Model-based realization of provisioning in the cloud*

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

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

Built: February 23, 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 [3] Map with section 2 from mde
- Model–driven engineering applied to the cloud

"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." [1]

2 Challenges in the cloud

To recognize challenges when doing cloud provisioning we used an example application [4]. The application (from here known as *BankManager*) is a prototypical bank manager system which support 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

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

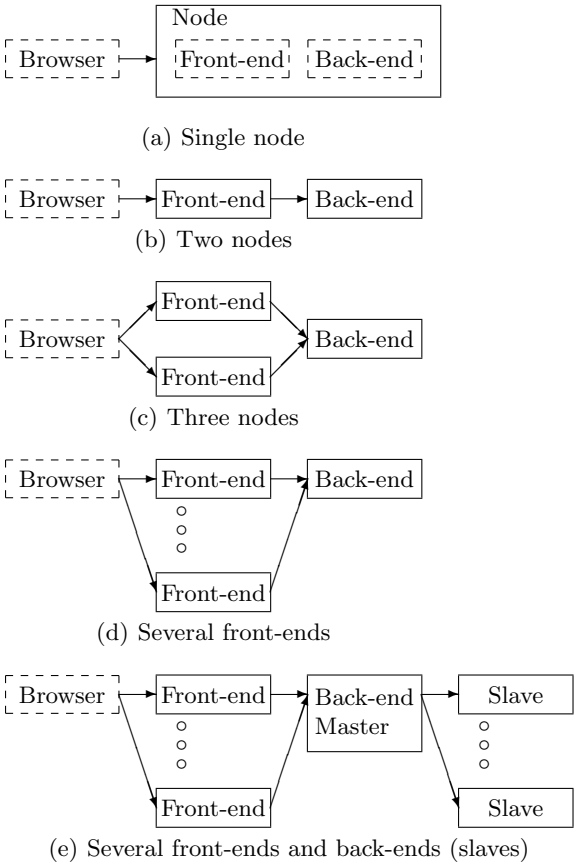

Fig. 1. Different architectural ways to provision nodes

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 slaves (Fig. 1(e)), although this is beside the scope of this article but we want to visualize the possibility. We used bash-scripts to prototype full deployments of *BankManager* against *Amazon Web Services* (AWS) [2] and Rackspace [10] 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: **Write about lambda user?**

- **Complexity**: The first challenge we encountered was to simply authenticate and communicate with the cloud. The two providers we tested had different approaches, AWS [2] 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, thus we had 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.

Envision. 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

Challenge	Addressed by
Complexity	One single entry point to multiple providers. Utilizing existing framework
Technical competence	Model-based approach. Models used to discuss, edit and design topologies for propagation
Multicloud	Utilizing existing framework designed to interface 24 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 Subersion or Git
Robustness	Utilizing existing framework and solid technologies
Metadata dependency	Models@run.time. Models that reflect the provisioning models but updates asynchronously

Table 1. Challenges addressed

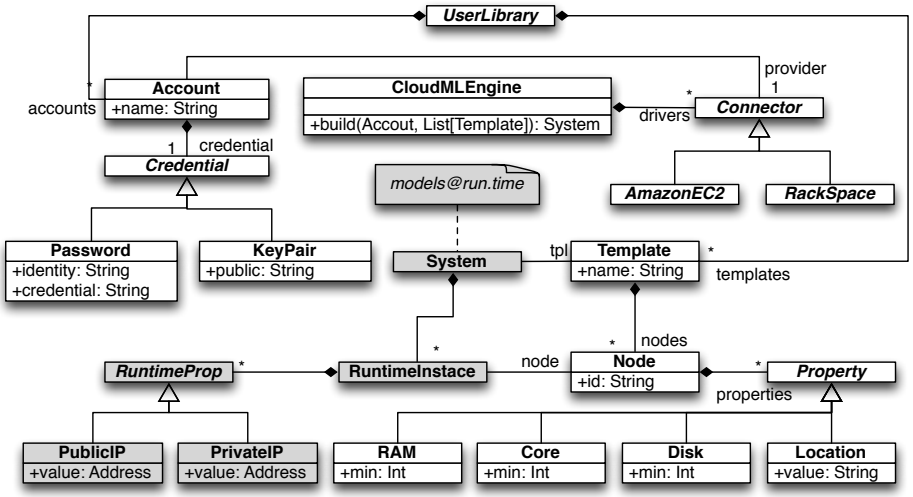

Fig. 2. Architecture of CloudML

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 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.

We describe the meta model in Fig. 2 and introduce CloudML by using a scenario where "Alice" is provisioning *BankManager* to AWS *Elastic Compute Cloud* (EC2) using the topology shown in Fig. 1(c). It is compulsory that she possesses an AWS account in advance of the scenario. She will retrieve security credentials for the 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 build 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 be define them all. With the 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 according to *models@run.time* 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 meta data is gathered Alice can start the deployment, CloudML has then completed its scoped task of provisioning. But then Alice decide to use another provider, either as replacement or complement to her current setup. Reasons for such actions can be availability, financial or support. All she needs to do is change the provider name in `Account` and call `build` on `CloudMLEngine` again, this will result in an identical topological setup on the new provider.

Implementation. CloudML is implemented as a proof of concept framework [5] (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*

4 Validation & Experiments

To validate how CloudML addressed the challenges from Table. 1 we provisioned the *BankManager* application using different topologies Fig[1(a), 1(c)].

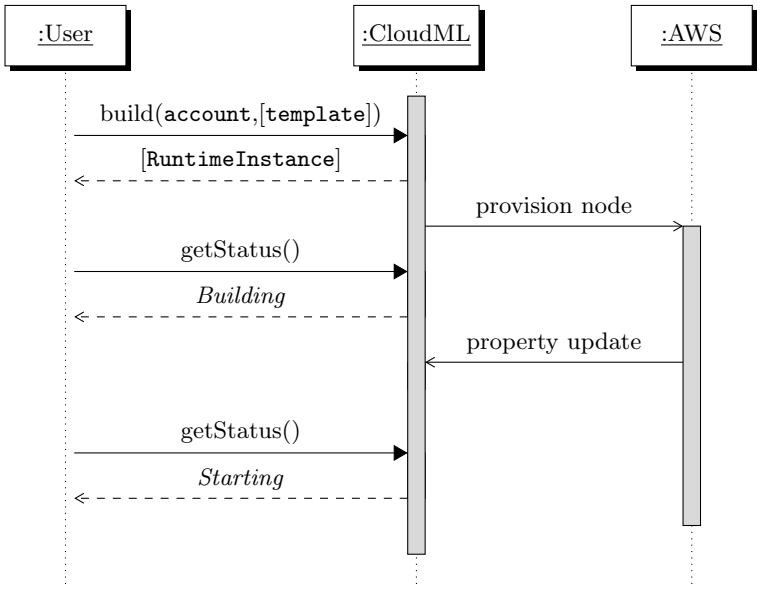

Fig. 3. Sequence diagram of CloudML

The implementation uses *JavaScript Object Notation* (JSON) to define templates as a human readable serialization mechanism. The lexical representation of Fig. 1(a) can be seen in Listing. 1.1. The whole text represents the `Template` of Fig. 2 and consequently "nodes" is an array of `Node` from the model. The JSON is lexical 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 (*multicloud*).

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

Listing 1.1. One single node

The topology described in 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.

```
1 {
2   "nodes": [
3     { "name": "frontend1", "minRam": 512, "minCores": 2 },
4     { "name": "frontend2", "minRam": 512, "minCores": 2 },
5     { "name": "backend", "minDisk": 100 }
6   ]
7 }
```

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 [7] 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@run.time* 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.

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.

One project that bears relations to ours is mOSAIC [9] 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.)"* [8]. The strongest similarities to CloudML are *(i)*multicloud with their API [8], *(ii)*metadata dependencies since they support full deployment and *(iii)*robustness through fault-tolerance. What mOSAIC is lacking compared to CloudML is model-based approach including *models@run.time*. 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. CloudML stands out from Reservoir in the same way as mOSAIC, but even more by relying on existing technologies to leverage robustness. Vega framework [6] is a deployment framework aiming at full cloud deployments of multi-tier topologies, they also follow a *(i)*model-based approach. The main difference between CloudML and Vega is support of multicloud provisioning.

There are also distinct technologies that bear similarities to CloudML. Of these 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 solve, but since these projects specifically 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

I'll write the conclusions afterwards.

References

1. Architecting for the Cloud : Best Practices 1
2. Amazon: Amazon web services (2012), http://aws.amazon.com/
3. 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
4. Brandtzæg, E.: Bank manager (2012), https://github.com/eirikb/grails-bank-example
5. Brandtzæg, E.: cloudml-engine (2012), https://github.com/eirikb/cloudml-engine
6. 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)
7. 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
8. 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)
9. Petcu, D., Macariu, G., Panica, S., Crciun, C.: Portable Cloud applications-From theory to practice. Future Generation Computer Systems (0), – (2012), http://www.sciencedirect.com/science/article/pii/S0167739X12000210
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)