UNIVERSITY OF OSLO
Department of Informatics

CloudML

A DSL for model-based realization of applications in the cloud

Master thesis

Eirik Brandtzæg

Spring 2012
Built: 23rd April 2012

CloudML

Eirik Brandtzæg

Spring 2012

Built: 23rd April 2012

Abstract

Contents

List of Figures

List of Tables

Preface

Part I

Introduction

0.1 Introduction

Short and sharp

0.1.1 Publications

This paper is based on a paper submitted to the ECMFA [8]. And a technical report submitted to Cloud'12 [4], involvement in REMICS (Deliverable 4.1). And a paper already published to BENEVOL'11 [19].

Chapter 1

Background: Cloud computing and Model-Driven Engineering

In this chapter the essential background topics for this thesis are introduced. The first topic is cloud computing, a way of providing computing power as a service instead of being a product. The second topic is about Model-Driven Engineering and Model-Driven Architecture and these in some relation to cloud computing.

1.1 Cloud computing

Cloud computing is gaining popularity and more companies are starting to explore the possibilities as well as the limitation to the cloud. A good example of cloud adaptation in a large scale scenario is when the White House moved their recovery.gov [13] operation to a cloud infrastructure, this was estimated to save them $750,000 at the current time, and even more on a long-term basis. This

Provider	**Service**	**Service Model**
AWS	Elastic Compute Cloud	Infrastructure
AWS	Elastic Beanstalk	Platform
Google	Google App Engine	Platform
CA	AppLogic	Infrastructure
Microsoft	Azure	Platform and Infrastructure
Heroku	Different services	Platform
Nodejitsu	Node.js	Platform
Rackspace	CloudServers	Infrastructure

Table 1.1: Common providers available services

section is based on the *National Institute of Standards and Technology* (NIST) standard.

The definitions under are mainly based on definitions provided by the NIST which is one of the leaders in cloud computing standardization. The main providers of cloud computing in April 2012 are Google, Amazon with AWS [1] and Microsoft. A non-exhaustive list of common providers is reproduced in TABLE. 1.1.

1.1.1 Characteristics

Cloud computing is about providing computation as services[18], such as virtual instances and file storage, rather than products. Cloud characteristics are what define the difference between normal hosting and computing as a service.

On-demand self-service. With *on-demand self-service* consumers can achieve provisioning without any human interaction. On-demand means dynamic scalability and elasticity of resource allocation, self-service so that users do not need to manually do these allocations themselves. Consider an online election system, for most of the year it will have low usage demands, but before and under election days it will have to serve a much larger amount of requests. With *on-demand self-service* the online election system could automatically be given more resources such as memory, computation power or even increase the number of instances to handle peak loads. The previous example has planned (or known) peak intervals, so even though automatic handling is appealing it could be solved by good planning. But sometimes predicting peak loads can be difficult, such as when a product suddenly becomes more popular than first anticipated. Twitter is a good example of a service that can have difficulties in estimating the amount of user demand and total amount of incoming requests. For instance in the year 2011 they achieved a record in *Tweets Per Second* (TPS) of $25,088$ triggered by a Japanese television screening of the movie “Castle in the Sky”. Each “tweet” essentially becomes at least one request to Twitter services. On a normal basis the service does not have to cope with this amount of requests, so this event was a strong and unpredicted fluctuation in day-to-day operations, and this is the kind of scenarios that cloud computing can help to tackle with characteristics such as *on-demand self-service*. With *on-demand self-service* allocation will automatically scale upwards as popularity increases and downwards as resources become superfluous.

Broad network access. When working against cloud solutions, in form of management, monitoring or other interactions it is important that capabilities are available over standard network mechanisms, supporting familiar protocols such as *Hypertext Transport Protocol* (HTTP)/HTTPS and *Secure Shell* (SSH). So users can utilize tools and software they already possesses or will have little difficulty gaining, such as web browsers. This is what the characteristic *broad network access* is all about, ensuring familiar mechanisms for communicating with a cloud service. Most cloud providers also provide web based consoles/interfaces that users can use to create, delete and manage their resources.

Resource pooling. Physical and virtual resources are pooled so they can be dynamically assigned and reassigned according to consumer demand. Users do not need to be troubled with scalability as this is handled automatically. This is a provider side characteristic which directly influence *on-demand self-service*. There is also a sense of location independence, users can choose geographical locations on higher abstracted levels such as country or sate, but not always as detailed or specific as city. It is important that users can choose at least country as geographical location for their product deployments for instance to reduce latency between product and customers based on customer location or data storing rules set by government.

Rapid elasticity. Already allocated resources can expand vertically to meet new demands, so instead of provisioning more instances (horizontal scaling) existing instances are given more resources such as *Random-access Memory* (RAM) and *Central Processing Unit* (CPU). Towards the characteristic of *on-demand self-service* allocation can happen instantly, which means on unexpected peak loads the pressure will be instantly handled by scaling upwards. It is important to underline that such features can be financially cost heavy if not limited, because costs reflect resource allocation.

Measured service. Monitoring and control of resource usages. Can be used for statistics for users, for instance to do analytical research on product popularity or determine user groups based on geographical data or browser usage. The providers themselves use this information to handle *on-demand services*, if they

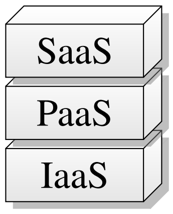


Figure 1.1: Cloud architecture service models.

notice that an instance has a peak in load or has a noticeable increase in requests they can automatically allocate more resources or capabilities to leave pressure. *Measuring* can also help providers with billing, if they for instance charge by resource load and not only amount of resources allocated.

1.1.2 Service models

Service models are definitions of different layers in cloud computing. The layers represent the amount of abstraction developers get from each *service model*. Higher layers have more abstraction, but can be more limited, while lower levels have less abstraction and are more customizable. Limitations could be in many different forms, such as bound to a specific operating system, programming language or framework. There are three main architectural service models in cloud computing[18] as seen as vertical integration levels in FIG. 1.1, namely *Infrastructure-as-a-Service* (IaaS), *Platform-as-a-Service* (PaaS) and *Software-as-a-Service* (SaaS). IaaS is on the lowest closest to physical hardware and SaaS on the highest level close to runnable applications.

IaaS. This layer is similar to more standard solutions such as *Virtual Private Servers* (VPS), and is therefore the *service model* closest to standard hosting solutions. Stanoevska-Slabeva [27] emphasizes that *”infrastructure had been available as a service for quite some time“* and this *”has been referred to as utility computing, such as Sun Grid Compute Utility“*. Which means IaaS can also be compared to grid computing, a well known computing model in the academic world.

> “ The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications.
>
> MELL AND GRANCE [18]

This underline the liberty this *service model* provides to users, but this also means that developers need to handle software and tools themselves, from operating system and up to their application. In some cases this is advantageous, for instance when deploying native libraries and tools that applications rely on such as tools to convert and edit images or video files. But in other cases this is not necessary and choosing this *service model* can be manpower in-effective for companies as developers must focus on meta tasks.

> “ The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, deployed applications, and possibly limited control of select networking components (*e.g.*, host firewalls).
>
> MELL AND GRANCE [18]

Users have control over which operating system they want, in some cases users can only pick from a set of pre-configured operating systems. It is common for providers to include both Linux and Windows in their selections. Some providers such as Amazon let users upload their own disk images. A similarity to VPS is that operating systems are not manually installed, when selecting an operating system this is copied directly into the instance pre-installed and will therefore be instantly ready for usage. Examples of providers of IaaS are AWS *Elastic Compute Cloud* (EC2) and Rackspace CloudServers.

PaaS. Cloud computing is built to guide and assist developers through abstractions, and the next layer in the *service model* is designed to aid developers by detaching them from configuration of operating system and frameworks. Developers are limited to capabilities the provider support, such as programming languages (Java, C#), environments (*Java Virtual Machine* (JVM), .NET, Node.js), storage systems (flat files, NoSQL databases, *Relation Database Management System* (RDBMS)), services (load balancers, backup, content delivery) and tools (plugin for Eclipse, command line tools) [18]. For example the first versions of *Google App Engine* (GAE) did only support an internal key-value based database called BigTable, which is still their main database. This database is transparently interfaced using their *Application Programming Interface* (API), but also support technologies such as *Java Persistence API* (JPA) and *Java Data Objects* (JDO), users are bound to Java and these frameworks, and even limitations to the frameworks as they have specific handlers for RDBMS. The disconnection from operating system is the strength of PaaS solutions, on one hand developers are restricted, but they are also freed from configuration, installments and maintaining deployments. Some PaaS providers support additional convenient capabilities such as test utilities for deployed applications and translucent scaling. In the end developers can put all focus on developing applications instead of spending time and resources on unrelated tasks.

PaaS providers support deployments through online APIs, in many cases by providing specific tools such as command line interfaces or plugins to *Integrated Development Environment* (IDE)s like Eclipse. It is common for the API to have client built on technologies related to the technology supported by the PaaS, for instance Heroku has a Ruby-based client and Nodejitsu has an executable Node.js-module as client.

Examples of PaaS providers are Google with GAE and the company Heroku with their eponymous service. Amazon also entered the PaaS market with their service named Elastic Beanstalk, which is an abstraction over EC2 as IaaS underneath. Multiple PaaS providers utilize EC2 as underlying infrastructure, examples of such providers are Heroku and Nodester, this is a tendency with increasing popularity.

SaaS. The highest layer of the *service models* farthest away from physical hardware and with highest level of abstraction.

“The capability provided to the consumer is to use the provider’s applications running on a cloud infrastructure.

MELL AND GRANCE [18]

The core purpose is to provide complete applications as services, in many cases end products. Google products such as Gmail, Google Apps and Google Calendar are examples of SaaS applications. What separates SaaS applications from other applications are the underlying cloud infrastructure, by utilizing the five characteristics of cloud computing SaaS applications achieve cloud computing advantages.

It is not imposed that SaaS deployments are web applications, they can also consist of different technologies such as *Representational state transfer* (REST) APIs or SOAP services, but in any case it is most common to utilize the HTTP. In SaaS applications end users are most likely not the companies renting from providers, but instead the companies customers. The abstraction layer covers most of all aspects around an application, the only exception could be customizations and settings that end users can do albeit this can be application specific. In some cases providers have services that affect these users as well, such as *Single Sign-on*.

1.1.3 Deployment models

Deployment models define where and how applications are deployed in a cloud environment, such as publicly with a global provider or private in local data centers. There are four main *deployment models*.

Public cloud. In this deployment model infrastructure is open to the public, so companies can rent services from cloud providers. Cloud providers own the hardware and rent out IaaS and PaaS solutions to users. Examples of such providers are Amazon with AWS and Google with GAE. The benefit of this model is that companies can save costs as they do not need to purchase physical hardware or manpower to build and maintain such hardware. It also means that a company can scale their infrastructure without having to physically expand their data center.

Private cloud. Similar to classical infrastructures where hardware and operation is owned and controlled by organizations themselves. This deployment model has arisen because of security issues regarding storage of data in public clouds. With *private cloud* organization can provide data security in forms such as geographical location and existing domain specific firewalls, and help comply requirements set by the government or other offices. Beside these models defined by NIST there is another arising model known as *Virtual Private Cloud* (VPC), which is similar to *public cloud* but with some security implications such as sandboxed network. With this solution companies can deploy cluster application and enhance or ensure security within the cluster, for example by disabling remote access to certain parts of a cluster and routing all data through safe gateways or firewalls. In *public clouds* it can be possible to reach other instances on a local network, also between cloud customers.

Community cloud. Similar as *private clouds* but run as a coalition between several organizations. Several organizations share the same aspects of a private cloud (such as security requirements, policies, and compliance considerations), and therefore share infrastructure. This type of *deployment model* can be found in universities where resources can be shared between other universities.

Hybrid cloud. One benefit is to distinguish data from logic for purposes such as security issues, by storing sensitive information in a private cloud while computing with public cloud. For instance a government can establish by law how and where some types of informations must be stored, such as privacy law. To sustain such laws a company could store data on their own *private cloud* while doing computation on a *public cloud.* In some cases such laws relates only to geographical location of stored data, making it possible to take advantage of *public clouds* that can guarantee geographical deployment within a given country.

1.2 Model-Driven Engineering

By combining the domain of cloud computing with the one of modeling it is possible to achieve benefits such as improved communication when designing a system and better understanding of the system itself. This statement is emphasized by Booch *et al.* in the UML:

“Modeling is a central part of all the activities that lead up to the deployment of good software. We build models to communicate the desired structure and behavior of our system. We build models to visualize and control the system’s architecture. We build models to better understand the system we are building, often exposing opportunities for simplification and reuse. We build models to manage risk.

[5]

When it comes to cloud computing these definitions are even more important because of financial aspects since provisioned nodes instantly draw credit. The definition of “modeling” can be assessed from the previous epigraph, but it is also important to choose correct models for the task. Stanoevska-Slabeva emphasizes in one of her studies that grid computing “*is the starting point and basis for Cloud Computing.*” [27]. As grid computing bear similarities towards cloud computing in terms of virtualization and utility computing it is possible to use the same *Unified Modeling Language* (UML) diagrams for IaaS as previously used in grid computing. The importance of this re-usability of models is based on the origin of grid computing, *eScience*, and the popularity of modeling in this research area. The importance of choosing correct models is emphasized by [5]:

“*(i)*The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped. *(ii)*Every model may be expressed at different levels of precision. *(iii)*The best models are connected to reality. *(iv)*No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

[5]

These definition precepts state that several models (precept *(iv)*) on different levels (precept *(ii)*) of precision should be used to model the same system. From this it is concludable that several models can be used to describe one or several cloud computing perspectives. Nor are there any restraints to only use UML diagrams or

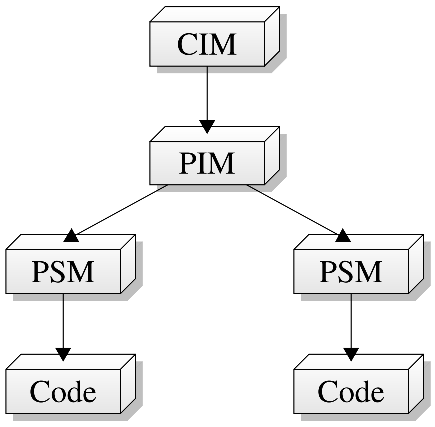


Figure 1.2: Model-Driven Architecture.

even diagrams at all. As an example AWS CloudFormation implements a lexical model of their *cloud services*, while CA AppLogic has a visual and more UML component-based diagram of their capabilities.

Model-Driven Architecture. *Model-Driven Architecture* (MDA) is a way of designing software with modeling in mind provided by the *Object Management Group* (OMG). When working with MDA it is common to first create a *Computation Independent Model* (CIM), then a *Platform-Independent Model* (PIM) and lastly a *Platform-Specific Model* (PSM) as seen in FIG. 1.2. There are other models and steps in between these, but they render the essentials. Beside the models there are five different steps as explained by Singh [26]:

1. *Create a CIM.* This is done to capture requirements and describe the domain. To do this the MDA developer must familiarize with the business organization and the requirements of this domain. This should be done without any specific technology. The physical appearance of CIM models can be compared to *use case* diagrams in UML, where developers can model actors and actions (use cases) based on a specific domain.

2. *Develop a PIM.* The next step aims at using descriptions and requirements from the CIM with specific technologies. The OMG standard for MDA use UML models, while other tools or practices might select different technologies. Example of such *Platform Independent Models* can be class diagrams in UML used to describe a domain on a technical level.

3. *Convert the PIM into PSM.* The next step is to convert the models into

something more concrete and specific to a platform. Several PSM and be used to represent one PIM as seen in FIG. 1.2. Examples of such models can be to add language specific details to PIM class diagram such as types (String, Integer) for variables, access levels (private, public), method return types and argument types. Kent, Stuart [16] emphasizes the importance of this mapping in one of his studies:

> “A PSM is not a PIM, but is also not an implementation. [...] In MDA, the main mapping is between PIM and PSM, and this is often associated with code generation. However, this is not the only kind of mapping required.
>
> KENT [16]

From this it is possible to determine that a PSM is more specific to a platform then PIM, such as programming language or environment.

4. *Generate code form PSM.* A PSM should be specific enough that code can be generated from the models. For instance can class diagrams be generated into entities, and additional code for managing the entities can be added as well. Some diagrams such as *Business Process Model and Notation* (BPMN) can generate *Business Process Execution Language* (BPEL) which again can generate executable logic.

5. *Deploy.* The final step is based on deploying the PSM, which concludes the five steps from loosely descriptions of a domain to a running product. Different environmental configurations can be applied in step 4 to assure deployments on different systems, this without changing the PSM from step in step 3.

Chapter 2

State of the Art in Provisioning

There already exists scientific research projects, APIs, frameworks and other technologies which aim at consolidating, interfacing and utilizing cloud technologies. This chapter introduces some of these concepts, and does this by dividing the chapter into four parts:

1. Model-Driven Approaches which aims presenting frameworks and projects that utilize models on a larger scale.

2. APIs are about frameworks that connects to cloud providers, often with multicloud support, these projects can be used as middleware for other systems.

3. Deployments are about projects that do full deployment of applications, inclusing provisioning. These are often more academic. Lastly the chapter will discuss *(v)*Example of cloud surveys, some real-world examples that might not be directly related to provisioning but are important and interesting regardless.

2.1 Model-Driven Approaches

Amazon AWS CloudFormation. [1]

This is a service provided by Amazon from their popular AWS. It give users the ability to create template files in form of JSON as seen in FIG. 2.1, which they can load into AWS to create stacks of resources. A *stack* is a defined set of resources in different amount and sizes, such as numerous instances, one or more databases and a load balancer, although what types and sizes of resources

```

"Description": "Create an EC2 instance",
"Parameters": {
  "KeyPair": {
    "Description": "For SSH access",
    "Type": "String"
  }
},
"Resources": {
  "Ec2Instance": {
    "Type": "AWS::EC2::Instance",
    "Properties": {
      "KeyName": { "Ref": "KeyPair" },
      "ImageId": "ami-1234abcd"
    }
  }
},
"Outputs" : {
  "InstanceId": {
   "Description": "Instace ID of created instance",
   "Value": { "Ref": "Ec2Instance" }
  }
},
"AWSTemplateFormatVersion": "2010-09-09"

```

Figure 2.1: AWS CloudFormation template.

is ambiguous. To provision a stack with CloudFormation the template file (in JSON format) is first uploaded to AWS which makes it accessible from AWS Management Console.

The template consist of three main sections, *(i)Parameters*, *(ii)Resources* and *(iii)Outputs*. The *Parameters* section makes it possible to send parameters into the template, with this the template becomes a macro language by replacing references in the *Resources* section with inputs from users. Input to the parameters are given inside the management console when provisioning a stack with a given template. The *Resource* section define types of resources that should be provisioned, the *Type* property is based on a set of predefined resource types such as *AWS::EC2::Instance* in Java package style. The last section, *Output*, will generate output to users when provisioning is complete, here it is possible for

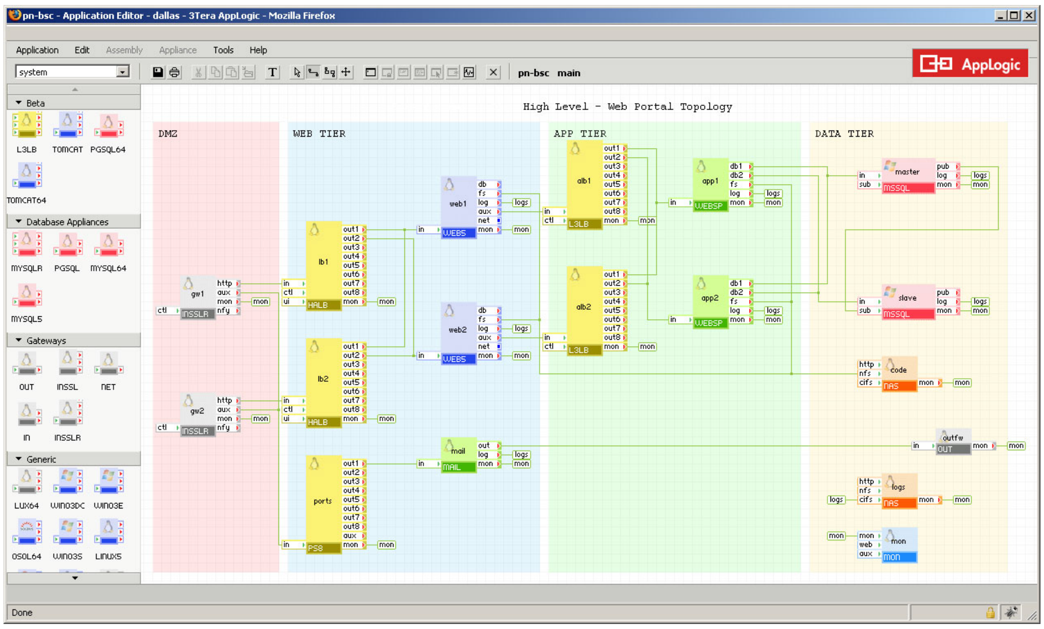


Figure 2.2: CA Applogic screenshot.

users to pick from a set of variables to get the information they need.

This template system makes it easier for users to duplicate a setup many times, and as the templates support parameters this process can be as dynamic as the user design it to be. This is a model in form or lexical syntax, both the template itself and the resources that can be used. For a company that is fresh in the world of cloud computing this service could be considered too advance. This is mainly meant for users that want to replicate a certain stack, with the ability to provide custom parameters. Once a stack is deployed it is only maintainable through the AWS Management Console, and not through template files. The format that Amazon uses for the templates is a good format, the syntax is in form of JSON which is readable and easy to use, but the structure and semantics of the template itself is not used by any other providers or cloud management tooling, so it can not be considered a multicloud solution. Even though JSON is a readable format, it does not make it viable as a presentation medium on a business level.

CA Applogic. [9]

The Applogic platform is designed to manage CAs private cloud infrastructure [24]. It also has a web based interface which let users manage their cloud resources as shown in FIG. 2.2 which use and benefit from a model based approach. It is based on graphical models which support interactive “drag and drop” func-

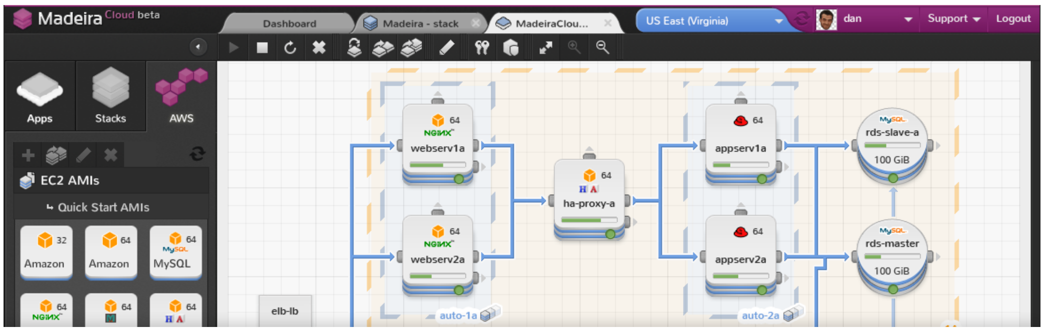


Figure 2.3: Madeira Cloud screenshot.

tionalities. This interface let users configure their deployments through a diagram with familiarities to UML component diagrams with interfaces and assembly connectors. They let users configure a selection of third party applications, such as Apache and MySQL, as well as network security, instances and monitoring. What CA has created is both an easy way into the cloud and it utilizes the advantages of model realizations. Their solution will also prove beneficial when conducting business level consulting as it visualizes the structural layout of an application. But this solution is only made for private clouds running their own controller, this can prove troublesome for migration, both in to and out of the infrastructure.

Madeira Cloud. [17]

Madeira have created a tool which is similar to CA Applogic, but instead of focusing on a private cloud solution they have created a tool specifically for AWS EC2. Users can create *stacks* with the available services in AWS through dynamic diagrams. Like CA Applogic their tool is also proprietary, but on the other hand they support a free account which can be used to test their service out. One would also need a AWS account as an addition.

These *stacks* are live representations of the architecture and can be found and managed in the AWS console as other AWS services. They also support storing running systems into template files which can be used to redeploy identical copies and it should also handle configuration conflicts. For identifying servers they use hostnames, which are bound to specific instances so end users don't have to bother with IP addresses.

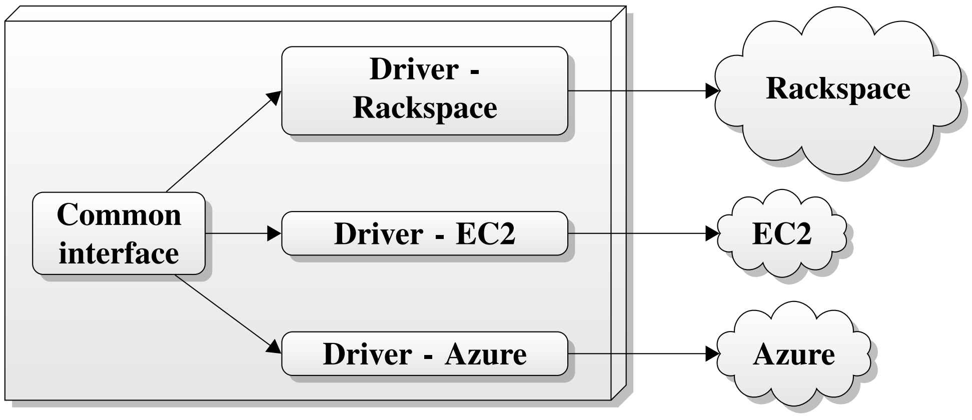


Figure 2.4: Cloud drivers.

2.2 APIs

Extensive work have been done towards simplifying and combining cloud technologies through abstractions, interfaces and integrations. Much of this work is in form of APIs, mostly in two different forms. Either as programming libraries that can be utilized directly from a given programming language or environment such as Java or Python. The other common solution is to have an online facade against public providers, in this solution the APIs are mostly in REST form. REST [12] is a software architecture for management of web resources on a service. It uses the HTTP and consecutive methods such as GET, POST, PUT and DELETE to do tasks such as retrieve lists, items and create items. APIs can be considered modeling approaches based on the fact they have a topology and hierarchical structure, but it is not a distinct modeling. A modeling language could overlay the code and help providing a clear overview, but the language directly would not provide a good overview of deployment. And links between resources can be hard to see, as the API lacks correlation between resources and method calls.

Driver. All the API solutions use the term "*driver*", it represents a module or component that fit into existing software and extend the support of external connections without changing the interface. A cloud *driver* connects a given software to an existing cloud provider through this providers web based API (REST), illustrated in FIG. 2.4.

jclouds. [15]

This is a library written in Java and can be used from any JVM-based language. Provider support is implemented in *drivers*, and they even support deployments to some PaaS solutions such as GAE. *jclouds* divide their library in two different parts, one for computing powers such as EC2 and one for *blob* storage like S3. Some blob storage services are accessible on the compute side of the library such as *Elastic Block Storage* (EBS). They support "dry runs" so a stack can be deployed as a simulation, not actually deploying it to a public cloud. This is beneficial for testing deployments, and writing unit tests without initializing connections, the library enhance this by providing a stub aimed at testing.

libcloud. [3]

Libcloud is an API that aims to support the largest cloud providers through a common API. The classes are based around *drivers* that extends from a common ontology, then provider-specific attributes and logic is added to the implementation. Libcoud is very similar to jclouds but the API code base is written in Python. The API is Python-only and could therefor be considered to have high tool-chain dependency.

Deltacloud. [2]

Deltacloud has a similar procedure as jclouds and libcloud, but with a REST API. So they also work on the term *driver*, but instead of having a library to a programming language the users are presented with an web-based API they can call on Deltacloud servers. As well as having similar problems as other APIs this approach means that every call has to go through their servers, similar to a proxy. This can work with the benefits that many middleware softwares have, such as caching, queues, redundancy and transformations. The main disadvantages are single point of failure and version inconsistencies. Deltacloud provide two sets of native libraries, one in Ruby and another in C, which makes it easier to communicate with the REST API. Previously discussed *jclouds* also support Deltacloud, as it would interface this with a *driver* as any other web-based API.

2.3 Deployments

There are also some solutions that specifically aim at full deployments, contra provisioning single instances or services these solutions provision everything needed to fully deploy an application with a given ontology.

mOSAIC. [22]

Aims at not only provisioning in the cloud, but deployment as well. They focus on abstractions for application developers and state they can easily enable users to *“obtain the desired application characteristics (like scalability, fault-tolerance, QoS, etc.)”* [21]. There are two abstraction layers, one for cloud provisioning and one for application-logic. mOSAIC will select a proper cloud based on how developers describe their application, several clouds can be selected based on their properties. mOSAIC will use the IaaS solutions of cloud providers to deploy users application, then communication between these clouds will be done using “cloud based message queues technologies”.

RESERVOIR. [25]

Resources and Services Virtualization without Barriers (RESERVOIR) is a European Union FP7 project, aiming at *cloud federation* between private and hybrid clouds. With this a deployed application can distribute workload seamlessly between private and public clouds based on the applications requirements. This is done by creating *Reservoir sites*, one for each provider. Each site is independent and run a *Virtual Execution Environment* (VEE) which is managed by a *Virtual Execution Environment Manager* (VEEM). The VEEM communicate with other VEEM and are able to do federation between clouds. Each site must have the Reservoir software components installed, which makes it self-maintainable and self-monitoring.

Vega. [10]

Vega framework is a deployment framework aiming at full cloud deployments of multi-tier topologies, they also follow a model-based approach. The description of a given topology is done by using *eXtensible Markup Language* (XML) files, with these files developers can replicate a *stack*. The XML contain information about the instances, such as *ostype* for Operating System and *image-description* to describe properties of an instance such as amount of memory (*req_memory*). They also allow small scripts to be written directly into the XML through a node *runoncescript* which can do some additional configuration on a propagated node. A resource manager keep track of resources in a system, grouping instances after their attributes.

2.4 Example of cloud surveys

In this section real-world examples are presented, such as popular IaaS and PaaS solutions and other technologies which are widely used today. Some solutions bear strong similarities to others, such as EC2 and Rackspace cloudservers, for these only one solution will be discussed.

EC2. A central part of AWS, it was Amazons initial step into the world of cloud computing when they released EC2 as a service as public beta in 2006. This service is known to be the most basic service that cloud providers offer and is what makes Amazon an IaaS provider. When users rent EC2 services they are actually renting VPS instances virtualized by Xen. Although the instance itself can be considered a VPS there are other factors that define it as a cloud service. For instance cost calculations, monitoring and tightly coupled services surrounding EC2. Examples of these services are EBS for block storage, *Elastic Load Balancer* (ELB) for load balancing and *Elastic IP* for dynamically assigning static IP addresses to instances.

Some of AWS other services rely on EC2 such as AWS Elastic Beanstalk and *Elastic MapReduce*. When purchasing these services it is possible to manage the EC2 instances through the EC2-tab in AWS console, but this is not mandatory as they will be automatically managed through the original purchased service. As mentioned earlier other PaaS solutions delivered by independent companies are using EC2 or other AWS solutions. Examples of these are Heroku, Nodester, DotCloud and Engine Yard which uses EC2. Example of companies using other AWS services is Dropbox which uses S3.

EC2 is similar to services offered by other providers, such as Rackspace cloudservers, GoGrid cloud servers and Linode Cloud. Some of the additional services such as ELB can also be found in other providers which also offer these capabilities as services.

Amazon Beanstalk. Amazon has been known for providing IaaS solutions (EC2) and services complementing either their IaaS or the *Storage as a Service* solution S3. Unlike some providers such as Microsoft and Google they had yet to introduce a PaaS based solution, until they created Beanstalk. This is the Amazon answer to PaaS, it is based on pre-configuring a stack of existing services such as EC2 for computing, EBS for storage and ELB for load balancing. At the writing moment they support Java with Tomcat and PHP deployments. The Java solution is based on uploading war-files to Beanstalk, then the service will handle the rest of the deployment. For PHP the deployment is based on Git repositories, when pushing code to a given repository Beanstalk will automatically deploy the new code to an Apache httpd instance.

Chapter 3

Challenges in the cloud

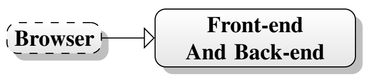
As cloud computing is growing in popularity it is also growing in complexity. More and more providers are entering the market and different types of solutions are made. There are few physical restrictions on how a provider should let their users do provisioning, and little limitations in technological solutions. The result can be a complex and struggling introduction to cloud computing for users, and provisioning procedure can alternate between providers.

This chapter will outline research on which has been conducted by physical provisioning of an example application. First the scenario will be introduced, describing the example application and different means of provisioning in form of topologies. Then the challenges identified from the research will be presented.

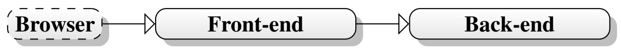
3.1 Scenario

The following scenario is chosen because of how much it resembles actual solutions used in industry today. It uses a featureless example application meant to fit into scenario topologies without having too much complexity. Challenges should not be affected from errors or problems with the example application. The application will be provisioned to a defined set of providers with a defined set of different topologies.

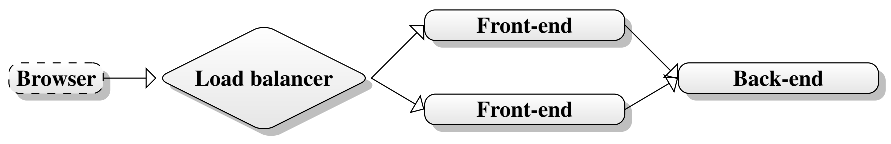
BankManager. To recognize challenges when doing cloud provisioning an example application [6] is utilized. The application (from here known as *BankManager*) is a prototypical bank manager system which support creating users and bank accounts and moving money between bank accounts and users. The application is based on a three-tier architecture with *(i)*presentation tier with



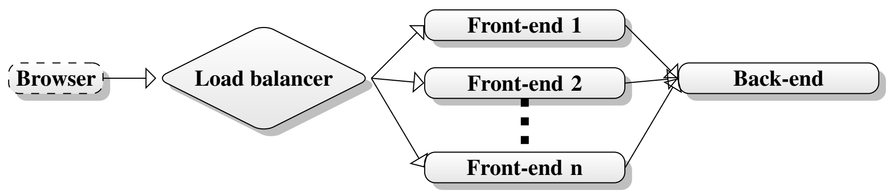
(a) Single node.



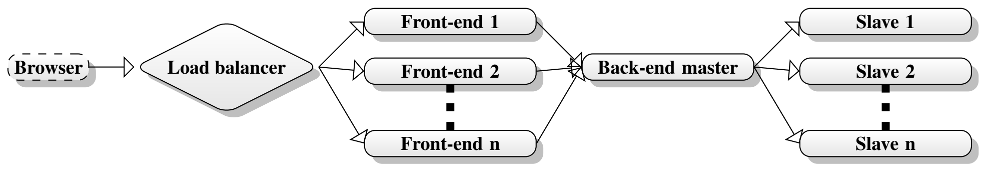
(b) Two nodes.



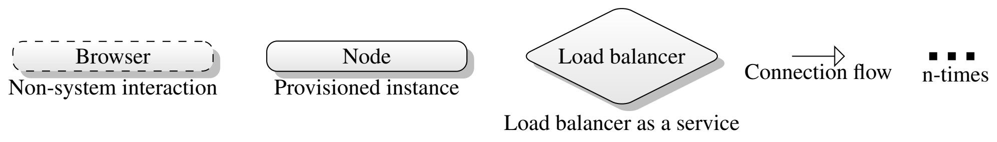
(c) Three nodes.



(d) Several front-ends.



(e) Several front-ends and back-ends (slaves).



(f) Legend.

Figure 3.1: Different architectural ways to provision nodes (topologies).

a web-based interface, *(ii)*logic tier with controllers and services and *(iii)*database tier with models and entities. Three or more tiers in a web application is a common solution, even more so for applications based on the *Model View Controller* (MVC) architectural pattern. The advantage with this architecture is that the lowest tier (database) can be physically detached from the tiers above, the application can then be distributed between several nodes. It is also possible to have more tiers, for instance by adding a *service* layer to handle re-usable logic. Having more tiers and distributing these over several nodes is an architecture often found in *Service-Oriented Architecture* (SOA) solutions.

Topologies. Some examples of provisioning topologies are illustrated in FIG. 3.1. Each example includes a `browser` to visualize application flow, `front-end` visualizes executable logic and `back-end` represents database. It is possible to have both `front-end` and `back-end` on the same node, as shown in FIG. 3.1(a). When the topology have several `front-ends` a `load balancer` is used to direct traffic between `browser` and `front-end`. The `load balancer` could be a node like the rest, but in this cloud-based scenario it is actually a cloud service, which is also why it is graphically different. In FIG. 3.1(b) `front-end` is separated from `back-end`, this introduces the flexibility of increasing computation power on the `front-end` node while spawning more storage on the `back-end`. For applications performing heavy computations it can be beneficial to distribute the workload between several `front-end` nodes as seen in FIG. 3.1(c), the number of `front-ends` can be linearly increased n number of times as shown in FIG. 3.1(d). *BankManager* is not designed to handle several `back-ends` because of RDBMS, this can be solved at the database level with master and slaves (FIG. 3.1(e)).

Execution. The main goal of the scenario is to successfully deploy *BankManager* on a given set of providers with a given set of topologies. And to achieve such deployment it is crucial to perform cloud provisioning. The providers chosen are *(i)*AWS [1] and *(ii)*Rackspace [23]. These are strong providers with a respectable amount of customers, as two of the leaders in cloud computing. They also have different graphical interfaces, APIs and toolchains which makes them suitable for a scenario researching multicloud challenges.

The topology chosen for this scenario is that of three nodesFIG. 3.1(c). This topology is advance enough that it needs a `load balancer` in front of two `front-end` nodes, and yet the simplest topology of the ones that benefits from a `load balancer`. It is important to include most of the technologies and services that needs testing.

To perform the actual provisioning a set of primitive Bash-scripts are developed. These scripts are designed to automate a full deployment on a two-step basis. First step is to provision instances:

- Authenticate against provider.
- Create instances.
- Manually write down IP addresses of created instances.

The second step is deployment:

- Configure *BankManager* to use one of provisioned instances IP address for database.
- Build *BankManager* into a *Web application Archive* (WAR)-file.
- Authenticate to instance using SSH.
- Remotely execute commands to install required third party software such as Java and PostgreSQL.
- Remotely configure third party software.
- Inject WAR-file into instances using *SSH File Transfer Protocol* (SFTP).
- Remotely start *BankManager*.

The scripts are provider-specific so one set of scripts had to be made for each provider. Rackspace had at that moment no command-line tools, so a REST client had to be constructed.

3.2 Challenges

From this example it became clear that there were multiple challenges to address when deploying applications to cloud infrastructure. This thesis is scoped to cloud provisioning, but the goal of this provisioning step is to enable a successful deployment, see CHAP. 11. It is therefore crucial to involve a full deployment in the scenario to discover important challenges.

Complexity. The first challenge encountered is to simply authenticate and communicate with the cloud. The two providers had different approaches, AWS [1] had command-line tools built from their Java APIs, while Rackspace [23] had no tools beside the API language bindings. So for AWS the Bash-scripts could do callouts to the command-line interface while for Rackspace the public REST API had to be utilized. This emphasized the inconsistencies between providers, and resulted in an additional tool being introduced to handle requests.

As this emphasizes the complexity even further it also stresses engineering capabilities of individuals. It would be difficult for non-technical participants to fully understand and give comments or feedback on the topology chosen since important information got hidden behind complex commands.

Feedback on failure. Debugging the scripts is also a challenging task, since they fit together by sequential calls and printed information based on Linux and Bash commands such as *grep* and *echo*. Error messages from both command-line and REST interfaces are essentially muted away. If one specific script should fail it would be difficult to know *(i)*which script failed, *(ii)*at what step it was failing and *(iii)*what was the cause of failure .

Multicloud. Once able to provision the correct amount of nodes with desired properties on the first provider it became clear that mirroring the setup to the other provider is not as convenient as anticipated. There are certain aspects of vendor lock-in, so each script is hand-crafted for specific providers. The most noticeable differences would be *(i)*different ways of defining instance sizes, *(ii)*different versions, distributions or types of operating systems (*images*), *(iii)*different way of connection to provisioned instances . The lock-in situations can in many cases have financial implications where for example a finished application is locked to one provider and this provider increases tenant costs. Or availability decreases and results in decrease of service uptime damaging revenue.

Reproducibility. The scripts provisioned nodes based on command-line arguments and did not persist the designed topology in any way, this made topologies cumbersome to reproduce. If the topology could be persisted in any way, for example serialized files, it would be possible to reuse these files at a later time. The persisted topologies could also be reused on other clouds making a similar setup at another cloud provider, or even distribute the setup between providers.

Shareable. Since the scripts did not remember a given setup it is impossible to share topologies "as is" between coworkers. It is important that topologies can be shared because direct input from individuals with different areas of competence can increase quality. If the topology could be serialized into files these files could also be interpreted and loaded into different tools to help visualizing and editing.

Robustness. There are several ways the scripts could fail and most errors are ignored. They are made to search for specific lines in strings returned by the APIs, if these strings are non-existent the scripts would just continue regardless of complete dependency to information within the strings. A preferable solution to this could be transactional behavior with rollback functionality in case an error should occur, or simply stop the propagation and throw exceptions that can be handled on a higher level.

Metadata dependency. The scripts were developed to fulfill a complete deployment, including *(i)*provisioning instances, *(ii)*install third party software on instances, *(iii)*configure instances and software, *(iv)*configure and upload WAR-file and *(v)*deploy and start the application from the WAR-file . In this thesis the focus is aimed at provisioning, but it proved important to temporally save run-time specific metadata to successfully deploy the application. In the *BankManager* example the crucial metadata is information needed to connect front-end nodes with the back-end node, but other deployments is likely to need the same or different metadata for other tasks. This metadata is collected in *(i)*, and used in *(iii)* and *(iv)*.

3.3 Summary

There are many cloud providers on the global market today. These providers support many layers of cloud, such as PaaS and IaaS. This vast amount of providers and new technologies and services can be overwhelming for many companies and small and medium businesses. There are no practical introductions to possibilities and limitations to cloud computing, or the differences between different providers and services. Each provider has some kind of management console, usually in form of a web interface and API. But model driven approaches are inadequate in many of these environments. UML diagrams such as deployment diagram and component diagram are used in legacy systems to describe system architectures, but this advantage has yet to hit the mainstream of cloud computing management. It is also difficult to have co-operational interaction on a business level without using the advantage of graphical models. The knowledge needed to handle one provider might differ to another, so a multicloud approach might be very resource-heavy on competence in companies. The types of deployment resources are different between the providers, even how to gain access to and handle running instances might be very different. Some larger cloud management application developers are not even providers themselves, but offer tooling for private cloud solutions. Some of these providers have implemented different types of web based applications that let end users manage their cloud instances. The main problem with this is that there are no standards defining a cloud instance or links between instances and other services a provider offer. If a provider does not offer any management interface and want to implement this as a new feature for customers, a standard format to set the foundation would help them achieve a better product for their end users.

Chapter 4

Requirements

In CHAP. 3 challenges were identified, in this chapter those challenges will be addressed and tackled through requirements. The requirements are descriptions of important aspects and needs derived from the previous chapter, A table overview will display consecutive challenges and requirements. This table is angled to the challenges point of view to clarify requirements relation to challenges, and one requirement can try to solve several challenges.

$\rightsquigarrow$ R_1 : **Software reuse.** There were several technological difficulties with the scripts from the scenario in CHAP. 3. And one requirement that could leverage several of the challenges originating from these particular issues would be to utilize an existing framework or library. If possible it would be beneficial to not *"Reinvent the wheel"* and rather use work that others have done that solve the same problems. In the chapter CHAP. 2 multicloud APIs were described, such as *libcloud* and *jclouds*. The core of this requirement is to find and experiment with different APIs to find one that suite the needs to solve some of the challenges from CHAP. 3. One of these challenges would be *complexity* where such software utilization could help to authenticate to providers and leverage understanding of the technology. Such library could also help with *feedback* in case an exception should occur, on one side because the error handling would be more thoroughly tested and used, and another side because the library would be more tightly bounded with R_2. And for the same reasons such framework could make the whole application more *robust*.

All of the API libraries from CHAP. 2 support *multicloud* so they can interact with several providers over a common interface, this would be a mandatory challenge to overcome by this requirement. Some research have already been done to indicate what their purpose is, how they operate and to some extent how to use them. The approach here is to select libraries or framework that could match well with a chosen R_2 or help fulfill this requirement. Then the chosen APIs must be narrowed down to one single API which will be used in the solution application.

$\leadsto$ R_2 : **Strong technological foundation.** Beside the benefits of R_1 (*software reuse*) there could be even additional gain by choosing a solid technology underneath the library, *e.g.*, *programming language*, *application environment*, *common libraries*, *distribution technologies*. The core of this requirement is to find, test and experiment with technologies that can solve challenges and even give additional benefits. Such technologies could be anything from Java for enterprise support to open source repository sites to support software distribution. It is also important that such technologies operate flawlessly with libraries or frameworks found and chosen from the requirement of R_1 (*software reuse*). The technology chosen should benefit the challenge of *robustness*. It could also help to solve other challenges such as *metadata dependency* by introducing functionality through *common libraries* or some built in mechanism.

Solid technologies have to be considered by several aspects, such as *(i)*ease of use, *(ii)*community size, *(iii)*closed/open source, *(iv)*business viability, *(v)*modernity and *(vi)*matureness. Another important aspect is based on library or framework chosen for the R_1 (*software reuse*) requirement, as the library will directly affect some technologies such as programming language. Different technologies have to be researched and to some degree physically tried out to identify which aspects they fulfill.

Types of technologies are undefined but some are mandatory such as *(i)*programming language (Java, C#) and *(ii)*application environment (JDK, .NET). Beside this it is also important to state which type of application the solution should be, *(i)*GUI application, *(ii)*API in form of public Web Service or *(iii)*API in form of native library. The amount of different technologies is overwhelming so looking into all of them would be impossible, therefore they must be narrowed down based on aspects such as popularity.

$\leadsto$ R_3 : **Model-Driven approach.** Models can be reused to multiply a setup without former knowledge of the system. They can also be used to discuss, edit and design topologies for propagation. These are important aspects that can help to leverage the challenge of *complexity*.

Main objective is to create a common model for nodes as a platform-independent model [20] to justify *multicloud* differences and at the same time base this on a human readable lexical format to address *reproducibility* and make it *shareable*.

Unlike the other requirements this is a non-physical need, and as seen in FIG. 4.1 there are no dependencies from or to this requirement. But other requirements such as R_4 are directly based on this one.

In the implementation there will be four different models:

1. The lexical template.

2. Nodes from the template represented in the implementation.

3. Nodes converted into *instances* for provisioning.

4. Instances in form of runtime instances (*models@run.time*).

Of these the first is influenced by the R_4 requirement and the last by the R_5 requirement.

$\rightsquigarrow R_4$: **Lexical template.** This requirement is tightly coupled with that of R_3 (*model-driven approach*) but narrowed even further to state the importance of model type in regard to the model-driven approach. When approaching a global audience consisting of both academics groups and commercial providers it is important to create a solid foundation, which also should be concrete and easy to both use and implement. The best approach would be to support both graphical and lexical models, but a graphical annotation would not suffice when promising simplicity and ease in implementation. Graphical model could also be much more complex to design, while a lexical model can define a concrete model on a lower level. Since the language will be a simple way to template configuration, a well known data markup language would be sufficient for the core syntax, such as JSON or XML.

Textual templates that can be shared through mediums such as E-mail or *Version Control System* (VCS) such as Subversion or Git. This is important for end users to be able to maintain templates that defines the stacks they have built, for future reuse.

The type of direct model representation of topologies will have great impact on the solution application. As described in CHAP. 4 this representation should be lexical, but there are several different styles and languages to achieve this.

Some examples of these languages are *(i)*XML, *(ii)*JSON, *(iii)*YAML, *(iv)Simple Declarative Language* (SDL) or *(v)Ordered Graph Data Language* (OGDL). As shown in FIG. 4.1 there is a two-way dependency between this requirement and R_2 (*strong technological foundation*) requirement. This dependency can have impact both ways, but unlike the other dependencies in FIG. 4.1 there exist bindings all the four precedings in most languages and systems. Templates could even be stored as any binary type of serialization, but this might not be as sufficient as lexical types, more on this in CHAP. 6.

$\rightsquigarrow$ R_5 : **Models@run.time.** Models that reflect the provisioning models and updates asynchronously. As identified by the scenario in CHAP. 3 metadata from provisioning is crucial to perform a proper deployment in steps after the provisioning is complete. One way to solve this issue is by utilizing *models@run.time*, which is the most obvious choice in a model-driven approach. Models will apply to several parts of the application, such as for topology designing and for the actual propagation.

The models@run.time are meant to support any deployment system which should be run sequentially after a complete provisioning. For such deployment to be successful metadata from the provisioning could be needed, so the core idea is to fetch this kind of data directly from models@run.time.

The approach for this requirement is to find sufficient solutions for such models, and at the same time keep in mind the dependency towards R_2 (*strong technological foundation*). There are several different approaches that could be made when implementing these models, such as using *(i)*observer pattern, *(ii)*command pattern, *(iii)*actor model or *(iv)*publish-subscribe pattern. It is also possible to combine one or several of these approaches. What needs to be done here is to identify which approaches that are most sufficient in regards to *(i)*finding an approach that solved the requirement, *(ii)*sustain constraints in regard of dependencies as seen in FIG. 4.1, and *(iii)*identify approaches that can be combined and what benefits this would give.

$\rightsquigarrow$ R_6 : **Multicloud.** One of the biggest problems with the cloud today is the vast amount of different providers. There are usually few reasons for large commercial delegates to have support for contestants. Some smaller businesses could on the other hand benefit greatly of a standard and union between providers. The effort needed to construct a reliable, stable and scaling computer park or data center will withhold commitment to affiliations. Cloud computing users are

Challenge	**Addressed by**
Complexity	R_1 (*software reuse*) and R_3 (*model-driven approach*)
Feedback on failure	R_1 (*software reuse*)
Multicloud	R_1 (*software reuse*)
Reproducibility	R_4 (*lexical template*)
Sharable	R_4 (*lexical template*)
Robustness	R_1 (*software reuse*) and R_2 (*strong technological foundation*)
Metadata dependency	R_5 (*models@run.time*) and R_2 (*strong technological foundation*)

Table 4.1: Requirements associated with challenges.

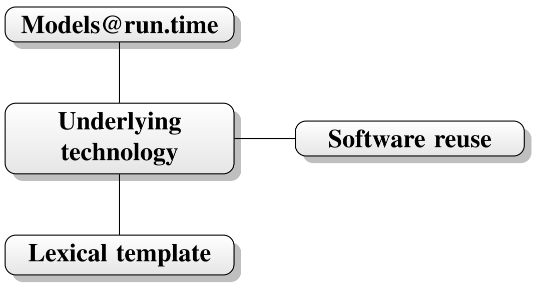
concerned with the ability to easily swap between different providers, this because of security, independence and flexibility. CloudML and its engine need to apply to several providers with different set of systems, features, APIs, payment methods and services. This requirement anticipate support for at least two different providers such as AWS and Rackspace.

4.1 Comparison

The requirements defined in this chapter are designed to tackle one or more of the challenges described in CHAP. 3. All of the challenges are associated with requirements, as seen in TABLE. 4.1. Three of the challenges are tackled by more than one requirement, and three other requirements tackle more than one challenge. *e.g.*, R_1 (*software reuse*) tackle four different challenges including *complexity*, while the challenge *complexity* is tackled by both R_1 (*software reuse*) and R_3 (*model-driven approach*).

4.2 Requirement dependencies

Some of the requirements have depend on each other, for instance R_1 (*software reuse*) is about finding and utilizing an existing library or framework, but this will also directly affect or be affected by programming language or application environment chosen in R_2 (*strong technological foundation*) requirement. There are three requirements, *(i)* R_5 (*models@run.time*), *(ii)* R_1 (*software reuse*) and *(iii)* R_4 (*lexical template*), where all have a two-way dependency to the *(iv)*



(a) Requirement dependencies.

(b) Legend.

Figure 4.1: Requirement dependencies.

R_2 (*strong technological foundation*) requirement, as illustrated in FIG. 4.1. These dependency links will affect the end result of all the four previous mentioned requirements. For example a library chosen in precept *(ii)* would affect precept *(iv)*, which again would affect how precept *(i)* will be solved. It could also affect precept *(iii)* as different textual solutions can function better in different environments. Since R_2 (*strong technological foundation*) is a central dependency (FIG. 4.1) this requirements is weighted more than the others.

Part II

Contribution

Chapter 5

Vision, concepts and principles

In this chapter the core idea of CloudML will be presented. The concept and principle of CloudML is to be an easier and more reliable path into cloud computing for IT-driven businesses of variable sizes. The tool is visioned to parse and execute template files representing topologies and provision these as instances available in the cloud.

The vision of CloudML is reflected through FIG. 5.1 which gives an overview of the procedure flow in and around CloudML.

Domain of CloudML. Inside FIG. 5.1 there is an area pointed out as *CloudML*, this area contain components necessary to implement in order to fulfill the vision as a whole. Every part within the designated area is some physical aspect in the implementation, and therefore core parts of the contribution.

The actors. In FIG. 5.1 there are three actors, *(i)*business person representing someone with administration- or manager position which defines and controls demands for application functionality and capabilities. The next actor, *(ii)*cloud expert has a greater knowledge of the cloud domain eg cloud providers, services these offer, limitations, API support and prices. The last actor, *(iii)*user is a person which directly utilize CloudML to do provisioning. This physical person may or may not have the role of cloud expert, hence the the cloud expert extends from the user actor.

Application and topologies. The business person is in charge of the application, he/she has a need for an application that can fulfill certain tasks, and to handle these tasks application demands are made. The cloud expert use the requirements

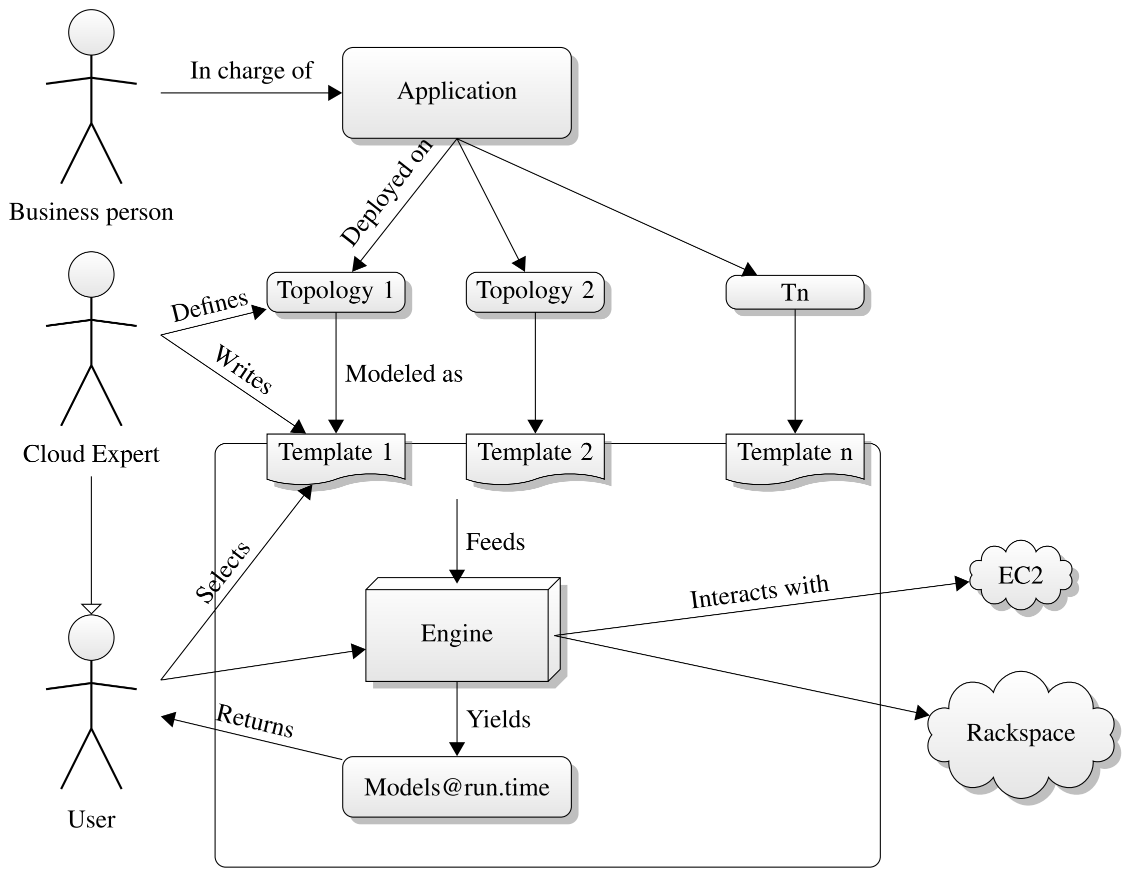


Figure 5.1: "Big picture", overview of CloudML procedure flow.

sketched by the business person to define and design node topologies which tackles the application demands. A topology is a map of nodes connected together in a specific pattern, defined by the cloud expert. In a topology there is also information about node attributes *e.g.*, CPU power and RAM sizes. He/she might create several topologies to fulfill the application demands.

Templates. The next step is to create templates based on the topologies, this is done by the cloud expert. A template is a digital reflection of a topology including the attributes and some additional information such as node names and template labeling. It is also possible to define more than one topology within a single template.

Engine. When the cloud expert have designed and created the necessary templates the next actor, user, will continue the procedure. The user selects the template and feeds them into the engine. The engine is the core of the implementation, handling several steps and executing most of the CloudML logic. The engine operates according to five different steps:

1. Convert the template files into a native format for later use.
2. Convert pure nodes into instances ready for provisioning.
3. Connect to all the desired providers.
4. Propagate the instances.
5. Produce models@run.time of the instances being propagated.

Providers. The engine interacts with the providers, in FIG. 5.1 EC2 and Rackspace are selected as examples, but any cloud provider that will be supported by CloudML can be utilized. As discussed in CHAP. 2 and CHAP. 3 different providers implement different solutions for communication *e.g.*, for provisioning nodes, managing nodes and services or terminating instances. For the engine to interact with a set of different providers, a tool, library or framework is needed. This additional software can connect to the different providers through a common interface.

Models@run.time. The last part of this implementation of CloudML (see CHAP. 11) is to reflect provisioned instances with models@run.time. These models are returned to the user when provisioning starts, and when attributes and statuses about instances are updated the user is notified about these updates through the models. In the implementation the models can extend from or aggregate *instances*.

Chapter 6

Analysis and design - CloudML

In the previous chapter, CHAP. 5, the core vision of CloudML was presented, including descriptions of surrounding elements, *e.g.*, actors and topologies. In this chapter the focus is narrowed down to the design and considerations of the implementation that constitutes CloudML.

6.1 Meta-model

In this section the meta-model of CloudML will be presented. The model is visualized in FIG. 6.1, and will be described through a specific scenario. The scenario also describe parts of the implementation design through how it is used.

Scenarios introduction. CloudML is introduced by using two different scenarios where a user named "Alice" is provisioning the *BankManager* application from CHAP. 3 to the AWS cloud. It is compulsory that she possesses an AWS account in advance of the scenario. This is essential information needed for the scenario to be successful, and since she is indirectly using AWS APIs, she must also have *security credentials*, *i.e.*, *Access Key ID* and *Secret Access Key*.

The roles assumed by Alice in this scenario, regarding FIG. 5.1, are both *cloud expert* and *users*. She will define the topologies, create the templates and use the *engine* to provision her models. In addition to these roles she is also partly *application designer/developer*, because of tight coupling between running instances and application deployment.

Authentication. She associates *Access Key ID* and *Secret Access Key* with `Credential` and `Password` in FIG. 6.1. `Credential` is used to authenticate

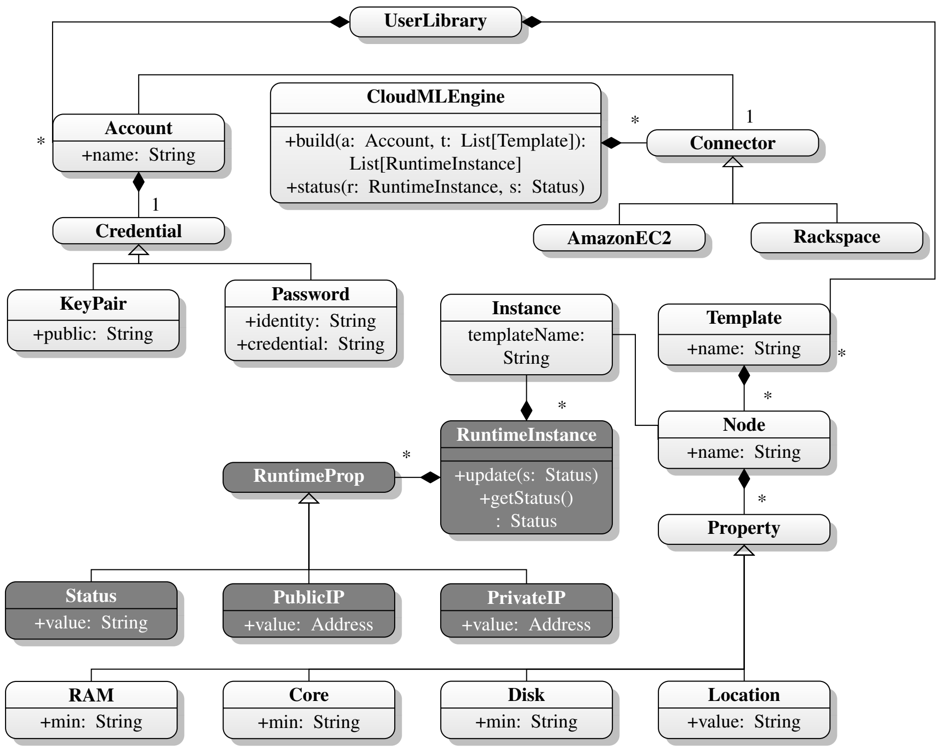


Figure 6.1: Meta model of CloudML.

her to supported providers through `Connector`. The `Connector` is a common interface against supported providers. This component of CloudML is directly associated with R_6 (*multicloud*). `Credential` is in this case in the form of an `Access Key ID` (random GUID), but with other providers it might be in another form, *e.g.*, a username for Rackspace. Although the form is different, the physical object type (String) is the same.

6.1.1 Single node topology

Initially she is using the topology shown in FIG. 3.1(a). This topology introduces a single node, which hosts every tier of the application. This is not an uncommon topology for development purposes.

Topology considerations. Alice establishes a *single-node* based topology, as seen in FIG. 3.1(a). Since this single node handles both computation and storage, Alice decides to increase capabilities of both processing (number of `Cores`) and `Disk` size on the `Node`. Both of these attributes are incremented because the instance hosts the main application as well as the database.

The approach of using one single node is good in terms of simplicity, since all important components of the application are located in one single place. Other advantages distinguish themselves as well, such as network connections where the address of other components are determined to be "this computer" (*localhost*).

Building templates. In the end Alice inserts all data about topologies into a `Template`. The template include physical descriptions of the `Node`, and a list of the type `Property` for the node. The `Node` has a `name` used to reference the node under provisioning. The properties the node can have are configurations of attributes on a set of given capabilities. These configurations are what define what type of tasks a node is suitable for. In Alice's case the node has increased two important attributes to support both higher computation demand and storage capabilities, *i.e.*, 2 cores and 500 *Gigabyte* (GB) [1] in hard drive size. By not altering any other attributes on the respective nodes, they will be set to minimal values. This is a positive expectation, since the nodes will handle specific tasks, which does not demand enhancing of other attributes.

Provisioning. With these models Alice initializes provisioning by calling `build` on `CloudMLEngine`, providing `Credential` and `Template`. This starts the asynchronous job of configuring and creating `Instances` based on `Nodes`. In FIG. 6.3(a) an object diagram describe the initial configuration at runtime, after CloudML has interpreted the template files. The instance produced by the template and node is in the form of a single object, as represented by the object diagram in FIG. 6.3(b). `Instance` only refer to template by a *String*, `templateName`. This is semantically correct because the template is a transparent entity in the context of provisioning, and is only used as a reference. `Instance` is also an *internal* element in CloudML, and might not have to be indirectly or directly exposed to end users.

`RuntimeInstance` is specifically designed to complement `Node` with `RuntimeProperties`, as `Properties` from `Node` still contain valid data. When `CloudMLEngine` start provisioning, a `RuntimeInstance` is created immediately, and returned to Alice. These are *Models@run.time* (M@RT) within CloudML, designed to provide asynchronous provisioning according to R_5 (*models@run.time*). They are reflections of `Instance`, and they aggregate

[1]Size is expressed in GB, but measurement can change depending on implementation of R_1 (*software reuse*).

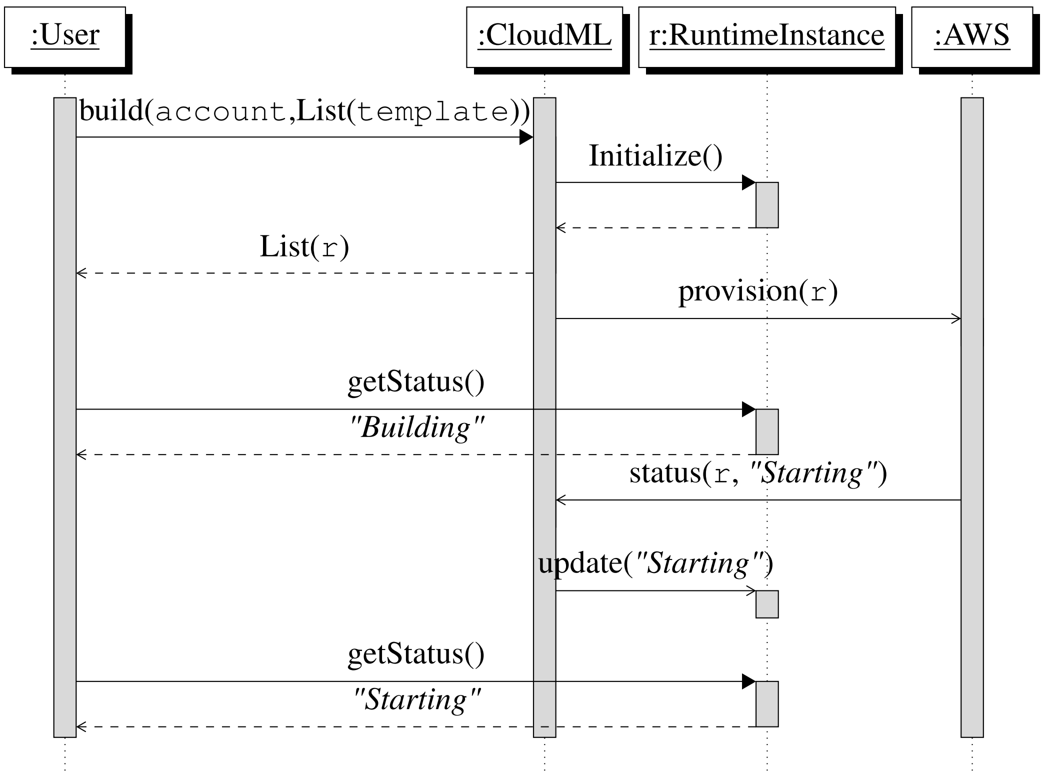


Figure 6.2: CloudML asynchronous provisionning process (Sequence diagram).

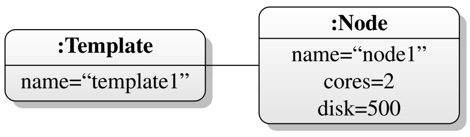
instances.

The method call to `build` is described in FIG. 6.2. In this figure `RuntimeInstance` is returned directly to Alice, because these are asynchronous elements within CloudML, which end users can gather information through. This is emphasized within FIG. 6.2 through `getStatus` method calls. `Instance` is never visualized, this is because it is an internal format within CloudML and does not need to be presented in the sequence diagram,

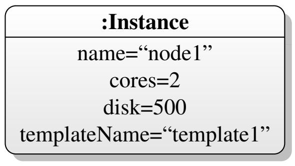
When the `Node` is provisioned successfully and sufficient meta-data is gathered, Alice can start the deployment. CloudML has then completed its scoped task of provisioning.

6.1.2 Three nodes topology

For scalability and modularity the *single-node* approach is restraining, *i.e.*, it does not scale very well, and does not benefit from cloud advantages. If the application consumes too much CPU power, this slows the application totality down and decreases usability. There is no strong link between CloudML and the application, but to maintain scalability some measures must be manually



(a) Template with nodes.



(b) Instance.

Figure 6.3: Object diagram of scenario with one node.

developed into *BankManager*. So the initial application code includes support for work load distribution through application design and deployment considerations. In *BankManager* these measures consists of manually setting physical database address before deploying the application.

In the described scenario, Alice provision *BankManager* to one single instance on AWS. In many cases this setup is sufficient, but major advantages could be gained through the opportunity of horizontal scalability (*scale out*). There are distinct benefits to this. If Alice deployed an application that should suddenly, rapidly and unexpectedly gain popularity, her current setup (one single node) might not be sufficient. In case of such event Alice should change her topology from her initial one seen in FIG. 3.1(a) with one node, to that of FIG. 3.1(c) with three nodes. Or even the topology seen in FIG. 3.1(d), with "unlimited" amount of nodes. This topology is more advanced and utilizes the cloud on a higher level. It has three nodes, two for the application logic (front-end) and one for the database (back-end). In front it has a *load balancer*, which is a cloud service ensuring that requests are spread between front-end nodes based on predefined rules. Even though this service is meant for balancing requests to front-ends, it can actually be used internally in between local nodes as well.

New template. Alice changes her topology by editing her existing `Template` to contain three nodes instead of one. She also changes the node attributes to suite their new needs better, *i.e.*, increasing amount of `Cores` on front-end, and increased `Disk` for back-end `Node`. The characteristics Alice choose for her `Nodes` and `Properties` are fitted for the chosen topology. All `Properties` are optional and thus Alice does not have to define them all.

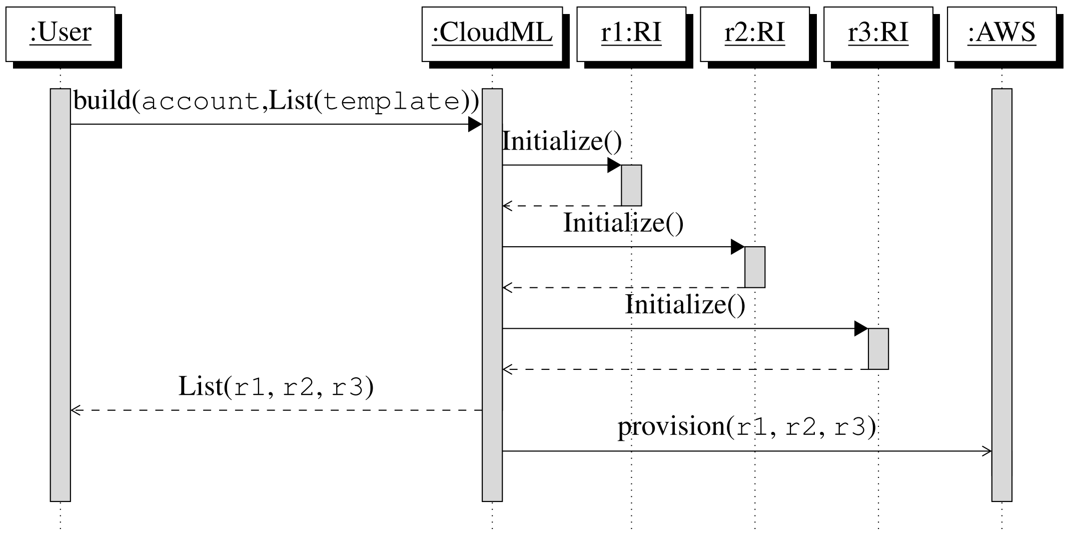


Figure 6.4: Three nodes provisioning: Provisioning.

Rebuild. Then she executes `build` on `CloudMLEngine` again, which will provision the new nodes for her. She will get three new nodes, and the previous provisioned nodes must be manually terminated. More about this in CHAP. 11. The outline of how nodes are provisioned is shown in FIG. 6.4. In the figure `RI` is an abbreviation of `RuntimeInstance`, to save space. It is also split into three parts expressing different steps, *(i)*provisioning, *(ii)*node communications and *(iii)*how a load balancer service is established. This figure is similar to FIG. 6.2, but with three nodes instead of one. Both percept *(i)* and *(ii)* are found in FIG. 6.2. Another difference is that `User` does more calls to `CloudML`, which express how statuses between `RIs` are updating.

When connecting front-end instances of *BankManager* to back-end instances Alice must be aware of the back-ends `PrivateIP` address, which she will retrieve from CloudML during provisioning according to M@RT approach. This was not necessary for the initial scenario setup, but could still be applied as good practice.

Three nodes summary. The benefits of a topology where the application is distributed over several nodes is the scalability and modularity, which were lacking in the *single-node* topology. For instance, if the user demand should rapidly increase, Alice could change her topology to provision more front-end nodes as seen in FIG. 3.1(d). This could be done presumably without greater changes to origin application, since the application is initially designed for such a

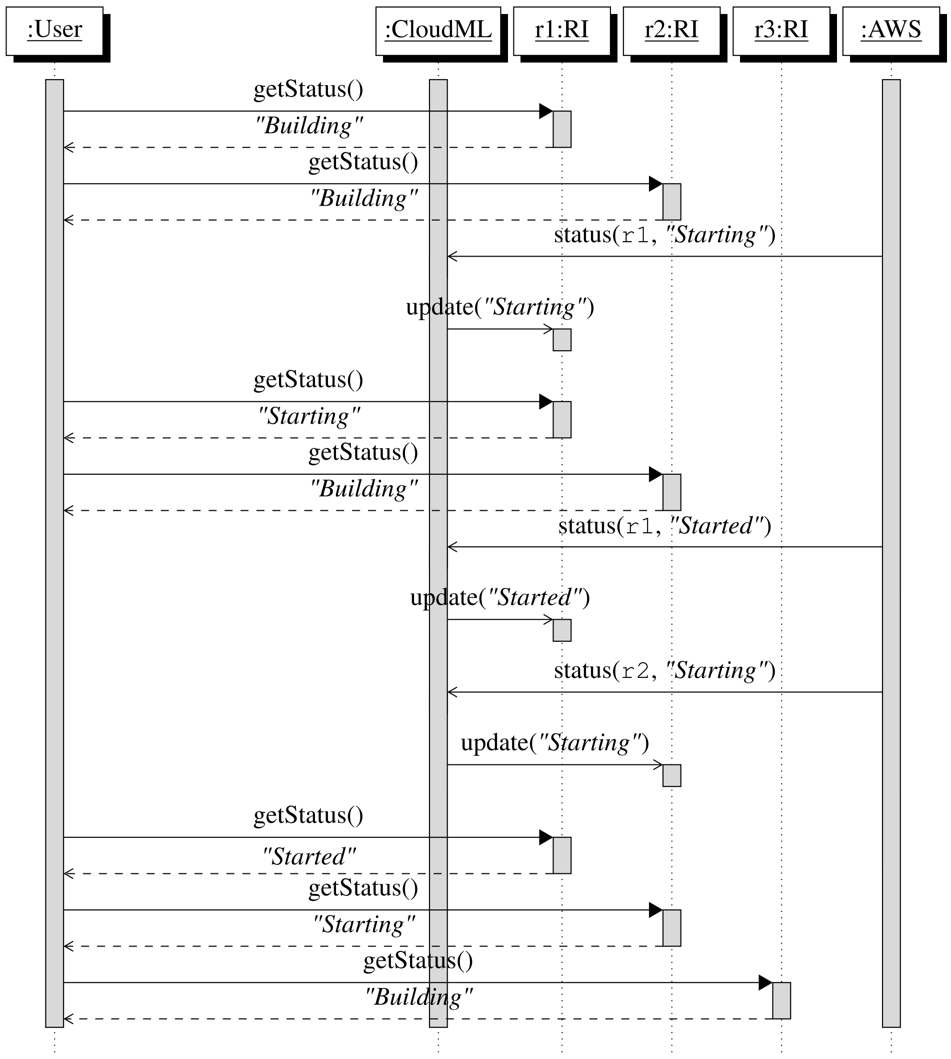


Figure 6.4: Three nodes provisioning: Asynchronous message communication.

distributed topology.

An object diagram of the topology is shown in FIG. 6.5(a). There is nothing that concretely separate front-end nodes from back-end nodes, this can only be determined from node `name` or what attributes are altered. The separation is completely up to Alice when doing the deployment, *i.e.*, nothing in CloudML will restrain or limit Alice when it comes to work load distribution between nodes.

As with the instances in FIG. 6.3(b) the instances in FIG. 6.5(b) are reflections of the nodes (and template) in FIG. 6.5(a). The template name is referenced within each `Instance` for the reasons mentioned earlier.

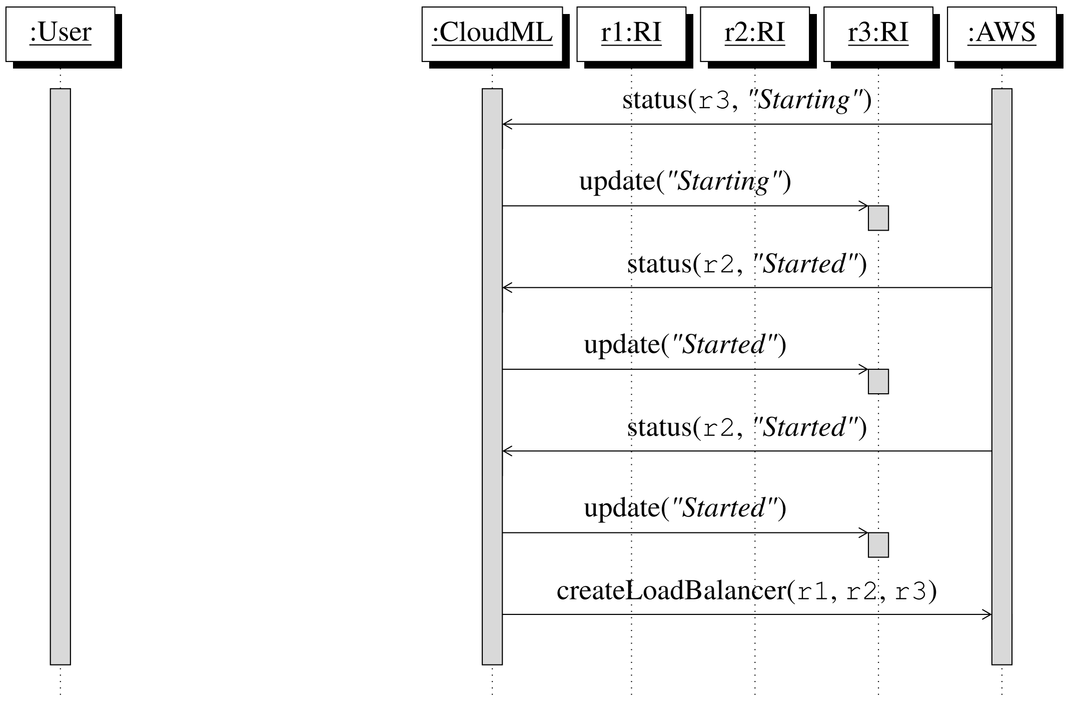


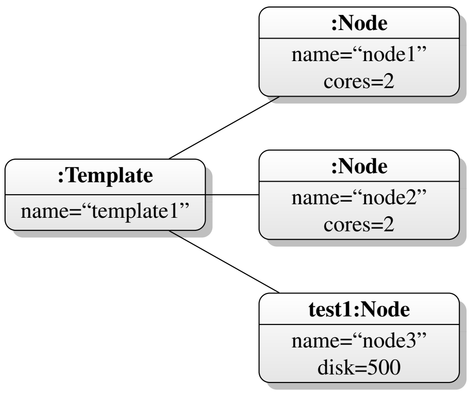
Figure 6.4: Three nodes provisioning: Load balancer.

6.1.3 Multicloud provisioning

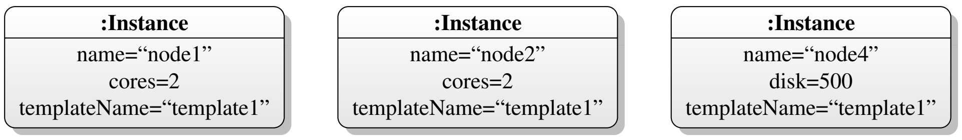
Alice could later decide to use another provider, either as replacement or complement to her current setup, because of availability, financial benefits or support. To do this she must change the provider name in `Account` and call `build` on `CloudMLEngine` again, this will result in an identical topological setup on a supported provider. `UserLibrary` in FIG. 6.1 visualizes that `Account` and `Template` are physical parts maintainable by the user.

The `build` method support provisioning of several templates to one account (same provider). There is also a constraint/limitation, a set of templates can not be simultaneously cross-deployed to different providers, *i.e.*, not possible to define cross-provider (*multicloud*) nodes in a single topology. This is for the sake of tidiness, clarity and technical limitations. CloudML support multicloud provisioning, just that such functionality is achieved by sequential re-provisioning, which will not retain a full multicloud deployment.

AWS and Rackspace combined. To describe the layout of a multicloud provisioning with CloudML, a scenario is invented and a complementary figure is crafted. In the new scenario Alice creates a topology spanning over two



(a) Template with nodes.



(b) Instance.

Figure 6.5: Scenario reimplemented with three nodes.

providers, as seen in FIG. 6.6. In the figure the user connects to a `load balancer`, then the request is distributed between a set of `front-ends` which will retrieve appropriate data from a `back-end`. This is similar to FIG. 3.1(c) with three nodes. Everything so far is under the realm of AWS, including a `load balancer`, *n* amount of nodes for `front-end` computation and one `back-end` node. In this case the `back-end` does not hold any data, instead it is connected to another node *cross provider* to Rackspace. On Rackspace a set of nodes are built to hold data, one `back-end master` manages a set of *n* `slaves`. The `slave` nodes hold all data.

6.2 Technological assessments and considerations

6.2.1 Programming language and application environment.

When considering *programming language* and *application environment* the important aspects are the ones mentioned in R_2 (*strong technological foundation*), *(i)*ease of use, *(ii)*community size, *(iii)*closed/open source, *(iv)*business viability, *(v)*modernity and *(vi)*matureness. For the implementation to be a successful approach towards CloudML the aspects chosen must also be relevant for future improvements. At the same time the aspects chosen must be appealing to existing

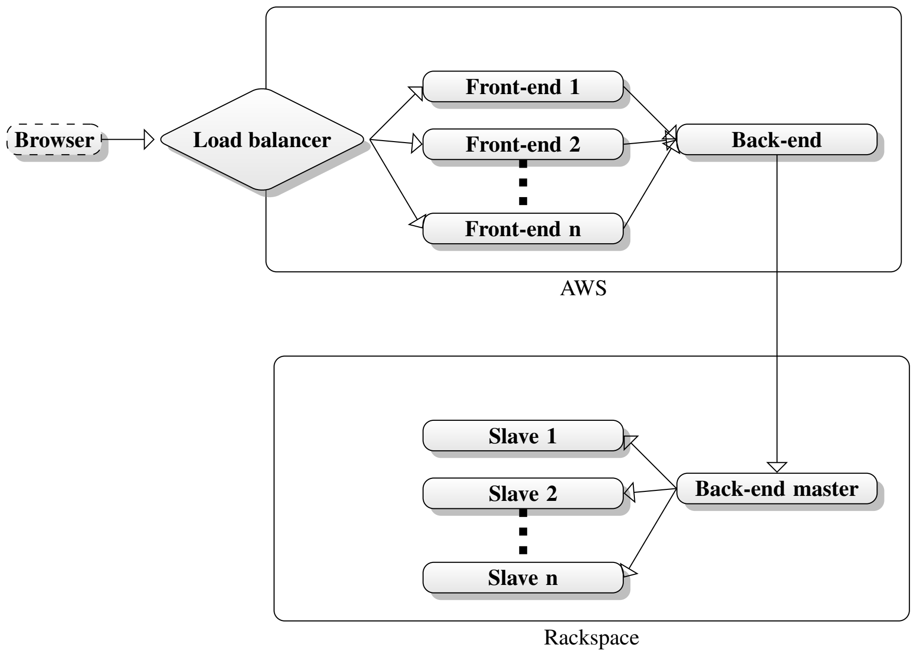


Figure 6.6: Multicloud topology.

communities of interest, without this it will not gain the footing it needs. The most important aspects are therefore precept *(ii)*, *(v)* and *(vi)*.

6.2.2 Asynchronous information gathering and distribution.

The design. When a node is being propagated it changes type from `Instance` to `RuntimeInstance`, which can have a different *state* such as *Configuring*, *Building*, *Starting* and *Started*. Considerations for implementation are described in SEC. 6.3.

When a `RuntimeInstance` reaches *Starting* state the provider has guaranteed its existence, including the most necessary metadata. When all nodes reaches this state the task of provisioning is concluded.

Patterns. Provisioning nodes is by its nature an asynchronous action that can take minutes to execute. For CloudML to compensate with this asynchronous behavior several approaches could be made, as mentioned in CHAP. 4, *i.e.*, observer pattern, command pattern, actor model, or publish-subscribe pattern. Patterns tightly bound to using an object-oriented language for R_2 (*strong technological foundation*). The core idea of integration and benefits of these approaches are:

Observer pattern. The essence of this pattern is a one-to-many distribution of events between objects. This can be explained with a restaurant as an example, with one chef and two waiters. The chef is an *observable* and the two waiters are *observers*. The chef does some work (cooking), and when he or she is complete with a dish all observers are notified. The observers receive this event and can act upon it, *e.g.*, the first waiter ignores the dish, because it does not belong to any of his or her tables, while the other waiter takes the dish to appropriate table.

Command pattern. Is about encapsulating data and logic. Implementations have a known method, *e.g.*, *execute* which comes from an *interface*. So logic could be transparent when invoking them. In the restaurant-example one can consider a general *waiter* to be the interface, and the two waiters would each be separate implementations, *e.g.*, *TableAToC* and *TableDToH*. When the chef *invokes* the method *execute* on any type of *waiter*, he or she does not need to know anything about their implementation, each waiter will know by themselves which table to visit.

Actor model. Solves the problem with asynchronous communication through passing messages in between *actors*. These messages are decoupled from the sender, resulting in proper asynchronous communication. Actors can also create new actors. With the restaurant-example one can imagine all three elements, *i.e.*, chef, first and second waiter, to be individual actors. The chef must know about the first waiter, but the waiters do not necessarily need to know about each other. When a meal is done the chef can inform the correct waiter (one associated with correct table). The benefit here is that the message is decoupled and asynchronous, so the chef can start working on the next meal without waiting for the waiter to return.

Publish-subscribe pattern. This pattern is very similar to observer pattern in behavior, but is more focused on message distribution logic. Compared to observer pattern the *observable* can be considered a *publisher* and *observers* can be called *subscribers*. When a *publisher* sends a message it does not send it directly to any receivers, instead there is a filter-based logic which selects receivers based on condition they set themselves. In the restaurant example, similar to that with observer pattern, the chef is *publisher* and both waiters are *subscribers*. When the chef completes a meal he or she notify the waiters based on their *conditions*, *e.g.*, the first waiter listen for events of type *TableA*, *TableB* and *TableC*. If the finished meal was assigned for table A the first waiter would get the message.

Of these solutions the most promising one is *actor model* because it is more directly aimed at solving asynchronous communication. The other solutions can provide assistance by accommodating communications through events, but by them selves they do not solve the issue of slow provisioning. An interesting approach is to combine the *actor model* with one of the patterns. By doing this the behavior functionality is combined with the asynchronous benefits within the actor model. Of the previous mentioned patterns only *observer* and *publish-subscribe* patterns fit well with designed behavior of CloudML. Of these *publish-subscribe* is the most scalable and dynamic one (in runtime), but also the most complicated. When combining the *actor model* with such pattern it is important to keep the implementation as simple as possible, without withholding functionality. Hence the best solution for CloudML is to combine *actor model* with *observer pattern*.

The *actor model* can be implemented in many different ways, even on different tiers. Both requirement R_2 (*strong technological foundation*) and R_1 (*software reuse*) can assist with either a built-in functionality in the language, environment or an external library or framework. The observer pattern can be implemented in many ways, *e.g.*, using `java.util.Observer` and `java.util.Observable`.

6.2.3 Lexical format.

Requirement R_4 (*lexical template*) from CHAP. 4 consists of two important core elements. The first is *important aspect*, *i.e.*, functional demands the format must fulfill, such as how popular it is and how well it is supported by programming languages. Second element is that of *data format*, *i.e.*, the actual language used to serialize data. This last element is not important for design, and will be introduced and further described in SEC. 7.1 (CHAP. 7). The important aspects of a lexical format for CloudML are:

- Renowned among software developer communities, industry and the academic domain. The language will be used directly by these parties, so it is beneficial that they are familiar with the syntax.

- Integrates (supporting libraries) with most common existing technologies. The range of R_2 (*strong technological foundation*) should not be limited by the format.

- Human-readable, to some extent. End users needs to create and edit these files, often by hand. Therefore it must be readable and manually maintainable.

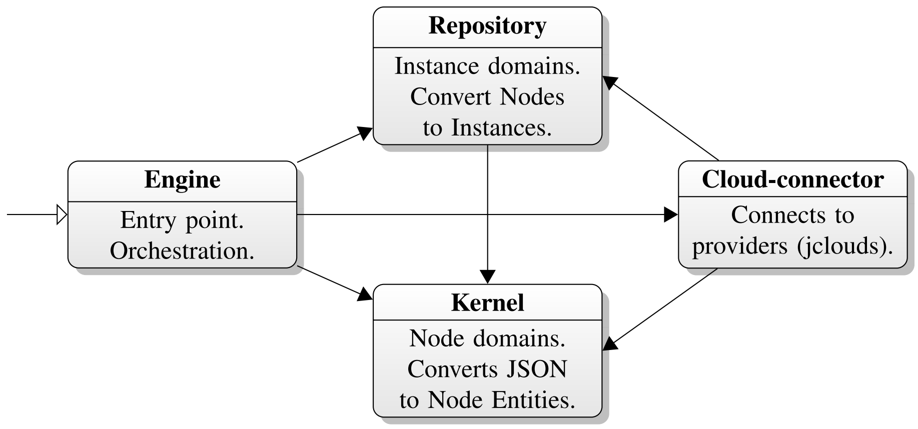
- Function with web-services and web-based technologies. CloudML is a language designed for *cloud environments*, so choosing a format that works well on the web is crucial.

6.3 Modules and application flow

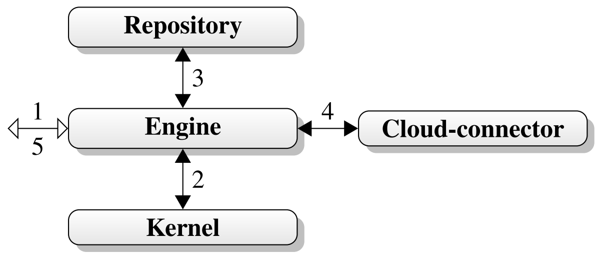
CloudML is divided into four main modules (FIG. 6.7). This is to distribute workload and divide CloudML into logical parts for different tasks.

Engine. The main entry point to the application is `Engine`. *i.e.*, it does not have to be initialized and there exists only one instance of it. Interaction between `user` and `Engine` is expressed in FIG. 6.8 where the user will initialize provisioning by calling `Engine`. `Engine` will also do orchestration between the three other modules as shown in FIG. 6.7(b). Since `Cloud-connector` is managed by `Engine`, other actions run against instances are done through `Engine`. The first versions of *cloudml-modules* did not use `Engine` as orchestrator but instead relied on each module to be a sequential step, this proved to be harder to maintain and also introduced cyclic dependencies. `Engine` has dependencies to all the other modules, as expressed in FIG. 6.7(a).

Kernel. CloudML specific entities are kept in `Kernel`, such as `Node` and `Template`. This is some of the core parts of the *Domain-specific language* (DSL), hence it is called *`Kernel`*. The logical task of `Kernel` is to map JSON formatted strings to `Templates` including `Nodes`. `Accounts` are separate parts that are parsed equally as `Templates`, but by another method call. All this is transparent for users as all data will be provided directly to `Engine` which will handle the task of calling `Kernel` correctly. In FIG. 6.7(a) is it shown that `Kernel` has no dependency to other modules, while all the other modules depend on `Kernel`. This stresses the fact that `Kernel` is the most fundamental part, closest to the DSL. According to FIG. 6.7(b) `Kernel` is the first module to be



(a) Application module dependencies.



(b) Application module orchestration/flow. The numbers indicate order of sequential flow.

Module
Application modules
Dependency
Entry

(c) Legend.

Figure 6.7: Modules in CloudML, dependencies and orchestration.

called by `Engine`, as it will convert incoming text (*String*) to internal format of `Nodes`.

Repository. `Instance` are entities within `Repository`, these are equivalent to `Nodes` in `Kernel`, but are specific for provisioning. Repository will do a mapping from *Nodes* (including *Template*) to *Instances*. Future versions of `Repository` will also do some logical superficial validation against *Node* properties. At the writing moment (April 2012) it is not possible to demand load balancers (as a service) on Rackspace for specific geographical locations.

Cloud-connector. The connections to providers are established through the module `Cloud-connector`. This module uses an external software allowing CloudML to connect to several providers through a common interface. This

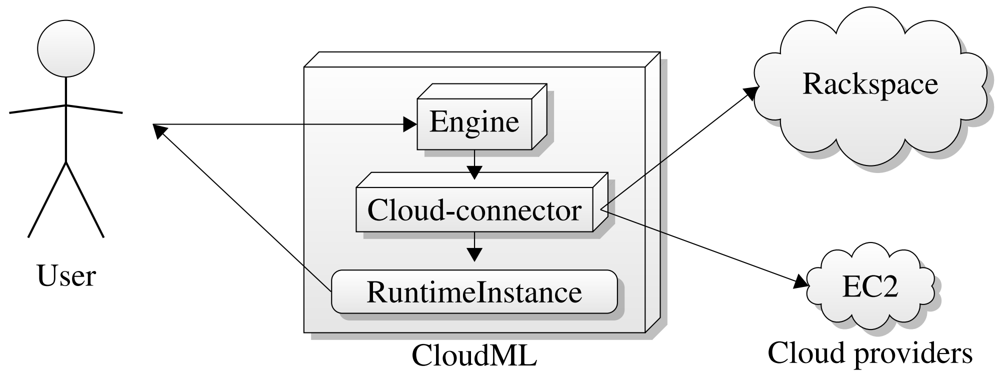


Figure 6.8: Usage flow in CloudML.

software fulfills R_1 (*software reuse*). The connections to providers are expressed in FIG. 6.8. As expressed in FIG. 6.7(a) `Cloud-connector` depend on both `Kernel` and `Repository`. These dependencies match its task, as it must have knowledge about both `Templates` and `RuntimeInstance`. This module is the last step before returning to the user, as seen in FIG. 6.7(b). In FIG. 6.8 it is expressed that `Cloud-connector` initializes the actors (`RuntimeInstance`) which are provided to the `User`.

Chapter 7

Implementation/realization - cloudml-engine

In the previous chapter the design of CloudML was introduced. In this chapter the implementation of this design will be presented and described. First section is a technical overview of some of the general solutions within the implementation. The rest of the chapter is split into different sections, one for each important part of the implementation. Topics are about *(i)*submodules and their tasks, *(ii)*the depdency system, *(iii)*communication with cloud providers, *(iv)*M@RT through actor model and *(v)*data serialization.

7.1 Technology overview

CloudML is implemented as a *proof-of-concept* framework [7] (from here known as *cloudml-engine*). The implementation, *cloudml-engine*, is based on *state-of-the-art* technologies that appeal to the academic community. Technologies chosen for *cloudml-engine* are not of great importance to the concept of CloudML itself, but it is still important to understand which technologies are chosen, what close alternatives exists and why they are chosen.

Language and environment. The importance of a *programming language* and *environment* were described in SEC. 6.2. These are core ideas to fulfill R_2 (*strong technological foundation*), emphasizing the importance of a strong foundation. Here is a list describing some of the languages and environments considered:

- Java (JVM)

- JavaScript (Node.js)
- Scala (JVM)
- Python
- C# (.NET)

Languages in this list are introduced because of their overall popularity. Some were introduced despite this, such as *Node.js*, which is brought in as an consideration because of the abilities to operate with JSON, cloud interaction and modernity.

Based on these considerations Scala is a good choice, it fulfills the core ideas and match the requirement R_2 (*strong technological foundation*):

Ease of use. Scala is a *state-of-the-art* language with many technological advantages, *e.g.*, supporting both functional and object-oriented programming. With both paradigms, imperative programmers such as Java developers as well as functional programmers such as Clojure developers would be familiar with the language design.

Emphasizes on functional programming which is leveraged in the implementation. Built in system for actor model [14], which is utilized in the implementation. This also assists in relation to R_5 (*models@run.time*).

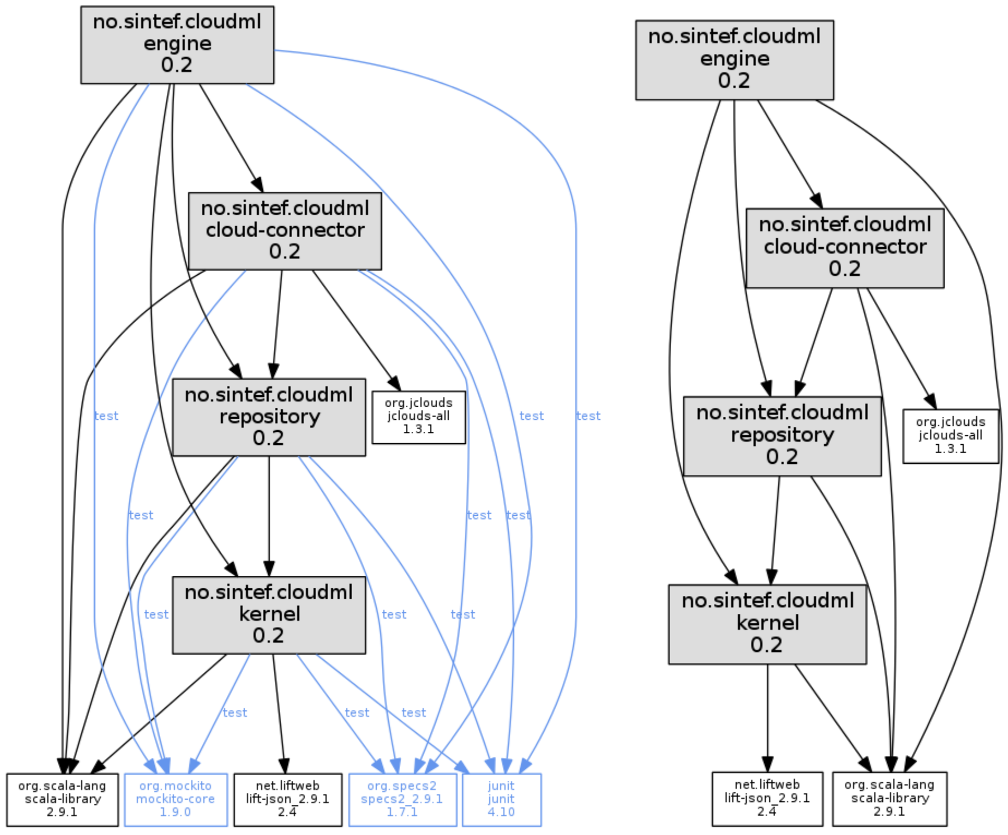
JVM is a popular platform, as the main platform for Java. Scala runs on the JVM, and support all the libraries and work put into Java over the years can be utilized directly by Scala.

Community size. This language is gaining popularity among both the academic and enterprise domain.

Closed/open source. Scala is open source.

Modernity. The reason not to use plain Java is because Scala is an appealing *state-of-the-art* language. The language is under constant development and new features and enhancements keeps improving the language.

Matureness. Compatible with Java and libraries written in Scala can be interacted with by Java as well. This means that even though Scala is chosen for the implementation of CloudML, it can still be used by other languages supported by the JVM, *e.g.*, Java, Groovy and Clojure.



no.sintef.cloudml engine 0.2
no.sintef.cloudml cloud-connector 0.2
no.sintef.cloudml repository 0.2
org.jclouds jclouds-all 1.3.1
no.sintef.cloudml kernel 0.2
org.scala-lang scala-library 2.9.1
org.mockito mockito-core 1.9.0
net.liftweb lift-json_2.9.1 2.4
org.specs2 specs2_2.9.1 1.7.1
junit junit 4.10
test
no.sintef.cloudml engine 0.2
no.sintef.cloudml cloud-connector 0.2
no.sintef.cloudml repository 0.2
org.jclouds jclouds-all 1.3.1
no.sintef.cloudml kernel 0.2
net.liftweb lift-json_2.9.1 2.4
org.scala-lang scala-library 2.9.1

(a) dependency-graph-without-test (b) dependency-graph-with-test

Figure 7.1: Maven dependency graph (with and without test scope).

Based on these benefits Scala is chosen, and *cloudml-engine* is written in this language.

7.2 Automatic build system

There are two main methods used to automatically build Scala programs, either using a Scala-specific tool called *Simple Build Tool* (SBT) or a more general tool called Maven. For *cloudml-engine* to have an academic appeal it is essential to choose the technology with most closeness to Java, hence Maven is the best option. It is also important to make the library available for developers using Java or other languages supported by the JVM. This is best achieved by using Maven, as SBT is directed at Scala.

In SEC. 6.3 CloudML were split into four different modules, as seen in FIG. 6.7. Maven support modules, these are used to split *cloudml-engine* into the appropriate modules.

A full Maven dependency graph is expressed in FIG. 7.1. There are two graphs, both of *cloudml-engine*. The first graph (FIG. 7.1(a)) excludes the *test-scope*, while FIG. 7.1(b) includes it. The *test-scope* is a Maven scope which marks the dependencies to only be included when running a specific *goal*, *e.g.*, test. In this graph the internal and external dependencies in *cloudml-engine* are expressed. All external dependencies, *e.g.*, `org.jclouds.jclouds-all` (FIG. 7.1), contain even more dependencies to other libraries and internal modules. These dependencies are all omitted in both graphs.

7.3 Cloud connection

The connection between CloudML and cloud providers is facilitated to a separate module, called `cloud-connector`. In the implementation this module is the bridge between providers and *cloudml-engine*. It is built to support several libraries and interface these, this is achieved by implementing a type of *facade pattern*. It does not contain any entities, and only executes logical code.

The bridge between *cloudml-engine* and cloud providers is an important aspect of the application, and as a requirement it is important to use an existing library to achieve this connection. Some libraries have already been mentioned in the *APIs* section (SEC. 2.2), of these only *jclouds* is based on Java-technologies and therefore suites *cloudml-engine*. This library gives CloudML support for 24 providers out of the box to minimize *complexity*, as well as stability and robustness. These advantages directly resolves resolves R_1 (*software reuse*). Jclouds uses Maven for building as well, and is part of Maven central which makes it possible to add jclouds directly as a module dependency. Jclouds contains a template system which is used through code directly, this is utilized to map CloudML templates to jclouds templates.

7.4 Asynchronous provisioning

Provisioning can consume up to minutes for each instance, it is therefore essential to make use of asynchronous behavior. The solution, *coudml-engine*, combines two different approaches to achieve this, actor model and observer pattern.

Actor model. The asynchronous solution in CloudML is based on actor model [14], resulting in concurrent communication with nodes under provision-

Requirement	XML	JSON	YAML
Community	2	2	1
Technology support	2	2	1
Human-readable	1	2	3
Web-service friendly	2	2	0

Table 7.1: Comparing lexical formats (SEC. 6.2) with aspects from requirement R_4 (*lexical template*). Weighting from zero (0) to three (3) respectively least to most supported.

ing. By adopting this behavior developers exploring the implementation can then choose to "listen" for updating events from each node, and do other jobs / idle while the nodes are provisioned.

The actor model in CloudML is based on built-in implementations in Scala, `scala.actor.Actor`. This approach solves R_2 (*strong technological foundation*), as the underlying technology provides the asynchronous solution. The actor model could be implemented using external libraries instead, *e.g.*, Akka, in this case the approach would solve R_1 (*software reuse*).

Observer pattern. Beside the standard model provided by Scala *cloudml-engine* uses a callback-based pattern to inform users of the library when instance statues are updated and properties are added.

Models@run.time. The terms are divided for a node before and under provisioning, the essential is to introduce *M@RT* to achieve a logical separation.

7.5 From text to objects

As described in CHAP. 4 there exists numerous implementations of different lexical formats. Three of these formats are chosen as the most important ones, *(i)*XML, *(ii)*JSON and *(iii)*YAML. The most important points about these formats are described in SEC. 6.2, *i.e.*, *(i)*community, *(ii)*technology support, *(iii)*human-readable and *(iv)*web-service friendly. The different points are expressed in TABLE. 7.1, compared against the three different formats. In the table they are weighted from zero (0) to three (3), where zero is least supported and three is most supported, *i.e.*, how well the formats cover the aspects described in SEC. 6.2.

For the lexical representation of CloudML, JSON is the best alternative. This format is used because of the values seen in TABLE. 7.1.

Here is an extracted list of the most relevant points compared against the JSON format:

Community. This format is growing in popularity as more adapt to using it for different purposes, *e.g.*, databases, web-service communications, configuration files or *Internationalization and localization* (i18n).

Technology support. There exists parsing libraries for most languages, for each of the languages listed in SEC. 7.1.

Human-readable. The format itself does not have any duplications, and is therefore easier read by humans than XML.

Web-service friendly. Used for both communicating between browsers and web servers, and interchange between nodes in a SOA environment.

The JSON format is parsed in Scala using the *lift-json* parser which provides implicit mapping to Scala case-classes. This library is part of the lift framework, but can be included as an external component without additional lift-specific dependencies. GSON was considered as an alternative, but mapping to Scala case-classes was not as fluent compared to lift-json.

7.6 Usage

This section gives information on general usage of *cloudml-engine*. Information such as how:

- To include it when developing applications.
- It is distributed on the Internet.
- To do method calls to initialize provisioning.

Cloudml-engine can be included into a vast variety of applications, ranging from native desktop applications to web-service based applications, or even smartphone apps.

```
<repositories>
 <repository>
  <id>cloudml-engine</id>
  <url>
   https://repository-eirikb.forge.cloudbees.com/release
  </url>
 </repository>
</repositories>
<dependencies>
 <dependency>
  <groupId>no.sintef</groupId>
  <artifactId>engine</artifactId>
  <version>0.2</version>
 </dependency>
</dependencies>
```

Figure 7.2: Example Maven conficuration section to include cloudml-engine.

```
import no.sintef.cloudml.engine.Engine
...
val runtimeInstances = Engine(account, List(template))
```

Figure 7.3: Example client (Scala) callout to cloudml-engine.

Inclusion through dependencies. *Cloudml-engine* is a Maven module, and therefore it can be included in applications by adding it as a Maven dependency. Such dependency reference is expressed as a Maven configuration in LISTING. 7.2.

Distribution. *Cloudml-engine* is not just a proof-of-concept for the sake of conceptual assurance, but it is also a running, functional library which can be used by anyone for testing or considerations. Beside the source repository[7] the library is deployed to a remote repository [11] as a Maven module. This repository is provided by CloudBees, how to include the library is viewable in LISTING. 7.2. In this figure *cloudml-engine* is reachable by Maven by adding a `repository` node to the configuration. This is a necessity because the library is yet (April 2012) to be deployed to Maven central.

Engine callout. After successfully included *cloudml-engine* as a dependency the library is accessible. A Scala callout to `Engine` is expressed in FIG. 7.3.

Chapter 8

Validation & Experiments

To validate how CloudML addresses the requirements from CHAP. 4, a topology of three nodes (FIG. 3.1(c)) is provisioned. This topology is the same as Alice used for her second scenario in SEC. 6.1. The setup is sufficient to do a full deployment of the *BankManager* application.

The validation will provision on two different providers, AWS and Rackspace.

Template. The implementation uses JSON to define templates as a human readable serialization mechanism. The lexical representation of FIG. 3.1(c) can be seen in LISTING. 8.1. There are a total of three files:

account.json. Expressed in LISTING. **??**, used to authenticate against a provider. In LISTING. 8.1 `aws-ec2` is set as `provider`, *i.e.*, nodes are created on AWS. The two other properties, `identity` and `credential` are used for authentication. For AWS that means *Access Key ID* and *Secret Access Key*, while for Rackspace this is *username* and *API Key*.

front-ends.json. Defines front-end nodes of the topology. Each node have specific attributes regarding their tasks, similar to Alice's scenario, but as an addiontal precaution the `front-end` nodes have increased RAM.

back-end.json. Defines the back-end node of the topology. Even though this is a separate file it is a part of the same topology, and it is provisioned beside the front-end nodes.

The topology is split into two templates to support a load balancer, as every node within a template will be bound to a given load balancer. This feature is implemented in *cloudml-engine*, but at writing moment (April 2012), is not front-ends.json:

```

"name": "front-ends",
"nodes": [{
  "name": "front-end1",
  "minCores": 2,
  "minRam": 1000
}, {
  "name": "front-end2",
  "minCores": 2,
  "minRam": 1000
}]

```

back-end.json:

```

"name": "back-end",
"nodes": [{
  "name": "back-end",
  "minDisk": 500
}]

```

Figure 8.1: Template for validation.

supported by jclouds. More about the load balancer in CHAP. 11. The splitting is by design, as a `template` is not directly bound to a topology, and is also why `build` accept a list of templates. The whole text represents the `Template` and consequently “`nodes`” is a list of `Node` from the model. The JSON is textual which makes it *shareable* as files. Once such a file is created it can be reused (*reproducibility*) on any supported provider (*multicloud*). These benefits match the requirement R_4 (*lexical template*).

Client. Validation is done through a *Command-line interface* (CLI)-based client application written in Scala. Code snippets of this client is expressed in LISTING. 8.3. The snippets does not contain code to read the files, this is left out to save space. File reading is done through Scala with account.json:

```

"provider": "aws-ec2",
"identity": "...",
"credential": "..."

```

Figure 8.2: Account JSON used for validation.

`io.Source.fromFile(`*`filename`*`).mkString`. What files to read is specified through command line arguments, where the first argument corresponds to account information, and subsequent arguments are template files.

To initialize provisioning the client calls `val runtimeInstances = Engine(account, templates)`. According to the meta model (FIG. 6.1) this is `build`. The name "build" is a general term, while the usage in LISTING. 8.3 is a Scala-specific pattern.

After receiving a list of `RuntimeInstances` the client runs through these objects and attach "listeners". These listeners works according to the observer pattern, when a property is added or the instance changes status, a callback is made to the client. This callback is done asynchronous through actor model. Both of these solutions solves the requirement R_5 (*models@run.time*).

Provisioning. The client is built using Maven, to execute it the `scala:run`-*goal* is called. The three files, *(i)*account.json, *(ii)*front-ends.json and *(iii)*back-end.json, are passed in as arguments to the client. The final command looks like this: `mvn scala:run -DaddArgs="account.json|front-ends.json|back-end.json"`. Output from executing this command is expressed in LISTING. 8.4. Some of the output is chunked away to save space. The text in LISTING. 8.4 correlates to the client expressed in LISTING. 8.3, *i.e.*, information about status changes and when each node is successfully provisioned.

The output is identical for both AWS and Rackspace.

After provisioning. To validate the experiment screenshots of the cloud console of each provider are made.

The screenshot of AWS console (FIG. 8.5) shows that three nodes are

```scala
import no.sintef.cloudml.engine.Engine
import no.sintef.cloudml.repository.domain._
 ...
val runtimeInstances = Engine(account, templates)
println("Got " + runtimeInstances.size + " nodes")
runtimeInstances.foreach(ri => {
  println("Adding listener to: " + ri.instance.name +
    " (" + ri.status + ")")
  ri.addListener( (event) =>  {
    event match {
      case Event.Property =>
      case Event.Status =>
        println("Status changed for " + ri.instance.name +
          ": " + ri.stat
        if (ri.status ==  Status.Started) {
          println("Node " + ri.instance.name +
            " is now running: " + ri)
        }
    }
  }
})
```

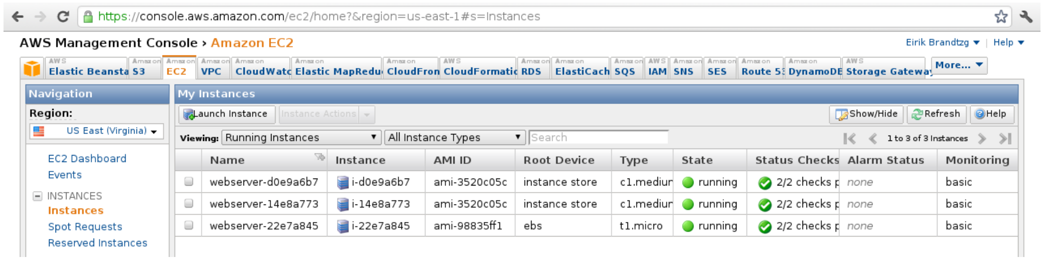
Figure 8.3: Code snipptes of client used for validation (Scala).

successfully provisioned. Two of the nodes is of the type `c1.medium` which corresponds to the template configuration. The last node is a `t1.micro`, *i.e.*, this is the `back-end` node. This is further emphasized by the fact that it uses EBS as `Root device`, to achieve additional disk space.

Screenshot of Rackspace indicate similar consistencies, *e.g.*, `RAM Amount` for each node, which corresponds with the template.

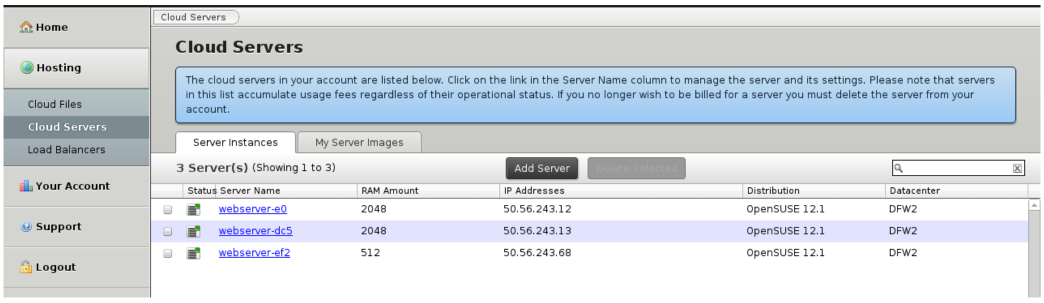
```
mvn scala:run
  -DaddArgs="account.json|front-ends.json|back-end.json"

[INFO] Scanning for projects...
[INFO]
   ...
Got 3 nodes
Adding listener to: front-end1 (Building)
Adding listener to: front-end2 (Configuring)
Adding listener to: back-end (Configuring)
Status changed for front-end1: Starting
Status changed for front-end1: Started
Node front-end1 is now running:
  RuntimeInstance(Instance(front-end1,2,2,0,))
Status changed for front-end2: Building
Status changed for front-end2: Starting
Status changed for front-end2: Started
Node front-end2 is now running:
  RuntimeInstance(Instance(front-end2,2,2,0,))
Status changed for back-end: Building
Status changed for back-end: Starting
Status changed for back-end: Started
Node back-end is now running:
  RuntimeInstance(Instance(back-end,0,1,500,))
```

Figure 8.4: Output from running validation client.


https://console.aws.amazon.com/ec2/home?&region=us-east-1#s=Instances
AWS Management Console › Amazon EC2
Eirik Brandtzg
Help
Elastic Beansta S3 EC2 VPC CloudWatc Elastic MapRedu CloudFron CloudFormati RDS ElastiCach SQS IAM SNS SES Route 5 DynamoDE Storage Gatewa More...
Navigation
Region:
US East (Virginia)
EC2 Dashboard
Events
INSTANCES
Instances
Spot Requests
Reserved Instances
My Instances
Launch Instance
Instance Actions
Show/Hide
Refresh
Help
Viewing: Running Instances
All Instance Types
Search
1 to 3 of 3 Instances

Name	Instance	AMI ID	Root Device	Type	State	Status Checks	Alarm Status	Monitoring
webserver-d0e9a6b7	i-d0e9a6b7	ami-3520c05c	instance store	c1.mediur	● running	✓ 2/2 checks p	none	basic
webserver-14e8a773	i-14e8a773	ami-3520c05c	instance store	c1.mediur	● running	✓ 2/2 checks p	none	basic
webserver-22e7a845	i-22e7a845	ami-98835ff1	ebs	t1.micro	● running	✓ 2/2 checks p	none	basic

Figure 8.5: Screenshot of AWS console after provisioning.



Cloud Servers

The cloud servers in your account are listed below. Click on the link in the Server Name column to manage the server and its settings. Please note that servers in this list accumulate usage fees regardless of their operational status. If you no longer wish to be billed for a server you must delete the server from your account.

3 Server(s) (Showing 1 to 3)

Status	Server Name	RAM Amount	IP Addresses	Distribution	Datacenter
	webserver-e0	2048	50.56.243.12	OpenSUSE 12.1	DFW2
	webserver-dc5	2048	50.56.243.13	OpenSUSE 12.1	DFW2
	webserver-ef2	512	50.56.243.68	OpenSUSE 12.1	DFW2

Figure 8.6: Screenshot of Rackspace console after provisioning.

Part III

Conclusion

Chapter 9

Conclusion

Short and sharp

- Summary of CloudML
  - What subsection in solution solves what subsection in problem
- CloudML
- Implementation
- Perspectives (2 paragraphs, can be section)
  - Look into the future
    - Deployments
  - short term
  - long term

Chapter 10

Results

Comparing challenges with some selected providers and technologies from CHAP. 2.

State of the art	***software-reuse***	***foundation***	***mda***	***m@rt***	***lexical-template***
Amazon CloudFormation			Yes	No	Yes
CA Applogic			Yes		No
jclouds		Yes	Partly	No	No
mOSAIC	Yes	Yes	No	No	No
Amazon Beanstalk	Yes	Yes	No	No	No
CloudML	Yes	Yes	Yes	Yes	Yes

Table 10.1: Comparing selected elements from CHAP. 2 with requirements.

Chapter 11

Perspectives

Full deployment is planed for next version of CloudML.

Live managing.

Load balancer.

```

"name": "test",
"loadBalancer": {
  "name": "test",
  "protocol": "http",
  "loadBalancerPort": 80,
  "instancePort": 80
},
"nodes": []

```

Figure 11.1: Template including load balancer.

Bibliography

[1] Amazon. Amazon web services, 2012. URL http://aws.amazon.com.

[2] Apache. Deltacloud, 2012. URL http://deltacloud.apache.org.

[3] Apache. Libcloud, 2012. URL http://libcloud.apache.org.

[4] Gorka Benguria, Andrey Sadovykh, **Sébastien Mosser**, Antonin Abhervé, and Eirik Bradtzæg. Towards a Domain-Specific Language to Deploy Applications in the Clouds (submitted). Technical Report D-4.1, EU FP7 REMICS, March 2012.

[5] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005. ISBN 0321267974.

[6] Eirik Brandtzæg. Bank manager, 2012. URL https://github.com/eirikb/grails-bank-example.

[7] Eirik Brandtzæg. cloudml-engine, 2012. URL https://github.com/eirikb/cloudml-engine.

[8] Eirik Brandtzæg, **Sébastien Mosser**, and Parastoo Mohagheghi. Towards CloudML, a Model-based Approach to Provision Resources in the Clouds (submitted). In *Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE 2012), co-located with ECMFA'12*, Lyngby, Danemark, July 2012. Springer LNCS.

[9] CA. Applogic, 2012. URL http://www.3tera.com/AppLogic.

[10] Trieu Chieu, A. Karve, A. Mohindra, and A. Segal. Simplifying solution deployment on a Cloud through composite appliances. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE*

International Symposium on, pages 1 –5, april 2010. doi: 10.1109/IPDPSW. 2010.5470721.

[11] CloudBees. Cloudbees cloudml-engine repository, 2012. URL https://repository-eirikb.forge.cloudbees.com/release/.

[12] R.T. Fielding and R.N. Taylor. Principled design of the modern Web architecture. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 407 –416, 2000. doi: 10.1109/ICSE. 2000.870431.

[13] U.S. Government. Recovery, 2012. URL http://www.recovery.gov.

[14] Philipp Haller and Martin Odersky. Actors that unify threads and events. In *Proceedings of the 9th international conference on Coordination models and languages*, COORDINATION'07, pages 171–190, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-72793-4. URL http://dl.acm.org/citation.cfm?id=1764606.1764620.

[15] jclouds. jclouds, 2012. URL http://jclouds.org.

[16] Stuart Kent. Model Driven Engineering. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer Berlin / Heidelberg, 2002. ISBN 978-3-540-43703-1.

[17] Madeira. Madeiracloud, 2012. URL http://www.madeiracloud.com.

[18] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology. *Nist Special Publication*, 145(6):7, 2011. URL http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf.

[19] Sébastien Mosser, Eirik Brandtzæg, and Parastoo Mohagheghi. CloudâĂŞ-Computing: from Revolution to Evolution (published). In *BElgian-NEtherlands software eVOLution seminar(BENEVOL'11), workshop: (extended abstract)*, , pages 1–2, Brussels, Belgium, December 2011. VUB.

[20] Viet Cuong Nguyen and X. Qafmolla. Agile Development of Platform Independent Model in Model Driven Architecture. In *Information and*

Computing (ICIC), 2010 Third International Conference on, volume 2, pages 344–347, june 2010. doi: 10.1109/ICIC.2010.180.

[21] Dana Petcu, Ciprian Crăciun, Marian Neagul, Silviu Panica, Beniamino Di Martino, Salvatore Venticinque, Massimiliano Rak, and Rocco Aversa. Architecturing a Sky Computing Platform. In Michel Cezon and Yaron Wolfsthal, editors, *Towards a Service-Based Internet. ServiceWave 2010 Workshops*, volume 6569 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-22759-2.

[22] Dana Petcu, Georgiana Macariu, Silviu Panica, and Ciprian Crăciun. Portable Cloud applicationsFrom theory to practice. *Future Generation Computer Systems*, 2012. ISSN 0167-739X. doi: 10.1016/j.future.2012.01.009. URL http://www.sciencedirect.com/science/article/pii/S0167739X12000210.

[23] Rackspace. Rackspace cloud, 2012. URL http://www.rackspace.com/cloud.

[24] Tejaswi Redkar and Tejaswi Redkar. Introducing Cloud Services. In *Windows Azure Platform*, pages 1–51. Apress, 2009. ISBN 978-1-4302-2480-8.

[25] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galan. The Reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):4:1 –4:11, july 2009. ISSN 0018-8646. doi: 10.1147/JRD.2009.5429058.

[26] Y. Singh and M. Sood. Model Driven Architecture: A Perspective. In *Advance Computing Conference, 2009. IACC 2009. IEEE International*, pages 1644 –1652, march 2009. doi: 10.1109/IADCC.2009.4809264.

[27] Katarina Stanoevska-Slabeva and Thomas Wozniak. Cloud Basics - An Introduction to Cloud Computing. In Katarina Stanoevska-Slabeva, Thomas Wozniak, and Santi Ristol, editors, *Grid and Cloud Computing*, pages 47–61. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-05193-7. URL http://dx.doi.org/10.1007/978-3-642-05193-7_4.