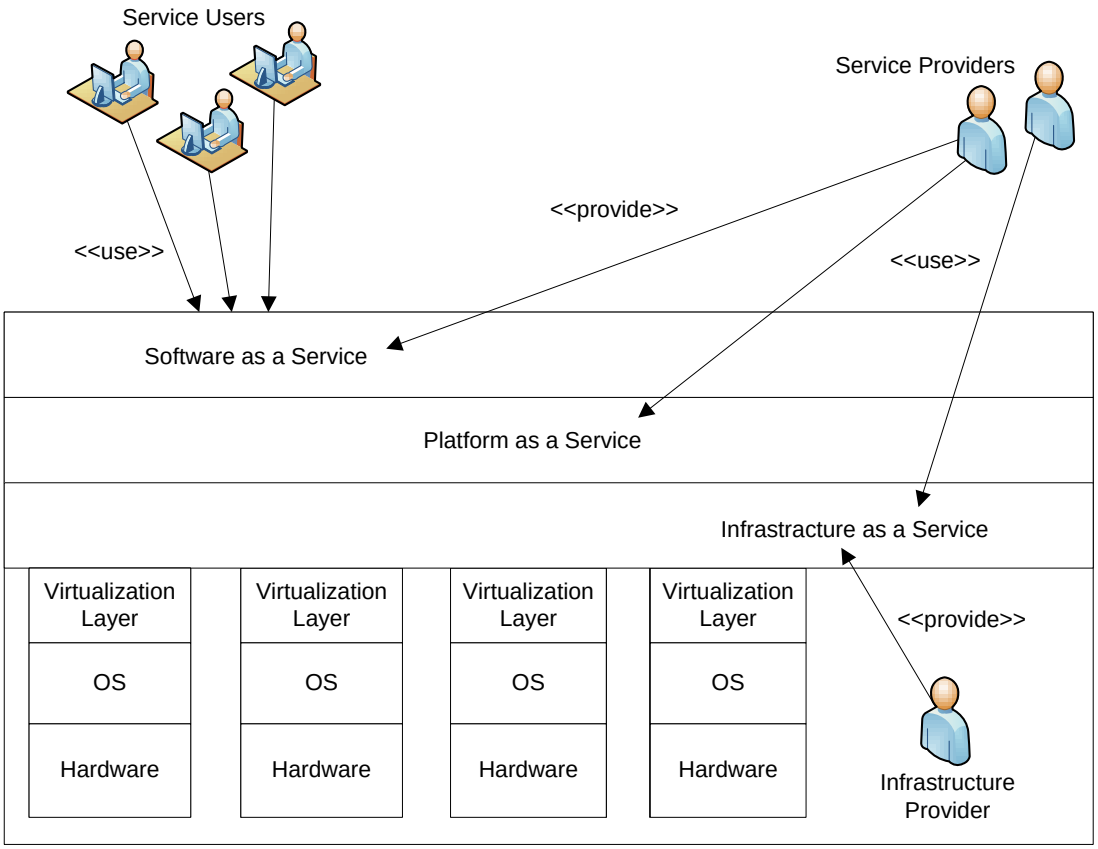


Figure 2: Cloud Actors (adapted from [110])

necessary for the *Service Providers* to operate resources by themselves. [110] The following three scenarios, respectively models, are mentioned in [110] and are also proposed in [89] by the NIST.

Infrastructure as a Service - IaaS Infrastructure as a Service (IaaS) is a Cloud computing scenario where virtualized infrastructure components like storage, network and computing resources are provided „as a service“. As will be illustrated in more detail in 2.2.3, Amazon provides such services with its Elastic Compute Cloud (EC2). Hereby, Amazon acts as the *Infrastructure Provider* who is operating the resources that are provided to his customers, the *Service Providers*, e.g., as virtual machines. The *Service Provider* on the other hand is able to deploy those virtual machines by means of self-service and is able to setup the necessary software stack to run his services.

Platform as a Service - PaaS A different Cloud scenario is Platform as a Service (PaaS). PaaS offerings provide an abstraction from the hardware resource layer and instead provide a complete

software stack for applications or services to run on. Examples of such Cloud offerings are Google Apps Engine and Microsoft Azure.

Software as a Service - SaaS In the Cloud computing scenario of Software as a Service (SaaS) the *Service Users* are able to access software applications like web-based email clients or web-based office and business applications from anywhere in the world. They do not have any knowledge about and are not able to access the underlying infrastructure. Prominent examples of such services are Google Gmail, the CRM applications of Salesforce.com and Microsoft Office Live.

Everything as a Service - XaaS Other terms like *Database as a Service* or *Storage as a Service* do exist that refer to even more Cloud computing offerings. Therefore the term „XaaS“ has been established that refers to *Everything as a Service* [64].

2.2.3 Amazon Elastic Compute Cloud - EC2

Amazon EC2 is part of the Cloud service offering AWS. EC2 is an IaaS Cloud service which enables users to run virtual machines on the data center infrastructure of Amazon. Management capabilities for EC2 are provided to users as web-based graphical interfaces and well-defined Web service interfaces (SOAP and REST [118]). The functionality of Amazon EC2 can be combined with the storage services Amazon Simple Storage Service (S3) and Elastic Block Store (EBS) which will be discussed later on in this section.

Locations

Amazon provides multiple geographically distributed locations where EC2 virtual machine instances can be deployed to by customers. Those locations are comprised of regions and availability zones. By the time of this writing the following regions are available to AWS customers:

- Virginia (US East),
- Northern Carolina (US West),
- Ireland (EU),
- Tokyo (Asia Pacific) and
- Singapore (Asia Pacific).

Every region consists of up to four availability zones. Availability zones are isolated from the errors that may occur in the other availability zones of the same region in so far as they do not share any critical resources. Applications can be distributed among different availability zones to provide higher availability and fault tolerance. Network traffic between availability zones in the same region is free of charge and offers high bandwidth as well as low latency.

	Cores	ECUs	Memory	Storage	Platform	I/O
	#		GB		bit	
m1.small	1	1.0	1.7	160	32	moderate
m1.large	2	4.0	7.5	850	64	high
m2.xlarge	2	6.5	17.1	420	64	moderate
m2.4xlarge	8	26.0	68.4	1690	64	high

Table 1: Resource Capacity of EC2 Instance Types [4] (as of 2011-05-23)

	Linux/Unix Usage	Windows Usage
	$	
m1.small	0.095	0.12
m1.large	0.380	0.48
m2.xlarge	0.570	0.62
m2.4xlarge	2.280	2.48

Table 2: EC2 Prices in EU (Ireland) per Hour [5] (as of 2011-05-23)

Instance Types

Amazon EC2 uses instance types to classify the amount of resources that are provided to a virtual machine instance. An instance type is chosen by the user at creation time of the virtual machine. A few representational instance types that are available on EC2 by the time of this writing are shown in Table 1. Amazon indicates processor capacity by EC2 Compute Units (ECUs), where one ECU provides the equivalent CPU capacity as a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. Input/Output (I/O) performance is indicated by the classifications *low, moderate, high* and *very high*. The higher the I/O performance of an instance, the larger is the allocation of shared resources and the smaller is the variance of I/O performance.

Pricing

All AWS offerings are priced on a pay-per-use basis where users only pay for resources that were actually used. EC2 instances are accounted on an hourly rate. The actual price per hour depends on the instance type used and the region the instance is deployed to. Table 2 shows the hourly prices of some EC2 instance types in region EU (Ireland).

Amazon S3 - Simple Storage Service

S3 is part of the Cloud service-portfolio AWS. It provides online storage with a simple structure for data organization. Amazon S3 offers both, a SOAP- and a REST-based Web service interface. Data is organized as *buckets* that contain *objects* which in turn contain the actual data. Buckets

in S3 can not be nested, thus providing a strictly flat hierarchy. Every object is addressable by its globally unique object- and bucket-name. The maximum size of an object is 5 GB.

Amazon EBS - Elastic Block Store

With Amazon EBS, customers can create block-based volumes and attach them to virtual machine instances within the same availability zone. EBS volumes behave like regular disks and can be formatted with any filesystem. They continue to exist after a virtual machine has been shut down, keeping all data, but can only be attached to a single machine at the same time. To persist a certain state of an EBS volume's data, a snapshot of the volume can be created and stored into Amazon S3.

AMIs - Amazon Machine Images

Virtual machine instances are created from Amazon Machine Images (AMIs). AMIs contain the operating system and pre-installed software that will constitute the software environment of a virtual machine after its creation. Amazon provides many pre-built AMIs with differing operating systems like Microsoft Windows and GNU/Linux. Customers can create their own AMIs by customizing already existing ones and uploading them into Amazon S3 either for private use or to provide them to the public.

Elastic Load Balancers and Elastic IPs

Some further concepts of EC2 are *Elastic IPs* and *Elastic Load Balancers*. Elastic IPs provide customers with the ability to allocate public Internet Protocol (IP) addresses. They can be allocated independently and may be attached to and released from virtual machines at any time. As such they are usable in public Domain Name System (DNS). Elastic Load Balancers are virtual load balancers that can be created to distribute the load (e.g., HTTP requests) among multiple EC2 instances. They may be configured in such a way to start and stop virtual machines automatically to ensure availability and performance of applications.

2.2.4 Eucalyptus

Eucalyptus is an open source software implementation that enables enterprises to use their existing IT resources and data center infrastructure to build an IaaS Cloud. The solution proposed in this thesis is based on the concepts of Eucalyptus, whose functionality is closely aligned to the features offered by Amazon EC2. Eucalyptus provides services for the management of virtual machines as well as block- and bucket-based storage. The Application Programming Interfaces (APIs) offered by Eucalyptus are fully compatible with the interfaces of AWS, therefore the AWS Software Development Kit (SDK) may be used to interact with Eucalyptus. Virtualization technologies supported by Eucalyptus are XEN [108], KVM [14] and VMWare [49], whereas VMWare is only supported in the commercial edition of Eucalyptus. Eucalyptus also supports concepts like *Elastic IPs*, availability zones and user self sign-up [37].

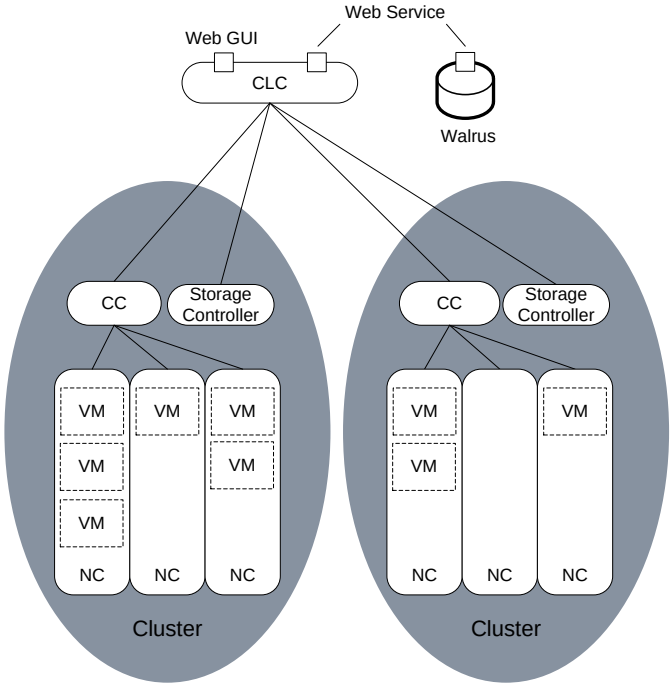


Figure 3: Eucalyptus Hierarchical Architecture

Eucalyptus is implemented as open source and runs on GNU/Linux. The source code can be downloaded from the Eucalyptus website [36]. Binary packages are available for multiple GNU/Linux distributions (e.g., Red Hat, Fedora, openSUSE).

Architecture

The architecture of Eucalyptus reflects underlying resource topologies by means of a hierarchical design. Three central software components, the Cloud Controller (CLC), the Cluster Controller (CC) and the Node Controller (NC), constitute this architectural concept. They are deployed to the various distributed hardware resources of the Cloud infrastructure and communicate with each other over the network using Web service technologies and well-defined interfaces described by WSDL contracts [91]. This hierarchical concept is shown in Figure 3. The controllers are described in the following.

Cloud Controller (CLC) The CLC is the single entry-point to the Cloud infrastructure. It is responsible for high level scheduling decisions by querying the underlying components for resource information and implementing decisions by sending requests to the cluster controllers. The CLC abstracts from the underlying, possibly heterogeneous, virtualization, storage and network components and provides a consistent view to the users and external components. The API is compatible to the

one provided by Amazon EC2 as it is built from the WSDL-service contract of AWS.

Cluster Controller (CC) The CC is a front-end component to a set of nodes that host virtual machines. It communicates with the CLC as well as the node controllers to distribute the execution of virtual machines among the hosts. The CC also manages the virtual network used by the virtual machines and participates in the enforcement of SLAs as directed be the CLC.

Node Controller (NC) The NC software is executing on every host that may potentially run virtual machines. It is responsible for the inspection, execution and termination of virtual machines on its host. The NC therefore communicates with the host's operating system and virtualization hypervisors in response to queries and control requests from the CLC. The NC also manages images of its virtual machines (i.e., operating system, root filesystem and ramdisk image) and their virtual network endpoints.

Virtual Storage

Eucalyptus also provides storage functionality like Amazon EBS, for block-based storage, and Amazon S3, for bucket-based storage. Block storage in Eucalyptus is controlled via a component that is called *Storage Controller*. The *Storage Controller* is capable of interfacing with various storage systems (e.g., NFS, iSCSI, FC) and provides block store devices that can be attached to the filesystem of a virtual machine. The storage service *Walrus* is a pendant to Amazon S3 where persistent data is organized in buckets and objects. Buckets support `create`, `list` and `delete` operations and objects can be accessed by the operations `put`, `get` and `delete`. To restrict the access of data, access control policies are supported. As stated by the Eucalyptus manual [37], the SOAP and REST interface provided by Walrus is fully compatible to the interface of Amazon S3.

Instance Types

Eucalyptus uses predefined instance types to describe the hardware resources of virtual machines. Eucalyptus does not implement all instance types of Amazon EC2 but in contrast to Amazon EC2, the amount of resources associated with an instance type can be modified. The creation of new instance types as well as changing the name of existing instance types is not supported. Whereas in Amazon EC2 the instance types *m1.small* and *c1.medium* can only be deployed to 32-bit platforms and all other instance types are based on a 64-bit architecture, Eucalyptus instance types can be deployed to any platform [17].

2.3 Autonomic Computing

The progress of information technology and its objective to build more powerful computing systems is accompanied by an increase in complexity that gets more and more difficult to be

handled and overseen by human beings. When IBM [51] published their manifesto on *Autonomic Computing* in 2001 they were motivated by exactly this development and warned that the benefits of IT are threatened from being undermined by the growing complexity of IT infrastructures. To overcome this development, the paradigm of autonomic computing, dealing with the topic of *Autonomic Systems*, systems that are able to fully manage themselves with just minimal human intervention, was proposed. The thesis at hand seizes on the idea of autonomic systems. To implement this functionality in the proposed composite service runtime, its architecture is based on the closed control loop of autonomic systems, which is described below.

As it is not trivial to implement autonomic computing capabilities into existing IT infrastructures, [88] defined the following levels that describe the path towards autonomic computing. They outline the degree of an IT infrastructure's ability to govern itself by means of autonomic computing.

- **Basic Level**: This level is the starting point. Each system is managed independently by system administrators. No aspects of self-management are implemented at this level. System administrators set up their systems and just enhance them as needed.
- **Managed Level**: At this level, system management technologies are used to collect information from different systems. They simplify the collection and evaluation of information. While this is the starting point of automation of IT management, most of the evaluation is still performed by system professionals.
- **Predictive Level**: This level makes use of new technologies that enable individual components to monitor themselves, analyze their collected data and offer advices to system administrators. Human interaction is drastically reduced and decision making is improved.
- **Adaptive Level**: At this level, systems are first able to perform actions on their own, based on the information that is available to them. Human interaction is still necessary to a minimal degree.
- **Autonomic Level**: This is the level of full autonomy where the system operations are only managed by business policies and objectives that were established by administrators. Systems are still monitored by humans but they only interact with the system to change their policies.

2.3.1 Self-management Capabilities

The ability of *self-management* [58] is the fundamental function of an autonomic system. Self-management involves the following areas:

- **Self-configuration** is the ability of automated configuration of components following high-level policies
- **Self-healing** provides automated detection and repair of defective components
- **Self-optimization** refers to the automated detection of opportunities to improve performance

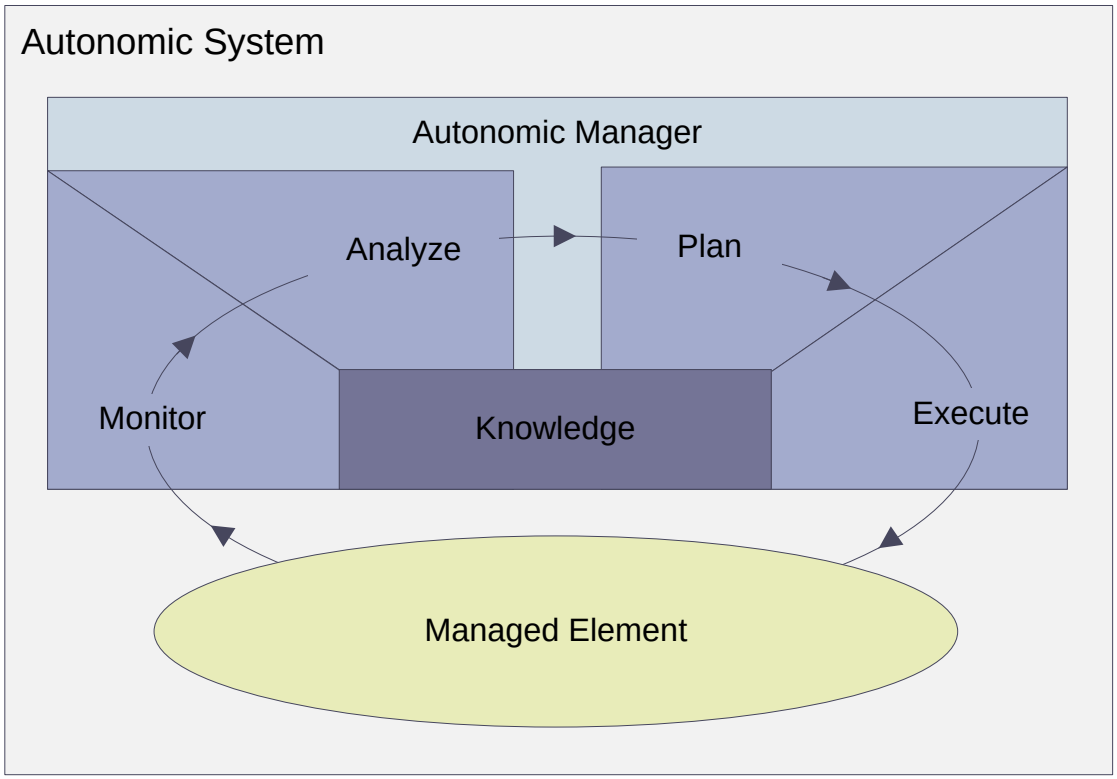


Figure 4: Autonomic System (adapted from [58])

- **Self-protection** refers to the ability of automated prevention and defending of malicious attacks

Apart from being master in these areas of self-management, an autonomic system not only needs to know itself but also the environment and the context surrounding it. It will have to function in heterogeneous environments and unavoidably coexist together with other systems where interdependence exist to a certain degree. Therefore autonomic computing strongly relies on the development of open standards, especially will autonomic systems have to implement open standards that enable them to interact regardless of their underlying platform [51].

2.3.2 Autonomic Computing Architecture

[58] describes the structure and architecture of autonomic systems. An autonomic system consists of multiple *autonomic elements* that interact with each other. They in turn contain resources and deliver services. Figure 4 shows the architectural concept of an autonomic element. Autonomic elements are managing themselves as well as their interactions with other autonomic elements in accordance to policies that were established by humans or other autonomic elements. An *autonomic element* consists of one or more *managed elements* that are coupled with one *autonomic manager*. The autonomic manager is responsible for controlling the managed

element and for representing the managed element to the outside environment. For example, the managed element is a hardware device, such as a Central Processing Unit (CPU), a storage system, a software service, a web server or a database.

As shown in Figure 4, the functionality of an autonomic manager involves the following four tasks: *monitor*, *analyze*, *plan* and *execute*. This sequence of tasks, the autonomic manager iterates over continuously, is defined by [58] to be the closed control loop of autonomic systems. By monitoring the managed element and the environment, data is collected and stored into a knowledge base that may be used by all functional components of the autonomic manager. The collected data is analyzed to see if any actions and alterations to the managed element are required. This may for example be needed because a policy is not being met anymore. In this case a planning function is triggered, that is responsible for the creation of an appropriate change plan. This plan represents the alteration, which will be applied to the managed element. The plan is then passed to the execution function, which schedules and performs the changes according to the plan. As can be seen from this procedure, the knowledge base is the central part of the autonomic manager. It has to be shared among the before mentioned four tasks. The goal of this procedure and the autonomic manager is to relieve humans of the responsibility of directly interacting with the managed element.

2.4 Event Processing

The concept of *event processing* [35] is applied by the composite service runtime, developed as part of this thesis, to monitor itself and the execution of a composite service. As the name implies, event processing deals with events, especially the operations performed on them. Among those operations are *reading*, *creating*, *transforming* and *deleting* events. Two major areas are related to event processing. On the one hand, the topic of event processing is concerned with the construction of software applications that make use of events. This is also being referred to as event-driven architecture or event-driven programming. On the other hand, operations processing such events, and used by event processing applications, are of concern. Event processing can be applied in several scenarios and used by several categories of applications. Among the use cases of event processing are monitoring of business processes by looking at exceptional behavior, diagnosing a problem based on observed symptoms or delivering of personalized information. A luggage handling system at an airport, tracing the boarding and routing of luggage by means of radio-frequency signals, as well as an emergency control system, that detects incidents and notifies people, are applications that may use event processing. The logic of event processing is often encapsulated within a dedicated event processing platform and separated from the software components that create or take notice of events. Some event processing platforms are for example Esper [34], which is used within this thesis, Apama[1] and Streambase[2].

2.4.1 Concepts of Event Processing

The following terms illustrate the concepts of event processing applications as defined by [35].

[1] http://web.progress.com/en/apama/ Visited: 2011-09-29

[2] http://www.streambase.com/index.htm Visited: 2011-05-29

Event An *event* represents an occurrence, something that has happened, within a particular system or domain. The term *event* also refers to the programming entity in a computing system that holds the information associated with an event.

Event Producer An *event producer* is a component that represents the source of events and emits them to other components inside an event processing application or event processing system. It can be a hardware device like a physical sensor or a part of a software application.

Event Consumer *Event consumers* are components that receive and process events inside an event processing application or event processing system. Consumers can be any kind of physical actuators that perform actions on receipt of an event or software modules like an event log or a business application.

Event Processing Agent Intermediary event processing might be done by dedicated *event processing agents*. They are able to consume and produce events by receiving and forwarding them. Event processing agents can be used to filter events, transform events and to detect patterns in the event flow.

Event Stream An *event stream* is a set of associated events that is often temporarily ordered on the basis of timestamps that are part of the events. Whereas homogeneous events streams only consist of events of the same type, heterogeneous event streams may contain events of different types.

2.4.2 Principles of Event Processing

The two key attributes of an event-driven architecture are the extremely loose coupling and the high distribution of its components. Producers of an event neither have knowledge of the parties that are interested in the events nor do they know any further processing steps of the events [71]. [35] defines the following general principles of event processing:

Decoupling The producer of an event does not depend on any particular processing or course of action being taken by the consumer of the event. Conversely, the consumer does not depend on any action performed by the producer other than the creation of the event itself.

Asynchrony Events are often sent as one-way messages. A producer does not have any expectations on a consumer to take a specific action after it received an event. After a producer has sent an event message, it gets on with other things and without having to wait for responses while the consumer of an event processes the event independently.

Push-style interaction With push-style event distribution the event is transported to interested parties by sending them a one-way message that contains the event as soon as the event has occurred.

Pull-style interaction Within pull-style event interaction the events are requested by interested parties on demand. When used in combination with the request-response pattern the event is passed within the response.

Channel based distribution Two issues arise with directly transporting events from a producer to consumers. Firstly, the producer must have knowledge of the consumers, e.g., by consulting an external information source or by the consumers subscribing dynamically at the consumer. In any way this implies additional responsibilities on the event producer. Secondly, the producer may have to transport multiple copies of the same event to individual consumers. Both responsibilities can be delegated to an intermediary event channel, that is logically positioned between the producer of an event and any consumers. The event channel takes care of forwarding events to the event consumers.

Request-response interaction There is a fundamental difference between the request-response pattern used in traditional software applications and the interaction style used with event processing. Whereas an event is the indication of something that has already happened, a request expresses the requester's wish that something specific should happen in the future. The request-response interaction pattern can also be used to transport events from a producer to a consumer. The event is thereby passed as a parameter on the request or the response.

2.4.3 Event Processing Styles

[71] describes three basic styles of event processing that differ in their complexity and that may also be used in combination with each other. Thereby it is distinguished between *ordinary* and *notable* events. While notable events trigger certain specific actions, ordinary events are screened for notability and just passed to information subscribers.

Simple event processing In simple event processing, notable events occur that directly dictate certain actions. It is commonly used to drive the real-time flow of work, reducing lag-time and cost.

Stream event processing Stream event processing deals with the correlation and aggregation of ordinary events. Ordinary events, usually of the same context, are thereby filtered, routed and reformatted into notable events that in turn dictate certain actions. Stream event processing is used by enterprises to enable in-time decision making by driving the real-time flow of information.

Complex event processing - CEP Complex Event Processing (CEP) is commonly used to detect and respond to business anomalies, threats, and opportunities. A confluence of ordinary and notable events that may occur over a long period of time are evaluated using sophisticated techniques for event pattern detection, matching and correlation. Common usage cases of complex event processing are the detection of the absence of an event or the comparison event patterns to multiple conditions and past data.

2.4.4 WS-Eventing

WS-Eventing [115] [96] was published by Microsoft, BEA Systems, Sun Microsystems and Tibco Software. Its specification allows Web services to subscribe to events that occur at other Web services. As part of the WS-* specifications, WS-Eventing is intended to work together with other WS-* specifications like WS-Addressing and WS-Security. The basic concepts of WS-Eventing are the following. Web services that send one-way messages, called *notifications*, to indicate that an event has occurred are called *event source*. An event source also accepts requests for subscriptions. Notifications are received by Web services that are called *event sinks*. A *subscription manager* is a Web service that manages the status, renewal and deletion of subscriptions on behalf of an event source and accepts requests from a *subscriber* which demand such management actions.

CHAPTER 3

Related Work

The following sections deal with work related to the concepts used in this thesis. To begin with, a study on hosting web servers in the Cloud showing the basic principles and capabilities of hosting in the Cloud is presented. After that, approaches to adapting and monitoring composite services are discussed. On the one hand, concepts focusing on single composite service instances and on the other hand, concepts focusing on concurrently executing composite service instances are shown. Finally, work on the topic of migrating processes, services and Virtual Machines (VMs) is presented as it is related to the migration of composite service instances investigated in this thesis.

3.1 Dynamic Hosting Scenarios in the Cloud

The authors of [67] discuss the various scenarios of how to host web servers in EC2. They also study and compare two different work load and traffic patterns. The conclusion of their work is that, in terms of resource utilization, a single web server setup is not best suited for all different kinds of considered traffic pattern. Rather each hosting scenario should be fitted to one specific pattern. The work in the thesis at hand is based on the principal hosting scenarios and uses them in a modified way to host the runtime environment of a composite service.

The authors of [67] also propose criteria for switching between the hosting scenarios dynamically and to provision additional resources to them to cope with changing loads on the web servers. In the thesis at hand resource provisioning mechanisms are used to serve the same purpose of handling a too high load in Eucalyptus instead of EC2.

For one of the two traffic patterns mentioned above, an encrypted network channel is used to transfer short requests and responses. This pattern results in a high amount of CPU utilization. The other traffic pattern results in a high consumption of network bandwidth, whereby files of a size up to 40 MB are downloaded over an unencrypted channel.

Benchmarks of the two traffic patterns were performed with three instance types, *m1.small*, *c1.medium* and *c1.xlarge* (see Table 1 for resource specifications), of Amazon EC2. The results

of the benchmarks show, that the *m1.small* instance is not able to saturate it's network interface, neither with the CPU intensive pattern nor with the bandwidth intensive pattern. The CPU intensive pattern exhausts the CPU at an amount of 1,350 parallel sessions, whereas the bandwidth intensive pattern exhausts the CPU at 1,190 parallel sessions. Once the CPU resources for the bandwidth pattern were slightly increased by using *c1.medium* instead of *m1.small*, saturation of the network interface was accomplished at 1,800 parallel sessions. A *c1.xlarge* instance was able to serve up to 7,000 parallel sessions during the load of the CPU intensive traffic pattern.

The following four different scenarios of hosting web servers in EC2 are considered in [67]:

- A web server is hosted on a single small instance (*m1.small*). This is the cheapest way of how to run a web server in EC2, but also provides the least resources. It is best suited for a low web server load, for instance during the night. The CPU capacity of the *m1.small* instance type is equivalent to a single 1.0-1.2 GHz 2007 Opteron of Xeon processor. The bandwidth of the network interface is about 800 Mbps.

- A load balancer is used to distribute the load among multiple VM instances hosting the web servers. While the CPU capacity for the web servers is virtually unlimited, the network traffic to the load balancer (and therefore the traffic of the clients) is limited to 400 Mbps because the load balancer needs to relay traffic to the web server instances. The load balancer is hosted by a *c1.medium* instance and *m1.small* instances are used for the web servers.

- The web server is hosted on a single large instance (*c1.xlarge*). The CPU power of a *c1.xlarge* instance is many times greater than the one of a *m1.small* instance. *c1.xlarge* provides 8 virtual CPUs with each being clocked at a higher frequency than that of *m1.small*. The network bandwidth is again 800 Mbps.

- DNS load balancing is used to distribute the load among multiple machine instances. This scenario is useful if the needed bandwidth is too high for a single Cloud component to cope with, or the CPU utilization exceeds the capacity of a single instance. DNS is configured in such a way, that it points to multiple IP addresses (instead of only one) where each one corresponds to a single *c1.xlarge* instance. This scenario is only bounded by the traffic limit into the Amazon EC2 infrastructure.

The criteria for switching dynamically among these scenarios, in case the load on the web servers changes, are based on the benchmarks described above. Switching from a *weaker* scenario to a *stronger* one takes place if either CPU capacities or network bandwidth exceed the limit of the current configuration. In case the bandwidth does not exceed 400 Mbps, but the CPU capacities are exhausted, a switch from the single *m1.small* instance to a *c1.medium* load balancer and *m1.small* instances would take place. If bandwidth utilization is between 400 and 800 Mbps a switch to the single large instance will be necessary. More than 800 Mbps can only be handled by the DNS load balancing configuration. The criteria for switching downwards are exactly the opposite.

The work in [67] shows the basic capabilities of the Cloud, specifically EC2, to deploy hosting environments that each provide different amounts of resources and to switch among them.

The concept of hosting with single instances is used in the thesis at hand to accomplish the task of dynamic resource provisioning for composite services. It is not considered in [67] how automated switching facilities among the discussed configurations may be implemented. In addition, no attention was payed to the cost, and time needed for a possible switching process The thesis at hand implements switching facilities and studies the costs and the effectiveness of dynamic resource provisioning mechanisms in the Cloud, specifically Eucalyptus. The benchmarks and deployment scenarios focus on Service-oriented Computing (SoC) instead of web servers and web applications. Furthermore, monitoring in [67] is based on measuring the consumption of resources instead of evaluating SLA conformance.

3.2 Adapting and Monitoring Composite Services

The papers presented in this section differ significantly in their approaches to coping with varying influential conditions that composite services are confronted with during their existence, specifically during their execution time. They also deal with different ways to establish mechanisms to monitor composite services and their hosting infrastructures.

[87] and [62] focus on adapting single composite service instances to prevent them from violating SLAs. They do not take into consideration the concurrency of multiple parallel composite service instances and the impact of their parallelism on their execution duration. Neither do they make adaptations to the infrastructure of the hosting environment, that operates the composite services or the called Web services. In contrast, approaches like [29], [68] and [42] exist, that focus on coping with concurrent invocations of composite services to either ensure SLA conformance or generally decrease the execution duration of composite services.

While [29] and [42] balance the load only at the layer of Web service invocations, [68] takes load balancing of requests to the WSBPEL runtime into consideration. In contrast to all other papers presented, the approaches of [29] and [68] are able to dynamically adapt the hosting infrastructure in the Cloud or in similar dynamic environments.

These approaches, that attempt to cope with concurrent composite service instances, are based on different manifestations of load balancing and extend the WSBPEL engines to support the dynamic selection of Web service endpoints at runtime, bind to them and invoke them. This is accomplished by intercepting the invocation of Web services in the WSBPEL engine.

3.2.1 Without Concurrency Considerations

[87] presents the *ADULA* system. It implements an approach to automatically detect and repair SLA violations of composite services that are implemented as WSBPEL processes. The detection of violations is accomplished by using statistical tests on the duration of Web service calls and WSBPEL processes. If a Web service is detected that causes a WSBPEL process to violate a SLA, ADULA will replace it with an alternative one by rebinding the Web service call to a different Web service endpoint. The duration of Web service calls is measured by invoking them with special dynamic proxies that are able to monitor the duration between requests to and responses from a Web service.

Instead of modifying a WSBPEL runtime, like other approaches do [29] [68], the WSBPEL processes are transformed automatically before being deployed to ADULA. This transformation modifies a WSBPEL process in such a way that it requests Web service endpoints from ADULA at startup and notifies ADULA about its completion.

The whole evolution of a WSBPEL process is tracked. This means that the history of endpoint replacements are stored in a database, starting with the very first replacement. Initial service endpoints, chosen at development time of the WSBPEL process, are preserved after transformation. They can be used again, if the corresponding Web services recover from their degraded state.

ADULA provides the abilities to detect SLA violations and to repair them, so that future violations can be prevented. The repair strategy is limited to the replacement of Web services cause a composite service to violate a SLA. Otherwise, ADULA does not implement any actions to react on SLA violations whose cause are not the invoked Web services, e.g., insufficient resources for executing composite services.

A similar approach to SLA protection is presented in [62]. The authors propose a system, the *PREvent* framework, that is able to prevent SLA violations of composite services by applying certain adaptation actions to them at runtime. Instead of statistical tests, as in [87], they use machine learning techniques and regression models to predict SLA violations. PREvent operates in three phases. During the monitoring phase, runtime data of composite service executions is collected by an event-based monitoring component and stored in a metric database. At certain checkpoints within the execution of a composite service, the system tries to predict whether the composite service will violate a Service Level Objective (SLO) or not. Checkpoints are predefined and stored in a database that is loaded by the prediction component during the prediction phase. If the system predicts a violation it will trigger the adaptation of the composite service. Which adaptations are applied depends on the adaptation strategy being used. The *safe* strategy applies all possible adaptations so that the SLO will most likely be satisfied. The *minimal* strategy applies only those adaptations that will make the composite service just pass the SLO. This latter strategy has the drawback that an inaccurate prediction could lead to the SLO not being satisfied. Three different kinds of adaptation actions are implemented within the PREvent framework.

- The data exchanged within the composite service can be manipulated. An example mentioned in [62] is the interception of input messages to an activity that handles the shipping of certain goods. The value within the message that indicates whether *express* shipment will take place or not could be set to *true* by the adaptation component.

- The binding to a Web service can be changed at runtime. An example is the rebinding from a bad performing service to a better performing one.

- The third kind of adaptation enables changes to the structure of the composite service. Such adaptation actions can remove activities from or add activities to a composite service at runtime.

The evaluation of the two mentioned adaptation strategies showed that the system was able to prevent 78% of predicted violations by the use of the *safe* strategy and 60% by using the *minimal* strategy.

The PREvent framework will adapt composite services at runtime autonomically, so that they will satisfy the SLAs. It is not limited to the replacement of Web service endpoints, but also supports additional adaptation mechanisms to influence the logic of a composite service. However, PREvent does not yet support adaptations regarding the hosting environment of a composite service to prevent SLA violations that arise from it.

The approach of [62] is based on eventing (see Section 2.4) to collect runtime information about composite services and external occurrences. Like the adaptation framework presented in this thesis, PREvent uses the event engine provided by the *VRESCo* runtime (see Section 4.2). While monitoring in PREvent considers the metrics of single composite service instances, the monitoring mechanism that is used in the thesis at hand considers the aggregation of metrics from multiple composite service instances.

3.2.2 With Concurrency Considerations

[42] proposes dynamic endpoint selection based on the scheduling strategies *round robin*, *lowest counter first* and *weighted random*. The number of concurrent invocations to a Web service and the duration of recent invocations is tracked.

In addition to invocation interception in the WSBPEL engine, the WSBPEL process is modified before being executed by the WSBPEL runtime. The ActiveBPEL engine is extended with a scheduler module. The invocation activities of the WSBPEL process are wrapped with invocations to the scheduler module which results in a transformation of the process. The scheduler module is invoked like a Web service by the process, before and after the actual Web service is called. It can track the invocation time and the number of parallel Web service invocations.

In their experiments, the *weighted random* strategy provides the best overall results. Average duration of Web service calls within many concurrent processes was reduced by 27% compared to static processes.

The solution presented in this thesis is based on VRESCo 4.2 and makes use of the eventing, dynamic service invocation capabilities. It is compatible to the PREvent framework discussed in 3.2 which is also based on VRESCo and does not support infrastructural adaptations so far.

An approach to scaling WSBPEL processes dynamically in the Cloud is presented in [29]. The authors use load balancing to distribute the invocations of the WSBPEL process to the Web services among multiple VM instances. Their implementation is an extension to the ActiveBPEL runtime.

The topology of the system is composed from three main components. A resolver component intercepts the invocations of the WSBPEL runtime to the Web services and queries the load balancer component to get information about concrete Web service endpoints. The load balancer manages a registry that contains information about available Web services and the load on their hosting VM instances. On incoming requests, the load balancer decides, based on the information stored in the registry, to which Web service endpoint the invocation should be scheduled. If the load on all of the currently available VMs is too high, the load balancer launches additional VM instances by using the interfaces provided by the cloud platform. On the other hand the load balancer will terminate idle machines, if the system is underutilized. Monitoring of the system load is implemented by a Web service, which represents the third component. It is hosted on

each VM instance and provides information about CPU utilization on its host. This Web service is queried on demand by the load balancer.

The provisioning of new Web service hosts is accomplished by using the Amazon AWS SDK which provides functionality to launch and terminate VM instances. They use a customized AMI that downloads the application software from a central web server at startup so that it is not necessary to rebuild the AMI if updates to the software are deployed. This concept is also used in a modified form in the work of this thesis as described in Chapter 5.

In contrast to the work presented in [68] and to what this thesis focuses on, the load on the system hosting the WSBPEL runtime is not taken into consideration. The intention of [29] is soley to scale at the level of Web service invocations.

In [68], the authors focus on scaling at the level of the composite service runtime. They identify the composite service runtime to potentially be a bottleneck, in contrast to [29], where the focus is on scaling the hosting components of Web services that are invoked by a composite service runtime. Again, the service invocation mechanism of ActiveBPEL is extended by the authors of [68] to accomplish the dynamic binding and resolving of Web services.

Their performance evaluations show how a load balancing approach outperforms a single WSBPEL engine. Their experiments were performed on a sample composite service whose execution time was around 10 seconds. While the throughput of executing composite service instances at lower request rates is equal to the request rate itself, higher request rates result in the throughput declining below the request rate. The throughput was measured to decline at 240 requests per minute on a single WSBPEL engine in contrast to the load balanced configuration that won't decline until 420 requests per minute. These results illustrate that the throughput of a composite service can be increased by adding more WSBPEL engines to the system.

The approach uses a *Heart Beat* function causing the clustered WSBPEL engines to regularly report their living status and their load to a central component, the *Engine Manager*, which is part of the load balancer. A registry that is used by the formerly mentioned interception mechanism to find available services and bind to them at runtime is also part of the system.

The load balancer is exposed to clients in form of a Web service that can be called like the actual composite service that is to be invoked. The load balancer then schedules the execution of the composite service to one of the clustered WSBPEL engines.

What they do not take into account is the way of how to dynamically start and stop the WSBPEL engines as well as how they would provision additional resources for their hosting. Like in [29] the scaling mechanism depends on a central management component for monitoring and scheduling.

3.2.3 Monitoring

The load balancing solutions proposed in [29] and [68] use contrary attempts to monitor their hosting environment. While both of them use the idea of a central and static component that monitoring is built upon, i.e., the load balancer, the concrete realization differs. In [29], the load balancer component queries the nodes that are hosting services on demand. In [68], the nodes regularly report their load to the load balancer. While the former approach may therefore be pictured as *pulling* the information from the source, the basic idea of the latter approach is to *push* monitoring data from distributed sources to the central component, the load balancer.

The scheduling component used in [42] tracks the invocations to Web services as they arrive at the load balancer. While the pull and the push style monitoring is *actively* performed the latter mentioned tracking mechanism takes place *passively*.

In [86], the authors present the architecture and a proof of concept implementation of their approach to monitor composite service infrastructures for the occurrence of certain situations of interest, e.g., fraudulent behavior, failing operations or absent situations. Their approach is based on CEP, which was discussed in Section 2.4. An event model is proposed in [86] that organizes the different kinds of events in a hierarchy to generalize the way of handling them. The model distinguishes between simple base events and more complex domain specific events. Base events have general attributes like a timestamp and a success indicator. They are used to build domain specific events that inherit their attributes and extend them with more specific attributes. Domain specific events are used to represent the occurrence of a specific situation, e.g., the invocation of a certain composite service or a certain Web service. The core of the proposed architecture is represented by the *monitoring runtime* that integrates the event processing engine and certain adapters to outside components like the various event sources, event sinks and the composite service runtime. The implementation uses WSBPEL and supports the ActiveBPEL and the ApacheODE runtime engines. The implementation uses the event processing engine Esper [34] in combination with the Esper Event Processing Language (EPL). Creation of events is realized by a message interception layer that is intercepting the WSBPEL runtime at the SOAP layer and accesses SOAP messages. As part of the interception procedure event objects are created, stored in a database and passed to the event processing runtime.

The CEP approach to monitor a composite service infrastructure is used in the thesis at hand to evaluate the SLOs of composite services. CEP provides the capabilities to filter complex event patterns and to aggregate different events. The evaluation in [86] shows that eventing with the Esper runtime has almost no impact on the performance of the executed composite services and that the event processing runtime implemented therein is able to handle around 2000 events per second. The PREvent framework [62] uses a similar approach that is based on CEP to measure the metrics of composite service instances.

3.3 Migration and Relocation

The thesis at hand implements mechanisms that accomplish the provisioning of resources to composite services in the Cloud at runtime. Because Cloud platforms do not support the upgrade of VM instances, the concept of migration is used to transfer executing composite services from a VM instance to a better equipped one.

This section deals with the migration of executing entities between different computing systems, starting with a review of the concept of process migration, followed by a presentation on work regarding the migration of virtual environments and concluding with an approach to move runtime services from one virtual server to another.

3.3.1 Process Migration

In the 1980s and 1990s, academic research considered the concept of migrating processes and tasks between system nodes aiming at improving system performance, reliability and maintenance. [85] provides an overview on migration concepts and compares implementations of process migration in distributed operating systems at that time, like MOSIX, Sprite and Mach. The main goals of process migration are mentioned to be the following.

Distribution of load	Processes may be migrated from overladed nodes to less loaded nodes for better utilization of available resources.
Fault resilience	Processes that are executing on failing nodes can be migrated to correct nodes and devices.
Resource locality	To improve system performance, processes may be migrated to nodes that are closer to the data and resources used by the processes.
Improve maintenance	By migrating processes to another node, the original node can be deactivated for maintenance tasks without having to stop the processes, which are hosted on that node.

Process migration refers to the procedure of transferring a process, i.e., the executing instance of a computer program, between two machines during its execution. It is therefore said to be migrated from the *source* to the *destination node*. A process consists of components like memory regions for data and code (e.g., stack, heap, etc.) that need to be taken into consideration when it is thought about migration. On the destination node a new process instance is created, which is referred to as *destination instance*. The process on the source node that still exists during migration is called the *source instance*. This terminology is also used throughout the thesis at hand when referring to the migration of composite service instances.

The algorithm for migrating processes is generalized and summarized in [85] as follows.

1. A migration request is sent to a remote node.
2. The migrating process a detached from source node.
3. Messages to the process are queued up and delivered after the process is completely migrated.
4. The state of the process is extracted.
5. An instance of the destination process is created.
6. The state is transferred and imported into the destination instance.
7. References to the process are forwarded to the new location.
8. The new process instance is resuming execution.

While this algorithm considers processes as the entity of migration, composite service instances are the targets of interest in the thesis at hand. The migration algorithm used within this thesis is therefore designed to be compatible with composite service instances. The approach used to transfer the state of composite service instances is based on the *eager (all)* strategy which extracts the state of a process and transfers it to the destination node as a whole. Systems like LSF [119] and Condor [66] implement this strategy in their *checkpoint-restart* mechanisms [66]. In contrast to *eager*, the *copy on reference* strategy transfers data not until it is referenced by the destination instance. While the latter approach has a lower initial migration cost than the fist one, the costs are increasing during runtime.

3.3.2 Virtual Machine Migration

Apart from process migration, the need for migrating VMs between physical nodes was emerging with the raising importance of virtualization and the transition from common data centers into Virtualized Data Centers (VDCs). Live migration, i.e., migrating VMs without downtime and loss of network connectivity, was topic of research [90], [25] and is meanwhile state of the art, since it is supported by virtualization software from vendors like VMWare, Red Hat and Microsoft. Migration enables the redistribution of VMs among physical machines, e.g., to ensure that VMs receive enough processing power from the underlying physical resources and to optimally utilize physical resources or to provide maintenance windows for the physical hardware.

[59] and [19] formulate the challenge of optimally placing VMs on physical nodes as a specific form of the *bin packing problem* [26] and propose approximation algorithms. Their goal is to fulfill SLAs by detecting VMs that cannot allocate enough resources to their applications and migrate them from overloaded physical nodes to lesser loaded nodes. The optimal destination node of a VM is decided by approximation algorithms. While the algorithm presented in [59] resolves SLA violations reported by VMs, the algorithm of [19] is used in combination with workload forecasting and therefore enables preventive migrations of VMs.

Live migration of VMs is used in [105] to accomplish autonomic adaptations on so called *VIOLIN* systems [54], a special kind of virtual distributed computation environments. VIOLINs are isolated from each other and logically separated from the underlying infrastructure. They are composed of VMs which are interconnected by a virtual network. The central components in the approach presented in [105] are the *adaptation manager* and *virtual machine monitors*. The virtual machine monitors are deployed to each physical host and provide information regarding the utilization and availability of CPU and memory. Based on this information, the adaptation manager autonomically decides whether adaptations are necessary or not. It adapts the allocation of available resources to the virtual environments by dictating the monitoring system on the physical host. The system supports mechanisms for local, as well as multi-domain adaptations. Local adaptations provide fine-grained control over the CPU and memory allocations for each VM. Multi-domain adaptations refer to the migration of single VMs or the whole virtual environment to remote physical nodes. In contrast to [59] and [19], the authors of [105] do not focus on optimizing the overall allocation of resources to VMs. Instead, their approach incrementally increases the performance of the whole system. If a VM demands resource allocation above its current allocation, the reallocation policy will attempt to assign appropriate resources to the VM. If this demand can not be satisfied locally, the VM or the whole VIOLIN will be migrated to other

nodes. [105] uses the capabilities provided by the virtualization platform for memory ballooning and CPU weighting to increase or decrease the resource allocations of VMs. The virtual environment and adaptations are directly based on the virtualization platform (i.e., XEN). In contrast, Clouds abstract from this virtualization layer and Cloud consumers do not have access to those functionalities.

The concepts discussed so far cannot upgrade the resources of VMs above their preconfigured amount. A VM created and started with 2 GB of memory and 2 CPUs cannot exceed this limit regardless of how many resources are available on the physical node.

The authors of [45] propose a model driven approach to a self-adaptive VDC that autonomically protects its SLAs. They refer to Virtual Execution Environments (VEEs) (e.g., VMs) that provide *elementary* services that in turn can be combined and provided to users. Their focal idea is an *adaptation controller* that implements the closed control-loop of autonomic computing (Section 2.3) and controls the VDC.

They assume that the adaptation controller can perform the assignment of additional resources to a VEE (e.g., add additional disk storage), migrate a VEE (e.g., a web server) from a physical server to a cluster, add or remove VEEs and balance the load among the components (e.g., use server A one time and use video server B the other time).

Those adaptation actions are trigger depending on the factors pictured by the following models:

- Workload models describe the amount of requests during a period of time, consider peak loads, normal usage and trends of increasing and decreasing load.
- Service composition models describe workflows and provide information about the next expected service invocation.
- Architectural models express dependencies and interactions between components that realize elementary services.
- VEE allocation models express the allocation of VEEs to physical nodes and the resource demand of VEE on their hosting node.
- Physical resource models represent resources that are available within the VDC, as well as their ability of being changed and allocated.

While their approach considers the assignment of additional resources to VEEs, this assignment is not trivial in practice. VMs cannot upgrade their resources like memory and CPU during runtime.

While the approach considers the migration of VEEs between physical nodes in the VDC, as well as the instantiation of new VEEs and components, no solution is presented that shows how the migration of VEEs may actually be accomplished. Live migration of VEEs, that are represented by VMs, is supported by XEN and VMWare. In contrast, no solution is provided that considers the migration of composite services between VMs.

3.3.3 Service Migration

In [44], the authors consider *service migration* as an extended concept to process migration. In contrast to process migration the concept of service migration is not only concerned with just moving the execution context of a process, but instead enabling the reconstruction of a whole execution environment on a remote node. The work presented in [44] is based on a mobile extension of a Distributed Shared Array (DSA) runtime system [16] and enables the migration of the Java Virtual Machine (JVM) runtime data structures used by a DSA service. The contents of the heap and the JVM stack are extracted and serialized into a platform independent data structure, called the *service pack*. The class files are instead prefetched by the remote node from a lightly loaded neighboring node, located in the same subnet. This enables the transfer of data (i.e., the service pack) and classes in parallel.

The migration procedure proceeds as follows.

1. If the *LoadMonitor* component detects an overloaded or unavailable node, it will send a request to the *migration coordinator*.
2. At synchronization points, the *migration coordinator* is requested to check if migration request is active.
3. The migration coordinator sends the migration request to the *bootstrap daemon* at the destination node.
4. The bootstrap daemon at the destination node contacts the bootstrap daemon at the source node.
5. The data (i.e., service pack) is transmitted to the destination node.
6. The destination node loads the data.
7. An acknowledgment is sent to the coordinator.

This approach enables the reconstruction of a service runtime onto a remote node without having to migrate the whole VM and provides the ability to migrate to a VM that is equipped with a larger amount of resources. The migrating entity in this approach is a DSA service and the loaded class objects are downloaded from a neighboring node that is part of the distributed system. In contrast, the thesis at hand focuses on the migration of composite service instances, but uses a similar approach to deploy the runtime code. Additionally, it discusses the deployment of destination nodes for dynamic resources provisioning in the Cloud.

CHAPTER 4

Background

First, this chapter provides the background to understand Windows Workflow Foundation (WF) [20] and VRESCo, as they are the key software components used in this thesis. WF is a programming framework for the development of workflow-based applications and provides the ability to build composite services. VRESCo is a runtime environment for SoC that serves as a Web service registry, enables dynamic service invocation and employs eventing facilities. Finally, this chapter presents the idea of dynamic scaling and explains the difference between horizontal and vertical scaling.

4.1 Windows Workflow Foundation

The concept of composing Web services to composite services, as well as the WSBPEL technology were already presented in Section 2.1.2. WSBPEL is a platform independent standard for specifying processes out of Web services. WSBPEL processes are executed by WSBPEL engines, such as ApacheODE [11] or ActiveVOS [1]. Another way of building composite services is provided by WF. WF is a component of Microsoft .NET 3.5 [65] allowing the development of workflow-based applications. Composite services are one example of such applications. WF is used in this thesis to express and run composite services.

WF supports two different styles of workflow authoring. *State Machine Workflows* are authored equally to state machines. They are composed of states that transition to other states on certain occurrences. The entry to the workflow is the *start state*. As soon as the workflow reaches the *final state,* the workflow has completed. The sequence of state transitions is controlled by external events and is not defined by a fixed flow of control within the workflow. State machine workflows are therefore well suited for problems that involve human interactions. *Sequential Workflows* are authored to describe a fixed series of activities that will be executed sequentially. They are useful for repetitive, predictable operations that are always the same. Common control flow structures, such as loops and conditional branches are only supported by sequential workflows [20]. The following discussions on WF are exclusively dedicated to sequential workflows, as state machine workflows were not used in this thesis.

Developers can use the *Workflow Designer* to develop their workflows. The workflow designer is a graphical editor part of Microsoft Visual Studio [100]. It is basically operated by dragging and dropping activities from a toolbox into the modeling area. The graphical definition of the workflow is then translated into the code that defines the workflow. Additionally, developers are able to define a workflow by writing the code manually, without using the workflow designer.

4.1.1 Activities

Sequential workflows are built from activities. An activity is an encapsulated entity in the sequence of a workflow of steps performing certain actions. WF supports more than thirty different types of activities that each serves an individual purpose. Activities are represented by the `Activity` class within WF, whereof all types of activities are derived from. The spectrum of activity types ranges from activities that represent flow control constructs, such as loops or conditional branches, to event-driven activities, or custom activities. The most important activities are described in the following.

- `IfElseActivity` is used to model conditional branches, such as the if-else construct of common programming languages.
- `WhileActivity` allows to define loops that are executed until a certain condition occurs.
- `ParallelActivity` allows to execute multiple sequences of activities concurrently.
- `CodeActivity` allows to implement individual programming logic which is executed when the activity is invoked.
- `SuspendActivity` suspends the execution of a workflow. Resumption has to be invoked explicitly.
- `InvokeWebServiceActivity` invokes a Web service by the use of a proxy class.
- `SequenceActivity` is a composite activity, composed of other activities, which are executed in a specified order.

A special kind of activity are *Custom Activities*. They are similar to `CodeActivity` activities in so far as they allow to execute custom code. But in contrast to a `CodeActivity`, they are more versatile and more complex. Their code is decoupled from the workflow and implemented within self-contained objects with clearly defined inputs and outputs. While a `CodeActivity` is bound to one workflow, a custom activity can be reused throughout different workflows. As they do not depend on one specific workflow, they can be tested independently and are suitable for unit testing. In their basic form, custom activities are derived from the `Activity` class. They may as well be derived from `SequenceActivity` to implement custom composite activities. Custom composite services can in turn execute a sequence of other activities, custom activities or even custom composite activities.

4.1.2 Workflow Runtime and Services

The `WorkflowRuntime` is the central element of WF. It is responsible for executing workflow instances and manages their life-cycle. Workflow instances are represented by `WorkflowInstance` objects that are created by the `WorkflowRuntime`. `WorkflowInstance` objects are accessible to the developer and can be used to control the workflow instance, e.g., to stop, to suspend or to resume it.

The `WorkflowRuntime` utilizes so called *Runtime Services* that are selected by the developer to customize the behavior of the runtime. WF ships with ready to use runtime services that provide means to persist workflow instances into a relational data base or to track their execution with the Structured Query Language (SQL). Developers are able to implement custom runtime services that can be used by the `WorkflowRuntime` for workflow persistence and tracking. The various types of runtime services are represented by abstract classes that do not provide any implementation. To use them, they have to be derived and all of their operations need to be implemented and overridden. WF also supports the creation of completely new runtime services that serve individual purposes. They can be used explicitly from within *Code Activities* or *Custom Activities*. The implementation of runtime services and the usage of a custom runtime service will be shown in detail in Section 5.2.

4.1.3 Workflow Tracking

WF provides facilities to track the life-cycle of workflows, for example the creation or completion of a workflow. It also reports the execution states of activities, such as activity initialization or activity compensation. Tracking of events is accomplished by registering a tracking service at the `WorkflowRuntime`. The various workflow events that are reported by WF are listed and described in Table 3. Table 4 shows the execution states of activities that WF is able to track.

Developers can track their code with custom events with the `UserEventData` class, e.g., from within custom activities.

4.1.4 Workflow Hosting

Hosting of workflows as composite services is accomplished by using Windows Communication Foundation (WCF) [21]. Two scenarios are possible for hosting workflows. Workflows can either be hosted with Microsoft IIS [83] or by self-hosting. The way of how composite services are hosted in this thesis is described in Section 5.2.

4.2 VRESCo - Vienna Runtime Environment for Service-oriented Computing

VRESCo [52] is a runtime environment for SoC. The motivation behind its development was to recover the *publish-find-bind triangle* of SoC that is said to be *broken* because service registries are rarely used in real-world scenarios and binding of clients to services happens at design-time. [72] VRESCo picks up those neglected concepts and addresses the current challenges

Event	Description
Created	A workflow instance was created
Started	A Workflow instance starts executing
Idle	Execution is delayed until the occurrence of a specific external event, e.g., a timer, a message, etc.
Unloaded	A workflow was unloaded from memory and persisted into a data store
Loaded	A workflow was loaded into memory from a data store
Completed	Execution of a workflow has finished
Persisted	A workflow instance was persisted (but may still be executing)
Suspended	Execution was suspended manually
Resumed	Execution was resumed from suspension
Exception	An exception was raised within a workflow instance
Terminated	A workflow instance was terminated; persisted and in-memory state was cleared
Aborted	A workflow instance was aborted
Changed	A workflow instance was changed

Table 3: Tracking Events for Workflows in WF

Event	Description
Initialized	An activity was initialized
Executing	An activity is executing
Compensating	The actions of an activity are being rolled back, i.e., reverted
Canceling	An activity is being canceled
Closed	Execution of an activity has finished
Faulting	An error occurred in the activity

Table 4: Execution States of Activities in WF

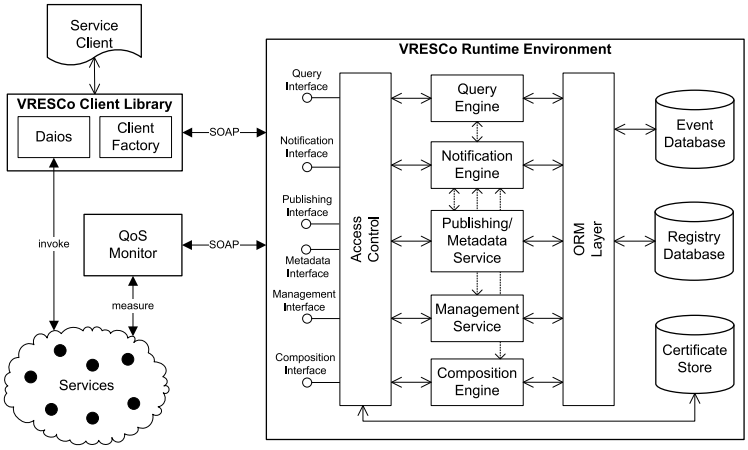


Figure 5: VRESCo Runtime Environment Architecture (from [75])

of SoC that especially encompass the management of metadata in service registries, as well as dynamic service binding and invocation.

VRESCo is implemented in C# [106]. Its subsystems are exposed as Web services and hosted with WCF. A client library for VRESCo exists that can be used to access the VRESCo runtime services. The messaging protocol SOAP which was briefly discussed in Section 2.1.1, is used by VRESCo to communicate with clients.

The architecture of VRESCo is shown in Figure 5 and comprised of the following subsystems.

- The *Publishing and Metadata Service* allows for the versioning of services and stores functional, as well as non-functional (QoS) attributes of Web services.

- The *Management Service* provides an interface to manage users and their permissions within the runtime.

- The *Query Engine* allows to search for stored entities within the VRESCo runtime by filtering for attributes or by using a full text query.

- The *Notification Engine* provides means for the notification of clients in a publish-subscribe style. Clients can subscribe for certain events and will be notified if they occur. The usage of the VRESCo notification engine within this thesis is described in Section 5.2.

VRESCo provides capabilities for the dynamic selection and invocation of services and service endpoints, based on QoS attributes. QoS attributes are reported by a standalone component, the *QoS Monitor*. The QoS Monitor constantly monitors target Web services and reports the measured attributes to the publishing service of VRESCo. A detailed discussion on the QoS capabilities of VRESCo can be found in [74]. VRESCo accomplishes the dynamic invocation of

Web services with the *Daios framework* [63]. Daios is capable of binding to Web services at runtime by using a protocol independent and message oriented approach. A detailed presentation on Daios can be found in [63].

4.2.1 Metadata and Service Model

VRESCo specifies two data models for the description of Web services. On the one hand VRESCo specifies a *metadata model* for the description of abstract functionalities offered by Web services. The central elements of the metadata model are `categories`, `features` and `data concepts`. `Categories` are used to group, i.e., categorize, Web services by their intended purposes or use. `Features` are concrete actions that are implemented within a system and associated to a `category`. `Data concepts` define the data types that are used within a system. On the other hand a *service model* is defined in VRESCo. It allows for the description of concrete service manifestations and QoS attributes of Web services. Within the service model, a Web services is represented by a `service` entity that is associated to certain `categories` and one or more `revisions`. The `revisions` describe the concrete versions of a Web service that implement a set of `operations` that may in turn require several `parameters`. `Parameters` are mapped to the `data concepts` specified within the metadata model and `operations` are concrete implementations of `features`. A detailed illustration of the data models used in VRESCo can be found in [103] and [52].

4.2.2 VRESCo Eventing

The notification engine of VRESCo utilizes NEsper [34] for the processing of events. NEsper is a customized version of Esper for Microsoft .NET. Subscriptions are realized in Esper by the registration of listeners at the runtime, together with queries that specify the events that will be filtered. A listener is invoked by the runtime on the occurrence of an event that matches the associated query. Queries are defined with the EPL. EPL is similar to the SQL, but designed to use event streams as its source of data, instead of relational databases.

The architecture of the VRESCo notification engine is illustrated in Figure 6. Clients use the *subscription interface* of the notification engine to subscribe for notifications. They pass an EPL query, a notification endpoint and an expiration date to the *subscription manager*. An Esper listener is then created by VRESCo and registered together with the query at the NEsper engine. On event occurrence, the *notification manager* extracts the information on the event from the listener and notifies subscribed clients by publishing the event at the declared endpoints. Furthermore, subscriptions are removed by the notification manager on expiration. VRESCo supports WS-Eventing which was described in Section 2.4 and which is used in this thesis together with VRESCo for event notifications. The utilization of the *eventing service*, that provides clients with the ability to report the occurrence of certain events, will be shown in Section 5.2. A detailed discussion on the VRESCo notification engine is provided in [73].

VRESCo implements a hierarchical event type model, similar to the approach that was shown in Section 3.2. The base event of VRESCo's event model is the `VRESCoEvent` class, whereof all other event types inherit from. The event model is extended as part of this thesis

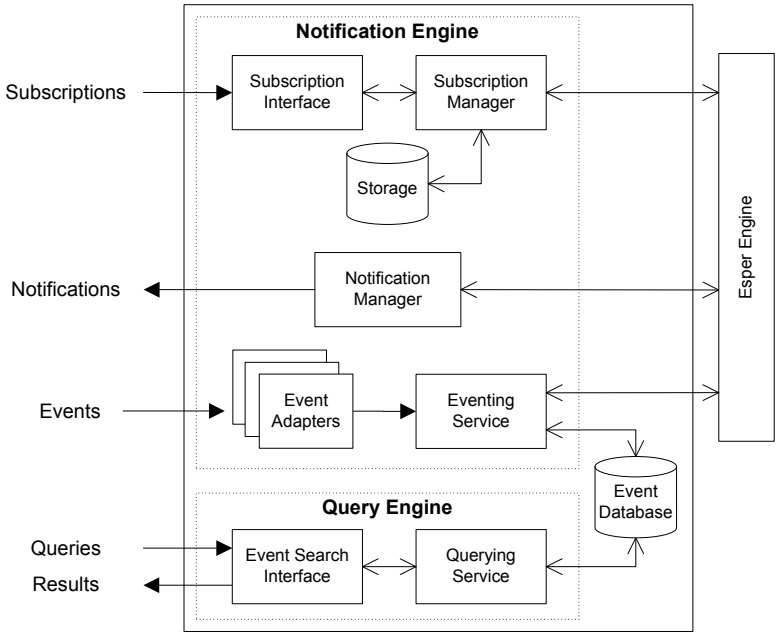


Figure 6: VRESCo Eventing Architecture (from [73])

to reflect specific occurrences in the composite service hosting infrastructure and to accomplish communication between infrastructural components that do not know of each other.

VRESCo allows events and their attributes to be persisted and stored in the event database, together with the date of their occurrence. This is especially useful for retrospective evaluations and the reconstruction of event sequences. The evaluations in Chapter 6 are based on events that were persisted during test runs.

4.3 Load Scaling

Load scaling a system is the ability to change its size in accordance to increasing or decreasing load (e.g., amount of requests from clients to a web server). Two distinct approaches to load scaling can be differentiated, *vertical* and *horizontal* scaling [109]. Horizontal scaling deals with adding or removing nodes to or from an already existing cluster of nodes which together constitute a system that performs a task or provides a service. The task is thereby split into separate parts, each being handled by a different node. To split the load of a service, provided over a network, e.g., the Internet, requests are distributed among different nodes. This process may be achieved by using a component (i.e., a *load balancer*) that dispatches requests and forwards them to nodes within the cluster based on a scheduling policy [23]. Figure 7 shows a scenario of horizontal scaling where a node is joining a cluster of web servers because two nodes are overloaded with requests.

Vertical scaling, in contrast, deals with upgrading the amount of resources (e.g., CPUs or memory) of a single node. Scaling a system vertically depends on the Operating System (OS) and

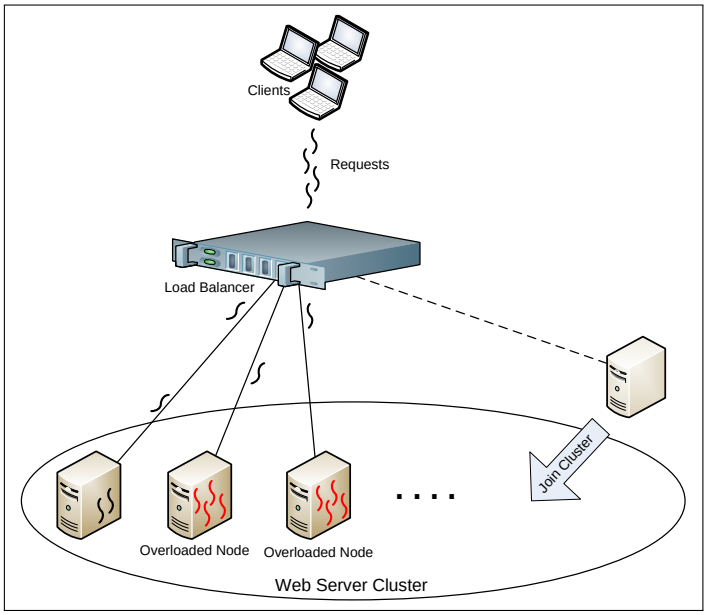


Figure 7: Scaling a Cluster of Web Servers Horizontally

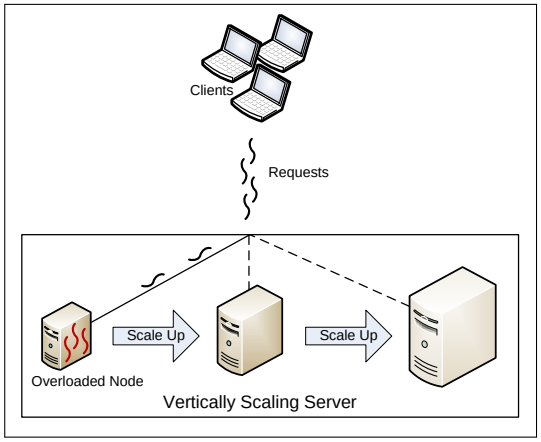


Figure 8: Scaling a Server Vertically

the hardware platform to support the installation of components during runtime. Otherwise it is necessary to shutdown the system, install additional resources and start the system again, which results in unavailability of the system. The approach to vertically scale a server is illustrated in Figure 8.

Finally, both approaches, vertical and horizontal scaling, may as well be combined as shown in Figure 9.

Virtualization and Cloud computing provide capabilities to scale systems dynamically. Amazon Auto Scaling [8] is a Cloud service, able to add and remove VMs automatically to or from

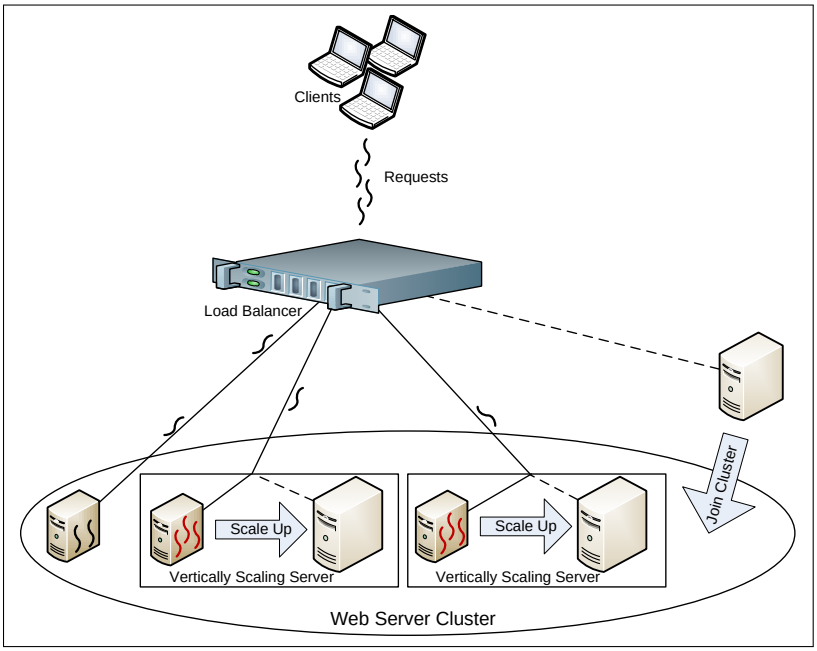


Figure 9: Combining Vertical and Horizontal Scaling

a load balancer. To use Amazon Auto Scaling, the user defines a threshold for a certain metric, such as CPU consumption and selects a type of VMs. Amazon Auto Scaling then creates VMs and registers them with a load balancer, based on monitoring the specified metric and threshold. As discussed in [67], load balancers are subject to limitations in network bandwidth and are vulnerable of being overloaded when too many nodes need to be served.

Dynamic assignment and removal of virtual resources to and from VM instances would eliminate the need for installing physical hardware components manually. Although, Clouds and their handling of virtual resources provide the base for automating vertical scaling, Cloud platforms, with Amazon EC2 and Eucalyptus leading the way, currently do not support changing the amount of resources of a VM. Vertical scaling of a VM instance is therefore limited to replacing this VM in way that will be shown in the following sections.

CHAPTER 5

Design & Implementation

This chapter presents the concept of an autonomic composite service runtime, hosted in the Cloud and able to extend its own resources during runtime as needed by evaluating SLAs. Event-driven monitoring is used to keep track of composite service executions and the VRESCo runtime environment, which was described in Section 4.2, is utilized for event notifications and event persistence.

To begin with, the architecture of the autonomic runtime and its capabilities are discussed. Then the implementation of a prototype is explained and finally is shown how this system is deployed to the Cloud.

5.1 Runtime Architecture

The composite service runtime presented in this section, can decide by itself when it is necessary to provision resources and it executes this step without human interaction. To accomplish this autonomic behavior, the architecture of the runtime is based on the closed control loop of autonomic systems (*monitor*, *analyze*, *plan* and *execute*) and the idea of distinguishing between the roles, or tasks, of an *autonomic manager* and a *managed element*. These concepts were discussed in Section 2.3. The composite service runtime, which is representing the autonomic manager, is therefore called *Autonomic Composite Service Runtime*, or shorter *Autonomic Runtime*, throughout this thesis. The managed element is represented by the composite service itself. The closed loop of the autonomic runtime is as follows:

1. Execution of composite services instances, as a result of clients invoking the composite service

2. Monitoring of executing composite service instances and the autonomic runtime

3. SLA evaluation by analyzing monitored data and reporting SLA violations

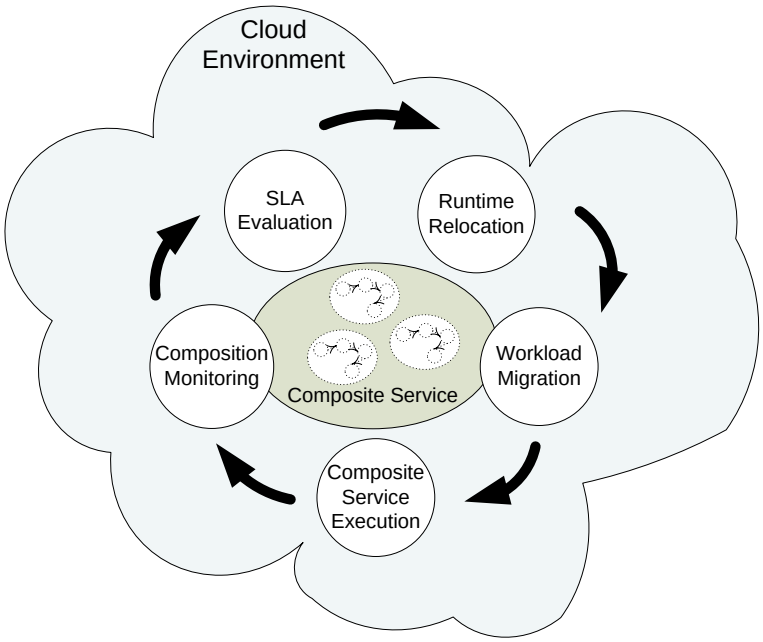


Figure 10: Autonomic Runtime Control Loop

4. Relocation of the autonomic runtime to a dynamically created virtual machine instance equipped with additional resources

5. Workload migration, i.e., transfer of currently executing instances of a composite service to the relocated autonomic runtime

Figure 10 shows the composite service and its executing instances, as well as the loop of tasks of the autonomic composite service runtime.

Two concepts, implemented by the autonomic runtime, are distinguished, *relocation* of the autonomic runtime itself and *migration* of composite service instances. The fundamental difference is that the autonomic runtime is actually duplicated and hosted on two virtual machine instances for a short period of time, as illustrated in Section 5.2, while composite service instances are removed from one autonomic runtime and transferred to the other.

The procedure of resource provisioning can therefore be separated into two phases, the *Runtime Relocation Phase* and the *Instance Migration Phase*. The *Runtime Relocation Phase* lasts from the moment the decision to provision resources was made, to the moment the newly created machine instance has loaded the autonomic runtime and serves composite service requests. The *Instance Migration Phase* lasts from the moment the migration of composite service instances was started until the migration of the last composite service instance has finished.

As discussed in Section 2.2, resources in Eucalyptus, as well as Amazon EC2, are represented by instance types which encompass different amounts of computing resources. The instance type of a VM is specified at the time the VM is created and can not be changed afterwards. Because

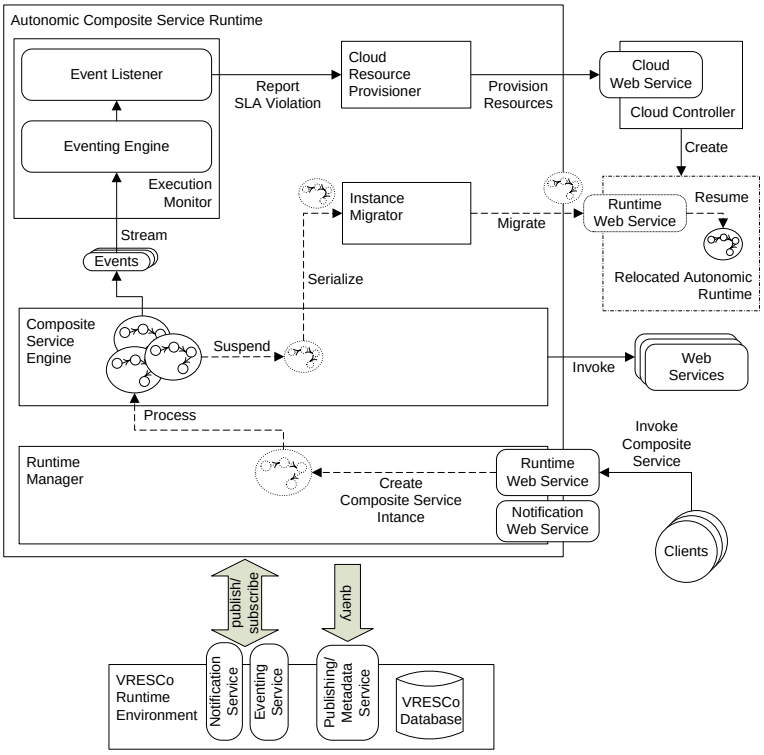


Figure 11: Autonomic Composite Service Runtime Architecture

of this, provisioning further resources can only be accomplished by creating an additional VM instance, either replacing or just supporting the original machine.

Provisioning resources to a system in the Cloud by scaling vertically makes it necessary to migrate or relocate the system between VM instances. The idea of relocating a composite service runtime is similar to the idea of relocating whole VMs among physical nodes which was discussed in Section 3.3. The vertical, as well as the horizontal approach to scaling, can be combined with taking over the whole or part of the current workload from the original VM instance to the new one. Papers, proposing horizontal scaling by utilizing load balancing, were discussed in Section 3.2. However, none of them considers the migration of the workload (i.e, the currently executing instances of composite services).

5.1.1 Runtime Overview

The architecture of the autonomic composite service runtime is shown in Figure 11. It is composed of the following components:

- The central component of the system is the *Runtime Manager* that orchestrates all other components and initializes the system.

- The *Composite Service Engine* is responsible for the execution of composite services, keeps track of running composite services and provides an interface for the communication with executing composite service instances.

- The *Runtime Monitor* receives event notifications about the execution states of composite services. It keeps track of SLA conformance and reports when the system is unable to ensure SLA conformance any longer.

- The *Cloud Resource Provisioner* communicates with the Cloud platform and uses the interfaces provided by the Cloud to provision additional resources.

- The *Instance Migrator* is responsible for the migration or relocation of the composite service runtime.

- The *Runtime Web Service* receives request from clients that invoke the composite service.

- The *Notification Web Service* receives notifications from VRESCo.

VRESCo acts as a central, static point of communication for the autonomic runtime and serves as a Web service registry. As will be discussed later on, the VRESCo endpoint is known to the Cloud platform and the autonomic runtime can query the Cloud for the exact location of the VRESCo runtime environment.

The CLC represents the single interface to the Cloud. It is used by the autonomic runtime to provision resources and to query for information about the surrounding infrastructure. Communication to the CLC is based on SOAP. A SDK from the Cloud provider can be used that encapsulates the concrete communication mechanism.

Composite service instances dynamically invoke Web services whose endpoints are stored centrally within the service registry of VRESCo. Their endpoints do not need to be statically provided to the composite service and it does not matter whether they are in or outside of the Cloud platform.

Resource provisioning is established through utilizing the capabilities that are provided by the Cloud platform. Clouds usually provide a SOAP based interface, as well as a SDK which enables the implementation of automatic interactions with the Cloud within software applications. Eucalyptus' endpoint for interactions is represented by the CLC. The CLC provides a Web service that is compatible with AWS. Therefore the AWS SDK is usable with Eucalyptus. It provides operations to create and terminate machine instances, to configure their networking attributes, as well as to query the Cloud for certain information.

To inform two different instances of the autonomic runtime about each other, a publish-subscribe mechanism is used. On resource provisioning, the autonomic runtime subscribes its *Notification Web Service* to the event that represents the successful launch of a relocated autonomic runtime. On startup, the relocated autonomic runtime publishes its successful launch to VRESCo, which in turn notifies the original autonomic runtime through the *Notification Web Service*.

5.1.2 Composite Service Execution

The autonomic runtime is represented to the outside by the *Runtime Web Service*, which provides a SOAP based interface that clients use to invoke the composite service. Each invocation from a client triggers the creation of a composite service instance by the *Composite Service Engine*. The *Composite Service Engine* needs to keep track of all composite service instances currently executing and it is able to communicate with them. A composite service instance is represented by an in-memory object that provides an interface for controlling the executing instance. After the creation of a composite service instance, the runtime stores a reference to the instance, which is removed when the instance has completed. Each composite service instance has to run in a separate thread to provide a high level of parallelism and to be able to utilize all available CPUs.

5.1.3 Composition Monitoring and SLA Evaluation

The autonomic runtime performs actions based on the evaluation of SLAs. To evaluate them, it constantly monitors its executing composite service instances. It decides grabbing additional resources from the Cloud, as soon as SLA conformance can no longer be established with the currently allocated resources.

Monitoring is performed by the *Runtime Monitor* component. It tracks the execution of each composite service instance and measures performance attributes of the runtime, which are relevant for SLA evaluation (e.g., execution duration of composite service executions or Web service invocations, amount of Web service invocations per second, etc.). To track composite service instances, the *Runtime Monitor* relies on the *Composite Service Engine* to publish the occurrence of events that are related to the composite service. The events that are reported and tracked depend on the metrics needed for evaluation. The execution duration of composite service instances can be tracked by notifying about the creation and the completion of a composite service instance. Web service invocations can be measured by publishing an event right before the request to the Web service and right after the response.

The central component of the *Runtime Monitor* is the *Event Processing Engine*, which is used for CEP. *Event Listeners*, that are registered with the *Event Processing Engine*, handle the events and evaluate the SLAs. The *Runtime Monitor* notifies the *Cloud Resource Provisioner* whether the runtime is unable to satisfy the SLAs at the current workload. The *Cloud Resource Provisioner* determines the resources that are needed and contacts the Cloud to provision these to the runtime.

Besides event-based tracking and monitoring of composite services, events are used by the autonomic runtime to track its own life-cycle. Table 5 lists and describes the states that were defined for this purpose. To identify the source of these events, they were attached with information about the originating autonomic runtime, such as the IP address of the hosting VM instance and an unique identifier of the composite service engine. The state `RuntimeStarted` is published via the VRESCo eventing service and used to inform the autonomic runtime through the *Notification Web Service* about its relocation. All of these events are persisted by VRESCo, together with their attributes, and used for the evaluation in Chapter 6.

State	Description
RuntimeStarted	Published as soon as an autonomic runtime has started and is ready to serve invocation of to the composite service. It may also indicate the completion of the *Runtime Relocation Phase*
RuntimeTerminated	Reports that the autonomic runtime and its hosting VM instance were terminated
ProvisioningResources	The provisioning of resources was triggered and the *Runtime Relocation Phase* has started
MigrationStarted	Reports the beginning of the *InstanceMigrationPhase* which implies that the migration of composite service instances has started
MigrationFinished	Published when all composite service instance were migrated and the *InstanceMigrationPhase* has finished

Table 5: Autonomic Runtime Life-Cycle Events

5.1.4 Runtime Relocation

The thesis at hand considers resource provisioning in terms of scaling the runtime vertically by relocating itself within the Cloud. Relocation is thereby based on four main concepts.

Central deployment Firstly, the code of the autonomic runtime, together with the parameters, are stored centrally and are available for automatic deployment during the phase of resource provisioning. When a VM instance has started, an initialization procedure, the *Autonomic Runtime Loader*, which will be described in detail in Section 5.3, automatically locates the repository where the autonomic runtime is stored, then downloads and launches it. The autonomic runtime is packaged together with its parameters and stored on the host that serves the VRESCo runtime environment.

Machine images Machine images serve as templates for the VMs, which will host the autonomic runtime. They contain an OS and any software components that were deployed to the OS when the image was created. The machine image that hosts the runtime contains the *Autonomic Runtime Loader*, which is executed automatically when the OS has started. The concept of machine images is called *Eucalyptus Machine Images (EMIs)* within Eucalyptus and *AMIs* within Amazon EC2. As part of this thesis, various EMIs were created to deploy the autonomic runtime and its environment to Eucalyptus. They are presented in Section 5.3.

Instance types Instance types specify the amount of provisioned resources. The autonomic runtime differentiates between three instance types (*m1.medium*, *m1.large* and *c1.xlarge*). It does not know about their individual hardware specifications, but only about their ranking. *c1.xlarge* provides more resources than *m1.large*, and *m1.large* provides more resources than *m1.medium*. If the decision was

made to provision resources, the autonomic runtime figures out the currently used instance type and then chooses the next higher ranked instance type for relocating itself. The instance types that are pre-configured in Eucalyptus can be customized as needed. Customizable attributes of an instance type are the number of virtual CPUs, the amount of memory and disk space. Eucalyptus uses the term *cores* to refer to the CPU attribute of an instance type, but they are presented as separate CPU *sockets* rather than CPU *cores* to the operating system of a VM instance. Therefore the operating systems needs to support the actual amount of CPU sockets. Operating systems such as Windows Server 2003 or Windows XP are available in different versions that support just a specific amount of CPU sockets [79].

Dynamic IP allocation To relocate the autonomic runtime from one host to another, transparent for any clients, a public IP address is used that can be allocated, attached to and released from a VM instance as needed. This feature is equivalent to the *Elastic IP* of EC2. As soon as this IP address is released from the old VM instance and attached to the new one, all client requests will be received by the relocated autonomic runtime.

The procedure that implements resource provisioning in the prototype, based on the concepts illustrated above, is shown in Figure 12. As the original and the relocated autonomic runtime do not know each other, they utilize the VRESCo runtime environment as a central endpoint for communication. WS-Eventing is used by VRESCo to notify the original runtime.

1. The autonomic runtime subscribes to the *RuntimeStarted* event at the VRESCo notification service and launches the *Notification Web Service*.
2. The next higher ranked instance type is chosen by the runtime and AWS SDK is used to launch a new VM instance of that type.
3. Eucalyptus creates, initializes and boots the VM instance. During this step the execution of composite services by the runtime is not affected.
4. The new machine instance loads the autonomic runtime.
5. The newly loaded runtime fetches the public IP address, which is released from the source VM instance and associated with the destination VM instance. The original runtime will be called *source runtime* and the newly loaded runtime *relocated runtime* from now on.
6. From now on the relocated runtime serves composite service requests.
7. The relocated autonomic runtime publishes the *RuntimeStarted* event using the VRESCo eventing service. VRESCo notifies the source autonomic runtime.
8. Either all executing composite service instances are suspended, serialized and then transferred to the relocated runtime at its *Runtime Web Service* or they are processed by the original runtime until their completion.

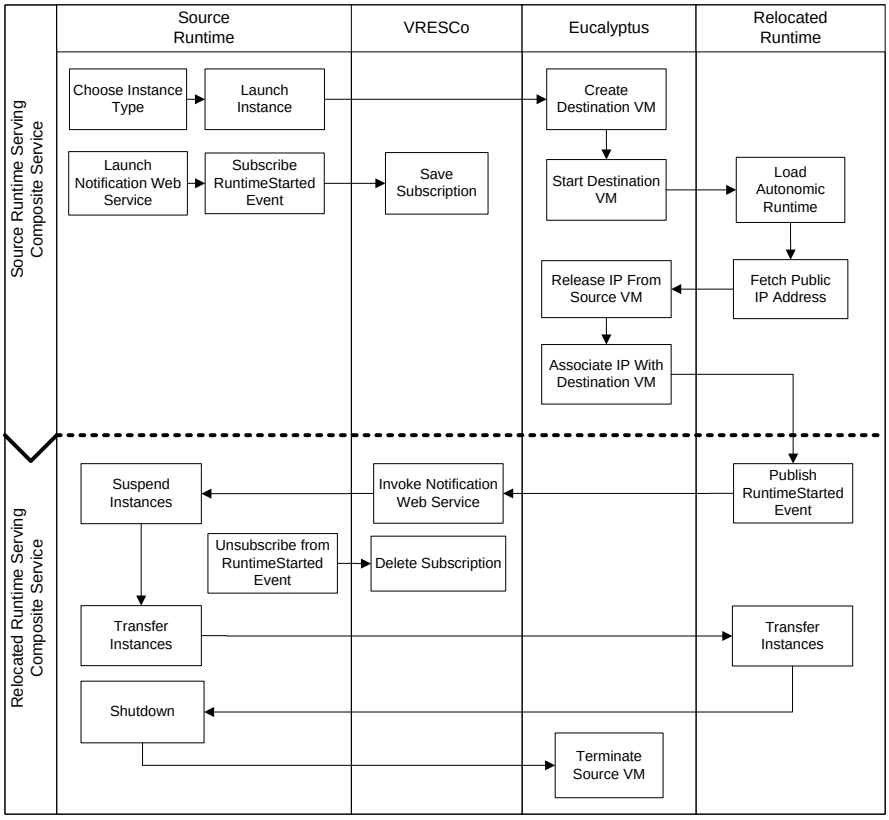


Figure 12: Resource Provisioning Procedure

9. The source runtime shuts itself down and terminated its hosting VM instance.

5.1.5 Workload Migration

Provisioning of resources by the autonomic runtime is accomplished by relocating itself onto a dynamically created VM instance that is equipped with more capacity for the execution of composite service instances. The relocation can either include the migration of the current workload, which is represented by currently executing composite service instances or leave the composite service instances on their original VM and let them finish there. The decision has to be based on several factors:

- Migration of composite service instances uses additional resources of the destination machine, as well as on the original machine.
- Composite service instances have to be suspended for a short period of time which makes them more likely to violate a SLA.
- Operation of multiple VM instances in parallel can lead to increasing financial costs (e.g., pay usage cost for resources, licenses, etc.).

Migration takes place by suspending a composite service instance and serializing its current state. This state is then transferred to the destination machine where the state will be deserialized and the composite service instance resumed. The *Instance Migrator* takes care of this procedure by fetching the executing composite service instances from the local *Composite Service Engine* and subsequently resuming them at the *Composite Service Engine* on the destination machine by invoking the remote *Runtime Web Service*.

5.2 Prototype Implementation

This section provides a detailed presentation of the prototype and its implementation issues. The prototype makes use of the VRESCo runtime services, Esper and Microsoft WF. For the communication with the Cloud platform it utilizes the AWS SDK, which is compatible with Eucalyptus.

5.2.1 Composite Service Hosting

The prototype implemented as part of this thesis is based on Microsoft .NET and handles composite services, built out of Windows workflows, using Microsoft WF technology. As discussed in Section 4.1, workflows in WF are represented by `WorkflowInstance` objects. The prototype implements interaction with `WorkflowInstances` by using a wrapper that encapsulates the WF specifics of handling workflows.

The *Runtime Web Service* is implemented using WCF, so *Operation Contracts* were necessary to specify the operations of the Web service. The *Runtime Web Service* provides two operations, `startWorkflow()` and `resumeWorkflow()`. An invocation of the `startWorkflow()` operation starts the execution of a composite service instance. The `resumeWorkflow()` operation differs from the `startWorkflow()` method as it does not start the composite service, but resumes a previously suspended composite service from a specified state.

The signatures of these operations are as follows:

```
[OperationContract]
Guid startWorkflow(String type, string user, string pwd);
```

`startWorkflow()` takes the type of the workflow (or composite service), the user ID and a password as parameters. It starts the workflow of the given type and uses the ID of the user, as well as the password to authenticate the client. The Globally Unique Identifier (GUID) of the started workflow instance is then returned to the caller. This ID, which is created by the *WorkflowRuntime*, will be used later on to identify the workflow instance within the *Instance Migration Phase*.

```
[OperationContract]
void resumeWorkflow(String type, byte[] state, Guid instanceId);
```

The first parameter of `resumeWorkflow()` specifies the type of the workflow that will be resumed. The current state of the suspended composite service instance, represented as an

```
1  SELECT
2    _event as msg,
3    _event.WFTimestamp as wftimestamp,
4    _event.CompositionId as id,
5    _event.WorkflowState as state
6  FROM WorkflowTrackingEvent _event
7  WHERE _event.WorkflowState = 'Created'
8    OR _event.WorkflowState = 'Completed'
```

Listing 1: EPL Filtering `WorkflowTrackingEvents`

array of bytes, is included in the second parameter. As the composite service will be recreated and resumed with its original ID, this ID has to be included within the third parameter.

5.2.2 Event Processing

The prototype uses Esper and VRESCo for the processing of events. It utilizes a `WorkflowTrackingService`, which is the default way of tracking workflows in WF. This concept was briefly presented on Section 4.1. An implementation of the `WorkflowTrackingService` has to be registered with the `WorkflowRuntime`. Developers have to attach a custom `WorkflowTrackingChannel` to the `WorkflowTrackingService` that implements ways of further processing the events. At the occurrence of an event, the tracking service puts the event on the tracking channel by issuing the `send()` operation of the individual `WorkflowTrackingChannel`.

The tracking channel used within the prototype converts the events, that were reported by WF, to the corresponding event types of VRESCo. Event types in VRESCo are derived from the base event type `VRESCoEvent`. These events are then published and persisted by VRESCo. Each event is provided with a timestamp of millisecond accuracy which is used to calculate durations and SLA conformance. The tracking profile used in the prototype specifies the following events to be tracked by WF: *Created*, *Started*, *Unloaded*, *Loaded*, *Completed* and *Persisted*

SLA evaluation in the prototype is based on the execution duration of composite service instances and takes place within an event listener that is registered at Esper together with an EPL query. Esper notifies the listener by calling the `Update()` operation, which has to be implemented by the listener.

The queries used in the prototype are shown below:

The query shown in Listing 1 filters for `WorkflowTrackingEvents` of the state *Created* or *Completed* and selects the timestamps and IDs of workflow instances. The duration of composite service instances is then calculated within the listener by subtracting the timestamp of the *Created* event from the timestamp of the *Completed* event.

As shown by the EPL query in Listing 2 that is used to calculate the execution duration of a Web service invocation, the duration can also be directly calculated by Esper. The query selects the duration of service invocations and the ID of the invoking composite service instance.

The prototype calculates the duration of composite service instances within the listener because the listener also displays the current state of the runtime at the terminal of the VM. Each

```
SELECT
  _before.WorkflowId as id,
  _before.ActivityName as name,
  (_after.WFTimestamp - _before.WFTimestamp) as duration
FROM
  BeforeWFInvokeEvent _before,
  AfterWFInvokeEvent _after
WHERE _before.WorkflowId = _after.WorkflowId
  AND _before.ActivityName = _after.ActivityName
  AND _after.BeforeReferer = _before.WFId
```

Listing 2: EPL Evaluating Web Service Invocations

```
// create the VRESCo client proxy
IVReSCOSubscriber subscriber = VReSCOClientFactory.CreateSubscriber(login);

// use EPL to subscribe for the RuntimeStarted event
string epl =
  'select * from CompositionEngineEvent where State = "RuntimeStarted"
    and EngineId = ' + newInstanceId;
subscriber.SubscribePerWS(eql, ... , notificationEndpoint, expirationDate);
```

Listing 3: Subscribing to `RuntimeStarted` Event Using the VRESCo Client Library

event and each change of a state triggers an update of the displayed information.

The SLA used within this prototype, represents an agreement on the duration of a composite service instance. It is specified as a parameter within a configuration file, that is loaded by the prototype during startup. The value for this parameter is specified as milliseconds. In addition to the SLA, a threshold is defined within the configuration file. It determines the amount of SLA violations tolerated by the runtime until the decision to provision resources will be made. The listener that was described above, tracks the number of SLA violations and triggers the provisioning of resources when the specified threshold is exceeded.

5.2.3 Runtime Life-Cycle Events and Notifications

A custom event type, `CompositionEngineEvent`, that extends the VRESCo event model, was created for the purpose of tracking the states from Table 5. `CompositionEngineEvent` is directly derived from `VRESCoEvent` and defines attributes that contain the IP address and the unique identifier of the originating VM instance.

Listing 3 shows the usage of the VRESCo client library with an EPL query to subscribe for the `RuntimeStarted` event. The endpoint of the *Notification Web Service*, which is included within the `noficationEndpoint` attribute, is registered at the VRESCo subscription service. `expirationDate` defines when the subscription will expire. The EPL query filters the `RuntimeStarted` events for the unique identifier of the VM instance that was launched.

The VRESCo eventing service was extended to support the reporting of this event. The usage of the extended VRESCo eventing service is shown in Listing 4.

```
1  // create proxy for communication with eventing service
2  ChannelFactory<IEventingService> eventingFactory =
3   new ChannelFactory<IEventingService>("BasicHttpBinding_EventingService");
4  IEventingService eSrv = eventingFactory.CreateChannel();
5  eSrv.WorkflowHostStarted(myId, myIp, "RuntimeStarted"); // publish the event
```

Listing 4: Publishing `RuntimeStarted` Event Using Modified VRESCo Eventing Service

```
1  class WorkflowEventListener: RKiss.WSEventing.IEventNotification
2  {
3   public void Notify(VRESCoEvent[] newEvents,
4                      VRESCoEvent[] oldEvents,
5                      string subscriptionId)
6   {
7    foreach (VRESCoEvent myEvent in newEvents) {
8     if (myEvent.Type == "CompositionEngineEvent") {
9      CompositionEngineEvent engineEvent = (CompositionEngineEvent)myEvent;
10     if (engineEvent.State == "started") {
11      IWorkflowManager wfMgr = WorkflowManagerFactory.getWorkflowManager();
12
13      // notify runtime manager about relocated autonomic runtime
14      wfMgr.MachineInstanceReady(engineEvent.HostIP, engineEvent.EngineId);
15
16    ...
```

Listing 5: Publishing `CompositionEngineEvent` Using VRESCo Client Library

Listing 5 shows how the *Notification Web Service* is implemented using an WS-Eventing library for .NET [102]. `IEventNotification`, which is a component of this library, contains an *Operation Contract* for `notify()` which is used by the prototype to expose the operation as a Web service with WCF.

5.2.4 Virtual Machine Launching

Interaction with the Cloud is accomplished by using the EC2 library that is part of the AWS SDK for .NET [9]. The operations `AssociateAddress()` and `RunInstances()` are used for launching VM instances and associating a public IP address. These operations encapsulate the SOAP communication with the Web service that is provided by the CLC of Eucalyptus.

Listing 6 shows how the autonomic runtime launches a VM instance using the AWS SDK. The EC2 library has to be configured with an access key and a secret key, used for authentication, and the Uniform Resource Locater (URL), identifying the CLC of Eucalyptus. The instance type, the amount of VM instances to be launched and the EMI, containing the *Autonomic Runtime Loader*, need to be specified withing the request to Eucalyptus.

The autonomic runtime completes its relocation by associating the public IP address as shown in Listing 7.

The autonomic runtime learns about its own attributes, such as its currently used instance

```
// create client proxy
ec2cfg = new AmazonEC2Config().WithServiceURL(url);
ec2 = AWSClientFactory.CreateAmazonEC2Client(key, secret, ec2cfg);

// initialize the request
RunInstancesRequest riRequest = new RunInstancesRequest();
riRequest.WithImageId(imageId); // EMI hosting the autonomic runtime
riRequest.WithInstanceType(instanceType); // destination instance type
riRequest.WithMaxCount(1); // defaults to 20 instances
riRequest.WithMinCount(1); // launch exactly one instance

// launch VM instance
RunInstancesResponse riResponse = ec2.RunInstances(riRequest);
```

Listing 6: Launching VM Instance with AWS SDK

```
// associate public IP address
AssociateAddressRequest r = new AssociateAddressRequest();
r.WithInstanceId(instanceId); // instance, IP address will be associated with
r.WithPublicIp(ip); // associating IP address
AssociateAddressResponse res = ec2.AssociateAddress(r);
```

Listing 7: Associate Public IP Address with VM Instance Using AWS SDK

```
System.net.WebClient client = new System.net.WebClient();
string url = "http://169.254.169.254/latest/meta-data/instance-type";
string myItem = client.DownloadString(url);
```

Listing 8: Querying Instance Metadata from Eucalyptus

type, the IP address and the unique identifier of its VM instance by querying its own metadata from the fixed IP address *169.254.169.254* as explained in [38]. This feature is available in Eucalyptus, as well as Amazon EC2. Listing 8 shows how the autonomic runtime uses the `WebClient` of .NET to accomplish metadata querying.

5.2.5 Migrating Composite Service Instances

The autonomic runtime subscribes to the *RuntimeStarted* event at the VRESCo runtime environment, when resource provisioning was triggered. During the startup of the relocating autonomic runtime, this event is published to VRESCo together with the private IP address of the newly created VM instance. VRESCo then notifies the original autonomic runtime via WS-Eventing about the occurrence of this event. The original autonomic runtime subsequently starts the migration of composite service instances to the relocated autonomic runtime by connecting to the *Runtime Web Service* at the IP address that was reported within the *RuntimeStarted* event.

The `WorkflowPersistenceService` of WF is used to migrate the state of composite service instances. It provides the base for saving and loading `WorkflowInstances`.

```
1  public class WorkflowMigrationService: WorkflowPersistenceService
2  {
3   protected override void SaveWorkflowInstanceState(Activity rootActivity,bool
        unlock)
4   {
5    Guid instanceId = WorkflowEnvironment.WorkflowInstanceId;
6    MemoryStream ms = new MemoryStream();
7
8    // serialize state of composite service instance
9    rootActivity.Save(ms);
10
11   // store state within WorkflowMigrationStore
12   WorkflowMigrationStore.unloadInstanceState(instanceId, ms);
13  }
14
15  protected override Activity LoadWorkflowInstanceState(Guid instanceId)
16  {
17   // load state from WorkflowMigrationStore
18   MemoryStream ms = WorkflowMigrationStore.loadInstanceState(instanceId);
19
20   // deserialize state of composite service instance
21   Activity rootActivity = Activity.Load(ms, null);
22
23   // delete state from WorkflowMigrationStore
24   WorkflowMigrationStore.deleteInstanceState(instanceId);
25
26   // return state to WorkflowRuntime
27   return rootActivity;
28  }
29 }
```

Listing 9: Serializing and Deserializing Composite Services

The `Unload()` and `Load()` operations of `WorkflowInstance` trigger the `WorkflowRuntime` to call the `SaveWorkflowInstanceState()` and `LoadWorkflowInstanceState()` operations of a registered `WorkflowPersistenceService`. To use this mechanism the `SaveWorkflowInstanceState()` and `LoadWorkflowInstanceState()` operations were implemented within the `WorkflowMigrationService` which is directly derived from the abstract `WorkflowPersistenceService`. Listing 9 shows the implementation of these operations.

The state of suspended composite service instances is stored in the in-memory residing `WorkflowMigrationStore`. Access to the `WorkflowMigrationStore` was implemented based on a thread-safe version of the *Singleton* [46] design pattern. The `WorkflowMigrationService` uses the `WorkflowMigrationStore` to either save the state of an `WorkflowInstance` when the `Unload()` operation is called, or retrieve the state from the `WorkflowMigrationStore` when the instance is loaded with the `Load()` operation. The `SaveWorkflowInstanceState()` operation serializes the workflow instance using the `Activity.Save()` operation and stores the state together with the ID of the instance to

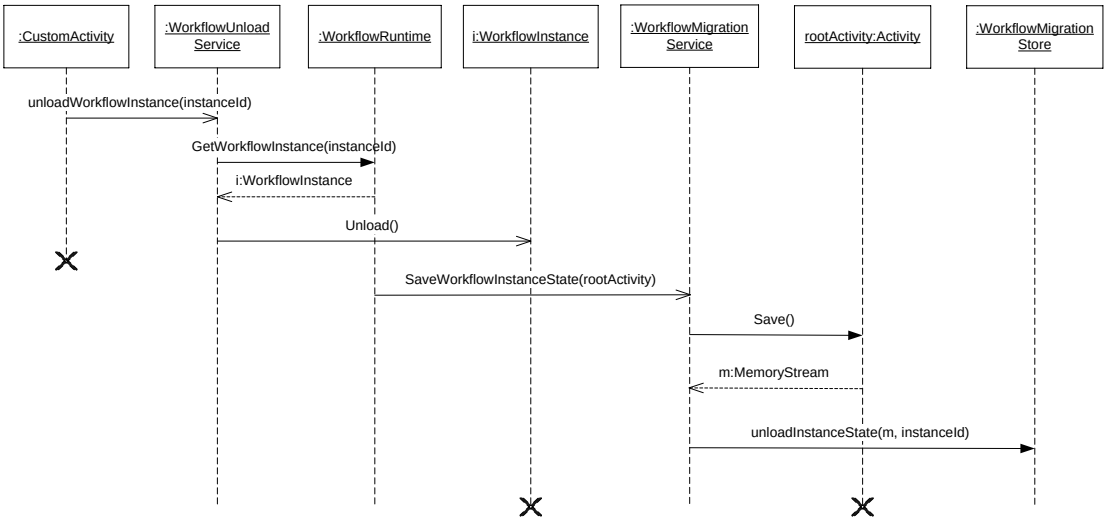


Figure 13: Saving a Comoposite Service Instance to `WorkflowMigrationStore`

the `WorkflowMigrationStore`. The `LoadWorkflowInstanceState()` method retrieves the state by the instance ID and deserializes it with the `Activity.Load()` operation.

Saving a composite service to the `WorkflowMigrationStore` is shown in Figure 13. To be able to serialize the state of a composite service instance, it has to be suspended at a point in time where its state is consistent and readable by the execution engine. In WF, the suspension of a workflow is only possible when the workflow instance is not currently executing an activity. To suspend and serialize all composite service instances at the earliest possible moment, a system-wide flag controls the suspension. This flag is checked by a workflow instance at the end of each activity. If this flag states an ongoing *Instance Migration Phase*, all workflow instances suspend themselves and trigger the serialization of their state. Because it is not possible to access a corresponding `WorkflowInstance` from within an activity of the workflow, a custom *Runtime Service* has been implemented. This custom service, the `WorkflowUnloadService`, has been attached to the `WorkflowRuntime` and is accessible from within an instance of the workflow. If the system-wide flag is active, the workflow instance will invoke the corresponding operation of the `WorkflowUnloadService` to suspend and serialize itself. The invocation of the `Unload()` operation within the custom runtime service has to take place asynchronously, as it would otherwise block forever. `Unload()` does not return until serialization has finished which in turn requires the workflow instance to be ready for suspension and the currently executing activity to be finished. Asynchronous invocation is accomplished by the help of a *delegate*. The implementation of `WorkflowUnloadService` and the use of a *delegate* are shown in Listing 10. Listing 11 shows how the runtime service is invoked from within a WF *Custom Activity*.

Transmission of composite service states is accomplished by using the *Runtime Web Service* of the destination runtime and invoking its `resumeWorkflow()` operation. The source runtime invokes this operation for each composite service instance that has to be transferred one by one. The destination runtime extracts the state and ID from the parameters of `resume-`

```
public class WorkflowUnloadService : WorkflowRuntimeService
{
 public void suspendWorkflowInstance(Guid instanceId)
 {

  // fetch corresponding WorkflowInstance
  WorkflowInstance i = base.Runtime.GetWorkflow(instanceId);

  // create delegate for Unload() operation
  MethodInvoker invoker = new MethodInvoker(i.Unload);

  // tell delegate invoke operation
  invoker.BeginInvoke(null, null);
 }
}
```

Listing 10: Asynchronously Invoking `Unload()` with a *Delegate*

```
class CustomActivity : System.Workflow.ComponentModel.Activity {
 override ActivityExecutionStatus Execute(ActivityExecutionContext ctx)
 {

  ... // dynamic invocation of a Web service

  if (InMigration.inMigration)
  {
   // fetch WorkflowUnloadService and trigger suspension
   WorkflowUnloadService unloadService =
    executionContext.GetService(typeof(WorkflowUnloadService));
   unloadService.suspendWorkflowInstance(this.WorkflowInstanceId);
  }
 }
}
```

Listing 11: Invoking Runtime Service from a Custom Activity

`Workflow()`. The state is stored in the `WorkflowMigrationStore` and a new workflow instance is created with the `WorkflowRuntime` using the original ID of the workflow.

Before the `Load()` operation can be invoked on the newly created instance, the `WorkflowInstance` has to be put into an *unloaded* state. This is accomplished by invoking the `Unload()` operation right before the call to `Load()`. The procedure that is used to resume a `WorkflowInstance` is shown in Listing 12.

5.3 Deployment in the Cloud

This section deals with the deployment of the autonomic composite service runtime to a private installation of the open source Cloud platform Eucalyptus. Eucalyptus was introduced in Section 2.2.4. To begin with, the way of how the Cloud was accessed and used as part of this

```
1  WorkflowRuntime wfRuntime = new WorkflowRuntime(); // create WorkflowRuntime
2
3  // attach runtime services
4  wfRuntime.AddService(new VRESCoTrackingService()); // tracking service
5  wfRuntime.AddService(new WorkflowMigrationService()); // migration service
6  wfRuntime.AddService(new WorkflowUnloadService()); // unload service
7
8  // create workflow using original instance ID
9  WorkflowInstance wfInstance =
10  wfRuntime.CreateWorkflow(wfType, null, instanceId);
11
12 wfRuntime.StartRuntime(); // start runtime services
13 wfInstance.Unload(); // puts workflow into unloaded state
14 wfInstance.Load(); // loads workflow state from migration store
```

Listing 12: Resuming a Composite Service

System	Dell PE M610 Blade Servers
Operating System	Debian 6.0 Squeeze, Linux 2.6.32
Memory	16 GB DDR3-1333
CPU	2 x Xeon E5620 Quad Core @ 2.4 GHz, 12M Cache
Disk	2 x 146 GB SAS HDDs (Hardware Raid-1)

Table 6: Hardware Specification

thesis will be shown. Next, the deployment of the various components of the autonomic runtime and how they communicate with each other will be presented. Then, the initialization of the autonomic runtime in the Cloud will be illustrated. Finally, the creation of EMIs and the various issues that came up in doing so will be described.

5.3.1 Eucalyptus Setup

The installation of Eucalyptus, available for this thesis, was provided by the Institute of Information Systems at the Vienna University of Technology. It consists of five Dell PE M610 Blade Servers. Their specifications are shown in Table 6. One of them is hosting the CLC, CC and Walrus services, while four servers host the NCs and form one Eucalyptus cluster, managed by the CC. Additionally, Dynamic Host Configuration Protocol (DHCP) and DNS services are hosted together with the CLC to dynamically assign IP addresses to VMs and to associate them with domain names. The NCs and the CLC are located in the physcical Local Area Network (LAN) of the institute and additionally connected through a virtual LAN, dedicated to Eucalyptus. This setup is illustrated in Figure 14.

Eucalyptus 2.0 is used within this setup. It was configured to use the *greedy* policy for placing VMs among nodes. This policy places VMs on the first node which is found to provide the necessary resources. Eucalyptus also supports *round robin* placement of VMs, where VMs are

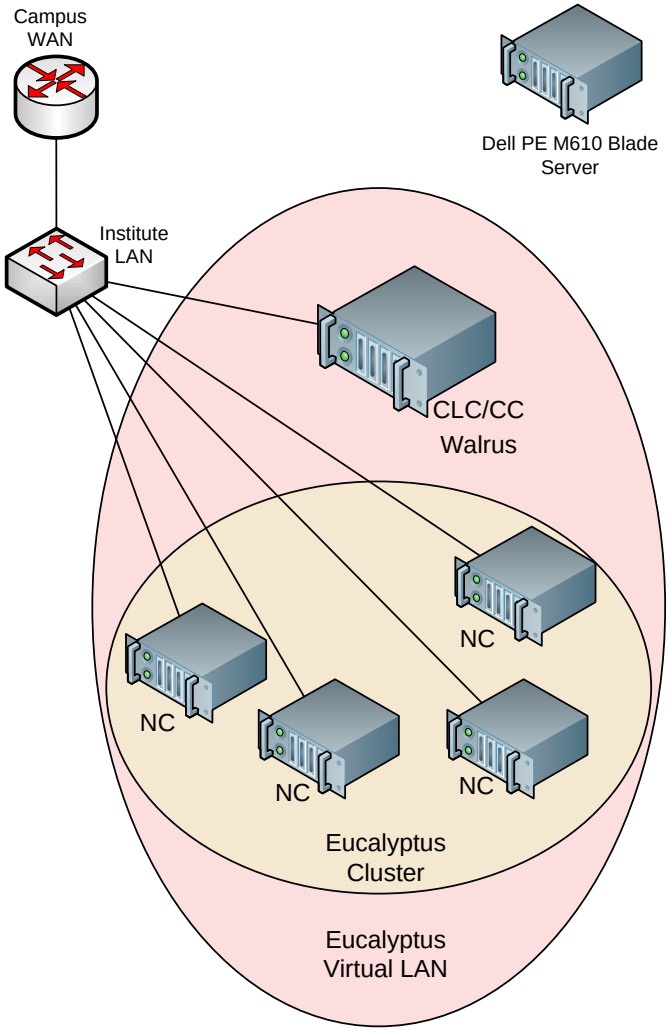


Figure 14: Eucalyptus Setup

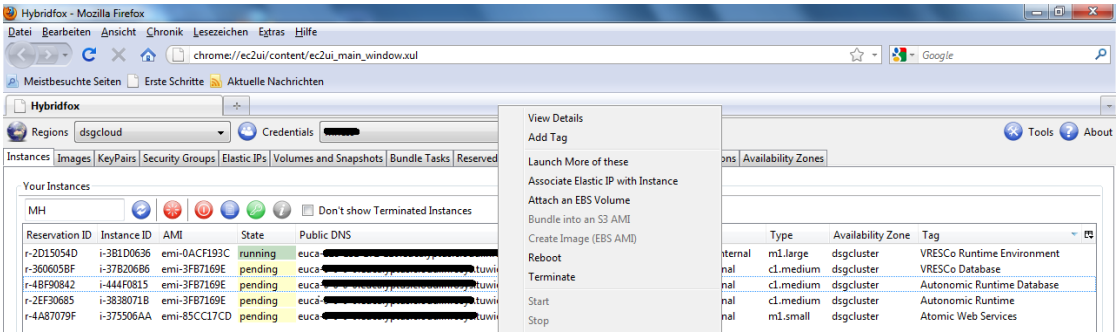


Figure 15: Graphical User Interface of Hybridfox

distributed evenly among the nodes. As an inefficient use of resources was experienced when using *round robin* policy, this setup is using *greedy* policy.

5.3.2 Interacting with the Cloud

The Web service of the CLC is the single communication endpoint for interactions with the Cloud platform. Both the Eucalyptus command line tools (*euca2ools*) and the Amazon AWS SDK can be used for communication with the CLC. While Amazon EC2 also provides a rich web-based user interface that enables users to graphically manage their VMs, the web-based user interface of Eucalyptus is limited to basic administration tasks. It provides user access management and the configuration of a few Cloud settings, for example, customizing instance types and the network settings of the CLC and *Walrus*. Cloud users are able to change their account information and their password, as well as download their credentials for the command line tools and the Eucalyptus Web service. To still enjoy the benefits of a graphical user interface that gives a clear view on VM instances and their states, Hybridfox[1], which is as browser plugin for Firefox[2] and a fork from Elasticfox[3], was used for this thesis. Hybridfox tries to support all features of Eucalyptus without breaking the EC2 functionality. Figure 15 shows a screenshot of Hybridfox displaying information about VM instances, for example, the state of each instance, their addresses and instance types. Table 7 describes the basic commands of *euca2ools*.

The following output from the command `euca-describe-availability-zones` shows the resources that are associated with the various instance types and their availability. At the time of this output, enough resources for the creation of eight *c1.xlarge* instances (from a maximum of twelve instances) were available.

```
$ euca-describe-availability-zones verbose
AVAILABILITYZONE        dsgcluster      10.11.11.1
AVAILABILITYZONE        |- vm types     free / max   cpu    ram  disk
AVAILABILITYZONE        |- m1.small     0045 / 0064   1    512    20
AVAILABILITYZONE        |- c1.medium    0042 / 0060   1   1024    20
AVAILABILITYZONE        |- m1.large     0019 / 0028   2   2048    20
AVAILABILITYZONE        |- m1.xlarge    0013 / 0020   2   3072    20
AVAILABILITYZONE        |- c1.xlarge    0008 / 0012   4   4096    20
```

[1] http://code.google.com/p/hybridfox/ Visited: 2011-10-10
[2] http://www.firefox.com/ Visited: 2011-10-10
[3] http://sourceforge.net/projects/elasticfox/ Visited: 2011-10-10

Command	Description
`euca-describe-instances`	Displays information about VM instances (e.g., their state, instance type, owner, etc.)
`euca-run-instances`	Creates one or more VM instances of a specified instance type from an EMI
`euca-terminate-instance`	Terminates a VM instance
`euca-allocate-address`	Allocates a public IP address that can be associated with a VM instance; the IP addresses is reserved by the allocating user until it is manually released
`euca-associate-address`	Associated a public IP address with a VM instance
`euca-authorize`	Configures security groups to allow certain networking traffic
`euca-create-volume`	Creates a dynamic block volume that can be attached to a running VM instance
`euca-describe-availability-zones`	Display information about an availability zone, for example, available and maximum resources

Table 7: Basic Commands of Eucalyptus *euca2ools*

The following execution of the `euca-run-instances` command creates an instance of the autonomic composite service runtime host by using an instance type of *c1.medium*, EMI *emi-3FB7169E*, kernel *eki-30CD0D30* and ramdisk *eri-8F570F48*.

```
$ euca-run-instances emi-3FB7169E --kernel eki-30CD0D30 --ramdisk eri-8F570F48 -t c1.medium
RESERVATION  r-360605BF   mhess  mhess-default
INSTANCE     i-37B206B6   emi-3FB7169E  euca-0-0-0-0.tuwien.ac.at euca-0-0-0-0.eucalyptus.internal
pending 2 011-09-20T09:24:43.96Z  eki-30CD0D30  eri-8F570F48
```

5.3.3 Deployment of the Runtime

All communication that occurs between the autonomic runtime and external systems is based on the SOAP protocol and therefore uses IP networking. External systems, the autonomic runtime is interacting with, are the *VRESCo Runtime Environment*, the *Web Services* that compose the composite service, the CLC of Eucalyptus and the clients invoking the composite service. It does not matter to the autonomic runtime whether the VRESCo runtime environment and the Web services are hosted in the same or a different Cloud infrastructure or not in a Cloud at all, as long as they are accessible by the autonomic runtime over the network. This thesis considers both of them being hosted in the same installation of Eucalyptus. This has the advantage of optimal network latency and bandwidth, because all systems are located in the same LAN. The deployment, considered by the thesis at hand, is as follows. Figure 16 illustrates this deployment.

- The *Autonomic Composite Service Runtime* is hosted on a single VM instance at a time. This instance may be replaced by the resource provisioning procedure. The autonomic runtime is self-hosting without the use of a dedicated web server.

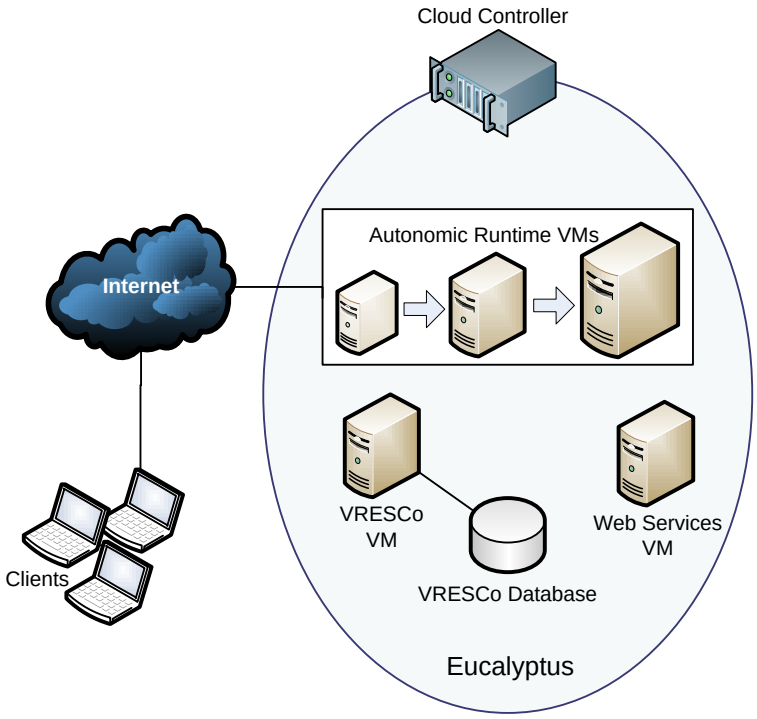


Figure 16: Autonomic Runtime Deployment

- The *VRESCo Runtime Environment* is hosted on a dedicated VM instance together with a web server that exposes the services of VRESCo to any clients.
- The *VRESCo Database*, which is needed by VRESCo, is hosted on the same VM instance as VRESCo itself.
- The *Web Services* invoked by the composite service, are hosted together on a single VM instance as they do not implement any resource expensive application logic. They need to be published to the VRESCo service registry to be available for being queried by the autonomic runtime at the time of their invocation.
- The CLC is a part of the Cloud infrastructure and therefore available to the autonomic runtime over the LAN.
- *Clients* may communicate with the autonomic runtime and invoke the composite service over the Internet or from any other location where the autonomic runtime is accessible from. They may as well be located in the same Cloud infrastructure as the autonomic runtime.

The autonomic runtime, VRESCo and the Web services communicate by using private IP addresses of the Eucalyptus LAN. Communication is therefore not effected by a changing public

IP address of the autonomic runtime during the phase of resource provisioning. Private IP addresses are likely to change after the termination or restart of a VM instance and are unable to be reserved in contrast to public IP addresses. Therefore, the autonomic runtime was designed in such a way that it does not need to know the endpoints of the Web services and VRESCo *a priori*. While the endpoints of the Web services are queried from the VRESCo registry and invoked dynamically, VRESCo is the only component that has to be located at initialization time. This is accomplished by providing the autonomic runtime with a static endpoint that is very unlikely to change, e.g. the CLC, and that can be queried for the exact location of VRESCo. The following sections present in detail how this was accomplished with the prototype.

5.3.4 Infrastructure Initialization

The initial startup of the autonomic runtime can take place with either one of the browser plugins or the *euca2ools*. Before startup a public IP address has to be allocated using the command `euca-allocate-address`. This address will be used by the autonomic runtime and associated with VM instances on demand. A VM instance has to be created with the corresponding EMI, the correct kernel and ramdisk images and an initial instance type using `euca-run-instances`. As soon as the VM instance has booted and the autonomic runtime is online, clients can be served with the composite service.

The security group settings for the VM hosting the autonomic runtime, needs to be configured to enable communication with components outside of the Cloud. This is done by using `euca-authorize`. The exact configuration depends on the concrete deployment of the system (e.g., whether VRESCo is within the same Cloud as the autonomic runtime). Above all, clients, expected to be outside of the Cloud, need to be able to send requests to the autonomic runtime.

5.3.5 Loading the Runtime

The startup of the autonomic runtime is performed by a dedicated component, the *Autonomic Runtime Loader*. It is included by the EMI that serves the autonomic runtime and starts automatically at the time its VM instance has booted. This was achieved by registering the autonomic runtime loader as a *Windows Service*.

The complexity of the autonomic runtime loader is reduced to a minimum, so that changes are very unlikely to be necessary and to prevent a redeployment of this component. Its usage was motivated by two issues that came up during the development of the autonomic runtime prototype. On the one hand, how the dynamic nature of the private IP addresses used for communication within the Cloud could be bypassed. On the other hand, how changes to the code of the autonomic runtime or its parameters could be applied without needing to recreate and upload the EMI.

To solve these issues, the autonomic runtime loader performs within two phases. In the first phase, the autonomic runtime loader locates the address of the VRESCo runtime environment. Eucalyptus does not support tagging of instances, therefore the VM instance that hosts VRESCo is identified by the name of its EMI. The name of the EMI that includes the VRESCo runtime environment, is known to the autonomic runtime loader. The loader queries the CLC for all

instances that were created from this EMI and that are located in the same availability zone as the loader. If the CLC returns more than one instance, the first one will be used. The private IP addresses of this instance is then extracted and from now on used for the communication with the VRESCo runtime environment.

The second phase downloads the compressed code and parameters of the autonomic runtime from the VRESCo host, decompresses them and starts the autonomic runtime. To centrally deactivate the automatic startup of the autonomic runtime for testing or debugging purposes, the loader reads a boolean attribute from the web server, also hosting the VRESCo runtime environment, that controls whether the runtime will be started.

5.3.6 Machine Images

To host the autonomic composite service runtime in Eucalyptus, an EMI was created, that includes the required software stack to load and execute the runtime. As this thesis considers the VRESCo runtime environment and the Web services to be hosted in Eucalyptus together with the autonomic runtime, EMIs were also created for VRESCo and the hosting of the Web services.

The creation of the EMIs was accomplished using KVM [14]. The steps that were taken to prepare a single image for Eucalyptus are summarized in the following:

1. A disk image of type *RAW* was created with the command `kvm-image` of KVM
2. The operating system was installed onto this image using KVM
3. The drivers for the network and disk interfaces were installed to this system
4. The individual software stack was installed to the system and the system configuration was customized
5. The image was bundled with the `euca-bundle-image` command of *euca2ools*
6. The image was uploaded to Walrus with `euca-upload-image` and registered with `euca-register`

Virtual disks of VM instances in Eucalyptus are ephemeral. They lose any changes that were made to them during the execution of the instance. Therefore, all EMIs, that were created as part of this thesis, were prepared in such a way to require only minimal or no changes to running VM instances.

The following EMIs were created and equipped with the required software stack to run the corresponding components of the autonomic runtime and its environment. Tables 8, 9, 10 show the individual configurations that were made during the preparation of the EMIs (i.e., before they were bundled and registered with Eucalyptus).

- The EMI shown in Table 8 is equipped with the required software to load and initialize the autonomic composite service runtime. The *Autonomic Runtime Loader* was converted into a Windows Service that starts automatically at Windows startup. No manual changes need to be made after a VM instance was created from this EMI.

Name	workflow-win2k3x64.raw
Operating System	Windows Server 2003 R2 64 Bit [80]
Additional Software	.NET 3.5, AWS SDK for .NET
Deployed Software Component	Autonomic Runtime Loader
Windows Firewall	8081 (Autonomic Runtime Web Service), 8082 (Autonomic Notification Web Service), permit autonomic runtime application to listen on these TCP ports
Size	Image 1.6 GB, Disk 4 GB

Table 8: Autonomic Composite Service Runtime EMI

Name	services-winxp32sp3.raw
Operating System	Windows XP SP3 32 Bit [84]
Additional Software	.NET 3.5
Deployed Software Component	Web Services Binaries
Windows Firewall	60000 - 60010 (Web Services), permit Web service application to listen on these TCP ports
Size	Image 1.1 GB, Disk 4.5 GB

Table 9: Web Services EMI

- The EMI shown in Table 9 hosts the Web services. If DNS is not used to access the Web services, the private IP address of the hosting VM instance has to be set in the configuration file of the Web services and registered at the VRESCo registry.
- The EMI shown in Table 10 hosts the VRESCo runtime environment and the VRESCo database. Therefore the VRESCo database was created and prepared as part of the EMI creation. That way VRESCo is ready to be used, as soon as a VM image is booted from this EMI. During EMI preparation, Microsoft IIS7 [83] was also configured to host two web folders, one that serves requests for the VRESCo runtime environment and one that hosts the deployment file of the autonomic composite service runtime which is downloaded by the *Autonomic Runtime Loader*. As VRESCo is accessed using HTTP over Secure Socket Layer (SSL), a x509 certificate was generated and deployed to IIS7.

5.3.7 Issues with Windows-based EMIs

By the time this thesis was written, Eucalyptus was available in version 2.0. This version did not officially support the usage of VM instances which run Windows operating systems. The

Name	vrescohost-win2k3x64r2.raw
Operating System	Windows Server 2003 R2 64 Bit
Additional Software	.NET 3.5, Microsoft IIS7, MySQL 5 (as part of XAMPP 1.7.4 [10])
Deployed Software Component	VRESCo Runtime Environment, VRESCo Database
Windows Firewall	80 (HTTP), 443 (HTTPS), 3306 (MySQL)
Size	Image 1.7 GB, Disk 10 GB

Table 10: VRESCo Runtime Environment EMI

roadmap for Eucalyptus scheduled the support for hosting Windows to be implemented in version 3.0 [39]. Creating Windows based EMIs and using them to run Windows in Eucalyptus was therefore afflicted with various issues. These issues and the online publications that helped to solve them are illustrated in the following.

Partitioning RAW Images The image type *RAW* had to be used because a bug in the used version of Eucalyptus prevented the usage of the more efficient *qcow2* images with Windows based EMIs. The images had to include two distinct partitions, so that Eucalyptus won't handle them in the same way as Linux images.

Kernel and Initrd Images To boot the Windows-based EMIs, customized images for kernel and an initial ramdisk (*initrd*)[4] had to be used as shown in [2]. The kernel image was created from the *SYSLINUX* bootloader[5] and the *initrd* was created from a conventional Windows boot floppy [76].

ATA and SCSI The default interface that Eucalyptus uses to connect virtual disks to VMs is *Small Computer System Interface (SCSI)*[6]. While guidelines, as for instance [2] and [60], that document the procedure of creating a Windows based EMI, describe the installation of SCSI drivers into the EMI, using SCSI was not able with the Eucalyptus installation used in this thesis. Therefore, Eucalyptus was configured to use *ATA*[7] instead of SCSI to connect virtual disks.

Networking Drivers Network driver for e1000 [53] network interfaces as described in [50] had to be installed, as the virtual network interfaces that are used by Eucalyptus are of this type. Without these drivers networking was not possible with the custom EMIs.

[4] http://en.wikipedia.org/wiki/Initrd Visited: 2011-10-02
[5] http://www.kernel.org/pub/linux/utils/boot/syslinux/ Visited: 2011-10-02
[6] http://en.wikipedia.org/wiki/SCSI Visited: 2011-10-02
[7] http://en.wikipedia.org/wiki/Parallel_ATA Visited: 2011-10-02

Debug EMIs with VNC To debug the EMIs, *VNC* [101] was used as described in [40]. If an instance was not able to boot or no network connections to the VM were possible a VNC client was used to connect to the terminal of the VM instance to see the output.

CPU and Memory Support of Windows As the various versions of Windows support different amounts of CPUs and memory, care has to be taken on the choice of which Window version is used with an instance type. The CPU and memory support of the various Windows versions is shown in [79].

Remote Management of Virtual Machines The Remote Desktop Service of Windows [77] was enabled on all EMIs to provide remote administration of the VMs. To permit traffic on this port, the Windows firewall, as well as the Eucalyptus security groups had to be configured to permit incoming traffic on Transmission Control Protocol (TCP) port 3389.

Windows Firewall and Automatic Windows Updates For security reasons, the Windows firewall was active on the EMIs, although it complicates debugging them. It was configured at preparation time to permit exactly the traffic needed by the systems to work. Automatic Windows updating was deactivated to prevent the update service from influencing the autonomic runtime. Besides, installed updates would get lost after termination because the virtual disks are ephemeral.

TCP Adjustments The Windows registry was modified to enable Windows to use more TCP ports and to reuse them faster, as the database connections which were made for persisting the high amount of events required that. This was done as recommended by [78]. The attributes `TcpTimedWaitDelay` [82] and `MaxUserPort` [81] were changed to 30 and 65534 respectively.

UTC Hardware Clock Support The Windows EMIs had to be configured to support a Coordinated Universal Time (UTC) hardware clock, because Eucalyptus runs VM instance with a UTC enabled hardware clock. Therefore the Windows Registry had to be adjusted to support UTC as shown in [43].

CHAPTER 6

Evaluation

This chapter shows the results of an extensive evaluation of the presented prototype. The goal is to prove the usefulness of the prototype and to discuss limitations of the autonomic runtime by answering the following questions:

1. Will the overall amount of SLA violating composite service instances be decreased by provisioning resources to the composite service runtime?
2. Can the violation of the SLA by a composite service instance be prevented by migrating this instance and in what circumstances is this possible?
3. What is the cost for provisioning resources and does the benefit outweigh the cost?

First of all, the setup of the evaluation and the approach are presented. Then the impact of resources on the composite service runtime is explained. Last but not least the three questions previously raised are answered by taking a closer look at the capabilities of the autonomic runtime prototype.

6.1 Evaluation Setup and Approach

The Eucalyptus instance types were customized to ensure the types used to host the autonomic runtime differ in their amounts of CPUs and memory. These choices had to be aligned with the physical resources available from the underlying hardware. The instance types and their resources are shown in Table 11.

The evaluation setup was based on the deployment described in Section 5.3. Each participating component, except for the composite service consumers, was hosted in the same installation of the Eucalyptus Cloud. The presented EMIs were used to launch VM instances, hosting these software components. The setup for the evaluation is shown in Figure 17.

The following components participate in the evaluation setup:

	Cores	Memory	Storage	Platform
	#	MB	GB	bit
m1.small	1	512	20	64
c1.medium	1	1024	20	64
m1.large	2	2048	20	64
m1.xlarge	2	3072	30	64
c1.xlarge	4	4096	30	64

Table 11: Customized Resource Capacity of Eucalyptus Instance Types

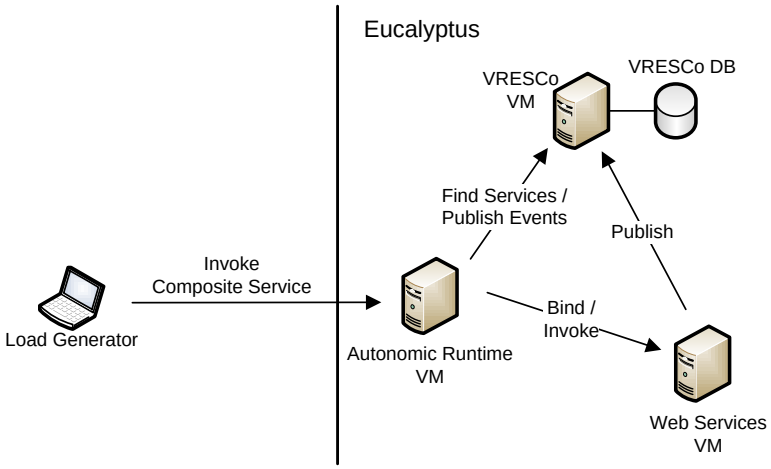


Figure 17: Evaluation Setup

- The *Autonomic Composite Service Runtime* was hosted on single VM instances of types *c1.medium*, *m1.large* and *c1.xlarge*.
- The *VRESCo Runtime Environment* and the *VRESCo Database* were hosted together on a single VM instance of type *m1.large*. VRESCo stores the endpoints of the Web services and provides them to the autonomic runtime to invoke the Web services dynamically.
- The *Web Services* were hosted on an instance of type *m1.small* together. These Web services do not implement any application logic but provide the interfaces to be invokable by the composite service. Their responses are randomly delayed for up to 50 milliseconds, to simulate implemented application logic. Web services have to be published to the VRESCo service registry.

The load was generated by a specifically built load generator. It invokes the composite service and thereby triggers the creation of composite service instances in the autonomic runtime. It supports invoking the composite service at two alternating rates. One rate continuously invokes the composite service at a constant frequency. The other rate sequentially invokes the composite

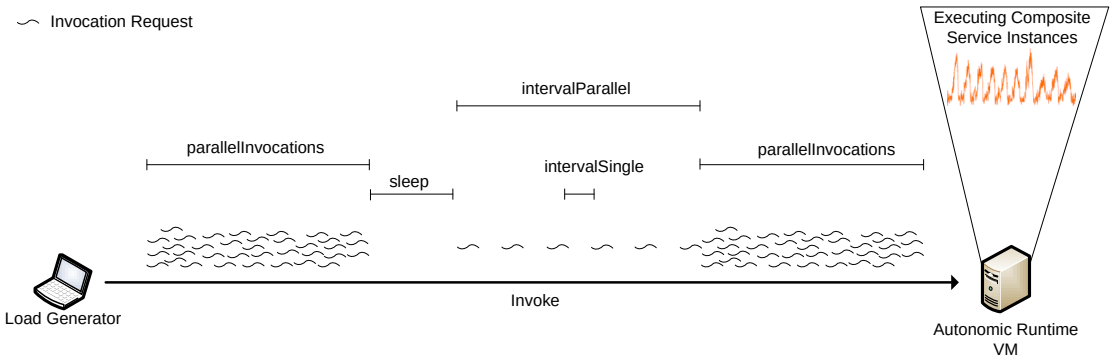


Figure 18: Load Generator Requests

Parameter	Description	Low Rate	High Rate
`intervalSingle` (ms)	Frequency of single invocations	1,000	900
`intervalParallel`	Amount of single invocations between parallel invocations	30	30
`parallelInvocations`	Number of parallel invocations	80	90
`sleep` (ms)	Pause after parallel invocations	10,000	10,000

Table 12: Load Generator Parameters

service as fast as possible. Two different rates are needed to create a certain amount of concurrently executing instances of the composite service. On the one hand, the resources have to be utilized extensively to force SLA violations. To evaluate migration, multiple composite service instances need to be available at any point in time. On the other hand, the runtime must not get overloaded by handling too many invocations, resulting in Denial of Service (DoS). Figure 18 illustrates invocation of the composite service by the load generator.

Table 12 describes the parameters of the load generator. Two distinct parameterizations were used throughout this evaluation to generate the load. A lower frequency of composite service invocations was used when provisioning resources to a *c1.medium* instance, a higher frequency when provisioning resources to a *m1.large* instance.

Evaluations were made by retrospectively observing the behavior and actions of the autonomic runtime. This was possible due to persisted events from composite services and the runtime, as described in Section 5.2. Therefore it was possible to query specific events and their time of occurrence from the VRESCo database. Two examples of evaluation queries are shown in the appendix. The SQL query in Listing 13 calculates the runtime (in milliseconds) of migrated composite service instances. Listing 14 shows an SQL query calculating the duration (in

Paramter	Value	Description
Public IP	E.g. 192.168.2.23	Defines the public IP address of the autonomic runtime
Migration	yes/no	Defines whether instances will be migrated
Startup	yes/no	Defines whether the autonomic runtime is loaded automatically after VM startup
Termination	yes/no	Defines whether the autonomic runtime terminates the source VM after resource provisioning
SLA Value	E.g. 35 ms	Defines the SLA for executing a composite service instance
SLA Violation Threshold	E.g. 200	Defines when resource provisioning will be triggered by the autonomic runtime

Table 13: Autonomic Runtime Parameters

minutes) of resource provisioning.

Each experiment involved preparing the environment of the autonomic runtime and individual parametrization of the load generator and the autonomic runtime. Table 13 shows the parameters of the autonomic runtime. Figure 19 illustrates the overall procedure of how experiments were performed.

6.2 Composite Service Performance in the Cloud

It is important to understand how the runtime performance of a composite service depends on the amount of provided resources. This will help to extensively evaluate the solution provided in this thesis and will allow to evaluate usefulness of the resource provisioning concept presented in Chapter 5. The metric chosen to measure the performance of the runtime is represented by the execution duration of composite service instances. In other words, it is represented by the time passing between the creation of a composite service instance and its completion.

To learn how provided resources impact the execution duration, the autonomic runtime was hosted on the three previously mentioned instance types independently from each other. On each instance type, the runtime served a certain amount of composite service invocations and thereby executed a corresponding amount of composite service instances in parallel. Autonomic resource provisioning was deactivated for this evaluation. The composite service was invoked repeatedly at the fastest possible rate. I.e., invoking the service without any delays between invocations. The amount of parallel executing composite service instances thereby depends on the speed the runtime is working off the instances, and how fast the invocation requests could be

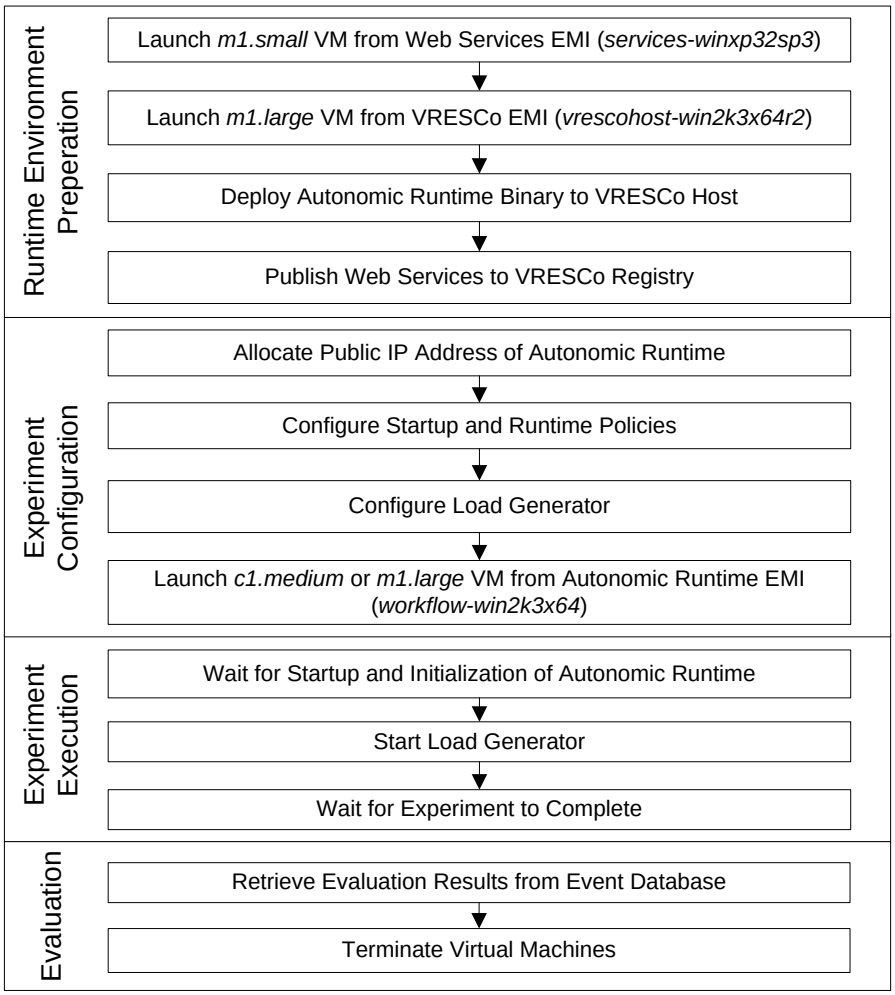


Figure 19: Experimentation Procedure

handled.

Figure 20 shows the results of this evaluation. As can be seen, when hosting the runtime on *c1.medium*, the durations increase much faster than on *m1.large* and *c1.xlarge*. Starting from 400 executing composite service instances, the duration on *c1.medium* breaks out, while *m1.large* and *c1.xlarge* can still handle the load. Because of these excessive durations and the limited amount of memory, no more than 600 instances were considered on *c1.medium*.

The more composite service instances are executed simultaneously, the more their durations vary. This is due to the longer lasting duration of composite services, started at the beginning of each experiment. During their execution, the runtime still needs to handle the ongoing invocations. Composite service instances started towards the end are affected to a much smaller extent.

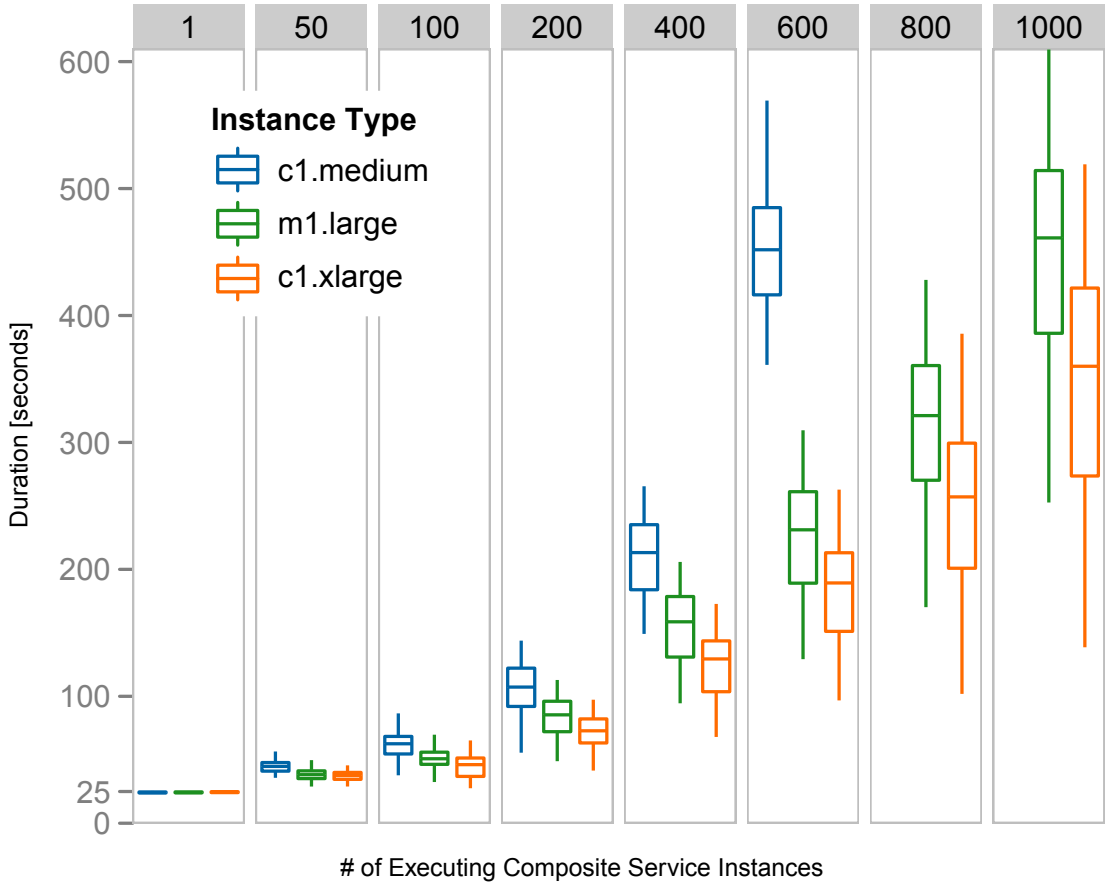


Figure 20: Execution Durations of a Composite Service with Increasing Concurrency

Figure 21 illustrates the increasing averages of the results discussed. A certain limit on each instance type can be identified, from where the averages grow almost exponentially. *c1.medium* reaches this limit when executing 200 parallel instances. While the proof for an exponential growth on *m1.large* and *c1.xlarge* is not apparent from Figure 21, exponential growth is assumed to occur from 1000 parallel instances upwards. On the other hand, it is safe to assume that the more CPUs a VM instance has, the flatter the slope of the curve will be. With only one CPU, as on *c1.medium*, just one composite service instance can be served at the time, while the other executing composite service instances have to wait for a CPU cycle. With an increasing amount of CPUs, more composite service instances can be handled simultaneously.

A composite service still depends on its called Web services and their performance characteristics. Assuming these Web services perform optimally, i.e., each requested Web service responds within the minimum possible time, a minimum execution duration for the composite service exists. Figures 20 and 21 show the minimum of the composite service, used throughout this evaluation, to be at about 25 seconds.

The results presented in this section show the impact of resources on the composite service

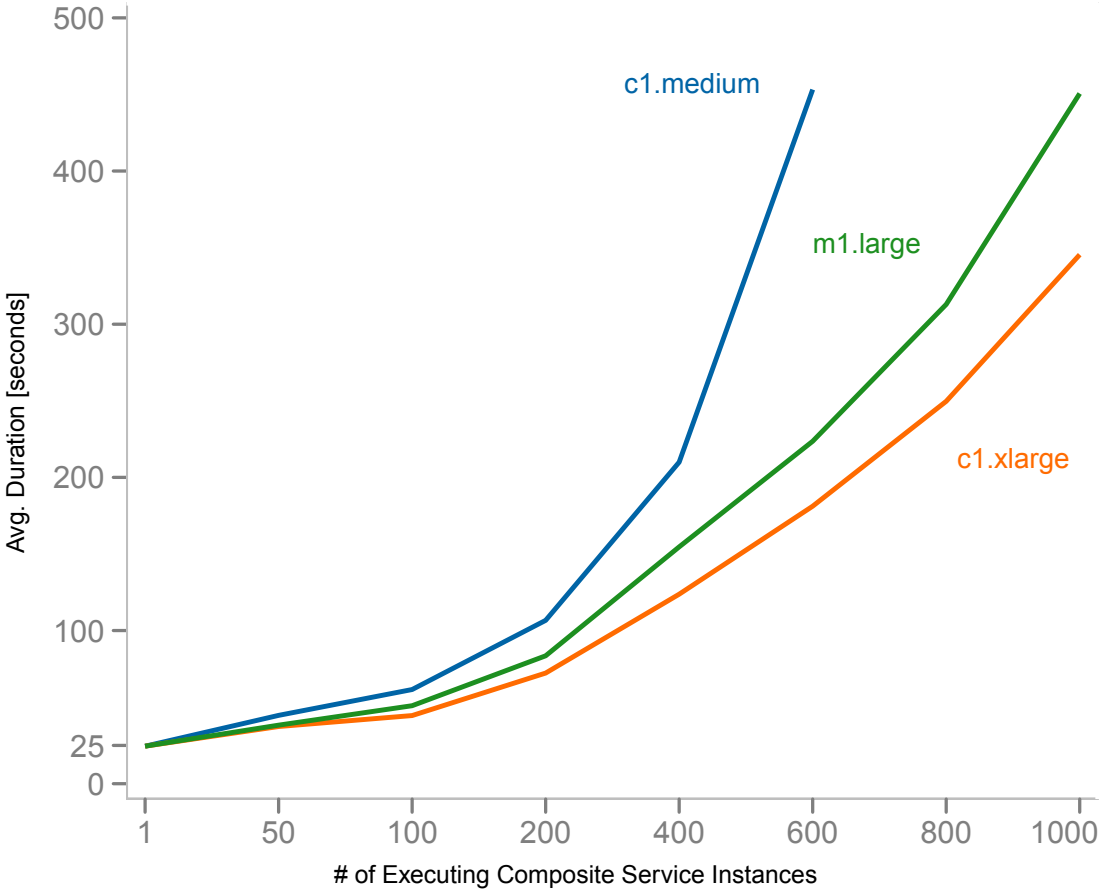


Figure 21: Average Execution Duration of a Composite Service with Increasing Concurrency

runtime and on the duration of executing concurrent composite service instances. The time spent to execute a composite service can be decreased by providing more resources, especially CPUs to the composite service runtime.

6.3 Resource Provisioning

The following experiments show the benefits of using the autonomic runtime in terms of SLA compliance. Three different resource provisioning scenarios were investigated. The autonomic runtime relocated itself from VM instances of type *c1.medium* to *m1.large*, *c1.medium* to *c1.xlarge* and *m1.large* to *c1.xlarge*. Each experiment involved invoking about 2000 composite service instances, 1000 before and 1000 after relocating the runtime, independently from the SLA violations detected so far. The lower rate, as shown in Table 12, was used to generate the load. The SLA on the execution duration of a composite service instance was defined at 35 seconds.

To illustrate savings, the results of the experiments are compared to results without resource provisioning and relocating. Violations without provisioning (column *Without Prov.* in Table 14)

		c1.medium	*m1.large*	**Total**	**Without Prov.**
Instances	#	1173	923	2096	2096
Violations	#	902	356	1258	1612
	%	76.9	38.6	60.0	76.9
Avg. Dur.	sec.	51.093	33.746	43.454	

Table 14: Relocation from *c1.medium* to *m1.large*

		c1.medium	*c1.xlarge*	**Total**	**Without Prov.**
Instances	#	1000	997	1997	1997
Violations	#	902	231	1133	1798
	%	90.0	23.2	56.7	90.0
Avg. Dur.	sec.	52.597	32.109	42.368	

Table 15: Relocation from *c1.medium* to *c1.xlarge*

are extrapolated by applying the percentage of violations measured before provisioning (column *c1.medium*) to the total number of instances (column *Total*). The difference between violations without provisioning and total violations represent savings.

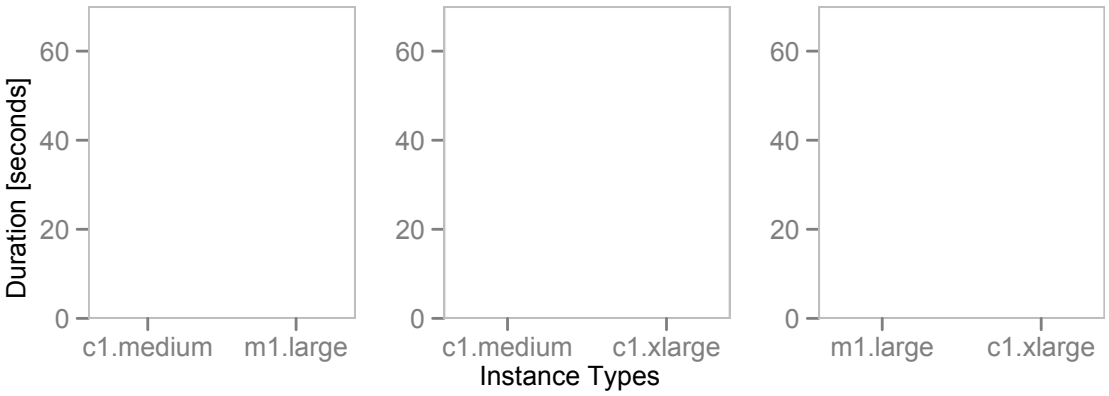
Table 14 shows the ratio of processed composite service instances and their violations within a runtime that relocated itself from *c1.medium* to *m1.large*. The 76.9% of violations on *c1.medium* were reduced to 38.6% on *m1.large*, which amounts to 60.0% violations in total. If all 2096 instances were executed without resource provisioning, the total amount of violations would have been increased to 1612 violations instead of 1258 violations. The benefit of relocating the runtime therefore amounts to approximately 22%.

The results of the experiment of relocating the runtime from *c1.medium* to *c1.xlarge* are presented in Table 15. This time the ratio between violated and completed composite service instances on *c1.medium* is higher. This results from resources being allocated differently in the Cloud or the OS on the VM instance than before. In contrast to 22% from the previous experiment, the improvement of resource provisioning now amounts to 37%. This arises from the high amount of SLA violatins on *c1.medium*. Even if there were fewer violations on *c1.medium*, a significant improvement would still be experienced.

As *m1.large* is equipped with more resources than *c1.medium*, the higher rate from Table 12 was used for the experiment with relocating the autonomic runtime from *m1.large* to *c1.xlarge* to force about the same amount of SLA violations as with the previous two experiments. The SLA was again defined at 35 seconds. Table 16 shows the results of this experiment. This time the improvement by provisioning amounts to 27%. Because the proportion of resources between *c1.medium* and *m1.large* is equal to *m1.large* and *c1.xlarge*, this improvement is almost the same as that of the first experiment shown in Table 14.

		m1.large	*c1.xlarge*	**Total**	**Without Prov.**
Instances	#	1074	1152	2226	2226
Violations	#	917	462	1379	1901
	%	85.4	40.1	61.9	85.3
Avg. Dur.	sec.	43.340	33.964	38.488	

Table 16: Relocation from *m1.large* to *c1.xlarge*



(a) (b) (c)

Figure 22: Durations of Composite Services Instances

Figure 22 visualizes the individual durations and their distribution on the three different instance types from the experiments presented in this section so far. Red instances violate the SLA, blues do not. As can be seen, the minimum duration on all instances types is about 25 seconds as was discussed in Section 6.2. Most composite service instances are reported above the SLA on the source runtime, and below this value on the destination runtime. While SLA violations on the destination runtime concentrate just strait above the limit, they are distributed over the whole y-axis on the source runtime.

6.3.1 Runtime Relocation Duration

To learn the duration of resource provisioning, whose longest part is the *Runtime Relocation Phase*, the time passed between the occurrence of the *ProvisioningResources* and *RuntimeStarted* events was calculated from their timestamps. Table 17 presents durations from various experiments.

As can be seen from these results, the duration of provisioning resources does not depend on the provisioned instance type. The procedure of how Eucalyptus launches a VM instances clarifies their occurrence. The EMI is specified by the client when requesting to launch a VM instance. First, Eucalyptus downloads the compressed and encrypted EMI from *Walrus* to the

		m1.large	*c1.xlarge*
Duration min.	Avg.	6:09	5:27
	Min.	4:59	5:02
	Max.	12:26	6:01

Table 17: Durations of *Runtime Relocation Phase* in Minutes

		# Migrations						
		16	**35**	**47**	**71**	**82**	**88**	**88**
Migration sec.	Avg.	0.8	3.3	3.5	4.0	4.2	7.3	9.9
	Min.	0.6	0.5	0.7	0.6	1.0	1.2	1.6
	Total	9.0	9.7	17.7	13.0	16.2	23.5	29.0
Execution sec.	Avg.	26.7	34.1	47.0	46.6	56.3	62.3	60.6
	Min.	24.3	28.0	38.2	32.3	44.4	49.7	44.5
	Max.	29.4	39.7	56.7	56.3	67.9	77.3	85.0
Violations	#	0	14	47	68	82	88	88

Table 18: Durations of Instance Migrations Without Load

node, hosting the instance, and stores it in the local cache of the node. Then it is decrypted and decompressed on this node into a file representing the disk of the VM instance. Because of the high amount of I/O that comes with reading from the network interface and interacting with local or remote disks, the duration of launching a VM instance strongly depends on the size of the EMI. If the EMI was already stored within the local cache of the node, the transfer over the network would not be necessary and the duration would be reduced drastically.

The EMI used to host the prototype of the autonomic runtime is 1.6 GB large and has a virtual disk size of 4 GB. The time to launch an instance from this EMI is around 5 to 6 minutes, if the EMI is stored in the cache as shown in Table 17.

6.3.2 Instance Migration Duration

Experiments on migrating a set of composite service instances between *c1.medium* and *c1.xlarge* resulted in durations shown in Table 18. First, the composite service was invoked multiple times. As soon as a certain amount of composite service instances were executing, their migration was triggered by relocating the autonomic runtime. To avoid affecting the migration procedure, no invocations occurred during the time of migration. The source, as well as the relocated runtime were able to use all their available resources for migrating and resuming composite service instances.

The minimum and the average duration for migrating a single instance, as well as the duration of the whole *Instance Migration Phase* vary. They depend on the number of migrated composite service instances. A higher amount of migrated instances seems to lead to longer

		c1.medium	*m1.large*	**Migrated**	**Total**	**Without Migr.**
Instances	#	631	601	77	1309	1309
Violations	#	555	420	77	1052	975
	%	88	69.8	100	80.4	74.5

Table 19: Migration from *c1.medium* to *m1.large*

durations because the relocated runtime immediately resumes migrated instances and therefore shares resources between a larger amount of already resumed instances and the migration procedure. Additionally, the more instances being migrated, the higher the effort of serializing and sending the instances on the source runtime.

Composite service instances are suspended during migration. The time consumed for their migration has a direct impact on their total execution duration. The longer they are suspended, the more likely do they violate the SLA defined on their execution duration. As can be seen in Table 18, when many instances were migrated, only few satisfied the SLA.

6.4 Resource Provisioning Cost

As discussed in the previous section, resource provisioning is afflicted with two issues. Launching a VM instance during relocation of the runtime lasts several minutes on the one hand and migrating composite service instances involves their suspension on the other hand. These issues increase the number of SLA violations. From now on they will be referred to as *costs*. Costs can be expressed in terms of additional SLA violations and are analyzed in this section.

6.4.1 Migration Cost

To see the *cost*, the three scenarios from Section 6.3 are considered. The durations in Table 18 were measured without invoking the composite service during migrations. This time, the load generator was used to continuously invoke the composite service even during migrations. The SLA was again configured to be 35 seconds. Additionally, a threshold of 200 violations was specified that triggered the provisioning of resources by the runtime.

Table 19 shows the migration of 77 composite service instances from *c1.medium* to *m1.large*. As all of them violated the SLA, the *cost* amounts to an additional 77 violations. Despite the cost, resource provisioning pays off, because the violation rate on *m1.large* is low enough to compensate the cost.

Similar results were measured when migrating from *c1.medium* to *c1.xlarge*, shown in Table 20. Cost amounts to 74 violations, which increases overall violations by 5.7 percentage points.

Even when migrating instances from *m1.large* to *c1.xlarge*, none of the migrated composite service instances was able to satisfy the SLA, as illustrated in Table 21. It uses the higher invocation rate from Table 12. The cost amounts to 69 violations.

		c1.medium	*c1.xlarge*	Migrated	Total	Without Migr.
Instances	#	605	630	74	1309	1309
Violations	#	511	154	74	739	665
	%	85.5	24.4	100	56.5	50.8

Table 20: Migration from *c1.medium* to *c1.xlarge*

		m1.large	*c1.xlarge*	Migrated	Total	Without Migr.
Instances	#	737	688	69	1494	1494
Violations	#	639	250	69	958	889
	%	86.7	36.3	100	64.1	59.5

Table 21: Migration from *m1.large* to *c1.xlarge*

Source	Dest.	Migrations	Avg.	Total
Instance Type		#	Dur. (sec.)	
c1.medium	*c1.large*	77	2.800	46.000
c1.medium	*c1.xlarge*	74	3.343	34.797
m1.large	*c1.xlarge*	69	5.499	13.830

Table 22: Durations of Instance Migrations During Load

The average suspended duration is show in Table 22. The *Instance Migration Phases* involving the *c1.medium* VMs lasted much longer than migration from *m1.large* to *c1.xlarge* because memory and CPU were already exhausted at the time of the migration.

In contrast to the results shown in Table 18, where some instances were able to satisfy the SLA, now all instances violated the SLA. Firstly, this was caused by using the load generator at a higher rate, resulting in a higher amount of migrated instances. Secondly, the load generator continued to invoke the composite service even during migrations. Because of this migrating and executing instances took much longer on the source as well as on the destination runtime.

Assuming all instances violate the SLA in a worst case scenario, inequation 6.1 defines whether instances should be migrated:

$$x_{source}/y_{source} < (x_{dst} + a)/(y_{dst} + a) \tag{6.1}$$

It will be satisfied if and only if 6.2 holds:

$$(x_{dst}y_{source} - x_{source}y_{dst})/(x_{source} - y_{source}) > a \tag{6.2}$$

where

x_{source}, x_{dst}	amount of completed instances on the source and the destination runtime
y_{source}, y_{dst}	amount of violating instances on the source and destination runtime
a	amount of migrated, and therefore violated, instances
$\frac{x_{source}}{y_{source}}$	ratio between completed and violated instances on the source runtime
$\frac{x_{dst}}{y_{dst}}$	ratio between completed and violated instances on the destination runtime

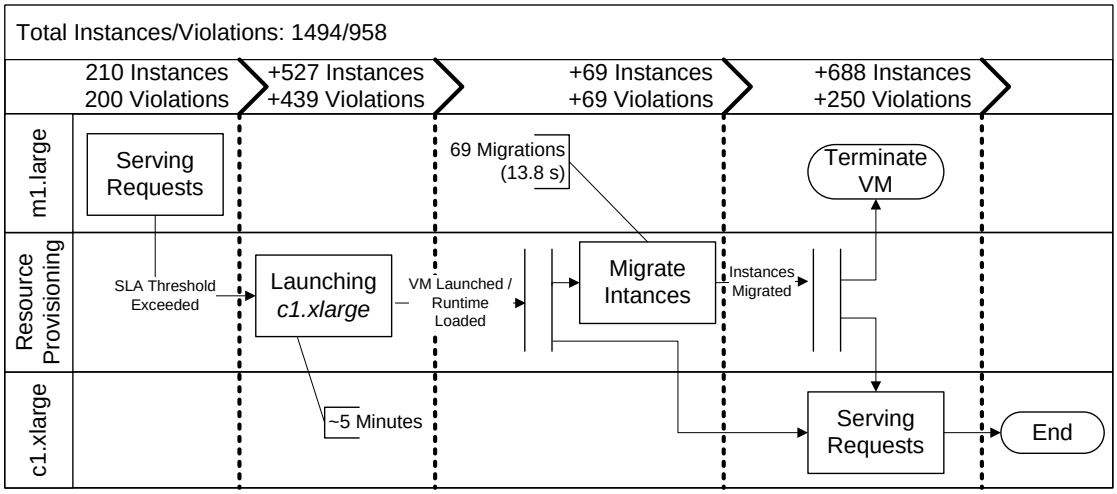


Figure 23: Resource Provisioning to *m1.large*

In other words, the level of SLA conformance on the relocated runtime has to be adequate to compensate violations caused by migration.

6.4.2 Relocation Cost

As can be seen in Figure 23, which illustrates Table 21, only 210 composite service instances executed until the threshold of 200 SLA violations exceeded. This corresponds with a 95% rating of violations. During a five minute resource provisioning, another 527 instances were processed. 439 violated the SLA, corresponding with a rating of 83% violations. When adding this up with migrations and violations occurred on *c1.xlarge*, 64% violations occured in total. If the time for resource provisioning was reduced to just one minute, approximately 105 instances would have been executed within this timespan on *m1.large* and 422 instances would have been executed on *c1.xlarge*. This amounts to 87 violations on *m1.large* and 151 violations on *c1.xlarge*. A shorter duration of resource provisioning significantly decreases the number of violated instances. As during resource provisioning, no additional resources are consumed by the currently executing runtime, the duration of relocation can be consider as not being too critical. While a shorter duration reduces the number of violations, a longer duration does not increase the ratio of violations. The number of violations during relocation can not be referred to as *cost*.

6.5 Evaluation Summary

After extensively studying the capabilities of the prototype by means of various experiments, the questions formulated at the beginning of this chapter, regarding the usefulness and limitations of the autonomic composite service runtime, can now be answered.

Will the overall amount of SLA violating composite service instances be decreased by provisioning resources to the composite service runtime? The amount of SLA violations could be decreased drastically by means of resource provisioning. Each provisioning scenario showed a significant reduction of violations after the autonomic runtime relocated itself to VM instances equipped with more resources. To measure the benefit of resource provisioning, the violations encountered with resource provisioning were compared to those when no resource provisioning was triggered. The experiments showed reductions of SLA violations between 22% and 37%. These experiments were ended after a specific amount of composite service instances was finished. It can be assumed that the reductions would increase the longer the experiments would last.

Can the violation of the SLA by a composite service instance be prevented by migrating this instance and in what circumstances is this possible? The minimum time measured for migration, and therefore suspension, was 500 milliseconds. This additional time adds to the duration of executing a composite service instance. To decrease the whole execution duration, this amount has to be compensated after migration by benefiting from a larger amount of available resources. While the duration of migrating a composite service instance is low when only a few instances are in need of migration, the duration increases with a growing amount of migrated instances. Summarized, these durations are subject to the amount of resources available to the source and the destination runtime for migration. The experiments showed, that if the autonomic runtime was exposed to composite service invocations during migration, none of the migrated composite service instances would satisfy the SLA.

What is the cost for provisioning resources and does the benefit outweigh the cost? The costs were defined to be the amount of additional SLA violations caused by resource provisioning. Because relocating the autonomic runtime does not utilize any of its resources, it is not affected by relocating itself. In contrast, migrating composite service instance utilizes resources provided to the autonomic runtime and hence impacts the level of SLA conformance. All experiments on resource provisioning conducted with the prototype, resulted in a high amount of prevented SLA violations which always outweighed additional SLA violation caused by migration. Therefore in any case it will turn to account to bear the cost for migration in order to minimize SLA violations.

CHAPTER 7

Conclusion and Future Work

Service compositions play a fundamental role in SOAs. They are exposed as Web services and are subject to guaranteed quality aspects in terms of SLAs. To ensure SLAs, a service provider has to align the amount of resources provided for hosting the service in accordance to his promised quality aspects. He needs to constantly monitor the current level of SLA compliance and take adequate actions in case of their violation. As the amount of resources demanded by a service may vary over time, it is hard to keep up with provisioning physical resources. The Cloud offers dynamic handling of resources and provides the base for their instant provisioning. This thesis proposed deploying a composite service to the Cloud.

Based on these capabilities of the Cloud, this thesis presented a solution for reducing SLA violations of composite services. It illustrated the design of a composite service runtime, able to monitor itself, thereby evaluating SLAs, based on an event-driven approach, and finally able to provide itself with additional resources needed to ensure a certain level of SLA compliance. This composite service runtime was designed to decide and act autonomically, based on predefined policies, without any human interaction. It runs on top of the open source Cloud platform Eucalyptus, which shares the same Cloud concepts as Amazon EC2 and provides a compatible interface.

To show the practicability and feasibility of this solution, a prototype was implemented as part of the practical work in this thesis, that implements the fundamental concepts of the proposed approach. Developing the prototype involved setting up its environment, including the VRESCo runtime environment for SoC, in the Cloud, which in turn required the extensive preparation of its deployment. To develop and evaluate the prototype, the Institute of Information Systems at the Vienna University of Technology provided its installation of Eucalyptus.

A general study, that was part of the evaluation in this thesis, demonstrated the impact of the amount of resources available to a composite service runtime on its performance and therefore the compliance with SLAs. Evaluating the prototype showed its capability to significantly reduce the amount of SLA violations by provisioning itself with resources in the Cloud. While the evaluation illustrated the costs connected to the resource provisioning procedure, these costs were always outweighed by the benefits in terms of preventing SLA violations.

7.1 Future Work

Both, the prototype and the solution itself, leave room for improvements in terms of flexibility and effectiveness. First of all, extending the resource provisioning procedure to support more accurate scaling would allow for a preciser selection and provisioning of resources. Currently, resource provisioning is limited to three different instance types. They are ranked by their amount of resources. More instance types could be introduced. Distinguishing between their individual characteristics, i.e., differentiating between memory- and CPU-rich instance types, could enable their provisioning according to particular needs. Additionally, it would be interesting to see how the proposed solution and the prototype can be extended and benefit from concepts provided by Amazon EC2 but missing in Eucalyptus. While Eucalyptus does not differentiate between the performance of I/O operations of instance types, instance types in Amazon EC2 are provided with different levels of I/O performance. Another interesting aspect of Amazon EC2 is their distribution of data centers around the world. A service may be relocated between different regions (e.g., from US to EU) to further reduce the distance between consumers and itself.

Further extensions of the resource provisioning procedure, deserving attention, comprise the runtime's ability to scale itself down by releasing resources currently not needed. Additionally, the specific vertical scaling approach, followed by the resource provisioning procedure, may as well be combined with a horizontal scaling approach, balancing the load of composite services among multiple runtime engines and simultaneously scaling individual runtime engines vertically as needed.

In terms of effectiveness, the proposed solution leaves room for optimizing the duration of resource provisioning and thereby further reducing the amount of SLA violations. The relocation routine depends on Eucalyptus fetching an EMI and preparing the virtual disk of the requested virtual machine instance. To reduce the deployment time of a virtual machine, the size of the EMI may either be reduced or the infrastructure hosting Eucalyptus may be upgraded to allow a better throughput of I/O operations. Additionally, it would be interesting to examine how this way of provisioning resources performs with Amazon EC2. On the other hand, future versions of Eucalyptus may improve the procedure of deploying virtual machines.

Besides further investigating composite service relocation, migrating running composite service instances deserves additional analysis as well. Evaluations showed, that the amount of migrated instances and the amount of available resources impacts the duration of migrating instances. It would be interesting to see whether the progress of a composition at the time of its migration, and to what extent the size of a serialized state, effect migration performance. Clearly, transferring the state over the network is subject to bandwidth and latency, but analyzing the performance of migrating composite services between data centers is interesting in terms of utilizing regions in Amazon EC2.

A List of Acronyms

AMI	Amazon Machine Image
API	Application Programming Interface
AWS	Amazon Web Services
CC	Cluster Controller
CEP	Complex Event Processing
CLC	Cloud Controller
CPU	Central Processing Unit
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DoS	Denial of Service
DSA	Distributed Shared Array
EAI	Enterprise Application Integration
EBS	Elastic Block Store
EC2	Elastic Compute Cloud
ECU	EC2 Compute Unit
EMI	Eucalyptus Machine Image
EPL	Event Processing Language
FTP	File Transfer Protocol
GUID	Globally Unique Identifier
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IaaS	Infrastructure as a Service
IP	Internet Protocol
IT	Information Technology
JVM	Java Virtual Machine
LAN	Local Area Network
NC	Node Controller
NIST	National Institute of Standards and Technology
OASIS	Organization for the Advancement of Structured Information Standards
OS	Operating System
PaaS	Platform as a Service
QoS	Quality of Service
RMI	Remote Method Invocation
S3	Simple Storage Service
SaaS	Software as a Service
SCSI	Small Computer System Interface
SDK	Software Development Kit
SLA	Service Level Agreement
SLO	Service Level Objective
SMTP	Simple Mail Transport Protocol
SOA	Service-oriented Architecture

SoC Service-oriented Computing
SQL Structured Query Language
SSL Secure Socket Layer
TCP Transmission Control Protocol
URL Uniform Resource Locater
UTC Coordinated Universal Time
VDC Virtualized Data Center
VEE Virtual Execution Environment
VM Virtual Machine
W3C World Wide Web Consortium
WCF Windows Communication Foundation
WF Windows Workflow Foundation
WSBPEL Web Services Business Process Execution Language
WSDL Web Service Description Language
XML eXtensible Markup Language

B SQL for Retrieving the Duration of Composite Service Execution

```
1  SELECT
2    begin.InstanceType,
3    ((end.wftimestamp - begin.wftimestamp)/1000) AS duration_ms,
4    begin.timestamp
5  FROM VRESCoEvent AS begin
6  INNER JOIN
7    (SELECT
8      Timestamp, wftimestamp, compositionid, instancetype
9    FROM VRESCoEvent
10   WHERE EVENTTYPE = 'WorkflowTrackingEvent'
11     AND WorkflowState = 'Completed'
12   ) AS end
13 ON begin.compositionid = end.compositionid
14 WHERE begin.EVENTTYPE = 'WorkflowTrackingEvent'
15   AND WorkflowState = 'Created'
16   AND begin.instancetype != end.instancetype;
```

Listing 13: SQL for composite service duration

C SQL for Retrieving the Time for Resource Provisioning

```
1  SELECT
2    begin.instancetype, end.instancetype,
3    ((end.wftimestamp - begin.wftimestamp)/60000) AS duration_minutes
4  FROM VRESCoEvent AS begin
5  INNER JOIN
6    (SELECT
7      Timestamp, wftimestamp, engineid, instancetype, state
8    FROM VRESCoEvent
9    WHERE EVENTTYPE = 'CompositionEngineEvent'
10     AND State = 'migrationFinished'
11   ) AS end
12 ON begin.engineid = end.engineid
13 WHERE begin.EVENTTYPE = 'CompositionEngineEvent'
14   AND begin.State = 'provisioningResources';
```

Listing 14: SQL for resource provisioning duration

D Composite Service of the Evaluation Scenario

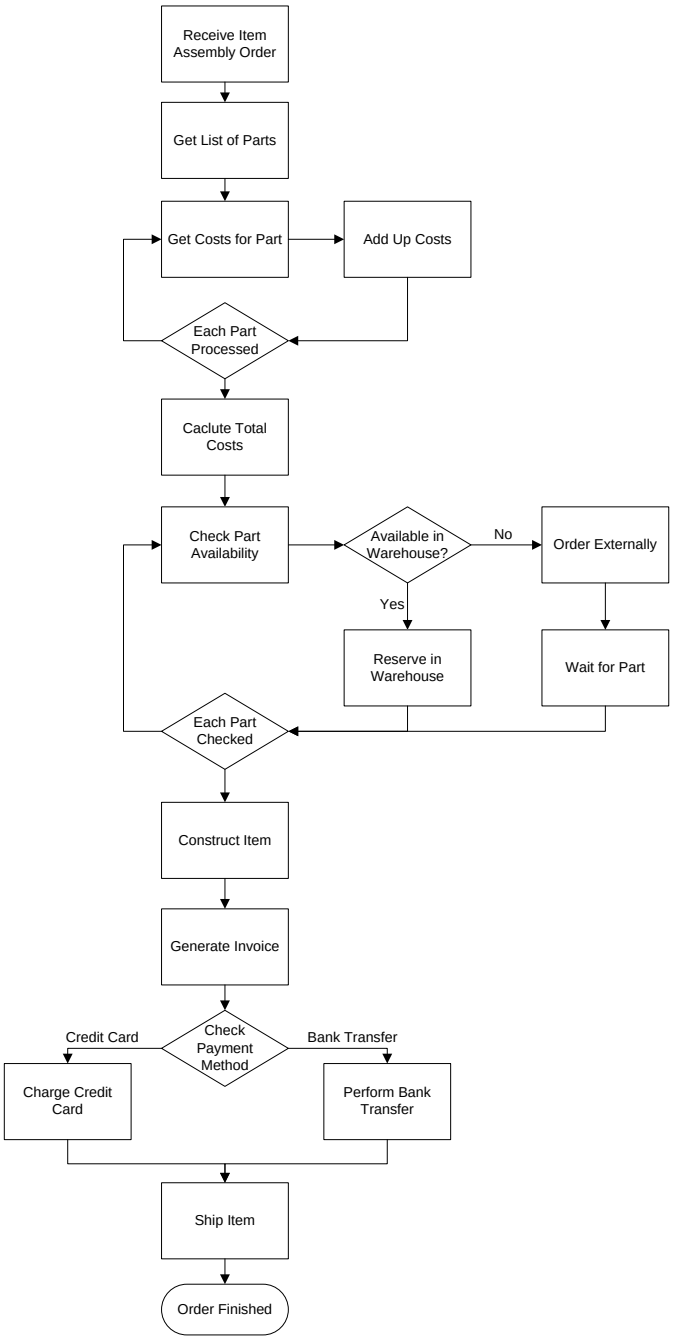


Figure 24: Composite Service Workflow

Bibliography

[1] Active Endpoints Inc. ActiveVOS. http://www.activevos.com/, 2011. Visited: 2011-09-15.

[2] ajmf. Running Windows on Eucalyptus (Improved). http://ajmf.wordpress.com/2009/10/14/running-windows-on-eucalyptus-improved/, 2009. Visited: 2011-09-15.

[3] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.

[4] Amazon Web Services LLC. Amazon EC2 Instance Types. http://aws.amazon.com/ec2/instance-types. Visited: 2011-09-23.

[5] Amazon Web Services LLC. Amazon EC2 Pricing. http://aws.amazon.com/ec2/pricing. Visited: 2011-09-23.

[6] Amazon Web Services LLC. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/. Visited: 2011-09-10.

[7] Amazon Web Services LLC. Amazon Web Services. http://aws.amazon.com/. Visited: 2011-09-10.

[8] Amazon Web Services LLC. Amazon Auto Scaling. http://aws.amazon.com/de/autoscaling/, 2011. Visited: 2011-09-10.

[9] Amazon Web Services LLC. AWS SDK for .NET. http://aws.amazon.com/de/sdkfornet/, 2011. Visited: 2011-09-15.

[10] Apache Friends. XAMPP Webpage. http://www.apachefriends.org/xampp.html, 2011. Visited: 2011-09-15.

[11] Apache Software Foundation. Apache ODE. http://ode.apache.org/, 2011. Visited: 2011-09-15.

[12] Arjuna, BEA, Hitachi, IBM, IONA, and Microsoft. Web Services Transactions specifications. http://www.ibm.com/developerworks/library/specification/ws-tx/.

[13] Michael Armbrust et al. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[14] Christoph Arnold, Michel Rode, Jan Sperling, and Andreas Steil. *KVM Best Practices*. dpunkt Verlag, 2011.

[15] Colin Barker. HP dismisses cloud 'hype'. http://www.zdnet.com/news/hp-dismisses-cloud-hype/255222, December 2008. Visited: 2011-09-16.

[16] Ramzi Basharahil, Brian Wims, Cheng-Zhong Xu, and Song Fu. Distributed Shared Arrays: An Integration of Message Passing and Multithreading on SMP Clusters. *Journal of Supercomputing*, 31:161–184, February 2005.

[17] Christian Baun, Marcel Kunze, Jens Nimis, and Stefan Tai. *Cloud Computing, Web-basierte dynamische IT-Services*. Springer-Verlag Berlin Heidelberg, 2011.

[18] P. Bhoj, S. Singhal, and S. Chutani. SLA management in federated environments. *Computer Networks*, 35:5–24, January 2001.

[19] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *IM'07: Proceedings of 10th IFIP/IEEE International Symposium on Integrated Network Management*, 2007.

[20] Bruce Bukovics. *Pro WF: Windows Workflow in .NET 3.5*. Apress Series. Apress, 2008.

[21] Michele LeRoux Bustamante. *Learning WCF*. O'Reilly Media, Inc., 2007.

[22] Rajkumar Buyya et al. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009.

[23] Valeria Cardellini, Michele Colajanni, and Philip S. Yu. Dynamic Load Balancing on Web-Server Systems. *IEEE Internet Computing*, 3:28–39, May 1999.

[24] Malu Castellanos, Fabio Casati, Umeshwar Dayal, and Ming-Chien Shan. Intelligent Management of SLAs for Composite Web Services. In Nadia Bianchi-Berthouze, editor, *Databases in Networked Information Systems*, volume 2822 of *Lecture Notes in Computer Science*, pages 158–171. Springer Berlin / Heidelberg, 2003.

[25] Christopher Clark et al. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, 2005.

[26] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[27] David S. Linthicum. *Enterprise Application Integration*. Addison-Wesley, 1999.

[28] Jimmy Desai. *Service Level Agreements: A Legal and Practical Guide*. IT Governance Publishing, 2010.

[29] Tim Dornemann, Ernst Juhnke, and Bernd Freisleben. On-Demand Resource Provisioning for BPEL Workflows Using Amazon's Elastic Compute Cloud. In *CCGRID'09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2009.

[30] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1:1–30, August 2005.

[31] Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, 2005.

[32] Thomas Erl. *SOA: principles of service design*. Prentice Hall, 2008.

[33] Thomas Erl et al. *Web Service Contract Design and Versioning for SOA*. The Prentice-Hall service-oriented computing series from Thomas Erl. Prentice Hall, 2008.

[34] Esper contributors & EsperTech Inc. Esper. `http://esper.codehaus.org/`. Visited: 2011-09-02.

[35] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning, August 2010.

[36] Eucalyptus Systems Inc. Eucalyptus Open Source Cloud Platform. `http://open.eucalyptus.com/`. Visited: 2011-10-04.

[37] Eucalyptus Systems Inc. Eucalyptus Open-Source Cloud Computing Infrastructure - An Overview. `http://www.eucalyptus.com/resources/whitepapers`, August 2009. Visited: 2011-09-02.

[38] Eucalyptus Systems Inc. Eucalyptus - Accessing Instance Metadata. `http://open.eucalyptus.com/participate/wiki/accessing-instance-metadata`, 2011. Visited: 2011-09-15.

[39] Eucalyptus Systems Inc. Eucalyptus 3.0 Roadmap. `http://open.eucalyptus.com/participate/roadmaps/eucalyptus_3.0`, 2011. Visited: 2011-10-02.

[40] Eucalyptus Systems Inc. Using VNC to debug an image. `http://open.eucalyptus.com/participate/wiki/using-vnc-debug-image`, 2011. Visited: 2011-09-15.

[41] Dan Farber. Oracle's Ellison nails cloud computing. `http://news.cnet.com/8301-13953_3-10052188-80.html`, September 2008. Visited: 2011-09-16.

[42] Marvin Ferber, Sascha Hunold, and Thomas Rauber. Load Balancing Concurrent BPEL Processes by Dynamic Selection of Web Service Endpoints. In *ICPPW'09: Proceedings of the 2009 International Conference on Parallel Processing Workshops*, 2009.

[43] findleyd. Set hardware clock to UTC on Windows. `http://weblogs.asp.net/dfindley/archive/2006/06/20/Set-hardware-clock-to-UTC-on-Windows-_2800_or-how-to-make-the-clock-work-on-a-Mac-Book-Pro_2900_.aspx`, 2006. Visited: 2011-09-15.

[44] Song Fu and Cheng-Zhong Xu. Service migration in distributed virtual machines for adaptive grid computing. In *ICPP'05: International Conference on Parallel Processing, 2005*, pages 358–365, June 2005.

[45] Alessio Gambi, Mauro Pezze, and Michal Young. Sla protection models for virtualized data centers. In *SEAMS'09: ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems 2009*, pages 10–19, May 2009.

[46] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[47] Chunye Gong et al. The Characteristics of Cloud Computing. In *ICPPW'10: Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, pages 275–279, Washington, DC, USA, 2010. IEEE Computer Society.

[48] Google Inc. Google App Engine. `http://code.google.com/appengine/`. Visited: 2011-10-04.

[49] Edward Haletky. *VMware ESX and ESXi in the Enterprise: Planning Deployment of Virtualization Servers (2nd Edition)*. Prentice Hall, 2011.

[50] Haydn Solomon. How do you use e1000 option on a windows Guest? `http://www.linux-kvm.com/content/how-do-you-use-e1000-option-windows-guest`, 2008. Visited: 2011-09-15.

[51] Paul Horn. Autonomic computing: IBM's Perspective on the State of Information Technology. `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf`, 2001.

[52] Waldemar Hummer et al. *VRESCo - Vienna Runtime Environment for Service-oriented Computing*, pages 299–324. Service Engineering. European Research Results. Springer, 2010.

[53] Intel Corporation. Drivers and software for Intel Gigabit and PRO/1000 Wired Ethernet Adapters. `http://www.intel.com/SUPPORT/NETWORK/SB/CS-006120.HTM`, 2011. Visited: 2011-09-15.

[54] Xuxian Jiang and Dongyan Xu. VIOLIN: Virtual Internetworking on Overlay Infrastructure. In *Proceedings of the 2nd Snternational Symposium on Parallel and Distributed Processing and Applications*, pages 937–946, 2003.

[55] Li jie Jin, Vijay Machiraju, and Akhil Sahai. Analysis on Service Level Agreement of Web Services. Technical report, HP Laboratories, 2002. Visited: 2011-09-14.

[56] John Rhoton. *Cloud Computing Explained: Implementation Handbook for Enterprises*. Recursive Press, 2009.

[57] Alexander Keller and Heiko Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11:57–81, March 2003.

[58] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36:41–50, January 2003.

[59] Gunjan Khanna, Kirk Beaty, Gautam Kar, and Andrzej Kochut. Application Performance Management in Virtualized Server Environments. In *NOMS'06: 10th IEEE/IFIP Network Operations and Management Symposium*, pages 373 –381, april 2006.

[60] kiranmurari. UEC: Bundling Windows Image. `http://kiranmurari.wordpress.com/2010/03/29/uec-bundling-windows-image/`, 2010. Visited: 2011-09-15.

[61] Sriram Krishnan. *Programming Windows Azure: Programming the Microsoft Cloud*. O'Reilly Media, 2010.

[62] Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. Monitoring, Prediction and Prevention of SLA Violations in Composite Services. In *ICWS'10: Proceedings of the 2010 IEEE International Conference on Web Services*, pages 369–376, 2010.

[63] Philipp Leitner, Florian Rosenberg, and Schahram Dustdar. Daios: Efficient Dynamic Web Service Invocation. *IEEE Internet Computing*, 13:72–80, May 2009.

[64] Alexander Lenk et al. What's inside the Cloud? An architectural map of the Cloud landscape. In *CLOUD'09: Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 23–31, Washington, DC, USA, 2009. IEEE Computer Society.

[65] Jesse Liberty and Alex Horovitz. *Programming .NET 3.5*. O'Reilly Media, Inc., 2008.

[66] Michael Litzkow and Marvin Solomon. Supporting checkpointing and process migration outside the Unix kernel. In *Proceedings of the USENIX Winter Conference*, pages 283–290, 1992.

[67] Huan Liu and Sewook Wee. Web Server Farm in the Cloud: Performance Evaluation and Dynamic Architecture. In *CloudCom'09: Proceedings of the 1st International Conference on Cloud Computing*, pages 369–380, Berlin, Heidelberg, 2009. Springer-Verlag.

[68] Ru-Yue Ma et al. Grid-Enabled Workflow Management System Based On BPEL. *International Journal of High Performance Computing Applications*, 22:238–249, August 2008.

[69] Anbazhagan Mani and Arun Nagarajan. Understanding quality of service for Web services. `http://www.ibm.com/developerworks/library/ws-quality.html`, January 2002.

[70] Daniel A. Menascé. QoS Issues in Web Services. *IEEE Internet Computing*, 6:72–75, November 2002.

[71] Brenda M. Michelson. Event-Driven Architecture Overview. `http://www.omg.org/soa/Uploaded%20Docs/EDA/bda2-2-06cc.pdf`, 2006. Visited: 2011-09-25.

[72] Anton Michlmayr et al. Towards recovering the broken SOA triangle: a software engineering perspective. In *IW-SOSWE'07: Proceedings of the 2nd international workshop on Service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting*, pages 22–28, New York, NY, USA, 2007. ACM.

[73] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Advanced event processing and notifications in service runtime environments. In *DEBS'08: Proceedings of the second international conference on Distributed event-based systems*, pages 115–125, New York, NY, USA, 2008. ACM.

[74] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Comprehensive QoS monitoring of Web services and event-based SLA violation detection. In *MWSOC'09: Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing*, pages 1–6, New York, NY, USA, 2009. ACM.

[75] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. End-to-End Support for QoS-Aware Service Selection, Binding, and Mediation in VRESCo. *IEEE Transactions on Services Computing*, 3:193–205, July 2010.

[76] Microsoft Corporation. How to create a bootable floppy disk for an NTFS or FAT partition in Windows XP. `http://support.microsoft.com/kb/305595`, 2007. Visited: 2011-09-15.

[77] Microsoft Corporation. How to use the remote desktop feature of windows xp professional. `http://support.microsoft.com/kb/315328/en`, 2007. Visited: 2011-09-12.

[78] Microsoft Corporation. Description of TCP/IP settings that you may have to adjust when SQL Server connection pooling is disabled. `http://support.microsoft.com/kb/328476`, 2009. Visited: 2011-09-15.

[79] Microsoft Corporation. Processor and memory capabilities of Windows XP Professional x64 Edition and of the x64-based versions of Windows Server 2003. `http://support.microsoft.com/kb/888732`, 2010. Visited: 2011-09-15.

[80] Microsoft Corporation. Microsoft Windows Server 2003. `http://www.microsoft.com/germany/windowsserver2003/default.mspx`, 2011. Visited: 2011-09-15.

[81] Microsoft Corporation. TechNet Library: MaxUserPort. http://technet.microsoft.com/en-us/library/cc938196.aspx, 2011. Visited: 2011-09-12.

[82] Microsoft Corporation. TechNet Library: TcpTimedWaitDelay. http://technet.microsoft.com/en-us/library/cc938217.aspx, 2011. Visited: 2011-09-12.

[83] Microsoft Corporation. The Official Microsoft IIS Site. http://www.iis.net/, 2011. Visited: 2011-09-15.

[84] Microsoft Corporation. Windows XP - Microsoft Windows. http://windows.microsoft.com/de-AT/windows/products/windows-xp, 2011. Visited: 2011-09-15.

[85] Dejan S. Milojičić et al. Process migration. *ACM Computing Surveys*, 32:241–299, September 2000.

[86] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. Event driven monitoring for service composition infrastructures. In *WISE'10: Proceedings of the 11th international conference on Web information systems engineering*, pages 38–51, Berlin, Heidelberg, 2010. Springer-Verlag.

[87] Adina Mosincat and Walter Binder. Automated maintenance of service compositions with SLA violation detection and dynamic binding. *International Journal on Software Tools for Technology Transfer*, 13:167–179, April 2011.

[88] Richard Murch. *Autonomic Computing*. IBM Press, 2004.

[89] National Institute of Standards and Technology (NIST). The nist definition of cloud computing (draft). http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf, January 2011. Visited: 2011-09-11.

[90] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[91] Daniel Nurmi et al. Eucalyptus: A Technical Report on an Elastic Utility Computing Architecture Linking Your Programs to Useful Systems. *UCSB Computer Science Technical Report Number 2008-10*, August 2008.

[92] Open Grid Forum (OGF). Web Services Agreement Specification (WS-Agreement). http://forge.gridforum.org/projects/graap-wg/. Visited: 2011-09-14.

[93] Open Grid Forum (OGF). WS-Agreement Negotation. http://forge.gridforum.org/projects/graap-wg/. Visited: 2011-09-14.

[94] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, May 2006.

[95] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Coordination (WS-Coordination). http://docs.oasis-open.org/ws-tx/wscoor/2006/06, 2009. Visited: 2011-09-15.

[96] Michael P. Papazoglou. *Web Services: Principles and Technology*. Prentice Hall, 2008.

[97] Harmon Paul. Analysing activities. *Business Process Trends*, 1(4):1–13, April 2003.

[98] Chris Peltz. Web Services Orchestration and Choreography. *Computer*, 36:46–52, October 2003.

[99] Ken Pepple. *Deploying OpenStack*. O'Reilly Media, 2011.

[100] Nick Randolph and David Gardner. *Professional Visual Studio 2008*. Wrox Press Ltd., 2008.

[101] Kevin Roebuck. *Virtual Network Computing (Vnc): High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors*. Tebbo, 2011.

[102] Roman Kiss. WS-Eventing for WCF (Indigo). http://www.codeproject.com/KB/WCF/WSEventing.aspx, 2006. Visited: 2011-09-15.

[103] Florian Rosenberg, Philipp Leitner, Anton Michlmayr, and Schahram Dustdar. Integrated Metadata Support for Web Service Runtimes. In *Proceedings of the 2008 12th Enterprise Distributed Object Computing Conference Workshops*, pages 361–368, Washington, DC, USA, 2008. IEEE Computer Society.

[104] Igor Rosenberg, Antonio Conguista, and Roland Kuebert. Management for Service Level Agreements. In *Service Oriented Infrastructures and Cloud Service Platforms for the Enterprise*, pages 103–124. Springer Berlin Heidelberg, 2010.

[105] Paul Ruth et al. Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure. In *ICAC'06: IEEE International Conference on Autonomic Computing 2006*, pages 5–14, June 2006.

[106] Jon Skeet. *C# in Depth, Second Edition*. Manning Publications Co., Greenwich, CT, USA, 2010.

[107] Lambert M. Surhone, Mariam T. Tennoe, and Susan F. Henssonow. *OpenNebula*. Betascript Publishing, 2011.

[108] Chris Takemura and Luke S. Crawford. *The Book of Xen: A Practical Guide for the System Administrator*. No Starch Press, 2009.

[109] Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41:45–52, January 2011.

[110] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39:50–55, December 2008.

[111] W. Vogels. A head in the clouds-the power of infrastructure as a service. In First workshop on Cloud Computing and in Applications (CCA '08), October 2008.

[112] World Wide Web Consortium (W3C). Web Services Architecture. `http://www.w3.org/TR/ws-arch/`, February 2004. Visited: 2011-09-17.

[113] World Wide Web Consortium (W3C). Web Services Glossary. `http://www.w3.org/TR/ws-gloss/`, February 2004. Visited: 2011-09-17.

[114] World Wide Web Consortium (W3C). Web Services Choreography Description Language Version 1.0. World Wide Web Consortium, Candidate Recommendation CR-ws-cdl-10-20051109, November 2005.

[115] World Wide Web Consortium (W3C). Web Services Eventing (WS-Eventing). `http://www.w3.org/Submission/WS-Eventing/`, March 2006. Visited: 2011-09-17.

[116] World Wide Web Consortium (W3C). SOAP Version 1.2. `http://www.w3.org/TR/soap12-part1/`, 2007. Visited: 2011-09-15.

[117] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) 2.0. http://www.w3.org/TR/wsdl20/, 2007.

[118] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, 2010.

[119] Songnian Zhou, Xiaohu Zheng, Jingwen Wang, and Pierre Delisle. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. *Software: Practice and Experience*, 23:1305–1336, December 1993.