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: 10th April 2012

Chapter 1

Junk

Table 1.1: 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 1.2: Analysis

Abstract

Contents

List of Figures

List of Tables

Preface

Part I

Introduction

Chapter 2

Introduction

Short and sharp

- Main introduction
- Write lastly

Talk about the two submitted papers (CloudMDE, Sebastiens paper). Involvement in REMICS (Deliverable 4.1).

These should also be woven into *contribution*

Chapter 3

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.

3.1 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 **ac:NIST!** (**ac:NIST!**) which is one of the leaders in cloud computing standardization. The main providers of cloud computing at writing moment are Google, Amazon with **ac:AWS!** (**ac:AWS!**) [**?**] and Microsoft. A non-exhaustive list of common providers is reproduced TABLE. **??**. A good example of cloud adaptation in a large scale scenario is when the White House moved their Revocery.gov [**?**] 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 section is based on the **ac:NIST!** standard.

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 3.1: Common providers available services

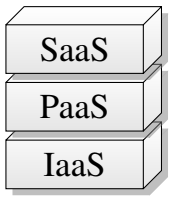


Figure 3.1: Cloud architecture service models

3.1.1 Characteristics

Cloud computing is about providing computation as services [**?**], such as virtual instances and file storage, rather than products. Cloud characteristics are what 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 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 estimating amount user demand and total amount of incoming requests. For instance in the year 2011 they achieved a record in **ac:TPS!** (**ac:TPS!**) of 25, 088, 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 **ac:HTTP!** (**ac:HTTP!**)/HTTPS and **ac:SSH!** (**ac: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. Automatic scaling of capabilities. 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 **ac:RAM!** (**ac:RAM!**) and **ac:CPU!** (**ac: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 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.

3.1.2 Service models

Service models are definitions of different layers in cloud computing. The different models The layers refer to different levels of abstractions ways of 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 [**?**] as seen as vertical integration levels in FIG. **??**, namely **ac:IaaS!** (**ac:IaaS!**), **ac:PaaS!** (**ac:PaaS!**) and **ac:SaaS!** (**ac: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 **ac:VPS!** (**ac:VPS!**), and is therefore the *service model* closest to standard hosting solutions. Stanoevska-Slabeva [**?**] 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 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.

> "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

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 **ac: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 **ac:AWS! ac:EC2!** (**ac: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 (**ac:JVM!** (**ac:JVM!**), .NET, Node.js), storage systems (flat files, NoSQL databases, **ac:RDBMS!** (**ac:RDBMS!**)), services (load balancers, backup, content delivery) and tools (plugin for Eclipse, command line tools) [**?**]. For example the first versions of **ac:GAE!** (**ac: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 **ac:API!** (**ac:API!**), but also support technologies such as **ac:JPA!** (**ac:JPA!**) and **ac:JDO!** (**ac:JDO!**), users are bound to Java and these frameworks, and even limitations to the frameworks as they have specific handlers for **ac:RDBMS!**. The disconnection from operating system is the strength of **ac:PaaS!** solutions, on one hand developers are restricted, but they are also freed from configuration, installments and maintaining deployments. Some **ac: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 **ac:API!**s, in many cases by providing specific tools such as command line interfaces or plugins to **ac:IDE!** (**ac:IDE!**)s like Eclipse. It is common for the **ac: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 **ac: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 **ac:EC2!** as IaaS underneath. Multiple PaaS providers utilize **ac: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.
>
> NIST, 2011

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 **ac:REST!** (**ac:REST!**) **ac:API!**s or SOAP services, but in any case it is most common to utilize the **ac: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*.

3.1.3 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 **ac:AWS!** and Google with **ac: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 **ac:NIST!** there is another arising model known as **ac:VPC!** (**ac: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. 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. 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.

3.2 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 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.
>
> BOOCH *et al.* , 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.*” [**?**]. As grid computing bear similarities towards cloud computing in terms of virtualization and utility computing it is possible to use the same **ac:UML!** (**ac: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 [**?**]:

> “*(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 *et al.* , 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 **ac:UML!** diagrams or even diagrams at all. As an example **ac:AWS!** CloudFormation implements a lexical model of their *cloud services*, while CA AppLogic has a visual and more **ac:UML!** component-based diagram of their capabilities.

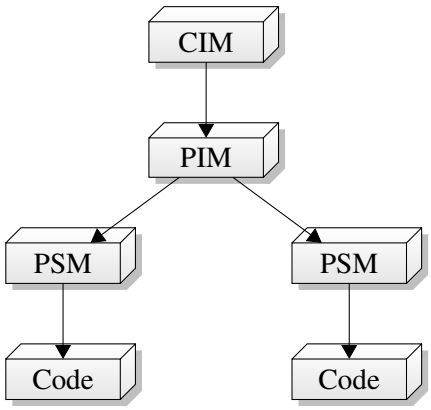


Figure 3.2: Model-Driven Architecture life cycles.

Model-Driven Architecture. **ac:MDA!** (**ac:MDA!**) is a way of designing software with modeling in mind provided by **ac:OMG!** (**ac:OMG!**). When working with **ac:MDA!** it is common to first create a **ac:CIM!** (**ac:CIM!**), then a **ac:PIM!** (**ac:PIM!**) and lastly a **ac:PSM!** (**ac:PSM!**) as seen in FIG. **??**. 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 [**?**]:

1. *Create a **ac:CIM!***. This is done to capture requirements and describe the domain. To do this the **ac: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 **ac:CIM!** models can be compared to *use case* diagrams in **ac:UML!**, where developers can model actors and actions (use cases) based on a specific domain.

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

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

> “A **ac:PSM!** is not a **ac:PIM!**, but is also not an implementation. [...] In **ac:MDA!**, the main mapping is between **ac:PIM!** and **ac: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 **ac:PSM!** is more specific to a platform then **ac:PIM!**, such as programming language or environment.

4. *Generate code form **ac:PSM!***. A **ac: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 **ac:BPMN!** (**ac:BPMN!**) can generate **ac:BPEL!** (**ac:BPEL!**) which again can generate executable logic.

5. *Deploy.* The final life cycle is based on deploying the **ac: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 **ac:PSM!** from life cycle in step 3.

Chapter 4

State of the Art in Provisioning

There already exists scientific research projects, **ac:API!**s, 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. *(i)*Model-Driven Approaches which aims presenting frameworks and projects that utilize models on a larger scale. *(ii)***ac:API!**s are about frameworks that connects to cloud providers, often with multicloud support, these projects can be used as middleware for other systems. *(iii)*Deployments are about projects that do full deployment of applications, inclusing provisioning. These are often more academic. Lastly the chapter will discuss *(iv)*Example of cloud surveys, some real-world examples that might not be directly related to provisioning but are important and interesting regardless.

4.1 Model-Driven Approaches

The following technologies are in some way model based. That means they either use concrete diagrams or any other means of modeling that can relate to Model-Driven Engineering.

Amazon AWS CloudFormation. [**?**]

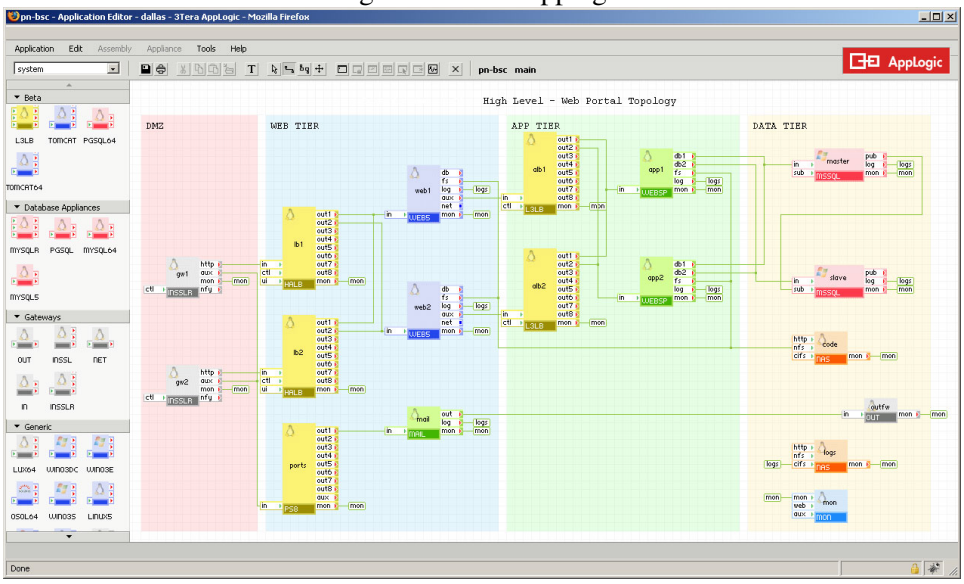
This is a service provided by Amazon from their popular **ac:AWS!**. It give users the ability to create template files in form of **ac:JSON!** (**ac:JSON!**) as seen in FIG. **??**, which they can load into **ac:AWS!** to create stacks of resources. A *stack* is a defined set of resources in different amount and sizes, such as numerous instances, one or more databases and a load balancer, although what types and sizes of resources is ambiguous. To provision a stack with CloudFormation the template file (in **ac:JSON!** format) is first uploaded to **ac:AWS!** which makes it accessible from **ac: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*

```
1
2  "Description": "Create an EC2 instance",
3  "Parameters": {
4    "KeyPair": {
5      "Description": "For SSH access",
6      "Type": "String"
7    }
8  },
9  "Resources": {
10   "Ec2Instance": {
11     "Type": "AWS::EC2::Instance",
12     "Properties": {
13       "KeyName": { "Ref": "KeyPair" },
14       "ImageId": "ami-1234abcd"
15     }
16   }
17 },
18 "Outputs" : {
19   "InstanceId": {
20    "Description": "Instace ID of created instance",
21    "Value": { "Ref": "Ec2Instance" }
22   }
23 },
24 "AWSTemplateFormatVersion": "2010-09-09"
25
```

Figure 4.1: AWS CloudFormation template

Figure 4.2: CA Applogic



in Java package style. The last section, *Output*, will generate output to users when provisioning is complete, here it is possible for 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 **ac: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 **ac: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 **ac:JSON!** is a readable format, it does not make it viable as a presentation medium on a business level.

CA Applogic. [**?**]

The Applogic platform is designed to manage CAs private cloud infrastructure [**?**]. It also has a web based interface which let users manage their cloud resources as shown in FIG. **??** which use and benefit from a model based approach. 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 **ac: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

Figure 4.3: Madeira Cloud

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. [?]

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 **ac:AWS! ac:EC2!**. Users can create *stacks* with the available services in **ac:AWS!** through dynamic diagrams. These *stacks* are live representations of the architecture and can be found and managed in the **ac:AWS!** console as other **ac: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.

4.2 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 **ac:API!**s, 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 **ac:REST!** form. **ac:REST!** [?] is a software architecture for management of web resources on a service. It uses the **ac:HTTP!** and consecutive methods such as GET, POST, PUT and DELETE to do tasks such as retrieve lists, items and create items. **ac:API!**s 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 **ac:API!** lacks correlation between resources and method calls.

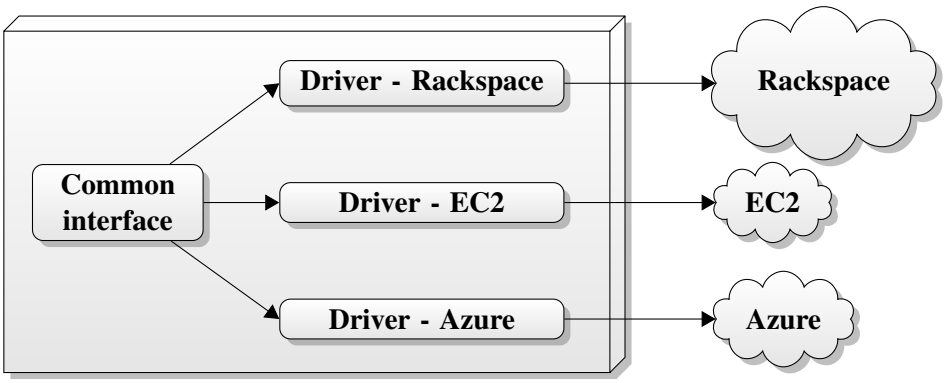


Figure 4.4: Cloud drivers

Driver. any of the **ac: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 **ac:API!** (**ac:REST!**), illustrated in FIG. **??**.

jclouds. [**?**]

This is a library written in Java and can be used from any **ac:JVM!**-based language. Provider support is implemented in *drivers*, and they even support deployments to some **ac:PaaS!** solutions such as **ac:GAE!**. *jclouds* divide their library in two different parts, one for computing powers such as **ac:EC2!** and one for blob storage like S3. Some blob storage services are accessible on the compute side of the library such as **ac:EBS!** (**ac: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. [**?**]

Libcloud is an **ac:API!** that aims to support the largest cloud providers through a common **ac: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 **ac:API!** code base is written in Python. The **ac:API!** is Python-only and could therefor be considered to have high tool-chain dependency.

Deltacloud. [**?**]

Deltacloud has a similar procedure as jclouds and libcloud, but with a **ac:REST! ac: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 **ac:API!** they can call on Deltacloud servers. As well as having similar problems as other **ac:API!**s 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 **ac:REST! ac:API!**. Previously discussed *jclouds* also support Deltacloud, as it would interface this with a *driver* as any other web-based **ac:API!**.

4.3 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. [**?**]

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.)"* [**?**]. 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 **ac:IaaS!** solutions of cloud providers to deploy users application, then communication between these clouds will be done using "cloud based message queues technologies".

RESERVOIR. [**?**]

ac:RESERVOIR! (**ac: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 **ac:VEE!** (**ac:VEE!**) which is managed by a **ac:VEEM!** (**ac:VEEM!**). The **ac:VEEM!** communicate with other **ac: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. [**?**]

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 **ac:XML!** (**ac:XML!**) files, with these files developers can replicate a *stack*. The **ac: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 **ac: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.

4.4 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 **ac:EC2!** and Rackspace cloudservers, for these only one solution will be discussed.

EC2. A central part of **ac:AWS!**, it was Amazons initial step into the world of cloud computing when they released **ac: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 **ac:IaaS!** provider. When users rent **ac:EC2!** services they are actually renting **ac:VPS!** instances virtualized by Xen. Although the instance itself can be considered a **ac:VPS!** there are other factors that define it as a cloud service. For instance cost calculations, monitoring and tightly coupled services surrounding **ac:EC2!**. Examples of these services are **ac:EBS!** for block storage, **ac:ELB!** (**ac:ELB!**) for load balancing and *Elastic IP* for dynamically assigning static IP addresses to instances.

Some of **ac:AWS!** other services rely on **ac:EC2!** such as **ac:AWS!** Elastic Beanstalk and *Elastic MapReduce*. When purchasing these services it is possible to manage the **ac:EC2!** instances through the **ac:EC2!**-tab in **ac:AWS!** console, but this is not mandatory as they will be automatically managed through the original purchased service. As mentioned earlier other **ac:PaaS!** solutions delivered by independent companies are using **ac:EC2!** or other **ac:AWS!** solutions. Examples of these are Heroku, Nodester, DotCloud and Engine Yard which uses **ac:EC2!**. Example of companies using other **ac:AWS!** services is Dropbox which uses S3j

ac: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 **ac: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 (**ac: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 **ac:EC2!** for computing, **ac:EBS!** for storage and **ac: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 5

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 challenges identified from the research will be presented.

5.1 Scenario

The following scenario was 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 [**?**] 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. The application is based on a three-tier architecture with *(i)*presentation tier with 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 **ac:MVC!** (**ac: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 **ac:SOA!** (**ac:SOA!**) solutions.

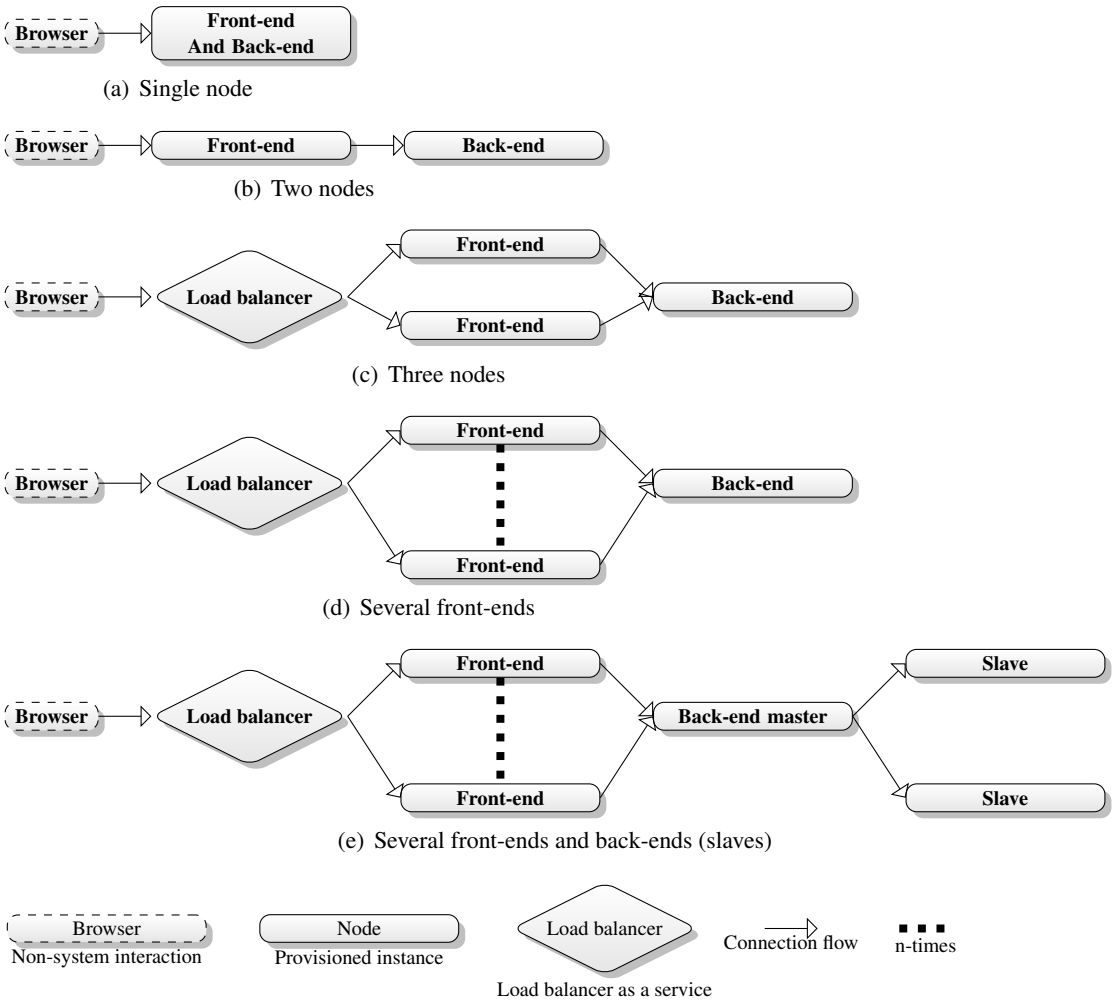


Figure 5.1: Different architectural ways to provision nodes (topologies).

Topologies. Some examples of provisioning topologies are illustrated in FIG. **??**. 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. **??**. 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. **??** `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. **??**, the number of `front-ends` can be linearly increased n number of times as shown in FIG. **??**. *BankManager* is not designed to handle several `back-ends` because of **ac:RDBMS!**, this can be solved at the database level with master and slaves

(FIG. **??**).

Execution. The main goal of the scenario was to successfully deploy *BankManager* on a given set of providers with a given set of topologies. And to achieve such deployment it was crucial to perform cloud provisioning. The providers chosen were *(i)***ac:AWS!** [**?**] and *(ii)*Rackspace [**?**]. These are strong providers with a respectable amount of customers, as two of the leaders in cloud computing [source]. 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 was that of three nodesFIG. **??**. 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 were developed. These scripts were designed to automate a full deployment on a two-step basis. First step was to provision instances:

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

The second step was deployment:

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

The scripts were 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 **ac:REST!** client had to be constructed.

5.2 Challenges

From this research 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 is to enable a successful deployment. It was therefore crucial to involve a full deployment in the scenario to discover important challenges.

Complexity. The first challenge encountered was to simply authenticate and communicate with the cloud. The two providers had different approaches, **ac:AWS!** [**?**] had command-line tools built from their Java APIs, while Rackspace [**?**] had no tools beside the **ac:API!** language bindings. So for **ac:AWS!** the Bash-scripts could do callouts to the command-line interface while for Rackspace the public **ac:REST! ac: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 were 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 **ac:REST!** interfaces were essentially muted away. If one specific script should fail it was 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 was not as convenient as anticipated. There were certain aspects of vendor lock-in, so each script was 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 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. 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 were several ways the scripts could fail and most errors were ignored. They were made to search for specific lines in strings returned by the APIs, if these strings were 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 **ac:WAR!**-file and *(v)*deploy and start the application from the **ac: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 was 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)*.

Chapter 6

Requirements

The requirements are descriptions of important aspects and needs derived from the previous chapter, CHAP. **??** where challenges were identified. In this chapter the requirements will be listed and described. 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.

Underlying technologies. Beside the benefits of *software reuse* there could be even additional gain by choosing a solid technology underneath the library, meaning *programming language*, *application environment*, *common libraries*, *distribution technologies* etc. 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 *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.

Table 6.1: Requirements

Challenge	**Addressed by**
Complexity	Software reuse. Model-Driven approach.
Feedback on failure	Software reuse.
Multicloud	Software reuse.
Reproducibility	Lexical templates.
Sharable	Lexical templates.
Robustness	Software reuse. Underlying technologies.
Metadata dependency	Models@run.time. Underlying technology.

Table 6.2: Requirements

Software reuse. There were several technological difficulties with the scripts from the scenario in CHAP. **??**. 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. **??** multicloud **ac:API!**s 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. **??**. 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 underlying technology. And for the same reasons such framework could make the whole application more `robust`. All of the libraries from CHAP. **??** support `multicloud` so they can interact with several providers over a common interface, this would be a mandatory challenge to overcome by this requirement.

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`.

Lexical template. This requirement is tightly coupled with that of *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 **ac:JSON!** or **ac:XML!**.

Textual templates that can be shared through mediums such as e-mail or **ac:VCS!** (**ac: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.

Models@run.time. Models that reflect the provisioning models and updates asynchronously. As identified by the scenario in CHAP. **??** metadata from provisioning was crucial to perform a proper deployment in steps after the provisioning was 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.

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 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, **ac:API!**s, payment methods and services. This requirement anticipate support for at least two different providers such as **ac:AWS!** and Rackspace.

Part II

Contribution

Chapter 7

Envision, concepts and principles

In this chapter the core approach and steps on the research to implementing CloudML will be described. In the previous chapters, CHAP. **??** and CHAP. **??**, challenges and requirements were identified and described. In this part of the thesis (*contribution*) the challenges will be resolved by applying and implementing the requirements.

There are four main steps from start of the problem to an functional implementation.

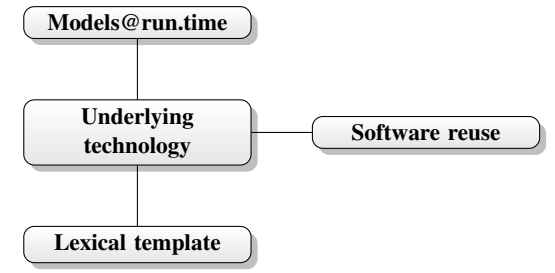
1. Identify and research in *state of the art*.
2. Recognize *challenges*.
3. Determine *requirements* based on *challenges*.
4. *Analyze* solutions, tools and procedure to implement *requirements*.
5. *Implement* a solution based on *analyzed* results.

Of these steps step 1 (one) to 3 (three) are already covered by CHAP. **??** , CHAP. **??** and CHAP. **??**. This chapter is an intermediate chapter which will introduce the chapter of CHAP. **??** and CHAP. **??**.

7.1 Core envision and concept

Note: This part of this section is almost direct copy/paste of what was already here, might not seem to "fit in" Maybe remove, thoughts?

The core envision is to solve requirements from CHAP. **??** by applying a model-driven approach supported by modern technologies. 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



(a) Requirement dependencies

(b) Legend

Figure 7.1: Requirement dependencies

being inconsistent with possibilities provided by a specific provider this error will be informed to the user and provision will halt.

7.2 Requirement dependencies

Some of the requirements have depend on each other, for instance `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 `underlying technology` requirement. There are three requirements, *(i)*`Models@run.time`, *(ii)*`Software reuse` and *(iii)*`lexical template`, where all have a two-way dependency to the *(iv)*`underlying technology` requirement, as illustrated in FIG. **??**. 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.

7.3 Requirements step by step

In this section each of the requirements from CHAP. **??** will be presented with a description overview of how to proceed to achieve them.

Underlying technologies. This will be the foundation of the solution, so finding good technologies has a high priority. To find good technologies they 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 `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)***ac:API!** in form of public Web Service or *(iii)***ac: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.

Software reuse. The **ac:API!**s described in CHAP. **??** are worthy candidates to meet 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 `underlying technology` or help fulfill this requirement. Then the chosen **ac:API!**s must be narrowed down to one single **ac:API!** which will be used in the solution application.

Model-driven approach. Unlike the other requirements this is a non-physical need, and as seen in FIG. **??** there are no dependencies from or to this requirement. But other requirements such as `lexical template` 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 `lexical template` requirement and the last by the `models@run.time` requirement.

Lexical templates. Main objective is to create a common model for nodes as a platform-independent model [**?**] 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 type of direct model representation of topologies will have great impact on the solution application. As described in CHAP. **??** this representation should be lexical, but there are several different styles and languages to achieve this. Some examples of these languages are *(i)***ac:XML!**, *(ii)***ac:JSON!**, *(iii)*YAML, *(iv)***ac:SDL!** (**ac:SDL!**) or *(v)***ac:OGDL!** (**ac:OGDL!**). As shown in FIG. **??** there is a two-way dependency between the `lexical templates-` and `underlying technology` requirement. This dependency can have impact both ways, but unlike the other dependencies in FIG. **??** 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. **??**.

Models@run.time. 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 `underlying technology`. There are several different approaches that could be made when implementing these models, such as using *(i)*standard Java classes and expect users to do *short polling* [need source?], *(ii)command pattern* to send events on node updates, *(iii)aspect-oriented programming* to enhance asynchronous support and modularity or *(iv)publish subscribe pattern* to let users subscribe to events of updating instances. 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. **??**, and *(iii)*identify approaches that can be combined and what benefits this would give.

7.4 Move or remove this

This section should probably be moved to CHAP. **??** or be completely removed, thoughts?

There are many cloud providers on the global market today. These providers support many layers of cloud, such as **ac:PaaS!** and **ac: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. **ac: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 8

Analysis and design - CloudML

8.1 Meta model.

The meta model for CloudML is visualized in FIG. **??**. `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. **??** to AWS *Elastic Compute Cloud* (EC2) using the topology shown in FIG. **??** and FIG. **??**. 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. **??**. `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. **??**). 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. **??** 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

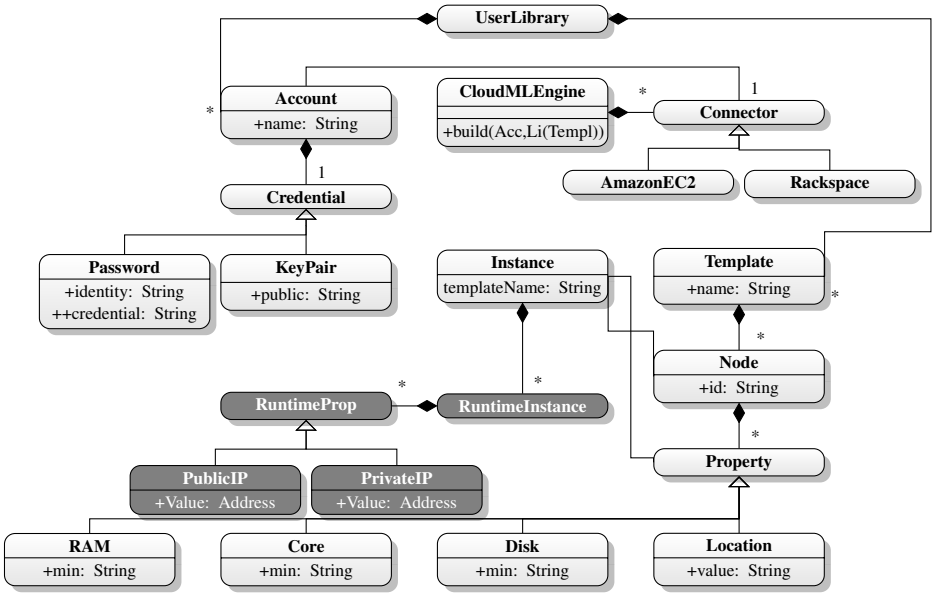
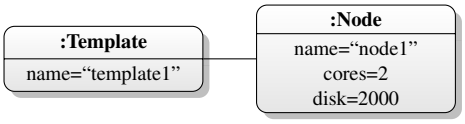
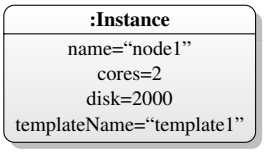


Figure 8.1: Architecture of CloudML

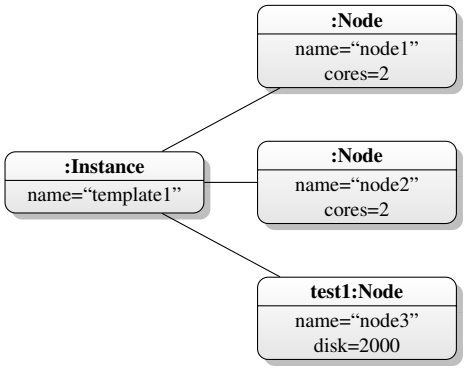


(a) Template with nodes

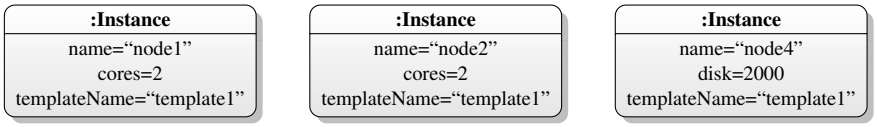


(b) Instance

Figure 8.2: Object diagram of scenario with one node



(a) Template with nodes



(b) Instance

Figure 8.3: Scenario1

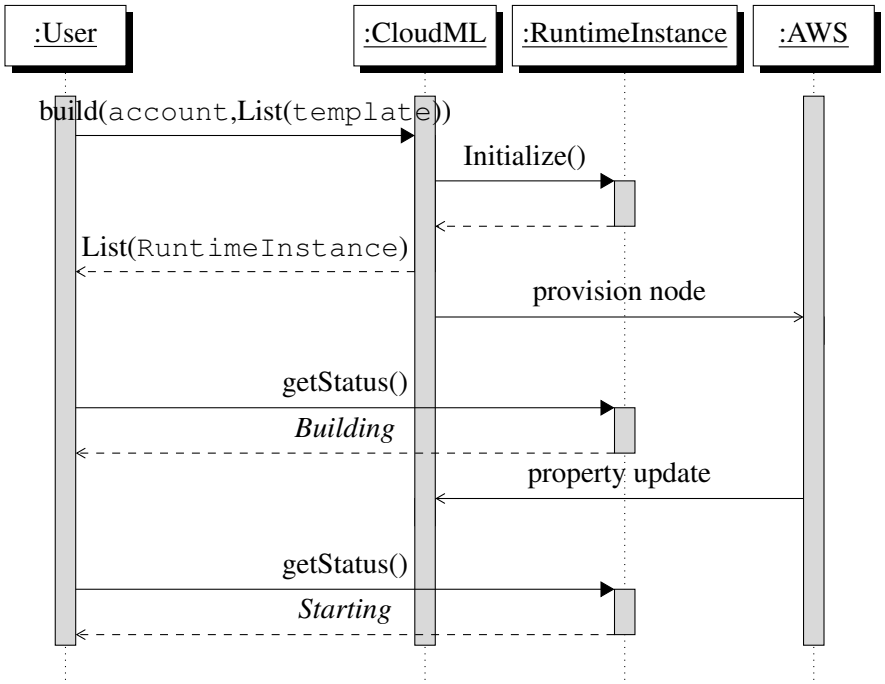


Figure 8.4: Sequence diagram of CloudML.

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?

8.2 Actors model

Provisioning nodes is by its nature an asynchronous action that can take minutes to execute, therefore CloudML relied on the actors model [**?**]. 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.

Chapter 9

Implementation/realization - cloudml-engine

Maven dependency graph

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. **??**). 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 [**?**] (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*.

9.1 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 [**?**] 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

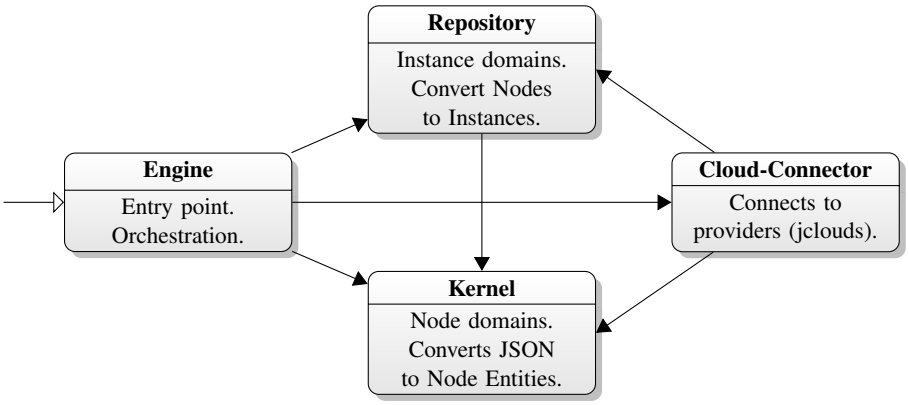


Figure 9.1: Architecture of cloudml-engine

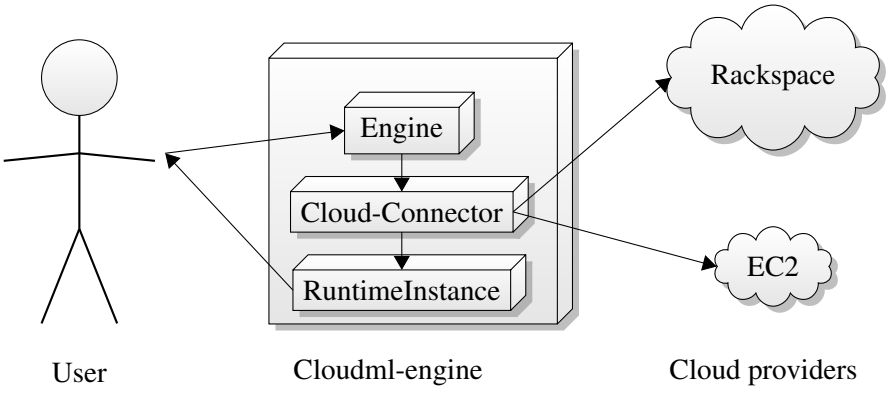


Figure 9.2: Usage flow in cloudml-engine

```
<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 9.3: Example Maven configureation section to include cloudml-engine

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

Figure 9.4: Example Scala callout to *cloudml-engine*

because of popularity in the cloud community source? and its usage area as data 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. **??**. The dependency system in Maven between modules is used to match the dependencies outlined in FIG. **??**. Parts of a dependency reference in a Maven configuration can be seen in FIG. **??**, 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. **??**, 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 [**?**] the library is deployed to a remote repository [**?**] as a Maven module. This repository is provided by CloudBees, how to include the library is viewable in FIG. **??**.

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.

9.2 Modules and application flow

Cloudml-engine is divided into four main modules (FIG. **??**). 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. **??** where the user will initialize provisioning by calling `Engine`. `Engine` will also do orchestration between the three other modules as shown in FIG. **??**. 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 10

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[**??**, **??**]. The implementation uses *JavaScript Object Notation* (JSON) to define templates as a human readable serialization mechanism. The lexical representation of FIG. **??** can be seen in LISTING. **??**. The whole text represents the `Template` of FIG. **??** 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. **??** 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 11

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 12

Results

Chapter 13

Perspectives

Full deployment is planed for next version of CloudML.