Architectural Operations in Cloud Computing

Ragnar Skúlason

ARCHITECTURAL OPERATIONS IN CLOUD COMPUTING

Ragnar Skúlason

60 ECTS thesis submitted in partial fulfillment of a
Magister Scientiarum degree in Software Engineering

Advisors
Klaus Marius Hansen
Helmut Wolfram Neukirchen

Faculty Representative
Hjálmtýr Hafsteinsson

Faculty of Faculty of Industrial Engineering, Mechanical Engineering and
Computer Science
School of Engineering and Natural Sciences
University of Iceland
Reykjavik, August 2011

Architectural Operations in Cloud Computing
Cloud ASL
60 ECTS thesis submitted in partial fulfillment of a MSc degree in Software Engineering

Faculty of Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Hjarðarhagi 2-6
107, Reykjavik, Reykjavik
Iceland

Telephone: 525 4000

Bibliographic information:
Ragnar Skúlason, 2011, Architectural Operations in Cloud Computing, MSc thesis, Faculty of Faculty of Industrial Engineering, Mechanical Engineering and Computer Science, University of Iceland.

Printing: Háskólaprent, Fálkagata 2, 107 Reykjavík
Reykjavik, Iceland, August 2011

Dedicated to my family
Life's never dull with you!

Abstract

Rapid scalability is important in cloud computing in order to serve growing communities and optimize hardware costs. This scalability can be hard to achieve, especially in software with static architecture. Changing software architecture of running systems on multiple devices over the Internet is a hard and delicate process as updating live software can cause faults and failures while software systems are being restarted. Taking the study of software architecture to the dynamics of the cloud computing can be beneficial in this case and increase cloud computing possibilities.

The Architectural Scripting Language (ASL) is a language for expressing the dynamic aspect of run-time and deployment-time software architecture. In the following thesis, ASL is taken to cloud computing which enables dynamic software architecture changes to meet the dynamics of a computing infrastructure. We present Cloud ASL, which is an external domain-specific language which enables architectural operations and architectural scripting in cloud computing environments. Cloud ASL is modeled and tested by the creation of a distributed cloud computing ray tracing system which was built to utilize Cloud ASL for its distributed and cloud computing mechanism.

Cloud ASL is a framework which enables modelling dynamic aspects of runtime software architecture with architectural operations in cloud computing and suitable to use for creating a scalable and modifiable cloud computing software.

Preface

I have been interested in software development and in particular web development and distributed development for a decade. I spent the academic year 2009-2010 as an exchange student at the University of California, Berkeley, and by coincidence I ended up in a cloud computing course. After attending this course I came very interested in the subject, so much that when I arrived back in Iceland to do my MSc thesis I started by looking for possible cloud computing based projects. After discussing project ideas with professor Klaus Marius Hansen we settled with this very interesting research topic, which joined cloud computing and software architecture.

Contents

List of Figures

List of Tables

List of Listings

Acronyms and Abbreviation

AD Architectural Description

ADD Attribute-Driven Design

ADL Architecture Description Language

API Application Programming Interface

AQS Architectural Quality Scenarios

ASL Architectural Scripting Language

ATAM Architecture Trade-off Analysis Method

AWS Amazon Web Services

CPU Central Processing Unit

DSL Domain Specific Language

EBS Elastic Block Storage

EC2 Elastic Cloud Computing

GPU Graphics processing unit

IaaS Infrastructure as a Service

JVM Java Virtual Machine

PaaS Platform as a Service

QAS Quality Attribute Scenario

SAAM Software Architecture Analysis Method

SaaS Software as a Service

SDK Software Development Kit

UI User Interface

UML Unified Modelling Language

VM Virtual Machine

Acknowledgements

I would first of all like to thank my advisor, professor Klaus Marius Hansen, for the great guidance he provided me with and what often seemed like unlimited knowledge in our fields of study. I would like to thank my second advisor, associate professor Helmut Wolfram Neukirchen, for joining us in the later period of the work and managing the last parts of the thesis defense.

I also want to thank the University of Iceland Research Fund for allowing me to work on this project full time, I want to thank Amazon Web Services for giving me funds to use their public cloud, Reiknisstofnun Háskóla Íslands for a valid try to give us access to cloud computing services through their hardware, and GreenQloud for giving us access to their available infrastructure.

Lastly I want to thank all the faculty of the computer science department of University of Iceland for their dedication to the science.

Thanks
Ragnar Skulason

1. Introduction

In the following chapter the subject of the thesis will be introduced. First, I motivate the work by describing some of the problems treated in the thesis. Secondly, the challenges will be summarized in a problem statement and finally the attempts at solving them will briefly be outlined.

It is assumed that the reader of this thesis has a background in computer science and possesses general knowledge of distributed systems and software architecture; some knowledge of architecture operations, cloud computing, the Java OSGi framework; and computer graphics.

1.1. Motivation

Cloud computing is a new and emerging technology where technical infrastructure is provided as a "utility". In this way, users of clouds (software developers) can use virtualized resources as a service, often flexibly scaling resource usage (and payment) up or down. One of the cloud computing service layers, Infrastructure as a Service (IaaS), gives developers freedom to develop their own platform and use their software as they would do on their own infrastructure, but this comes with a cost: managing software architecture on a big cloud at runtime can be a difficult and delicate task, as one small error in a deployment script or an architectural change can result in serious faults resulting in denial of service or other severe failures as a consequence.

The work of this thesis is done as an attempt to solve this problem and ease deployment and architectural change. Enabling software developers to manage software architecture dynamically on top of a cloud computing infrastructure can simplify deployment processes significantly. This can give software developers a useful tool to update software and scale software systems which will benefit cloud computing development. Building complex scripts or programs to deploy and update software on clouds would be replaced by simple architecture scripts that can be reviewed and tested.

1.2. Problem Statement

With the scalability of cloud computing infrastructure, scalable software architecture and fault tolerance is equally important in the form of architectural change.

Changing software architecture can be important for scalable software, as for any high available software. Any change in the software architecture can be a difficult task and would require shutting down old software components and starting new software components, resulting in software downtime. Architectural scripting could be a solution to this problem. It could allow architectural modification on runtime software, resulting in minimum downtime in general support for architectural change could be available in a cloud computing environment, enabling dynamic scaling of cloud computing software architectures.

In this thesis, we argue that modeling dynamic aspects of runtime software architecture with architectural scripting complements cloud computing tools/techniques; especially Infrastructure as a Service (IaaS). Architectural scripting can be the basis of managing and modeling the dynamic aspects of architectural change.

The Architectural Scripting Language (ASL) has been the focus of two studies, Hydra [8] and Alloy [9] but has not been implemented in other cases. Therefore the question arises: Can architectural scripting be tailored for cloud computing infrastructure? And if so, how would we implement architectural scripting on cloud computing infrastructure?

1.3. Thesis Outline

The theoretical and historical background of the concepts this thesis is based on will be described in chapter 2. Chapter 3 demonstrates the implementation of our approach and the process of building it. In chapter 4, the work will be evaluated. Chapter 5 will discuss our work and in the last chapter and we will summarize the work we did for this thesis.

2. Background

The work in this thesis can be categorized into software architecture and distributed computing. In this chapter, we will present related work in these areas, mainly cloud computing, software architecture and its subset architectural operations. Through the rest of the thesis, the contents of this chapter will be used as input to design and to govern discussions.

2.1. Cloud Computing

'Cloud computing' is a relatively new concept in the computing world, although the idea has existed for a longer time. A few years after the dot-com bubble, companies like Amazon started leasing out their underutilized and unused hardware with cloud computing technology, resulting in cloud computing gaining attention and popularity within the computing industry. Cloud computing has become a viable option in recent years for several reasons. The "web 2.0" shift can be named as an example, as providers are shifting their services from localy stored and hosted services to external services. A few years ago a web company would have needed to host and maintain its own billing system and payment gateway, making long-term and expensive contracts with credit card companies, banks, security companies, etc. With the emergence of companies like PayPal and Chargify, any individual can now accept credit cards without a contract or long-term commitment and use the services on a pay-as-you-go basis. As the Internet has become a part of almost any household and is viewed as a commodity everyone has access to, companies can move their software services more securely to the internet, and by that make cloud computing platforms a possible choice. The economy of scale is greatly in favor of cloud computing. Cloud users can lease computing power from anywhere and cloud providers can cut their prices by investing in huge data centers and potentially save on each server, which makes this service interesting to users, such as small and medium-sized companies. Table 2.1 shows the cost difference in medium vs. very large data centers. Companies also have valid reasons to offer cloud computing services. The big computing companies have similar financial incentives as users, as they can buy and operate computing instances at a fraction of the price small or medium-sized companies do and resell them at higher costs. Vendors like Microsoft also need to defend their franchise by offering cloud computing services on their platform to leverage their users

	Cost in medium sized data center	Cost in Very large data center	ratio
Network	$95/Mbps	$13/Mbps	7.1
Storage	$26.00/GB/year	$4.6/GB/year	5.7
Administration	140 servers/admin	> 1000 servers/admin	7.1

Table 2.1.: Economy of scale in 2006 for medium-size data center vs. very large data center [5, 6]

Price per KWH	Location	Reasons
3.6¢	Idaho	Use of local hydroelectric power
4.0¢	Iceland	Use of hydroelectric and geothermal
10.0¢	California	Strict laws on environmental power generation, electricity sent in long distance
18.0¢	Hawaii	No local electricity source, fuel must be shipped

Table 2.2.: Price of electricity by region [5, 7]

to stay within their franchise [10].

The concept "utility computing", which cloud computing is in many ways based on, has been the vision of computer scientists for decades [11]. In the 1960s, the American computer scientist John McCarthy stated that

> "If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility... The computer utility could become the basis of a new and important industry" [12]

Utility computing is based on the concept of computing resources as a utility, just as water, gas and electricity are. One pays for one's computing resources (CPU, data storage, data transfer etc.) while using them. Like tap water, the user turns on his computing resources and when finished using them, turns them off and only pay for the amount used. This type of computing service has not been available until a few years ago. One of the factors that influences companies to become cloud computing infrastructure providers is the accessibility of underutilized computing resources [10], and for Amazon their web service (AWS) started for their internal operations [13]. The on-line book retailer Amazon is a good example of a cloud service provider. Amazon needs to be able to provide hardware for its peak time usage, which is only fully used a few times per year. On a regular basis the hardware is sitting at a very low utilization, or on average at 10% [14] of its capacity, to leave room for occasional spikes. What Amazon did to use their underutilized hardware was start leasing their computing power to users on an hourly basis. Users are now able to buy a virtualized computing instance, with their software and op-

erating system of choice, and be billed by the hour [14, 15] as result.

Cloud computing has been a buzzword in the computing industry recently and has been gaining a lot of atraction, see figure 2.1.

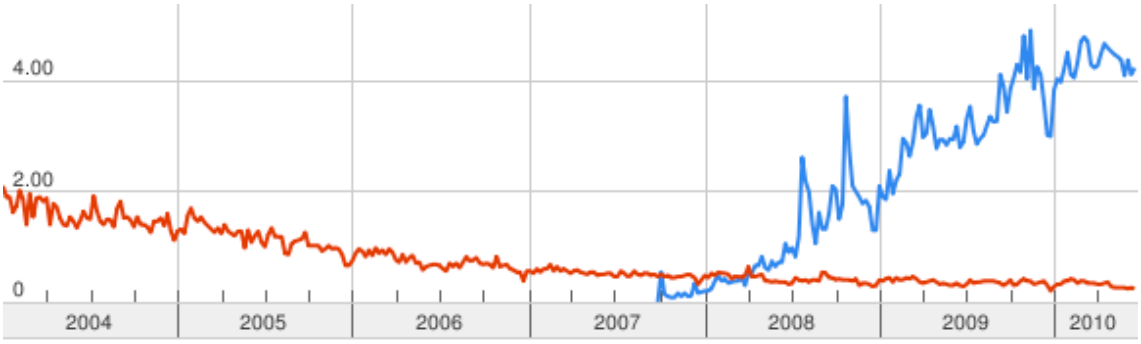

Figure 2.1.: Cloud computing (blue) vs. grid computing (red) trends[1]

The term "cloud computing" is a recent concept and it therefore has no single definition that has been accepted by cloud computing users. However, there are a few key principles that are generally accepted as central to cloud computing, and the differences between definitions is usually not great. In this thesis we are going to use a definition made by the UC Berkeley RAD Lab [10] which states:

> "Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services. The services themselves have long been referred to as Software as a Service (SaaS), so we use that term. The data center hardware and software is what we will call a Cloud. When a Cloud is made available in a pay-as-you-go manner to the public, we call it a Public Cloud; the service being sold is Utility Computing. We use the term Private Cloud to refer to internal data centers of a business or other organization that are not made available to the public. Thus, Cloud Computing is the sum of SaaS and Utility Computing, but does not normally include Private Clouds."

and the definition from the National Institute of Standards and Technology (NIST) [16]:

> "Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interac-

[1] Google Trends, http://trends.google.com, accessed: 29. July 2010

> tion. This cloud model promotes availability and is composed of five essential characteristics, three service models, and four deployment models."

These two definitions complement each other and can be used jointly or separately. Whereas the UC Berkeley definition defines the cloud computing essentials, the NIST definition defines key elements of cloud computing, or five characteristics, three service models and four deployment models. These elements will be reviewed below.

2.1.1. Essential Characteristics

In the dawning of cloud computing its definition was disputed and many different computing services defined themselves as cloud computing services, on September 25, 2008 Larry Ellison, CEO of Oracle, argued [17]:

> "The interesting thing about cloud computing is that we've redefined cloud computing to include everything that we already do. I can't think of anything that isn't cloud computing with all of these announcements. The computer industry is the only industry that is more fashion-driven than women's fashion. Maybe I'm an idiot, but I have no idea what anyone is talking about. What is it? It's complete gibberish. It's insane. When is this idiocy going to stop?"
>
> We'll make cloud computing announcements. I'm not going to fight this thing. But I don't understand what we would do differently in the light of cloud computing other than change the wording of some of our ads. That's my view.

The NIST definition of Essential Characteristics of Cloud Computing lists those characteristics that are required of a service to make it qualify as true "Cloud Computing", in other words, if it doesn't do this, it isn't Cloud Computing:

On-demand self-service
: Just like electricity, a consumer can provision computing power on-demand such as computing instances, networking or storage, without human interaction.

Broad network access
: Cloud computing is network based and accessible from anywhere by any standard platform, thin or thick clients (for example desktop computers, mobile phones or PDAs).

Resource pooling
: Resources are shared within the cloud. This means that numerous clients may be using the same set of resources at the same time, and that clients have no control

or knowledge of exact location or details of provided resources.

Rapid elasticity

Resources can be rapidly end elastically provisioned, giving clients opportunities to quickly scale up or scale down. To the consumer, the resources seem unlimited and can be purchased in any quantity at a time.

Measured Service

The cloud provider acts like any utility provider who measures and bills the amount of service provided.

2.1.2. Service Models

The real highlight of cloud computing is its versatility and adaptability. One can categorize the service provided by cloud computing in three classes, Infrastructure as a Service, Platform as a Service and Software as a Service where each service category can be leveraged independently or consumed in combination with other service tiers:

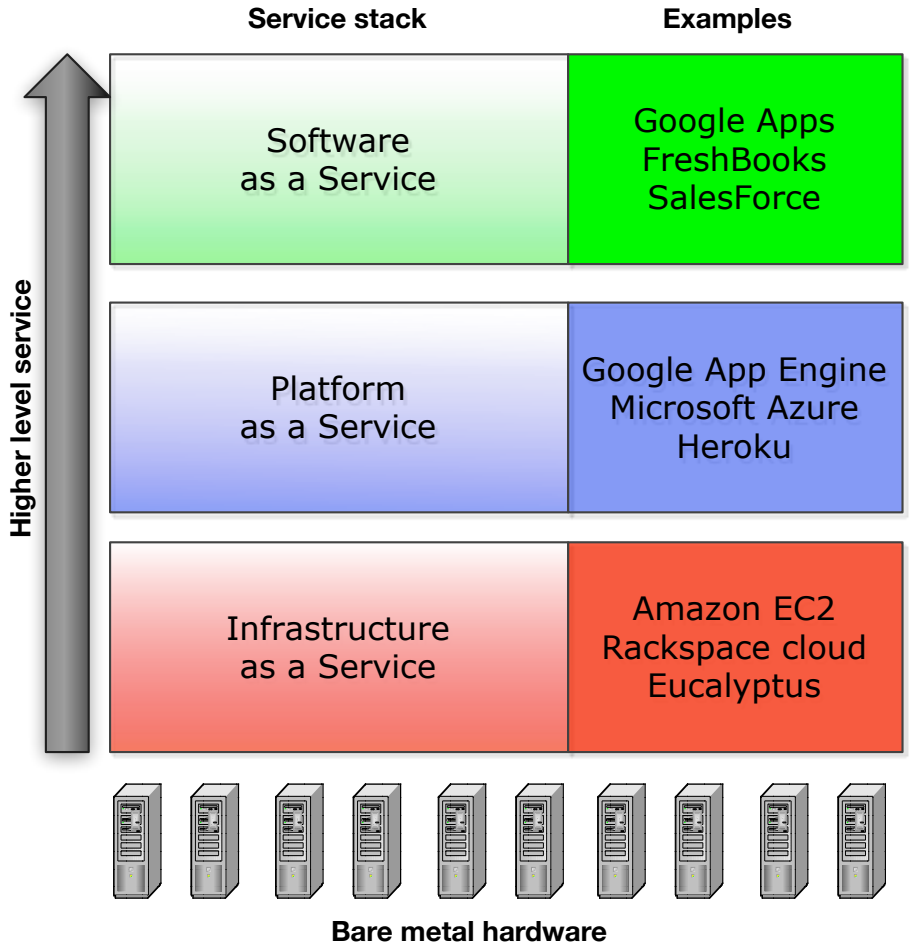

Figure 2.2.: IaaS vs. PaaS vs. SaaS

Infrastructure as a Service

Infrastructure as a Service (IaaS) is one of the service models that cloud computing is based on. IaaS delivers computer infrastructure to clients as a service, usually in the form of a virtualized platform. Users can lease server- or networking hardware as a fully outsourced service instead of purchasing it. Examples of IaaS services are Amazon EC2[2], Rackspace Cloud[3] and the Eucalyptus Private cloud software infrastructure [18]. The Amazon IaaS will be described in more detail as it is the service we ended up using and it is one of the leading cloud computing service providers on the market, therefore it is the model many cloud providers follow.

Amazon offers multiple cloud computing services. Examples are Relational Database

[2]Amazon Elastic Cloud Computing. http://aws.amazon.com/ec2
[3]http://www.rackspacecloud.com

Instance Name	CPUs	Memory	Storage	Platform	Price $/hour
Small	1	1.7 GB	160 GB	32 bit	0.095
Large	4	7.5 GB	850 GB	64 bit	0.38
Extra Large	8	15 GB	1690 GB	64 bit	0.76

Table 2.3.: Amazon EC2 standard prices[7]

Services, Simple Queuing Services, Mechanical Turk, CloudFront and EC2[4]. All these services, except EC2 can be though as SaaS or PaaS, see below, but Amazon EC2 is their main IaaS service. With Amazon EC2, a user can lease a virtualized computing instance for any time period, from minutes to years. These computing instances are virtual machines powered by the XEN hypervisor[5] and bundled with a customized operating system and software. The operating system can be anything supported by Amazon, but the operating system's kernel needs to interact with the XEN hypervisor. Therefore there are limited amount of kernels available, but there are many pre-installed software bundles available, both open source and commercial. Users can select multiple sizes of computing instances, from 1 core to 33.5 EC2 CPU cores, 613 MB to 68.4 GB virtual memory, up to 1TB per-volume hard drive storage and both 32 bit and 64 bit platforms in preselected instance types[6]. When users request an instance, they first need to select the size of instance needed, then machine image (operating system and bundled software), next a public/private key-pair to access the virtual machine and lastly the security group which defines allowed firewall rules. They then get a public IP address and DNS name from which they can access the instance. This gives them administrator rights to that virtual computer which they can use to install any software on and use in any way they prefer. When users have finished using the computer instance, they can terminate it and only pay for the amount of time used. All data stored on the instance will be destroyed unless copied to more permanent storage.

Amazon's EC2 includes three standard instances, which include different amounts of CPU memory and are priced differently, see table 2.3. On these instances the client can choose from multiple virtual images to be pre-installed. These images include an operating system (e.g. with Debian-based Linux, RedHat-based Linux and Windows) with different types of software pre-installed (e.g. database, batch processing, or web hosting software). The user can even create a customized virtual image which can be installed on these instances.
An example of an IaaS service user is GoGoYoKo[8], a new Icelandic on-line music store which sells and streams music in digital audio format and allows users to

[4]See http://aws.amazon.com/ for details about each service
[5]The XEN hypervisor is hardware virtualization layer created by the University of Cambridge Computer Laboratory and licensed under the GNU General Public License. http://www.xen.org
[6]See http://aws.amazon.com/ec2/instance-types/, accessed 12.04.2011
[7]Amazon EC2 quotas, http://aws.amazon.com/ec2/pricing/, accessed: 10. June 2010
[8]gogoyoko music store - Fair Play in Music. http://www.gogoyoko.com/

listen to music on-line as well as being a social network site. This site is entirely hosted on Amazon's EC2. What is gained by using an IaaS service for this kind of company are low costs of entrance in terms of hardware, rapid scaling, a reduced number of initial employee and simplified operations. When startup companies use cloud computing initially they do not need to invest initial capital in estimated future hardware requirements. Instead they set up their environment and services on few small instances, and when they are ready to go public they simply increase the running instances or the number of instances. This way the company can use the capital to make their services better and have a better service to provide when launched. If the company does not need scalability in their computing environment and the computing requirements are quite stable, then cloud computing would probably not be a financially viable option.

Most startup companies do not gain great popularity on day one, instead it takes time to gain publicity. At some periods of times they can get very high traction in a short time, for example if their website is published in the news or gets good publicity on social networking sites. When this happens, a company will receive a huge usage spike for a short period of time. If it is using a cloud computing environment, it can increase its computing power almost instantly and when the computing load reduces, they simply release some of the computing power. In this way a company does not need to have all the hardware required to handle this kind of spike, and would not lose possible customers because of a lack of service if its hardware could not handle the traffic.

For most new companies it can be very hard and bothersome to recruit capable employees. However, by not hosting their own hardware and all computing related interactions being done through the web, they can reduce the number of administrators needed which can ease some of the start-up human resources problems.

Companies can become constrained by their data centers. Let us take as an example a telecommunication company (telco) that starts up in a certain location. This company uses regular hardware and sets up its data-center at its starting location. At a future point in time the telco can have increased in size enough so that its original location can not hold its operation anymore. It will need to move to a larger location and will face a hardware problem. Moving a live and operating data-center can be extremely expensive and might not even be possible in some cases. This can scatter the company and its administration into multiple locations and make regular daily operation harder than necessary. This problem can be solved by using cloud computing.

Platform as a Service

Platform as a Service (PaaS) is another service model of cloud computing. PaaS is a layer above IaaS in figure 2.2. PaaS delivers a computing platform and/or solution stack as a service and is often consumed by the IaaS layer and can consume the

Resource	*Unit*	*Unit cost*	*Free limit*
Outgoing Bandwidth	gigabytes	$0.12	1 GB per day
Incoming Bandwidth	gigabytes	$0.10	1 GB per day
CPU Time	CPU hours	$0.10	6.5 CPU hour per day
Stored Data	gigabytes per month	$0.15	1 GB
Recipients Emailed	recipients	$0.0001	2000 Emails per day

Table 2.4.: Google App Engine pricing[12]

SaaS layer (see below). Compared to IaaS, a virtual machine as a service, PaaS can be viewed as programming language environment as a service, e.g. Java Virtual Machine as a service.

Examples of PaaS service models are Google App Engine[9], which supports the Java and Python programming languages; Microsoft Azure[10], which supports the .Net programming framework and Heroku[11], which supports the Ruby programming language and the Rails framework.

The Google App Engine (GAE) service supports Java and Python and it virtualizes applications across multiple servers and data centers. GAE only supports Google-specific data storage and database engines. Like previously stated, programs can be written in Java, or other JVM languages such as Groovy, JRuby, Scala, Clojure, Jython, a special version of Quercus, and in Python with Python web frameworks that run on the Google App Engine such as Django, CherryPy, Pylons, web2py and Google's own web app framework.

The programs written for the Google App Engine must use Google-supported APIs and many common APIs are not supported, e.g. the Java Thread API. In contrast to Amazon Web Services where you can set up you own database on a virtual instance or use a Amazon driven non-relational database, Google also only supports its own non-relational database, based on Google BigTable.

The Google App Engine is free of charge for minimal usage, but for more usage, a user pays for the CPU time consumed by their software, data it transfers and data stored. The CPU time is calculated in the number of hours of a 1.4 GHz processor, running on full capacity. The prices for billable resources are as shown in table 2.4.

The main differences between PaaS and IaaS are ease of scalability, flexibility, data lock-in and simplicity. For PaaS there is not much need for the user or the

[9] Google App Engine. http://code.google.com/appengine/
[10] Windows Azure Platform. http://www.microsoft.com/windowsazure/
[11] Heroku, Ruby Cloud Platform as a Service. http://heroku.com/
[12] Google App Engine quotas, http://code.google.com/appengine/docs/quotas.html, accessed: 10. June 2010

administrator of the service to handle scalability. The platform itself is running on multiple instances, and even multiple data centers, and it should be able to scale automatically to needs, but the user or administrator can often set a hard limit on scalability. For example, if a hard limit on outgoing bandwidth is set and a picture or a video hosted on this program/platform becomes very popular, instead of paying a large sum of money for that, outgoing transfer would be suspended that day, possibly saving money on unnecessary excess bandwidth. In IaaS, a user needs to terminate or create instances to meet with changes in scalability.

PaaS critics have been warning users about data and functionality lock-in [10]. If a user creates a program for the Google App Engine for example, the user is restrained to using the Google App Engine, as no other provider supports that functionality directly. The data stored in the supported database would also need to be directly modeled and set up for this kind of database. Moving the data from GAE would therefore require transformation which again could become hard and expensive. Although there have been a few projects [19, 20] aiming at creating an open source implementation of GAE, they do not provide all the features of GAE and there is a high risk of these platforms not being stable enough for commercial computing.

The complexity of IaaS in comparison to PaaS is a factor also. Users of IaaS services need to know how to work with and configure the underlying operating systems and middleware, and be sure the their software is scalable enough. With PaaS this is not a factor and therefore PaaS can be simpler to use in the long run.

Software as a Service

Software as a Service (SaaS) has been regarded as the main aspect of cloud computing, but it is more a product of cloud computing than a definition of cloud computing. SaaS delivers applications as a service over the internet, usually in the form of web pages, dismissing the need of installing and running the program on the user's computer. SaaS is usually hosted on PaaS or IaaS. Examples of SaaS services are Google Apps[13], Chargify[14] and SalesForce[15].

On Google Apps, a SaaS user can register for the service for free or pay for premium service, which includes support, more storage and a higher uptime guarantee. The user can then access an online email client, online office suite, which includes spreadsheet, presentation and word processing software and more.

[13] Google Apps, not to be confused with Google App Engine. http://www.google.com/apps/
[14] Recurring billing SaaS solution. http://chargify.com/
[15] Customer Relationship managemnt (CRM) SaaS service. http://www.salesforce.com/

2.1.3. Deployment Models

There are three primary cloud deployment models. Each can exhibit the previously listed characteristics; their differences lie primarily in the scope and access of published cloud services, as they are made available to service consumers.

Private clouds

Private cloud infrastructures are operated on the infrastructure and used by a single organization. This enables an organization to use existing hardware as a cloud computing resource. This can be managed by the organization itself or by any third party. This can be very useful if the organization owns its own hardware which it wants to enable for cloud usage. An example of a private cloud is a university's cloud. The University of Iceland owns a cluster which is set up as a grid, but a grid is limited to the software currently running on it and can be more complicated for the end user to operate than cloud instances. Work has been done in setting up the Eucalyptus[16] IaaS software on this grid, which would enable users (students or faculty) to get computing instances of their own where they can set up their own software without the need of administrators.

Community clouds

Community cloud infrastructures are cloud infrastructures which are shared by several organizations that serve a certain community with shared concerns. Similar to private clouds, they can be managed by these organizations or by any third party. An example of a community cloud is NEON[17], or the North European cloud computing project, which was a cloud computing project of NDGF[18]. Several north European universities and research facilities share their computing facilities on a very large grid. The idea of the NEON project is to evaluate the possibility of using these grids for a large community cloud. If so, private cloud infrastructures will be set up on each university's or research center's grid and these clouds will then be shared as one common community cloud.

Public clouds

Public cloud infrastructures are available to the general public or large industry groups. Public clouds are usually owned by a single organization that sells the service. Examples of public clouds are the Amazon EC2 and Rackspace clouds

Hybrid clouds

When two types of clouds are connected or used they are termed hybrid clouds. For example if a private cloud facility has limited amount of resources, when the

[16]The Eucalyptus private IaaS sytem. http://www.eucalyptus.com/
[17]NEON, Northern Europe Cloud computing. http://www.necloud.org/
[18]Nordic DataGrid Facility, http://www.ndgf.org/

demand of resources is greater than those available, they can scale to a public cloud, instead of having resources for peak usage it switches to a public cloud when needed.

2.2. Software Architecture

In this thesis we are using software architecture definitions and terminologies from Bass et al. [1] and Hilliard, R. [2] are used. Software architecture is what is essential or unifying about a software system: the set of properties which define the software system's structure, or form, the behavior, function, value, cost, and risk. Software architecture can be defined as:

> "The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution" [2].

The view held on software architecture in this thesis, joined with the above definition, is that an architecture is a conception of a system, an abstract form. Software architecture can exist without any documentation or any concrete or physical representation. Software architecture embodies the essential or key properties about a software system. The architecture is understood in the context of the software system, not in isolation. To understand the architecture it is essential to understand the environment and how the system relates to it. Software architecture is, on the other hand, not an overall physical structure of the system.

In other words, every software system has an architecture, just as every house, bridge or airplane, but it does not need to be documented or understood.

The practice and the study of software architecture is concerned with the tools, methods and ideas to create fundamental system structure. Just like building a house, it is not necessary to use architectural discipline or build it with any architectural basis, but doing so makes it more likely that the house withstands more external or internal intrusion like wind, snow, earthquakes, ageing, etc. The same goes for software architecture, it is a discipline for increasing the quality of the software, helping the developer work with the requirements of the system and extend the lifetime and overall quality of the system. Architectural requirements, architectural design, architectural description and architectural evaluation are examples of techniques and activities a software architect uses for the architectural design process (see figure 2.3) and defines the software architecture.

There is a whole range of software architectural characteristics that can be of interest to a software architect. Such as, how does the system perform under load? How does the

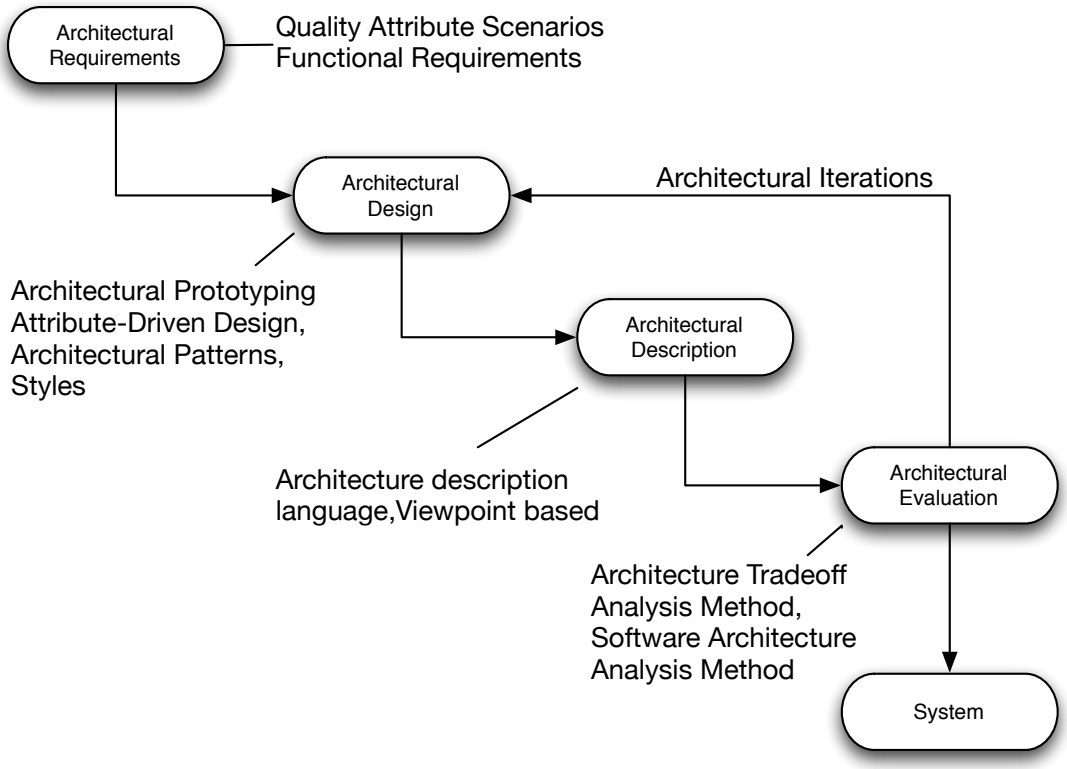

Figure 2.3.: Architectural Design Process

system scale? What is the peak throughput of a software system given certain hardware? How likely is the software to fail? How easy is it to manage? How can it be used for people who are disabled? These characteristics are called "quality attributes" and are the subject of the next section.

2.2.1. Architectural Qualities

Functionality, or the ability of the system to perform the work for which it was intended, and quality attributes, are closely related but independent from each other. Many architectural decisions address concerns that are common, driven by the need for the system to exhibit a certain quality property rather than provide a particular function. Functionality often takes the front seat, and even the only seat, in the development process. This is short sighted when systems are frequently redesigned, because they are difficult to maintain, port, or scale or are too slow. Software architecture is the first stage in software creation where quality requirements can be addressed [1].

The choice of function does not for example dictate the level of security, performance, availability or usability. However, a software architect can choose a desired level of each, though this is not to say that any level of any quality attribute is achievable, in the sense that high modifiability often means lower performance, etc. Furthermore, one can not, for example, define complete scalability of a given system, but one can set a desired scalability level of the system. A quality attribute can be defined as a relative level of quality to fulfil a set requirement.

> "Achieving quality attributes must be considered throughout design, implementation, and deployment. No quality attribute is entirely dependent on design, nor is it entirely dependent on implementation or deployment. Satisfactory results are a matter of getting the big picture (architecture) as well as the details (implementation) correct." Bass et al. [1]

System quality attributes have been of interest to the software community since the 1970s. In this thesis the Bass et al. [1] description and characteristics will be used. Here quality attribute scenarios (QAS) are used to define quality attribute requirements. Bass et al. define quality attribute scenarios as follows: "a quality attribute scenario is a quality-attribute-specific requirement" [1]. A quality attribute scenario consists of six parts:

Source of stimulus. This is who generated the stimulus. This can be some role, such as developer, a computer system or any other actor.

Stimulus. This is the condition that needs to be considered when the stimulus arrives at a system.

Artifact. This is the artifact that is stimulated. It can be the system itself or some part of it.

Environment. This defines the conditions when the stimulus occurs. This might be for example a running state of the system or a development state.

Response. The response is the activity, or the change, undertaken after the arrival of the stimulus.

Response measure. This is a measure of the response that the requirement can be tested against.

Table 2.5 shows an example of a quality attribute scenario simplified from a quality attribute scenario for our prototype. Figure 2.4 shows an example of a modifiability scenario.

Scenario(s):		A developer wants to allow the user to change the background color of the user interface.
Relevant Quality Attributes:		Modifiability, usability
Scenario Parts	**Source:**	Developer
	Stimulus:	Wants to allow the user to change background color
	Artifact	System, user interface
	Environment:	Development
	Response:	Code is changed, such that a user can change user interface background color
	Response Measure:	5 hours of development time

Table 2.5.: Quality attribute scenario example

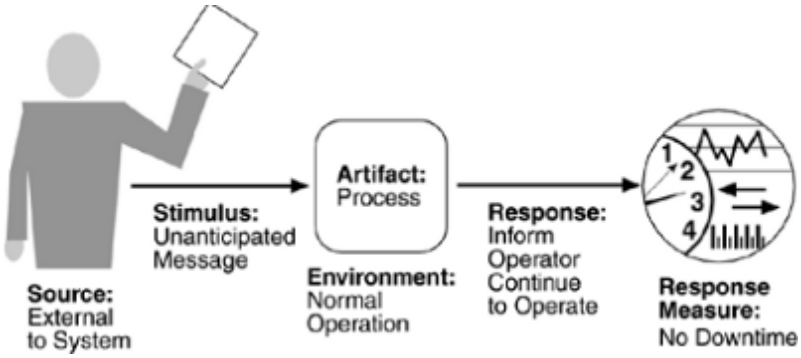

Figure 2.4.: Quality attribute parts, from Bass et al.[1]

Bass et al. [1] list six main types of system quality attributes along with examples of business qualities and architectural qualities. The main system quality attributes types are:

Availability is concerned with a system failure and its associated consequences. Some aspects of systems failures are the frequency the system failure may occur and what happens then, the amount of time the system may be out of operation, when failures may occur safely, how failures can be prevented, and what kinds of notifications are required when a failure occurs.

Note that failure and fault is not the same thing, fault is a system state that is not observed by the system's users but if not handled correctly it may become a failure, which will be observable by the system's users. A fault might be a lack of storage

resources, which might be solved by freeing some disk space, but if it is not handled it might stop the operation of the software system thus leading to a failure.

The availability of a system is the probability that it will be operational when it is needed. This is typically defined as:

$$\alpha = \frac{\text{mean time to failure}}{\text{mean time to failure} + \text{mean time to repair}}$$

Modifiability is concerned with the cost of change to a software system. It can be broken down as what can change and when, and who makes the change (artifact or environment). A change can occur to any aspect of the system (artifact), for example the functions of the system, its platform or its environment. Change can also happen at any time (environment), for example a developer may change the source code or a user may change a language settings on his website.

When a change has been specified, a new implementation must be designed, implemented, tested, and deployed. All of these actions take time and money, both of which can be measured.

Performance is concerned with timing, e.g. with how long it takes the system to respond when an event occurs. A performance scenario begins with a request for some service arriving at the system. Satisfying the request requires resources to be consumed. While this is happening, the system may be simultaneously servicing other requests. An example of a performance scenario is: "A user starts 10 requests a minute under normal operation, and each operation takes less then a second to respond."

Security is concerned with the systems ability to serve its own users while denying any unauthorized usage. An attempt to breach security is called an attack and can be an unauthorized attempt to access data, modify data, or deny service to legitimate users. A secure system can be characterized as a system providing non-repudiation, confidentiality, integrity, assurance, availability, and auditing [1].

Testability is concerned with the ease with which software can be made to demonstrate its faults or correctness. Testability refers to the probability, assuming that the software has at least one fault, that the software will fail on its next test execution [1].

Usability is concerned with how easy it is for the user to accomplish a desired task and how the system supports its users. It can be broken down into five areas: learning system features, using a system efficiently, minimizing the impact of errors, adapting the system to user needs and increasing users confidence and satisfaction.

Along with these main types of quality attributes scenarios, Bass et al. [1] mention custom systems quality attributes, such as scalability, portability, and interoperability. They also define a generic QAS for each quality attribute. For each use of these QAS the user has to fill out the six parts of the scenario generation framework, that is source, stimulus, environment, artefact, response, and response measure. The authors also include business qualities (cost, schedule, market, and marketing considerations) and architectural qualities (conceptual integrity, correctness and completeness and build-ability) which we will not discuss further.

The above mentioned qualities can be a good basis when designing software architecture and working on the architectural description of the software architecture. To create quality attribute scenarios, as seen in table 2.5, a software architect needs to think about the software architecture with these quality attributes in mind. These quality attributes were created to include common quality use cases and to be extended to new and customized quality attributes.

2.2.2. Architectural Description

An architectural description (AD) is defined by IEEE as: "A collection of products to document an architecture." [2]. Architectural description is by definition the description of a software system's architecture and should be in such a way that its stakeholders can understand it and it should demonstrate that the architecture meets their requirements. This documentation can be from simple text documents to a description using an architectural description language [21]. In this thesis, we are going to use the IEEE recommended practice for architectural description of software-intensive systems [2, 22]. This recommended practice introduces the concept of a viewpoint from which the system's software architecture is described, see figure 2.5.

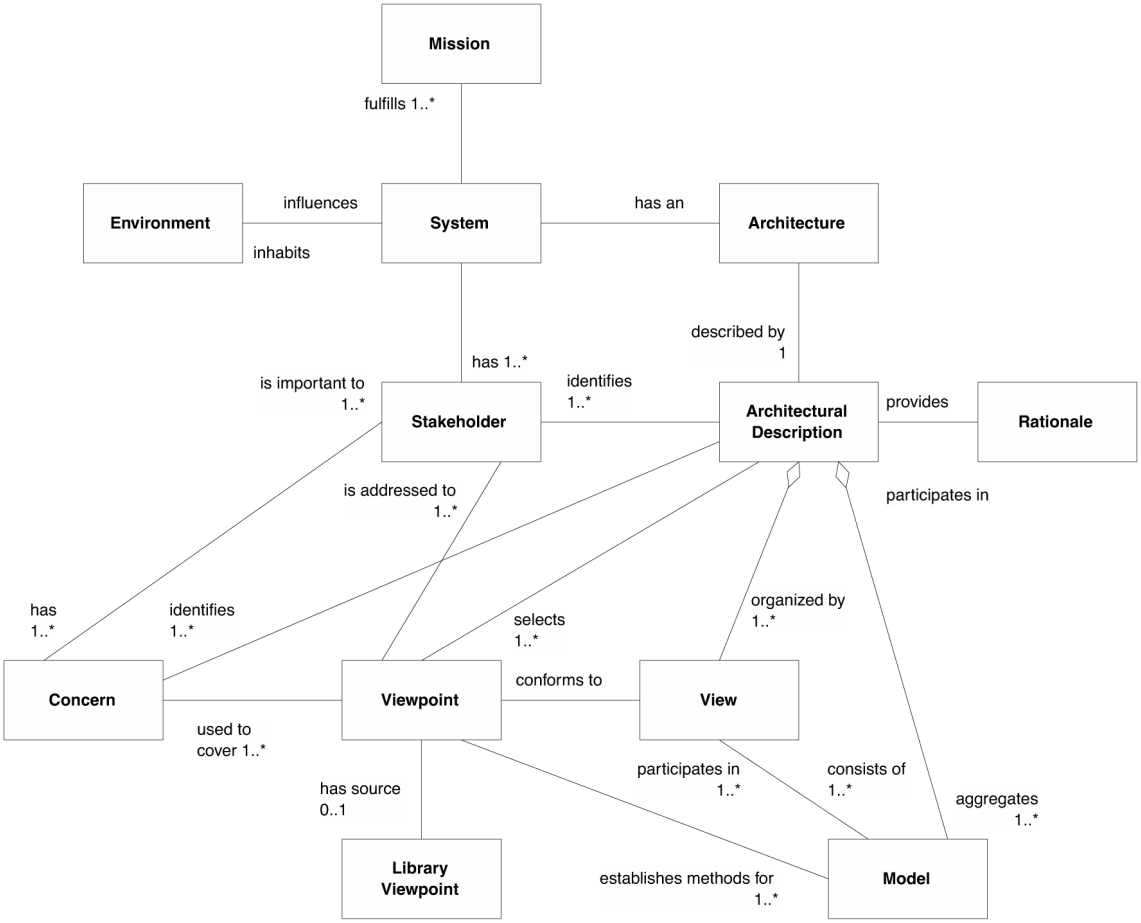

Figure 2.5.: Architectural Description Ontology, from IEEE 1471 [2]

A concrete architectural description consists of multiple views that each correspond to a viewpoint [22]. Clements et al. [23] recommend the use of three viewpoints, Module, C&C and Allocation viewpoints, for architectural description. Using multiple viewpoints helps in analyzing the architecture from multiple views and reduces the possibility of over analyzing the architecture from one specific viewpoint. This thesis will base architectural descriptions on these viewpoints.

Module viewpoint is concerned with how the functionality is mapped to the units of implementation. It visualizes the static view of the systems architecture by showing the elements that comprise the system and their relationships. A module view contains modules with their interfaces and their relations. Here a module is a code or implementation unit, including packages, classes and interfaces in Java. Its relations include associations, generalizations, realizations and dependencies.

Module Viewpoint example The examples in this section will use an imaginary web messenger software system. In this system, users can identify themselves and start or join a chat session "a chat room". There they can see everyone joined in the chat and send messages to the chat session. When a new message arrives at the chat session the client pulls it from the server. The module view of the web messenger can be described by using the class diagrams of UML, describing the system top down by starting with the a top-level diagram and ending at the class or interface level. Figure 2.6 shows a package diagram for the web messenger software and figure 2.7 displays a class diagram for the messenger logic package.

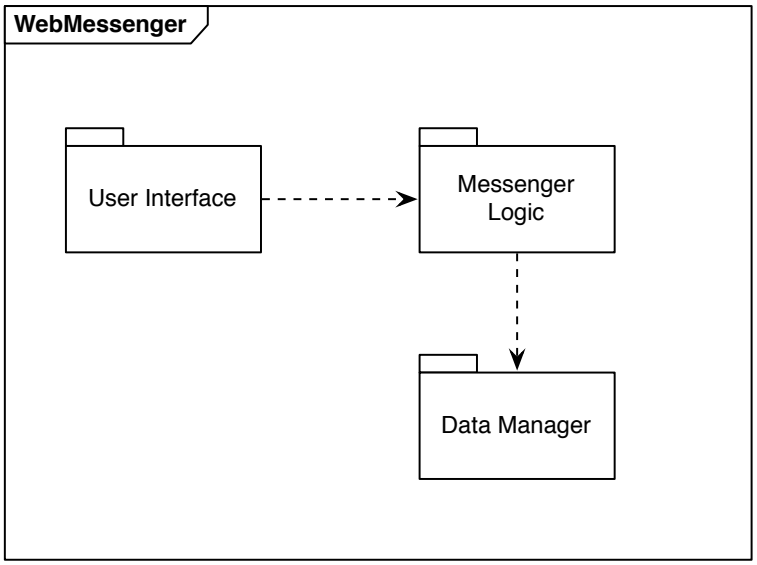

Figure 2.6.: Model View example, package overview of the web messenger software system

Component and Connectors viewpoint (C&C) is concerned with the runtime functionality of the system. In other words, what does the system do? In this viewpoint, the system's software consists of components and connectors, where components are units of functionality which define what parts of the system are responsible for which functionality and connectors which are communication and coordination relationships between components and define how components exchange control and data.

The properties of both the components and the connectors in the architectural descriptions will be described below. This is done with both written explanations and diagrams, showing protocols, state transitions, threading, concurrency issues or what is relevant to the architecture at hand.

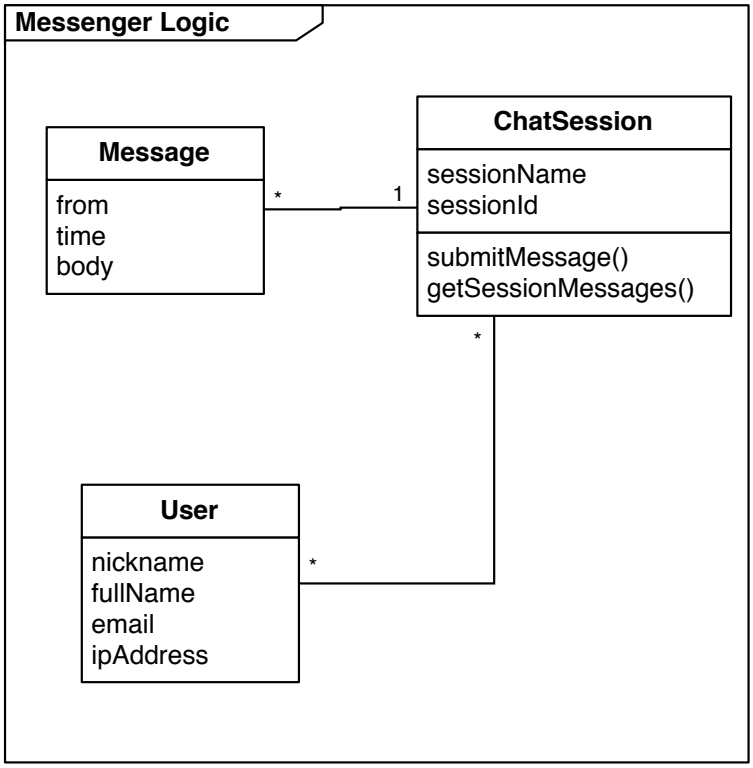

Figure 2.7.: Module View example, decomposition of the Messenger Logic Package

C&C Viewpoint example The web messenger has four major functional parts, as shown in figure 2.8. Components are represented by UML active objects and connectors by links with association names and possibly role names. The diagram in figure 2.8 cannot stand alone, as component and connector names are only indicative of the functional responsibilities related to each. A description of a component's functionalities in term of responsibilities should therefore be provided:

- Browser is responsible for 1) Client-side functionality, 2) displaying the UI correctly, 3) updating new messages by getting new messages from the UI, 4) notifying the UI of new submitted messages from the user.

- User Interface is responsible for 1) rendering the presentation of the user interface, 2) managing requests from the user, mainly submitting new messages and update all session messages, 3) handling new messages and make sure they are representable with the Messenger Logic, 4) rendering all chat messages.

- Messenger Logic is responsible for 1) knowing the status of users in conversations, 2) knowing what messages each user in conversation has received, 3) transmitting messages to/from database.

- Database is responsible for 1) storing messages, 2) fetching messages, 3) storing sessions and, 4) fetching sessions.

Just as the components, the connectors also need to be described in more detail. The level of detail needed depends on the architecture at hand. For some connectors, it may be sufficient to have short textual description, but for others it may be best to explain them by UML sequence diagrams. Our Messenger application has three connectors:

- AJAX. Asynchronous JavaScript and XML, is a web standard for making clients communicate with servers.

- MVC. A standard Model-View-Controller pattern is the protocol for this connector that connects the messenger logic serving as the model and the User Interface serving as View and Controller.

- JDBC is the connector that handles standard SQL queries with the JDBC protocol.

Sequence diagrams can be used either to describe the connectors protocol individually or to provide the "big picture" showing interaction over a set of connectors. In our example an overall sequence diagram describes the big picture, see figure 2.9.

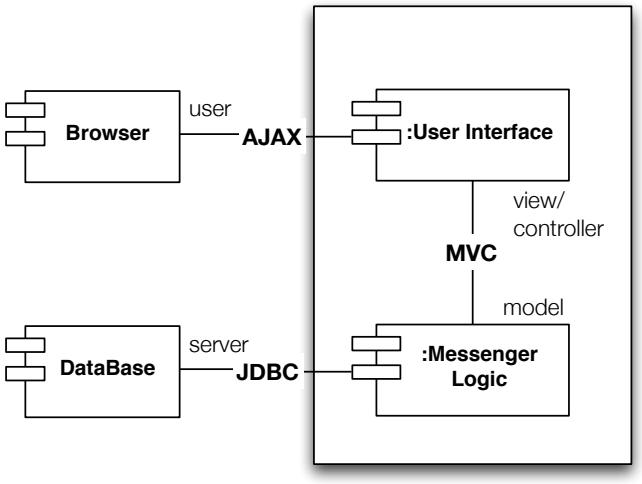

Figure 2.8.: C&C view example, web based messenger

Allocation viewpoint is concerned with how the software elements of the software system are mapped to platform elements in the environment of the system.

The allocation viewpoint includes deployment, implementation and work assignment

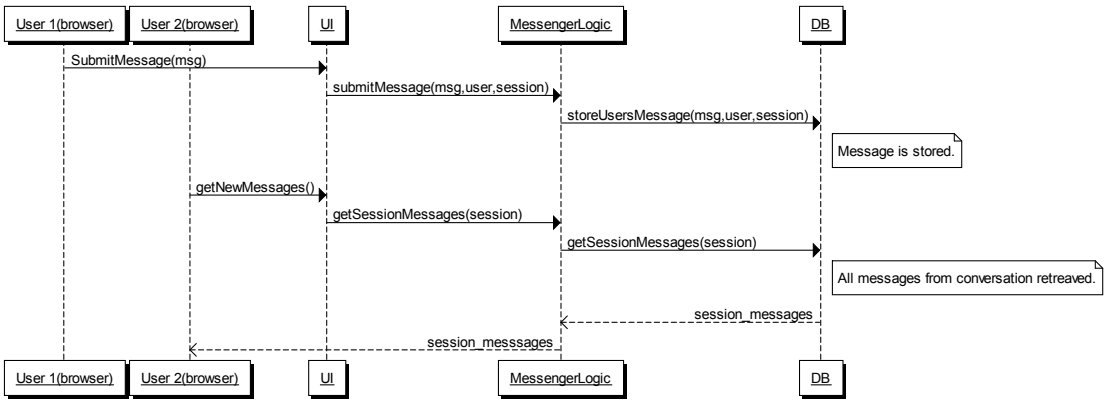

Figure 2.9.: C&C view example, sequence diagram, web based messenger

structure, and a deployment diagram is the main view.

The deployment viewpoint has two element types, software elements and environment elements, and three relation types, allocated-to relations, dependencies among software elements and protocol links among environmental elements showing the communication protocol used between nodes [1].

Deployment Viewpoint example Figure 2.10 shows the deployment view of the web messenger software system using a UML deployment diagram. The deployment is a three-tier deployment, where presentation is to run on the client, domain code to run on a Java application server, and data is stored on a database server.

- Environmental elements (shown as UML nodes)
  - The Browser is the input and final output for the messages.
  - The Application Server is the machine serving the UI through a web server and serving all other application level functionality.
  - The Database Server provides secondary storage.
- Software elements (shown as UML components)
  - The Browser displays the client-side presentation and runs client side scripts and interacts with the User Interface via AJAX.
  - The User Interface renders messages and the presentation layer and delivers

messages to/from users from/to the Messenger Logic.

- The Messenger Logic keeps track of chat sessions, users participating in these sessions and messages associated. It interacts with the Data Manager for secondary permanent storage.

- The Data Manager takes messages, sessions and users and sends them to a relational database and retrieves sessions from the database.

- MySQL is an open source SQL database which handles database related functionality of the system.

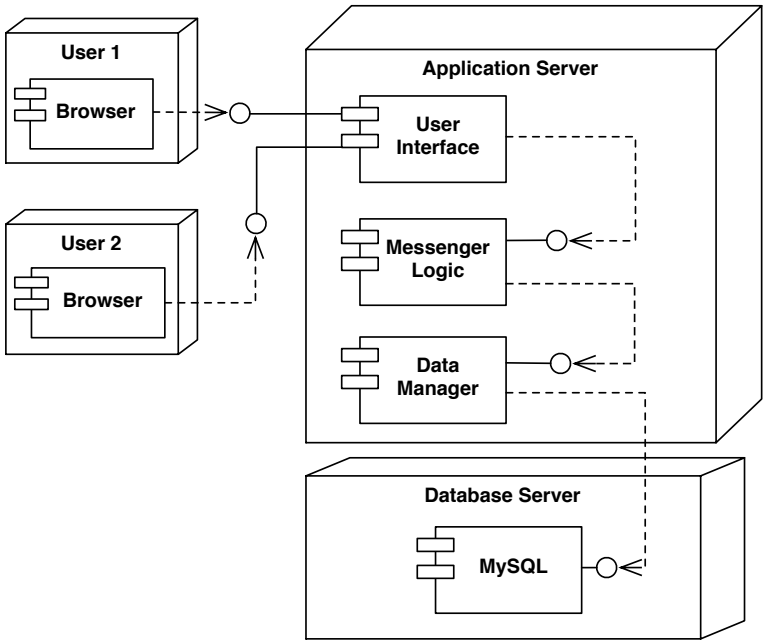

Figure 2.10.: Allocation/Deployment view example, web based messenger

Having made and reviewed an architectural description, the next logical step might be to create a architectural prototype, which will be the background of the next section.

2.2.3. Architectural Prototype

So far a few theoretical techniques a software architect has at his disposal have been presented, next an experimental technique will be reviewed, namely architectural prototyping. Examples of other experimental techniques, that will not be further described, include simulation and scenario-based methods with explicit stakeholder involvement [24].

Once an architecture has been defined, an architectural description documented and the architectural quality examined, it can be analyzed and prototyped as a skeletal system. Having an executable software system early in our development cycle, can help us in several ways. First by verifying whether the quality requirements are fulfilled, we can exchange prototype parts with complete software version allowing us to review and measure the complete parts and last we can detect performance problems early in the development cycle [25]. Bardam et al. [3] define the concept of architectural prototyping as:

> "An architectural prototype consists of a set of executables created to investigate architectural qualities related to concerns raised by stakeholders of a system under development. Architectural prototyping is the process of designing, building, and evaluating architectural prototypes."

Bardam et al. [3] also point out a number of architectural prototyping characteristics as they define an ontology of architectural prototyping, which relates to their definition of architectural prototyping, see figure 2.11. Below these characteristics will be discussed along with architectural prototyping in general.

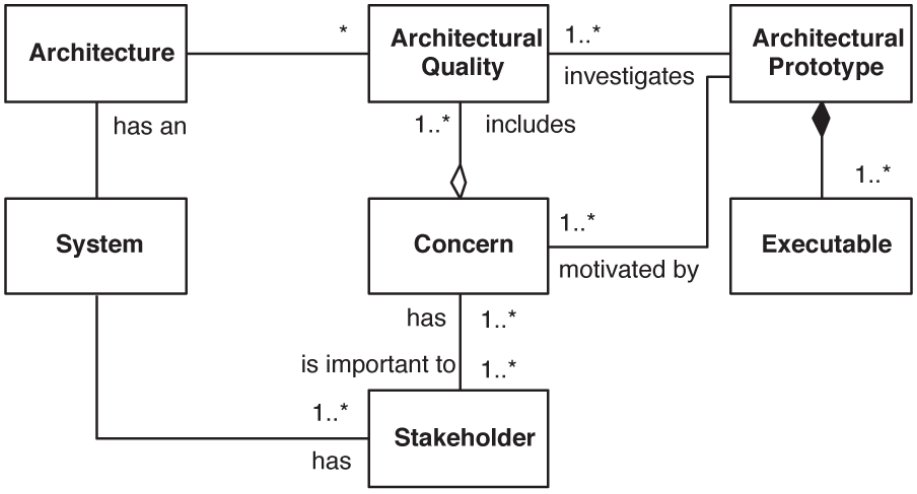

Figure 2.11.: Ontology of Architectural Prototypes as seen from [3]

Architectural prototypes can be classified into three general types: exploratory, experimental and evolutionary. Exploratory architectural prototypes are created to explore the architecture design space, multiple prototypes are usually created, analyzed and executed in order to find a solution to a given problem. Experimental architectural prototypes are created to evaluate a specific architectural decision, a single prototype is usually created and evaluated. Evolutionary architectural prototypes are created as a series of prototypes, where each prototype is built as revision of the last one.

The architectural prototypes can be described by five characteristics: Exploration and learning, Quality attributes, no function per se, architectural risk and knowledge transfer and conformance [26, 27, 3]. Exploration and learning prototypes are constructed to learn about the effect of architectural decisions made, typically ignoring the main function of the system itself. Quality attributes is often the main motivation for building architectural prototypes and therefore measuring the quality implications of decisions. No functionality per se is implemented into the prototype, that is little or no business or a user functionality is implemented in the prototype. Architectural risk is often addressed in architectural prototypes. Knowledge transfer and architectural conformance is addressed by making developers learn about the software architecture through the prototype's code.

2.3. Architectural operations

Architectural operations are operations on the runtime architecture of a system. They can be ways to modify the architecture by patching errors and bugs, updating software to a new version or even change libraries and connectors used by the software. These kinds of operations are usually done on software that is not running, but they could be more useful if done on running software or live systems. In this section we will describe ways to modify the architecture of running software.

2.3.1. Architectural Change

Software changes range from modifying a tiny part of the system, like a line in a configuration file, to changing the whole system, but these changes can be categorized into three types [1]: local, non-local and architectural. A local change is usually a very small change and can be accomplished by modifying a single element. A non local change can be bigger and requires multiple element modifications, but leaves the underlying architectural structure intact. An architectural change affects the fundamental structure of the system (the elements interact with each other and will likely require change all over the system, this therefore changes the pattern of the architecture).

Architectural change is about the ability to model and analyze the change of software architecture, and our interest mainly is in runtime architectural change. That is, how does a developer change an architecture on a running software system. This change is the key to building autonomic systems [4]. In cloud computing, scalability is a big factor in running software systems. These systems need to be able to run on multiple computing instances, they need to be able to scale instantly and need to be able to modify themselves to adapt to rapidly changing environments. It can be extremely hard to build

programs to handle these requirements with a static and solid architecture, which brings architectural change in as a favorable factor to scalability of a cloud computing software architecture.

For a developer, administrator, or an end user, an architectural change is usually the process of shutting down the running instance of the program to be changed, installing the update by overwriting the program or some of the program components with the changed version and then running the program again. This process can take time, especially if the program is big and the change process is complex. For some software this kind of update, or upgrade, is not possible, e.g. mission critical software and/or high dependence systems such as payment processing gateways for credit-card companies or air traffic control software. These kinds of systems usually require strict update and change protocols where a secondary system, a direct duplicate of the original one, takes over while the original system is being updated. With architectural change that models and analyzes the change of runtime architecture, which is the focus here, there is no need to turn the whole software system off for an architectural update. But rather, the system is built on multiple small modules/components which can be updated individually without restarting the software. This kind of update step is called architectural operation.

2.3.2. Architectural Scripting and Architectural Operations

Architectural operation is a unit of architectural change and an architectural script is a sequence of these operations. Architectural operations and architectural scripting have been the research of [4], [28], and [8] and is modelled by the ontology in figure 2.12. As previously stated, little work has been done in modelling the dynamic aspects of soft-

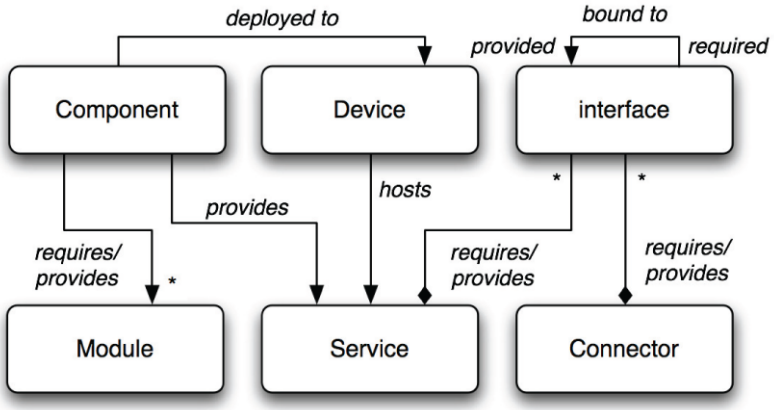

Figure 2.12.: Ontology of Architectural Scripting Language architecture, according to [4]

ware architecture. Architectural scripting is a way to model the dynamic aspects of run-

time and deployment-time software architecture. An architectural script consists of sequences of operations on a runtime architecture, such as deploying a binary component to a device or instantiating a service from a binary component. This set of operations is named architectural scripting language, or ASL [4]. As the ontology in figure 2.12 displays, Architectural Scripting Language operations operate on six parts: a device, a component, a service, a module, an interface and a connector.

Device
A device is a physical or virtual (VM or JVM in Java) device.

Component
A component is a unit of deployment. This can be a package of executable code with explicit dependencies, usually a binary package. Components can export modules that other components may require. Components are deployed to devices and provide services, which can be an instantiated form of the component.

Module
A module is a typed library, a class, an API etc.

Service
A service is a typed unit of instanced software, a running instance of component, with explicit dependencies in the form of interfaces and explicit capabilities in the form of provided interfaces.

Interface
An interface is a typed unit of association between services. In many cases implemented as an object reference (required) or as an object (provided)

Connector
A connector is a way to expose a method to show an interface or a service.

An architectural operation is a unit of change for each of these parts. For example an updated module, new exposed service, added component, etc. The Architectural Scripting Language is a scripting language that defines these operations and allows multiple and sequential change on the software architecture. In the Hydra project [8] examples of operations are: deploying a component onto a device, *deploy_component(component, device)*; starting a component's service on a device, *start_service(service, component, device)*; and starting a device *start_device(device)*.

Architectural operations are explained below with the example of the Web Messenger:

From the allocation viewpoint, we can think of the environmental element Application Server as a *device*.

The software elements from the allocation viewpoint, User Interface, Messenger Logic and Data Manager would be presented as *Components*

Libraries that the Web Messenger uses, such as MySQL JDBC connector, are examples of *modules*.

The components expose *services*. For example (from the C&C viewpoint sequence diagram) the messenger logic might have a public interface for the UI to communicate with (with functions like: sendMessage() and getMessages()). The instance of that class would be exposed as a service.

This MessengerLogic class would be abstracted by an interface.

The *connectors* from the C&C viewpoint can be represented as connectors.

Now if one would want to update the MessengerLogic component and start it as a service, one could do so with an ASL script:

```
1 update_component(ApplicationServer, MessengerLogic)
2 start_service(MessengerLogicService, MessengerLogic, ApplicationServer)
```

Listing 2.1: Example of ASL script on the WebMessenger

This script would 1) update the MessengerLogic component on the device ApplicationServer, assuming that it knows where the updated version is and 2) start the service MessengerLogicService which the MessengerLogic component exposes on the device ApplicationServer. As can be seen, although this is a serious architectural change, deploying this kind of change is easy and given that the ASL implementation is correct, the operations taken can be tried and proven with mathematical measures. The other components should not be directly affected by this change and therefore the downtime of the software is minimal. Modelling runtime change to an architecture with the notion of a script has several advantages [4]:

- A script can be analysed independently of the particular system configuration it is executed on, to asses, for instance, whether it may introduce violations to a certain style.
- A script can be reused across different contexts. For instance, in autonomic computing a reconfiguration script may capture a reusable solution to a specific problem.
- A script, as will be demonstrated, is operational enough to support direct implementation while at the same time being amenable to useful analyses.

However, this modelling has its limitations. For example:

- A script has its limitation, i.e. an ASL script cannot be used to configure devices, modules, or services, e.g. by setting web module to use port 443 for SSL.

- An ASL device can hardly fully encapsulate the device it represents, e.g. does the device have a touch screen? Is the required kernel installed? Does it have firewall configured? To make an ASL script fully encapsulate the scripting language and interpreter needs to know the most aspects of the device, such as screen resolution and type, keyboard buttons available, kernel modules available, etc. No implementation of ASL can include all possibilities of device types and setups, but given limited and fixed set of attributes of devices this can however be done.

3. Cloud Computing and Architectural Operations

After having viewed the background of cloud computing and architectural operations, it can be argued that the Architectural Scripting Language can provide advantages for runtime management of cloud computing software systems architecture. As a benefit of cloud computing scalability, cloud applications need to be changeable to adapt to the changes of the environment. The Architectural Scripting Language can help with that adaptation with architectural change on runtime software. Thus it is concluded that Architectural Scripting Language in cloud computing is a combination worthy of further investigation. Below our design and implementation of a Cloud Architectural Scripting Language (Cloud ASL) will be described, in addition to related work done in that field.

3.1. Architectural Scripting in a Cloud

The Architectural Scripting Language and cloud computing could complement each other and the results of the joined forces might result in scalable cloud computing software. However, ASL is not designed for cloud computing, which provides the opportunity to design and developing on ASL implementation specified for our needs for cloud computing infrastructure. This implementation will hereafter be called Cloud ASL. The aim is to use the IaaS model for our implementation. The other service models do not apply as well to our problem, since they do not offer the same freedom to use languages and libraries and other technology as IaaS does. Other service models in contrast offer more and simpler scalability options, and do not require knowledge of operating systems and distribution.

For designing Cloud ASL architecture and operations we can review section 2.3.2 and see some modifications and simplifications that are possible, for example removing connectors and modules from the ontology, leaving devices, components, services and interfaces.

Device is mapped to a cloud instance, or a VM running on a cloud instance. There can be multiple VMs running on each cloud instance, therefore multiple devices on each instance.

Component is mapped to a unit of deployment, perhaps a binary package with an executable code. A component should provide a specific functional behavior. A component is deployed to Devices and provides services through interfaces.

Service is an instanced component, with explicit dependencies in the form of interfaces and explicit capabilities in the form of provided interfaces.

Interface is a typed unit of association between services. A component provides a service as an interface.

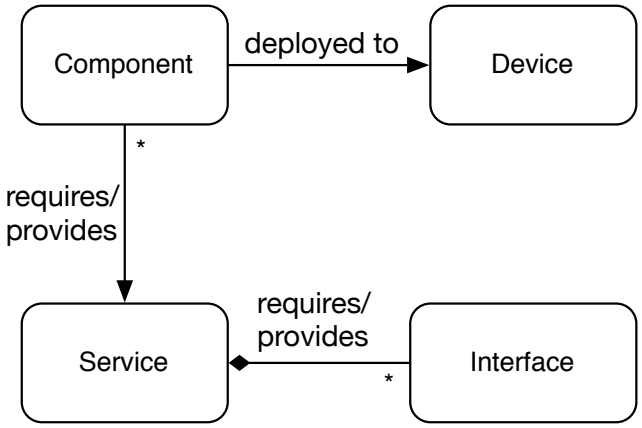

Figure 3.1.: Ontology of Cloud ASL

3.1.1. ASL Operations

The Cloud ASL operations are the main way to interact with Cloud ASL. The operations needed to model the architectural change on cloud computing architecture can be limited to three types of operations: device, component and service operations.

Devices

Operation	Description
device:create_instance_device(type)	Create a new cloud instance of a certain type. This instance will hold the VM.
device:create_vm_device(device)	Creates a new VM on an already running device/instance. This can be useful for running multiple programs on the same instance.
boolean:start_device(device)	Start a stopped device.
boolean:stop_device(device)	Stop a running instance.
boolean:destroy_device(device)	Shuts down and destroys cloud instance. This action is irreversible and all information on that device will be lost. If multiple VMs are running on this device they will be lost.
device:clone_device(device)	Creates a direct replica of a device.

Table 3.1.: List of Device ASL operations

Components

Operation	Description
component:install_component(url, device)	This installs a component, from a given URL to a given device
boolean:uninstall_component(component, device)	Uninstalls a given component from a given device
boolean:start_component(component, device)	Start a given component on a given device
boolean:stop_component(component, device)	Stops a given component on given device
boolean:update_component(component, url, device)	Updates or upgrades a given component, ona given device with component from a given URL

Table 3.2.: List of Component ASL operations

Services

Operation	Description
service:register_service(component, device)	Register a new service from a component on a device
boolean:unregister_service(service, component, device)	Unregister a service from a component on a device
boolean:enable_service(service, component, device)	Start a stopped service
boolean:disable_service(service, component, device)	Stop a running service

Table 3.3.: List of Service ASL operations

As an example of Cloud ASL script, a deployment of the web messenger previously introduced will be used. As seen in section 2.2.2, the example of the web messenger consists of three deployment modules, or components from now on, User Interface (ui.jar), Messenger Logic (mess.jar) and DataConnector (db.jar). These components all run on the same server, or device from now on.

Example of Cloud ASL operations:

```
//creates a small instance referenced as ''server''
Devive server = create_instance_device(''large'')
//lets start the device (i.e.\ start the vm image)
start_device(server)
//next we install our components on the server
Component userInterface = install_component(''http://URI/ui.jar'', server)
Component messengerLogic = install_component(''http://URI/mess.jar'',
    server)
Component dataConnector = install_component(''http://URI/db.jar'', server)
//now all the components have been installed on the device, we can start
    them one by one
start_component(userInterface, server)
start_component(messengerLogic, server)
start_component(dataConnector, server)
```

Listing 3.1: Cloud ASL Example

In this example we assume that the component JAR files are located on some location, http://URI. After running this script the web messenger should be running on the device "server". Nothing has to be deployed to the user device as all functionality needed for the user comes from the user's browser.

3.2. Implementing Cloud ASL

For our implementation of Cloud ASL more implementation decisions had to be made. The first decision was the programming language of use, because other aspects, such as frameworks and the Cloud ASL language specifications, might depend on it. When choosing the programming language, prior knowledge, platform independence, modularity, performance, framework availability, and support had to be kept in mind. Java[1] was chosen as the main programming language. Java is an cross-platform, object-oriented, VM-based programming language, which is easily modular with the use of jar files to distribute Java applications or libraries, in the form of classes and associated metadata and resources (text, images, etc.). OSGi[2] is an modular framework specification for Java which suites our need to abstract Java modules (JAR) and the Java Virtual Machine. The author of this thesis has experience of it making development and implementation faster. As our OSGi implementation Apache Felix OSGi[3] was used.

Amazon EC2 was chosen as our IaaS cloud computing service provider due to many factors, see section 3.3. Amazon EC2 and Amazon AWS is the cloud industry's leading cloud provider for IaaS and has a large and active community. Amazon AWS has database and queueing services that could be used directly, and a complete supported Java client for their API's. For this project Amazon AWS provided generous funding to use for development and testing.

For our Cloud ASL language implementation it had to be decided whether this domain specific language should be internal or external, and if external should it be Turing complete? The difference of internal or external DSL has been described by M. Fowler [29]:

> "External DSLs are written in a different language than the main (host) language of the application and are transformed into it using some form of compiler or interpreter. The Unix little languages, active data models, and XML configuration files all fall into this category. Internal DSLs morph the host language into a DSL itself - the Lisp tradition is the best example of this."

To make an external domain specific language (external DSL) we would need to create it from scratch. We could, for example, build it in XML, or HTML, making it easily viewable, or we could use a simple scripting language. Because making the language an internal DSL could give us a Turing complete language with possible less effort it was decided that we direct our work there. By using an internal DSL it was possible to use or extend an existing language implementation, which would be Turing complete and

[1] http://www.java.com/

[2] The OSGi framework is a module system and service platform for the Java programming language that implements a complete and dynamic component model. http://www.osgi.org/

[3] Apache Felix is a community effort to implement the OSGi R4 Service Platform and other interesting OSGi-related technologies under the Apache license. http://felix.apache.org/

would include the features that were needed. As using the Java programming language had been decided, there were many possibilities of implementations, such as importing Java code as a script using on embedded compiler, like Janinio[4] or importing Java classes dynamically. A Java embedded scripting language was chosen for this project. This enables us to use the full API of Java, interaction to and from Cloud ASL to the software using it and importing scripts into a running JVM. After comparing Java scripting languages, such as Groovy, JRuby, Scala, Clojure and Jython, Groovy was chosen to be our scripting language. Groovy has gained a lot of momentum in the Java world and is now supported in most Java IDEs and there have been a number of frameworks made for Groovy, such as Grails[5]. The Groovy language is a subset of the Java language and supports the Java Language natively just as the Groovy Language. The Groovy and Java languages can also be used together. Groovy's features are similar to those of Python, Ruby, Perl, and Smalltalk, making it easy for many to use. Groovy has native support for importing Groovy scripts into a Java program, which makes it ideal for our use.

As we have chosen a programming language and framework, we can map them to our Cloud ASL ontology:

Devices

Devices are mapped to a JVM running on a cloud instance. There can be multiple JVMs running on each cloud instance, therefore multiple devices on each instance. The cloud computing infrastructure used here is Amazon's EC2 and we interact with it, with the device operations, through the cloud service API, or Amazon AWS SDK. Details of our cloud implementation is the subject of section 3.3.

Components

In the OSGi framework, software is modularized into so called bundles. Each bundle should be focused on specific functionalities, just like a normal Java JAR file is. We associated a component to a single OSGi bundle, which is deployed to a single running OSGi framework, on a JVM, on a single device/instance. This component can provide a service through an interface. The Cloud ASL operations use Telnet as a basis to interact with the OSGi framework on a device, which gives control over the framework's API.

Services

Like previously stated, services are instances of components and are registered as in-

[4] Janino is a compiler that reads a Java expression, block, class body, source file or a set of source files, and generates Java byte-code that is loaded and executed directly. http://www.janinio.org

[5] Grails, former Groovy on rails. http://grails.org

terfaces. We use OSGi's declarative services[6] for our Cloud ASL implementation and needs, which is responsible for starting, stopping, and registering services as needed. This makes Cloud ASL operations of services unnecessary and are therefore not implemented in our version.

Interfaces

An Interface is a typed unit of association between services. A component provides a service through an interface. Since components require the interface of a service to be available on compile time, creation and registration of interfaces can not be as dynamic as devices and components. We implemented the Cloud ASL interfaces as an special bundle, which was installed on all devices, but this can be done in multiple ways.

From the original ASL ontology the following were omitted:

Module

Our implementation relies on bundling libraries, APIs or other modules as or in components, so we can leave the module concept from our implementation.

Connector

The OSGi declarative services is used to connect services internally, which takes care of all internal service connections.

3.2.1. Cloud ASL Operations

Where we are not implementing operations for services, interfaces, models or connectors we need to implement operations for devices and components.

Devices

Table 3.4 does not include start_instance(), stop_instance() and clone_device(). As Amazon EC2 does not support stopping running instances existing in the "instance store (S3)" compared to "EBS", which is the way we started, starting and stopping of instances was left out. The instance store (S3, Simple Storage Service) is the original way to store images of virtual machines but EBS (Elastic Block Storage) is a more recent way to store images and data. Cloning instances is supported by EBS. Therefore, instead of implementing a complex cloning mechanism, that functionality was left out for it to be sup-

[6] OSGi Declarative Services allow automatic registration, activation and deactivation of OSGi services. http://felix.apache.org/site/apache-felix-service-component-runtime.html

Method Summary	
Device	`create_instance_device(String type)`
This creates a new cloud instance of a certain type, for example "m1.small" for a regular small Amazon EC2 instance type. Next it installs JVM and OSGi onto that instance and starts it.	
Device	`create_jvm_device(String type, Device device)`
This creates a new JVM on an already running device/instance and starts OSGi on it. This can be useful for running multiple programs on the same instance.	
void	`restart_device(Device device)`
This restarts the device, which means stops the OSGi framework, reboots the cloud instance and starts the OSGi framework again. The OSGi framework is cached and therefore is started in the same context as it was when shut down.	
void	`destroy_device(Device device)`
This stops the OSGi framework and shuts down the cloud instance. This action is irreversible and all information on that device will get lost.	
Device[]	`get_devices()`
This returns array of currently running devices.	

Table 3.4.: List of implemented device ASL operations from table 3.1

ported when changing the implementation to support EBS. This limitation is further discussed in chapter 5.

Components

Method Summary	
Component	`install_component( Device device, String URI )`
This installs an OSGi bundle from a given URI into given device OSGi framework.	
void	`uninstall_component(Component component)`
This uninstalls an OSGi bundle from a OSGi framework.	
void	`start_component(Component component)`
This starts an OSGi bundle, by default bundles are not started automatically when installed.	
void	`stop_component(Component component)`
This stops a OSGi bundle.	
void	`update_component(Component component, String componentURI)`
this updates/upgrades a bundle with a version from given URI.	
Component[]	`get_components( Device device )`
This returns an array of components installed on a given device.	

Table 3.5.: List of implemented component ASL operations from table 3.2

3.2.2. Architectural Description

Although the architecture of Cloud ASL being designed before implementation, it had to be changed as the limitations and restrictions in our external environment were discovered. Below the final architecture description will be shown using the methods in section 2.2.2.

Component and Connectors Viewpoint Cloud ASL has five major functional parts as shown in the diagram in figure 3.2.

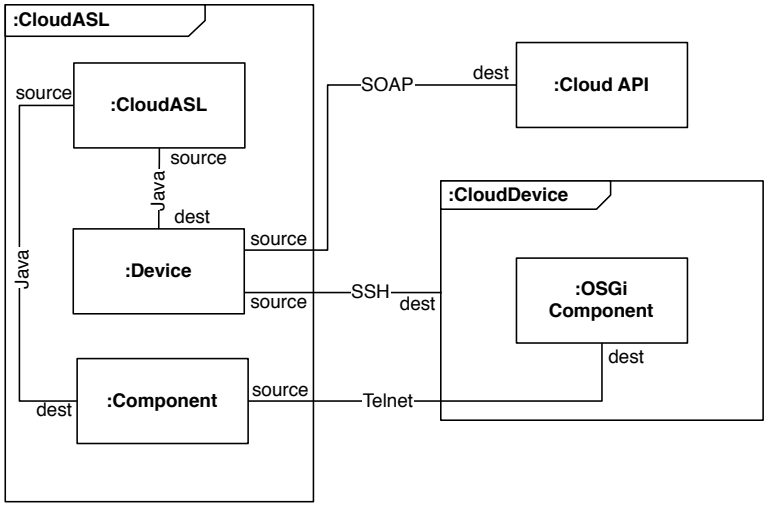

Figure 3.2.: C&C overview of Cloud ASL

- *Cloud ASL.* Responsible for interacting with other modules and the main interface to Cloud ASL. Cloud ASL includes all functions of the Cloud ASL API and should be a sufficient way for any external module to interact with Cloud ASL. Cloud ASL scripts can be passed to this component, which parses the script and runs as a Groovy script with the Cloud ASL framework.

- *Device.* Responsible for 1) managing devices through Cloud ASL's device methods and 2) interacting with cloud API's through the Cloud component.

- *Component.* Responsible for 1) managing components through Cloud ASL's components methods and 2) interacting with components through OSGi Component

- *Cloud API.* Responsible for managing cloud instances. This is where all our cloud functionality lies, currently our EC2 interactions and API. To add or switch cloud providers only this part needs to be modified. This could be, for example, by adding a REST or SOAP connector to another cloud platform.

- *OSGi Component.* Responsible for managing the OSGi framework.

And the connectors associated to the C&C view are

- *SOAP* or Simple Object Access Protocol is a web service protocol based on XML messages through HTTP messaging.

- *SSH* or Secure Shell is a network protocol that allows data to be exchanged using a secure channel between two networked devices.

- *Telnet* is a network protocol that provides a bidirectional interactive text-oriented communications facility via a virtual terminal connection.

- *Java* method calls are used to communicate between classes, components and libraries.

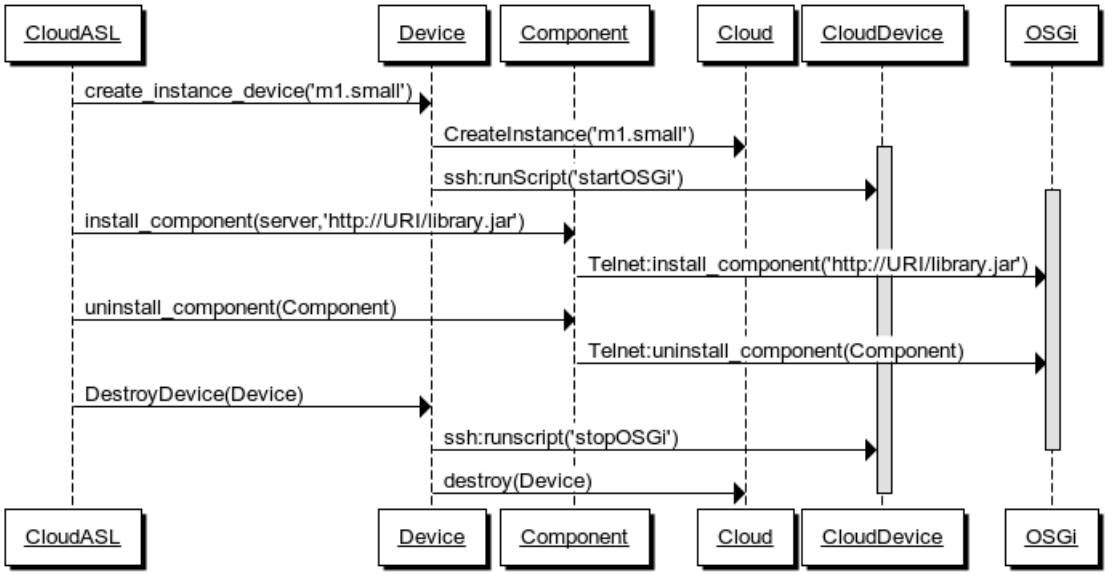

Figure 3.3.: C&C Cloud ASL example sequence diagram

In figure 3.3 a sequence diagram displays the scenario of starting a device, install and run a component, uninstall the component and then destroy the device.

Module View The module structure of our architecture will be shown by the UML diagrams below, starting with the package overview of Cloud ASL in figure 3.4. The ASL package and its dependencies will be further described in figure 3.5, the device package in figure 3.6, the component package in figure 3.7, the cloud package in figure 3.8, the ASL package in figure 3.5, and at last the OSGi package in figure 3.9. Dependencies among packages are also shown; these dependencies arise because of relationships among classes in different packages.

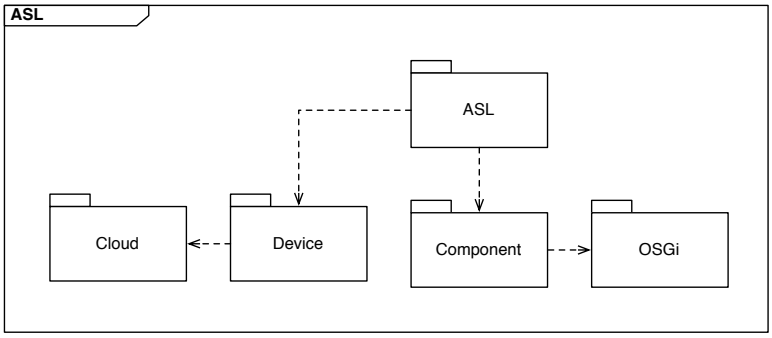

Figure 3.4.: Package view of Cloud ASL

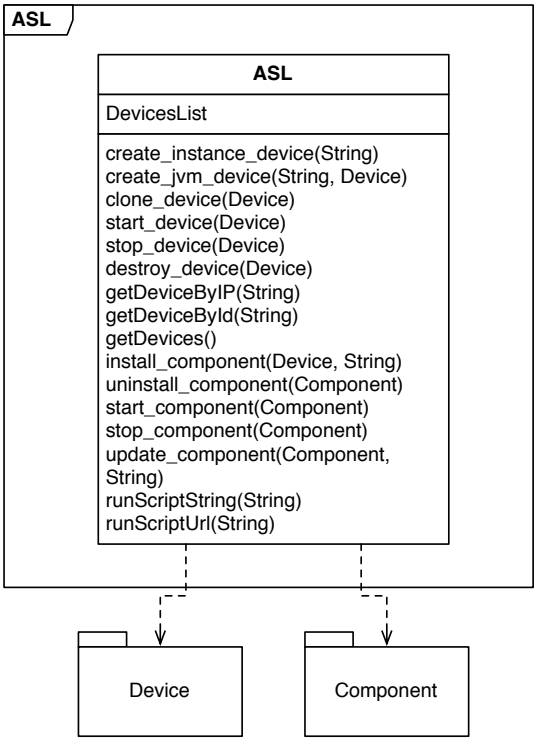

Figure 3.5.: Interface view of the ASL Package

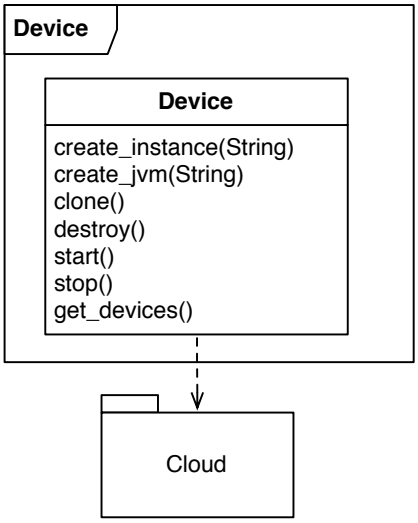

Figure 3.6.: Interface view of the device Package

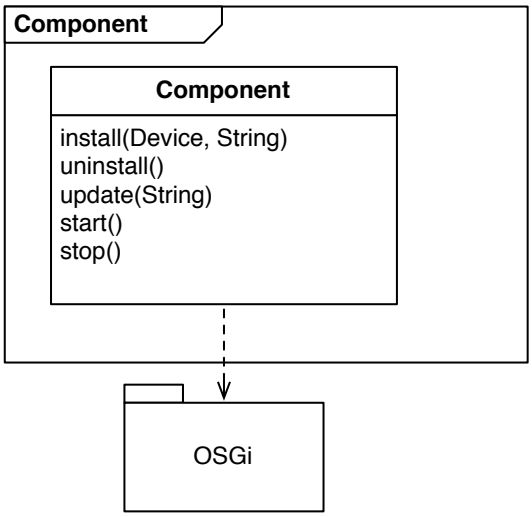

Figure 3.7.: Interface view of the component Package

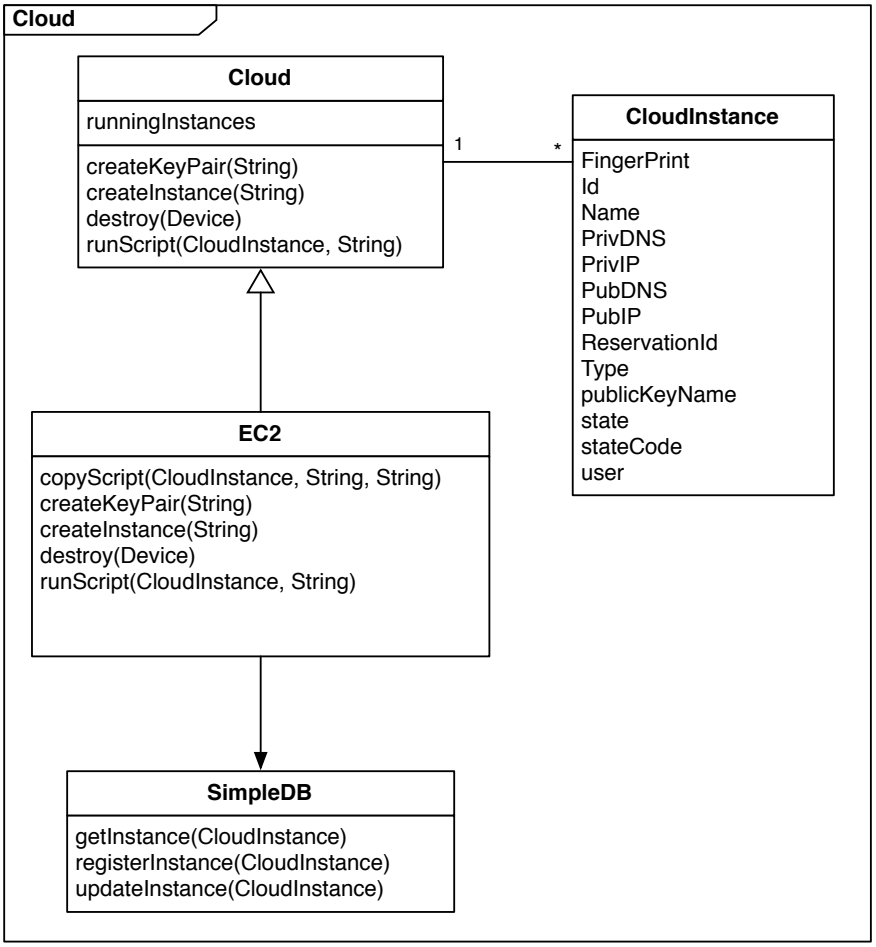

Figure 3.8.: Interface view of the cloud Package

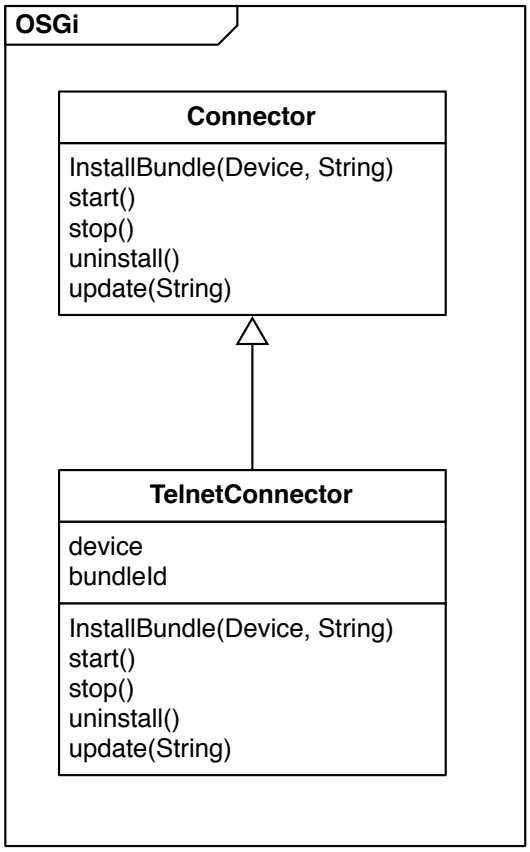

Figure 3.9.: Interface view of the component Package

Deployment View Figure 3.10 demonstrates the deployment view of Cloud ASL using a UML deployment diagram. The deployment diagram shows Cloud ASL to be running on one server and the OSGi components running on different servers. However, in reality, they could be running on the same server, and multiple OSGi instances could be running on a single server.

The following elements are of interest:

- Environmental elements (shown as UML nodes)
  - *Cloud ASL server* is the device running the Cloud ASL code. This can be one of the Cloud ASL nodes.
  - The *Cloud ASL device* is a device or devices Cloud ASL is interacting with and hosts Cloud ASL components.

- Software elements (shown as UML components)
  - The *Cloud ASL* is an executable code that manages Cloud ASL devices. It communicates with the Cloud through SOAP, the Cloud ASL device through SSH and the OSGi through telnet.
  - *Cloud ASL device*. This is the cloud device that hosts OSGi.
  - *OSGi*. This is the OSGi framework which hosts Cloud ASL components.
  - *Cloud API*. This is the web service that operates the cloud instances and is responsible for creating, destroying, starting and stopping cloud devices.

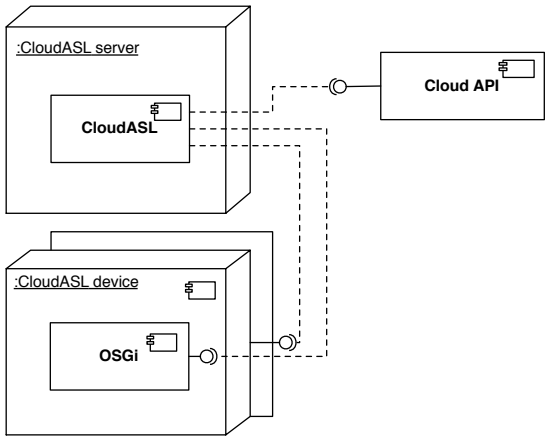

Figure 3.10.: Deployment view of Cloud ASL

3.2.3. Example in Use

The web messenger example will be used again in this section, but because our implementation requires that interfaces be stored in a special library component, that component will be added to this example. In listing 3.2 we create a device, install and start required components to run our web messenger example.

```
//creates a small instance with the variable name ''server''
Device server = create_instance_device("m1.small")
//where our implementation devices start on creation we do not need to
    start it.
//next we install our components on the server
Component libCmp = install_component( server, "http://URI/library.jar" )
Component uiCmp = install_component( server, "http://URI/ui.jar" )
Component messCmp = install_component( server, "http://URI/mess.jar" )
Component dbCmp = install_component( server, "http://URI/db.jar" )
```

```
//now all the components have been installd on the device, we can start
    them one by one
start_component( libCmp )
start_component( uiCmp )
start_component( messCmp )
start_component( dbCmp )
```

Listing 3.2: Example of Cloud ASL implementation

3.3. Binding Cloud ASL to a Cloud

When thinking about binding ASL to clouds, some decisions had to be made. First is the type of cloud computing to use. The choice was between IaaS or PaaS, but because Google App Engine is the only PaaS provider that supports Java and it does not support threading nor our basic model of interchangeable executable components, IaaS was chosen. For IaaS models, private and public clouds could be used, but as there was no access to community or hybrid clouds, they were out of question. At the beginning of this project, access was granted to hardware from the University of Iceland and it was decided to set up our own private cloud based on the Eucalyptus cloud computing infrastructure software.

3.3.1. Eucalyptus

In early October 2009 work began on setting up the Eucalyptus open source cloud computing infrastructure on the University of Iceland computer grid. However, due to many unforeseen problems this work was not successful. At first we had problems getting the Eucalyptus infrastructure to work with the networking environment, mainly the main DHCP server at the University of Iceland. When this problem was solved we could not carry on due to lack of access to administrators. After unsuccessful attempts at getting access to hardware from other sources we decided to use a public infrastructure.

3.3.2. Amazon Web Services

On deciding on what public infrastructure to use, we decided to build our prototype usable with multiple IaaS services. A beta version of Jclouds was used[7] as a cloud layer initially, but after few tries in getting it to work for multiple cloud services, it was found

[7]JClouds is an open source framework based on Java to manage multiple cloud computing services. http://www.jclouds.org

not to be mature and functional enough for our use and therefore Amazons SDK was chosen. Amazon granted us $100 funding to use with the Amazon Web Services, which gave us opportunity to use the Amazon cloud as we needed.

First an Amazon Instance Image (AMI) was created, which is a virtual image containing the operating system (32 bit Ubuntu 9.10) partition, a swap partition and data partition, and the AMI was stored on Amazon's instance store. On that virtual image an Open-JDK was installed, which is an open source community-maintained Java implementation and Apache Felix, a OSGi implementation and necessary OSGi bundles/ASL components to start with the ASL implementation.

The cloud ASL implementation in our case implements cloud operations in the following way:

start_instance_device starts by requesting a new instance with the AMI pre-installed. After the request has been sent, it can take a couple of minutes for the instance to start. When the instance has started, JSCH[8] is used to run a boot-up script which installs missing required bundles, downloads required components for the prototype software and starts the OSGi platform. This can all be done beforehand with a customized instance image, but creating a new AMI for each application is time consuming whereas setting a boot-up script takes relatively short time.

start_jvm_device requires a running device with OSGi setup. We connect to that machine via JSCH and copy the already running implementation of OSGi to a new location and start the OSGi framework from there.

start_device and stop_device : As Amazon's EC2 does not enable stopping or starting instances stored in the instance store, these operations were skipped. If the AMI would be transferred to the Amazon's EBS these features could be easily enabled.

restart_device : As restarting instances is a feature in Amazon EC2, it was used instead of starting and stopping devices.

3.4. An Experiment With Cloud ASL

When Cloud ASL is defined and set up it has to be possible to test it in a real world scenario. Therefore a software system had to be created, capable of running on a cloud computing infrastructure that has something to gain from Cloud ASL. That is, the software system has to be able to run on multiple computing instances at the same time and

[8] JSch is a pure Java implementation of SSH2, allowing connecting and file transfer to an sshd server with Java programs. http://www.jcraft.com/jsch/

needs to be scalable. For this work the architectural design process from figure 2.3 is used. Each part of that figure is a topic of a sub section where individual design steps are show for our software. Thus ending up with an architectural prototype. This prototype or software was named “The Turnip” and that name will be used from now on.

As the diagram in figure 2.3 displays, the first step is to gather architectural requirements, see section 3.4.1, which are then evaluated and become the basis of architectural design, section 3.4.2. From these designs architectural descriptions are created, 3.4.3. The evaluation is a part of chapter 4. The resulting system is the topic of section 3.4.4

3.4.1. Architectural Requirements

To evaluate requirements of a software system, two types of architectural descriptions are significant, functional and quality based requirements as recommended by Christensen et al. [22]. The functional requirements can be described in the form of use cases and architectural quality in form of quality attribute scenarios.

Functional Requirements

As the main function of our software prototype have not been revealed, the starting point will be focusing on underlying requirements that Cloud ASL and our project require. The approach of Christensen et al. [22] will be used, by creating architecturally significant scenarios containing a subset of the overall scenarios providing the functional requirements for the system.

In short these requirements are: Cloud ASL support, cloud connectivity and modularity. As a scenario for our experiment it was decided to implement a distributed ray tracing application. One which has the advantage of being time and computing intensive, is visual in the case that a ray tracing job results in a resulting image, can easily be broken down from one big job into multiple smaller jobs, and might interest third-party individuals. The disadvantages of using ray tracing are that there are not many active open source implementations, ray tracing is a complex process and might be hard to distribute and individuals that have little knowledge of computer graphics might not be familiar with the concept.

Use Cases

As a developer I can change the runtime architecture through ASL scripts.

As a user I can create ray-tracing jobs and choose on how many computing instances they run at a time, and when a job is finished I can save the results of the ray tracing job.

As a user I can shut down running computing instances after I have finished working with them.

As a user I would like to be able to monitor the result of currently running ray tracing job.

Quality requirements

Our architectural requirements will be described in this section, with a set of significant quality attribute scenarios. The goal of describing quality requirements is to help with construction of "test cases", where architectural quality attributes may be compared and evaluated, see section 2.2.1.

Driving architectural attributes for our experiment are:

1. Modifiability. The developer shall be able to change the system's architecture on runtime.

2. Scalability. The system shall be able to perform satisfactorily from one to hundreds of computing instances and shall be able to scale both down and up.

3. Performance. The system shall be able to perform as satisfactorily as a a non Cloud ASL based system.

Here, a quality attribute will be used that is not part the main quality attributes of Bass et al. [1]. Our definition on a scalability scenario generation framework is shown as Bass et al. generic scenario:

Scenario type:		Scalability
Scenario Parts	**Source:**	Someone requesting more/less computing power
	Stimulus:	The request of modifying system capacity, for example increasing the number of supported users.
	Artifact	Computing instances, computing monitoring framework or something similar.
	Environment:	Runtime or development for pre-emptive measures.
	Response:	What has been done to meet with the change of scalability, for example there have been more cloud computing instances deployed.
	Response Measure:	Increased/reduced number of computing instances, more/less disk space given or increased network throughput.

Table 3.6.: Scalability quality attribute

Here availability, security, testability and usability are left out. The reasons why those are not thought to be a main quality attribute requirement in our case are:

Availability Although availability is an important factor for cloud computing services, we did not include it for our project where we did not find it important for an architectural prototype. Where our project is not running as a service it can be started or restarted for every usage.

Security As for availability, security is an important attribute in cloud computing, but our project is not intended for real users and including security would increase the complexity and difficulty of the project greatly.

Testability Making a distributed and cloud enabled architectural prototype which is running a foreign domain specific language, testable in a execution-based environment could be a whole new thesis on it is own, and therefore not in the scope of this project.

Usability As the goal of this experiment is not to create an easy to use application, but to create an case study to use our Cloud ASL implementation, Usability was not defined as a required quality attribute.

As architectural operations are about dynamic architectural changes, the modifiability attribute is the biggest factor in our quality requirements because our ASL implementa-

tion consists of the cloud scalability and performance.

3.4.2. Architectural Design

Now that the requirements are established, a better look can be taken at the architectural design and decisions made regarding the design. Architectural design can be done as an architectural prototype, attribute-driven design, architectural patterns and styles.

For architectural prototype decisions several choices are available, see section 2.2.3. The prototype is the experimental type, which means that only one prototype had to be made to evaluate whether our architectural decisions are valid. The characteristics of the prototype aimed for are exploratory and learning and quality attributes. For exploration and learning the goal is to learn about how our prototype works for Cloud ASL and how architecture can or should be made. As quality attributes are big motivations of this prototype and help us with attribute-driven design, this is a characteristic that can help.

The architectural pattern we are using is the client and server pattern.

3.4.3. Architectural Description

For the architectural description module viewpoint, component and connector viewpoint and allocation viewpoint were used as referenced in Christensen et al. [22]. We start by describing the C&C viewpoint.

Component and Connectors Viewpoint The experiment has five major functional and three assisting functional parts, or components, as shown in the C&C view in figure 3.11, which are presented as UML active objects. Connectors are presented by links with association names and possibly role names. As this figure does not explain the C&C view on its own, descriptions of components functionalities are provided in term of responsibilities:

- *User Interface* is responsible for 1) giving the user available functionalities and 2) giving the user information about the status of rendering jobs.

- *Request Manager* is responsible for 1) managing requests from the user; 2) working as a server in the software layout and being the communication layer between other critical components where needed and 3) being the front-end for the user interface.

- *Worker Factory* is responsible for managing workers by 1) creating and destroying cloud computing instances (devices) through ASL and 2) managing workers services, by registering distributed services on new instances and unregistering on removed devices.

- *ASL* is responsible 1) for creating and destroying cloud computing instances (devices) through Cloud API and 2) installing, starting, stopping, updating, stopping and uninstalling components on devices. This is Cloud ASL from section 3.2.2

- *Cloud API* is responsible for 1) being the interface and connector to the cloud; 2) Creating and destroying cloud instances and 3) being as independent against specific cloud provider as possible. This is the package Cloud from section 3.2.2

- *Worker* is responsible for 1) doing requested work; 2) being independent from other workers and 3) being fault tolerant.

- *Sunflow* is responsible for 1) rendering images and 2) exposing basic interfaces and methods for the rendering process.

- *Web Browser* is responsible for 1) presenting the User Interface and 2) delivering the functionalities needed for the user interface to communicate with the request manager.

The connectors also have to be described in more detail:

- *r-OSGi* provides a transparent way to access services on remote OSGi platforms. It uses a very efficient network protocol and has a small footprint [30].

- *AJAX* Asynchronous JavaScript and XML, is a web standard for making clients (web browsers) communicate with servers.

- *SSH* or Secure Shell is a network protocol that allows data to be exchanged using a secure channel between two networked devices.

- *Telnet* is a network protocol that provides a bidirectional interactive text-oriented communications facility via a virtual terminal connection.

- *Java* method calls are used to communicate between classes, components and libraries.

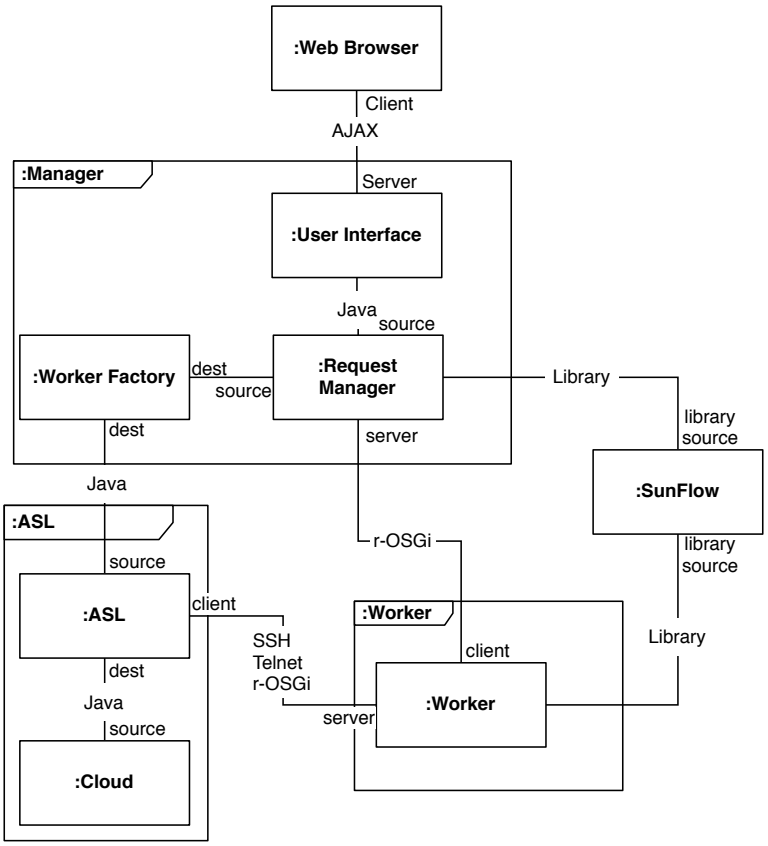

Figure 3.11.: C&C view of the architectural prototype

The interaction between a set of connectors is not revealed in a single sequence diagram. Therefore a set of examples is presented, that show the interaction between connectors in concrete contexts. In figure 3.12 an example of worker creation is shown and in figure 3.13 the rendering process is displayed.

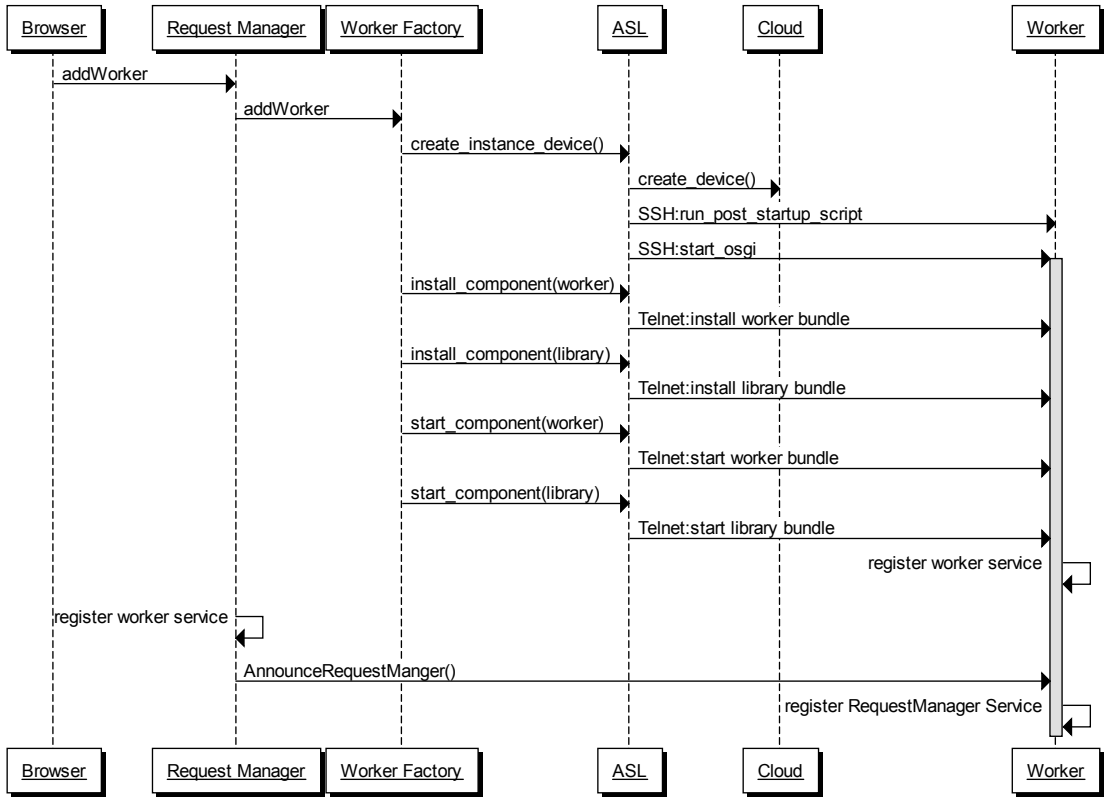

Figure 3.12.: C&C sequence diagram displaying a new worker added

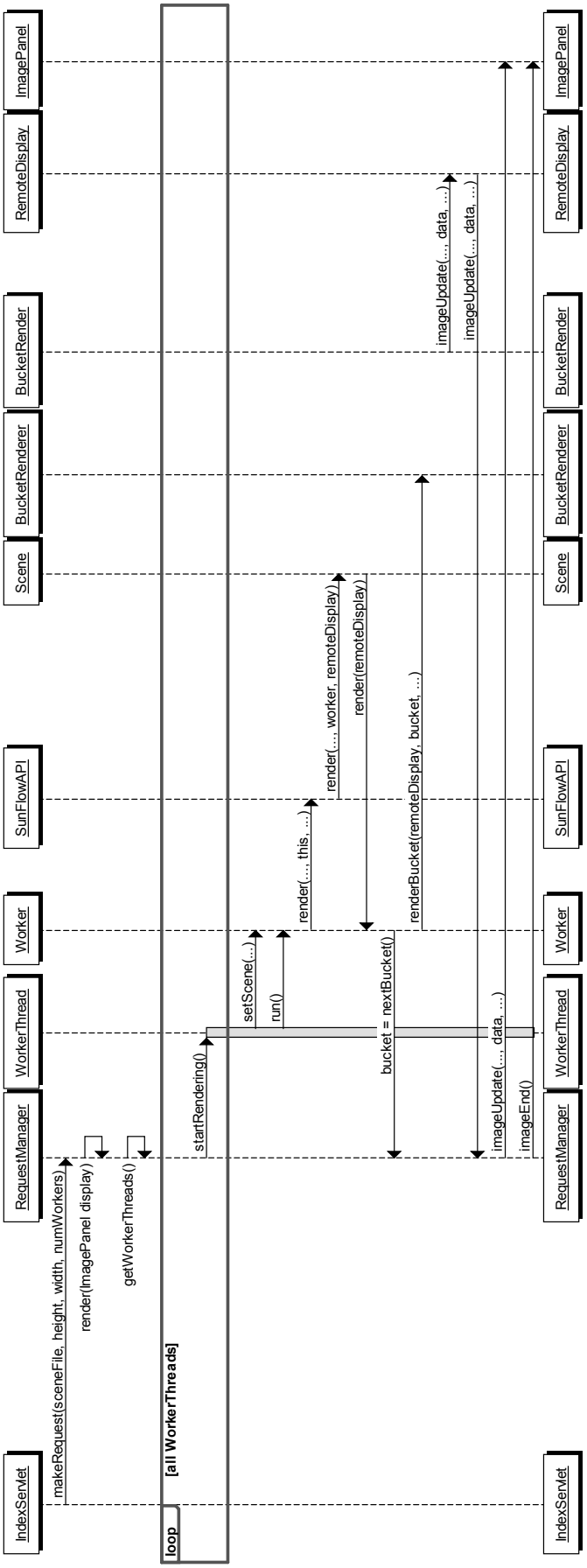

Figure 3.13.: C&C sequence diagram displaying the rendering process

Module View. The module view of our architecture will be described from top-down starting with the most top-level diagram, beginning with the package overview on figure 3.14. Then, a better look at individual packages will be taken, starting with the user interface, figure 3.15; the worker factory, figure 3.16; the request manager, figure 3.17; Cloud ASL, figure 3.18 and ending on the worker, figure 3.19.

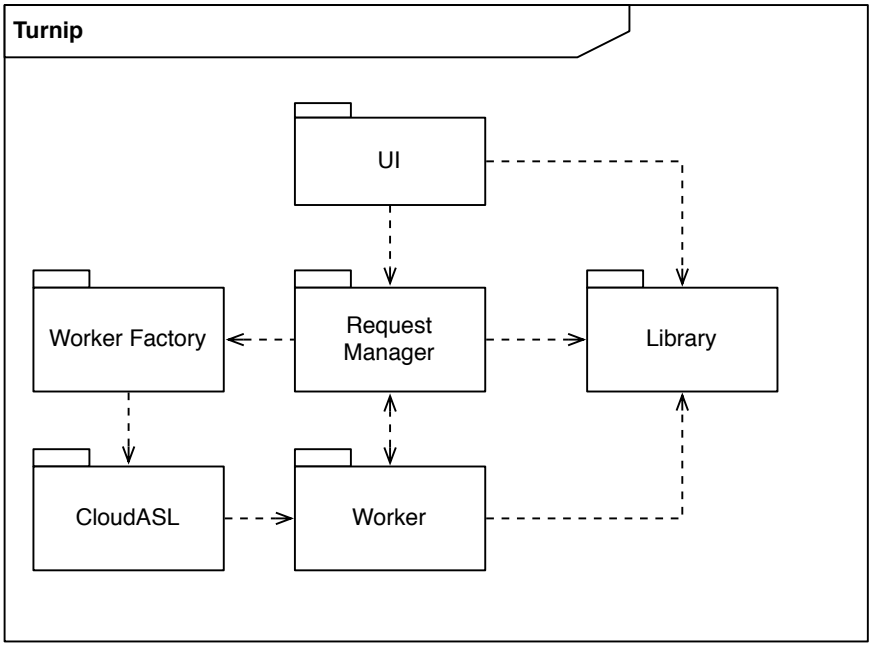

Figure 3.14.: Package overview for the architectural prototype

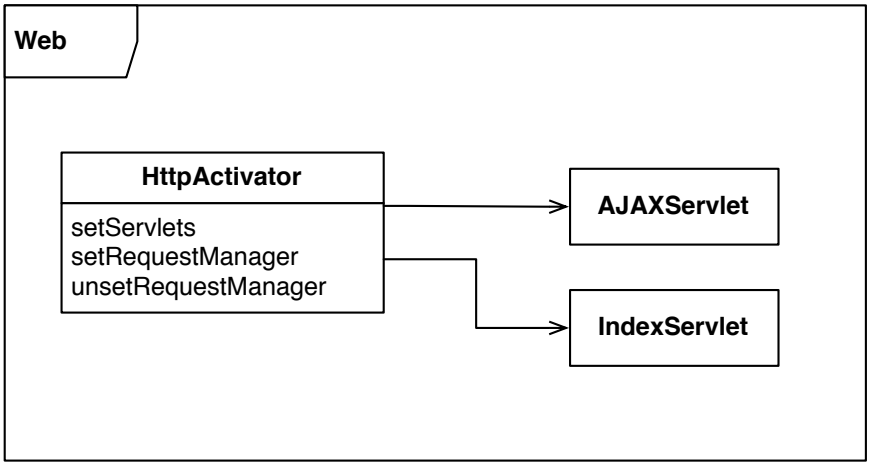

Figure 3.15.: Interface overview for the user interface

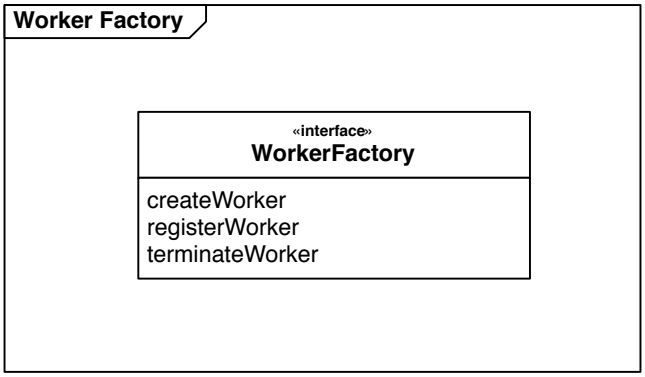

Figure 3.16.: Interface overview for the worker factory

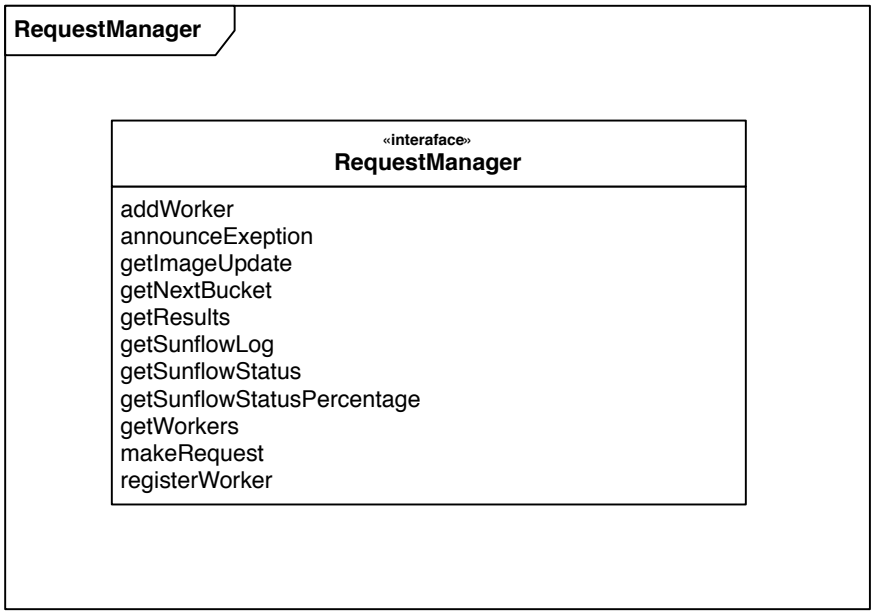

Figure 3.17.: Interface overview for the request manager

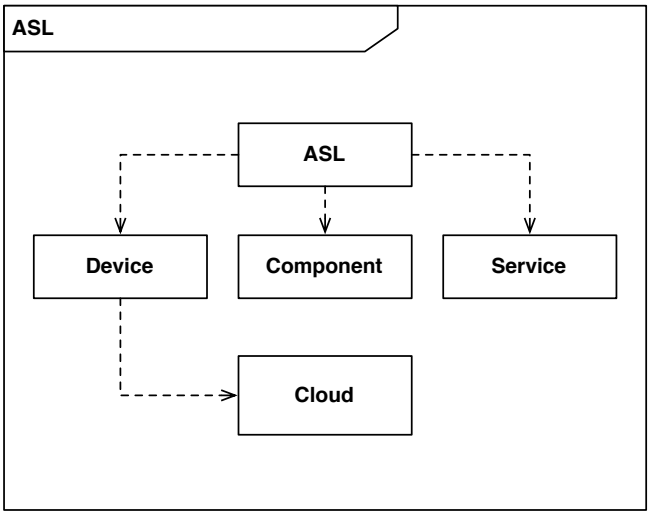

Figure 3.18.: Interface overview for Cloud ASL

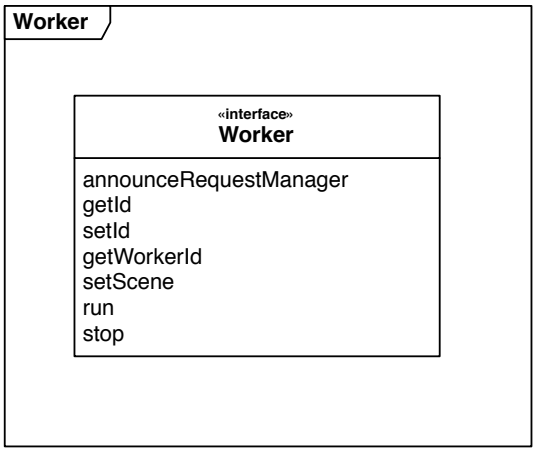

Figure 3.19.: Interface overview for the worker

Deployment View Figure 3.20 shows the deployment of the architectural prototype. The deployment is based on the client-server architectural pattern. The following elements are of interest:

- Environmental elements (shown as UML nodes)
  - The *Manager* is the server in the client-server pattern and acts as job distributer for the software.

  - The *Worker* is the client in our client-server pattern and acts as a job processor for the software.

- Software elements (shown as UML components)

  - The *Request manager* is the server in the client-server pattern and acts as job distributer for the software.

  - *ASL*. See figure 3.18. This is our ASL implementation from section 3.2.

3.4.4. The Turnip

The experiment ended with a working prototype, that got the working name "the Turnip". The Turnip is a distributed ray-tracing software built for cloud computing usage. It is made to be used with Cloud ASL and utilizes it for cloud operations, mainly for creating and destroying workers.

Setup The overall setup of the Turnip starts with two types of software deployments, "manager" and "worker", which behave in a client-server architecture where the manager acts as a server and the worker acts as client. The manager is responsible for interacting with the user, managing the workers, creating and destroying workers and distributing work. The worker's only responsibility is to process the work from the manager.

The system is made from several components, which are:

User Interface
The UI is web based stateless terminal, i.e. it does not include any concrete logic. It uses the Request Manager for all communication and business logic. See the User Interface paragraph below. This component exists on the manager.

Request Manager
The Request Manager is the heart of the manager. It is used by the UI and communicates to other components/servers for results. For example if the user wants to add more workers, the Request Manager calls the worker factory to add or remove workers. If the user adds a new job, the Request Manager takes the job and splits it into buckets and activates the worker for that job. This component exists on the manager.

Worker Factory
The worker is responsible for taking care of the worker instances. It uses Cloud ASL

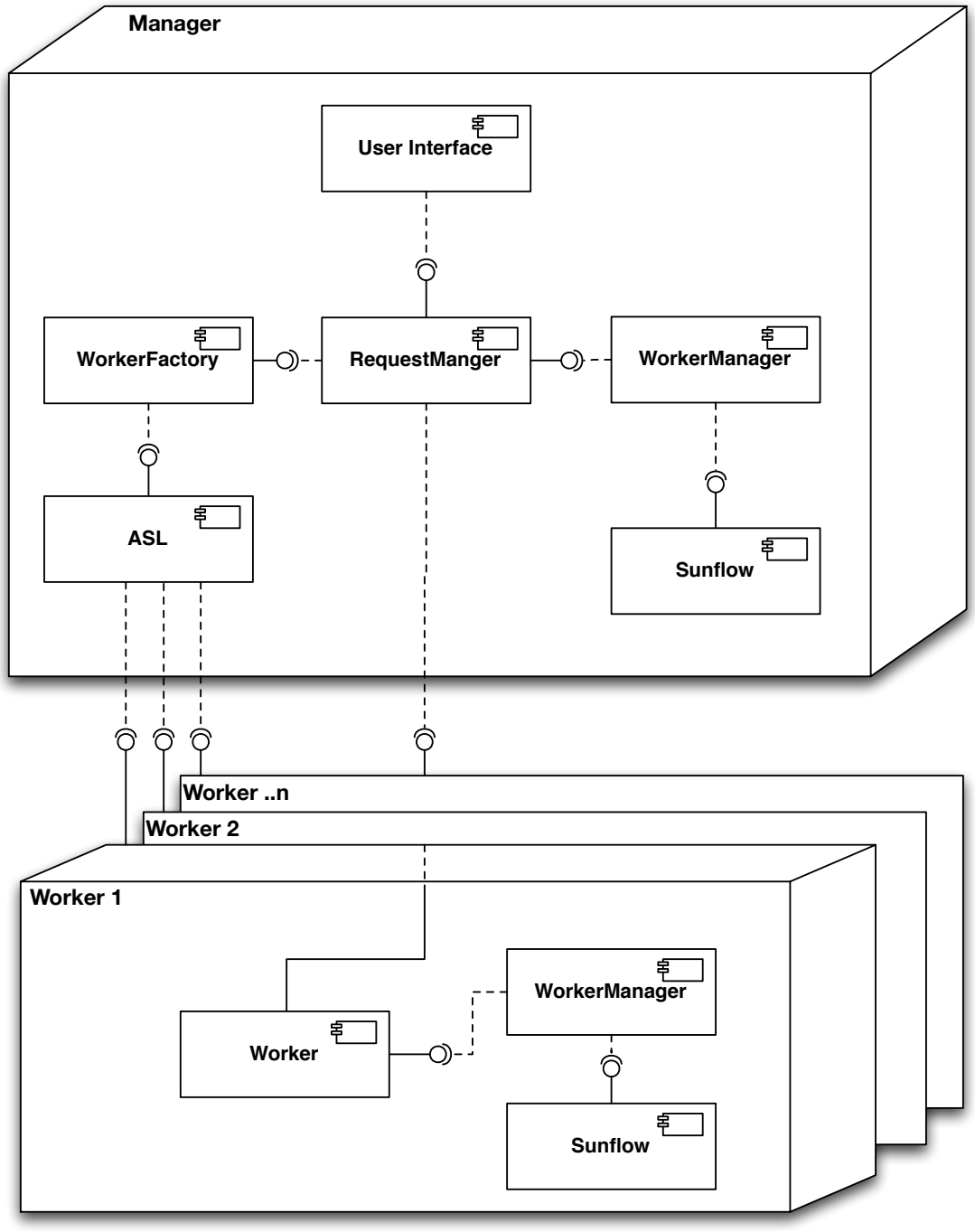

Figure 3.20.: Deployment diagram for the Turnip

to create or destroy instances and r-OSGi to register worker instances as a service. This component exists on the manager.

Library

The Library is one of the critical parts of Cloud ASL and the Turnip. The Library holds all interfaces for Cloud ASL. Services are registered as interfaces through the Library. The Library also holds any external libraries used by the system, like Sunflow for the rendering. The Library exists on both manager and worker.

ASL

This is our ASL implementation, Cloud ASL. It takes care of devices (creating, terminating, starting and stopping) and components (installing, uninstalling, starting and stopping). This component exists on the manager.

Worker

The worker is responsible for processing the jobs. The worker communicates with the Request Manager, through r-OSGi, and pulls new buckets to render from there. It utilizes Sunflow from the library component to render active bucket. When finished rendering, it sends the bucket back to the Request Manager as an array of colors.

Ray Tracing The ray-tracing software used as a basis for the Turnip is Sunflow[9], which needed some modification to be able render on a distributed system. Helios[10] was chosen in this project as a model for Sunflow job distribution. An example of Sunflow rendered image can be seen in figure 3.21.

A rendering job consists of a scene file and textures. The scene file defines objects in the scene, such as a ball model with a texture here, a teapot model there with other texture etc., and properties, such as camera location and direction and light sources. The texture files are plain images which are applied to objects giving them a texture.

Rendering jobs are rendered with a bucket rendering technique. This means that each job is split into multiple smaller jobs or buckets, and each of these buckets is a small part of the resulting image. For example, if one decides to render a 100 pixel wide and 100 pixel tall image, that image could be split into 100 smaller images where each image is 10x10 pixel width/height. This rendering method is convenient for distribution.

Sunflow was leveraged by including the class files in the library component. This makes sure that Sunflow is always available on all workers and the manager, however this makes updates of Sunflow more complex than necessary. The rendering work-flow starts by the user giving the Turnip a job to process. This means that the UI sends a message to the

[9] Sunflow is an open source rendering system for photo-realistic image synthesis. It is written in Java and built around a flexible ray tracing core and an extensible object-oriented design. http://sunflow.sourceforge.net/

[10] Helios is an distributed rendering system based on Sunflow and made for grid systems. http://sfgrid.geneome.net/

Figure 3.21.: Example of Sunflow rendered image

worker manager, which then analysis the job, extracts information such as job complexity and image size, and then creates image buckets for the job at hand. When this process is over the worker manager sends a signal to the workers, through r-OSGi connection, to start rendering. Each worker interacts with the request manager by requesting a bucket to render and then sends a part of the image to the request manager when it has been rendered. The request manager arranges the image parts to a result image based on the bucket location. The current status of the image is regularly saved so that the user can see the current status of the job. This process can be seen in figure 3.13.

Cloud operations The Turnip uses Cloud ASL as its basis for cloud computing operations. The cloud operations that the Turnip uses are starting one cloud computing instance or terminating a cloud computing instance. This is done through the UI, where a user starts a new rendering job or adds a new worker to the pool of running workers.

When the user adds a new worker, the UI sends a request to the worker manager, which then runs a set of ASL operations; which starts a cloud instance and installs the components required for the worker. The list of operations are shown at listing 3.3. The worker factory starts a thread that waits for the new worker to come on-line. When it is

on-line it is registered through r-OSGi as a new worker service, which then again makes the worker register the request manager as a service. This process can be viewed in figure 3.12.

```
1 Device device = asl.create_instance_device("m1.small");
  Component c1 = asl.install_component(device, "http://bjolfur.com/
      turnip_library.jar");
3 Component c2 = asl.install_component(device, "http://bjolfur.com/remote
      -1.0.0.RC4.jar");
  Component c3 = asl.install_component(device, "http://bjolfur.com/com.
      springsource.org.codehaus.janino-2.5.15.jar");
5 Component c4 = asl.install_component(device, "http://bjolfur.com/
      turnip_worker.jar");

7 asl.start_component(c1);
  asl.start_component(c2);
9 asl.start_component(c3);
  asl.start_component(c4);
```

Listing 3.3: Start worker Cloud ASL script as used by worker manager

The User Interface The User Interface(see figure 3.22) of the Turnip was designed to

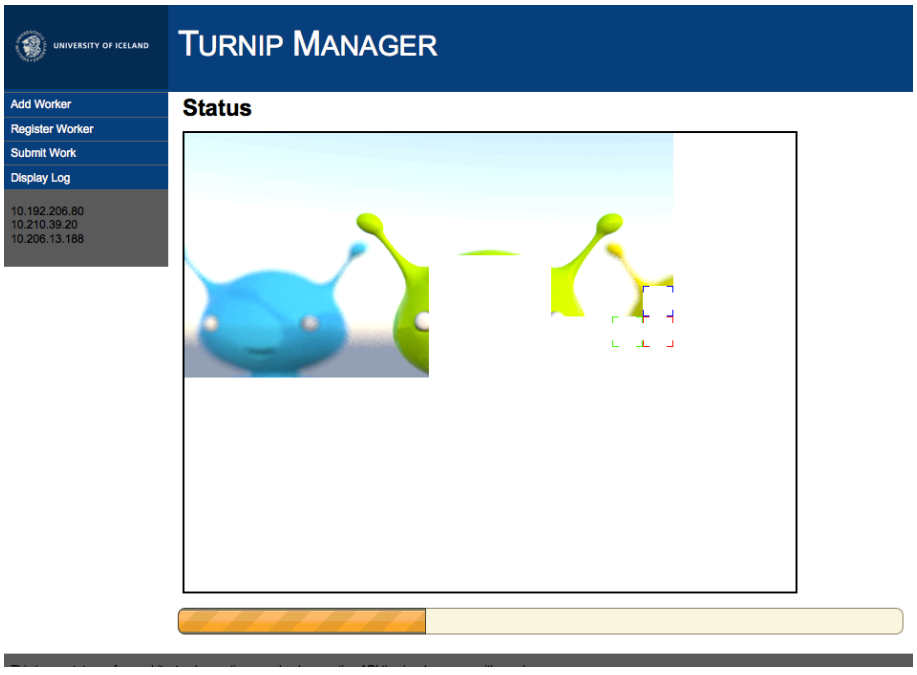

Figure 3.22.: Screenshot of the Manager UI

be simple but functional enough to do all required work in a "web 2.0" way. That is, it is a simple HTML page, created and registered as a Java servlet through the OSGi HTTP

admin. All functionality of the web page is provided through AJAX[11] interfaces which include updating the status of a working job, adding workers, and viewing the Log. The status update is done by the request manager call, which takes the current status of the rendered image and saves it over the original image. This way the user can set the interval of updates for the status.

[11] Asynchronous JavaScript and XML

4. Evaluation

After designing and implementing Cloud ASL and our prototype, the results can be evaluated and the system can be tested by running real jobs and observing how it performs. Firstly a qualitative evaluation will be carried out to measure how the architecture stands. Secondly, the system will have to perform a few tasks and its performance is measured.

4.1. Qualitative Evaluation

In the work done for this thesis, Cloud ASL has been created. It is a system to modify software architecture through architectural operations, a prototype was also created to utilize Cloud ASL. The question that now can be raised is, are these systems usable and have they solved the set requirements? In this section we are going to answer this question by evaluating the architecture by first utility and completeness and secondly quality attributes

4.1.1. Utility and Completeness

Utility denotes the applied use of the system for the goal of modelling and managing runtime architecture of cloud computing applications, i.e. does the system enable others to make dynamic software on cloud computing infrastructure and manage it with architectural operations, and if so how difficult is it? Completeness is evaluated in the terms of the requirements of the system. The system should have resolved the requirements presented in 3.4.1. The Turnip was created to demonstrate and argue for the utility of the system.

Utility Utility is determined in terms of how the system, Cloud ASL, allows other developers to build dynamic software on clouds and manage it by architectural scripting. The main parts of Cloud ASL have been implemented, and this process is the main foundation for discussing utility. Cloud ASL has not been used or tested by other software

developers, which might be necessary to get a good picture of its utility. Now the utility will be examined from views other than direct usage of others, a developer scenario will be described and Cloud ASL compared to alternative ways of developing and modelling architectural change.

Developer scenario: A developer is making a distributed imaging rendering application to be used on cloud instances. He/she uses Helios, a Jini[1] based Sunflow rendering application system.

The developer creates two separate Java applications, one to manage the cloud environment and other to be able to manage Helios/Sunflow through a web interface. This is a similar approach as in the prototype, but Cloud ASL was used to manage the cloud.

When the developer has finished the implementation and has the system running, he/she needs to update the software, e.g. upgrade Jini to a newer version. What the developer would do in this case is to shutdown all the virtual instances, through the cloud managing software, update the code on the virtual machine template and start with fresh instances and updated code. Another, less intrusive but more complex way would be to update the code on all running instances, which would then again require shutdown of the running code. A better long term solution would be to create a configuration script or application to do this automatically, which is again the aim of Cloud ASL.

As this specific scenario shows, Cloud ASL can have an advantage compared to no configuration aimed solution.

Completeness To determine if the architecture for this system is complete, two questions need to be answered. One, does the architecture satisfy all of the requirements identified in section 3.4.1? Second, is it possible to build cloud computing software using this implementation of Cloud ASL?

At first we should examine the functional requirements for the architectural prototype from section 3.4.1.In short these requirements are:

> Cloud ASL support, cloud connectivity and modularity.

As these requirements were the basis of the design they were followed closely. The prototype was built on top of Cloud ASL and is closely integrated to it, which again enables cloud connectivity by default. Because developing for Cloud ASL forces modular development if done correctly, this requirement is accepted. Taking a closer look at these requirements by analysing the use cases from section 3.4.1 a better picture can be painted

[1] Jini is a network architecture for the construction of distributed systems in the form of modular cooperating services. http://jini.org

of how these requirement are met.

As a developer I can change the runtime architecture through ASL scripts:
As discussed earlier, the prototype was closely integrated with Cloud ASL, so all Cloud ASL functionality was therefore enabled for this project. Tests were done on changing runtime architecture on the code with successful results.

As a user I can create ray-tracing jobs and choose on how many computing instances they run at a time, and when a job is finished I can save the results of the ray tracing job.
If we take a look at this use case in a literal context the requirements are met, but there are few things that could have been done better. Currently the job selection mechanism is hard-coded into the program, so the end user can not easily select or modify a scene to render. This can however be done through Cloud ASL but a better solution would be to distribute the rendering job through a file based distribution mechanism, such as Amazon S3 in case of AWS, but there does not seem to be a standard distributed storage with cloud providers. The final result of a rendering process is a picture which is easily stored. This result is not stored in the cloud but should have been stored with the rendering job, which would be simple if the distributed file based mechanism would have been implemented.

As a user I can shut down running computing instances after I have finished working with them.
This feature was not fully integrated with the user interface, although it was fully completed and tested for Cloud ASL scripts.

As a user I want to be able to monitor the result of the currently running ray tracing job.
This use case was completed successfully. With a certain user specified interval a picture with the current status of the rendering job, see figure 3.22, is presented displaying which buckets are in process with workers and the log from the manager can be viewed with a click of a mouse.

After we created the first version of the prototype a list was made of possible changes that could or should be done that might improve the prototype, and this list became the basis of quality attribute scenarios. These scenarios will be discussed in the following section.

4.1.2. Quality Attribute Scenarios

To describe and measure the quality of Cloud ASL, we made a list of possible change scenarios early in the implementation phase, based on quality attribute scenarios for the

Turnip. These scenarios describe, for example, the change that has to be done to achieve certain quality requirements, improved features or some other architecture change. Secondly, these scenarios were evaluated in regards to whether Cloud ASL could support them and lastly it was concluded how these changes would be implemented through Cloud ASL scripts if it could support them.

These scenarios are evaluated by:

1. Can the scenario be supported (through Cloud ASL) and how?

2. Is the scenario important?

3. How should the architecture be changed to support the change scenario?

These scenarios were evaluated, and based on the outcome Cloud ASL scripts were created.

Evaluation of Scenario Attributes The evaluation of scenarios needs to be viewed on the basis of which of them can be supported by the implementation, which of them are not supported by the implementation and which of them do not fall into our focus and are therefore not important for this evaluation. Of those scenarios that are not supported it has to be examined why they are not supported and whether this project can be altered in some way to enable support for those scenarios.

Unsupported scenarios:
Scenarios that can not be supported by Cloud ASL and the prototype are:

1. File-system storage support, table 4.1.

2. Change of cloud layer from IaaS to PaaS, table 4.2.

3. Change of OSGi platform, from example Apache Felix to Equinox, table 4.3.

Here each scenario will be discussed in more detail.

File-system storage support:
In this case, a change on the operating system level and even on the cloud instance image would be necessary, therefore adding this scenario would require more than a software architecture change to work.

Scenario(s):		A developer wants to add a storage feature to store result images in a file system.
Relevant Quality Attributes:		Modifiability
Scenario Parts	**Source:**	developer
	Stimulus:	Wants to add storage functionality
	Artifact	System
	Environment:	Development
	Response:	Filesystem image store is added
	Response Measure:	5 Hours of development time and only storing component affected
Priority:		Medium
Difficulty:		Medium

Table 4.1.: Storage Features - File System

Change of cloud layer from IaaS to PaaS:
Changing a cloud computing layer is a very complex task. Because Google App Engine does not support OSGi or threads, Cloud ASL would not directly fit this kind of model and therefore this scenario is not supported.

Scenario(s):		A developer wants to change runntime platform from eucalyptus to Google App Engine (Java based PaaS)
Relevant Quality Attributes:		Modifiability, Availability
Scenario Parts	**Source:**	Developer
	Stimulus:	Wishes to change cloud platform
	Artifact	Code, System, Infrasturcture
	Environment:	Development
	Response:	Turnip is running on AppEngine
	Response Measure:	
Priority:		Low
Difficulty:		High

Table 4.2.: Eucalyptus -> AppEngine

Change of OSGi platform:
Changing OSGi platforms is theoretically possible, but the implementation of Cloud ASL

does not support it, because Cloud ASL uses a Apache Felix specific remote shell to interact with the OSGi framework, our virtual image has pre-installed Apache Felix, and changing OSGi platform would require modification on the virtual image thus making it too complicated for this scenario.

Scenario(s):		As a developer I want to change OSGi platfrom from Felix to Equinox
Relevant Quality Attributes:		Modifiability
Scenario Parts	**Source:**	Developer
	Stimulus:	wants to be able to change OSGi platform
	Artifact	Code
	Environment:	Development
	Response:	Turnip is running on Equinox
	Response Measure:	Some development time
Priority:		Low
Difficulty:		Medium

Table 4.3.: Felix -> Equinox

Unimportant scenarios:

For each scenario two importance factors were made, priority and difficulty, and they were measured from low to high. Scenarios were thought as important if importance was more than 4 where $importance = priority + difficulty$ and high difficulty = 1, medium = 2 and low = 3 and high difficulty = 3, medium = 2 and low = 1. If a scenario was thought to be both difficult and with low priority they were marked to be out of the scope of the project. Scenarios that were not deemed important enough for the project were:

1. Adding a Database based storage feature, table 4.4.

2. Distributed request manager, table 4.5.

3. Changing the underlying tracing platform to Pov-Ray, table 4.6.

Here, each scenario will be discussed in more detail.

Adding Database based storage feature:
Adding a database storage feature would require standard available distributed storage or some sort of embedded database, which would then require a storage feature which

is not currently in our scope.

Scenario(s):		A developer wants to add a storage feature to store images, job files, a program status and other information in a DataBase.
Relevant Quality Attributes:		Modifiability, Usability
Scenario Parts	**Source:**	Developer
	Stimulus:	wants to add a database storage functionality
	Artifact	Code, System
	Environment:	Development
	Response:	Program status, resulting images and other information is stored in a DB
	Response Measure:	
Priority:		Medium
Difficulty:		High

Table 4.4.: Storage Features - DB

Distributed request manager:
Distributing the request manager is not supported by the architecture design and implementing that would require a change of most of our components and therefore too difficult for our scope.

Scenario(s):		A developer wants to be able to run multiple instances of the request manager on multiple nodes
Relevant Quality Attributes:		Modifiability
Scenario Parts	**Source:**	Developer
	Stimulus:	wants to be able to run multiple instances of the factory
	Artifact	Code
	Environment:	Development
	Response:	Multiple factorys are running simultaneously running the same task
	Response Measure:	Medium development time
Priority:		Low
Difficulty:		Medium

Table 4.5.: Dynamic factory

Changing the underlying ray-tracing platform to Pov-Ray:
Changing the ray-tracing platform to Pov-Ray included moving the process from a Java module to a compiled C program. Because OSGi is a Java platform a new method to bundle and load/unload external libraries on demand would be needed, and as there is no simple way to do this, this scenario is too complicated to support.

Scenario(s):		As a developer I want to use Pov-Ray as a ray-tracer instead of Sunflow
Relevant Quality Attributes:		Modifiability
Scenario Parts	**Source:**	Developer
	Stimulus:	Wants to use Pov-Ray for ray-tracing
	Artifact	Code
	Environment:	Development
	Response:	Turnip is running PovRay as a ray-tracer
	Response Measure:	Some development time, faster ray-tracing
Priority:		Low
Difficulty:		High

Table 4.6.: Different raytracing program

Supported Scenarios: The rest of the scenarios are supported by Cloud ASL, they are:

1. Start new cloud instances on demand, table 4.7.
2. Shutdown cloud provider on demand, table 4.8.
3. Turnip upgrade, table 4.9.
4. Update the User Interface, table 4.10.
5. Change the User Interface, table 4.11.
6. Use BOINC for processing, table 4.12.
7. Updating r-OSGi, table 4.13.
8. Change r-OSGi to other distributed OSGi framework, table 4.14.
9. Migrate from Amazon EC2 public cloud to Eucalyptus private cloud, table 4.15.
10. Change cloud providers, table 4.16.

For each of these scenarios, the original quality attribute scenario and the Cloud ASL script supporting it will be listed and the scenario discussed.

Start new cloud instances on demand:
This scenario is a basic requirement of Cloud ASL and has been tested thoroughly and is used in the prototype to create workers. In table 4.7 our quality attribute scenario is displayed and in listing 4.1 we start a new cloud instance and install and run required components.

Scenario(s):		As an operator I want to be able to start new instances of workers on demand
Relevant Quality Attributes:		Modifiability
Scenario Parts	**Source:**	Operator
	Stimulus:	Wants to be able to increase the numbers of workers running
	Artifact	Code
	Environment:	Development
	Response:	Turnip can now start new instances of workers on demand
	Response Measure:	Some development time, number of instances should be in consistent with workers.
Priority:		High
Difficulty:		Medium

Table 4.7.: Startup of instances

```
Device device = asl.create_instance_device("m1.small");
Component c1 = asl.install_component(device, "http://bjolfur.com/
    turnip_library.jar");
Component c2 = asl.install_component(device,"http://bjolfur.com/remote
    -1.0.0.RC4.jar");
Component c3 = asl.install_component(device,"http://bjolfur.com/com.
    springsource.org.codehaus.janino-2.5.15.jar");
Component c4 = asl.install_component(device,"http://bjolfur.com/
    turnip_worker.jar");

asl.start_component(c1);
asl.start_component(c2);
asl.start_component(c3);
asl.start_component(c4);
```

Listing 4.1: Start new cloud instance Cloud ASL script

Shutdown of instances:
Although this scenario was tested and implemented in the project, it did not make it into the final version of the user interface. In table 4.8 our quality attribute scenario is displayed and in listing 4.2 we destroy a given device.

Scenario(s):		As a developer I want to be able to run shut down instances and therefore reduce the number of workers running
Relevant Quality Attributes:		Modifiability
Scenario Parts	**Source:**	Operator
	Stimulus:	Wants to be able to reduce the numbers of workers running
	Artifact	Code
	Environment:	Development
	Response:	Turnip can now gracefully shut down instances and therefore reducing the number of workers
	Response Measure:	Some development time, number of instances should be in consistent with workers.
Priority:		High
Difficulty:		Medium

Table 4.8.: Shutdown of instances

```
//pre: device is running
destroy_device(device);
```

Listing 4.2: terminate cloud instance Cloud ASL script

Turnip upgrade/update:
Here we might update a single component or upgrade the whole project. For a single component a simple component update should be enough. But for a whole project more drastic measures are needed. For our Cloud ASL script it is assumed that we are upgrading the whole software and will therefore shut down all instances and set everything up from scratch. Here it is assumed that Cloud ASL is running from an external location, i.e. not from the manager, and OSGi and basic required components are available on the created cloud instance. In table 4.9 our quality attribute scenario is displayed and in listing 4.3 we start by destroying all our devices and then create one instance and install and start required components to run the manager.

Scenario(s):		A Developer wants to upgrade Turnip
Relevant Quality Attributes:		Modifiability
Scenario Parts	**Source:**	Developer, System administrator
	Stimulus:	Wants to upgrade turnip to newer version
	Artifact	Code, System
	Environment:	Runtime
	Response:	Turnip is updated
	Response Measure:	Minimum downtime
Priority:		High
Difficulty:		Medium

Table 4.9.: Upgrade Turnip

```
//first we terminate all instances.
Device[] devices = asl.getDevices()
for(device in devices)
        asl.destroy_device(device)

//then we create a manager
Device manager = asl.create_instance_device("m1.small")
//and install required components
// Library
Component c1 = asl.install_component(manager, "http://URI/library.jar");
// Cloud~ASL
Component c2 = asl.install_component(manager, "http://URI/asl.jar");
// User interface
Component c3 = asl.install_component(manager, "http://URI/rosgi.jar");
// User interface
Component c4 = asl.install_component(manager, "http://URI/UI.jar");
// Request Manager
Component c5 = asl.install_component(manager, "http://URI/req_manager.jar"
    );
//Worker Factory
Component c6 = asl.install_component(manager, "http://URI/w_factory.jar");

//And then we start the components
asl.start_component(c1);
asl.start_component(c2);
asl.start_component(c3);
asl.start_component(c4);
asl.start_component(c5);
asl.start_component(c6);

//As the manager is capable of starting workers
//there is no need to to that manually
```

Listing 4.3: Turnip upgrade script

Update the User Interface:
The user interface is a stateless component and therefore updating it would not have any external consequences. Examples of user interface updates are a modified look with new background image and new UI functional features done through JavaScript. In table 4.10 our quality attribute scenario is displayed and in listing 4.4 we update the user interface by stopping the UI component, update the component and start it again.

Scenario(s):		A developer wants to update the web interface
Relevant Quality Attributes:		Modifiability, Usability
Scenario Parts	**Source:**	Developer, System administrator
	Stimulus:	Wants to update the user interface
	Artifact	Code, System
	Environment:	Runtime
	Response:	The user interface has been changed
	Response Measure:	No downtime, the only noticeable change is the UI
Priority:		High
Difficulty:		Low

Table 4.10.: updated UI

```
1 //pre: Component ui exists on Device manager
  asl.stop_component(ui);
3 asl.update_component(ui, "http://URI/updatedUI.jar");
  asl.start_component(ui);
```

Listing 4.4: Update user interface Cloud ASL script

Change the User Interface:
An example of change of user interface is a change from web based UI to systems client user interface, i.e. Java Swing, an upgraded UI v.s. updated UI. In table 4.11 our quality attribute scenario is displayed and in listing 4.5 we assume that the new UI communicates to the manager through a new rest API and therefore install and start a new rest interface alongside our current UI. Now a Java Swing client can communicate to our manager via the new REST interface.

Scenario(s):		A developer wants to change the user interface from web interface to Java Swing on a desktop
Relevant Quality Attributes:		Modifiability, Usability
Scenario Parts	**Source:**	Developer
	Stimulus:	Wants to change the user interface from web based to Java Swing desktop interface
	Artifact	Code, System
	Environment:	Runtime
	Response:	New interface is available
	Response Measure:	No downtime, the only noticeable change is the UI, current program state is preserved
Priority:		Low
Difficulty:		Medium

Table 4.11.: New UI

```
//pre: Component ui exists on Device manager
Component rest = asl.install_component(device,"http://URI/newRESTService.
    jar");
asl.start_component(rest);
//now the new user interface can communicate via REST services
```

Listing 4.5: Change user interface Cloud ASL script

Use BOINC[2] for processing:
BOINC can be implemented in multiple ways, using local or external severs, but in this case it will be assumed that the server is going to be running on the manager and the processing component is going to be located on the library, like the current implementation of the Turnip. In table 4.12 our quality attribute scenario is displayed and in listing 4.6 we change our implementation by stopping, updating and starting the Request Manager and the Library.

[2]http://boinc.berkeley.edu/: The Berkeley Open Infrastructure for Network Computing (BOINC) is a non-commercial middle-ware system for volunteer and grid computing

Scenario(s):		As a developer I want to use Turnip as a BOINC process
Relevant Quality Attributes:		Modifiability, usability
Scenario Parts	**Source:**	Developer
	Stimulus:	Wants to use Turnip for other processes than ray-tracing, like BOINC
	Artifact	Code
	Environment:	Development
	Response:	Turnip is running BOINC or other distributed process
	Response Measure:	Some development time,
Priority:		High
Difficulty:		High

Table 4.12.: new use case

```
//to change the service the library
//and the request manager need to be switched.
//pre: Component rManager runs request manager and Component lib runs
    Library
asl.stop_component(lib);
asl.stop_component(rManager);
asl.update_component(lib, "http://URI/boinc_lib.jar");
asl.update_component(rManager, "http://URI/boinc_request_manager");
asl.start_component(lib)
asl.start_component(rManager)
```

Listing 4.6: Boinc Cloud ASL script

Updating r-OSGi:
Because r-OSGi runs on all devices and connects all workers to the manager and it can not be assumed that the new version reconnects lost connections correctly, all workers will be terminated and the manager allowed to be in charge of recreating all workers. In table 4.13 our quality attribute scenario is displayed and in listing 4.7 we start by terminate all of our workers and stopping, update and starting r-OSGi again.

Scenario(s):		A new/bug-fixed version of r-OSGi is added.
Relevant Quality Attributes:		Performance, security
Scenario Parts	**Source:**	Developer
	Stimulus:	A new improved version of r-OSGi is available
	Artifact	System
	Environment:	Runtime
	Response:	A new and improved version of r-OSGi is added with low downtime
	Response Measure:	Minimum downtime
Priority:		High
Difficulty:		Low

Table 4.13.: A new/bug-fixed version of r-OSGi is added.

```
1 //pre: Component rOSGi runs r-OSGi on the manager device,
  //workers is a array of all worker devices
3 for( worker in workers )
    asl.terminate_device(worker)
5 asl.stop_component(rOSGi);
  asl.update_component(rOSGi, "http://URI/new_rosgi.jar");
7 asl.star_component(rOSGi)
  //the manager is running the new version of r-OSGi and will create new
9 //workers with the new r-OSGi version
```

Listing 4.7: r-OSGi Cloud ASL

Change r-OSGi to other distributed OSGi framework:
As the difference between the new framework and r-OSGi can not be defined beforehand it is assumed that the only change will be with a new component instead of r-OSGi and different ways for the worker factory to create workers. Here we will use Apache CFX as an example of a new distributed OSGi framework. In table 4.14 our quality attribute scenario is displayed and in listing 4.8 we start by terminating all workers, stopping Worker Factory and r-OSGi, update the Worker Factory and install CFX and finally starting the updated/installed components again.

Scenario(s):		A change of distributed OSGi framework is needed, for example r-OSGi to Apache CFX
Relevant Quality Attributes:		Modifiability, performance, security, usability, availability
Scenario Parts	**Source:**	Developer and System Administrator
	Stimulus:	Wishes to change underlying distributed framework
	Artifact	System
	Environment:	Runtime
	Response:	Apache CFX is running instead of r-OSGi and the manager is still operating
	Response Measure:	Minimum downtime
Priority:		Low
Difficulty:		Medium-High

Table 4.14.: r-OSGi -> Apache CFX

```
1  //pre: Component rOSGi runs r-OSGi on the manager device,
   //Component wFactory runs worker factory on the manager device,
3  //workers is an array of all worker devices
   for( worker in workers )
5    asl.terminate_device(worker);
   asl.stop_component(wFactory);
7  asl.stop_component(rOSGi);
   asl.uninstall_component(rOSGi);
9  asl.update_component(wFactory, "http://URI/worker_factory.jar");
   cfx = asl.install_component(manager, "http://URI/cfx.jar");
11 asl.start_component(cfx)
   asl.start_component(wFactory)
13 //the manager is running CFX and will create new
   //workers with CFX instead of the new r-OSGi version
```

Listing 4.8: Change from r-OSGi to CFX Cloud ASL script

Migrate from Amazon EC2 public cloud to Eucalyptus private cloud:
As Amazon EC2 and Eucalyptus have the same API a simple configuration modification in the Library is enough to change the these providers, but all devices still have to be shut down to migrate them to the new cloud. Before the new instance are started the local Cloud ASL needs to be updated. In table 4.15 our quality attribute scenario is displayed and in listing 4.9 we start by destroying all running devices, then we update running Cloud ASL implementation and last we install and start required components to run the manager.

Scenario(s):		A developer wants to change runtime platform from eucalyptus to EC2
Relevant Quality Attributes:		Modifiability, Availability
Scenario Parts	**Source:**	Developer
	Stimulus:	Wishes to change cloud platform
	Artifact	System
	Environment:	Development
	Response:	Turnip is running on EC2
	Response Measure:	Few development hours and development on new VM images
Priority:		High
Difficulty:		Medium

Table 4.15.: Eucalyptus -> EC2

```
//first we terminate all instances.
Device[] devices = asl.getDevices()
for(device in devices)
        asl.destroy_device(device)

//the local Cloud ASL implementation is updated
asl_update_component(this, http://URI/cloud_asl.jar);

//then a manager is created
Device manager = asl.create_instance_device("m1.small")
// Library
Component c1 = asl.install_component(manager, "http://URI/library.jar");
// Cloud-ASL
Component c2 = asl.install_component(manager, "http://URI/asl.jar");
// User interface
Component c3 = asl.install_component(manager, "http://URI/rosgi.jar");
// User interface
Component c4 = asl.install_component(manager, "http://URI/UI.jar");
// Request Manager
Component c5 = asl.install_component(manager, "http://URI/req_manager.jar"
    );
//Worker Factory
Component c6 = asl.install_component(manager, "http://URI/w_factory.jar");

//And then we start the components
asl.start_component(c1);
asl.start_component(c2);
asl.start_component(c3);
asl.start_component(c4);
asl.start_component(c5);
asl.start_component(c6);
```

```
32 //Because the manager is capable of starting workers
   //there is no need to that manually
```

Listing 4.9: Migrate from EC2 to Eucalyptus Cloud ASL script

Change cloud providers:
As in the previous scenario, all instances have to be shut down and recreated at the new cloud provider. Because the cloud API would change, the Cloud ASL implementation needs to be modified to work with the new provider.

Scenario(s):		A developer wants to change runtime platform from eucalyptus to OpenNebula (or some other IaaS)
Relevant Quality Attributes:		Modifiability
Scenario Parts	**Source:**	Developer
	Stimulus:	Wishes to change cloud platform
	Artifact	Code, System
	Environment:	Development
	Response:	Turnip is running on OpenNebula
	Response Measure:	Few development hours and development on new VM images
Priority:		Medium
Difficulty:		Medium

Table 4.16.: Eucalyptus -> OpenNebula

```
1  //first all instances are terminated.
   Device[] devices = asl.getDevices()
3  for(device in devices)
       asl.destroy_device(device)
5
   //then a manager is created
7  Device manager = asl.create_instance_device("m1.small")
   //and install required components
9  // Library
   Component c1 = asl.install_component(manager, "http://URI/library.jar");
11 // Cloud ASL
   Component c2 = asl.install_component(manager, "http://URI/asl.jar");
13 // User interface
   Component c3 = asl.install_component(manager, "http://URI/rosgi.jar");
15 // User interface
   Component c4 = asl.install_component(manager, "http://URI/UI.jar");
17 // Request Manager
   Component c5 = asl.install_component(manager, "http://URI/req_manager.jar");
19 //Worker Factory
```

```
Component c6 = asl.install_component(manager, "http://URI/w_factory.jar");

//And then the components are started
asl.start_component(c1);
asl.start_component(c2);
asl.start_component(c3);
asl.start_component(c4);
asl.start_component(c5);
asl.start_component(c6);

//Because the manager is capable of starting workers
//there is no need to to that manually
```

Listing 4.10: Migrate from EC2 to other provider

4.2. Quantitative Evaluation

Quantitative evaluation is an important factor in evaluating the usefulness of the system. The focus was on two distinct yet relevant measures on quantitative evaluation: performance and scalability.

4.2.1. Performance

The performance of systems can become a critical factor if it is going to be useful for real world usage. The performance of Cloud ASL was measured by measuring it in comparison to scripting manual architectural change. Two scripts were created, one Cloud ASL script and one manual script, which had the same functionality. The manual script used the same third party components as Cloud ASL to minimize the factor of uncertainty between different libraries. These scripts began by creating device instances and then did a few component operations. These operations can be seen in the ASL script in listing 4.11:

```
Device device = asl.create_device("m1.small")
Component com = asl.install_component(device, "http://bjolfur.com/
    turnip_library.jar")
asl.start_component(com)
asl.stop_component(com)
asl.update_component(com,"http://bjolfur.com/turnip_library.jar")
asl.start_component(com)
```

Listing 4.11: ASL performance test

The groovy script was designed to do the exactly the same, with same libraries, see appendix A. In short, both the scripts do the following: start, by creating a Amazon EC2 instance through the Amazon AWS SDK; wait for the instance to run, when running it used SSH to access the machine; set configuration switches and run the OSGi platform. Next it uses telnet to interact with the OSGi platform and install, update start and stop bundles/components. The time it took the scripts to run instances was measured as well as the time it took to do operations on the OSGi platform (configuring components). The results are listed in table 4.17, in figure 4.1 a scatter chart displays the startup times of Cloud ASL scripts versus the manual startup times, and in figure 4.2 a scatter chart displays the operations timing of Cloud ASL scripts versus the manual script times.

	ASL Device Startup	Manual Device Startup	ASL Component Operations	Manual Component Operations
Count	56	56	50	50
Average	01:48.2	01:36.8	00:07.7	00:06.6
Standard Deviation	00:47.6	00:38.2	00:01.1	00:01.1
Min	01:12.7	01:00.7	00:06.8	00:05.7
Max	04:33.3	03:51.8	00:13.2	00:09.5

Table 4.17.: List of ASL vs. manual change statistics.

The scripts were run on many instances, and the number of instances was increased for each iteration, with the average number of instances equalling 10. The time of starting each instance was measured as well as the time of running the Cloud ASL/Manual operation on these instances. Detailed measurements of timing can be found in appendix B. Basic statistical analysis were done on the measurements; standard deviation, average, minimum and maximum data points; and they are listed in table 4.17. From these results it can be observed that there is about one second penalty of using the Cloud ASL script for creating devices and a one second penalty in component operations, but where the standard deviation is higher than this difference it can be assumed that the time difference is not significant. It can be argued that as the system is more complex and has more functional properties, it should result in less performance than a manual architectural change script. With basic code refactoring and optimization current difference could be lowered. The error in these numbers is quite high, indicating a high uncertainty factor in external environments. Because the cloud infrastructure does not have uniform capability and timing, doing uniform tests would become very expensive and time-consuming. Creating uniform testing would require a controlled private cloud or a much larger sample from a public cloud as standard deviation and standard error for creating devices with greater certainty would have to be figured out.

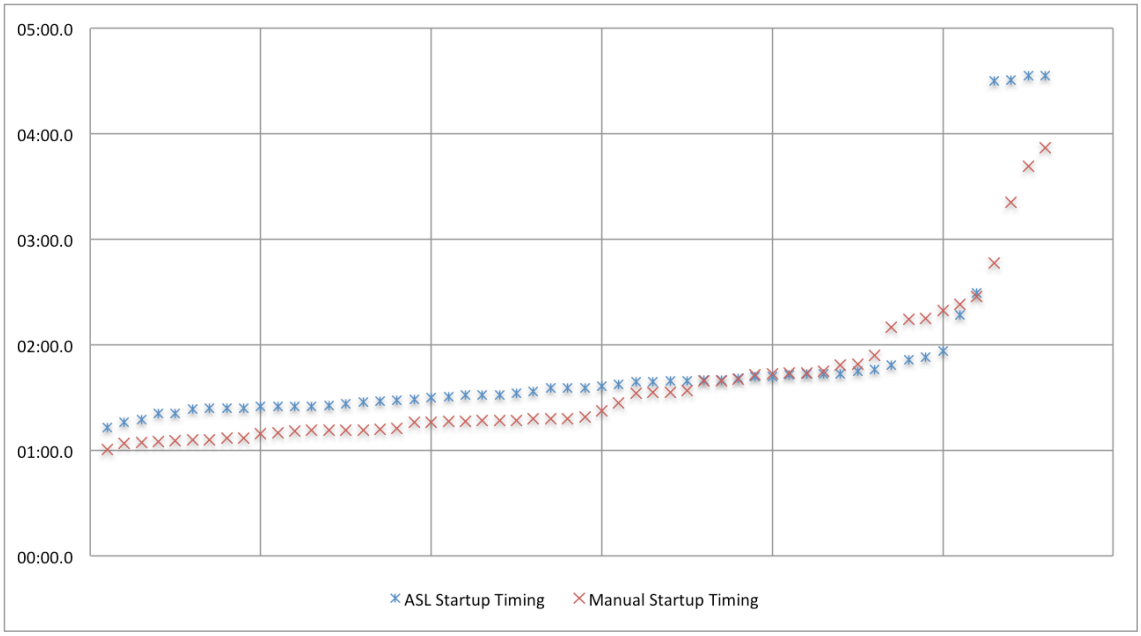

Figure 4.1.: A scatter chart of startup timing of Cloud ASL script vs. manual script

4.2.2. Scalability

After comparing Cloud ASL to performing manual architectural change through Groovy and OSGi, it can be assumed that ASL performance is efficient, but it might be useful to know how it scales. Because Cloud ASL is created to run and operate on multiple devices, simultaneously doing that will be the next subject of evaluation. It was measured by running ASL scripts on 1, 2, 4, 8, 16 and 32 instances at once. First a script was run that created a device and did a few component operations, then a script was run four times that did a few component operations on a currently running device to measure any statistical difference by devices.

The first script used was a script which creates a device and does a few basic operations on components:

```
Component com = asl.install_component(device, "http://bjolfur.com/
    turnip_library.jar");
asl.start_component(com);
asl.stop_component(com);
asl.stop_component(com);
asl.update_component(com,"http://bjolfur.com/turnip_library.jar");
asl.start_component(com);
```

Listing 4.12: ASL device creation scalability test

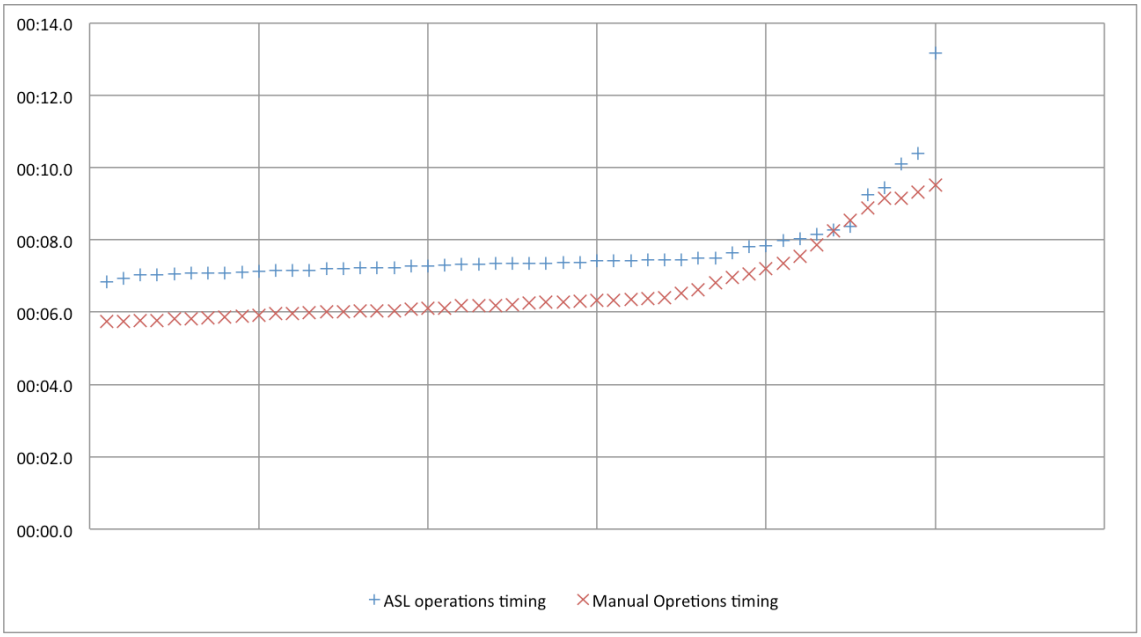

Figure 4.2.: A scatter chart of operations timing of Cloud ASL script vs. manual script

And a second script that did more advanced operations on components:

```
Component com = asl.install_component(device, "http://archive.apache.org/
    dist/felix/org.apache.felix.ipojo-1.6.0.jar");
asl.start_component(com);
asl.stop_component(com);
asl.update_component(com, "http://apache.osuosl.org/felix/org.apache.felix
    .ipojo-1.6.2.jar");
asl.stop_component(com);
asl.uninstall_component(com);
```

Listing 4.13: ASL component scalability test

The first script is the same as in the performance test, but this time the difference in multiple number of instances was measured.

The results are shown in table 4.18 and full raw data of the tests are provided in appendix C. Figure 4.3 displays a bar chart of averaged worker efficiency, i.e. average runtime, minimum runtime, maximum runtime and standard deviation averaged over number of workers running set job, and figure 4.4 displays an error chart of same data.

Number of devices created	1	2	4	8	16	32
#Devices successfully running	1	2	4	8	13	19
number of Cloud ASL processes	4	8	16	32	52	76
Average timing of each process	00:05.7	00:05.4	00:05.9	00:06.0	00:06.1	00:05.9
Min	00:05.3	00:04.4	00:04.5	00:04.3	00:05.0	00:04.9
Max	00:05.9	00:06.2	00:07.6	00:08.2	00:09.4	00:07.8
Standard deviation	00:00.2	00:00.7	00:00.8	00:00.8	00:00.7	00:00.7

Table 4.18.: List of ASL vs. manual change statistics.

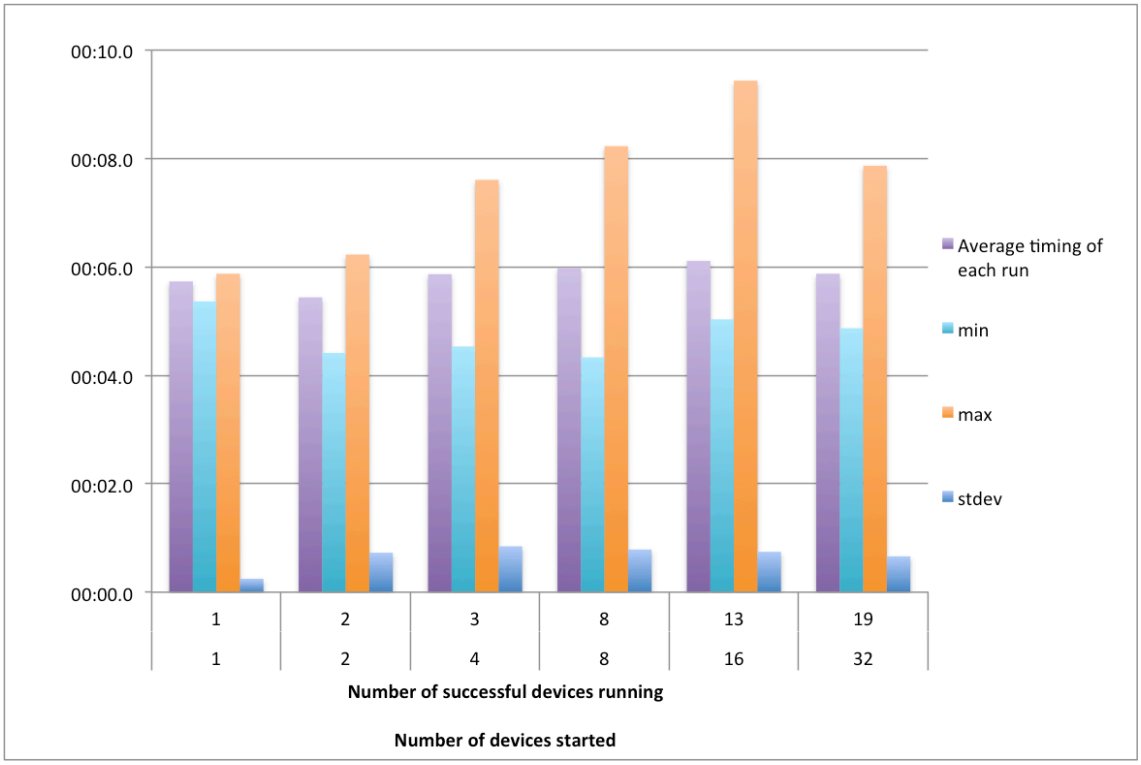

Figure 4.3.: Bar chart of ASL performance, rendering efficiency with different amount of workers

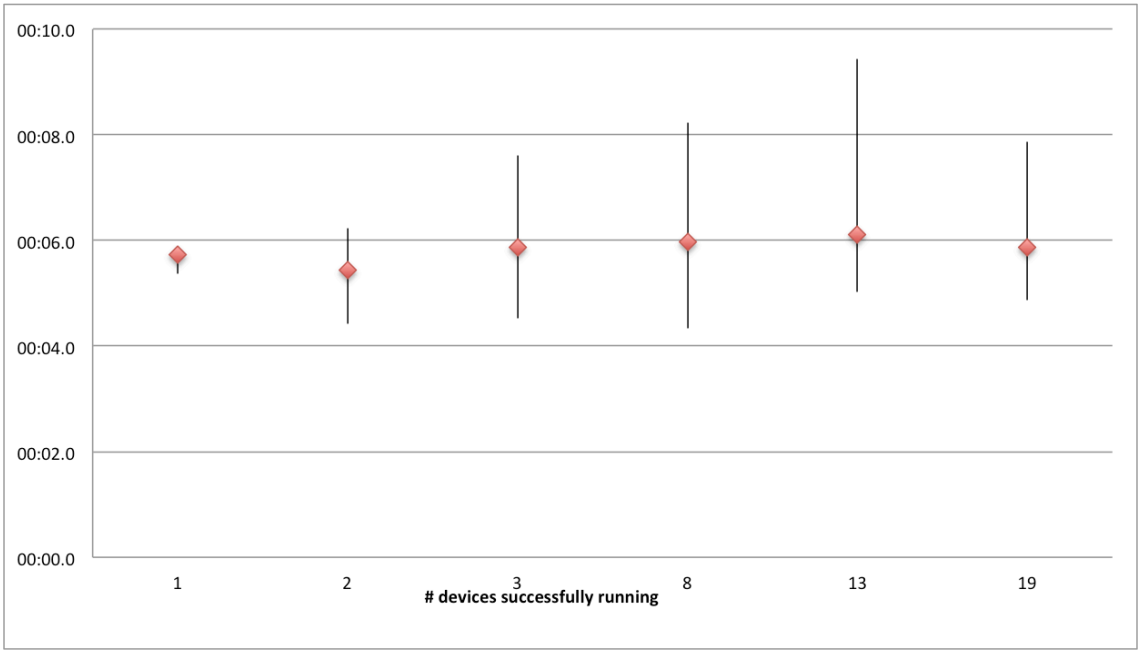

Figure 4.4.: Error chart of ASL performance, rendering efficiency with different amount of workers

What was found out by doing these measurements is that Amazon has a limit of running 20 instances at a time, but up to 24 instances were running at a time (where we started our instances all at once the Amazon API seems to be unable to detect the exact number of running instances at given time), which limited the 32 device run, and trying to start this many instances, 16 or 32, at once usually resulted in failures in starting some of the devices. From what can be seen in table 4.18, there is not much penalty for running Cloud ASL on multiple devices, however the variance runtime increases as the amount of devices increases.

The prototype was also tested by rendering an image on different numbers of devices. A rendering job was set up, which ran three times on 2, 4, 8 and 10 workers at the same time, where each worker had an independent device for itself. The job was to render an image of three aliens, with a resolution of 648x480, anti aliasing (4 samples), Gaussian filter, etc. The result of this rendering can be seen in table 4.19. An example of rendering a job with three workers can be viewed in figure 4.5 and figure 4.6 for rendering with ten workers. In the rendering figures each coloured box represents a different worker and the area under the box is the current bucket the worker is rendering.

Number of Workers	avg. Render time	avg. Render time per worker
3	06:44.3	20:12.9
5	04:08.8	20:44.2
8	02:34.0	20:31.7
10	02:04.2	20:41.7

Table 4.19.: List of rendering time compared to number of workers

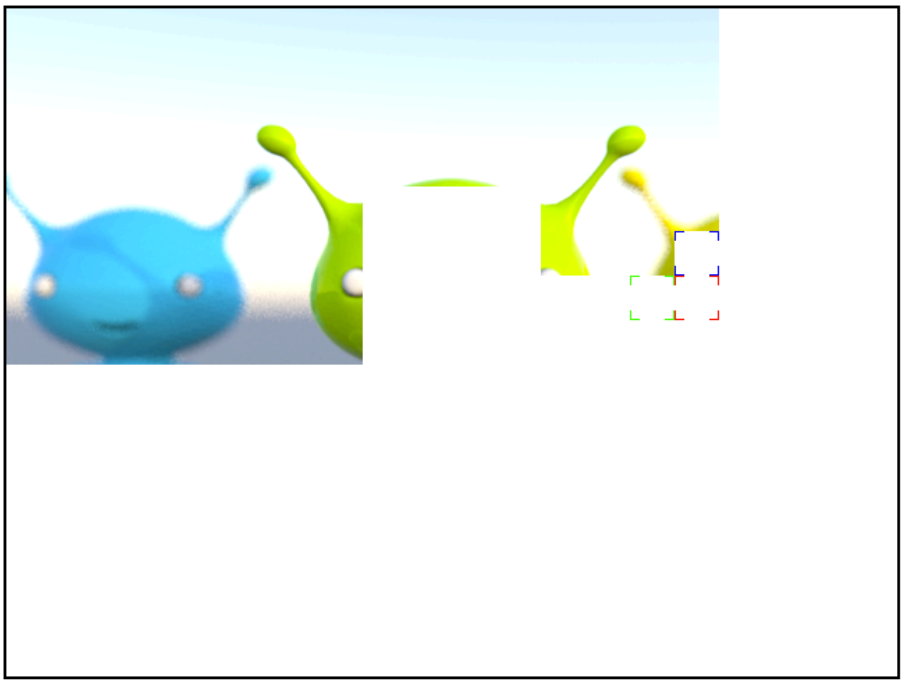

Figure 4.5.: Rendering job with 3 workers

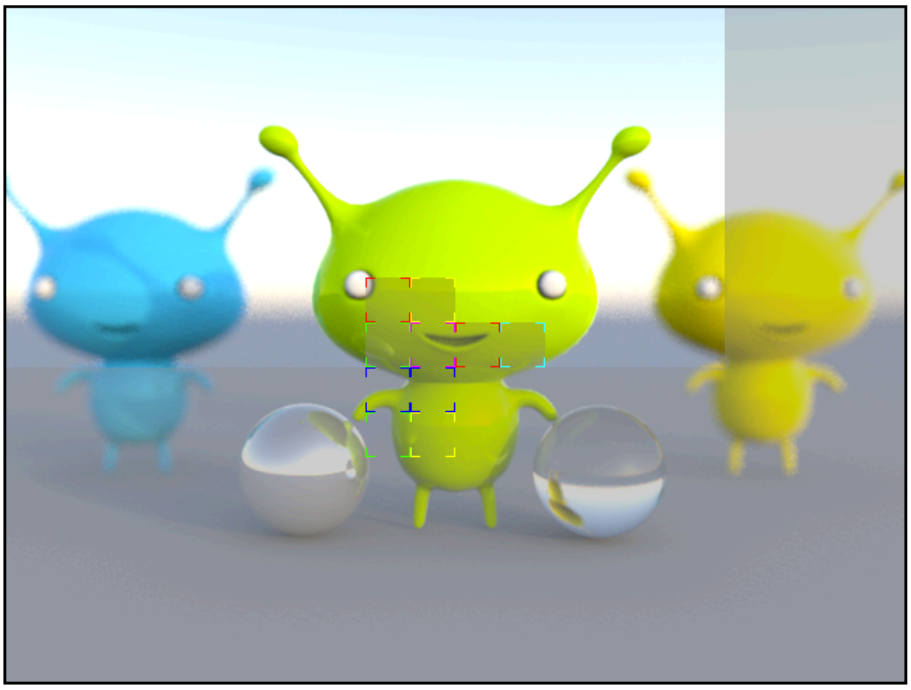

Figure 4.6.: Rendering job with 10 workers

The results from the prototype rendering are quite promising. It scales well where the time it takes to complete a job is almost linear compared to the number of workers. From the logs of the Amazon EC2 cloud it could be seen that in all cases the worker's CPU was fully utilized, while the manager was running on average 50% utilization. Serving the user interface was what made the most impact on the manager, as for each user the UI serves the current state of the rendering job to his/her preferences. This behavior does not scale effectively but could be fixed with simple optimization.

Although Cloud ASL scripting is slower than the measured alternative, manual architectural change script, the development time of writing Cloud ASL script is less then manual script. In the example from section 4.2.1, a 5 line ASL script was created. However, the alternative manual script consisted of more than 300 lines of code. While developing Cloud ASL, no performance optimizations were made, and it is predicted that with simple optimizations, Cloud ASL would perform as well as manual scripts.

All the tests were made on Amazon EC2 "m1.small" type of instances on customized 32 bit Ubuntu 9.04 Linux. As all the measurements were done on complete rendering work of all workers combined, there is no detailed information on the efficiency of individual workers. In future work this could be an interesting part of analysing the difference of efficiency between different types of cloud providers, cloud instance types and cloud instances.

5. Discussions and Conclusions

In this section the results of this thesis will be discussed and summarized, what went wrong, what might have been done better and future work on Cloud ASL and the Turnip.

In this thesis, Cloud ASL has been presented. It is an architectural scripting language focused on infrastructure as a service (IaaS) cloud computing services. This tool, or framework, enables controlling dynamic aspects of runtime software architecture with architectural operations in cloud computing. This is a suitable framework to use for creating a scalable and modifiable cloud computing software. On top of this framework, a prototype was created to evaluate the features of Cloud ASL for dynamic scaling and runtime architectural changes. Although Cloud ASL is not a feature complete system, and it might be difficult for inexperienced users to understand, it has a good educational perspective by thinking of architectural changes with an architectural framework. Using architectural scripting can help and simplify updates on software systems and software architecture. When software developers are making software systems nowadays, modifiability and scalability tend to become lower priority than they should. This is because og costs, lack of time, and the fact that the tools used do not require thoughts on future modifications and scalability. Cloud ASL could help here, as it requires a different kind of thought, by forcing developers to create modular software where each module is independent, in terms of it being able to maintain its state, while other modules are not present, and as devices are a part of the framework, allowing developers to design scalability from the start, making modifiability and scalability the core of the framework.

For this thesis, RHI (Reiknistofnun Háskóla Íslands) promised access to a part of a computer cluster, a few nodes to start with. Full root access was not granted to these machines. Instead an administrator was relied on to set them up and install required software for and with us, but due to how busy this administrator was and due to the complex network and software setup it was not possible to finish setting up Eucalyptus. Having a private cloud could have eased the first phase of the development given a steady access to a cloud environment, but it could also have slowed down the work in later phases as these solutions are not as complete and stable as the public cloud used in this project. Since the main work on this thesis was completed, NASA (US Space Agency), in cooperation with the IaaS service provider Rackspace, introduced a new private IaaS service software, OpenStack[1]. This IaaS software is simpler than Eucalyptus and might have

[1]OpenStack is a robust EC2 compatible IaaS software layer which includes an object store compatible

gotten a private cloud running if that offer had been available when this project started.

As the private cloud did not work, a few public cloud providers were signed up for, and in the the end Amazon AWS was chosen for this project as it seemed the most supported by other software libraries. Later, financial support was provided by Amazon to use their service for the project. A virtual machine image was created with the software bundles needed and stored in Amazon's Simple Storage Service. This was a complex task and very time consuming. Later, we found out a newer service from Amazon could have been used, Elastic Block Storage, to store the virtual images which would have provided more flexibility. For example it could have been possible to start and stop instances and creating a new virtual machine image from an existing image could have been possible. This would have given Cloud ASL more modifiability and eased development and testing. Another Amazon AWS service that would have helped is Amazon Virtual Private Cloud: with that service a virtual network could have been created and all network protocols enabled within that network. R-OSGi relies on multicast over SLP[2] to be able to automatically discover OSGi services over the network. Because multicast is disabled on Amazon's regular network, it was necessary to create the service connection manually, which caused more overhead and complexity. Although these services would have helped in many ways, it would also force us to rely more on Amazon as a service provider and make switching service providers a more difficult task.

The cloud computing integration into Cloud ASL began by using a cloud computing abstraction framework, JCloud, which exposes interfaces to operate on multiple cloud frameworks. This framework was not complete enough and did not expose common cloud computing interactions well enough, so to use common cloud operations, such as starting a virtual instance, it was necessary to go through the framework and interact directly with the Amazon API. A final version was not foreseeable in the near future and the Amazon API was used directly instead. This problem brings a question for Cloud ASL: Can Cloud ASL be used to support multiple IaaS providers?

Integrating more cloud computing providers into Cloud ASL would improve modifiability, and that could be done by extracting cloud functionality into a Cloud ASL interface and cloud operations into a Cloud ASL component and exposing specific cloud operations as a Cloud ASL service through the common cloud interface. This will limit the functionality of Cloud ASL as cloud providers do not support the same functionality, and therefore the cloud interface would only include common cloud provider operations.

Testing applications in cloud environments can be difficult, as one application can be running on multiple instances and be using multiple components and services. To be able to test and debug Cloud ASL and the Turnip it was necessary to connect to each

with Amazon's S3. http://openstack.org/

[2] Service Location Protocol is a service discovery protocol that allows computers and other devices to find services in a local area network without prior configuration.

virtual machine via SSH and monitor the components by running OSGi and viewing logs and outputs. This is a time consuming and hard process, and becomes more complex with a larger cloud. A future feature for Cloud ASL would be added testability, by remote logging, remote debugging on multiple instances and/or distributed unit tests.

After this thesis was completed, Amazon EC2 introduced a new type of cloud computing instance, cluster GPU instance, which is an instance containing powerful GPU's including multiple CPU's. An interesting approach would be to optimize the Turnip to use these GPU's for rendering and measure any improvements and compare the efficiency to other instances. This could also be the basis of some sort of benchmark, rendering time compared to cost of instance.

For the implementation of the Turnip done in this project there was room for improvement. Firstly, all libraries and interfaces were put in a single module, the library, which existed on all devices. This simplified deployment, as only a single module had to be deployed for all libraries and interfaces, but this made the library an oversized and complicated module with difficult maintainance which could make updates on any library or interface a relatively big task. A better solution would have been keeping all libraries in separate modules and interfaces in one or more modules. Due to limited time, UI and auto-scaling features were not implemented, such as automatic startup and shutdown of devices, storing resulting images and uploading and customizing rendering jobs. These features were not important for the Cloud ASL implementation but would have made the Turnip more complete.

Making a ray-tracing program prototype for Cloud ASL took too much effort and time, as the complexity of this prototype was too great, and this time should have been spent focusing on the ASL implementation. The architecture of the prototype did therefore not become as good as it should have been. Features were missing, and stability and testability might have been better. Evolutionary architectural prototype would have been useful in this situation, that is taking the good working parts of the prototype and iterate the prototype a few times.

We have demonstrated with our work that Cloud ASL can be used as a high level architectural framework for cloud computing and is a valid architectural language. With some modifications and improvements Cloud ASL can become a basis for a Platform as a Service with Java and OSGi as the platform.

At the end, a question arises. As our Cloud ASL implementation is based on Java and does not support other programming languages, would a programming language neutral implementation be a feasible and/or possible advance of Cloud ASL?

Bibliography

[1] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2 ed., 2003.

[2] R. Hilliard, "IEEE recommended practice for architectural description of software-intensive systems," *IEEE Std 1471-2000*, pp. i –23, 2000.

[3] J. Bardram, H. Christensen, and K. Hansen, "Architectural prototyping: An approach for grounding architectural design and learning," in *Proc. 4th Working IEEE/IFIP Conference on Software Architecture*, pp. 15–24, Citeseer, 2004.

[4] M. Ingstrup and K. M. Hansen, "Modeling architectural change: Architectural scripting and its applications to reconfiguration," in *WICSA/ECSA*, pp. 337–340, 2009.

[5] J. Hamilton, "Internet-scale service efficiency," in *Large-Scale Distributed Systems and Middleware (LADIS) Workshop (September 2008)*, 2008.

[6] Visir.is, "Verne greiði fjögur sent á kílóvattstund." `http://www.visir.is/article/20100216/FRETTIR01/606626051/-1`, February 2010.

[7] S. Helgason, "Loks upplýst um raforkuverð." `http://www.herdubreid.is/?p=1487`, February 2010.

[8] M. Ingstrup and W. Zhang, "D4.8 self-* properties ddk prototype and report," tech. rep., Hydra, 2008.

[9] J. Kim and D. Garlan, "Analyzing architectural styles with alloy," in *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, p. 80, ACM, 2006.

[10] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "Above the clouds: A Berkeley view of cloud computing," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

[11] D. Parkhill, *The challenge of the computer utility*. Addison-Wesley Reading, MA, 1966.

[12] L. Qian, Z. Luo, Y. Du, and L. Guo, "Cloud Computing: An Overview," *Cloud Computing*, pp. 626–631, 2009.

[13] W. Vogels, "A Head in the Clouds? The Power of Infrastructure as a Service," in *First workshop on Cloud Computing and in Applications (CCA'08)(October 2008)*, October 2008.

[14] D. Robert, "Jeff Bozes' risky bet," *Business Week*, vol. 4009, p. 53, 2006.

[15] R. Buyya, C. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.

[16] P. Mell and T. Grance, "The NIST Definition of Cloud Computing. National Institute of Standards and Technology," *Information Technology Laboratory, Version*, vol. 15, pp. 10–7, 2009.

[17] C. Boulton, "Oracle CEO Larry Ellison Spits on Cloud Computing Hype," *eWeek.com, September*, vol. 29, pp. 11–14, 2009.

[18] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid-Volume 00*, pp. 124–131, IEEE Computer Society, 2009.

[19] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski, "AppScale Design and Implementation," *Computer Science Department University of California, Santa Barbara, Research Report*, 2009.

[20] J. C. Anderson, "Announcing AppDrop.com (host Google App Engine projects on EC2)." `http://jchrisa.net/drl/_design/sofa/_show/post/announcing_appdrop_com__host_go`, April 2008.

[21] N. Medvidovic and R. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on software engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[22] H. Christensen, A. Corry, and K. Hansen, "An approach to software architecture description using UML," *Computer Science Department, University of Aarhus*, 2004.

[23] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little, *Documenting software architectures: views and beyond*. Pearson Education, 2002.

[24] R. Kazman, M. Klein, and P. Clements, "ATAM: Method for Architecture Evaluation," Tech. Rep. CMU/SEI-2000-TR-004, Software Engineering Institute, 2000.

[25] N. Rozanski and E. Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2005.

[26] H. Christensen and K. Hansen, "An empirical investigation of architectural prototyping," *Journal of Systems and Software*, vol. 83, no. 1, pp. 133–142, 2010.

[27] H. Christensen and K. Hansen, "Architectural prototyping in industrial practice," *Software Architecture*, pp. 196–209, 2008.

[28] K. M. Hansen and M. Ingstrup, "Modeling and analyzing architectural change with alloy," in *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, (New York, NY, USA), pp. 2257–2264, ACM, 2010.

[29] M. Fowler, "Language workbenches: The killer-app for domain specific languages," *Accessed online from: http://www.martinfowler.com/articles/languageWorkbench.html*, 2005.

[30] J. Rellermeyer, G. Alonso, and T. Roscoe, "R-osgi: Distributed applications through software modularization," in *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*, pp. 1–20, Springer-Verlag, 2007.

A. Performance Script

A.1. Cloud ASL Script

```
Component com = asl.install_component(device, "http://bjolfur.com/
    turnip_library.jar")
asl.start_component(com)
asl.stop_component(com)
asl.update_component(com,"http://bjolfur.com/turnip_library.jar")
asl.start_component(com)
```

Listing A.1: ASL performance test

A.2. Groovy manual "ASL" Script

```
import com.amazonaws.auth.AWSCredentials
import com.amazonaws.auth.BasicAWSCredentials
import com.amazonaws.services.ec2.AmazonEC2
import com.amazonaws.services.ec2.AmazonEC2Client
import de.mud.telnet.TelnetWrapper
import com.amazonaws.services.ec2.model.*
import com.jcraft.jsch.*

// Manual OSGi update through following script:

class test {
  public def testId

  public static void main(String[] args) {

    for (i in 1..1) {
      println i;
      def newTest = new test();
      newTest.testId = i;
      def th = Thread.start {
        newTest.doTest()
      }
    }
```

```
25  }

27
    public def doTest() {
29
      //variables
31    String accessKeyId = "access key here";
      String secretKey = "secret key here";
33    String ami = "ami-4552bb2c"; // linux server with Java + OSGi
      String keyPrefix = "g-test-";
35    String availabilityZone = "us-east-1a";
      String type = "m1.small"
37    def id = null
      def state = "pending"
39
      private AmazonEC2 ec2;
41    Instance instance = null

43    private long startTime
      private long instanceTime
45    private long preTelnetTime
      private long endTime
47
      System.gc();
49    startTime = System.currentTimeMillis();

51    AWSCredentials credentials = new BasicAWSCredentials(accessKeyId,
          secretKey);
      ec2 = new AmazonEC2Client(credentials);
53
      // create one EC2 Instance
55    System.out.println("EC2.create_instance");
      RunInstancesRequest request = new RunInstancesRequest();
57    request.setInstanceType(type);

59    // set to zone
      Placement placement = new Placement();
61    placement.setAvailabilityZone(availabilityZone);
      request.setPlacement(placement);
63
      //Set the image ID to a custom generated AMI, which include Java +
          OSGi
65    request.setImageId(ami);

67    // Create key pair for user..
      CreateKeyPairRequest kpReq = new CreateKeyPairRequest();
69
      String newKeyPairName = keyPrefix + new Random().nextInt();
71    kpReq.setKeyName(newKeyPairName);
      CreateKeyPairResult kpres = ec2.createKeyPair(kpReq);
73    KeyPair keyPair = kpres.getKeyPair();
```

```
    request.setKeyName(newKeyPairName);// assign Keypair name for this
        request
75
    // make sure to have access to SSH port 22 on the default group on the
        EC2console
77  RunInstancesResult runInstancesRes = ec2.runInstances(request);
    System.out.println("run instance results: " + runInstancesRes);
79
    def ReservationId = runInstancesRes.getReservation().getReservationId
        ();
81  DescribeInstancesResult describeInstancesResult = ec2.
        describeInstances();
    List<Reservation> reservations = describeInstancesResult.
        getReservations();
83  Set<Instance> instances = new HashSet<Instance>();

85  for (Reservation reservation: reservations) {
      instances.addAll(reservation.getInstances());
87    if (reservation.getReservationId().equals(ReservationId)) {
        id = reservation.getInstances().get(0).getInstanceId();
89      state = reservation.getInstances().get(0).getState().getName();
        System.out.println("instance found!");
91    }
    }
93
    while (!state.equalsIgnoreCase("running")) {
95    println "state: " + state
      println "id: " + id
97    def describeInstancesRequest = new DescribeInstancesRequest();
      Collection<String> instanceIds = new ArrayList<String>();
99    instanceIds.add(id);
      describeInstancesRequest.setInstanceIds(instanceIds);
101
      describeInstancesResult = ec2.describeInstances(
          describeInstancesRequest);
103   reservations = describeInstancesResult.getReservations();


105
      for (Reservation reservation: reservations) {
107     for (Instance j: reservation.getInstances()) {
          if (j.getInstanceId().equals(id)) {
109         state = j.getState().getName();
            instance = j;
111       }
        }
113   }
    }
115
    System.gc();
117 instanceTime = System.currentTimeMillis();

119 //instance is running
```

```
    //Lets start OSGi and do some SSH magic
121 def postStartupScript = " \n" +
            //"sudo apt-get install coreutils &"
123         "cd /turnip\n" +
            "wget bjolfur.com/sunflow.tar \n" +
125         "tar xvzf sunflow.tar \n" +
            "cd /turnip/bundle\n" +
127         "wget http://apache.deathculture.net/felix/org.apache.felix.
                shell.remote-1.0.4.jar\n" +
            "cd /turnip\n" +
129         "echo osgi.shell.telnet.ip=" + instance.getPrivateIpAddress()
                + ">> /turnip/conf/system.properties\n" +
            "nohup ./run.sh\n";
131
    System.out.println("EC2.runScript");
133 Channel channel;
    ChannelSftp channelSftp;
135 final byte[] privateKey = keyPair.getKeyMaterial().getBytes();
    final byte[] emptyPassPhrase = new byte[0];
137 String user = "ubuntu";
    String host = instance.getPublicDnsName()
139
    try {
141   JSch jsch = new JSch();

143   jsch.addIdentity(
            user,    // String userName
145         privateKey,          // byte[] privateKey
            null,            // byte[] publicKey
147         emptyPassPhrase  // byte[] passPhrase
      );
149
      Session session = jsch.getSession(user, host, 22);
151   java.util.Properties config = new java.util.Properties();
      config.put("StrictHostKeyChecking", "no");
153   session.setConfig(config);
      session.connect();
155   channel = session.openChannel("shell");
      ByteArrayInputStream bi = new ByteArrayInputStream(postStartupScript
          .getBytes());
157   channel.setInputStream(bi);
      channel.setOutputStream(System.out);
159   channel.connect();
    } catch (JSchException e) {
161   e.printStackTrace();
    }
163
    System.gc();
165 preTelnetTime = System.currentTimeMillis();

167 TelnetWrapper telnet = new TelnetWrapper();
    def bundleId = 0;
```

```
169     try {
          telnet.connect(instance.getPublicIpAddress(), 6666);
171       System.out.printf("telnet.connect(%s, 6666);", instance.
              getPublicIpAddress());

173       telnet.setPrompt("-> ");
          System.out.println(telnet.waitfor("-> "));
175       String newBundleText = telnet.send("install http://bjolfur.com/
              turnip_library.jar");
          System.out.println(newBundleText);
177       bundleId = getOnlyNumerals(newBundleText);
          System.out.println(bundleId);
179       telnet.disconnect();
          //to be sure that connection has ben properly terminated, lets wait
              for a sec.
181       Thread.sleep(1000);
        } catch (java.io.IOException e) {
183       e.printStackTrace();
        } catch (InterruptedException e) {
185       e.printStackTrace();
        }
187     System.out.println("telnetConnector.stop " + bundleId);

189     telnet = new TelnetWrapper();
        try {
191       telnet.connect(instance.getPublicIpAddress(), 6666);
          System.out.printf("telnet.connect(%s, 6666);", instance.
              getPublicIpAddress());
193
          telnet.setPrompt("-> ");
195       System.out.println(telnet.waitfor("-> "));
          System.out.println(telnet.send("start " + bundleId));
197       telnet.disconnect();
          //to be sure that connection has ben properly terminated, lets wait
              for a sec.
199       Thread.sleep(1000);
        } catch (java.io.IOException e) {
201       e.printStackTrace();
        } catch (InterruptedException e) {
203       e.printStackTrace();
        }
205
        telnet = new TelnetWrapper();
207     try {
          telnet.connect(instance.getPublicIpAddress(), 6666);
209       System.out.printf("telnet.connect(%s, 6666);", instance.
              getPublicIpAddress());

211       telnet.setPrompt("-> ");
          System.out.println(telnet.waitfor("-> "));
213       System.out.println(telnet.send("stop " + bundleId));
          telnet.disconnect();
```

```
215       //to be sure that connection has ben properly terminated, lets wait
              for a sec.
          Thread.sleep(1000);
217     } catch (java.io.IOException e) {
          e.printStackTrace();
219     } catch (InterruptedException e) {
          e.printStackTrace();
221     }

223     System.out.println("telnetConnector.update " + bundleId + "  URI: http
            ://bjolfur.com/turnip_library.jar");

225     telnet = new TelnetWrapper();
        try {
227       telnet.connect(instance.getPublicIpAddress(), 6666);
          System.out.printf("telnet.connect(%s, 6666);", instance.
              getPublicIpAddress());
229
          telnet.setPrompt("-> ");
231       System.out.println(telnet.waitfor("-> "));
          System.out.println(telnet.send("update " + bundleId + " http://
              bjolfur.com/turnip_library.jar"));
233       telnet.disconnect();
          //to be sure that connection has ben properly terminated, lets wait
              for a sec.
235       Thread.sleep(1000);
        } catch (java.io.IOException e) {
237       e.printStackTrace();
        } catch (InterruptedException e) {
239       e.printStackTrace();
        }
241
        telnet = new TelnetWrapper();
243     try {
          telnet.connect(instance.getPublicIpAddress(), 6666);
245       System.out.printf("telnet.connect(%s, 6666);", instance.
              getPublicIpAddress());

247       telnet.setPrompt("-> ");
          System.out.println(telnet.waitfor("-> "));
249       System.out.println(telnet.send("start " + bundleId));
          telnet.disconnect();
251       //to be sure that connection has ben properly terminated, lets wait
              for a sec.
          Thread.sleep(1000);
253     } catch (java.io.IOException e) {
          e.printStackTrace();
255     } catch (InterruptedException e) {
          e.printStackTrace();
257     }

259     System.gc();
```

```
    endTime = System.currentTimeMillis();

    System.out.println(testId + " startTime = " + startTime);
    System.out.println(testId + " instanceTime = " + instanceTime);
    System.out.println(testId + " preTelnetTime = " + preTelnetTime);
    System.out.println(testId + " endTime = " + endTime);
    System.out.println(testId + " instance startup took : " + (
        instanceTime - startTime));
    System.out.println(testId + " ssh startup took : " + (preTelnetTime -
        startTime));
    System.out.println(testId + " whole startup took : " + (endTime -
        startTime));
  }

  private static String getOnlyNumerals(String str) {

    if (str == null) {
      return null;
    }

    StringBuffer strBuff = new StringBuffer();
    char c;

    for (int i = 0; i < str.length(); i++) {
      c = str.charAt(i);

      if (Character.isDigit(c)) {
        strBuff.append(c);
      }
    }
    return strBuff.toString();
  }
}
```

Listing A.2: Groovy manual ASL Script

B. Numerical results for performance tests

Table B.1.: Numerical results for performance tests

ASL startup timing	Manual startup timing	ASL operations timing	Manual operations timing
01:12.7	01:00.7	00:06.8	00:05.7
01:16.0	01:03.8	00:06.9	00:05.7
01:17.3	01:04.7	00:07.0	00:05.8
01:21.0	01:05.2	00:07.0	00:05.8
01:21.2	01:05.7	00:07.0	00:05.8
01:23.4	01:06.0	00:07.1	00:05.8
01:23.8	01:06.0	00:07.1	00:05.8
01:23.9	01:06.9	00:07.1	00:05.9
01:24.2	01:07.1	00:07.1	00:05.9
01:24.9	01:09.3	00:07.1	00:05.9
01:25.0	01:10.2	00:07.1	00:06.0
01:25.1	01:10.9	00:07.1	00:06.0
01:25.1	01:11.4	00:07.1	00:06.0
01:25.3	01:11.5	00:07.2	00:06.0
01:26.6	01:11.6	00:07.2	00:06.0
01:27.7	01:11.7	00:07.2	00:06.0
01:27.9	01:12.0	00:07.2	00:06.0
01:28.6	01:12.6	00:07.2	00:06.0
01:29.2	01:16.0	00:07.3	00:06.1
01:30.2	01:16.0	00:07.3	00:06.1
01:30.6	01:16.7	00:07.3	00:06.1
01:31.5	01:16.7	00:07.3	00:06.2
01:31.5	01:16.7	00:07.3	00:06.2
01:31.7	01:16.9	00:07.3	00:06.2
01:32.5	01:17.0	00:07.3	00:06.2
01:33.3	01:17.8	00:07.3	00:06.2
01:35.4	01:17.8	00:07.3	00:06.3
01:35.5	01:17.8	00:07.4	00:06.3

Continued on Next Page...

ASL startup timing	Manual startup timing	ASL operations timing	Manual operations timing
01:35.7	01:19.2	00:07.4	00:06.3
01:36.3	01:22.3	00:07.4	00:06.3
01:37.5	01:27.1	00:07.4	00:06.3
01:39.1	01:32.6	00:07.4	00:06.4
01:39.2	01:32.9	00:07.4	00:06.4
01:39.2	01:33.2	00:07.4	00:06.4
01:39.6	01:34.2	00:07.4	00:06.5
01:40.0	01:39.4	00:07.5	00:06.6
01:40.1	01:39.6	00:07.5	00:06.8
01:41.0	01:40.7	00:07.6	00:06.9
01:42.0	01:43.0	00:07.8	00:07.0
01:42.0	01:43.7	00:07.8	00:07.2
01:43.2	01:43.8	00:08.0	00:07.3
01:43.4	01:44.2	00:08.0	00:07.5
01:43.5	01:45.0	00:08.1	00:07.8
01:43.6	01:48.3	00:08.3	00:08.2
01:45.0	01:49.1	00:08.3	00:08.5
01:46.2	01:54.2	00:09.2	00:08.9
01:48.7	02:09.9	00:09.4	00:09.1
01:51.4	02:14.7	00:10.1	00:09.1
01:53.0	02:14.8	00:10.4	00:09.3
01:56.6	02:19.7	00:13.2	00:09.5
02:16.8	02:22.9		
02:29.8	02:27.6		
04:30.0	02:46.6		
04:30.6	03:20.9		
04:33.1	03:41.3		
04:33.3	03:51.8		

C. Numerical results for scalability tests

Column 1: Number of devices started
Column 2: Device number
Column 3: Startup time
Column 4: ASL runtime
Column 5: Iteration 1
Column 6: Iteration 2
Column 7: Iteration 3
Column 8: Iteration 4
Column 9: Average iteration timing
Column 10: Complete timing

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	1	01:26.2	00:06.7	00:05.9	00:05.8	00:05.3	00:05.9	00:05.7	01:32.9
2	1	01:25.3	00:07.4	00:05.6	00:04.4	00:04.4	00:04.9	00:04.8	01:32.6
	2	01:25.1	00:10.1	00:06.0	00:05.8	00:06.0	00:06.2	00:06.0	01:35.1
4	1	01:25.1	00:07.3	00:07.4	00:05.9	00:06.0	00:06.2	00:06.4	01:32.4
	2	01:25.0	00:07.3	00:06.1	00:05.6	00:05.2	00:05.9	00:05.7	01:32.3
	3	02:29.8	00:07.3	00:07.6	00:05.9	00:06.0	00:06.2	00:06.4	02:37.1
	4	01:12.7	00:07.1	00:05.4	00:04.5	00:04.6	00:04.9	00:04.9	01:19.9
8	1	01:39.2	00:07.2	00:06.2	00:05.9	00:06.1	00:06.0	00:06.0	01:46.4
	2	01:37.5	00:07.8	00:06.2	00:06.0	00:05.9	00:06.2	00:06.1	01:45.3
	3	01:32.5	00:07.4	00:05.9	00:06.0	00:05.9	00:05.9	00:05.9	01:39.9
	4	01:26.6	00:07.6	00:06.1	00:05.8	00:05.3	00:05.9	00:05.8	01:34.2
	5	01:40.0	00:07.1	00:06.1	00:05.9	00:05.9	00:05.8	00:05.9	01:47.1
	6	01:21.0	00:09.2	00:06.2	00:06.1	00:06.1	00:06.0	00:06.1	01:30.2
	7	01:41.0	00:07.2	00:06.2	00:05.8	00:05.9	00:05.2	00:05.8	01:48.2
	8	-02:07.4	**Instance failed to start**						
8	1	01:23.9	00:08.1	00:05.4	00:04.5	00:04.3	00:05.0	00:04.8	01:32.0
	2	01:43.5	00:07.4	00:06.2	00:05.9	00:06.2	00:06.2	00:06.1	01:50.9
	3	01:23.8	00:07.1	00:08.2	00:05.8	00:05.3	00:06.0	00:06.3	01:30.9
	4	01:27.7	00:07.1	00:06.7	00:05.9	00:06.0	00:07.1	00:06.4	01:34.8
	5	01:24.2	00:08.3	00:07.4	00:06.0	00:05.9	00:05.8	00:06.3	01:32.4
	6	01:35.7	00:08.0	00:06.0	00:05.9	00:05.8	00:05.1	00:05.7	01:43.6

	7	01:31.7	00:07.4	00:06.0	00:05.3	00:06.0	00:05.9	00:05.8	01:39.1
	8	01:27.9	00:07.4	00:07.4	00:05.9	00:05.9	00:05.9	00:06.3	01:35.3
16	1	01:24.4	00:07.1	00:06.3	00:06.4	00:06.0	00:06.1	00:06.2	01:31.5
	2	-18:52.8	**Instance failed to start**						
	3	01:26.4	00:07.0	00:07.6	00:06.1	00:06.0	00:05.9	00:06.4	01:33.4
	4	01:26.0	00:08.0	00:08.3	00:05.9	00:06.0	00:06.0	00:06.5	01:34.0
	5	01:26.3	00:06.8	00:07.6	00:06.0	00:05.9	00:06.1	00:06.4	01:33.1
	6	-18:52.7	**Instance failed to start**						
	7	01:52.3	00:07.1	00:05.9	00:05.2	00:06.2	00:05.8	00:05.8	01:59.3
	8	-18:52.7	**Instance failed to start**						
	9	01:47.0	00:10.1	00:05.7	00:05.0	00:05.0	00:05.4	00:05.3	01:57.1
	10	01:22.9	00:07.5	00:06.9	00:05.9	00:05.8	00:05.3	00:06.0	01:30.4
	11	01:34.4	00:07.3	00:06.2	00:06.2	00:06.1	00:06.3	00:06.2	01:41.8
	12	01:44.8	00:07.7	00:06.0	00:05.7	00:06.1	00:05.8	00:05.9	01:52.5
	13	01:34.3	00:06.9	00:05.8	00:05.8	00:05.8	00:05.8	00:05.8	01:41.2
	14	01:34.5	00:08.6	00:06.4	00:09.4	00:06.1	00:05.8	00:06.9	01:43.1
	15	01:34.4	00:06.9	00:06.0	00:06.1	00:06.1	00:06.2	00:06.1	01:41.3
	16	02:06.9	00:07.1	00:06.2	00:06.1	00:05.4	00:05.2	00:05.7	02:14.0
32	1	01:23.4	00:07.1	00:07.7	00:07.0	00:07.3	00:07.1	00:07.3	01:30.5
	2	01:43.6	00:07.4	00:06.6	00:06.0	00:05.9	00:06.3	00:06.2	01:51.0
	3	01:46.2	00:07.2	00:06.2	00:05.7	00:05.0	00:05.0	00:05.5	01:53.4
	4	-31:00.7	**Instance failed to start**						
	5	01:39.2	00:07.4	00:06.2	00:06.0	00:06.0	00:05.8	00:06.0	01:46.7
	6	-31:00.8	Instance failed to start						
	7	01:24.9	00:07.2	00:07.8	00:05.9	00:05.8	00:05.1	00:06.1	01:32.1
	8	01:42.0	03:46.5	-36:29.3	00:00.0	00:00.0	00:00.0	-24:07.3	05:28.5
	9	01:39.6	00:07.3	00:06.2	00:05.9	00:06.0	00:05.8	00:06.0	01:46.9
	10	04:33.3	00:07.1	00:06.4	00:05.7	00:05.3	00:05.9	00:05.8	04:40.4
	11	01:51.4	00:06.9	00:07.2	00:05.8	00:05.8	00:05.1	00:06.0	01:58.3
	12	04:30.6	00:07.0	00:06.1	00:05.8	00:05.8	00:05.4	00:05.8	04:37.6
	13	-31:00.9	**Instance failed to start**						
	14	-31:00.9	**Instance failed to start**						
	15	-31:01.0	**Instance failed to start**						
	16	01:21.2	00:07.2	00:06.5	00:05.7	00:05.7	00:06.1	00:06.0	01:28.4
	17	01:35.5	00:26.2	00:05.9	00:05.1	00:04.9	00:05.2	00:05.3	02:01.6
	18	01:43.4	00:07.1	00:07.4	00:05.8	00:05.8	00:05.3	00:06.1	01:50.4
	19	01:48.7	00:07.4	00:06.4	00:05.7	00:05.2	00:05.7	00:05.7	01:56.1
	20	01:43.2	00:07.3	00:06.5	00:05.9	00:05.8	00:05.1	00:05.8	01:50.5
	21	-31:01.1	**Instance failed to start**						
	22	01:56.6	00:08.3	00:06.7	00:05.8	00:05.9	00:05.1	00:05.9	02:04.9
	23	01:36.3	00:01.0	00:01.0	00:01.1	00:01.1	00:01.0	00:01.0	01:37.3
	24	-31:01.2	**Instance failed to start**						
	25	04:33.1	00:07.1	00:06.0	00:05.0	00:05.0	00:05.0	00:05.3	04:40.2

	26	01:42.0	00:06.8	00:06.1	00:05.8	00:05.2	00:05.9	00:05.8	01:48.8
	27	-31:01.3	**Instance failed to start**						
	28	-31:01.4	**Instance failed to start**						
	29	01:17.3	00:01.9	00:01.0	00:01.0	00:01.1	00:01.0	00:01.0	01:19.2
	30	-31:00.6	**Instance failed to start**						
	31	04:30.0	00:07.0	00:06.0	00:05.8	00:05.0	00:05.0	00:05.5	04:37.0
	32	01:15.5	00:07.0	00:06.0	00:05.6	00:04.9	00:05.3	00:05.4	01:22.6