Towards CloudML, a Model-based Approach to Provision Resources in the Clouds*

Eirik Brandztæg[1,2], Sébastien Mosser[1], and Parastoo Mohagheghi[1]

[1] SINTEF IKT, Oslo, Norway
[2] University of Oslo, Oslo, Norway
{firstname.lastname}@sintef.no

Abstract. The Cloud-computing paradigm advocates the use of resources available "in the clouds". In front of the multiplicity of cloud providers, it becomes cumbersome to manually tackle this heterogeneity. In this paper, we propose to define an abstraction layer used to model resources available in the clouds. This cloud modelling language (CloudML) allows cloud user to focus on their needs, *i.e.*, the modelling the resources they expect to retrieve in the clouds. An automated provisioning engine is then used to automatically analyse these requirements and actually provision resources in clouds. The approach is implemented, and was experimented on prototypical examples to provision resources in major public clouds (*e.g.*, Amazon EC2 and Rackspace).

1 Introduction

Cloud–Computing [2] was considered as a *revolution*. Taking its root in distributed systems design, this paradigm advocates the share of distributed computing resources designated as "*the cloud*". The main advantage of using a cloud-based infrastructure is the associated scalability property (called *elasticity*). Since a cloud works on a *pay–as–you–go* basis, companies can rent computing resources in an elastic way. A typical example is to temporary increase the server–side capacity of an e–commerce website to avoid service breakdowns during a load peak. According to Amazon (one of the major actor of the Cloud market): *"much like plugging in a microwave in order to power it doesnt require any knowledge of electricity, one should be able to plug in an application to the cloud in order to receive the power it needs to run, just like a utility"* [14]. However, there is still a huge gap between the commercial point of view and the technical reality that one has to face in front of "*the cloud*".

The Cloud-computing paradigm emphasises the need for automated mechanisms, abstracted from the underlying technical layer. It focuses on the reproducibility of resource provisioning: to support the horizontal scaling of cloud-applications, such a provisioning of on-demand resources will be performed by a program. The main drawback associated is the heterogeneity of cloud providers.

* This work is funded by the European commission through the REMICS project (www.remics.eu), contract number 257793, with the 7th Framework Program.

At the infrastructure level, more than 10 different providers publish different mechanisms to provision resources in their specific clouds. It generates a *vendor lock-in* syndrome, and an application implemented to be deployed in cloud C will have to be re-considered if it now has to be deployed on cloud C'. All the deployment scripts that were designed for C have to be redesigned to match the interface provided by C' (which can be completely different, *e.g.*, shell scripts, RESTful services, standard API).

Our contribution in this paper is to describe the first version of CloudML, a cloud modelling language specifically designed to tackle this challenge. This research is done in the context of the REMICS EU FP7 project, which aims to provide automated support to migrate legacy applications into clouds [9]. Using CloudML, an user can express the kind of resources needed for a specific application, as a model. This model is automatically handled by an engine, which returns a "run-time model" of the provisioned resources, according to the models@run.time approach [3]. The user can then relies on this model to interact with the provisioned resources and deploy the application. The approach is illustrated on a prototypical example used to teach distributed systems at the University of Oslo.

2 Challenges in the cloud

To recognise challenges when doing cloud provisioning we use an example application [5]. The application (known as *BankManager*) is a prototypical bank manager system which support *(i)* creating users or bank accounts and *(ii)* 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. 1, each example includes a browser to visualize application flow, front-end visualises 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. 1(a). In Fig. 1(b) front-end is separated from the 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. 1(c), the number of front-ends can be linearly increased n number of times as shown in Fig. 1(d). *BankManager* is not designed to handle several back-ends because of the relational database, this can solved on a database level with master and slaves (Fig. 1(e)) although this is beside out of the scope of this article.

We used bash scripts to implement the full deployments of *BankManager* against *Amazon Web Services* (AWS) [1] and Rackspace [12] with a topology of three nodes as shown in Fig. 1(c). From this prototype it became clear that there were multiple challenges that we had to address:

- **Complexity**: The first challenge we encountered was to simply support authentication and communication with the cloud. The two providers we tested against had different approaches, AWS [1] had command-line tools

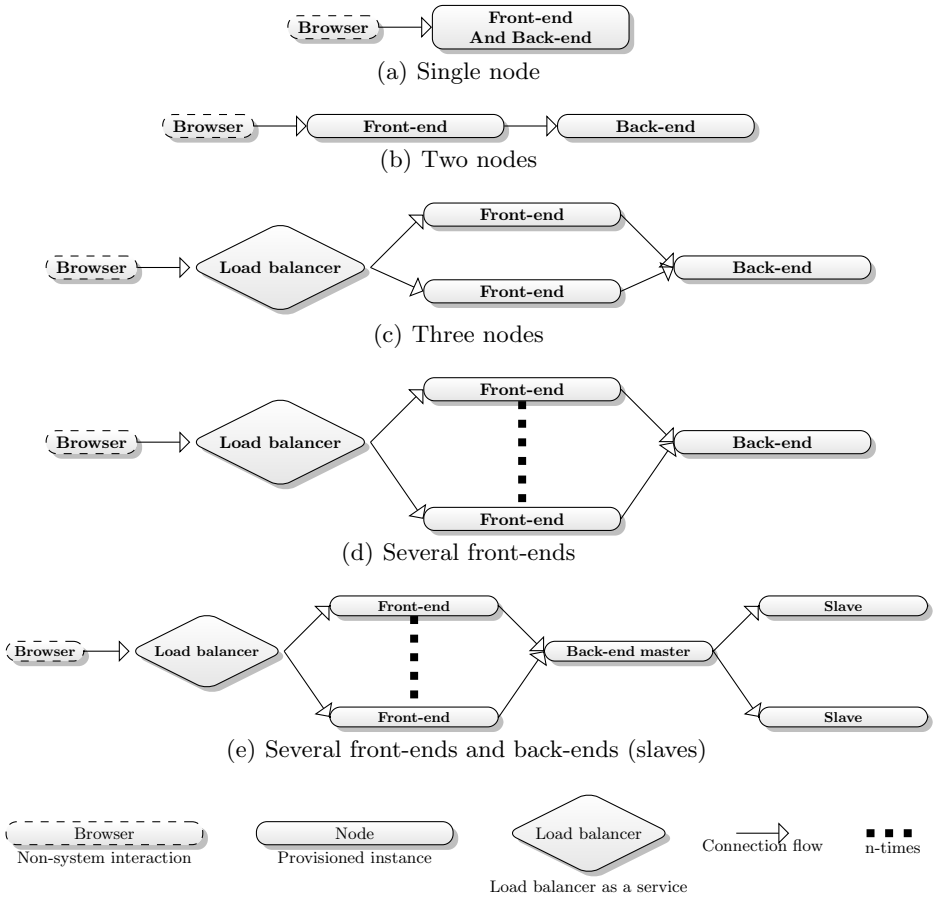

Fig. 1. Different architectural ways to provision nodes (topologies).

built from their Java APIs, while Rackspace [12] had no tools beside the API language bindings, thus we had to operate against the command-line tools and public APIs. As this emphasises the complexity even further it also stresses engineering capabilities of individuals executing the tasks to a higher technical level.

– **Multi-cloud**: Once we were 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 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[3]. Or availability decreases and results in decrease of service up-time 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 enhance quality.
- **Robustness**: There were several ways the scripts could fail and most errors were ignored. Transactional behaviours were non-existent.
- **Run-time dependency**: The scripts were developed to fulfil a complete deployment, and to do this it proved important to temporally save run-time specific meta-data. This was crucial data needed to connect front-end nodes with the back-end node.

Envision: Towards a CloudML environment. Our envision is to tackle these challenges by applying a model-driven approach supported by modern technologies. Our objective is to create a common model for nodes as a platform-independent model [4] to justify *multi-cloud* 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. We envision a tool 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 meta-data 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.

3 Contribution

We have developed a metamodel that describe CloudML as a DSL for cloud provisioning. It addresses the previously identified challenges, as summarised in Tab. 1. We provide a class-diagram representation of the CloudML meta-model in Fig. 2.

[3] For example, Google decided in 2011 to change the pricing policies associated to the GoogleAppEngine cloud service. All the applications that relied on the service had basically two options: *(i)* pay the new price or *(ii)* move to another cloud-vendor. Due to vendor lock-in, the second option often implied to re-implement the application.

Challenge	Addressed by
Complexity	One single entry point to multiple providers. Utilizing existing framework. Platform-independent model approach used to discuss, edit and design topologies for propagation.
Multicloud	Utilizing existing framework designed to interface several providers.
Reproducibility	Lexical model-based templates. Models can be reused to multiply a setup without former knowledge of the system.
Shareable	Lexical model-based templates. Textual files that can be shared through mediums such as e-mail or version control systems such as Subversion or Git.
Robustness	Utilizing existing framework and solid technologies.
Metadata dependency	*Models@run.time*. Models that reflect the provisioning models and updates asynchronously.

Table 1. CloudML: Challenges addressed.

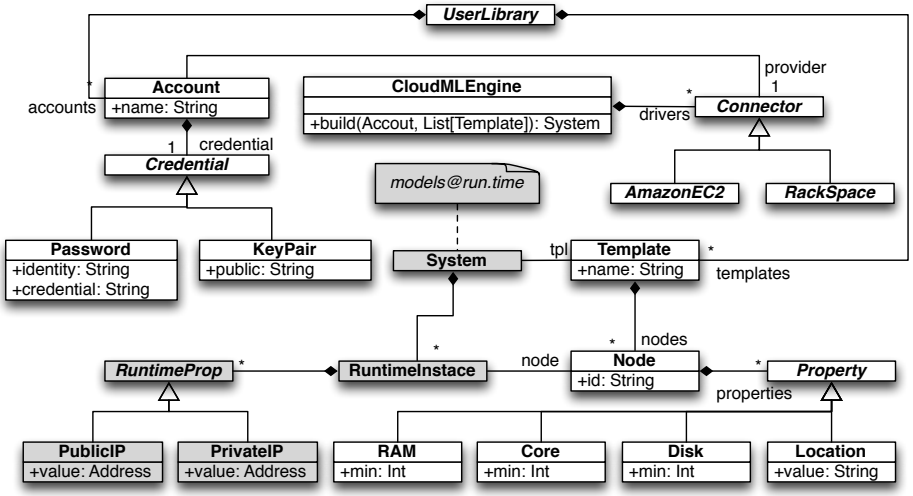

Fig. 2. Architecture of CloudML

Illustrative Scenario. CloudML is introduced using a scenario where an end-user (named Alice) is provisioning the *BankManager* to Amazon Web Services *Elastic Compute Cloud* (EC2) using the topology shown in Fig. 1(c). It is compulsory that she possesses an EC2 account in advance of the scenario. She will retrieve security credentials for account and associate them with `Password` in Fig. 2. `Credential` is used to authenticate the user to supported providers through `Connector`. The next step for Alice is to model the appropriate `Template` consisting of three `Nodes`. The characteristics Alice choose for `Node Properties` are fitted for the chosen topology with more computational power for front-end `Nodes` by increasing amount of `Cores`, and increased `Disk` for back-end `Node`. All

`Properties` are optional and thus Alice does not have to define them all. With this model 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 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.

Implementation. CloudML is implemented as a proof of concept framework [6] (from here known as *cloudml-engine*). Because of Javas popularity we wrote cloudml-engine 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*.

We represent in Fig. 3 the provisioning process implement in the CloudML engine, using a sequence diagram. Provisioning nodes is by its nature an asynchronous action that can take minutes to execute, therefore we relied on the actors model [8] using Scala actors. With this asynchronous solution we got concurrent communication with nodes under provisioning. We extended the model by adding a callback-based pattern allowing each node to provide information on property and status changes. Developers exploring our 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. We have divided the terms of 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 *state* 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.

4 Validation & Experiments

To start the validation of the approach and the implemented tool, we provisioned the *BankManager* application using different topologies in Fig[1(a), 1(c)]. The implementation uses *JavaScript Object Notation* (JSON) to define templates as a human readable serialisation mechanism. The lexical representation of Fig. 1(a)

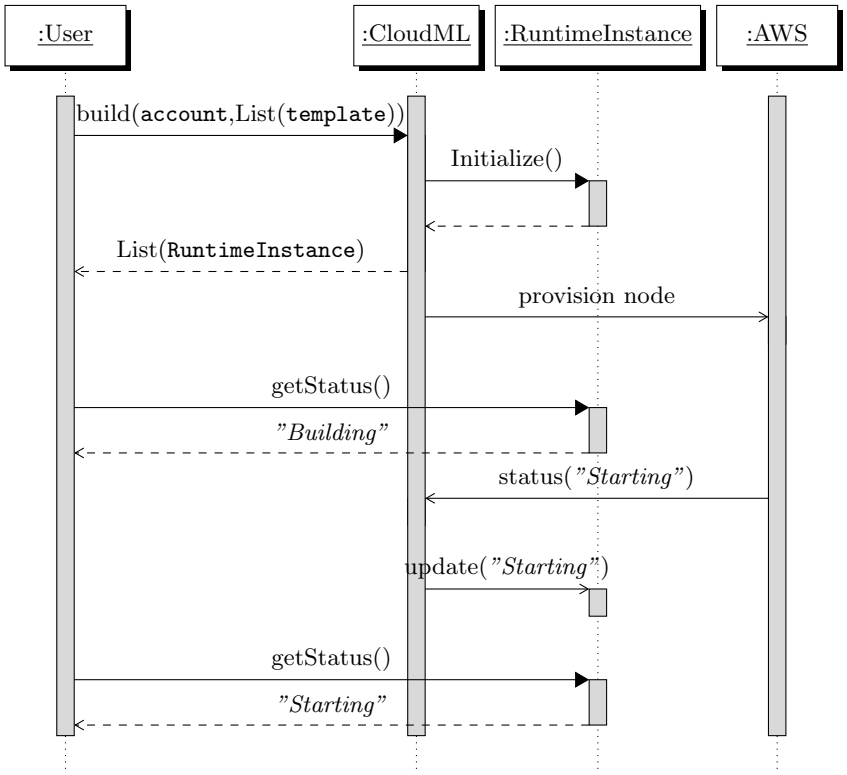

Fig. 3. CloudML asynchronous provisionning process (Sequence diagram).

can be seen in Listing. 1.1. The whole text represents the `Template` of Fig. 2 and consequently "nodes" is a list of `Node` from the model. JSON is textual which makes it *shareable* as files. We implemented it so once such a file is created it can be reused (*reproducibility*) on any supported provider (*multi-cloud*).

```
1 { "nodes": [
2     { "name": "testnode" }
3   ]
4 }
```

Listing 1.1. One single node (topology: Fig. 1(a))

The topology described in Fig. 1(c) is represented in Listing. 1.2, the main difference from Listing. 1.1 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. The key idea is that the meta-model is extensible, and can support new properties in the language thanks to the extension of the `Property` class.

```
{
  "nodes": [
    { "name": "frontend1",
      "minRam": 512,
      "minCores": 2 },
    { "name":
      "frontend2",
      "minRam": 512,
      "minCores": 2 },
    { "name": "backend",
      "minDisk": 100 }
  ]
}
```

Listing 1.2. Three nodes

5 Related Works

There already exists scientific research projects and technologies which have similarities to CloudML both in idea and implementation. First we will present three scientific research projects and their solutions, then we will introduce pure technological approaches. We also discuss how our approach differ from theres.

One project that bears relations to ours is mOSAIC [11] 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.)"* [10]. The strongest similarities to CloudML are *(i)*multicloud with their API [10], *(ii)*metadata dependencies since they support full deployment and *(iii)*the robustness through fault-tolerance. What mOSAIC is lacking compared to CloudML is model-based approach including *M@RT*. Reservoir [13] is another project that also aim at *(iv)*multicloud. The other goal 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 our goals. CloudML stands out from Reservoir in the same way as mOSAIC. Vega framework [7] is a deployment framework aiming at full cloud deployment of multi-tier topologies, they also follow a 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.

6 Conclusions & Perspectives

In this paper, we presented the initial version of CloudML, a cloud modelling language used to model the resources that a given application can require from existing clouds. The approach is defined as a meta-model, associated to a reference implementation using the Scala language. This reference implementations is connected to several cloud providers, and we described preliminary experiments that address major cloud providers: Amazon EC2 and Rackspace.

The first perspective of this work is to emphasise its validation. In the context of the REMICS project, our partners provide us several case studies (tourism, banking, scientific computation) that require the provisioning of resources in the clouds. We are also interested in refining the set of properties available in the CloudML meta-model to properly categorise the available resources. For now, we focus our effort on computational power, but other dimension of clouds (*e.g.*, data location, costs) should be taken into account at the CloudML level. The next challenge to be tackled by the CloudML environment is to model the complete deployment of cloud-applications. By coupling the current version of CloudML with an architecture description language, it will be possible to model the needed resources and the deployment plan to be followed to support the automated deployment of the application.

References

1. Amazon: Amazon web services (2012), http://aws.amazon.com
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the Clouds: A Berkeley View of Cloud Computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley (Feb 2009), http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html
3. Aßmann, U., Bencomo, N., Cheng, B.H.C., France, R.B.: Models@run.time (dagstuhl seminar 11481). Dagstuhl Reports 1(11), 91–123 (2011)
4. Bézivin, J., Gerbé, O.: Towards a precise definition of the omg/mda framework. In: Proceedings of the 16th IEEE international conference on Automated software engineering. pp. 273–. ASE '01, IEEE Computer Society, Washington, DC, USA (2001), http://dl.acm.org/citation.cfm?id=872023.872565
5. Brandtzæg, E.: Bank manager (2012), https://github.com/eirikb/grails-bank-example
6. Brandtzæg, E.: cloudml-engine (2012), https://github.com/eirikb/cloudml-engine
7. Chieu, T., Karve, A., Mohindra, A., Segal, A.: Simplifying solution deployment on a Cloud through composite appliances. In: Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on. pp. 1 –5 (april 2010)
8. Haller, P., Odersky, M.: Actors that unify threads and events. In: Proceedings of the 9th international conference on Coordination models and languages. pp. 171–190. COORDINATION'07, Springer-Verlag, Berlin, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=1764606.1764620

9. Mohagheghi, P., Sæther, T.: Software Engineering Challenges for Migration to the Service Cloud Paradigm: Ongoing Work in the REMICS Project. In: SERVICES. pp. 507–514. IEEE Computer Society (2011)
10. Petcu, D., Crăciun, C., Neagul, M., Panica, S., Di Martino, B., Venticinque, S., Rak, M., Aversa, R.: Architecturing a Sky Computing Platform. In: Cezon, M., Wolfsthal, Y. (eds.) Towards a Service-Based Internet. ServiceWave 2010 Workshops, Lecture Notes in Computer Science, vol. 6569, pp. 1–13. Springer Berlin / Heidelberg (2011)
11. Petcu, D., Macariu, G., Panica, S., Crăciun, C.: Portable Cloud applications-From theory to practice. Future Generation Computer Systems (0), – (2012), http://www.sciencedirect.com/science/article/pii/S0167739X12000210
12. Rackspace: Rackspace cloud (2012), http://www.rackspace.com/cloud
13. Rochwerger, B., Breitgand, D., Levy, E., Galis, A., Nagin, K., Llorente, I.M., Montero, R., Wolfsthal, Y., Elmroth, E., Caceres, J., Ben-Yehuda, M., Emmerich, W., Galan, F.: The Reservoir model and architecture for open federated cloud computing. IBM Journal of Research and Development 53(4), 4:1 –4:11 (july 2009)
14. Varia, J.: Architecting for the Cloud : Best Practices. Compute 1(January), 1–23 (2010)