**UNIVERSITY OF OSLO**
**Department of Informatics**

# CloudML

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

Master thesis

Eirik Brandtzæg

**Spring 2012**

# CloudML

Eirik Brandtzæg

Spring 2012

**Built: 19th March 2012**

# Abstract

# Contents

# List of Figures

# List of Tables

# Preface

# Part I

# Introduction

# Chapter 1

# Introduction

### Short and sharp

- Main introduction
- Write lastly

# Chapter 2

# 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.

## 2.1 Cloud computing

Cloud computing is gaining popularity and more companies are starting to explore the possibilities as well as the limitation to the cloud. The definitions under are mainly based on definitions by the *National Institute of Standards and Technology* (NIST) which is one of the leaders in cloud computing standardization. The main providers of cloud computing at writing moment are Google, Amazon with *Amazon Web Service* (AWS) [1] and Microsoft. A non-exhaustive list of common providers are visualized in TABLE. 2.1.

### 2.1.1 Characteristics

Cloud computing is about providing computation as services [14], such as virtual instances and file storage, rather than products. Cloud characteristics are what

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

Table 2.1: Common providers available services

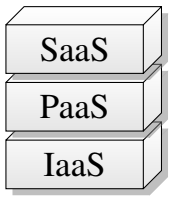



Figure 2.1: Cloud architecture service models

defines the difference between what is normal hosting and what is 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 means 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. With *on-demand self-service* this is a trivial issue as resource allocation will automatically scale upwards as popularity increases and downwards as resources become superfluous.

**Broad network access.** This means that cloud capabilities are available over standard network mechanisms, supporting familiar protocols such as HTTP/HTTPS and *Secure Shell* (SSH). This means that users can utilize tools and software they already possesses or will have little difficulty gaining, such as web browsers. 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. This means 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, which means users do not have specific geographical information about where resources are hosted from, only on higher levels of abstraction (country, state).

**Rapid elasticity.** Automatic scaling of capabilities. Already allocated resources can expand to meet new demands. Towards the characteristic of *on-demand self-service* this means that 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 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.

### 2.1.2 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 [14] as seen as vertical integration levels in FIG. 2.1, 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 as 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 [22] 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 term in the academic world. The NIST Definition of Cloud Computing [14] define IaaS as

> "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.
>
> NIST, 2011

This underline the liberty this *service model* provide 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. NIST continue to state that

> "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).
>
> NIST, 2011

This means 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. NIST state that for developers are limited to making *"applications created using programming languages, libraries, services, and tools supported by the provider"* [14]. This means that developers are limited to capabilities the provider support, such as programming languages (Java, C#), environments (JVM, .NET, Node.js), storage systems (flat files, NoSQL databases, RDBMS). 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 API, but also support technologies such as JPA and JDO, this means users are bound to Java and these frameworks, and even limitations to the frameworks as they have specific handlers for RDBMS.

PaaS providers support deployments through online APIs, in many cases by providing specific tools such as command line interfaces or plugins to IDEs 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 *Google App Engine* (GAE) and the company Heroku with their service with the same name. 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, Nodester and Nodejitsu, 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. NIST describe SaaS as

> “The capability provided to the consumer is to use the provider’s applications running on a cloud infrastructure.
>
> NIST, 2011

The core purpose is to provide whole 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 RESTful APIs or SOAP services, but in any case it is most common to utilize the HTTP protocol. In SaaS applications end users are most likely not the companies renting from providers, but instead the companies customers. This means that 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*.

### 2.1.3 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 different *deployment models* according to The NIST Definition of Cloud Computing [14]:

**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.

**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.

**Hybrid cloud.** Combining private cloud or community cloud with public 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.

**Virtual private cloud.** Beside these models defined by NIST there is another arising model known as *virtual private cloud*, which is similar to *public cloud* but with some security implications such as sandboxed network.

## 2.2 Model-Driven Engineering

By combining the world 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 his study about 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.
>
> BOOCH, 2005

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.*” [22]. As grid computing bear similarities towards cloud computing in terms of virtualization and utility computing it is possible to use the same 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 Booch [5]:

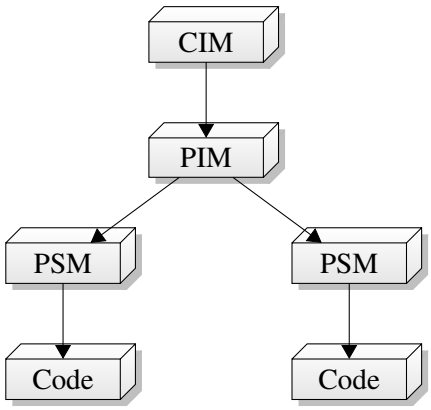



Figure 2.2: Model-Driven Architecture life cycles.

> *(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.
>
> BOOCH, 2005

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 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. 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. 2.2. There are other models and steps in between these, but they render the essentials. Beside the models there are five different life cycles as explained by Singh [21]:

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 life cycle aims at using descriptions and requirements from the CIM with specific technologies. 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. 2.2. 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 [13] 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

   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 BPMN can generate BPEL which again can generate executable logic.

5. *Deploy.* The final life cycle 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 life cycle in step 3.

# Chapter 3

# State of the Art in Provisioning

This chapter introduction must be split up and merged with the paragraph underneath!

Introduce *drivers*.

**There already exists scientific research projects and technologies which have similarities to CloudML both in idea and implementation. First scientific research projects will be presented with their solutions, then pure technological approaches will be introduced.**

One project that bears relations to CloudML is mOSAIC [18] which 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.)"* [17]. The strongest similarities to CloudML are *(i)*multicloud with their API [17], *(ii)*metadata dependencies since they support full deployment and *(iii)*robustness through fault-tolerance. What mOSAIC is lacking compared to CloudML is model-based approach including *M@RT*. Reservoir [20] is another project that also aim at *(iv)*multicloud. The other goals of this project is to leverage scalability in single providers and support built-in *Business Service Management* (BSM), important topics but not directly related to the goals of CloudML. CloudML stands out from Reservoir in the same way as mOSAIC. Vega framework [9] is a deployment framework aiming at full cloud deployments of multi-tier topologies, they also follow a *(v)*model-based approach. The main difference between CloudML and Vega are support of multicloud provisioning.

There are also distinct technologies that bear similarities to CloudML. None of AWS CloudFormation and CA Applogic are *(i)*model-driven. Others are plain APIs supporting *(ii)*multicloud such as libcloud, jclouds and DeltaCloud. The last group are projects that aim specifically at deployment, making *Infrastructure-as-a-Service* (IaaS) work as *Platform-as-a-Service* (PaaS) like AWS Beanstalk and SimpleCloud. The downside about the technical projects are their inability to solve all of the challenges that CloudML aims to address, but since these projects solve specific challenges it is appropriate to utilize them. Cloudml-engine leverages on jclouds in its implementation to support multicloud provisioning, and future versions can utilize it for full deployments.

## 3.1 Model-Driven Approaches

**Amazon AWS CloudFormation** [1]

This is a service provided by Amazon from their popular Amazon Web Services. It give users the ability to create template files in form of *JavaScript Object Notation* (JSON), which they can load into AWS to create stacks of resources. This 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 Console, and not through template files. The format that Amazon uses for the templates is a very good format, the syntax is in form of JSON which is very 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, does not make it viable as a presentation medium on a business level.

**CA Applogic** [8]

Applogic is a web application for management of private clouds. It is based on graphical models which support interactive "drag and drop" functionalities. This interface let users configure their deployments through a diagram with familiarities to component diagrams with interfaces and assembly connectors. This is one of the solutions that use and benefit from a model based approach. 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. They have made a version of ADL (Architecture Deployement Language), a good step on its way to standardization. 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.

## 3.2 APIs

**libcloud and jclouds** [3, 12]

Libcloud is a 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. jclouds is very similar to libcloud but the API code base is written in Java and Clojure. This library also have *drivers* for different providers, but they also support some PaaS solutions such as Google App Engine. APIs can be considered modelling approaches based on the fact they have a topology and hierarchical structure, but it is not a distinct modelling. A modelling 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. Libcloud have solved the multicloud problem in a very detailed manner, but the complexity is therefore even larger. The API is also Python-only and could therefor be considered to have high tool-chain dependency.

**OPA** [15]

OPA is a cloud language aimed at easing development of modern web applications. It is a new language, with its own syntax, which is aimed directly at the web. The language will build into executable files that will handle load balancing and scalability, this is to to make this a part of the language and compilation. OPA is a new language, so it might be difficult to migrate legacy systems into this lanugage. There are no deployment configurations, as this is built into the language. The compiler will generate an executable that coWeb-based vs native application

In public cloud enviromants managing, monitoring, payment and other administrative tasks can be done through web interfaces or APIs. Web applications are becoming more popular by the day, with HTML5, EcmaScript 5 and CSS3. The user experience in web applications today can in many cases match native applications, with additional benefits such as availability and ease of use. A web-based interface would prove beneficial for quickly displaying the simple core functionality of the language. In this era of cloud computing and cloud technologies a user should not need to abandon his or hers browser to explore the functionality of CloudML. Cloud providers are most likely to give customers access to customize their cloud services through web-based interfaces, and if customers are to take advantage of CloudML, the language should be graphically integrated into existing tool chains. Providers would probably find it pleasing if a example GUI wauld be run on most cloud providers instances, and so it can also benefit from some cloud based load balancers, even though this is part of the language. The conclusion about OPA is that it is not a language meant for configuration, and could not easily benefit from a model based approach, and it does not intentionally solve multicloud.

**Whirr** [4]

Fill this one out

This is a binary and code-based application for creating and starting short-lived clusters for hadoop instances. It support multiple cloud providers. It has a layout for configuration but it is mainly property-based, and aimed at making clusters.

**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 API they can call, on Deltacloud servers. This means users can write in any language they may choose. 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 software have, such as cahing, queues, redundancy

and transformations, but it also has the disadvantages such as single point of failure and version inconsistencies.

## 3.3 Deployments

**Amazon Beanstalk**

**simplifying-solution-deployment-on-a-cloud-through-composite-appliances**

**architecture-for-virtual-solution-composition-and-deployment**

# Chapter 4

# Challenges in the cloud

## 4.1 Outlined problem

To recognize challenges when doing cloud provisioning an example application [6] was 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. *BankManager* is designed but not limited to support distribution between several nodes. Some examples of provisioning topology is illustrated in FIG. 4.1, 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. 4.1(a). In FIG. 4.1(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. 4.1(c), the number of front-ends can be linearly increased $n$ number of times as shown in FIG. 4.1(d). *BankManager* is not designed to handle several back-ends because of relational model based database, this can solved on a database level with master and slaves (FIG. 4.1(e)). Bash-scripts were used to prototype full deployments of *BankManager* against *Amazon Web Services* (AWS) [1] and Rackspace [19] with a topology of three nodes as shown in FIG. 4.1(c). From this prototype it became clear that there were multiple challenges to address:

- **Complexity**: The first challenge encountered was 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 [19] had no tools beside the API language bindings, thus the need of operating both command-line tools as well as public APIs. As this emphasizes the complexity even further it also stresses engineering capabilities of individuals executing the tasks to a higher technical level.
- **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 was not as convenient as anticipated. There were certain aspects of vendor lock-in, so each script was hand-crafted for

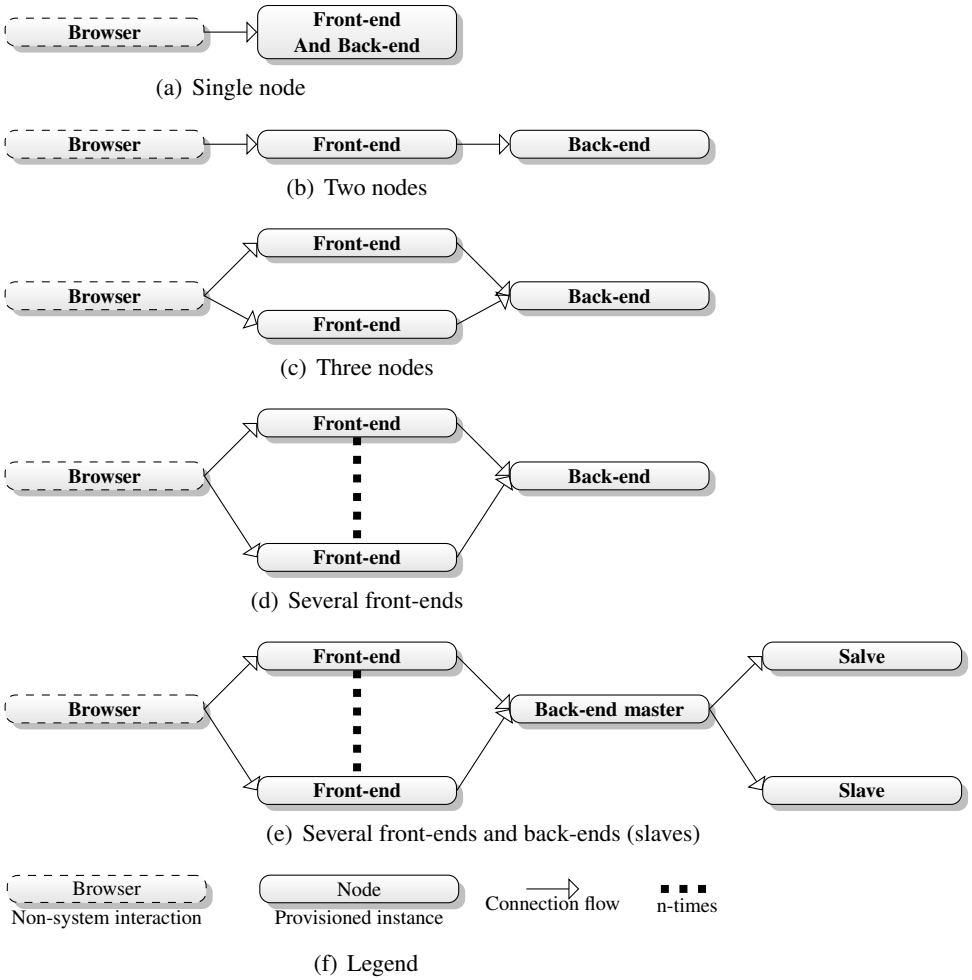



Figure 4.1: Different architectural ways to provision nodes (topologies).

Table 4.1: Analysis

| **Solution** | **Learning curve** | **Business level viable** | **Model driven** | **Multicloud** |
|---|---|---|---|---|
| Amazon CloudFormation | No | Hard | No | No |
| CA Applogic | Yes | Easy | Yes | N |
| Libcloud | No | Hard | No | Yes |
| jclouds | No | Hard | No | Yes |
| OPA | Yes | Hard | No | No |
| Whirr | No | Hard | No | Yes |
| Deltacloud | No | Hard | No | Yes |
| CloudML | Yes | Easy | Yes | Yes |

Table 4.2: Analysis

specific providers. 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.
- **Shareable**: Since the scripts did not remember a given setup it was 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.
- **Robustness**: There were several ways the scripts could fail and most errors were ignored. Transactional behaviors were non-existent.
- **Run-time dependency**: The scripts were developed to fulfill a complete deployment, and to do this it proved important to temporally save run-time specific metadata. This was crucial data needed to connect front-end nodes with the back-end node.

## 4.2 Why is it important to solve the problems

- Cloud domain is state of the art
- model driven approach with benefits (no special tooling)
- Easier for businesses (especially SMBs) to reach out to Cloud
- Easier for larger more time-constraint businesses to try out the cloud
- Opening the eyes of big providers for a larger cross-cloud language

# Chapter 5

# Requirements to solution

**Model** When approaching a global audience consisting of both academics and professional 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.

**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 datacenter will withhold commitment to affiliations. Cloud computing users are 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 Amazon AWS and Rackspace.

**Executable** The language must be dependant of an underlying engine, this is because creating stacks can be in form of a process, and the language should not be an impediment for deployment flows. The engine will not be a part of the PIM version of CloudML, but the language must reinforce this reasoning.

**API** The engine underlying CloudML should be easily accessible on a state of the art basis. This is most correctly achieved by implementing an REST based API, which can process CloudML template files correctly.

**Versoning** The file format should be in such form it can be stored a VCS system such as Git, Subversion or Mercurial. This is important for end users to be able to maintain templates that defines the stacks they have built, for future reuse.

Table 5.1: Requirements

| Requirement | Short description | Importance |
|---|---|---|
| Lexical model | Language should be based on a lexical model. | 3 |
| Graphical model | Lexical model should be represented in diagrams. | 2 |
| Multicloud | The language should work against more than one provider. | 2 |
| Adaptable (?) | Providers should be able to express what they offer according to the CloudML vocabulary to support automation. | 3 |
| Executable | The language will be accompanied by an execution engine able to process it and perform static analysis on a given CloudML file. | x |
| API | The language should be easy to use through an API. | x |
| Versoning (VCS) | The lexical language should be easy to maintain in a VCS such as Git, Mercurial or SVN. | x |

Table 5.2: Requirements

**Granularity** Cloud computing is often defined into different categories, such as IaaS (Infrastructure as a Service), PaaS (Platform as a Service) and SaaS (Software as a Service), although for CloudML it needs to narrow it down or rather redefine the point of view. The concepts around the language are not defined by what levels of a vendor management responsibilities it should support, but rather more concretely what parts of a system stack that can be configured.

The figure above, Figure 1, show the different layers that CloudML can and should support. The top most level is services that a provider might support, such as CDN, geo-based serving, monitoring and load balancing. All in all services that are external from customers actual application, but that can influence or monitor it. The next level is software, this is for any software that are co-existing with or for the customers application, such as databases, application servers, logging services. All in all software that are running on the same instance as the application, but that the customer would like to have automatically or semi-automatically configured and reconfigured. On the bottom there are two levels, both representing instances. In the instance-level CloudML should bind together instances such as different virtual machines. This level is tightly connected to the Software-layer as connections between instances is very likely to be defined through software, such as , web accelerators and application servers.

# Part II

# Contribution

# Chapter 6

# Envision, concepts and principles

The core envision is to tackle challenges from CHAP. 4 by applying a model-driven approach supported by modern technologies. Main objective is to create a common model for nodes as a platform-independent model [16] to justify *multicloud* differences and at the same time base this on a human readable lexical format to resolve *reproducibility* and make it *shareable*.

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 envisioned to parse and execute template files representing topologies of instances in the cloud. Targeted users are application developers without cloud specific knowledge. The same files should be usable on other providers, and alternating the next deployment stack should be effortless. Instance types are selected based on properties within the template, and additional resources are applied when necessary and available. While the tool performs provisioning metadata of nodes is available. In the event of a template being inconsistent with possibilities provided by a specific provider this error will be informed to the user and provision will halt.

There are many cloud providers on the global market today. These providers support many layers of cloud, such as PaaS (Platform as a Service) and IaaS (Infrastructure as a Service). 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. These are some of the problems with cloud hosting today, and that CloudML will be designed to solve.

# Chapter 7

# Analysis and design - CloudML

## 7.1 Meta model.

The meta model for CloudML is visualized in FIG. 7.1. `CloudMLEngine` is the main entry point, it has the method `build` which is used to initialize provisioning. `Property` have four children but is designed to be extendable in case new types of properties should be included. The same design principle is applied to `RuntimeProp`. `UserLibrary` visualizes that `Account` and `Template` are physical parts maintainable by the user.

**Scenario introduction.** CloudML is introduced by using two scenarios where "Alice" is provisioning the *BankManager* from CHAP. 4 to AWS *Elastic Compute Cloud* (EC2) using the topology shown in FIG. 4.1(a) and FIG. 4.1(c). It is compulsory that she possesses an AWS account in advance of the scenario. She will retrieve security credentials for account and associate them with `Password` in FIG. 7.1. `Credential` is used to authenticate her to supported providers through `Connector`. 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.

**Scenario with one single node.** The first scenario Alice want to establish is a single node based one (FIG. 4.1(a)). Since this single node will handle both computation and storage Alice decides to increase capabilities of both processing (number of `Cores`) and `Disk` size on the `Node`.

**Scenario with three nodes.** The second scenario is based on FIG. 4.1(c) with two more nodes than in the first scenario. Alice models the appropriate `Template` consisting of three `Nodes`. by increasing amount of `Cores`, and increased `Disk` for back-end `Node`.

**Provisioning.** With these models Alice can initialize provisioning by calling `build` on `CloudMLEngine`, and this will start the asynchronous job of configuring and creating `Nodes`. 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

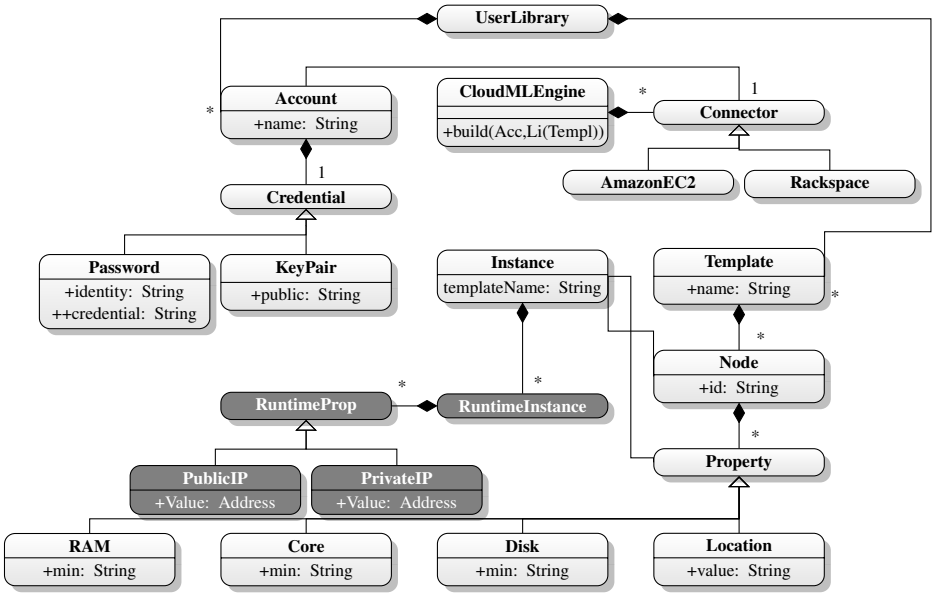
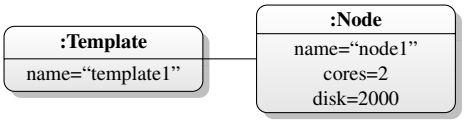
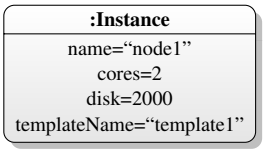



Figure 7.1: Architecture of CloudML

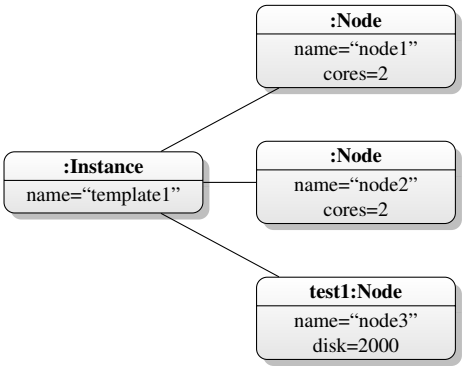



(a) Template with nodes

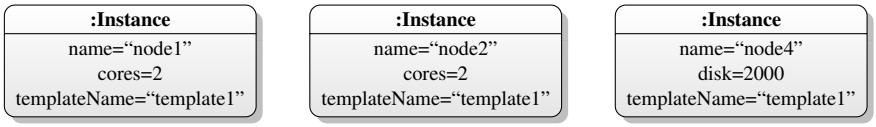



(b) Instance

Figure 7.2: Object diagram of scenario with one node




(a) Template with nodes




(b) Instance

Figure 7.3: Scenario1

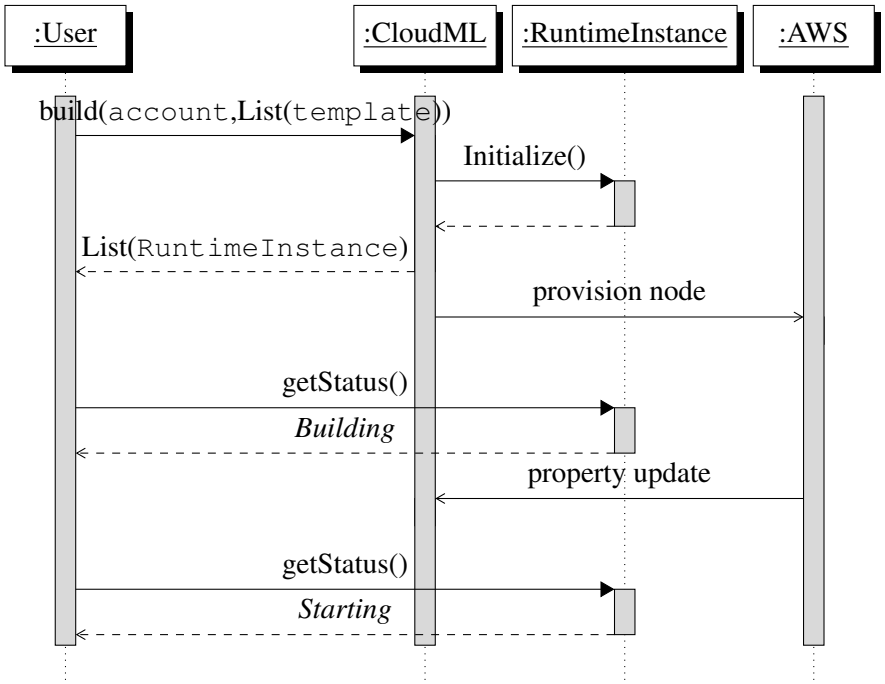



Figure 7.4: Sequence diagram of CloudML.

according to *models@run.time* (M@RT) approach. `RuntimeInstance` is specifically designed to complement `Node` with `RuntimeProperties`, as `Properties` from `Node` still contain valid data. When all `Nodes` are provisioned successfully and sufficient metadata are gathered Alice can start the deployment, CloudML has then completed its scoped task of 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.

**And we saw?**

## 7.2 Actors model

Provisioning nodes is by its nature an asynchronous action that can take minutes to execute, therefore CloudML relied on the actors model [11]. With this asynchronous solution CloudML got concurrent communication with nodes under provisioning. The model is extended by adding a callback-based pattern allowing each node to provide information on property and status changes. 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 with the actors model. The terms are divided for a node before and under provisioning, the essential is to introduce *M@RT* to achieve a logical separation. When a node is being propagated it changes type to `RuntimeInstance`, which can have a different *states* such as *Configuring*, *Building*, *Starting* and *Started*. 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.

**Full deployment is planed for next version of CloudML.**

# Chapter 8

# Implementation/realization - cloudml-engine

The envision and design of CloudML is implemented as a proof-of-concept project *cloudml-engine*. The project is split into four different modules (FIG. 8.1). Each module serves a logical task of CloudML. This chapter will go into depths of technologies and structures of the implementation.

**Implementation.** CloudML is implemented as a proof of concept framework [7] (from here known as *cloudml-engine*). Because of Javas popularity cloudml-engine was written in a JVM based language with Maven as build tool. Cloudml-engine use jclouds.org library to connect with cloud providers, giving it support for 24 providers out of the box to minimize *complexity* as well as stability and *robustness*.

## 8.1 Technologies

*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 still important to understand which technologies were chosen, what close alternatives exists and why they were chosen.

**Language.** *Cloudml-engine* is written in Scala, a multi-paradigm JVM based programming language. This language was chosen because JVM is a popular platform, and then especially Java. Scala is compatible with Java and Java can interact with libraries written in Scala as well. The reason not to use plain Java was because Scala is an appealing state-of-the-art language that emphasizes on functional programming which is leveraged in the implementation. Scala also has a built in system for Actors model [11] which is utilized in the implementation.

**Lexical format.** For the lexical representation of CloudML *JavaScript Object Notation* (JSON) was chosen. JSON is a web-service friendly, human-readable data interchange format and an alternative to XML. This format was chosen because of popularity in the cloud community source? and its usage area as data

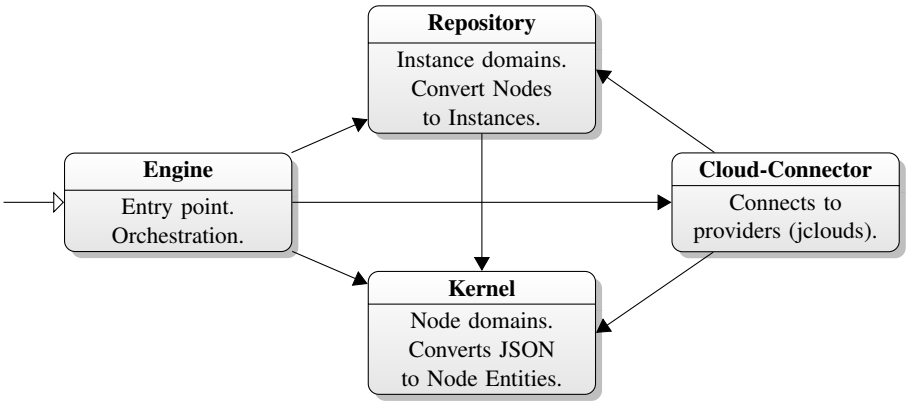



Figure 8.1: Architecture of cloudml-engine

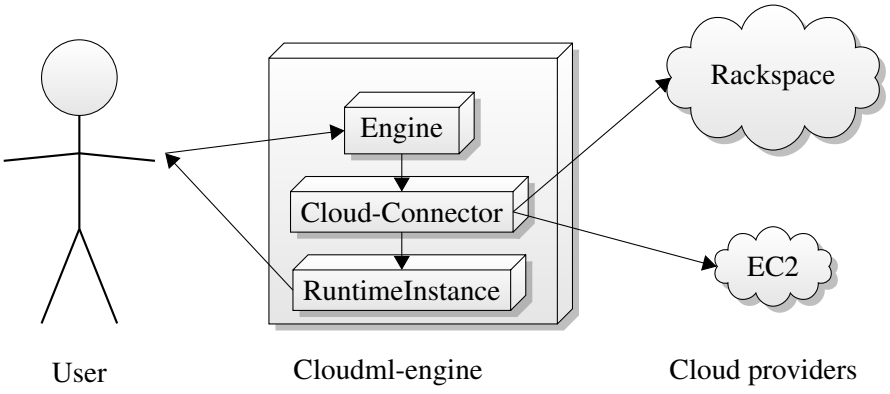



Figure 8.2: Usage flow in cloudml-engine

```
<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>0.1</version>
 </dependency>
</dependencies>
```

Figure 8.3: Example Maven configureation section to include cloudml-engine

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

Figure 8.4: Example Scala callout to *cloudml-engine*

transmit format between servers and web applications. This means *cloudml-engine* can be extended to work as a RESTFul web-service server.

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.

**Automatic build system.** There are two main methods used to build Scala programs, either using a Scala-specific tool called *Scala Build Tool* (SBT) or a more general tool called Maven. For *cloudml-engine* to have an academic appeal it were essential to choose the technology with most closeness to Java, hence Maven was chosen. Maven support modules which were used to split *cloudml-engine* into the appropriate modules as shown in FIG. 8.1. The dependency system in Maven between modules is used to match the dependencies outlined in FIG. 8.1. Parts of a dependency reference in a Maven configuration can be seen in FIG. 8.3, although this is not dependency management in between *cloudml-engine* modules but rather how to add *cloudml-engine* as a dependency itself.

**Cloud connection.** The bridge between *cloudml-engine* and cloud providers is an important aspect of the application, and as a requirement it was important to use an existing library to achieve this connection. Some libraries have already been mentioned in the *APIs* section in CHAP. 3, of these only *jclouds* is based on Java-technologies and therefore suites *cloudml-engine*. 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.

**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 [10] as a Maven module. This repository is provided by CloudBees, how to include the library is viewable in FIG. 8.3.

**Actors.** As mentioned earlier *cloudml-engine* utilizes the actors model through Scala, this approach is used to achieve asynchronous provisioning. This is important as provisioning can consume up to minutes for each instance. 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.

## 8.2 Modules and application flow

*Cloudml-engine* is divided into four main modules (FIG. 8.1). This is to distribute workload and divide *cloudml-engine* into logical parts for each task.

**Engine.** The main entry point to the application, this is a Scala Object used to initialize provisioning. Interaction between `user` and `Engine` is visible in FIG. 8.2 where the user will initialize provisioning by calling `Engine`. `Engine` will also do orchestration between the three other modules as shown in FIG. 8.1. Since `Cloud-Connector` is managed by `Engine` other actions against instances are done through `Engine`. The first versions of *cloudml-engine* 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.

**Kernel.** `Kernel` contains CloudML specific entities such as Node and Template. The logical task of `Kernel` is to map JSON formatted strings to `Templates` including `Nodes`. This is some of the core parts of the DSL, hence it is called *`Kernel`*. `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.

**Repository.** Has `Instance` entities, 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, for instance at the writing moment it is not possible to demand LoadBalancers on Rackspace for specific geographical locations.

**Cloud-Connector.** is the module bridging between *cloudml-engine* and providers. It does not contain any entities, and only does logical code. It is built to support several libraries and interface these. At the moment it only implements the earlier mentioned library jclouds.

# Chapter 9

# Validation & Experiments

- How BankManager proves concepts of the templates (subsection 1) with cloudml-engine

To validate how CloudML addressed the challenges from TABLE. **??** The *BankManager* application were provisioned using different topologies in Fig[4.1(a), 4.1(c)]. The implementation uses *JavaScript Object Notation* (JSON) to define templates as a human readable serialization mechanism. The lexical representation of FIG. 4.1(a) can be seen in LISTING. **??**. The whole text represents the `Template` of FIG. 7.1 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*).

The topology described in FIG. 4.1(c) is represented in LISTING. **??**, the main difference from LISTING. **??** is that there are two more nodes and a total of five more properties. Characteristics of each node are carefully chosen based on each nodes feature area, for instance front-end nodes have more computation power, while the back-end node will have more disk.

# Part III

# Conclusion

# Chapter 10

# 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 11

# Results

# Chapter 12

# Perspectives

# Bibliography


[1] Amazon. Amazon web services, 2012.

[2] Apache. Deltacloud, 2012.

[3] Apache. Libcloud, 2012.

[4] Apache. Whirr, 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.

[6] Eirik Brandtzæg. Bank manager, 2012.

[7] Eirik Brandtzæg. cloudml-engine, 2012.

[8] CA. Applogic, 2012.

[9] 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.

[10] CloudBees. Cloudbees cloudml-engine repository, 2012.

[11] 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.

[12] jclouds. jclouds, 2012.

[13] Stuart Kent. Model Driven Engineeringx , booktitle = Integrated Formal Methods, series = Lecture Notes in Computer Science, editor = Butler, Michael and Petre, Luigia and Sere, Kaisa, publisher = Springer Berlin / Heidelberg, isbn = 978-3-540-43703-1, keyword = Computer Science, pages = 286-298, volume = 2335, year = 2002.

[14] 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.

[15] MLState. Opa, 2012.

[16] 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.

[17] Dana Petcu, Ciprian CrÄČciun, 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.

[18] Dana Petcu, Georgiana Macariu, Silviu Panica, and Ciprian CrÄČciun. Portable Cloud applicationsâĂŤFrom theory to practice. *Future Generation Computer Systems*, (0):–, 2012.

[19] Rackspace. Rackspace cloud, 2012.

[20] 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.

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

[22] 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.