Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU WIEN UB Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Cloud Computing Application Adaptation as a Service

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Michael Osl
Matrikelnummer 0828763

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Mag. Dr. Schahram Dustdar
Mitwirkung: Dipl.-Ing. Dr. Waldemar Hummer
Dipl.-Ing. Dr. Christian Inzinger

Wien, 28.11.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Cloud Computing Application Adaptation as a Service

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Michael Osl
Registration Number 0828763

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Mag. Dr. Schahram Dustdar
Assistance: Dipl.-Ing. Dr. Waldemar Hummer
Dipl.-Ing. Dr. Christian Inzinger

Vienna, 28.11.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Michael Osl
Brestelgasse 12/17, 1160 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)

Acknowledgements

First and foremost, I thank Waldemar Hummer and Christian Inzinger. Not only did they come up with the initial idea for this thesis. With their helpful, constructive and prompt feedback they guided me in the right direction whenever I was stuck on the wrong way.

For advising this thesis I also want to thank the head of the Distributed Systems Group, Prof. Schahram Dustdar.

It was a great and insightful opportunity for me to write this thesis at such a committed and productive academic research group.

A special thank goes to Mark Poreda, who provided me with a server to run the evaluation tests. Because besides all the virtuality in cloud computing, in the end you still need some good old solid hardware to run your tests on. I am also grateful to Mike Heininger for providing valuable feedback on this thesis.

Finally, there are plenty of other people I am thankful for their indirect contributions to this thesis. To make it short: Thank you all.

Abstract

In cloud computing, orchestration languages allow a user to specify the static structure of resource compositions in the form of templates. When a user submits the template to the orchestration service, the provider creates the specified resource stack on behalf of the user. Despite their usefulness, existing template languages provide only limited support to specify provider managed, elastic run-time behavior of cloud computing resources. Usually, this support is limited to scaling on the resource level and "if-then" rules. A user who needs more fine grained elasticity (e.g. on the application level) must implement the capabilities on his or her own.

In this thesis we explore the possibilities to extend orchestration template languages so that a user can specify more fine grained elastic behavior with them. To support the elastic behavior, we research the question how a provider can offer a managed service that supports the extended orchestration templates within an existing cloud infrastructure. Finally, we explore the question if it is possible for the provider to utilize the monitored data from the client applications in order to derive adaptive decisions.

We implement a prototype of a provider managed cloud computing application adaptation service based on the open source cloud computing platform OpenStack. The presented prototype consists of a plug-in for the orchestration service OpenStack Heat, a component that monitors and manages client stacks by adjusting configurations in an autonomic way, a service that collects observation points from applications and provides the configurations for the application as well as a client agent that transmits observation points to the service.

We evaluate the prototype both with a simulator and a real world scenario to demonstrate that it is possible to extend an existing orchestration language in order to support elastic runtime behavior. We also show that it is possible to integrate such an extension seamlessly into OpenStack Heat with resource plug-ins. Finally, our evaluation demonstrates that the autonomic manager can use the set of collective observation points to derive reasonable configuration results with respect to user defined application objectives using a distance based algorithm.

Kurzfassung

Orchestrierungssprachen ermöglichen es Anwenderinnen und Anwendern von Cloud-Computing-Diensten, die statische Struktur von Ressourcen-Kompositionen in Form von Vorlagen (engl. *Templates*) zu spezifizieren. Übermittelt eine Anwenderin oder ein Anwender eine solche Vorlage an den Orchestrierungsdienst, erstellt der Anbieter den spezifizierten Stack von Ressourcen für die Anwenderin oder den Anwender. Trotz ihrer Nützlichkeit verfügen diese Orchestrierungssprachen jedoch nur über eingeschränkte Möglichkeiten, vom Anbieter verwaltetes, elastisches Laufzeitverhalten von Cloud-Computing-Ressourcen festzulegen. In der Regel beschränkt sich diese Unterstützung auf Skalierung auf Ressourcen-Ebene und "Wenn-Dann"-Regeln. Benötigt eine Anwenderin oder ein Anwender eine detailliertere Kontrolle über das elastische Laufzeitverhalten (z. B. auf Anwendungsebene), so muss sie oder er diese Fähigkeiten selbst implementieren.

In dieser Diplomarbeit erkunden wir die Möglichkeiten, Orchestrierungssprachen so zu erweitern, dass es einer Anwenderin oder einem Anwender möglich wird, detailliertes elastisches Verhalten zu spezifizieren. Um das elastische Verhalten zu unterstützen, untersuchen wir die Frage, wie ein Anbieter innerhalb einer existierenden Cloud-Infrastruktur einen Dienst zur Verfügung stellen kann, der die erweiterten Orchestrierungs-Templates unterstützt. Abschließend befassen wir uns mit der Frage, ob die gesammelten Daten der Stack-Anwendungen vom Anbieter verwendet werden können, um daraus adaptive Entscheidungen ableiten zu können.

Wir entwickeln dazu einen Prototyp eines vom Cloudanbieter verwalteten Adaptierungs-Dienstes auf Basis der quelloffenen Cloud-Computing-Plattform OpenStack. Der Prototyp besteht aus einem Plug-In für den Orchestrierungs-Dienst OpenStack Heat; einer Komponente, welche den Stack überwacht und autonom Konfigurationen anpasst; einem Dienst, welcher Beobachtungspunkte der Anwendung sammelt und die Konfigurationen bereitstellt sowie einem Programm, welches die Beobachtungspunkte an den Dienst übermittelt.

Wir evaluieren den Prototypen sowohl mit einem Simulator als auch einem realen Anwendungs-Szenario, um zu zeigen, dass es möglich ist, bestehende Orchestrierungssprachen um elastische Gesichtspunkte zu erweitern. Ebenso zeigen wir, dass sich eine solche Erweiterung mittels Ressourcen-Plug-Ins nahtlos in OpenStack Heat einfügen lässt. Abschließend demonstriert unsere Evaluierung, dass man die gesammelten Beobachtungspunkte dazu nutzen kann, mit einem distanzbasiertem Algorithmus sinnvolle Konfigurationen abzuleiten, um von der Anwenderin oder vom Anwender definierte Vorgaben einzuhalten.

Contents

CHAPTER 1

Introduction

The key characteristic of cloud computing is the on-demand, pay-per-use computation model. This model eliminates the need for upfront investments and long term commitments [5]. Cloud computing brings a shift in paradigm that views the consumption of computational capacity no longer as closely tied to the underlying physical entities such as servers or network switches. Instead, it abstracts these physical entities away into virtual entities. Consequently, *cloud providers* offer the resources in cloud computing as virtually unlimited *utilities* on demand to *cloud users*, following the analogy of other every day utilities such as electricity or water [15]. Binz et al. subsume the "*essence*" of cloud computing as the "*industrialization of IT*" [9]. Although being no new concepts per se, the cloud computing model fosters the creation of *scalable* and *elastic* systems. In this context, scalability refers to the static property of a system to support growth by being able to add more resources [35]. Elasticity, on the other hand, is the dynamic property of a computer system to adapt itself autonomously to external influences, such as the current demand [26].

After having received an initial "hype" [15], cloud computing has by now become a well established, multi-billion dollar market [50]. The wealth of commercial cloud providers reflects this development, as they are able to offer cloud computing capacities as mature, publicly available products to their users. Examples of such commercial cloud computing platforms are Amazon Web Services (AWS), Google App Engine and Microsoft Azure. But also in academia cloud computing has been receiving increased attention in recent years, as the plethora of scientific conferences, journals and published papers on the topic cloud computing demonstrate. For instance, a search in the DBLP Computer Science Bibliography returns more than 3 000 results for the term "cloud computing", with a steadily increasing trend between the years 2009 and 2013 [23].

Two often used classifications in the literature distinguish cloud computing services between the level of responsibility and the accessibility of the services. Although Armbrust et al. disputes this classification due to its fuzziness [5] and Hilley et al. claim that "*no two experts seem to agree*" on the terms [36], the "*XaaS*" terminology describes the different levels of responsibility, with *X* standing for the particular service layer. In that sense, Infrastructure as a Service (IaaS)

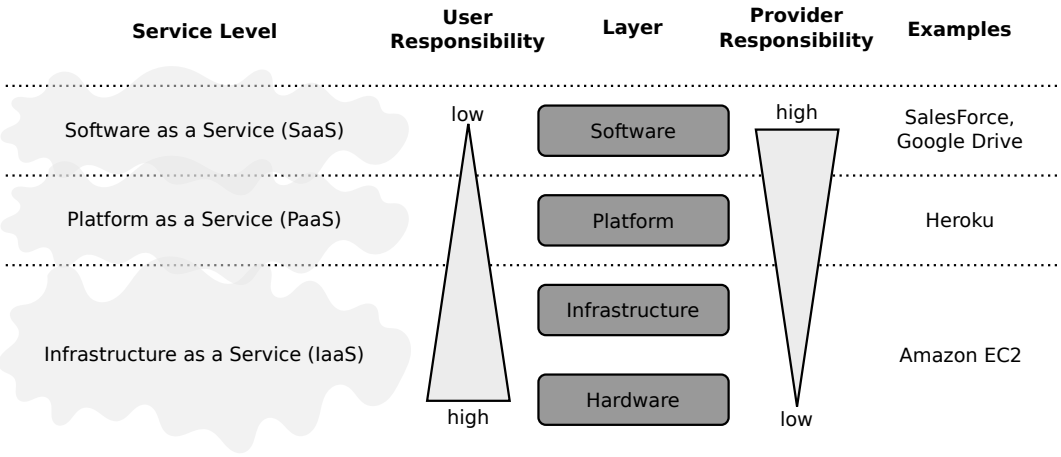


Figure 1.1: An illustration of the different layers in cloud computing. The overlapping clouds illustrate the "fuzziness" of the terms. Image based on [85].

describes the bottom layer that provides resources at the (virtual) hardware level. On top of this layer Platform as a Service (PaaS) offers provider managed system components, such as database services. Finally, Software as a Service (SaaS) offers ready to use web applications, where the provider bears most of the responsibility for the operation. Figure 1.1 depicts this layered architecture. The other classification of cloud computing services describes the level of access. This classification differentiates between *public*, *private* and *hybrid* clouds [85]. As the name implies, a public cloud makes its services available to a general audience. On the other side of the spectrum are private clouds, where an organization operates the cloud for its own internal use. Finally, the term *hybrid* cloud describes a deployment that utilizes elements of both public and private clouds.

1.1 Motivation

In cloud computing, *stacks* organize individual resources into functional entities. Such stacks consist of interconnected resources and services. Figure 1.2 illustrates an example of such a stack that drives a typical web application. The depicted stack comprises a load balancer that distributes the load to the web servers, which in turn query the data from a master/slave database back-end. Setting up such a stack out of standardized resources in order to become ready to serve its intended purpose involves two core tasks: The *orchestration* of individual resources to a stack and the *configuration* of the individual resources, both of which we describe now in more detail.

In general, the orchestration of services handles the composition of independent services into larger functional entities [71]. To enable resource orchestration for its users, AWS offers the CloudFormation service [3]. This service allows to formally specify a stack with the help of template files. These template files contain lists of resources along with their parameters and connections in a human readable JavaScript Object Notation (JSON) based configuration

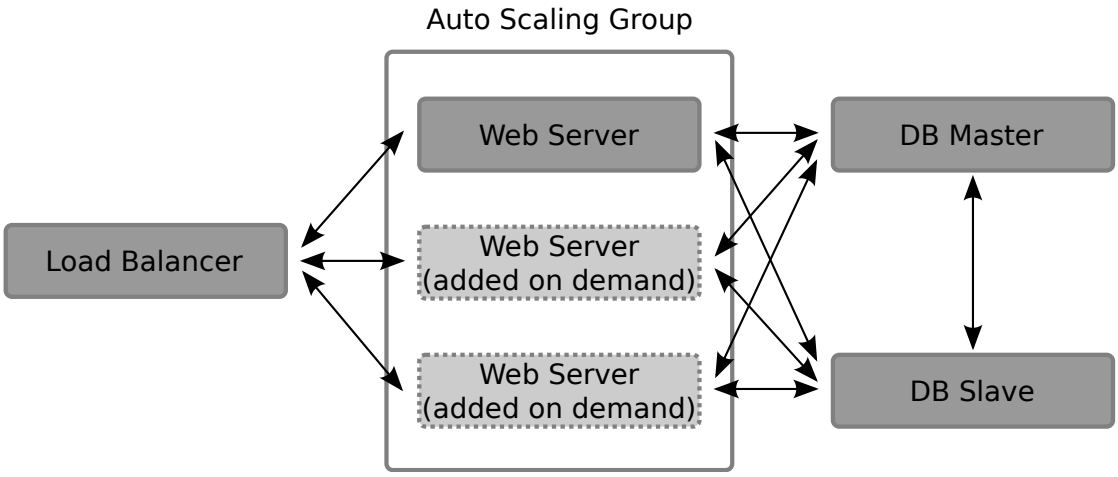


Figure 1.2: A stack contains resources (rectangles) that link together (arrows). The auto scaling group performs elasticity only on a resource level by launching additional web servers (light gray) on demand.

language format. We describe this configuration language in more detail in Section 2.2.3. To launch a new stack, the user submits the template file to the CloudFormation service endpoint. Besides the possibility to create new stacks, the service also supports to update and to delete existing stacks. This enables users to manage the entire life-cycle of a stack in a predictable and repeatable manner. By employing load balancers and auto scaling groups, CloudFormation also supports the creation of resource elastic web services.

The second step deals with the configuration of the individual resources. This step usually involves downloading and setting up the required software packages in well defined, tested and known to work combinations of versions. Although CloudFormation provides some rudimentary support for configuring resources, productive environments require a more sophisticated approach to configuration management. Two prominent examples of configuration management systems are *Chef*[1] and *Puppet*[2]. Like their orchestration counterparts, these configuration management tools allow specifying the configurations with the help of a Domain Specific Language (DSL).

1.2 Problem Statement

Even with the availability of sophisticated orchestration and configuration management services, setting up and configuring stacks is a non-trivial and error-prone task. Configuration requires both expertise and in depth knowledge of the employed resources [19]. The complexity further increases in systems that manage a large number of heterogeneous resources, possibly reaching hundreds or even thousands of individual resources. As an example, Oppenheimer et al. de-

[1] `http://www.getchef.com/`
[2] `http://puppetlabs.com/`

scribe an internet service with more than 2 000 machines in their case study [68]. Along with a larger number and heterogeneity of components in a stack the probability for misconfigurations increases as well. Yin et al. show in their empiric study that wrongly configured systems have a negative impact on availability [83]. This increased complexity also induces a rise in the Total Cost of Ownership (TCO) of such systems [19]. Therefore, there exists a high incentive in cloud computing to automate configuration and management tasks.

The responsibility of configuration management tools (such as *Puppet* or *Chef*) usually stops at the point when the system has been fully deployed [39]. This means that configuration management tools have no intrinsic support for application elasticity. Existing tools for orchestration on the other hand possess some rudimentary support for setting up elastic systems. AWS CloudFormation limits this support to setting up auto scaling groups. Such auto scaling groups are able to react in an Event Condition Action (ECA) style to changes of low level metrics such as CPU utilization or network latency. The user must manually define the thresholds for these low level metrics. Once a metric meets the threshold, the auto scaling group performs a high level scaling task, such as adding or removing instances [39]. However, elasticity in cloud computing is a multi-dimensional problem that, besides the resource dimension, also includes the cost and quality dimensions [26]. Another limitation of the auto scaling approach is that in this case managing and defining thresholds also requires expertise and system knowledge.

Ultimately this implies that if a user wants to build an elastic application on a more fine grained level than resource elasticity, the adaptation logic has to be implemented by the user on his or her own. To our best knowledge, existing cloud computing services do no sufficiently support the automated configuration of user resources on an application level yet.

We therefore envision a system that enables users to specify desired application behavior and in turn receive the resulting configurations via a well defined service. The provider of the cloud service is responsible for managing this service. However, in order to create such a service, we need to address the following questions.

- How can we extend existing cloud orchestration template languages so that a user can specify elasticity requirements to the application with them?
- How can we integrate a provider managed cloud computing adaptation service into an existing cloud computing service infrastructure?
- Is it possible to utilize the collected data from all clients in a way that allows the autonomic manager to derive reasonable specific adaptations with respect to user defined objectives?

The ultimate goal of a cloud computing adaptation service is to offer benefits for both the cloud user and the cloud provider. First of all such a system relieves the user from the complexity of setting up, developing and maintaining a custom adaptation logic on his or her own. By utilizing a centralized service that the cloud provider manages, the user is able to take advantage of the accumulated global knowledge from all other users that operate the same application. Overall we assume that such a system *a*) lowers the TCO by reducing the need for manual configuration and by providing elasticity on a more fine grained level than resource elasticity and *b*) it is able to derive reasonable configuration decisions from the collaboratively collected

data. The provider on the other hand gets a deeper insight into the states of the applications that are running at the client side. This enables the provider to better consolidate and manage its resources. In this thesis, however, we focus on the benefits for the cloud computing user.

1.3 Contribution

Next to giving an overview of the current research in the field of dynamic applications adaptation in the context of cloud computing this thesis addresses the problems described above in Section 1.2 by providing the following contributions.

- We propose a method to specify adaptation points, desired application objectives as well as the overall optimization strategy within the AWS CloudFormation template format [3]. The method we propose embeds the directives into the existing format description and therefore adheres to the existing template files specification. To show the feasibility of the proposed method, we provide a concrete implementation of a plug-in for Heat, the orchestration service from OpenStack. This plug-in introduces three new resources to the Heat templates that we use to specify the adaptation points, objectives and strategy. The plug-in implementation parses the specified templates and invokes the necessary service at the adaptation service prototype. Using this prototype we present how to describe desired elastic behavior by means of the existing template format.

- We present an architecture along with a concrete prototype that provides an adaptation service for cloud computing applications within OpenStack. This prototype collects the observation points across all users that leverage the adaptation service for a specific application. We then use the collectively gathered observation points from all users as the basis in order to derive specific adaptive decisions for a user's application depending on the current context. Independent from the concrete adaptive algorithm presented in this thesis the prototype also serves as the basis for future research on this topic.

- We propose an algorithm that derives a configuration for an application based on the collaboratively collected observation points from all users that run the same stack. The proposed algorithm filters and normalizes the observation points and sorts them into buckets. Each bucket contains all observation points that share the same adaptation point attribute values and whose external attribute values are similar to the current observation points. We then calculate the quality score for each bucket by evaluating how many observation points in the bucket violate the user objectives and with adaptation point specific quality functions. Finally, we select the bucket with the best quality.

1.4 Methodological Approach

The research by Inzinger et al. in [39] laid the foundation for this thesis by introducing the idea of a cloud application configuration service. Figure 1.3 depicts the proposed architecture of such a service. In this thesis we now build further upon this idea and verify the feasibility of the

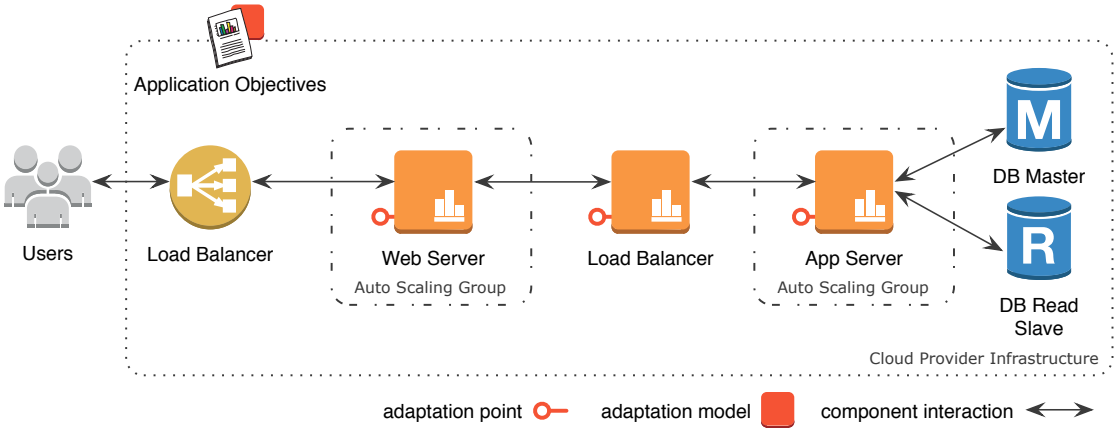


Figure 1.3: The original cloud adaptation architecture envisioned by Inzinger et al. in [39]. This architecture supports other elastic dimensions by creating well defined adaptation points that allows a third party service to modify the behavior of a resource.

proposed approach by building and experimenting with a prototype for the open source cloud computing software OpenStack.

In order to achieve this goal, we first review the scientific literature with a focus on adaptive cloud computing systems, autonomic computing as well as on self configuring systems. Based on this review and the a priori requirements we then derive the final requirements for a cloud application adaptation service prototype. These requirements serve as the basis for our proposal for an architecture for the adaptation service presented above. We also propose a method that enables cloud users to specify adaptation points for an application which enables the adaptive service to interact with the applications inside the stack. Finally, the user can specify the desired target objectives that the application shall obey inside the existing range.

Next we use the prototype's framework as the basis for implementing a configuration selection algorithm. We first generate an initial data set that executes the same benchmark workload with different adaptation point settings. Based on this body of data we iteratively develop a measure for quality that reflects the configuration option that is best suited for a specific workload. In order to demonstrate the feasibility of our approach we implement this algorithm in our prototype and perform benchmark tests, in which we compare non-managed stacks to the managed stacks.

1.5 Organization of the Thesis

We now describe the organization of the remainder of this thesis as follows.

- In Chapter 2 we describe the open source cloud computing platform OpenStack in more detail. This platform illustrates the technical concepts of a state of the art cloud computing platform. OpenStack also serves as the technical basis for the prototype implementation.

- Chapter 3 gives an overview of state of the art scientific work relates to this thesis' topic. Although overlapping, we categorize the related work into four major topics, namely *i*) autonomic computing, *ii*) cloud computing and elasticity, *iii*) Service Level Agreements (SLAs) and *iv*) misconfiguration detection.

- In Chapter 4 we discuss the requirements for a cloud computing configuration service. Based on the requirements we describe the architecture of our autonomic adaptation service. We conclude the chapter with a description of the prototype's implementation details.

- Chapter 5 describes the configuration selection algorithm that we develop based on benchmark data. We first provide a formal definition of the configuration selection problem, followed by a description of the algorithm. We conclude the chapter with a discussion of the algorithm's limitations.

- In order to prove the concept of the presented adaptation service and to verify the hypotheses, we provide an evaluation of the prototype's performance in Chapter 6. First we verify the claims with a self developed simulator, followed by an evaluation on a real world web application scenario.

- Finally, we revisit the research questions in Chapter 7 and conclude this chapter with an outlook on future research topics related to this thesis.

CHAPTER 2

Background: OpenStack

OpenStack provides a bundle of components to operate an IaaS and markets itself as an "*operating system for the cloud*" [65]. Initially founded by the NASA and Rackspace Hosting [65], today a number of industry partners back the development of OpenStack, such as IBM, AT&T, Red Hat and others [64, 82]. Users from both the private and the public sector use OpenStack to operate public, hybrid and private cloud setups [66].

In this chapter we describe the OpenStack *Icehouse* release. This release serves as the technical platform for our prototype implementation. In addition, OpenStack represents the architecture and components of a typical IaaS platform. In Section 2.1 we first give an overview over OpenStack's architecture. We then present the individual components of OpenStack in Section 2.2. Because the orchestration service *Heat* forms an integral building block of our prototype, we conclude this chapter with a working example of a Heat template in Section 2.3 that illustrates the concepts of this service.

2.1 Architecture

The core of OpenStack consists of individual components that communicate with each other via well defined Application Programming Interfaces (APIs) [61, p. 3]. All the components that we will cover in Section 2.2 are written in Python and are licensed under the terms of the Apache 2.0 license.

Figure 2.1 shows the individual components of OpenStack and their relationship to each other. The image illustrates that the core of an IaaS such as OpenStack revolves around the management and execution of Virtual Machines (VMs). OpenStack aims to provide a cloud software that is agnostic to the underlying hardware and hypervisor [40, ch. 3]. Consequently, OpenStack does not include the virtualization components (i.e. the *hypervisor*) on its own. Instead, OpenStack provides a set of tools that allows the execution management of VMs of different hypervisor providers, such as *Xen*, *VMware vSphere* or *KVM* [61, p. 57].

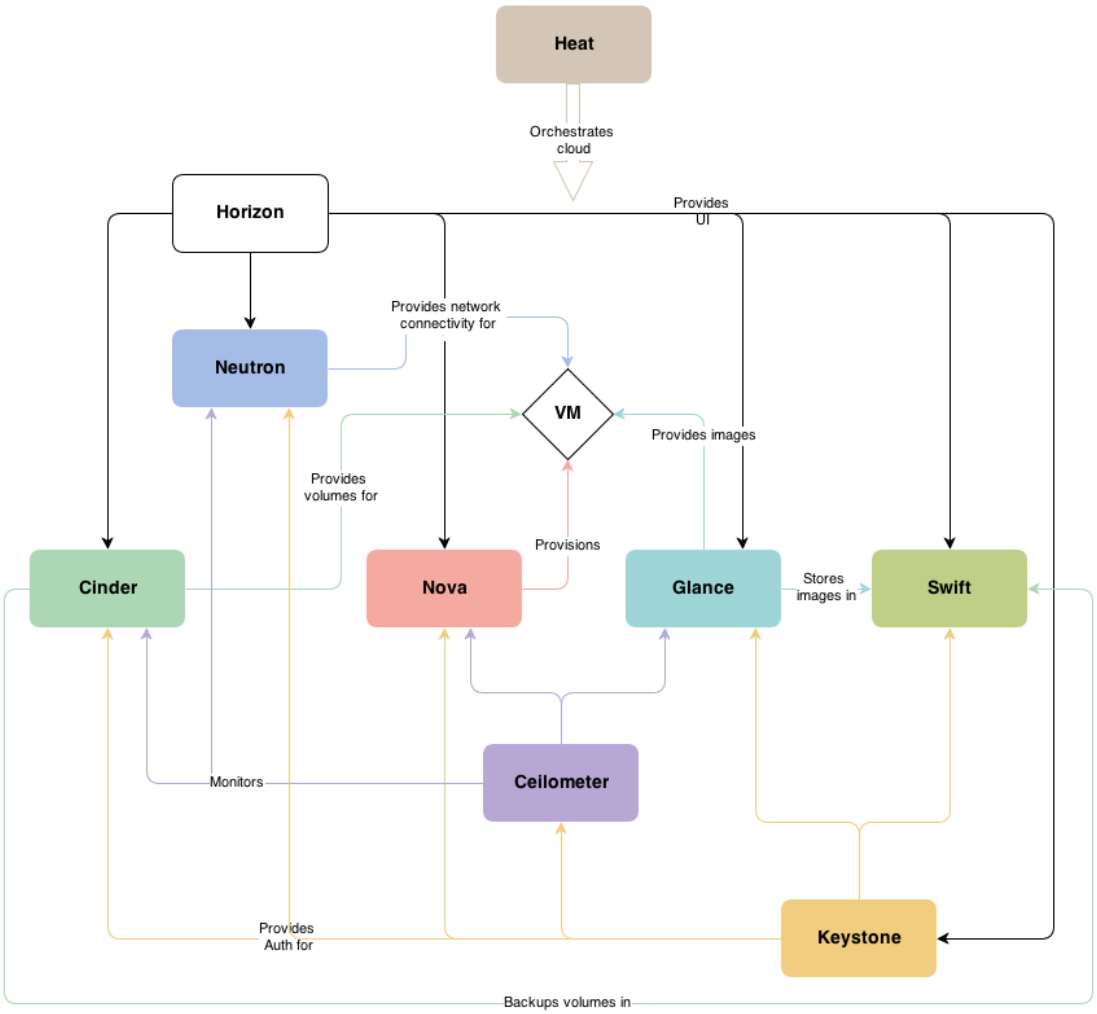


Figure 2.1: Overview of the OpenStack services and their relationship. Image source: [61, p. 2], Author: OpenStack Foundation, License: CC 2.0 BY SA

2.2 Services

In this section we now briefly present the nine OpenStack services from Figure 2.1. If not stated otherwise, we refer to the functionality and properties of OpenStack's *Icehouse* release.

2.2.1 Compute (Nova)

Nova controls the execution and the management of VMs in OpenStack. It also provides the necessary interfaces for the computation service. An *instance* is a VM that executes inside OpenStack [62, p. 11]. A user launches new instances from images in different classes of computational capacity, the so called *flavors*. The main characteristics of a flavor are its RAM, Disk and VCPU size. Table 2.1 provides an overview of the default flavors in OpenStack. Besides the default flavors the cloud provider can also specify custom flavors. Other parameters that a user must specify in order to launch an instance are the availability zone, the network properties and the key pair that grants a user access to the instance.

Table 2.1: Default flavors in OpenStack

Name	Memory (MB)	Disk (GB)	VCPUs
m1.nano	64	0	1
m1.micro	128	0	1
m1.tiny	512	1	1
m1.small	2,048	20	1
m1.medium	4,096	40	2
m1.large	8,192	80	4
m1.xlarge	16,384	160	8

Nova also provides the metadata service. This HTTP service listens at the address `http://169.254.169.254` and responds to queries from instances with the instance specific metadata. The service provides two response formats, one that is compatible to the metadata service in AWS and a proprietary OpenStack metadata format [61, p. 73]. The metadata contains information about the instance such as the id or the host name. When the user launches an instance, he or she may also pass arbitrary user data to the instance. To retrieve this user data, applications on the instance also query the metadata service [61, p. 74].

2.2.2 Image (Glance)

The *OpenStack Virtual Machine Image Guide* describes virtual machine images as "*a single file which contains a virtual disk that has a bootable operating system on it.*" [63, p. 1]. In the OpenStack ecosystem, Glance is the service that manages the virtual machine images. It provides interfaces to launch instances from images, create images from instances as well as to upload and manage images from a range of different image disk and container formats [61, p. 6f.].

2.2.3 Orchestration (Heat)

We already introduced the term *service orchestration* in Chapter 1. Heat is the concrete implementation of an orchestration service in OpenStack that creates resource compositions out of template files. Heat consists of the following four services [67].

1. The `heat-engine` is the core application that runs under the supervision of the cloud provider. It creates and manages the resources that the user specifies in a template.

2. The `heat-api` provides a well defined REST API for Heat and communicates with the engine using Remote Procedure Calls (RPC).

3. The `heat-api-cfn` service provides an API that is compatible to AWS CloudFormation, which is a comparable orchestration of AWS. Like the Heat API this service also communicates with the Heat engine via RPC [61, p. 17].

4. The command line application `heat` enables a user to submit, modify and delete templates in Heat and to manage application stacks. This application parses the user input such as the templates and then submits them to the REST API endpoint of Heat.

Templates

Heat is a *one file template* orchestration system [61, p. 16]. This means that a user specifies all resources and properties that are necessary to launch a stack within a single template file. Heat supports the specification of templates in two different file formats, namely JSON and YAML Ain't Markup Language (YAML). The JSON format aims to achieve compatibility with the AWS CloudFormation templates format, although we have found that on our test system Heat does not support all constructs. We can express the functionality of our prototype that we will describe in Chapter 4 in any of the two file formats. Because the AWS template format version 2010-09-09 provides a sufficient subset of functionalities for our prototype, we will use this template version throughout the remainder of this thesis, if not specified otherwise.

Each valid Heat template file consists of an unordered set of key/value pairs that specify required stack resources and their associated properties and relationships. The keys are as follows.

- The **Template format** key specifies the used format and version of the template. If a user specifies a stack in the JSON format, this key is either `AWSTemplateFormatVersion` or `HeatTemplateFormatVersion`. The value contains the version of the specification. Heat supports the versions 2010-09-09 for the AWS template format and 2012-12-12 for the Heat template format. In the YAML format the template format is `heat_template_version` and the only supported template version is 2013-05-23. In Heat templates the template format key is the only mandatory key.

- **Description** is for organizational purposes. A user can describe the content or purpose of a template as the value for this key. The value is a single string that can contain any arbitrary text. The parser does not process this value.

- **Parameters** contain user data that a user can modify each time he or she creates a new stack, such as passwords and service endpoints. For each parameter the user must specify the specific parameter type. The type can either be a string, a number or a list of comma separated values. It is also possible to specify constraints that determine the format and the expected value of the parameters. Possible constraints are the minimal and maximal length of a string, as well as the minimal and maximal value of a numeric value. Another possibility to constrain the input parameters is to specify a regular expression that Heat validates when the user submits the template. The "AllowedValues" key allows a user to specify a list of allowed values. To make a parameter optional, a user can provide a default value for the parameter.

- **Mappings** contain a two level map (i.e. a key-value pair that contains another map of key-value pairs as value). In conjunction with the function `Fn::FindInMap` these mappings translate input values to output values. For instance, a user can specify a map to launch instances using different images depending on the availability zone. We show an example of a mapping in Section 2.3.

- **Conditions** allow a user to create resources or to output values only if certain constraints evaluate to true. A user defines these constraints with intrinsic functions such as `Fn::And`, `Fn::Not` and `Fn::Or`. In the version that we used for testing, this key was not supported by the OpenStack Heat client. Instead, the client rejected templates with a Conditions section.

- **Resources** This section contains the resources that belong to the stack. A resource must be of a certain type that describes the type of service associated with this resource. Resources are arbitrary entities that refer to the different services and concepts that OpenStack offers. Examples or resources are Nova instances, network interfaces or Swift containers. In total, OpenStack offers 30 resource types[1]. The specification of AWS CloudFormation states that this is the only mandatory key [3, p. 8]. In contrast to that the Heat client accepts templates in the AWSTemplateFormatVersion 2010-09-09 without resources.

  Resources can define parameters that contain the necessary information a resource needs to operate properly. Resource properties may reference to a user defined parameter using the `Ref` function. Properties have a fixed schema that Heat validates when it creates a stack. The concrete implementation in the source code of the resource determines this fixed schema.

- **Outputs** In this section the user specifies key/value pairs that contain information about the stack. Each entry in the Outputs section consists of a key whose value is a map that contains the required key `Value` and an optional key `Description`. A user then can quickly retrieve certain information about a stack, for instance the host name by specifying the key name of the output.

[1] We determined the number of 30 resources by counting the number of class definitions in the Heat resource source code directory that inherit from `resource.Resource`.

Functions

Heat provides a set of functions to support dynamic templates. The user must specify functions within quotes which the template parser evaluates at runtime. The exact number of functions that Heat supports depends on the used template type and version. For instance, the Heat template version 2012-12-12 provides the `Fn::Split` function that the AWS CloudFormation template version 2010-09-09 does not contain. The latter template version specifies the following functions.

- **Fn::Base64** encodes an input string with Base64. With this function, a user can encode arbitrary binary data with ASCII characters [41].
- **Condition Functions** are `Fn::If`, `Fn::Equals`, `Fn::Not`, `Fn::Or`, `Fn::And` that the conditions section of a template may contain. The Heat version that we used for evaluation does not support condition functions.
- **Fn::FindInMap** selects a specific value from the template's `Mapping` section. To retrieve the value, a user must specify the name of the mapping and the keys for both map levels. Listing 2.1 gives an usage example for this function.
- **Fn::GetAtt** retrieves an attribute value from a resource. Each resource optionally has a set of attributes. The Nova instance resource, as an example, has an attribute that allows a user to retrieve the public DNS name of that instance. To access these resource attributes within a template, a user can employ the built in function `Fn::GetAtt`.
- **Fn::GetAZs** returns a list of availability zones within a region. This command is useful for resources that require a list of availability zones upon launch. According to the Heat developer documentation in [20], availability zones are not fully implemented yet.
- **Fn::Join** takes a delimiter and a list of strings as parameters. `Fn::Join` then joins the list elements with the specified delimiter.
- **Fn::Select** takes an index i and a list of objects as parameter. The function picks the element with index i (starting with 0) from the specified list of objects.
- **Ref** denotes a reference to either a parameter or a resource. For instance the property `myproperty` in Listing 2.1 uses a reference to the parameter `param1`.

2.2.4 Telemetry (Ceilometer)

Ceilometer is the telemetry service in the OpenStack framework. This service has its roots in a metering component for a billing service in OpenStack, but has since then grown to a multi purpose telemetry solution [61, p.278]. The project aims to provide a central facility that handles all concerns that have to do with the collection and processing of system metrics [58]. In Ceilometer it is possible to monitor metric values with alarms. If a user specifies an alarm, the telemetry service checks whether the metric value is above, below or equal to a specified threshold value [17]. Each alarm is in either one of the states `Ok`, `Alarm` or `Insufficient data`.

For each state the user can specify an optional "hook" Uniform Resource Locator (URL) that Ceilometer calls when the metric enters a new state.

2.2.5 Networking (Neutron)

Neutron is the networking service of OpenStack. The aim is to provide a central and flexible networking API for all the other services in OpenStack. This abstraction allows OpenStack to provide concepts such as "floating IPs", that allow to re-assign an existing IP address to a different instance [61, p. 68]. Apart from the low level networking services, Neutron also contains support for high level network services, such as the Load-Balancer-as-a-Service (LBaaS) [60] or Virtual Private Networks (VPNs) [61, p. 182].

2.2.6 Identity (Keystone)

Keystone is the central authentication and authorization service in OpenStack. The three central concepts in Keystone are users, tenants and roles [61, p. 19f.]. Users are physical persons that interact with the system and who identify themselves with a user name and a password. Tenants describe projects and organizations that a user must specify with every request to OpenStack [61, p. 19]. Finally, as the name implies, Roles assign a certain role to a user for a tenant [61, p. 20].

2.2.7 Object Storage (Swift)

Swift is the cloud storage service that provides a universal, redundant and scalable data storage [61, p. 127]. Because Swift makes use of advanced replication techniques, the service can provide reliable storage on commodity hardware [61, p. 127]. The structural elements of Swift are containers and objects. Each container holds an arbitrary number of objects. An object consists of a file and the metadata that is associated with it [62, p. 19]. Each object has a URL and is accessible via the REST API. OpenStack services also utilize Swift internally. Glance, for instance, stores its images in Swift, as Figure 2.1 shows.

2.2.8 Block Storage (Cinder)

Cinder provides an abstraction layer for block storage devices. A user can attach these storage devices to Nova instances and mount them as volumes. A user can also detach a volume from an instance and attach it to another instance. However, OpenStack allows attaching each volume only to one instance at most [62, p. 23]. Block storage devices therefore are the cloud equivalent of a "*USB flash drive*" [40, ch. 7].

2.2.9 Dashboard (Horizon)

Besides the command line client, OpenStack also provides a web based Graphical User Interface (GUI) that acts as a client to the provided API [62, p. 7]. Horizon offers functionalities for the cloud users as well as the administrators to manage the OpenStack services. Figure 2.2 shows a screenshot of Horizon.

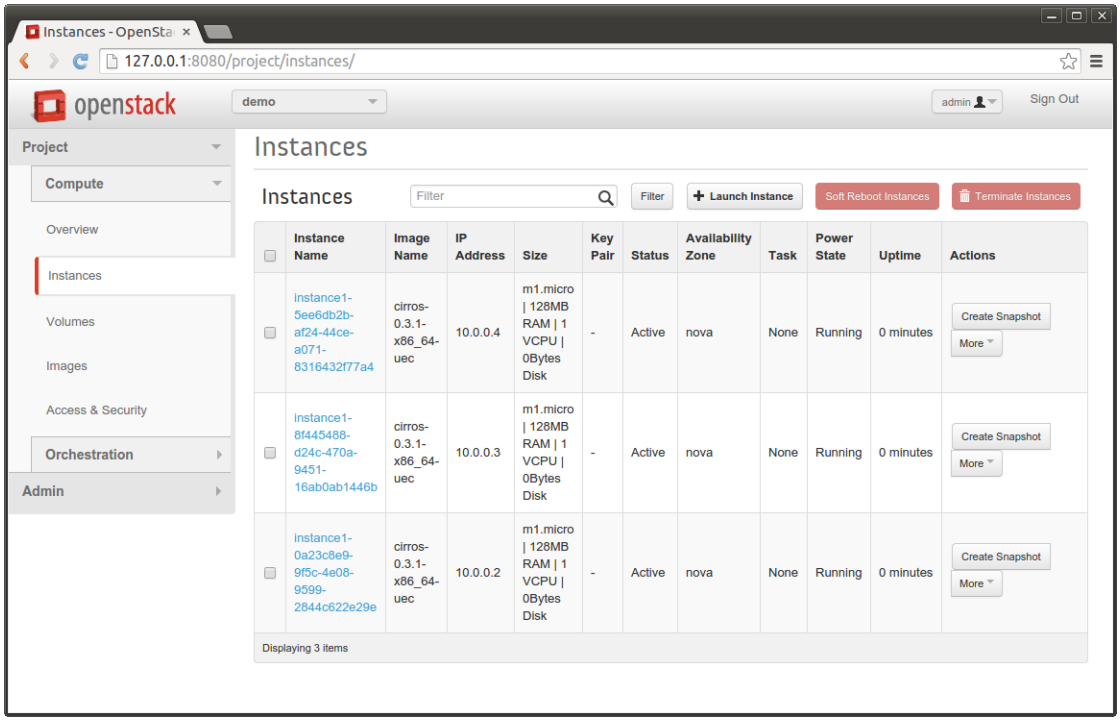


Figure 2.2: A screenshot of the GUI in Horizon with three running Nova instances.

2.3 A Sample Heat Template

We conclude this chapter with a working example of a Heat template. The template in Listing 2.1 contains all the six keys that the Heat client supports for the AWS template format version 2010-09-09. The presented template is not particularly useful in a real world scenario, but it illustrates the concepts that we described in the previous section.

In the provided example, the first key specifies the template format and version. Next, the key `Description` contains an arbitrary value that describes the purpose of the template. In the `Parameters` section we declare a parameter named `param1` that is of type string and that rejects all parameter values that are not "value1". In the `Mappings` section we define the `InputTransformationMap` that maps "value1" to the second level map with the key "transformedValue" and the value "mapped_value1'". The next key `Resources` contains our template's only resource. The specified resource type `OS::Heat::RandomString` is an internal Heat resource that generates a random string and makes this random string available to the template via a resource attribute. Finally, the `Outputs` section specifies an output named "MappedValue" that uses the function `Fn::FindInMap` to transform the value of the template parameter "param1". In our example, the output value will always be "mapped_value1", because the only allowed value for "param1" is "value1".

Listing 2.1: An example Heat template in the AWS CloudFormation format.

```
{
  "AWSTemplateFormatVersion" : "2010-09-09",

  "Description" : "An example template",

  "Parameters" : {
    "param1" : { "Type" : "String", "AllowedValues" : [ "value1" ] }
  },

  "Mappings" : {
    "InputTransformationMap" : {
        "value1" : { "transformedValue" : "mapped_value1" }
    }
  },

  "Resources" : {
    "RandomStringResource" : { "Type" : "OS::Heat::RandomString" }
  },

  "Outputs" : {
    "MappedValue" : { "Value" : { "Fn::FindInMap" : [
      "InputTransformationMap",
      { "Ref" : "param1" },
      "transformedValue"
    ] } }
  }
}
```

CHAPTER 3

State of the Art

In this chapter we provide an overview of related scientific work to this thesis. Although the borders between the categories are overlapping and fuzzy, we classify the reviewed state of the art research into the following four categories: First we present the principles of autonomic computing in Section 3.1. We put a special focus on the self-configuration and self-organization aspects, from which we present related work. In Section 3.2 we cover the state of the art in cloud computing and elastic systems. Next, in Section 3.3, we discuss SLAs and related work from this area. We conclude this chapter with a review of state of the art scientific work from the field of misconfiguration detection in Section 3.4.

3.1 Autonomic Computing

The term "autonomic computing" was first coined in a manifesto released by IBM in 2001 [43]. The idea of autonomic computing is inspired by biological autonomic systems that manage complex systems with the help of feedback loops [37]. Autonomic systems emerge from the increasing complexity that the maintenance of large distributed systems incurs [43]. There are four *self-** aspects that characterize autonomic systems, namely self-configuration, self-optimization, self-healing and self-protection [30, 43].

IBM also proposed an architectural model that describes the individual components of an autonomic system and that we show in Figure 3.1. According to this model, an autonomic system consists of an autonomic manager and a managed element.

The managed element is an element at an arbitrary level of detail, as long it is possible to manage it [22]. Examples of managed elements are software components, databases or even hardware [37, 43].

The autonomic manager forms a feedback loop with the managed element via effectors and sensors. This loop is the so called Monitor, Analyse, Plan, Execute, Knowledge (MAPE-K) loop, named after the four consecutive phases in the autonomic manager (Figure 3.1). For any of the four phases the global knowledge is the basis for the actions. As Figure 3.1 shows,

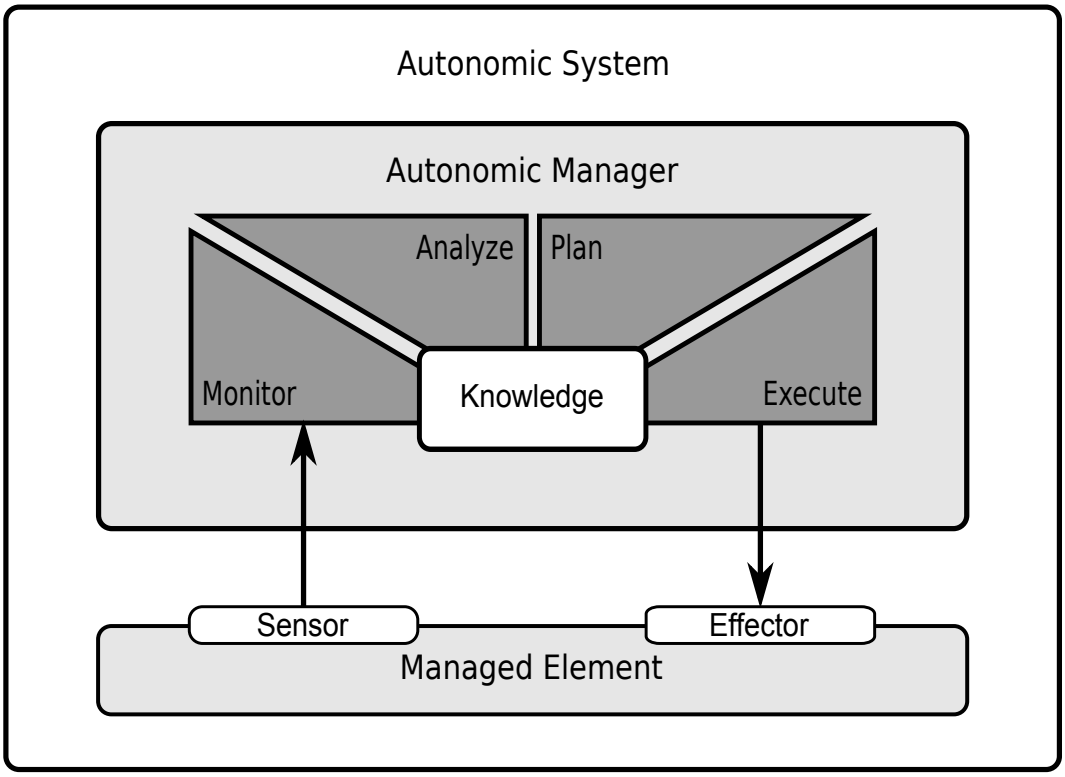


Figure 3.1: The MAPE-K architecture as proposed by IBM [43]

this global knowledge is a "cross cutting concern" that all the four phases utilize. The *IBM architectural blueprint for autonomic computing* classifies the knowledge into information about the system topology, knowledge about policies and problem determination knowledge such as the monitored data [22]. We now cover the four phases of the autonomic manager in more detail.

1. **Monitor**. In this phase the autonomic manager retrieves data from the sensors of the managed elements. Huebscher and McCann distinguish between passive monitoring with system utilities such as `vmstat` (covered in [80]) and active monitoring where the user modifies the software to retrieve relevant metrics [37].

2. **Analyze**. Based on the monitored data, this phase analyzes whether the autonomic manager performs an action or not [22]. The basis for this decision is the information whether the managed system can meet the specified high-level goals now and in the future [22]. To predict the future behavior, autonomic systems can employ techniques such as time series forecasting [22].

3. **Plan**. In this phase, the autonomic manager creates a plan that contain the sequence of actions to perform on the managed element [22]. Huebscher and McCann emphasize that the planning should also take the system history into account, as approaches that take only the current state into account are limited in their usefulness [37].

4. **Execute**. In this stage, the autonomic manager translates the plan into real actions by calling the effectors on the managed element which then execute the necessary actions on the managed element [22].

As outlined above, one of the four *self-** aspects of autonomic computing is the ability for self-configuration based on high-level goals [30, 43]. Because this self-configuration aspect, along with the self-optimization aspect, are integral parts of this thesis, we study the related work from this research area in more detail. In our state-of-the-art review we focus on systems that use autonomic concepts and that are able to make a configuration decision within a configuration space in response to some system state. Approaches to self-configuring and self-optimization systems cover a wide range of techniques such as reinforcement learning, control theory and others (e.g. game theory and neural networks), all of which we will now describe in more detail.

3.1.1 Control Theory

Some of the work on self-configuring systems proposes control theory as a promising mathematical foundation for autonomic systems, for instance that from Diao et al. in [24], Karamanolis et al. in [42] or Padala et al. in [69]. Scientific fields such as mechanical and electrical engineering have been using control theory since decades [24]. Diao et al. present a work in which they use the ABLE toolkit (described in [8]) to build a framework for the automatic tuning and setting of web server parameters. They demonstrate their concept with an AutoTune agent that manages Apache HTTP Servers by adjusting the values of the `MaxClients` and `KeepAlive` configuration options [19]. Control theory has also received criticism, for instance by Huebscher and McCann, who argue that control theory is not a "*panacea*" for all aspects of autonomic systems [37]. Salehie and Tahvildari point out that in the context of software systems, control theory involves more complexity than necessary [75].

3.1.2 Reinforcement Learning

Also reinforcement learning has received considerable attention as the basis for autonomic system configuration on the application level, for instance in [6, 12, 13, 86].

Essentially, reinforcement learning systems consist of *i*) a policy that describes the possible states, *ii*) a reward function that specifies the "*desirability*" of a state, *iii*) a value function that models the probable "long term reward" of a state and optionally *iv*) a model of the environment [77]. That the model is not strictly necessary is a main advantage of reinforcement learning approaches [37, 73]. Another benefit of reinforcement learning is that it allows a user to take the delay between a configuration change and its effect into account [73].

Bu et al. [12] use a reinforcement learning approach to configure a multi-tier web application that is hosted in a VM environment. In their work, the configuration space consists of four parameters for the Apache HTTP Server and four parameters for the application server. Bahati et al. also present a reinforcement learning approach to configure Apache HTTP Server settings [6].

Reinforcement learning has received criticism for not scaling well [37] and that it can only generate a decision whether to adjust a setting, but cannot assign a concrete value for it [33].

Guo et al. also suggest that reinforcement learning is not able to deal well with bursty workloads [33].

3.1.3 Other Methods

Beside approaches to self-configuration that utilize control theory or reinforcement learning, recent research has brought forth self-configuring systems based on other techniques. Although a complete survey is out of the scope of this thesis, we present two examples of this category.

García-Galán et al. propose an approach to configure multi-tenant desktop services according to the user specified preferences. They represent the configuration space with an Extended Feature Model (EFM) where a user expresses functional and non-functional preferences with a semantic ontology [32]. To find a configuration, they formulate the problem as a modified cooperative game which they solve with the Nash bargaining solution [32].

Guo et al. tune the Apache HTTP Server configuration settings `MaxClients`, `KeepAliveTimeout` as well as `MaxSpareServers` and `MinSpareServers` with a method that combines neural networks and fuzzy control systems [33]. Compared to a rule based approach, Guo et al. found that their method is able to deal with bursty workloads better [33].

3.2 Elasticity and Cloud Computing

A number of scientific fields make use of the term elasticity, for instance physics and economics [26, 35]. Applied to the field of cloud computing, Herbst, Kounev and Reussner propose a formal definition of elasticity as "*the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible.*" [35]. Although this definition covers the resource facet of elasticity, Dustdar et al. point out in their work that there are two more dimensions that elastic computing systems must cover, namely the cost and quality dimension [26]. In order to describe elasticity properties of cloud computing systems, Dustdar et al. transfer the concept of elasticity from the field of economics to computer science [26].

As the example from the field of economics that Dustdar et al. present, the *price elasticity of demand* puts the change of a price in relation to the change in demand that the price change induces. Applied to formula 3.1 this means that the two parameters of the function $e(Y, X)$ are the demand function Y and the price X. The elasticity of the price X then is defined as $e(Y, X) = Y'(X) \cdot X/Y(X)$.

Formula 3.1: Elasticity of Y with respect to X

$$e(Y, X) = \frac{\mathrm{d}Y}{\mathrm{d}X} \cdot \frac{X}{Y}$$

From the theoretical foundation of elastic processes by Dustdar et al. in [26] we pick up the idea of viewing elasticity as a multi-dimensional problem. We do this by implementing a

quality function that can reflect any combination of elasticity dimensions, such as cost or quality. Because the target is always to maximize the quality, regardless of the quality function's concrete definition, our system is able to perform adaptations in this multi-dimensional space.

3.2.1 Specification of Elasticity

The specification of constraints and preferences in order to express the desired behavior of elastic services is a major research challenge [26].

Researchers have addressed this problem for instance with rule based systems. Moran et al. researched rule engines for their suitability to define the runtime behavior of cloud applications [56]. From the evaluated rule languages, the authors identified the rule interchange format (RIF) as the best suited format because of its interoperability, the object oriented data model and because its W3C recommendation [56].

Rodero-Merino et al. propose to define elastic behavior with *Service Description Files* that are based on the Open Virtual Format (OVF) [18, 74]. However, as Copil et al. point out in [21], the work mentioned above only focuses on the resource aspect of elasticity.

Dustdar et al. propose to transfer the principle of parallel programming languages to separate "high level" behavior from application logic with directives to the specification of elasticity behavior [25]. For this they introduce *SYBL*, the "*Simple Yet Beautiful Language*", which is fully specified in [21]. SYBL allows a user to formulate elastic behavior with directives. A user can integrate these directives into existing programming languages for instance with annotations or use them in configuration languages such as TOSCA. Sofokleous et al., for instance, present an approach that utilizes SYBL constraints and strategies to specify the elasticity behavior of cloud applications and translate the specification into TOSCA profiles [76].

SYBL itself features three classes of directives, namely Monitoring, Constraint and Strategy.

- *Monitoring* directives let a user assign system metrics such as the application response time to variables [21]. SYBL groups the directives into three classes according to the elasticity dimensions in which they belong.

- *Constraints* define the conditions that the system must meet for monitoring variable values. An example for a constraint is the specification that a variable that holds the application response time must stay below a certain threshold. Constraints allow a user also to specify trade-offs between cost and quality [25].

- *Strategies* let the user specify the desired behavior of the application if some conditions holds. For example, a user can let the application scale in more resources if a metric violates a constraint [21].

Although we do not use directives in our prototype, there are similarities to SYBL. Like SYBL, we also use the monitored data to enable elasticity on the three different dimensions. The constraints in SYBL are comparable to the objectives in our elasticity specification that we will present in Chapter 4. However, in our prototype there is no equivalent to SYBL's strategies, because the algorithm that we present in Chapter 5 selects a configuration automatically based on previously observed data.

3.2.2 Elastic Management of Cloud Services

Besides the specification of elasticity, research effort has also been put into elastically managed cloud services.

Martin et al. propose an event driven management model to manage elastic cloud services [51]. Martin et al. point out the importance of effective service management in elastic applications and describe the challenges that the management of elastic services involves [51]. These challenges, according to Martin et al., include the inherently dynamic nature of elastic systems with constantly changing environments and configurations.

Truong et al. present *CoMoT*, a PaaS that natively supports to control, monitor and test multi-dimensional elastic services [78]. In their work they use the concept of "*Elastic Service Units*". Characteristics of elastic service units are their function, service model, dependencies and elastic capabilities [78]. These elastic service units can be either offered by the provider or specified by the user. To describe the properties of the elastic service units, the authors use SYBL [21]. Finally, the system outputs a TOSCA description file [78], which is a language to describe the components of a service and their relationship to each other [9].

The monitoring component of CoMoT utilizes *MELA*, a framework designed for monitoring and analyzing the elasticity of cloud services [55]. In this framework, the three dimensions of elasticity are captured by assigning metrics into either one of the categories. For instance, data transfer cost belongs to the cost category, response time to quality and network bandwidth to the resource dimension [55].

In [39], Inzinger et al. argue that models already have found wide spread adoption in cloud computing in the form of DSLs. For instance, AWS CloudFormation templates such as described in Chapter 2 contain a model of the cloud application's structural architecture [39]. Inzinger et al. propose to use and extend these models so that a user can also specify the desired elastic runtime behavior with them. For this, a user has to specify the adaptation points, objectives and metrics within the template [39]. Based on these extended templates, the provider then can offer an autonomic system that is able to control the runtime aspects of the managed cloud application. In this thesis, we build upon the proposed idea from Inzinger et al. by providing a prototype implementation and an evaluation of their concept. We will present this prototype and the evaluation in Chapters 4 and 6, respectively.

3.3 Service Level Agreements (SLA)

SLAs specify the conditions under which a service provider offers a particular computation service to a user. An SLA consists of Service Level Objectives (SLOs) that introduce a formal agreement about the Quality of Service (QoS) between a provider and a consumer [52]. There are initiatives to standardize the specification of SLAs, for instance with the XML based *Web Services Agreement Specification* (WS-Agreement) [4] or the Web service level agreement (WSLA) language [49].

Leitner et al. provide a formal model that describes SLAs as functions that map the duration of a specific request to the cost of violation that the issued request incurs [47]. According to Leitner et al., the following two properties are characteristic for an SLA function [47]:

1. The SLA function monotonically increases, i.e. the cost for a violation increases with the degree of violation.

2. There are two function points t_1 and t_2 that describe the point until no violation occurs (t_1) and until a request is "*given up*" (t_2) and the cost of violation increases no further.

3.3.1 SLA Enforcement

A plethora of research has been conducted regarding the enforcement of agreed SLAs. These works feature a wide range of aspects such as monitoring SLA parameters or the use of autonomic services to avoid SLA violations.

The "*Foundations of Self-governing ICT infrastructure*"(FoSII) project implements an autonomic component to manage and enforce SLAs [29]. This framework contains a monitoring component that maps "low level" system metrics to "high level" SLA parameters with the help of rules [27].

To enforce the SLAs in within the FoSII project, Maurer et al. evaluates different methods. First, they present a method that uses Case Based Reasoning (CBR) as the knowledge component for the MAPE-K feedback loop [52]. They demonstrate the feasibility of their approach by utilizing a simulator. Like the approach in this thesis, Maurer et al. use knowledge that is already stored in the database in order to retrieve new decisions. To determine the similarity between two individual cases, they calculate the euclidean distance between the min/max normalized attributes of the two cases. In the algorithm that we will present in Chapter 5, we also use normalized euclidean distances to determine the similarity between two observation points. Maurer et al. use a utility function that calculates the sum of a violation function and a weighted utilization function. Again, our method of determining the observation point qualities with a quality function is similar to the method from Maurer et al. Despite these similarities, our approach differs from that to Maurer et al. in the following ways: While Maurer et al. focus on the provider side, we focus on the cloud user's side. Consequently, our work is not concerned with efficient resource allocation but rather with tuning a cloud application. Another major difference is that we output a concrete set of configuration values, while the method from Maurer et al. returns discrete actions, such as "increase value by 10%" [52].

Next, Maurer et al. evaluate the FoSII project with a rule based approach to enact the SLAs [53]. For this they introduce the concept of "*Threat Thresholds*" (TT). Based on two TTs TT_{low} and TT_{high}, they classify each parameter value into one of the three regions $\{-1, 0, +1\}$, where -1 denotes a parameter value below the lower bound TT_{low}, $+1$ a parameter value above the upper bound TT_{high}, and 0 if the parameter value is in the desired range [53]. Compared to the CBR approach in [52], Maurer et al. found that the rule based method performs better in terms of avoiding both over- and underutilization of resources [53]. To deal with the question how to set the threat thresholds autonomously, Maurer et al. proposed two methods, one based on the cost function and another that is based on the observed range of the parameter values, the so called "*workload volatility*" [54].

In [47], Leitner et al. present an approach to schedule requests to resources that takes the trade off between the time into account that a resource needs to perform a scheduling decision. Assuming that the duration of all requests is known beforehand (or can be estimated),

Leitner et al. present a model that take the following two characteristics of cloud computing environments into account:

1. The billing model found in real world cloud computing environments such as AWS has the form of a step function that increases by a fixed amount per Billing Time Unit (BTU) [47].

2. When a new resource launches, it is not available immediately, but it takes a certain time for provisioning. Leitner et al. assume in their model that this time is similar for each resource and therefore model it as a constant.

Based on this model, Leitner et al. present a procedure that decides whether it is more economical to place a request onto an existing resource or to launch a new resource. This procedure is a greedy approach that evaluates the costs for all possible placements decisions [47]. Although the method that we present in this thesis does not consider the scaling duration, we plan to incorporate a model similar to the one proposed by Leitner et al. in a future version of our prototype.

Although most of the research on SLA focuses on the provider view and consequently covers topics such as minimizing the cost of SLA violations or optimizing physical resource placement, the research work on SLA enforcement still is closely related to the topic of this thesis. In our thesis, we also are concerned with the enforcement of user specified targets while trying to use the resources as efficiently as possible. In [46], Leitner et al. present a formal description of the optimization problem that selects a set of adaptations to perform in order to minimize both the cost of adaptation and the cost for SLA violation. Although Leitner et al. use a business process as motivating scenario, the similarity between the problem formulation in [46] and the problem formulation of our algorithm in Chapter 5 demonstrates the close relation between SLA enactment and the work in this thesis.

3.4 Detection of Misconfigurations

Erroneous system configurations are a substantial cause for system failures and unavailability, as the study from Yin et al. shows [83]. Oppenheimer et al. present a case study of three large-scale internet services in [68]. In all three services that the study presents, the most common mistake by operators that led to a failure were configuration errors [68]. Therefore, a wealth of research therefore has been directed both at systems that are able to prevent and detect misconfigurations which Zhang et al. categorize into the following two classes.

1. "*Black box*" methods use known configurations in order to derive rules that are known to work with the help of e.g. statistical methods or data mining.

2. "*White box*" methods inspect the application in order to detect misconfigurations.

Because in our work we also want to derive configurations from a large set of known configurations with the help of data mining methods, particularly the black box approaches of misconfiguration detection are of interest for our state-of-the art review. We now describe some of the methods in more detail.

Zheng et al. present an architecture that automatically generates server configuration files with the aim to prevent misconfigurations [86]. For this the authors propose a scripting language that generates configuration files for servers. Another contribution of the paper is a parameter dependency graph that they create with an algorithm based on a Classification And Regression Tree (CART) and dependency tests [86]. With the help of this parameter model, Zheng et al. prune the search space for configuration. To find the final configuration, the authors use a simplex algorithm [86].

In order to detect wrongly configured system registry settings on desktop Personal Computers (PCs) running the Windows operating system, Wang et al. present a statistical model based on the Bayes rule [79]. Kiciman and Wang propose a different method to detect wrongly configured entries in the system registry. In their approach, they cluster similar keys with a distance measure and infer correctness constraints. Constraints can be of any of the four classes size constraints, value constraints, reference constraints and equality constraints [44].

Zhang et al. examine the correlation of configuration options in order to detect misconfigurations of Apache HTTP Servers, MySQL databases and PHP settings. They present a framework that learns "*best practice*" configuration rules based on correlation to a large body of known configurations from existing systems [84]. Zhang et al. describe that their work is based on the observation that configuration entries are semantically tied to the concrete environment and that usually there is a correlation between configuration entries [84]. Although the work from Zhang et al. targets at "classic" system configuration, it is interesting for us as it uses data mining methods.

CHAPTER 4

Adaptation Service Architecture

In this chapter we describe the architecture of our cloud application adaptation prototype. For the sake of readability we arrange the sections in this chapter in a consecutive order following the "waterfall model". Despite this ordering, we developed the prototype iteratively in accordance to state-of-the-art methodologies in software development. In Section 4.1 we summarize the requirements of our prototype. Next we discuss and justify the system's design decisions in Section 4.2. In Section 4.3 we then describe the implementation details of the prototype. Finally, we illustrate the presented concepts with a use-case example in Section 4.4 and provide an outlook on future directions for further development in Section 4.5.

4.1 Requirement Analysis

In the first step of building the prototype we analyze the requirements. The goals of the requirements analysis phase are *a*) to identify the relevant actors of the system, *b*) to define the necessary terms precisely and *c*) to specify the functionality that the prototype must provide. Table 4.1 summarizes the outcome of the requirements specification phase.

To specify the requirements we use the wording that RFC 2119 proposes [11]. Following this suggestion, we consecutively use the term *shall* to describe requirements that are absolutely necessary. With the term *should* we indicate a recommended, but not strictly necessary requirement. Finally, we use the term *may* if a requirement is "nice to have", but not essential for the prototype to serve its purpose [11].

4.1.1 Actors

In Chapter 1 we already identified the **cloud provider** and the **cloud user** as the two key actors of our system. The cloud provider operates the facilities so that the cloud user is able to execute **cloud applications**. This distinction does not necessarily mean that the cloud provider and the cloud user must be different persons or organizations. In a private cloud deployment, for instance, the same organization is both the provider and the user at the same time.

4.1.2 Terminology

We will now provide precise definitions of the terms “objective” and “adaptation point” that we use in the requirements phase and later on throughout this thesis.

- An **Objective** is a quantifiable goal that the applications shall meet. This quantifiable goal must be observable from outside the application.
- An **Adaptation Point** is a well defined interception interface that allows the cloud provider to take influence on the cloud user’s application at an arbitrary level of granularity. The cloud provider achieves this by setting a discrete value between a specified minimum and a maximum.

Finally, we need a name for the prototype. We decided to name the prototype **CA4S**, because it realizes our vision of **C**loud **A**pplication **A**daptation **a**s **a** **S**ervice. When we refer to the prototype as whole, we will use the term CA4S Prototype throughout this thesis.

Table 4.1: CA4S Prototype Requirements Summary

Requirement	Description
R1	The CA4S Prototype should allow the cloud user to express the objectives and adaptation points as a valid AWS template format in version “2010-09-09”.
R2	In order to operate the CA4S Prototype it should not be required for the cloud provider to modify the code base of OpenStack Heat.
R3	The CA4S Prototype shall enable a user to express the desired objectives of an application with the help of well defined expressions that the user embeds within the stack template file.
R4	The CA4S Prototype shall enable a user to specify the adaptation points of an application with the help of well defined expressions that the user embeds within the stack template file.
R5	The CA4S Prototype shall enable a user to define a strategy in case there are multiple options to perform adaptation point changes.
R6	The CA4S Prototype shall be implemented as an autonomic system in order to reduce the need for manual intervention for both the cloud provider and the cloud user.
R7	There shall be a way for the CA4S Prototype to propagate changed adaptation point values to the cloud application.
R8	The CA4S Prototype shall provide an endpoint so that the cloud application can transmit observation points reflecting the application’s current state to the CA4S Prototype.
R9	The CA4S Prototype should support multiple dimensions of elasticity, namely resource, cost and quality elasticity.
R10	The CA4S Prototype shall allow a developer to change the configuration selection algorithm.

4.2 Design

We now present a system design that addresses and justifies the requirements from Section 4.1.

4.2.1 R1: Valid CloudFormation Template

In our prototype, the cloud user should be able to express the desired objectives and adaptation points for his or her cloud application within a valid Heat template that adheres to the specification of the AWS CloudFormation template in version 2010-09-09. As we already pointed out in Section 2.2.3, we chose this template version because it is a minimum subset that allows an easy transition to later template versions. For our application we define a template to be valid if the command `template-validate` [62] reports no errors for the *WordPress Application Server Template* in Appendix C.2. Although this method is not a proof that the implementation fully adheres to the CloudFormation specification, we assume that this method is sufficient to show the viability of the CA4S Prototype.

If we want to specify the objectives within the Heat template file, we must examine the maximum number of characters that the Heat engine is able to process. The configuration file of Heat contains a configuration setting `max_template_size` which constrains the possible template size. In empiric tests that we conducted with our OpenStack installation we found that the practical limit of a template's maximum size is at around 1 Megabyte (MB). AWS CloudFormation constraints the template size to 300 KB [2]. We assume that even the smaller of both limits at 300 KB is a reasonable size for our proposed approach that is sufficient for most use cases. We back this assumption by the following two observations. *a*) In case the template size gets too large then the user still has the option to split the template file into multiple smaller ones. *b*) The largest template in the repository of Heat example templates is 13 KB large [57] which is considerably smaller than the maximum template size.

4.2.2 R2: No modification of Core Code

Research suggests that the development of new software components is more productive than the modification of existing code [10]. Therefore, our system architecture design follows a component based design. Our main objective is to present an architecture that does not require the provider to modify existing code from OpenStack in order to operate the CA4S Prototype. OpenStack Heat fosters the modular development of components by implementing a plug-in system. By providing a plug-in for OpenStack Heat we can extend the Heat templates in a way that allows a cloud user to specify the objectives and adaptation points within the template. We assume that this approach will promote the acceptance of our prototype. We will describe this plug-in system in more detail later in this chapter in Section 4.3.

4.2.3 R3: Specification of Objectives

In order to specify the cloud application objectives, we propose a distinct resource that holds the application objectives. This resource is "virtual" in that sense that it does not create a real resource in OpenStack but merely is a container for holding values. Such virtual resources are not

a new concept as they are already available in Heat. For instance, the `OS::Heat::RandomString` resource that we used in Listing 2.1 also is a virtual resource that provides a method for generating random strings, but does not actually create a resource. Therefore, we argue that the usage of virtual resources is a viable concept on which we can build our solution as well.

For each objective, the user adds one such "virtual" resource to the stack. Each of these resources will hold the objective in its properties definition. We argue that this approach has two advantages.

1. We create a clear semantic distinction between the user parameters in the templates and the objectives

2. The Heat engine refuses the template if the CA4S Prototype is not available on an OpenStack installation. This behavior gives a clear signal to the user that the managed adaptation capabilities are not available instead of failing silently.

We call the resource that we will use to specify the objective `CA4S::Objective`. This resource contains the mandatory properties `Objective`, `Cmp` and `Value`. The comparison operator `Cmp` specifies the desired range of acceptable values. Table 4.2 shows a listing of supported comparison operators. In order to define ranges it is possible to specify multiple `CA4S::Objective` resources that contain the same objective but constrain different ranges. For example, assume that the objective `ResponseTime` in an application should be ≥ 400 ms and $\leq 1,000$ ms. Listing 4.1 shows how a user can specify such a range with the help of two objectives. Apart from the mandatory keys that all objectives must contain, some objectives need additional keys. For example, if an objective specifies the desired response time of a certain endpoint, then the user must specify the address of this endpoint. It is possible to employ Heat's template functions (see Section 2.2.3) as parameter values.

Table 4.2: Supported Comparison Operators

Keyword	Symbol	Description
lt	$<$	Less than
lte	$\leq$	Less than or equal to
gt	$>$	Greater than
gte	$\geq$	Greater than or equal to
eq	$=$	Equal to
neq	$\neq$	Not equal to

4.2.4 R4: Specification of Adaptation Points

In accordance with the single template file design of Heat, the adaptation points are also defined within the Heat template file. In contrast to the high level objectives which the uses imposes on the stack, it is not immediately clear where an adaptation point does belong semantically. The question is whether these configuration values belong to the cloud application as a whole or to the individual resources. Both variants have arguments that speak for and against them. In a first

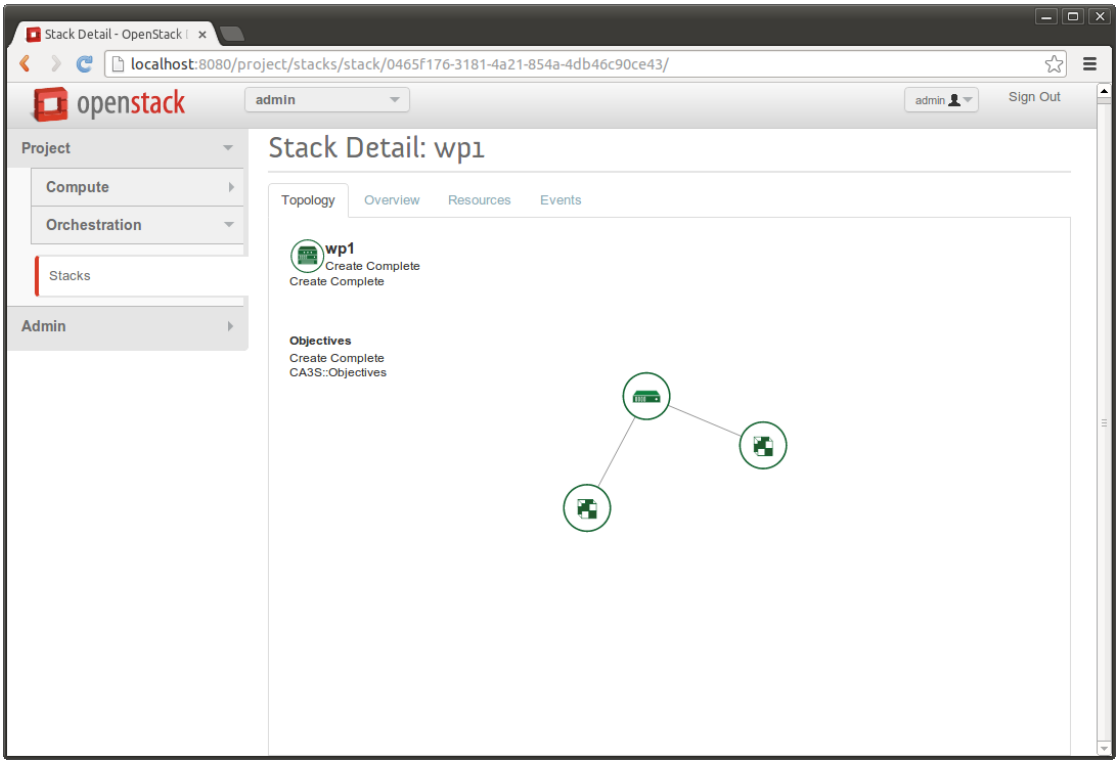


Figure 4.1: A managed stack that contains three resources: The computing instance, the resource that holds the adaptation points and the resource that holds the objectives.

revision of the prototype design we specified the adaptation points on a per-resource basis. In our experiments, the following two observations revealed the difficulties of this design.

- Resources are independent of each other: If in a horizontal scaling scenario there is a large number of similar resources that perform the same task, then each of the resource must receive the same configuration. But if we define the adaptation points on a per-resource basis then the prototype must synchronize the adaptation points, so that each resource of the same kind receives the same configuration. As an example, consider a web application like the one shown in Figure 1.2, where n identical web servers deliver the content to the client and that $n \geq 1$. In such a scenario we want that all web servers retrieve the same adaptation point values.

- Resources are volatile: When there is a load balancer that controls the number of instances, then each time the load balancer adds a new resource to the stack we must also create a new adaptation point resource. This is not trivial because we must create a hook that sends a signal to the CA4S Engine when the load balancer creates a new resource.

Therefore, we decided to move the adaptation point specification from the resource level up to the stack level, even though this increases the number of possible states. This means that

Listing 4.1: Specification of a high level objective within the properties of a dedicated resource type CA4S::Objective. In this case, the HTTP endpoint at localhost should have a maximum response time of $1,000$ ms.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "Template to demonstrate objective specification",
    "Resources": {
        "Objective_RT_gt_400": {
            "Type": "CA4S::Objective",
            "Properties" : {
                "Objective" : "ResponseTime",
                "Target" :  "http://localhost",
                "Cmp" : "gte",
                "Value" : 400
            }
        },
        "Objective_RT_lte_1000": {
            "Type": "CA4S::Objective",
            "Properties" : {
                "Objective" : "ResponseTime",
                "Target" :  "http://localhost",
                "Cmp" : "lte",
                "Value" : 1000
            }
        }
    }
}
```

the user defines all adaptation points for the cloud application for the stack. Consequently, the adaptation points are not linked to a particular resource.

Analogously to the objective definition, we propose a second auxiliary resource named `CA4S::AdaptationPoint` and that holds the adaptation points for the cloud application. The properties of this `CA4S::AdaptationPoint` resource are `AdaptationPoint`, `lte`, `gte` and optionally `value`. The `AdaptationPoint` property specifies the name of the adaptation point. With the keys `gte` ("greater than or equal") and `lte` "(less than or equal to)" the user expresses the range of possible target values for an adaptation point. A user can optionally set the adaptation point's initial value. If the user sets no value, then the CA4S Engine determines the initial value when it creates the stack. We present an example of such a specification in Listing 4.2.

4.2.5 R5: Specification of Strategies

In some cases it happens that multiple adaptation point values fulfill the desired objectives. As an example consider an application as Figure 1.2 shows it. Now assume that an objective that

Listing 4.2: Specification of adaptation points. In this example the adaptation point "QuickCacheTimeout" may have a value that is ≥ 0 and ≤ 20.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "Template to demonstrate adapation point definitions.",
    "Resources": {
        "AdaptationPoint_QuickCacheTimeout" : {
            "Type": "CA4S::AdaptationPoint",
            "Properties": {
                "AdaptationPoint" : "QuickCacheTimeout",
                "gte" : 0,
                "lte" : 20
            }
        }
    }
}
```

requires the application to respond within 300 ms. Let's further assume that the load balancer spreads the incoming requests evenly across the web server instances and that already a single web server instance is able to fulfill this objective. Consequently, also two instances will be able to fulfill the objective. The cloud user should be able to specify the prototype's desired behavior for such cases. For this we created the resource `CA4S::Strategy` with two properties `Objective` and `Function`. The property `Objective` refers an objective name (for instance "ResponseTime") and `Function` is either "min" or "max", depending whether the strategy is to minimize or to maximize the objective's value.

4.2.6 R6: Autonomic Component

In our design, the granularity of the autonomic management is on the stack level. This means that there is an autonomic control loop for each stack that a user creates. Because the control loop for the stacks are independent of each other the engine can execute them in parallel and on different machines. Consequently, the architecture allows scaling up with an increasing number of stacks to manage.

In reference to the MAPE-K model originally presented by IBM [37], we split the autonomic component into four stages: monitor, analyze, plan and execute. In the monitor phase the CA4S Prototype collects the black box data in accordance to the objectives. Analyzing and planning takes part in the concrete implementation that we will describe later. The prototype performs the execution by adjusting the concrete values of an adaptation point. Figure 4.2 depicts how we embedded our autonomic system into a MAPE-K autonomic control loop.

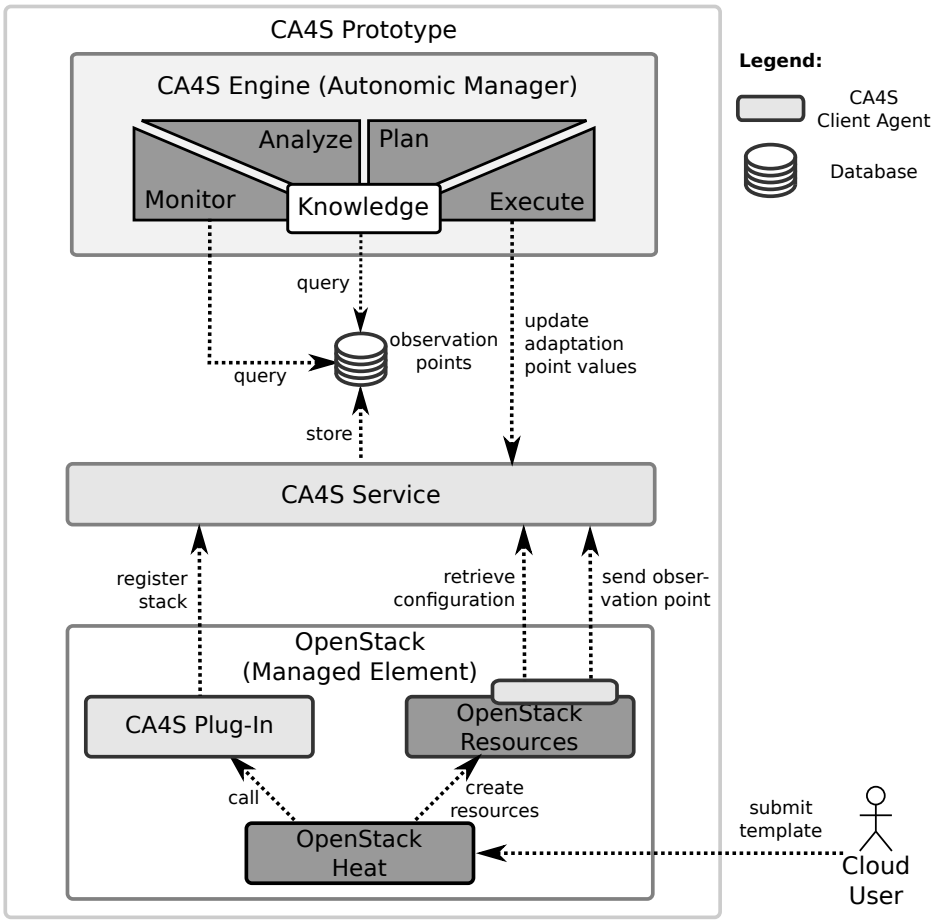


Figure 4.2: The CA4S Prototype implemented with a MAPE-K loop as proposed by IBM in [43]

4.2.7 R7: Configuration Propagation

To support elasticity, the CA4S Prototype must propagate changes in the adaptation point values to the client. In our prototype the autonomic manager will update the concrete value for the adaptation point by performing a REST API call to the CA4S Service. After the CA4S Prototype modifies an adaptation point, the application needs to process the changed values. For this the application must receive the changed values and then execute the appropriate actions. The way how the adaptation service transfers the adaptation point changes to the application is a fundamental design question. We evaluated the following two possibilities to address this issue.

1. Use the existing *Metadata service*. We described in Section 2.2.1 how instances query the metadata service to retrieve instance specific information. If we employ this service to propagate changed adaptation point values, we could utilize already existing infrastructure from OpenStack. Unfortunately, the metadata service does not allow the cloud operator to extend its functionality with plug-ins. Adding the functionality directly to the core code

would violate requirement R2 (No modification of core code). We therefore discard this option.

2. Create a custom service. This service works like the metadata service provided by Heat. It listens on a specific endpoint that is accessible from all cloud applications and delivers the adaptation point values for a specific cloud application on request. Like the OpenStack Metadata service, the CA4S configuration service resolves the configuration for a specific client based on the IP address that issued the query. The cloud application then processes the retrieved adaptation point values. This solution allows us to tailor the service exactly to our needs. The drawback of writing a custom service is that it incurs additional implementation overhead.

The resulting configuration service is a key element in the design of the CA4S Prototype (Figure 4.2). This service maintains a list of all objectives and adaptation points for which the model shown in Figure 4.4 is the basis. When the user creates a new stack, the CA4S Prototype registers the objectives and adaptation points via a REST service. The full specification of the configuration service is available in Appendix B.

4.2.8 R8: Observation Point Collection

In order for the CA4S Prototype to derive meaningful adaptation point value decisions it must have an insight into the current state and particular environment of the cloud application and its resources. Our prototype creates this insight by sending snapshots from the cloud application to the CA4S Prototype. In the remainder of this thesis we will refer to these application state snapshots as *observation points*. Each observation point contains a set of metrics where each metric contains a measured value of a certain application aspect. The exact source of an observation point, the level of granularity, the rate at which the application transmits the observation points as well as the exact list of captured metrics are dependent on the individual stack that the CA4S Prototype controls. We partition the set of observation point metrics into the following four different attribute classes: *Internal* attributes, *External* attributes, *Stack* attributes and *Adaptation Point* attributes. In Chapter 6 we will give concrete examples of the captured metrics as well as their classification into one of the four categories for different use cases. We now describe the four categories of observation point metrics in more detail.

- **Internal Attributes** reflect the internal state of the application. This may or may not be in response to external influences, i.e. not all internal attributes are necessarily influenced by some other attribute. A subset of these influences are the adaptation point attributes and the external attributes. On a computing resource, examples of metrics that belong to this category are system load, response time and the amount of active system memory. Not all collected internal attributes are necessarily needed for performing adaptation decisions.

- **External Attributes** capture the current demand to a system. In contrast to the other three attribute types, a cloud user cannot take influence on the external attributes. The only way to react to external attributes is to change the adaptation point attributes. The goal of the cloud application is to handle the current demand in the best way possible. In our

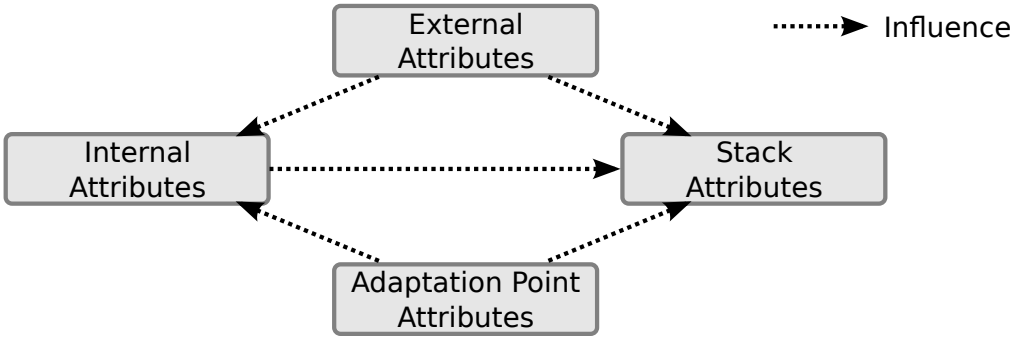


Figure 4.3: Influence graph that shows the relation between internal, external, stack and adaptation point attributes

system, we define "best way" as the system state that yields the lowest number of violated objectives with respect to the current demand.

- **Stack Attributes** Internal and external attributes reflect the state of the cloud application at an arbitrary level of granularity. Such a level of granularity, for instance, might be an individual resource. However, the user imposes objectives on "black box", high-level cloud application states. Therefore, the observation point must also contain the global state of the cloud application at the time of the observation. The sum of all individual resources of a cloud application determine these stack attributes. An example of a stack attribute is the total cost for the application. The adaptation point attribute that specifies the number of resources in a stack influences the cost. Figure 4.3 shows that also the other two attribute types influence the stack attributes.

- **Adaptation Point Attributes** take influence on the internal attributes and the stack attributes. In contrast to the external attributes, a user can take influence on the adaptation point attributes by assigning new values to them. As the adaptation point attributes belong to the stack, they are a special case of stack attributes in the sense that they also reflect the stack's global state. Again, the difference to stack attributes is that the cloud user can set any arbitrary discrete adaptation point attribute value, as long as this value remains within a specified range. We refer to the concrete set of adaptation point attributes as the configuration of the cloud application.

Figure 4.3 depicts the relationship between these four categories that we described above.

4.2.9 R9: Supporting Multi-Dimensional Elasticity

The prototype should support multi-dimensional elasticity. The notion of multi-dimensional elasticity that we use in this thesis is the one that Dustdar et al. propose in their research [26]. We now describe how our prototype design addresses the three elasticity dimensions *resource elasticity*, *cost elasticity* and *quality elasticity*.

Listing 4.3: Specification of cost elasticity

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "Template to demonstrate cost elasticty specification",
    "Resources": {
        "Objective_Costs_lte_20": {
            "Type": "CA4S::Objective",
            "Properties" : {
                "Objective" : "Costs",
                "Cmp" : "lte",
                "Value" : 20
            }
        }
    }
}
```

Resource Elasticity

In Section 1.1 we described how a user can create resource elastic cloud applications using Heat templates that contain load balancers, launch configurations and auto scaling groups. The problem with this approach is that the user must define and set the thresholds for the metrics on his or her own. To support resource elasticity in our prototype, the cloud user can add an adaptation point for the load balancer resource that specifies the number of running instances. We will provide an example of such an adaptation point that controls the number of instances in a load balancer in Chapter 6.

Cost Elasticity

The cost elasticity dimension covers the placement of instances with respect to their cost. For example, AWS charges the costs for "*Spot Instances*" based on the supply and demand of currently available computing resources [1]. A user sets a maximum amount that he or she is willing to pay for an instance. In order to model the cost elasticity dimension we propose the `Costs` objective that a user can employ. Listing 4.3 shows an example how to employ the `Costs` objective in order to express cost elasticity. In this example, the CA4S Prototype observes the "cost" stack attribute and places the configuration decisions in a way that the cost for running the cloud application stays below 20 per BTU.

Quality Elasticity

The CA4S Prototype supports the quality elasticity dimension because it is able to influence the cloud application by adjusting adaptation point attribute values. An example of such a quality adjustment is to enable a static cache. By enabling a cache, the server does not need to create the served page dynamically on the fly. This means that in the end the server can handle more users

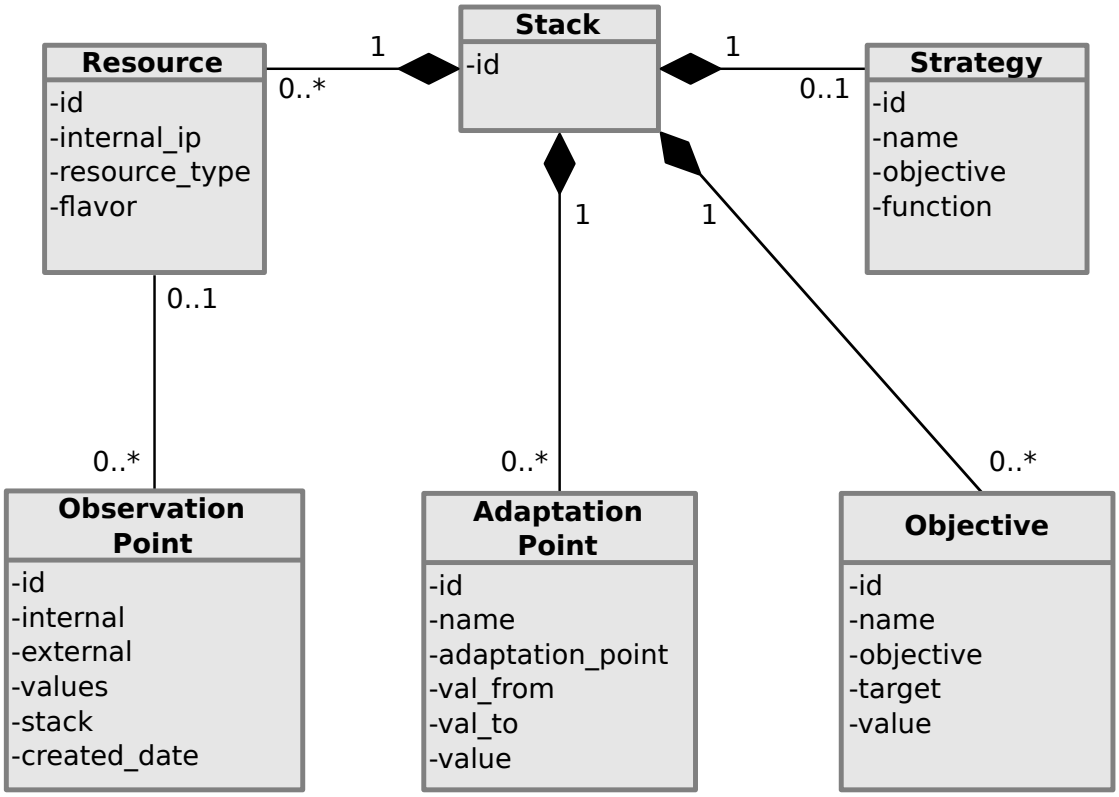


Figure 4.4: The stack model as UML class diagram.

without the need to increase the server capacity. On the other hand, enabling the cache incurs a reduction in the quality for the user, because the content that the web server delivers does not reflect the real time state, but the one at the time it created the cache.

4.2.10 R10: Change Configuration Selection Method

The cloud provider should be able to switch the implementation of the algorithm that selects the adaptation point attribute values. We achieve this with an interchangeable configuration selector class. This allows us to create alternative configuration selection implementations for the CA4S Prototype in order to perform additional experiments on the topic of adaptive cloud computing applications.

4.3 Implementation

We now present the components that comprise our prototype and describe concrete aspects of the implementation that are relevant to address the problem statements of this thesis. The scope of this section is not to provide a full documentation of every detail of the implementation. Such

a complete documentation is part of the technical documentation. Instead, we present the core aspects of the prototype.

We wrote all software components that we implemented for this thesis in Python. We chose this programming language because the OpenStack framework also is written in Python and therefore the CA4S Prototype integrates well into the OpenStack ecosystem. Our prototype implementation consists of four major components: *a*) The **CA4S Engine**, *b*) the **CA4S Service**, the *c*) **CA4S Heat Plug-in** and the *d*) **CA4S Client Agent** . All components are self contained entities that expose their service via openly available interfaces and interact with the other components as well as the OpenStack framework through these interfaces. With this design we follow the paradigm of a service oriented architecture [70] and the architecture of the other OpenStack components. Consequently, we reduced the dependency to OpenStack to a single component, namely the Heat Plug-in. The rest of the components have no direct coupling to OpenStack. Figure 4.2 shows the interaction between the individual components.

4.3.1 CA4S Engine

The CA4S Engine is an application that the cloud provider operates. This application implements the MAPE-K loop. For this, the CA4S Engine iterates over each running stack in the system.

For each stack the engine keeps a history of the last ten observation points. From this history the engine extrapolates the next external attribute value that it then passes to the configuration selector. We perform the extrapolation with univariate smoothing using a spline with a degree of one. There are two reasons why such an extrapolation is necessary: *a*) We can smooth single outliers and *b*) we have a simple prediction mechanism so that the CA4S Engine executes adaptive decisions before the violation occurs.

Based on this extrapolated value the CA4S Engine changes the adaptation point values for a particular stack or not. The CA4S Engine retrieves the observation points directly from the database. To change an adaptation point value, the CA4S Engine calls the REST API of the CA4S Service. Figure 4.5 shows the interaction between the CA4S Service, the database and the CA4S Engine.

4.3.2 CA4S Service

Figure 4.2 shows that the CA4S Service is the central component of the CA4S Prototype. This service is the hub that connects the OpenStack resources with the autonomic manager (i.e. the CA4S Engine). To maintain consistency with OpenStack Heat, we implemented the CA4S Service in Python. For the concrete implementation of the REST API we use the Pecan web framework [45]. The Ceilometer API also uses this lightweight framework. Pecan contains a development server that provides the HTTP endpoints. The server listens at an address on the internal network that all users from all running resources within an OpenStack installation can access. Appendix B lists all endpoints that the CA4S Service API supports along with a description of the concrete functionality.

For the instance the CA4S Service serves two main purposes: *a*) It provides the adaptation point attribute values (i.e. the configuration) to the application and *b*) it collects the observation points from the client. We now describe these two aspects in more detail.

Configuration Service

The CA4S Service delivers the configuration (i.e. the adaptation point attributes and their concrete values) to the cloud application. We implemented this as a REST service call that does not take any further parameters from the calling resource. As we already pointed out in Section 4.2.7, the OpenStack Metadata is the blueprint for this service. To implement this feature, we look up the IP address of the incoming request and match it against the registered resources in the database. The CA4S Service returns the stored adaptation points for the calling instance and their concrete value in the JSON format.

Observation Point Collection

The endpoint for the data collection receives the measured data from the resources. The CA4S Service then amends the observation point with the observed stack data and stores it to the database. In our implementation, we decided to use MySQL for storage. Basically we could also use Ceilometer for this task by creating custom metrics. Yet we decided to implement our own storage solution for the prototype implementation because it allows us to tailor it exactly to our needs. Due to the loose coupling via the REST API we can change the concrete storage engine implementation at a later point.

4.3.3 CA4S Heat Plug-in

OpenStack Heat provides a plug-in system that allows developers to create new template resources [59]. A developer implements such a resource plug-in for Heat by implementing a Python class that inherits from the `heat.engine.resource.Resource` base class. The plug-in can implement functionality for the life-cycle events create, update, delete, suspend and resume [59]. The implementation of any of these methods is optional. The Heat engine calls the respective method only if the method is available and the life cycle event occurs. The plug-in file also contains the registration of the resource name with the Heat engine. For this the plug-in file implements the method `resource_mapping` that returns a map which links the resource name to a concrete plug-in implementation. Listing 4.4 shows a skeleton of such a plug-in. In this example, the map connects the resource name `Sample::HeatPlugin` to the class `HeatPlugin`. Finally, the developer places the plug-in file inside one of the plug-in directories that the Heat configuration settings specify.

Resource properties have a fixed schema that a developer must specify inside the plug-in file. The plug-in contains a static map `properties_schema`. This map contains the name of the property as the key and an instance of `heat.engine.properties.Schema` as value. Among other properties, this instance specifies the data type, default values, constraints and so on. It also is possible to nest schemata which allows a developer to create multi-dimensional resource properties. Listing 4.5 shows an example definition of a property.

Listing 4.4: Heat Plugin Skeleton

```
from heat.engine import resource

class HeatPlugin(resource.Resource):

    properties_schema = {
        # properties schema
    }

    def handle_create(self):
        # called if the stack is created
        pass

    def handle_delete(self):
        # called if the stack is deleted
        pass

    def handle_update(self):
        # called if the stack is updated
        pass

def resource_mapping():
    return {
        'Sample::HeatPlugin': HeatPlugin
    }
```

When a user submits a template for a new cloud application that contains at least one of the resources `CA4S::Objective` and `CA4S::AdaptationPoint` to Heat, the `handle_create` method registers the new stack at the REST endpoint of the CA4S Service. According to the UML model that Figure 4.4 shows, the prototype first registers the stack, followed by the resources, objectives and adaptation points.

Adding new Adaptation Points

In the design that we propose in this thesis the cloud provider defines the available adaptation points. The reason why only the provider can add new adaptation points is that the data must be comparable between different users. In order to add a new adaptation point the provider has to implement a new class and create a new mapping for it. We organize the adaptation point class files analogously to the Heat plug-in files. They contain a mapping method `adaptation_point_mapping()` that returns a hash with the name of the adaptation point and the link to the class as value. This explicit mapping has the advantage that it is possible to map a single implementation to more than one adaptation point identifiers. The class file itself contains a method `quality` that receives an observation point as parameter. This method returns the

Listing 4.5: Definition of a property

```
from heat.engine import properties, resource

class CA4SAdaptationPoints(resource.Resource):
    properties_schema = {
        'QuickCacheTimeout': properties.Schema(
            properties.Schema.MAP,
            _('QuickCacheTimeout'),
            schema={
                "gte" : properties.Schema(
                    properties.Schema.INTEGER,
                    required=True
                ),
                "lte" : properties.Schema(
                    properties.Schema.INTEGER,
                    required=True
                )
            }
        ),
    }
```

Listing 4.6: Adaptation Point Definition File Skeleton

```
class InstancesAdaptationPoint:
    def quality(self, op):
        return 1.0

def adaptation_point_mapping():
    return {
        "Instances" : InstancesAdaptationPoint
    }
```

quality score for this adaptation point based on the passed observation point when the CA4S Engine performs a configuration selection. We will describe this selection process and the quality concept in Chapter 5. Listing 4.6 shows the skeleton of an adaptation point definition file.

Adding new Objectives

Adding new supported objectives to the CA4S Prototype happens in the same way as adding new adaptation points. The CA4S Prototype will parse all implementations that reside in a specific directory and make them available to the other services. Like the adaptation point definition file, the objective definition file also contains a mapping method that maps a map key to a concrete

Listing 4.7: Objective Definition File Skeleton

```
class CostsObjective:
    def amend(self, observationpoint):
        # set the stack values
        return observationpoint

    def filter(self, observationpoint):
        # specify an optional filter for this objective
        # if the filter returns True then the
        # passed observationpoint will not be considered
        return False

def objective_mapping():
    return {
        "Costs" : CostsObjective
    }
```

implementation. The key of the map specifies the name of the objective that the cloud user then uses in the Heat template file. Listing 4.7 demonstrates how to perform such an implementation. This example maps the key "Costs" to the instance `CostsObjective`. This instance implements two methods, namely `amend` and `filter`, both of which receive an observation point as parameter.

- The CA4S Service calls the `amend` method when the CA4S Client Agent submits an observation point. The purpose of this method is to amend the observation point with the stack data. After that, the method returns the amended observation point which the CA4S Service consecutively stores to the database.

- The `filter` method decides whether an observation point is usable in order to derive an adaptation point value decision for the implemented adaptation point or not. If the method returns `True` then the caller of the method will discard the observation point, otherwise the caller will add it to the list of observation points.

We will provide concrete examples for implementations of the methods `amend` and `filter` in Chapter 6.

4.3.4 CA4S Client Agent

The CA4S Client Agent is an application that runs on each stack instance. As a proof of concept, we created this client agent as a Linux *upstart* service. As the name suggests, the instance executes an upstart service once the operating system reaches a certain runlevel [38]. But because the configuration service is available as a REST service, it is possible to write agents for

other operating systems as well. This open design enables a developer to retrieve and process the configuration settings from any native application by accessing the REST service directly.

In our implementation the client agent serves two purposes: Firstly it executes configuration changes and secondly it submits the data to the CA4S Service, both of which we now describe in more detail.

Configuration Changes

First the CA4S Client Agent queries the CA4S Service to obtain the current values for the specified adaptation points. If a value for an adaptation point changes the CA4S Client Agent performs all necessary actions on the host resource to execute the changes.

The CA4S Client Agent features a modular design that allows a simple extension to add support for new adaptation points to the CA4S Client Agent. A configuration file specifies a mapping of adaptation points to *handler* classes. These handler classes contain methods that the CA4S Client Agent invokes when the CA4S Service returns a new, updated or deleted adaptation point. For instance the handler for the Apache HTTP Server adjusts the setting of the respective Apache configuration file and then reloads the Apache HTTP Server.

Observation Point Collection and Transmission

The second task that the CA4S Client Agent performs on the resource is to collect the observation point metrics and to transmit the observation points to the CA4S Service. This data collection is pretty straight forward: It runs in an infinite loop that collects and transmits the data.

4.4 Example Use Case

To illustrate the concept of the proposed CA4S Prototype in action, we now describe the sequence of steps that the prototype executes when a user creates a managed cloud application. Figure 4.5 depicts this sequence of steps as an UML sequence diagram. The steps in the sequence diagram are as follows.

1. First the user creates the template file. Next to the resources, this template file contains the newly introduced resources for the specification of the objectives and the adaptation points. The user then submits this template file to the Heat engine.

2. The Heat engine calls the `handle_create` method in the CA4S plug-in for each `CA4S::Objective` and `CA4S::AdaptationPoint` resource that the cloud user specified in the template.

3. The `handle_create` methods register the cloud application's objectives and adaptation points by calling the API endpoints of the CA4S Service.

4. The Heat engine launches the resources that the cloud user specified in the template file. When the resource boots up, it starts the CA4S Client Agent.

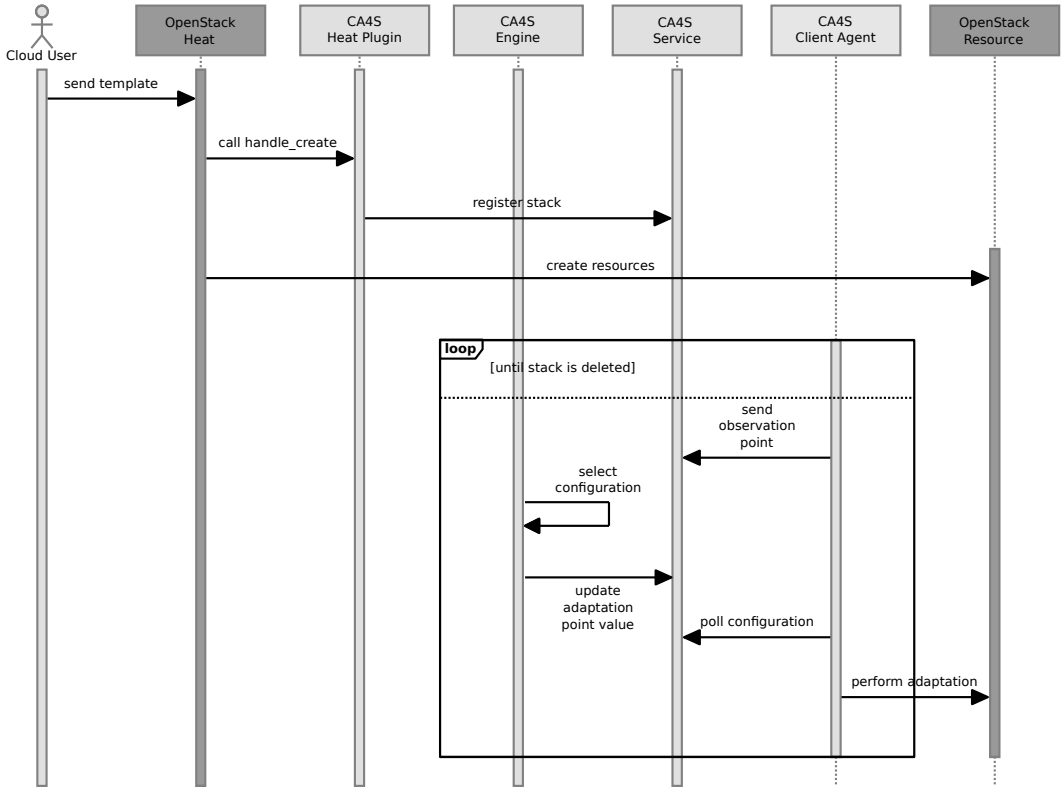


Figure 4.5: The autonomic feedback loop as simplified UML sequence diagram

5. The CA4S Client Agent continuously transmits the observation points of the resource it runs on to the CA4S Service.
6. The CA4S Engine continuously queries the list of all active stacks and calls the configuration selection implementation.
7. If the adaptation values change for a stack, then the CA4S Engine transmits the new values to the CA4S Service.
8. On the instance, the CA4S Client Agent periodically polls the adaptation point values from the CA4S Service.
9. If an adaptation point value changes, then the CA4S Client Agent applies the new settings on its host resource.

4.5 Future Improvements

The prototype that we present in this thesis is not a mature service in the sense that a provider can use it in productive environments. We left out some components that were not necessary to

prove the concept of our prototype but that we plan to address in future releases.

4.5.1 KeyStone Integration

The implementation of the presented prototype does not provide any security or authentication mechanisms. Without a doubt, the presented CA4S Prototype must provide effective measures to prevent unauthorized usage in a future release. For instance, it must not be possible to extract any sensitive data from other clients or to modify adaptation point values except for the CA4S Engine. The most straight forward way to implement an authentication mechanism in a future release is to employ the authentication and identity service that KeyStone provides.

4.5.2 Ceilometer Integration

The CA4S Prototype stores the observation points that the cloud application submits to the CA4S Service and that the CA4S Engine uses to derive adaptation decisions directly to a MySQL database. For the prototype we decided to use this native implementation of a database engine as it provided a direct and flexible way to provide a proof of concept. In a future release it is advisable to use Ceilometer as a storage engine to collect and process the cloud application's observation points.

CHAPTER 5

Configuration Selection Algorithm

In Chapter 4 we presented the architecture of the CA4S Prototype. One requirement of this prototype was that a developer can create custom implementations of the configuration selection algorithm that selects adaptation point attribute values (i.e., a configuration) for a given observation point. We now present such a concrete algorithm. The idea that motivates our approach is the following: For a given state that the external attribute values of an observation point reflect, we want to find out which concrete values for the adaptation points attributes likely maximize the quality with respect to a given set of objectives. To find this configuration, we compare the given observation point to the observation points in the knowledge base and select the configuration that increases the quality the most.

In the remainder of this chapter we first provide a formal definition of the adaptation point configuration selection problem in Section 5.1. Building on this definition, we propose an algorithm that solves the adaptation point configuration selection problem in Section 5.2. Finally, we discuss the properties and limitations of the presented approach in Section 5.3.

5.1 Formal Definition of the Configuration Selection Problem

We already pointed out in Section 4.2 that we categorize the attributes of an observation point into the four classes *internal attributes*, *external attributes*, *adaptation point attributes* and *stack attributes*. For the specification of the algorithm it suffices to define the attributes as arbitrary elements in a set (Definition 5.1).

Definition 5.1 ▶ Attribute sets

Let I be the set of all internal attributes, E be the set of all external attributes, A be the set of all adaptation point attributes and S be the set of all stack attributes. The attribute set AS is the set $AS = I \cup E \cup A \cup S$, where each element that exists in one of the sets I, E, A, S cannot exist in any of the other sets, i.e. $\forall s_1, s_2 \in \{I, E, A, S\} : s_1 \neq s_2 \Rightarrow s_1 \cap s_2 = \emptyset$.

The requirement that each element $a \in AS$ can only exist in either set I, E, A, S allows us to define a function that assigns a value to each attribute that belongs to an arbitrary observation point (Definitions 5.2 and 5.3). To determine to which type an attribute belongs to we can use intersections. For instance, to select all internal attributes of an arbitrary attribute set $AS_i \subseteq AS$, we create the intersection $AS_i \cap I$.

Definition 5.2 ▶ Observation Point Attribute Function

Let P be the set of all observation points. The observation point attribute function $o_{attr} : P \rightarrow \mathcal{P}(AS)$ returns the set of attributes that belong to an observation point $p \in P$.

Definition 5.3 ▶ Observation Point Value Function

The observation point value function $o_{val} : P \times AS \rightarrow \mathbb{R}$ assigns a concrete numerical value to an observation $p \in P$ and an attribute $a \in o_{attr}(p)$.

Next, we define a special set of observation points where all members have the same set of adaptation point attributes and where for each adaptation point attribute the value is the same. We call this special set an observation point bucket, or in short bucket (Definition 5.4).

Definition 5.4 ▶ Observation Point Bucket

Let $P_i \subseteq P$ be a set of observation points. We call P_i a bucket $b \in B$ if the following property holds: $\forall p_1, p_2 \in b : (o_{attr}(p_1) \cap A = o_{attr}(p_2) \cap A \wedge \forall a \in o_{attr}(p1) \cap A : o_{val}(p_1, a) = o_{val}(p_2, a))$.

Analogous to the observation point attribute function (Definition 5.2) and the observation point value function (Definition 5.3), we define the functions b_{attr} and b_{val} for buckets. In contrast to the observation point attribute function, the bucket attribute function is only defined for adaptation point attributes, as these are the only attributes where each element in the bucket must have the same value. The function b_{attr} returns the attribute set for a bucket and b_{val} returns the value of a given adaptation point attribute in a bucket.

Definition 5.5 ▶ Bucket Attribute Function

Let $b \in B$ be a bucket. The bucket attribute function $b_{attr} : B \rightarrow \mathcal{P}(A)$ returns the set of adaptation point attributes $A_i \subseteq A$, s.t. $\forall p \in b : o_{attr}(p) \cap A = A_i$.

Definition 5.6 ▶ Bucket Value Function

The bucket value function $b_{val} : B \times A \rightarrow \mathbb{R}$ returns the concrete numerical value of a bucket b and the adaptation point attribute $a_i \in b_{attr}(b)$ s.t. $\forall p \in b : o_{val}(p, a_i) = b_{val}(b, a_i)$.

In order to select a configuration later, we must define quality functions on adaptation points (Definition 5.10) and on buckets (Definition 5.11). But before we introduce these quality func-

tions, we must define objectives and the objective violation predicate. An objective is an arbitrary constraint that a user imposes on the application. For our definition, we define an objective as an element o from the set of all objectives O. The objective violation predicate allows us to determine whether an observation point meets a certain objective or not (Definition 5.7). With this predicate we then define a function that returns all those elements in a bucket that violate at least one objective (Definition 5.8). With the help of the violated observation points function we can define a violation ratio function that determines the percentage of the observation points that violate at least one objective in a bucket (Definition 5.9).

Definition 5.7 ▶ Objective Violation Predicate

Let $o \in O$ be an arbitrary objective from the set of all objectives O and let $p \in P$ be an arbitrary observation point. The objective violation predicate OB is a predicate such that $OB(o, p)$ holds if the observation point p meets the objective o.

Definition 5.8 ▶ Violated Observation Points Function

The violated observation points functions $violated : B \times \mathcal{P}(O) \rightarrow B$ takes a bucket $b \in B$ and a set of objectives $O_i \subseteq O$. The function returns a bucket $b_v \subseteq b$ that contains all the observation points from b that violate at least one objective $o \in O_i$, i.e. $b_v = \{p \in b | \exists o \in O_i : \neg OB(o, p)\}$.

Definition 5.9 ▶ Violation Ratio Function

Let $b \in B$ be a bucket and let $O_i \subseteq O$ be a set of objectives. We define the violation ratio function $vr : B \times \mathcal{P}(O) \rightarrow \mathbb{R}$ as

$$vr(b, O_i) = \begin{cases} \frac{|violated(b, O_i)|}{|b|} & \text{if } |b| > 0 \\ 0 & \text{if } |b| = 0 \end{cases}$$

We also need a function that determines the quality of an observation point with respect to a given adaptation point (Definition 5.10). With this quality function and the violation ratio function from Definition 5.9 we determine the quality of a bucket. This bucket quality is the product of the arithmetic means of the observation point qualities for each attribute, multiplied by one minus the violation ratio of the bucket (Definition 5.11). From this definition it follows that the bucket quality is 0, if all observation points in the bucket violate the objectives or the mean adaptation point quality for all observation points is 0.

Definition 5.10 ▶ Adaptation Point Quality Function

The quality function $q : A \times P \rightarrow \mathbb{R}$ maps an adaptation point attribute $a \in o_{attr}(p) \cap A$ and an observation point $p \in P$ to a numerical value in the range $[0, 1]$.

Definition 5.11 ► Bucket Quality Function

The bucket quality function $q_b : B \setminus \emptyset \times \mathcal{P}(O) \rightarrow \mathbb{R}$ maps a bucket $b \in B$ and a set of objectives $O_i \subseteq O$ to a numerical value in the range $[0, 1]$. We define the function value as $q_b(b, O_i) = \prod_{a \in b_{attr}(b)} (\frac{\sum_{p \in b} q(a,p)}{|b|}) \cdot (1 - vr(b, O_i))$.

Out of a set of buckets, the goal of our algorithm is to find those buckets with the maximum quality. Therefore, we must define a function that allows us to select a bucket in case that there exists more than one bucket with the same maximum quality. We call this function *strategy function* (Definition 5.12).

Definition 5.12 ► Strategy Function

Out of a set of buckets $B_S \subseteq B$, the strategy function $s : \mathcal{P}(B) \rightarrow B$ selects a single bucket $b_{sel} \in B_S$.

A user defines the concrete function mapping for the strategy function with respect to the overall goal of the application. In the most simple case, the strategy function returns an arbitrary bucket. However, if the overall goal of the application is to serve requests as fast as possible, the strategy function could select the bucket whose observation points minimize the mean value of the stack attribute "Response Time".

With Definitions 5.1 to 5.12 in place we now formulate the **Adaptation Point Configuration Selection Problem** (Definition 5.13).

Definition 5.13 ► Adaptation Point Configuration Selection Problem

Given a set of buckets $B_i \subseteq B$, a set of objectives $O_i \subseteq O$, and a given observation point $p_\varphi \in P$, find the set of buckets $b_{best} \subseteq B_i$ with the maximum bucket quality, i.e. $\forall b \in b_{best} : \forall b_i \in B_i : q_b(b, O_i) \geq q_b(b_i, O_i)$. If there is more than one element in b_{best}, then the strategy function $s(b_{best})$ determines the bucket to chose.

5.2 Algorithm Description

Based on the definitions in Section 5.1 we now describe the adaptation point configuration selection algorithm that we have created for the CA4S Prototype. The algorithm consists of four stages. The first three steps filter and normalize the input, before the final step performs the configuration selection. Figure 5.1 shows the steps through which the observation points go.

5.2.1 Step 1: Filtering

In the first step of the algorithm we take a set of historical observation points $P_i \subseteq P$ and filter out all elements of this set that are not relevant for the configuration decision (Algorithm 5.1).

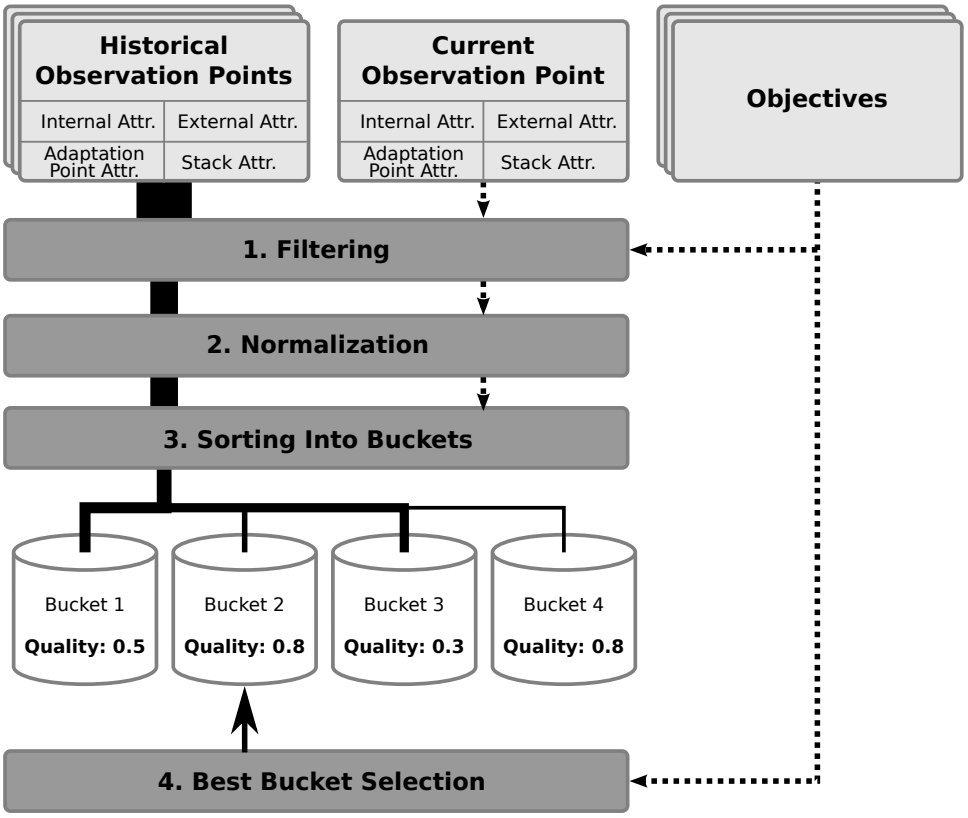


Figure 5.1: The four steps of the configuration selection algorithm

We consider an historical observation point $p \in P_i$ to be relevant to the current observation point p_φ if it matches the following criteria.

1. The historical observation point p contains the same set of external attributes as the current observation point p_φ, i.e. $o_{attr}(p) \cap E = o_{attr}(p_\varphi) \cap E$. The external attributes contain the values that the user cannot influence, such as the incoming requests per second (see Section 4.2.8 for a detailed description of the different attribute types). We need the same set of external attributes so that we can calculate the distance between two observation points in step 3.

2. The historical observation point p contains the same set of adaptation point attributes as the current observation point p_φ. We justify this requirement by pointing out the problems that arise if either observation point contains an adaptation point attribute that the other observation point does not contain.

    a) Let $a \in A$ be an adaptation point attribute that p contains but p_φ does not contain, i.e. $a \in o_{attr}(p) \cap A \wedge a \notin o_{attr}(p_\varphi) \cap A$. In this case the attribute a influences the state of p, but this state is invisible to p_φ. Therefore, we cannot use the historical observation point p to derive a decision for the current observation point p_φ.

b) Let $a \in A$ be an adaptation point attribute that p does not contain but p_φ contains, i.e. $a \notin o_{attr}(p) \cap A \wedge a \in o_{attr}(p_\varphi) \cap A$. In this case it is not possible to select a concrete value for the adaptation point a of p_φ because p does not make any statements about a.

3. We filter out any historical observation point $p \in P_i$ that is not suited to derive a decision about the current observation point p_φ for any particular reason. For this we introduce the objective filter predicate (Definition 5.14). Despite looking similar to the objective violation predicate in Definition 5.7, there is an important difference between these two predicates: If the objective filter predicate does not hold for a particular observation point p, then the algorithm does not consider p in any of the subsequent steps. The objective violation predicate on the other hand is used to determine the violation ratio for a bucket.

Definition 5.14 ▶ Objective Filter Predicate

Let $o \in O$ be an arbitrary objective from the set of all objectives O and let $p \in P$ be an arbitrary observation point. The objective filter predicate OF is a predicate $OF(o, p)$ that holds if the configuration decision algorithm cannot use the observation point p to derive a configuration for the objective o.

input : A set of objectives $O_i \subseteq O$, a set of observation points $P_i \subseteq P$ and a given observation point $p_\varphi \in P$
output: A set of filtered observation points $P_i^* \subseteq P_i$

```
1  P_i^* = ∅
2  for each p ∈ P_i do
3      if o_attr(p_φ) ∩ E ≠ o_attr(p) ∩ E then
4          continue ;                          // The EA sets must be equal
5      end
6      if o_attr(p_φ) ∩ A ≠ o_attr(p) ∩ A then
7          continue ;                          // The AP sets must be equal
8      end
9      for each o ∈ O_i do
10         if OF(o, p) then
11             continue 2;                         // Objective filter
12         end
13     end
14     P_i^* ← P_i^* ∪ {p}
15 end
16 return P_i^*
```

Algorithm 5.1: Observation Point Filtering

5.2.2 Step 2: Normalization

In the second step of our algorithm we normalize the external attributes values of the observation points that passed the filter step as well as the given observation point p_φ (Algorithm 5.2). We must normalize the observation point attribute values because otherwise the algorithm biases the euclidean distance calculation towards attributes with larger numerical ranges. As an illustrative example, consider two external attributes $e_1, e_2 \in E$. For some given set of observation points $P_i \subseteq P$, let the attribute values for e_1 be in the range $[0, 5]$, whereas the attribute values for e_2 are in the range $[0, 2^{32}]$. Without normalization, the attribute values of e_1 have a negligible impact on the distance calculation. Min/max scaling normalizes each external attribute value to the range $[0, 1]$, so that each attribute gets an equal weight in the distance calculation. In order to perform the Min/max scaling, we define a minimum and a maximum function on a set of observation points with respect to some given external attribute (Definition 5.15). With these functions, we construct a "copy" of an observation point p that has the same set of attributes and values, except that the external attribute values are min/max normalized with respect to a given set of observation points (Definition 5.16). The subset $\hat{P} \subseteq P$ is the set of all observation points where the external attribute values are in the range $[0, 1]$.

Definition 5.15 ▶ Observation Point Set Attribute Minimum and Maximum

The function $e_{min} : \mathcal{P}(P) \times E \to \mathbb{R}$ takes a set of observation points $P_i \subseteq P$ and an external attribute $e \in E$, where the following property must hold: $\forall p \in P_i : e \in o_{attr}(p)$. We define the function value as follows.

$$e_{min}(P_i, e) = \min(\bigcup_{p \in P_i} \{o_{val}(p, e)\})$$

Analogously, we define the bucket maximum function $e_{max} : \mathcal{P}(P) \times E \to \mathbb{R}$ with the function value as follows.

$$e_{max}(P_i, e) = \max(\bigcup_{p \in P_i} \{o_{val}(p, e)\})$$

Definition 5.16 ▶ Normalized Observation Point

The normalized observation point function $n_{op} : P \times \mathcal{P}(P) \to P$ takes an observation point $p \in P$ and a set of observation points $P_i \subseteq P$ and returns the normalized observation point $\hat{p} \in P$, for which the following properties hold:

1. $o_{attr}(p) = o_{attr}(\hat{p})$
2. $\forall a \in o_{attr}(p) \setminus E : o_{val}(\hat{p}, a) = o_{val}(p, a)$
3. $\forall e \in o_{attr}(p) \cap E : o_{val}(\hat{p}, e) = \frac{o_{val}(p,e) - e_{min}(P_i,e)}{e_{max}(P_i,e) - e_{min}(P_i,e)}$

input : A set of observation points $P_i \subseteq P$ and a given observation point $p_\varphi \in P$
output: The set of normalized observation points $\hat{P}_i \subseteq P$ and the normalized given observation point $\hat{p}_\varphi \in P$

$P_{all} \leftarrow P_i \cup \{p_\varphi\}$
$\hat{P}_i \leftarrow \emptyset$
for each $p \in P_i$ **do**
  $\hat{P}_i \leftarrow \hat{P}_i \cup \{n_{op}(p, P_{all})\}$
end
return $(\hat{P}_i, n_{op}(p_\varphi, P_{all}))$

Algorithm 5.2: Observation Point Normalization

5.2.3 Step 3: Sorting into Buckets

Next, we sort the data into buckets (Algorithm 5.3). In this step we read the filtered and normalized observation points from the two previous steps and calculate the euclidean distance of the external attribute values for each observation point $p \in \hat{P}_i$ to the normalized given observation point $\hat{p}_\varphi$ (Definition 5.17). If the calculated distance is within the user specified range $dist_{max}$ that the algorithm takes as parameter then we add the observation point to the set of the near observation points. The reason why we do not perform the distance filtering in the first step is that for the distance calculation we need the normalized external attribute values of the filtered observation points.

We then sort the observation points whose distance is within $dist_{max}$ into the set of buckets B_S. To formalize the construction of the bucket set, we define the bucket retrieval function (Definition 5.18). This function takes an observation point p and a set of buckets B_{in}, that initially is the empty set $\emptyset$. It then returns the bucket B_i to which p belongs as well as the set B_{out}, which is the input set excluding B_i. If B_{in} does not contain a bucket where p belongs to, then the bucket retrieval function returns the empty set $\emptyset$. Our algorithm then adds p to B_i and adds B_i to the set of buckets B_S. An important property of the bucket set B_S is that for each bucket it contains there must be at least one attribute value that is different from all the other buckets. With this property we construct a set that partitions the set of observation points into a set of buckets, in which for each combination of bucket attribute values only one set exists that contains all the observation points that belong to this bucket.

Definition 5.17 ▶ Observation Point Distance

We define the distance between two normalized observation points $p_1, p_2 \in \hat{P}$ that have the same set of external attributes $o_{attr}(p_1) \cap E = o_{attr}(p_2) \cap E$ as the function $opdist : \hat{P} \times \hat{P} \rightarrow \mathbb{R}$. The function value is the euclidean distance between the observation point's external attribute values, i.e.

$$opdist(p_1, p_2) = \sqrt{\sum_{e \in o_{attr}(p_1) \cap E} (o_{val}(p_1, e) - o_{val}(p_2, e))^2}$$

Definition 5.18 ▶ Bucket Retrieval Function

The bucket retrieval function $br : P \times \mathcal{P}(B) \rightarrow B \times \mathcal{P}(B)$ takes an observation point $p \in P$ and a set of buckets B_{in}. The function returns a bucket B_i and the set of buckets B_{out} with the following properties:

1. $B_{out} = B_{in} \setminus B_i$
2. $\forall b_1, b_2 \in B_{out} : (b_1 \neq b_2 \Rightarrow \exists a \in b_{attr}(b_1) : b_{val}(b_1, a) \neq b_{val}(b_2, a))$
3. $B_i \neq \emptyset \Rightarrow b_{attr}(B_i) = o_{attr}(p)$
4. $\forall a \in b_{attr}(B_i) : b_{val}(B_i, a) = o_{val}(p, a)$

input : A set of observation points $P_i \subseteq P$, a given observation point $p_\varphi \in P$ and a maximum distance $dist_{max} \in \mathbb{R}$
output: A set of buckets B_S

$P_{near} \leftarrow \emptyset$
for each $p \in P_i$ **do**
  if $opdist(p_\varphi, p) \leq dist_{max}$ **then**
    $P_{near} \leftarrow P_{near} \cup \{p\}$
  end
end
$B_S \leftarrow \emptyset$
for each $p \in P_{near}$ **do**
  $(B_i, B_S) \leftarrow br(B_S, p)$
  $B_i \leftarrow B_i \cup \{p\}$
  $B_S \leftarrow B_S \cup B_i$
end
return B_S

Algorithm 5.3: Sorting observation points into buckets

5.2.4 Step 4: Best Bucket Selection

In the final step, we select the bucket which yields the best quality (Algorithm 5.4). For this we iterate over the set of buckets that the previous step 3 has returned and perform the following steps.

- Initially, we look for the first bucket that has a quality > 0. The algorithm sets this bucket as the best.

- If any of the consecutive buckets yields a better bucket quality (Formula 5.11) than the current best bucket, we set this bucket as the new best bucket.

- If a bucket has the same quality score as the currently best bucket, then the strategy function (Definition 5.12) decides which bucket to choose. With the strategy function, the user can for instance choose to select the bucket that minimizes the cost.

In case that there is no bucket with a quality greater than 0 or there is no bucket at all, the algorithm returns the empty set. This leaves the decision how to deal with the case that there is no suitable configuration up to the cloud application.

input : A set of buckets $B_S \subseteq B$ and a set of objectives $O_i \subseteq O$
output: The bucket $b_{best} \in B_S$ with the best quality

```
1  b_best ← ∅
2  for each b ∈ B_S do
3      if b_best = ∅ then
4          if q_b(b, O_i) > 0 then
5              b_best ← b
6          end
7          continue
8      end
9      if q_b(b, O_i) > q_b(b_best, O_i) then
10         b_best ← b
11     end
12     if q_b(b) = q_b(b_best, O_i) then
13         b_best ← s({b, b_best})
14     end
15 end
16 return b_best
```

Algorithm 5.4: Best bucket selection

5.2.5 The Final Algorithm

We now put the four pieces together and construct the final configuration selection algorithm (Algorithm 5.5). This algorithm selects the bucket that contains the best configuration with respect to a set of objectives, a set of historical observation points, a given observation point and the maximum distance.

input : A set of objectives $O_i \subseteq O$, a set of observation points $P_i \subseteq P$, a given observation point $p_\varphi \subseteq P$ and the maximum distance $dist_{max}$
output: An adaptation point bucket $b_{best} \in B$

1 $P_i^* \leftarrow filter(O_i, P_i, p_\varphi)$; /* Algorithm 5.1 */
2 $(\hat{P}_i, \hat{p}_\varphi) \leftarrow normalize(P_i^*, p_\varphi)$; /* Algorithm 5.2 */
3 $B_S \leftarrow sort\_into\_buckets(\hat{P}_i, \hat{p}_\varphi, dist_{max})$; /* Algorithm 5.3 */
4 **return** $select\_best\_bucket(B_S, O_i)$; /* Algorithm 5.4 */

Algorithm 5.5: The algorithm for performing a configuration decision

5.3 Limitations

Before we perform a detailed, practical evaluation of the presented algorithm in Chapter 6, we conclude this chapter with a discussion of the presented algorithm's inherent limitations.

5.3.1 Maximum Number of Attributes

A fundamental problem that arises with an increasing number of adaptation point attributes and adaptation point values in the set of historical observation points is that the number of buckets increases exponentially. A cloud application with 10 adaptation point attributes where each adaptation point has 10 possible values potentially yields 10^{10} buckets that the algorithm must evaluate, if there are historical observation points for each combination. Moreover, for each bucket we need observation points that cover each possible combination of external attributes. Therefore, we have to limit the number of adaptation point attributes, external attributes and their respective values to a minimum.

5.3.2 The Curse of Dimensionality

Distance based approaches and consequently our algorithm as well suffer of an inherent weakness called the "*curse of dimensionality*". Beyer et al. have shown that with an increasing number of dimensions, the distance to the nearest neighbor converges towards the farthest neighbor's distance [7]. For certain distributions of data this effect appears in data sets with 15 dimensions [7]. Applied to our algorithm we conclude that the provider must choose the external attributes in a way so that they discriminate well between the different states. There is also an upper bound for the number of external attributes for which the algorithm will return reasonable results.

5.3.3 Performance

For deriving the proof of concept we use a naïve approach by calculating the euclidean distance for each entry in the set of historical observation points. This leads to a complexity of each configuration decision of $\mathcal{O}(|P_i|)$ with respect to the number of observation points in the set

$P_i \in P$. In a production environment with $2,000$ nodes where each node sends a data point every 10 seconds, the knowledge base grows by than $17,280,000$ entries every day. Consequently, a naïve approach of iterating through all observation points is not feasible in larger installations. Therefore, we propose two directions for future research to address the performance problem:

1. **Data Reduction**. A possible approach is to use prototype selection that k-Nearest Neighbors (k-NN) classifiers to increase the classifcation speed [31]. Another variant of data reduction is to limit the number of elements that each configuration bucket may contain.

2. **Pre-calculate and cache the decisions**, e.g. by translating them into rules. We will show in Chapter 6 that pre-calculation is a feasible approach for observation point sets with a low number of external attributes. Pre-calculated rules are faster in performing the configuration decisions. This approach has also the advantage that a user can acknowledge the rules before applying them on a productive system. The drawback is that the configuration selection is less dynamic.

CHAPTER 6

Evaluation

We chose three evaluation scenarios to evaluate the CA4S Prototype. In Section 6.1 we evaluate our system with the help of a simulator. In Section 6.2 we test the CA4S Prototype with a real world application scenario using an Apache HTTP Server and WordPress. Finally, we perform an end to end test of the autonomic system in Section 6.3. Each of these evaluations covers a different aspect of the CA4S Prototype, which we summarize in Table 6.1.

Table 6.1: Aspects covered in the evaluation

	Simulator *(Section 6.1)*	**WordPress** *(Section 6.2)*	**End To End** *(Section 6.3)*
CA4S Service			
Register Adaptation Points and Objectives	✓	✓	✓
Query Configuration from Service	✓	✓	✓
CA4S Plugin			
Register Template Resources		✓	✓
CA4S Engine			
Configuration Selection	✓	✓	✓
Autonomic Configuration Change			✓
CA4S Client Agent			
Send Observation Points		✓	✓
Execute Adaptation Point Changes			✓
Dimension of Elasticity			
Resource Elasticity	✓		
Cost Elasticity	✓		
Quality Elasticity	✓	✓	✓

6.1 Simulator

Testing algorithms on physical resources is time and cost intensive. Although the modeling of a simulator is a non-trivial task [34], researchers employ simulation engines to test the behavior of distributed systems, for instance in [52]. With the *CloudSIM* toolkit there even exists a dedicated simulation engine for simulating cloud computing environments [16]. Therefore, in addition to the evaluations on the VMs in Sections 6.2 and 6.3, we also chose to evaluate certain aspects of the prototype with a simulator (e.g. the support for different dimensions of elasticity) for the following reasons.

- While the in situ evaluation with WordPress shows slight variations in each test run, the results of the simulator are exactly reproducible with respect to the given input.
- No OpenStack dependency is necessary in order to evaluate the CA4S Service and the configuration selection algorithm.
- Compared to an in situ test, the simulator allows us to evaluate the CA4S Prototype in a fraction of the time, because we do not need to launch any resources in OpenStack.

6.1.1 Setup

Our setup simulates a cloud application that consists of a load balancer resource and a set W of n simulated web servers resources $W = \{w_1, w_2, \ldots, w_n\}$. Both resources implement the two methods `request` and `run`. The `request` method registers a single request to the resource. To simulate more than one request within a single time unit, we consecutively call the `request` method multiple times. To each incoming request the load balancer assigns a unique, strictly monotonically increasing number r_{num} and distributes the load to the web server resource $w_i \in W$, where $i = r_{num} \mod n$.

After the simulator has sent all requests to the load balancer resource, the `run` method increases an internal time counter and sends the observation point to the CA4S Service. We call such a sequence of registering requests and calling the `run` method as a cycle. Each time the simulator calls the `run` method, a cycle ends and a new one starts.

6.1.2 Observation Points

In the simulator, the observation points that the load balancer resource sends to the CA4S Service and consist of the following attributes.

- Internal Attributes
  - **Added Requests**: If a web server accepts an incoming request, then we add it to the list of added requests. On the load balancer the added requests internal attribute is the sum of all requests that the simulator has added to any of the web server instances during the past cycle.

  - **Rejected Requests**: Analogously to the number of added requests, the number of rejected requests is the sum of requests that the web server resources have rejected during the past cycle.

- External Attributes
  - **Incoming Requests**: This attribute value is a non-negative integer that counts the number of calls to the `requests` method during the past cycle. This attribute reflects the simulated demand to the application.

- Adaptation Point Attributes
  - **Instances**: This adaptation point attribute describes the number of simulated web server instances that the stack contains and simulates an adaptation point on the resource elasticity dimension. The evaluation with the simulator contains attribute values that are integers between 1 and 3. The number of instances does not influence the quality score for this adaptation point.
  - **BounceRate**: This adaptation point attribute enables elasticity on the quality dimension. In our simulation, the value for this adaptation point is either 0, 2, 3 or 4. A bounce rate of b rejects every b^{th} request, e.g. if the bounce rate b is 2 then the server drops every second request. The quality for this adaptation point is one minus the ratio of rejected requests to total requests (i.e. the sum of added requests and rejected requests).

- Stack Attributes
  - The **Simulator Response Time**. We simulate the response time behavior of a VM with a modified cumulative distribution function (Formula 6.1). We determined the constant values μ, σ and m of this formula with empiric tests and set μ to 34, σ to 16 and the multiplier value m to $3\,000$. This gives us a model that starts at a simulator response time of approximately 50 for one added request per cycle and monotonically increases until it reaches a peak of $3\,000$ at approximately 85 added requests per cycle. This approximates the response time behavior of an actual VM, for instance the one in Figure 6.2(a). Because the simulator response time is a function of the arithmetic mean of added requests of all web servers, the configuration selector can decrease the response time for a given number of incoming requests either by rejecting requests or adding additional web server resources.
  - The **Simulator Cost**. We calculate the simulated cost with the formula $c = (1 + |W|) \cdot 52$. Consequently, this cost model adds for each web server resource and the load balancer resource a fixed cost of 52 units per BTU.

Formula 6.1: Simulator Response Time

$$srt(x) = \frac{1+erf(\frac{x-\mu}{\sigma \cdot \sqrt{2}})}{2} \cdot m$$

6.1.3 Test data generation

To generate a corpus of observation points we use the following method: We send requests to the simulator in such a way that for each possible discrete value of the external attribute "incoming requests" in the range $[1, 100]$ the simulator sends exactly four observation points to the CA4S Service. We repeat this for each of the twelve possible value combinations of the adaptation point attributes "Instances" and "BounceRate". Therefore, our test data set contains $100 \cdot 4 \cdot 3 \cdot 4 = 4\,800$ observation points.

6.1.4 Results

We create the simulated stack with a *strategy* function that minimizes the total costs for the stack. Because the external attribute "Incoming Requests" is a discrete value, we set the maximum distance for the bucket selection algorithm to 0, i.e. the external attribute values must match exactly to be sorted into the respective bucket. We evaluate the results with the following four combinations of the objectives "Simulator Response Time" and "Costs".

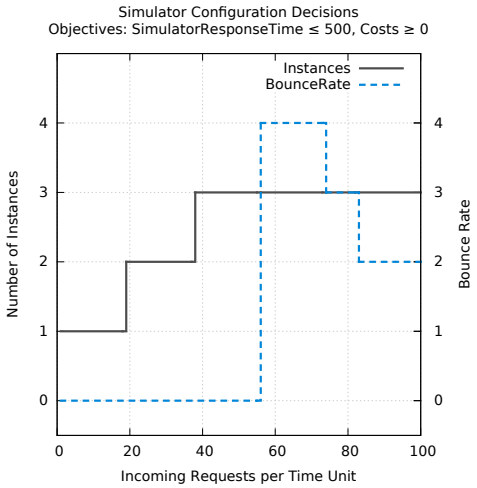
1. *Simulator Response Time* ≤ 500 *and Costs* ≥ 0: Because the number of instances does not affect the quality, the strategy function initially selects a single instance to minimize the costs. When the number of incoming requests increases, the configuration selector adds additional instances. If we reach the maximum number of three instances and the number of incoming requests increases further, the bounce rate changes from 0 to 4, i.e. the simulator rejects every 4^{th} request. After that, the bounce rate drops to 3 and finally to 2 (Figure 6.1(a)).

2. *Simulator Response Time* ≤ 500 *and Costs* ≤ 200: Again, the simulator first increases the number of instances. But because in this scenario we set an upper bound of 200 for the costs, the configuration selector cannot launch more than two instances. Instead, the configuration selector bounces the requests earlier (Figure 6.1(b)). The graph discontinues at approximately 75 incoming requests, because there are no more known configurations available that can handle the amount of incoming requests.

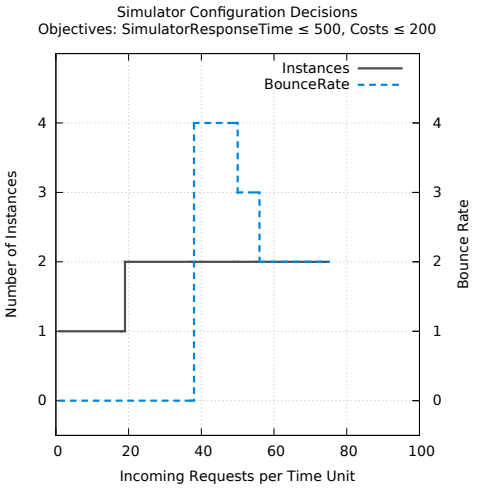
3. *Simulator Response Time* $\leq 1\,000$ *and Costs* ≥ 0: Compared to Figure 6.1(a), we increase the maximum response time from 500 to $1\,000$ this time. The behavior of the configuration selector is the same as in the case with a maximum response time of 500, except that this time the adaptations happen later on the x-axis (Figure 6.1(c)).

4. *Simulator Response Time* $\leq 1\,000$ *and Costs* ≤ 200: Here the behavior again is the same as with a lower response time, except that the configuration selector performs the adaptation point value changes later (Figure 6.1(d)).

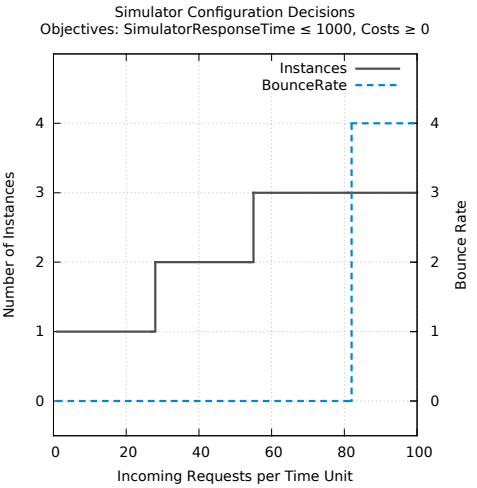
This result shows that for the same set of observation points the configuration selector behaves differently, depending on the provided objectives. The experiment also demonstrates the ability of the prototype to cover the three different dimensions of elasticity, namely the resource elasticity, the quality elasticity and the cost elasticity.



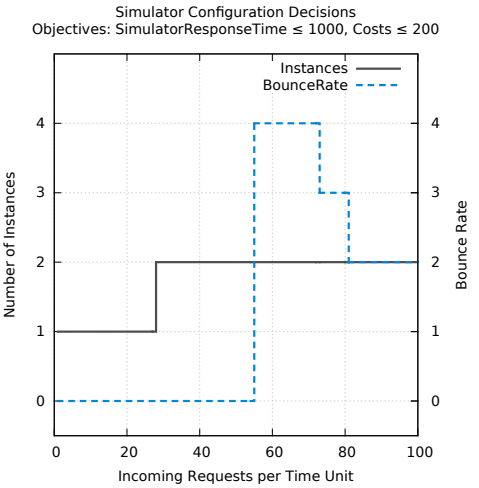
(a) With no upper cost limit, the number of instances scales up to the maximum of three. After that the configuration selector bounces the requests.



(b) With an upper bound of 200 for the Cost objective, the configuration selector starts at most two instances.



(c) With an increased maximum response time the instances scale up later



(d) Like in Figure 6.1(c), the configuration selector also performs the adaptations later if the costs have an upper bound.

Figure 6.1: The simulator results show how from the same set of observation points the CA4S Prototype derives configuration decisions based on the objectives.

6.2 Apache HTTP Server with WordPress

For this evaluation scenario we now launch actual VM instances in OpenStack in order to evaluate the specification of Objectives and Adaptation Points in the OpenStack Heat templates as well as the ability of the CA4S Client Agent to send observation points to the CA4S Service (Table 6.1).

6.2.1 Setup

We perform this test with a custom OpenStack machine flavor that has a RAM size of 512 MB, a disk size of 10 GB and one virtual CPU. With this flavor we launch an instance from the Ubuntu 12.04.4 LTS image and install the Apache HTTP Server on it. As the application for the web server we install the web blog software *WordPress*. This software is written in PHP and uses a MySQL database. According to [14], more than 13 million web sites use WordPress, which makes it a representative of a "typical" web application for our evaluation. When the instance launches, we import a data set with 56 blog posts and 15 comments into WordPress. Finally, we install the CA4S Client Agent on the host instance. We provide the full OpenStack Heat template listing that we use to provision the evaluation instance in Appendix C.

6.2.2 Observation Points

The CA4S Client Agent periodically collects the observation points on the host instance and submits them to the CA4S Service.

To determine the number of requests we implemented a parser for the Apache HTTP Server access log file. We read the file line by line in backward chronological order. For each entry in the log file we increment a counter for the respective HTTP status code group. If the parser reaches a log entry that is older than a specified threshold, the parser stops and returns the aggregated results. From this aggregated results we determine the number of requests per seconds as well as the total number of requests within the past 60 seconds, the past 5 and the past 15 minutes. If the system has been running for less than 1, 5 or 15 minutes, we extrapolate the corresponding number.

Although we do not need all the attributes that the CA4S Client collects, we still submit them to the CA4S Service, so that we potentially can use the attributes for new adaptation points and objectives in the future.

- External Attributes
  - The **number of requests** per second that the application serves. This is the sum of all requests with the HTTP status codes 2xx (successful) and 5xx (server error) [28].
- Internal Attributes
  - The **number of running Apache processes** and their **memory consumption**. We collect this value by parsing and filtering the list of running processes.

  - The **number of processes**, the information about the current **CPU usage**, the physical and virtual **memory** that is available and used. We collect this data with the **vmstat** utility [80].
  - The system's **load average** as an indicator for the current CPU wait queue length [72].
  - Statistical information from the network interfaces from the **ifstat** utility.

- Stack Attributes
  - We amend the observation points with the **Response Time** of the application. We do this by sending an HTTP request to the endpoint that the user specifies in the template and measuring the time it takes for the response to complete.

- Adaptation Point Attributes
  - We perform two evaluations with the different adaptation points "BounceRate" and "QuickCacheTimeout". We will describe both adaptation points in more detail in Section 6.2.4 when we discuss the evaluation results.

6.2.3 Test Data Generation

To create the payload on the test instances we use a custom payload generator that takes four parameters: *a*) The endpoint, which in our case is the launched OpenStack resource, *b*) the desired number or requests per second, *c*) the duration for which to run the payload generator and *d*) the number of threads. The payload generator distributes the load evenly across the threads that send requests to the endpoint at the given rate. In case the endpoint cannot deliver the specified number of requests per second we suspend the thread for a random amount of time between 0 and 15 seconds. In case a thread fails for three times consecutively, we interrupt the thread altogether.

To create the observation point data set we iterate over all adaptation point attribute values. For each value we launch a new instance, wait until it is ready and then use the payload generator to send requests in the range between 0 and 3.5 requests per second with a step size of 0.1. The CA4S Client Agent on the OpenStack instance sends the observation points to the CA4S Service, where we then use it for the evaluation.

Objective Filters

To remove unsuitable observation points from the data set we use the following two objective filters.

1. We filter out all observation points where the number of requests is equal to 0. This is necessary for the bounce rate adaptation point, because the quality function is not defined for zero requests per second.

2. In our experiments we found that observation points from an overloaded instance bias the quality calculation. Therefore, we filter all observation points where the internal attribute value "CPU idle" is less than 10%.

6.2.4 Results

For this setup, we present results for two different adaptation points. First we show the result for an adaptation point with a constant quality function that switches a cache either on or off. We then extend the experiment to an adaptation point with four possible attribute values and where the quality function depends on the observation point attributes.

QuickCache Adaptation Point

"Quick Cache" is a plug-in for WordPress that provides a file system based cache for WordPress installations [81]. For this plug-in we created the adaptation point "QuickCacheTimeout" that allows us to specify the amount of time for which a cached page is valid. In our evaluation scenario the setting for QuickCacheTimeout can be either "0" (i.e. we disable the cache) or "60" (i.e. we cache the content for a period of 60 seconds). The result of the quality function is a fixed value that depends on the concrete value for the QuickCacheTimeout adaptation point attribute, as Formula 6.2 shows.

Formula 6.2: Quality for the QuickCacheTimeout Adaptation Point

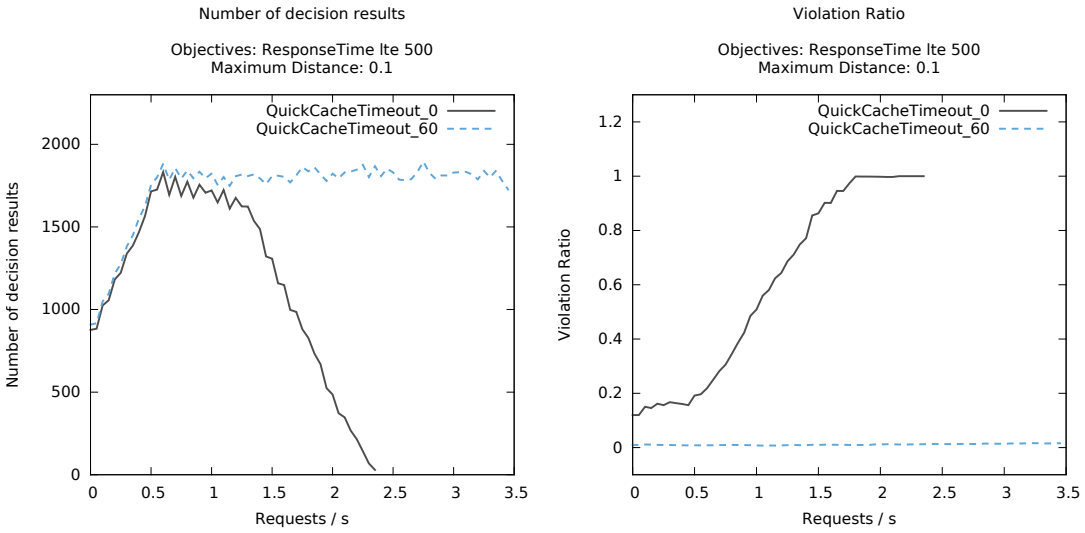
$$q = \begin{cases} 1 & \text{if } QuickCacheTimeout = 0, \\ 0.8 & \text{else.} \end{cases}$$

We used the payload generator to create an observation point data set for this evaluation scenario. From the generated observation points $2\,874$ have the cache disabled and $8\,068$ entries have a cache timeout setting of 60. This results in a total size of $10\,942$ observation points. We plot the results of this evaluation in Figure 6.2. The graph in Figure 6.2(a) shows that for the disabled cache adaptation point attribute value there are less decision results available as the number of incoming requests per seconds increases. Simultaneously, the average response time increases with an increasing number of requests. This leads to a higher violation ratio, because more and more observation points violate the specified objective of a response time below 500 ms. Consequently, the total quality score for the disabled cache drops below that of the enabled cache. In the result that Figure 6.2(d) shows, this happens at a rate of approximately 0.6 requests per second.

This result also demonstrates how the selection of the quality function influences the decision process: If we would set the quality for the enabled cache to 0.6 instead of 0.8 (as in Figure 6.2(d)), the configuration selector would enable the cache later, because the quality score for the disabled cache first had to drop to a total quality of 0.6, before the enabled cache would yield a higher quality.

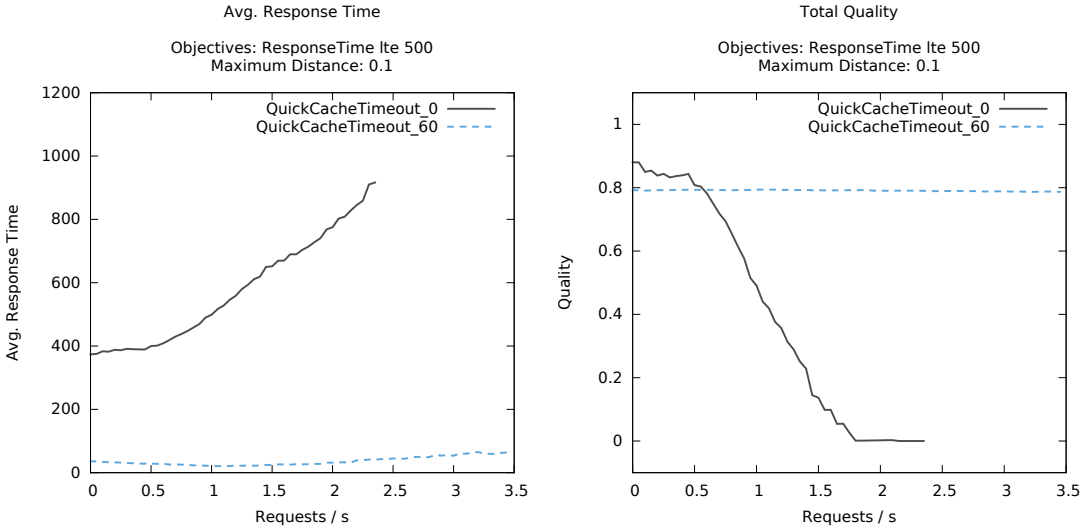
6.2.5 Bounce Rate Adaptation Point

To demonstrate that the *in situ* scenario also works for adaptation points with more than two options and where the quality depends on the observation point attribute values, we now present a second evaluation of the WordPress stack. This time we use the "BounceRate" adaptation



(a) Total number of decision results

(b) Violation ratio for a response time $\leq$ 500 ms



(c) Average response time

(d) Total quality score

Figure 6.2: The evaluation results for the adaptation point QuickCacheTimeout on the WordPress stack.

point that we already presented in Section 6.1. To enable a request bouncer for the WordPress blog, we created a script that the application calls before each request. Like in the simulator, we enumerate each incoming request and respond to every b^{th} request with the HTTP status code 500, where b denotes the bounce rate. For this evaluation we also use the payload generator to create a test set that contains a total of $17\,026$ observation points. Table 6.2 shows the distribution of the observation points.

Table 6.2: BounceRate Adaptation Point Distribution

BounceRate	**0**	**4**	**3**	**2**	**Total**
Number of entries	$4\,072$	$4\,347$	$4\,473$	$4\,134$	$17\,026$

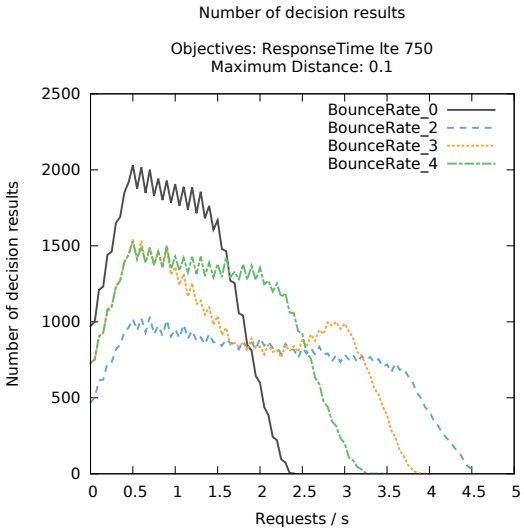
We summarize the results of this evaluation in Figure 6.3. The figure shows that at first the application accepts all incoming requests. At a rate of 1.4 requests per seconds we observe that the total quality for a bounce rate of 4 yields a higher quality than that for a bounce rate of 0. This is despite the fact that there are more observation points for a bounce rate of 0 than there are for a bounce rate of 4. This shows that the number of observation points does not influence the configuration selector.

6.3 End To End Evaluation

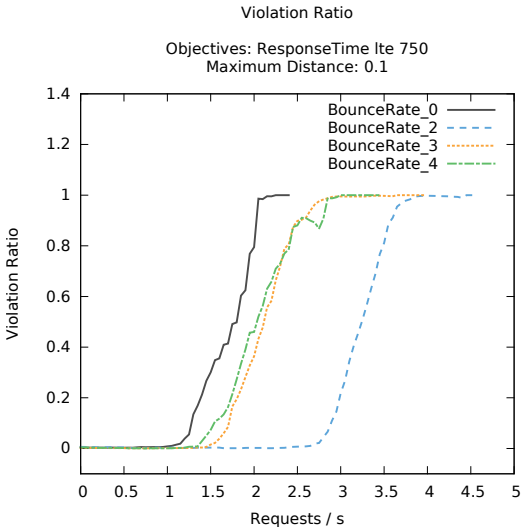
In the previous two sections we evaluated our prototype by visualizing the decision results of the configuration selector. We now conclude this chapter with an "end to end" evaluation of the CA4S Prototype and a Heat stack, where we apply the configuration decision to the instance.

6.3.1 Setup

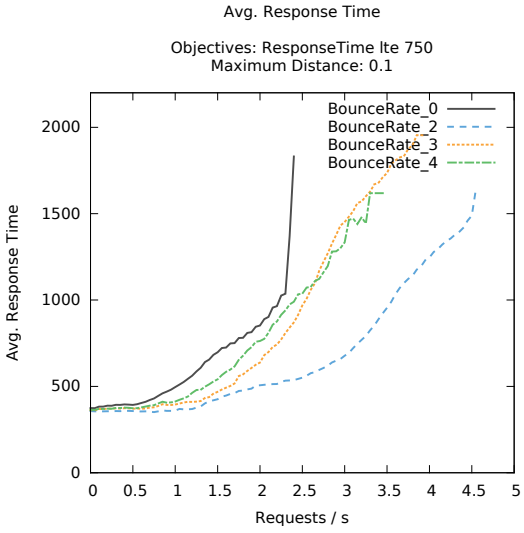
For this evaluation, we use the WordPress instance with the "BounceRate" adaptation point that we described in Section 6.2. From this experiment we also use the data set with $17\,026$ historical observation points as the initial knowledge base. In the Heat template file we specify a target response time of ≤ 750 ms as the objective and an initial value of 0 for the bounce rate. We then launch the stack and use the payload generator (see Section 6.2.3) to create an incoming request curve for the WordPress application that approximates a normal distribution over the course of elapsed time. The CA4S Client Agent records the number of requests per second that the resource actually serves along with the other attribute values and sends the observation points to the CA4S Service. The CA4S Engine constantly monitors the stack and performs the configuration changes if necessary. If an adaptation point attribute values changes, the CA4S Client Agent applies the changed configuration settings to the resource (see Figure 4.5). As a control we first run this test on a system where we disabled the CA4S Engine and then compare it to the results of the same setup with an enabled CA4S Engine.



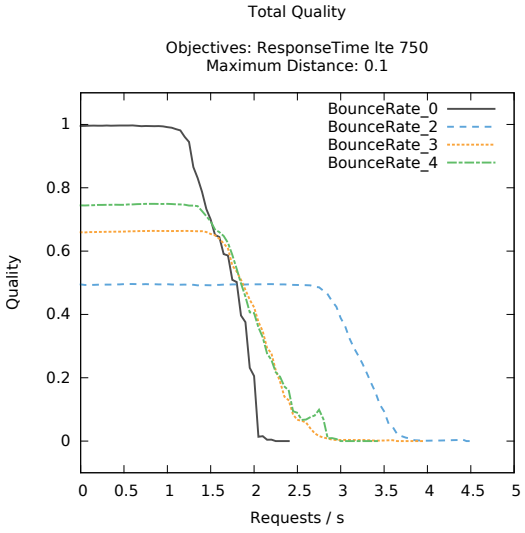
(a) Total number of decision results



(b) Violation ratio for a response time $\leq$ 750 ms



(c) Average response time



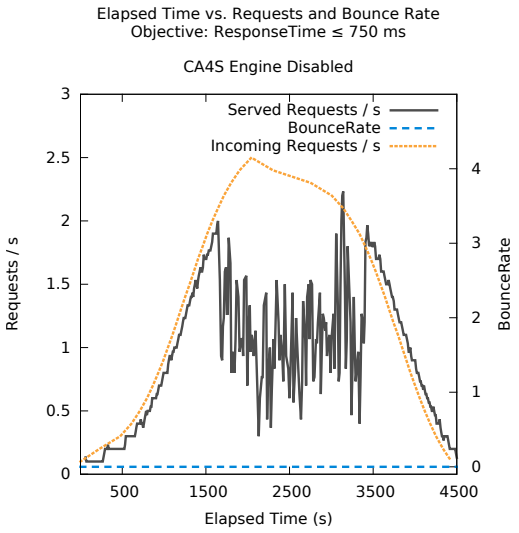
(d) Total quality score

Figure 6.3: Evaluation results for the adaptation point BounceRate on the WordPress stack.

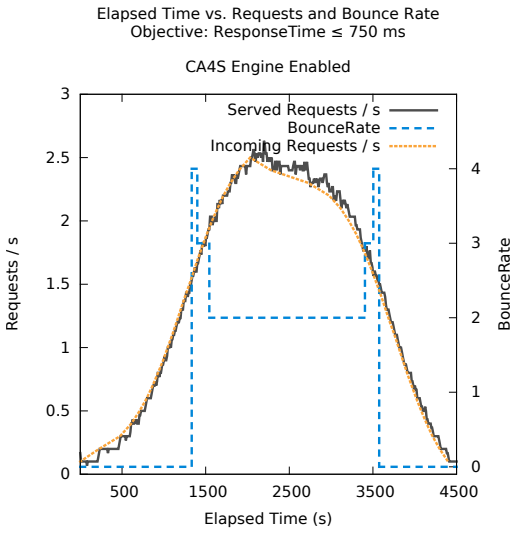
6.3.2 Result

Figure 6.4 shows the outcome of this evaluation. We show in Figures 6.4(a) and 6.4(b) the elapsed time on the x axis and put this in relation to the number of incoming requests per second as well as the served requests per second on the left y axis. The right y axis shows the value for the "BounceRate" adaptation point. This graph indicates when the CA4S Engine changes the adaptation point attribute value. Without the CA4S Engine, the WordPress application fails to deliver the desired amount of requests at a rate of approximately 2 requests per second (Figure 6.4(a). The staggered graph for the served requests/s indicates a server overload. When we enable the CA4S Engine, the configuration selector initially leaves the bounce at 0, because this setting yields the best quality. With an increasing number of incoming requests the CA4S Engine adjusts the bounce rate so that the application is able to handle the peak demand. When the number of incoming requests decreases and a different adaptation point value is able to handle the demand with a higher quality, the CA4S Engine adjusts the bounce rate. Eventually the bounce rate reaches 0 again (Figure 6.4(d)).

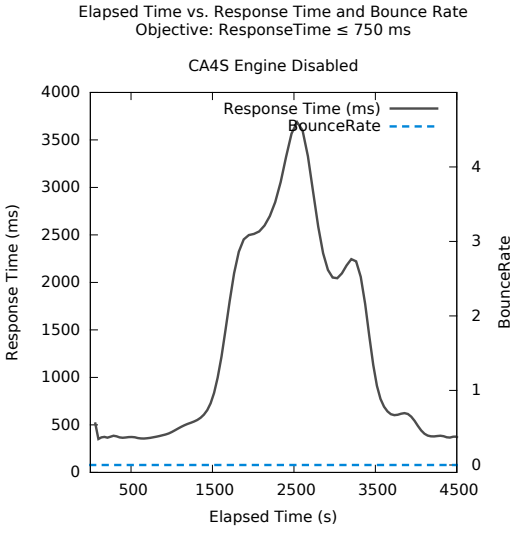
Figures 6.4(c) and 6.4(d) show a different aspect of this evaluation. Like in Figure 6.4(b), the data is in chronological order. This time we show the smoothed average response time on the left y axis. The right y axis shows the bounce rate. With a disabled CA4S Engine, the response time peaks at over $3\,500$ ms, consequently violating the 750 ms objective (Figure 6.4(c)). When we enable the CA4S Engine, the application continuously meets the specified objective, because now it adjusts the bounce rate.



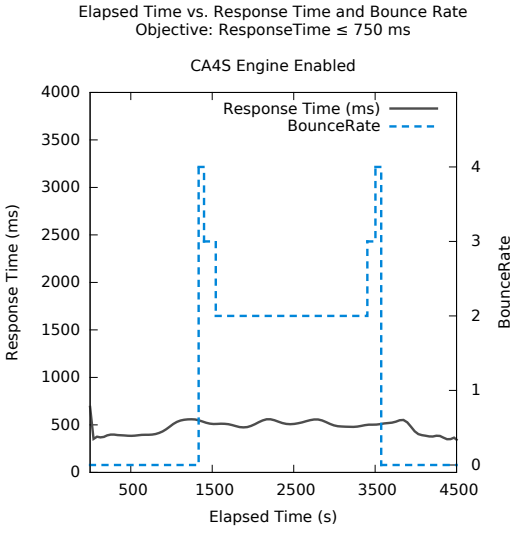
(a) Control benchmark with the disabled CA4S Engine



(b) The CA4S Engine adapts to the increased workload by adjusting the bounce rate.



(c) Control benchmark with the disabled CA4S Engine



(d) When we enable the CA4S Engine, the application meets the objective

Figure 6.4: The end to end evaluation shows the effect of the CA4S Engine.

CHAPTER 7

Conclusion

In this chapter we conclude the thesis by revisiting the research questions in Section 7.1 and by providing an outlook on future research directions in Section 7.2.

7.1 Research Questions Revisited

How can we extend existing cloud orchestration template languages so that a user can specify elasticity requirements to the application with them? In Chapter 4 we presented three novel resources for OpenStack Heat. One resource enables a user to specify the desired high-level objectives for an application. The second resource allows the user to specify the provided adaptation points. Finally, the third resource allows a user to specify the overall strategy that the application should aim for. We have shown that the presented method integrates seamlessly into the existing OpenStack Heat template structure because a user can specify the elasticity requirements with a valid CloudFormation template format.

How can we integrate a provider managed cloud computing adaptation service into an existing cloud computing service infrastructure? By implementing resource plug-ins for OpenStack Heat we did not need to modify the source code of OpenStack to integrate the new functionalities. Likewise, we designed our service in a way so that the cloud provider can add support for new adaptation points and objectives by creating plug-ins. We created a service oriented architecture with a central REST service that provides a clean interface between the autonomic adaptation service and the managed OpenStack resources.

Is it possible to utilize the collected data from all clients in a way that allows the autonomic manager to derive reasonable specific adaptations with respect to user defined objectives? In Chapter 5 we presented an algorithm that utilizes the collectively gathered data from all clients that use the same stack template to derive configuration decisions for individual clients. We have shown in Chapter 6 that this algorithm is able to derive different decisions with respect to the user formulated objectives from the same set of data.

7.2 Future Work

In this thesis we provided the basis for a cloud application adaptation service along with a sample implementation of a configuration selection algorithm. The generic nature of the presented prototype allows us to conduct future research on a range of topics.

- For our prototype, we assumed that there exists already a body of historical observation points. In future research, we systematically want to evaluate strategies how to generate an initial set of observation points. One possible research directions is to use evolutionary algorithms.
- For cases where no observation points exists, we want to create heuristics that estimate the quality of a configuration. For this we can use interpolation functions or regression functions, such as in [12].
- In the state of the art review in Chapter 3 we covered different methods to derive configuration decisions in autonomic systems. Future research can apply the presented methods (e.g. based on control theory, game theory or neural networks) to alternative configuration selection implementations.
- In a future iteration of the configuration selection algorithm we plan to take the adaptation time into account. The research of SLA prediction (e.g. [48]) and adaptation cost models (e.g. [46]) can be the basis for this research. Rao et al. also pointed out the usefulness of reinforcement learning approaches to model the time between a change and the effect [73].
- We illustrated the feasibility of the configuration selection algorithm with a distance based algorithm. For this proof of concept, we left aside any performance considerations. To address this issue in future work, we want to use heuristics to prune the search space for configurations (e.g. branch and bound as proposed in [46]). Prototype selection can be a promising approach for a faster distance calculation [31]. Also, the creation of dependency graphs like in [86] is a promising research direction.
- In this thesis, we focused on the self-optimization and self-configuration aspect of autonomic computing. Future research can address the other two pillars of autonomic computing, namely self-protection and self-healing.
- The work from Inzinger et al. that served as the basis for this thesis makes the hypothesis that a provider managed adaptation service provides benefits for both the cloud provider and the cloud user [39]. In this thesis we focused on the benefits for the cloud user. Future research can examine the benefits of the collaboratively collected observation points proposed method from the provider side.

APPENDIX A

Acronyms

API Application Programming Interface

AWS Amazon Web Services

BTU Billing Time Unit

CA4S Cloud Computing Application Adaptation as a Service

CBR Case Based Reasoning

CFN CloudFormation-compatible format

CPU Central Processing Unit

DNS Domain Name Service

DSL Domain Specific Language

ECA Event Condition Action

GB Gigabyte

GUI Graphical User Interface

HOT Heat Orchestration Template

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

IP Internet Protocol

JSON JavaScript Object Notation

KB Kilobyte

k-NN k-Nearest Neighbors

MAPE-K Monitor, Analyse, Plan, Execute, Knowledge

MB Megabyte

OOP Object Oriented Programming

OS Operating System

PaaS Platform as a Service

PC Personal Computer

QoS Quality of Service

RAM Random Access Memory

REST Representational State Transfer

RPC Remote Procedure Call

SaaS Software as a Service

SLA Service Level Agreement

SLO Service Level Objective

TCO Total Cost of Ownership

UML Unified Modeling Language

URL Uniform Resource Locator

VCPU Virtual Central Processing Unit

VM Virtual Machine

VPN Virtual Private Network

XaaS Everything as a Service

YAML YAML Ain't Markup Language

APPENDIX B

CA4S Service Endpoints

This appendix lists the endpoints of the CA4S Service.

B.1 Configuration

Data Structure

The configuration is a set of key/value pairs where the key denotes the adaptation point and the value the concrete adaptation point value.

```
{
    "key1" : "value1",
    "key2" : "value2"
}
```

Endpoints

GET /v1/config

Retrieve the current configuration for the issuer of the request, based on the client IP address. For testing purposes it is also possible to add an arbitrary request IP by appending the GET parameter `client_adder`. Returns `{}` if no configuration exists.

B.2 Objectives

Data Structure

```
{
    "id" : "",
    "name": "",
```

```
    "objective": "",
    "target" : "",
    "cmp" : "",
    "value" : "",
    "stack_id" : "",
}
```

Endpoints

DELETE /v1/objectives/{OBJECTIVE_ID}

Delete the objective with the id {OBJECTIVE_ID}.

DELETE /v1/objectives/stack/{STACK_ID}/{OBJECTIVE NAME}

Delete the objective from stack {STACK_ID} and the objective name {OBJECTIVE NAME}.

GET /v1/objectives

Get a list of all currently registered objectives.

GET /v1/objectives/{OBJECTIVE_ID}

Get objective with id {OBJECTIVE_ID}.

GET /v1/objectives/resource/{RESOURCE_ID}

Get the list of objectives for the resource with id {RESOURCE_ID}.

GET /v1/objectives/stack/{STACK_ID}

Get the list of objectives for the stack with id {STACK_ID}.

POST /v1/objectives

Add a new objective. If the objective already exists, update it.

B.3 Stacks

Data Structure

```
{
    "id": ""
}
```

Endpoints

DELETE /v1/stacks/{STACK_ID}

Delete the stack with the id `STACK_ID`.

GET /v1/stacks

Get a list of all stacks.

GET /v1/stacks/objectives/{STACK_ID}

Get the objectives that belong to the stack with id `STACK_ID`.

POST /v1/stacks

Add a new stack. If the stack already exists, update it.

B.4 Resources

Data Structure

```
{
    "id": "",
    "stack_id": "",
    "internal_ip": "",
    "resource_type": ""
}
```

Endpoints

DELETE /v1/resources/{ID}

Delete resource with id `ID`.

GET /v1/resources

Retrieve a list of all resources.

GET /v1/resources/{ID}

Get resource with id `ID`.

POST /v1/resources

Add a new resource.

B.5 Adaptation Points

Data Structure

```
{
    "id" : "",
    "name" : "",
    "adaptation_point" : "",
    "val_from" : "",
    "val_to" : "",
    "value" : "",
    "stack_id" : ""
}
```

Endpoints

DELETE /v1/aps/{ID}/{NAME}

If `NAME` is empty, delete the adaptation point with id `ID`. Otherwise delete the adaptation point from stack with id `ID` and the key `NAME`.

GET /v1/aps

Retrieve a list of all adaptation points.

GET /v1/aps/{ADAPTATION_POINT_ID}

Get adaptation point with id {ADAPTATION_POINT_ID}.

GET /v1/aps/ip/{INTERNAL_IP}

Get adaptation points that belong to the internal IP address INTERNAL_IP.

GET /v1/aps/stack/{STACK_ID}

Get adaptation points that belong to the stack with id STACK_ID.

POST /v1/aps

Register a new adaptation point.

PUT /v1/aps/{ADAPTATION POINT_ID}/{VALUE}

Update the value for adaptation point with id {ADAPTATION POINT_ID} to {VALUE}.

B.6 Observation Points

Data Structure

```
{
    id : "",
    resource_id : "",
    resource_type : "",
    internal : "",
    external : "",
    values : "",
    stack : "",
    created_date : ""
}
```

Endpoints

POST /v1/data

Add a new observation point.

B.7 Strategy

Data Structure

```
{
    "id" : "",
    "name" : "",
    "objectives" : "",
    "function" : "",
    "stack_id" : ""
}
```

Endpoints

DELETE /v1/strategies/{ID}/{NAME}

If no `name` is provided, delete the strategy with id `{ID}`. Otherwise, delete strategy from stack id `{ID}` and with the name `{NAME}`.

GET /v1/strategies

Get the list of all strategies.

GET /v1/strategies/{ID}

Get the strategy with id `{ID}`.

POST /v1/strategies

Add a new strategy.

APPENDIX C

Full Benchmark Template Listing

C.1 Base Server Template

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "Base server containing wordpress and sample data for image creation",
    "Outputs": {
        "WebsiteUrl": {
            "Value": {
                "Fn::Join": [
                    "",
                    [
                        "http://",
                        {
                            "Fn::GetAtt": [
                                "Webserver1",
                                "PublicDnsName"
                            ]
                        }
                    ]
                ]
            }
        }
    },
    "Parameters": {
        "KeyName": {
            "Description": "Name of an existing EC2 KeyPair to enable SSH access to the
                 instance",
            "Type": "String"
        }
    },
    "Resources": {
        "Webserver1": {
            "Properties": {
```

```
                "ImageId": "precise-server-cloudimg-amd64-disk1",
                "InstanceType": "m1.tiny",
                "KeyName": {
                    "Ref": "KeyName"
                },
                "UserData": {
                    "Fn::Base64": {
                        "Fn::Join": [
                            "\n",
                            [
                                "#!/bin/bash",
                                "apt-get update",
                                "export DEBIAN_FRONTEND=noninteractive",
                                "export PRIVATE_IP=‘(curl http://169.254.169.254/latest/meta
                                    -data/local-ipv4)‘",
                                "export HOSTNAME=‘(curl http://169.254.169.254/latest/meta-
                                    data/public-hostname)‘",
                                "apt-get -y install git apache2 libapache2-mod-php5 php5-
                                    mysql mysql-server python-pip python-mysqldb ifstat
                                    sysstat unzip",
                                "pip install requests",
                                "pip install dictalchemy",
                                "mysqladmin -u root password mypwd",
                                "curl -L https://raw.github.com/wp-cli/builds/gh-pages/phar/
                                    wp-cli.phar > /usr/bin/wp",
                                "chmod +x /usr/bin/wp",
                                "mkdir /var/www/wordpress",
                                "chown www-data:www-data /var/www/wordpress",
                                "sudo -u www-data wp core download --path=/var/www/wordpress
                                    ",
                                "sudo -u www-data wp core config --dbname=wp --dbuser=root
                                    --dbpass=mypwd --path=/var/www/wordpress",
                                "sudo -u www-data wp db create --path=/var/www/wordpress",
                                {
                                    "Fn::Join" : [
                                        " ",
                                        [
                                            "sudo -u www-data wp core install",
                                            "--path=/var/www/wordpress",
                                            "--url=http://${HOSTNAME}/wordpress",
                                            "--title=’Hello Benchmark’",
                                            "--admin_user=’admin’",
                                            "--admin_password=’nimba’",
                                            "--admin_email=’spam@osl.name’"
                                        ]
                                    ]
                                },
                                "sudo -u www-data wp plugin install quick-cache --path=/var/
                                    www/wordpress",
                                "sudo -u www-data wp plugin activate quick-cache --path=/var
                                    /www/wordpress",
                                "sudo -u www-data wp plugin install wordpress-importer --
                                    activate --path=/var/www/wordpress",
```

```
                                "sudo -u www-data wp option update quick_cache_options '{\"
                                    version\":\"140104\",\"crons_setup\":\"1396734524\",\"
                                    enable\":\"0\",\"debugging_enable\":\"1\",\"
                                    cache_purge_home_page_enable\":\"1\",\"
                                    cache_purge_posts_page_enable\":\"1\",\"
                                    allow_browser_cache\":\"0\",\"cache_dir\":\"/wp-content/
                                    cache\",\"cache_max_age\":\"7 days\",\"get_requests
                                    \":\"1\",\"feeds_enable\":\"0\",\"
                                    uninstall_on_deactivation\":\"0\"}' --format=json --path
                                    =/var/www/wordpress",
                                "sudo -u www-data wp eval \"\\$p=quick_cache\\plugin(); \\$p
                                    ->add_advanced_cache();\" --path=/var/www/wordpress",
                                "wget http://wpcandy.s3.amazonaws.com/resources/postsxml.zip
                                     -O /tmp/postsxml.zip",
                                "unzip /tmp/postsxml.zip -d /tmp",
                                "sudo -u www-data wp import /tmp/posts.xml --authors=create
                                    --path=/var/www/wordpress",
                                "/bin/dd if=/dev/zero of=/var/swap.1 bs=1M count=4096",
                                "/sbin/mkswap /var/swap.1",
                                "/sbin/swapon /var/swap.1",
                                "service apache2 reload"
                            ]
                        ]
                    }
                }
            },
            "Type": "AWS::EC2::Instance"
        }
    }
}
```

C.2 Wordpress Application Server Template

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "Launch a ubuntu precise64 instance with wordpress",
    "Outputs": {
        "WebsiteUrl": {
            "Value": {
                "Fn::Join": [
                    "",
                    [
                        "http://",
                        {
                            "Fn::GetAtt": [
                                "Webserver1",
                                "PublicDnsName"
                            ]
                        }
                    ]
                ]
```

```
            }
        }
    },
    "Parameters": {
        "CA4SClientPassword": {
            "Description": "The password for the CA4SClient Repo",
            "Type": "String"
        },
        "KeyName": {
            "Description": "Name of an existing EC2 KeyPair to enable SSH access to the
                 instance",
            "Type": "String"
        }
    },
    "Resources": {
        "Webserver1": {
            "Properties": {
                "ImageId": "ca4s-server-base",
                "InstanceType": "m1.tiny",
                "KeyName": {
                    "Ref": "KeyName"
                },
                "UserData": {
                    "Fn::Base64": {
                        "Fn::Join": [
                            "\n",
                            [
                                "#!/bin/bash",
                                "export PRIVATE_IP=‘(curl http://169.254.169.254/latest/meta
                                     -data/local-ipv4)‘",
                                "export HOSTNAME=‘(curl http://169.254.169.254/latest/meta-
                                     data/public-hostname)‘",
                                {
                                    "Fn::Join" : [
                                        "",
                                        [
                                            "wget http://stuff.r53.osl.name/ca4s.sh && sh
                                                 ca4s.sh ca3sclient ",
                                            { "Ref" : "CA4SClientPassword" }
                                        ]
                                    ]
                                },
                                "curl https://gist.githubusercontent.com/moee/
                                     ef50e9e5ad4613f050ae/raw/setup.php > /tmp/setup.php",
                                "php /tmp/setup.php",
                                "mv /var/www/wordpress/index.php /var/www/wordpress/index2.
                                     php",
                                "curl https://gist.githubusercontent.com/moee/
                                     ef50e9e5ad4613f050ae/raw/bouncer.php > /var/www/
                                     wordpress/index.php",
                                "/sbin/swapon /var/swap.1"
                            ]
                        ]
```

```
                    }
                }
            },
            "Type": "AWS::EC2::Instance"
        },
        "Objective_ResponseTime": {
            "Type": "CA4S::Objective",
            "Properties" : {
                "Objective" : "ResponseTime",
                "Target" :  { "Fn::Join" : ["", ["http://", { "Fn::GetAtt" : [ "Webserver1",
                      "PrivateIp" ]}, "/wordpress/"]] },
                "Cmp" : "lte",
                "Value" : 750
            }
        },
        "AP_QuickCacheTimeout" : {
            "Type": "CA4S::AdaptationPoint",
            "Properties": {
                "AdaptationPoint" : "BounceRate",
                "gte" : 0,
                "lte" : 4,
                "value" : %VALUE%
            }
        },
        "Strategy" : {
            "Type": "CA4S::Strategy",
            "Properties": {
                "Objective" : "ResponseTime",
                "Function" : "min"
            }
        }
    }
}
```

Bibliography

[1] Amazon Web Services, Inc. Amazon EC2 Spot Instances. http://aws.amazon.com/ec2/purchasing-options/spot-instances/. Accessed: 2014-08-06.

[2] Amazon Web Services, Inc. AWS CloudFormation API Reference API Version 2010-05-15. http://awsdocs.s3.amazonaws.com/AWSCloudFormation/latest/cfn-api.pdf. Accessed: 2014-08-01.

[3] Amazon Web Services, Inc. Aws cloudformation user guide api version 2010-05-15. http://awsdocs.s3.amazonaws.com/AWSCloudFormation/latest/cfn-ug.pdf. Accessed: 2014-07-13.

[4] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web services agreement specification (ws-agreement). In *Open Grid Forum*, volume 128, 2007.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[6] Raphael M. Bahati and Michael A. Bauer. Adapting to run-time changes in policies driving autonomic management. In *ICAS*, pages 88–93, 2008.

[7] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *Database Theory—ICDT'99*, pages 217–235. Springer, 1999.

[8] Joseph P. Bigus, Don A. Schlosnagle, Jeff R. Pilgrim, W Nathaniel Mills III, and Yixin Diao. Able: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371, 2002.

[9] Tobias Binz, Gerd Breiter, Frank Leymann, and Thomas Spatzier. Portable Cloud Services Using TOSCA. *IEEE Internet Computing*, 16(03):80–85, May 2012.

[10] Jan Bosch and PerOlof Bengtsson. Assessing optimal software architecture maintainability. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 168–175. IEEE, 2001.

[11] Scott Bradner. IETF RFC 2119: Key words for use in RFCs to Indicate Requirement Levels. Technical report, Internet Engineering Task Force (IETF), 1997.

[12] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. A reinforcement learning approach to online web systems auto-configuration. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 2–11. IEEE, 2009.

[13] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. *IEEE Transactions on Parallel and Distributed Systems*, 24(4):681–690, 2013.

[14] BuiltWith Pty Ltd. WordPress Usage Statistics. `http://trends.builtwith.com/cms/WordPress`. Accessed: 2014-11-03.

[15] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Comp. Syst.*, 25(6):599–616, 2009.

[16] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[17] Ceilometer System Architecture. `http://docs.openstack.org/developer/ceilometer/architecture.html#alarming`. Accessed: 2014-04-08.

[18] Clovis Chapman, Wolfgang Emmerich, F Galan Marquez, Stuart Clayman, and Alex Galis. Elastic service management in computational clouds. *CloudMan 2010*, 2010.

[19] Y Diao Chess, Joseph L Hellerstein, Sujay Parekh, and Joseph P Bigus. Managing web server performance with autotune agents. *IBM Systems Journal*, 42(1):136–149, 2003.

[20] CloudFormation Compatible Functions - heat 2014.2.dev91.g9cfe20b documentation. `http://docs.openstack.org/developer/heat/template_guide/functions.html`. Accessed: 2014-09-18.

[21] Georgiana Copil, Daniel Moldovan, Hong Linh Truong, and Schahram Dustdar. Sybl: An extensible language for controlling elasticity in cloud applications. In *CCGRID*, pages 112–119. IEEE Computer Society, 2013.

[22] IBM Corporation. An architectural blueprint for autonomic computing. *Autonomic Computing White Paper*, 2005. Third edition.

[23] DBLP. Completesearch. `http://www.dblp.org/search/#query=cloudcomputing&qp=H1.40:W1.3:F1.4:F2.4:F3.17:F4.4`. Accessed: 2014-07-14.

[24] Yixin Diao, Joseph L. Hellerstein, Sujay S. Parekh, Rean Griffith, Gail E. Kaiser, and Dan B. Phung. Self-managing systems: A control theory foundation. In *ECBS*, pages 441–448, 2005.

[25] Schahram Dustdar, Yike Guo, Rui Han, Benjamin Satzger, and Hong-Linh Truong. Programming directives for elastic computing. *IEEE internet computing*, 16(6):72–77, 2012.

[26] Schahram Dustdar, Yike Guo, Benjamin Satzger, and Hong Linh Truong. Principles of elastic processes. *IEEE Internet Computing*, 15(5):66–71, 2011.

[27] Vincent C Emeakaroha, Ivona Brandic, Michael Maurer, and Schahram Dustdar. Low level metrics to high level slas-lom2his framework: Bridging the gap between monitored metrics and sla parameters in cloud environments. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 48–54. IEEE, 2010.

[28] R Fielding, J Gettys, J Mogul, H Frystyk, L Masinter, P Leach, and T Berners-Lee. Rfc 2616. *Hypertext Transfer Protocol–HTTP/1.1*, 2(1):2–2, 1999.

[29] FoSII - Foundations of Self-Governing ICT Infrastructures. `http://www.infosys.tuwien.ac.at/linksites/fosii/`.

[30] Alan G Ganek and Thomas A Corbi. The dawning of the autonomic computing era. *IBM systems Journal*, 42(1):5–18, 2003.

[31] Salvador Garcia, Joaquín Derrac, José Ramón Cano, and Francisco Herrera. Prototype selection for nearest neighbor classification: Taxonomy and empirical study. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(3):417–435, 2012.

[32] Jesús García-Galán, Liliana Pasquale, Pablo Trinidad, and Antonio Ruiz Cortés. User-centric adaptation of multi-tenant services: preference-based analysis for service reconfiguration. In *SEAMS*, pages 65–74, 2014.

[33] Yanfei Guo, Palden Lama, and Xiaobo Zhou. Automated and agile server parameter tuning with learning and control. In *IPDPS*, pages 656–667, 2012.

[34] Jens Gustedt, Emmanuel Jeannot, and Martin Quinson. Experimental methodologies for large-scale systems: a survey. *Parallel Processing Letters*, 19(03):399–418, 2009.

[35] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 2013), San Jose, CA*, 2013.

[36] David Hilley. Cloud computing: A taxonomy of platform and infrastructure-level offerings. *Georgia Institute of Technology, Tech. Rep. GIT-CERCS-09-13*, 2009.

[37] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40(3), 2008.

[38] James Hunt and Clint Byrum. Upstart intro, cookbook and best practises. `http://upstart.ubuntu.com/cookbook/`. Accessed: 2014-07-18.

[39] Christian Inzinger, Benjamin Satzger, Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. Model-based Adaptation of Cloud Computing Applications. In Slimane Hammoudi, Luís Ferreira Pires, Joaquim Filipe, and Rui César das Neves, editors, *MODELSWARD*, pages 351–355. SciTePress, 2013.

[40] Kevin Jackson and Cody Bunch. *OpenStack Cloud Computing Cookbook Second Edition*. Packt Publishing Ltd, 2013.

[41] Simon Josefsson. The base16, base32, and base64 data encodings. RFC 3548, July 2003.

[42] Christos T Karamanolis, Magnus Karlsson, and Xiaoyun Zhu. Designing controllable computer systems. In *HotOS*, 2005.

[43] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[44] Emre Kiciman and Yi-Min Wang. Discovering correctness constraints for self-management of system configuration. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 28–35. IEEE, 2004.

[45] Jonathan LaCour. Pecan. `http://www.pecanpy.org/`. Accessed: 2014-07-18.

[46] Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. Cost-based optimization of service compositions. *Services Computing, IEEE Transactions on*, 6(2):239–251, 2013.

[47] Philipp Leitner, Waldemar Hummer, Benjamin Satzger, Christian Inzinger, and Schahram Dustdar. Cost-efficient and application sla-aware client side request scheduling in an infrastructure-as-a-service cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 213–220. IEEE, 2012.

[48] Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. Monitoring, prediction and prevention of sla violations in composite services. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 369–376. IEEE, 2010.

[49] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P King, and Richard Franck. Web service level agreement (wsla) language specification. *IBM Corporation*, pages 815–824, 2003.

[50] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. Cloud computing—the business perspective. *Decision Support Systems*, 51(1):176–189, 2011.

[51] Patrick Martin, Andrew Brown, Wendy Powley, and José Luis Vázquez-Poletti. Autonomic management of elastic services in the cloud. In *ISCC*, pages 135–140. IEEE, 2011.

[52] Michael Maurer, Ivona Brandic, and Rizos Sakellariou. Simulating autonomic sla enactment in clouds using case based reasoning. In *Towards a Service-Based Internet*, pages 25–36. Springer, 2010.

[53] Michael Maurer, Ivona Brandic, and Rizos Sakellariou. Enacting slas in clouds using rules. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par (1)*, volume 6852 of *Lecture Notes in Computer Science*, pages 455–466. Springer, 2011.

[54] Michael Maurer, Ivona Brandic, and Rizos Sakellariou. Self-adaptive and resource-efficient sla enactment for cloud computing infrastructures. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 368–375. IEEE, 2012.

[55] Daniel Moldovan, Georgiana Copil, Hong-Linh Truong, and Schahram Dustdar. Mela: Monitoring and analyzing elasticity of cloud services. In *CloudCom*. IEEE, 2013.

[56] Daniel Morán, Luis Miguel Vaquero, and Fermín Galán. Elastically ruling the cloud: Specifying application's behavior in federated clouds. In Ling Liu and Manish Parashar, editors, *IEEE CLOUD*, pages 89–96. IEEE, 2011.

[57] OpenStack Heat Templates GitHub Repository. `https://github.com/openstack/heat-templates`. Accessed: 2014-08-01.

[58] OpenStack Docs. Ceilometer Developer Documentation. `http://docs.openstack.org/developer/ceilometer/`. Accessed: 2014-03-18.

[59] OpenStack Docs. Heat Resource Plug-in Development Guide. `http://docs.openstack.org/developer/heat/pluginguide.html`. Accessed: 2014-09-8.

[60] OpenStack Docs. Load-Balancer-as-a-Service (LBaaS) - OpenStack Networking API v2.0 (neutron) Reference. `http://docs.openstack.org/api/openstack-network/2.0/content/lbaas_ext.html`. Accessed: 2014-09-18.

[61] OpenStack Foundation. Openstack cloud administrator guide. `http://docs.openstack.org/admin-guide-cloud/admin-guide-cloud.pdf`. Revision Date September 27, 2014, Accessed: 2014-09-27.

[62] OpenStack Foundation. Openstack end user guide. `http://docs.openstack.org/user-guide/user-guide.pdf`. Revision Date May 9, 2014, Accessed: 2014-07-15.

[63] OpenStack Foundation. Openstack virtual machine image guide. `http://docs.openstack.org/image-guide/image-guide.pdf`. Revision Date October 15, 2014, Accessed: 2014-10-21.

[64] OpenStack Website. Companies Supporting The OpenStack Foundation. `https://www.openstack.org/foundation/companies/`. Accessed: 2014-03-18.

[65] OpenStack Website. Home Page. `http://www.openstack.org/`. Accessed: 2014-03-18.

[66] OpenStack Website. User Stories. `http://www.openstack.org/user-stories/`. Accessed: 2014-10-18.

[67] OpenStack Wiki. Heat. `https://wiki.openstack.org/wiki/Heat`. Accessed: 2014-11-30.

[68] David Oppenheimer, Archana Ganapathi, and David A Patterson. Why do internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, volume 67. Seattle, WA, 2003.

[69] Pradeep Padala, Kai-Yuan Hou, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26. ACM, 2009.

[70] Mike P Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12. IEEE, 2003.

[71] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[72] proc(5) - linux man page. Retrieved from `http://man7.org/linux/man-pages/man5/proc.5.html`. Accessed: 2014-07-18.

[73] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. Vconf: a reinforcement learning approach to virtual machines auto-configuration. In *Proceedings of the 6th international conference on Autonomic computing*, pages 137–146. ACM, 2009.

[74] Luis Rodero-Merino, Luis Miguel Vaquero Gonzalez, Victor Gil, Fermín Galán, Javier Fontán, Rubén S. Montero, and Ignacio Martín Llorente. From infrastructure delivery to service management in clouds. pages 1226–1240, 2010.

[75] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14:1–14:42, May 2009.

[76] Chrystalla Sofokleous, Nicholas Loulloudes, Demetris Trihinas, George Pallis, and MariosD. Dikaiakos. c-eclipse: An open-source management framework for cloud applications. In Fernando Silva, Inês Dutra, and Vítor Santos Costa, editors, *Euro-Par 2014 Parallel Processing*, volume 8632 of *Lecture Notes in Computer Science*, pages 38–49. Springer International Publishing, 2014.

[77] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT Press, 1998.

[78] Hong-Linh Truong, Schahram Dustdar, Georgiana Copil, Alessio Gambi, Waldemar Hummer, Duc-Hung Le, and Daniel Moldovan. Comot–a platform-as-a-service for elasticity in the cloud. In *IC2E*. IEEE, 2014.

[79] Helen J Wang, John C Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI*, volume 4, pages 245–257, 2004.

[80] Henry Ware and Fabian Frédérick. vmstat(8) - linux man page. Retrieved from `http://linux.die.net/man/8/vmstat`. Accessed: 2014-07-18.

[81] WordPress Plugin Directory. Quick cache. `https://wordpress.org/plugins/quick-cache/`. Accessed: 2014-11-01.

[82] Fetahi Wuhib, Rolf Stadler, and Hans Lindgren. Dynamic resource allocation with management objectives—implementation for an openstack cloud. In *Network and service management (cnsm), 2012 8th international conference and 2012 workshop on systems virtualiztion management (svm)*, pages 309–315. IEEE, 2012.

[83] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 159–172. ACM, 2011.

[84] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 687–700. ACM, 2014.

[85] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[86] Wei Zheng, Ricardo Bianchini, and Thu D Nguyen. Automatic configuration of internet services. *ACM SIGOPS Operating Systems Review*, 41(3):219–229, 2007.