Model-based realization of provisioning in the cloud*

Eirik Brandztæg[1,2] and Sébastien Mosser[1] and Parastoo Mohagheghi1

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

Built: February 20, 2012

Abstract. ∼150 words expected. Will also be defined after the brainstorming phase. Must mention *(i)*the problem, *(ii)*the actual contribution and *(iii)*the obtained results.

1 Introduction

I'll write the introduction afterwards, when the content of the paper will be fixed.

- Cloud–computing research field [2] Map with section 2 from mde
- Model–driven engineering applied to the cloud

2 Challenges in the cloud

To recognize challenges when doing cloud provisioning we used an example application [3]. The application (from here known as BankManager) is a featureless bank manager system written in Grails [12], it supports creating users and bank accounts, 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 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. 1(a). In Fig. 1(b) front-end is separated from back-end, this introduces the flexibility of increasing computation power on the front-end node while spawning more storage on the back-end. For application 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 relational model based database, but this can solved on a database level with master and

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

slaves **source about master-slaves?** (Fig. 1(e)), although this is beside the scope of this article but we wanted to show the possibility. We used bash-scripts to prototype full deployments of BankManager against Amazon AWS [1] and Rackspace [10] with a design with three nodes as shown in Fig. 1(c). From this prototype it became clear that there were several challenges that CloudML had to tackle:

- **Complexity**: The first challenge we encountered was to simply authenticate and communicate with the cloud. The two providers we tested had different approaches, AWS [1] had command-line tools built from their Java APIs, while Rackspace [10] had no tools beside the API language bindings. For Rackspace we decided to use the HTTP data transferring tool *curl* and communicate directly with their RESTful API
- **Technical competence**: Because of the complexity encountered in our scenario the second challenge was to understand how to operate against the command-line tools and public APIs. As this emphasizes the complexity even further it also stresses engineering capabilities of individuals executing the tasks to a higher technical level
- **Multicloud**: Once 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. The first distinction was in the concrete differences between API callouts, even the node property structures and names were different
- **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 very difficult to share topologies between coworkers, as this was important to exchange ideas
- **Robustness**: There were several ways the scripts could fail and most errors were ignored
- **Metadata dependency**: The scripts were developed to fulfill a complete deployment, and to do this it proved important to temporally save run-time specific metadata. This was crucial data needed to connect front-end nodes with the back-end node.

Our envision is to tackle these challenges by applying model-driven approaches and modern technologies. One example is to create a common model for nodes to justify ***multicloud*** *differences and at the same time base this on a human readable lexical format to resolve* ***reproducibility*** *and make it* ***shareable****. .*

3 Contribution

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 based on template files representing topologies

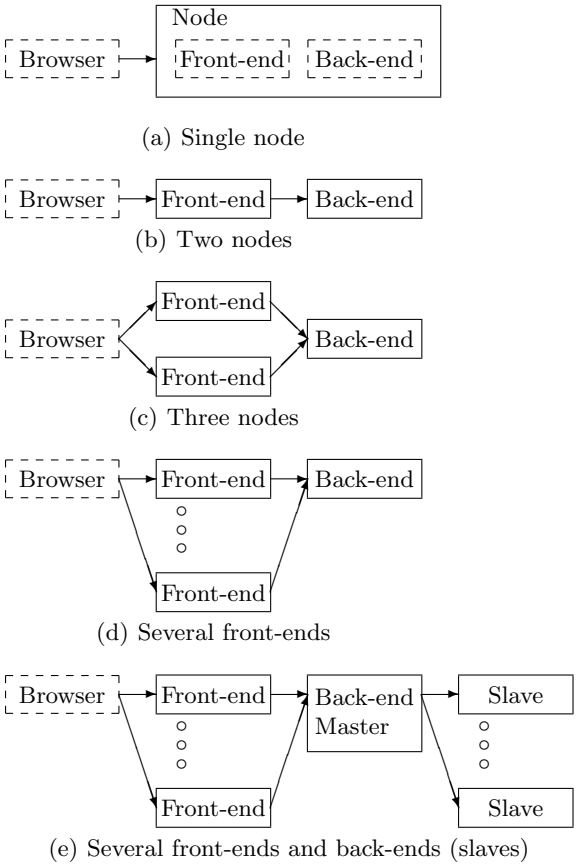

Fig. 1. Different architectural ways to provision nodes

of instances in the cloud. The same files should be usable on other providers, and alternating the next deployment stack should be effortless. Instance types are selected based on properties within the template, and additional resources are applied when necessary and available. While the tool performs provisioning metadata of nodes is available. In the event of a template being inconsistent with possibilities provided by a specific provider this error will be informed to the user and provision will halt.

CloudML is a lexical model-based language for cloud provisioning. The language has two main models shown in Fig. 3, *Account* and *Template*. The *Account* model is used for provider credentials to authenticate against a given provider such as Amazon or Rackspace. This model can be stored, shared and reused for the same provider multiple times. *Template* model on the other hand is used to define sets of resources **source** such as nodes or load balancers, also known as a *stack* **source**. This model can also be stored, shared and reused like the *Account* model, but it does not alternate between providers (*Accounts*).

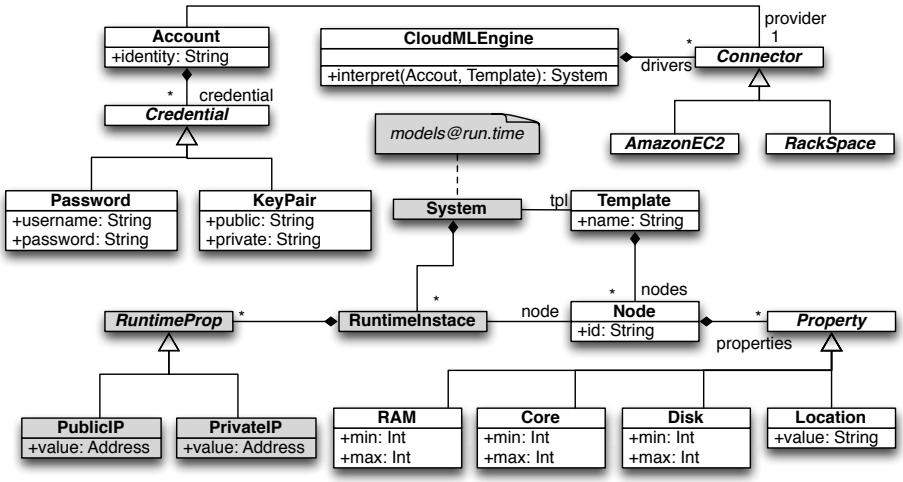

Fig. 2. Architecture of CloudML

Implementation. CloudML is implemented as a proof of concept framework [4] (from here known as cloudml-engine*). According to Ilham [7] "java is gaining popularity in software development." and then continues to state that "it is widely used in network computing and embedded systems". Because of Javas popularity we wrote cloudml-engine in a JVM language with Maven as build tool. Cloumlengine 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*** *.*

Challenge	Tackeled	Resolved by
Complexity	Yes	One single entry point to 24 providers
Technical competence	Yes	Model-based approach
Multicloud	Yes	Utilizing existing framework designed to interface 24 providers
Reproducibility	Yes	Lexical model-based templates
Shareable	Yes	Lexical model-based templates
Robustness	Yes	Technology choices? **More subtle**
Metadata dependency	Yes	Models@runtime

Table 1. Challenges tackeled

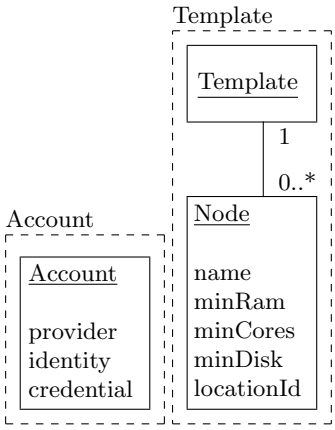

Fig. 3. Model of Account and Template

4 Validation & Experiments

To validate how CloudML tackled the challenges from Sec. 2 we provisioned the BankManager application using different topologies Fig[1(a), 1(c)]. The implementation uses JavaScript Object Notation (JSON) to define templates (Fig. 3), so the lexical representation of Fig. 1(a) can be seen in Listing. 1.1. The whole text represents the *Tempalate* of Fig. 3 and consequently "nodes" is an array of Node from the model. The JSON is lexical **source: and human readable?** which makes it **shareable** as files. We implemented it so once such a file is created it can be reused (**reproducibility**) on any given provider (**multicloud**).

```
{
  "nodes": [ { "name": "testnode" } ]
}
```

Listing 1.1. One single node

More advanced topology such as Fig. 1(c) is represented in Listing. 1.2, the main difference from Listing. 1.1 is that there are more nodes with 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 got more disk.

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

Listing 1.2. Three nodes

Provisioning nodes is by its nature an asynchronous action that can take minutes to execute, therefore we relied on the actors model [6] using Scala actors. With this asynchronous solution we get 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 actors are provisioned. We have divided the terms of a node before and under provisioning, the essential is to introduce models-at-runtime 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, and the task of provisioning is concluded. But when provisioning **here be talk about deployment?**.

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 some scientific research projects and their solutions, then we will introduce some pure technological approaches.

One project that bears relations to ours is mOSIAC [8] which aims at not only provisioning in the cloud, but deployment as well. The strongest similarities to CloudML are *(i)*model-based approach along with *(ii)*models-at-runtime and *(iii)*multicloud, it also support more advanced features for deployment, all this through their API [9]. RESERVOIR [11] is another project that also aim at *(i)*multicloud. The other goals of this project is to leverage scalability in single providers and support built-in business service management (BSM), important topics but not directly related to our goals. Vega framework[5] is a deployment framework aiming at full cloud deployments of multi-tier topologies, they also follow a *(i)*model-based approach.

There are also some distinct technologies that bear similarities to CloudML. Some of these are *(i)*model-driven such as AWS CloudFormation and CA Applogic. Others are plain APIs supporting *(ii)*multicloud such as libcloud, jclouds and DeltaCloud. The last group are projects that aim specifically at deployment, making IaaS work as 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 solve, but since some of these projects specifically solve some 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

I'll write the conclusions afterwards.

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. Brandtzæg, E.: Bank manager (2012), https://github.com/eirikb/grails-bank-example
4. Brandtzæg, E.: cloudml-engine (2012), https://github.com/eirikb/cloudml-engine
5. 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)

6. 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
7. Ilham, A., Murakami, K.: Evaluation and optimization of java object ordering schemes. In: Electrical Engineering and Informatics (ICEEI), 2011 International Conference on. pp. 1 –6 (july 2011)
8. mOSAIC: mosaic (2012), http://www.mosaic-cloud.eu/
9. Petcu, D., Crciun, 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)
10. Rackspace: Rackspace cloud (2012), http://www.rackspace.com/cloud/
11. 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)
12. SpringSource: Grails (2012), http://grails.org