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 28, 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

"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." [12]

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 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 applications performing heavy computations it can be beneficial to distribute the workload

---

(a) Single node

(b) Two nodes

(c) Three nodes

(d) Several front-ends

(e) Several front-ends and back-ends (slaves)

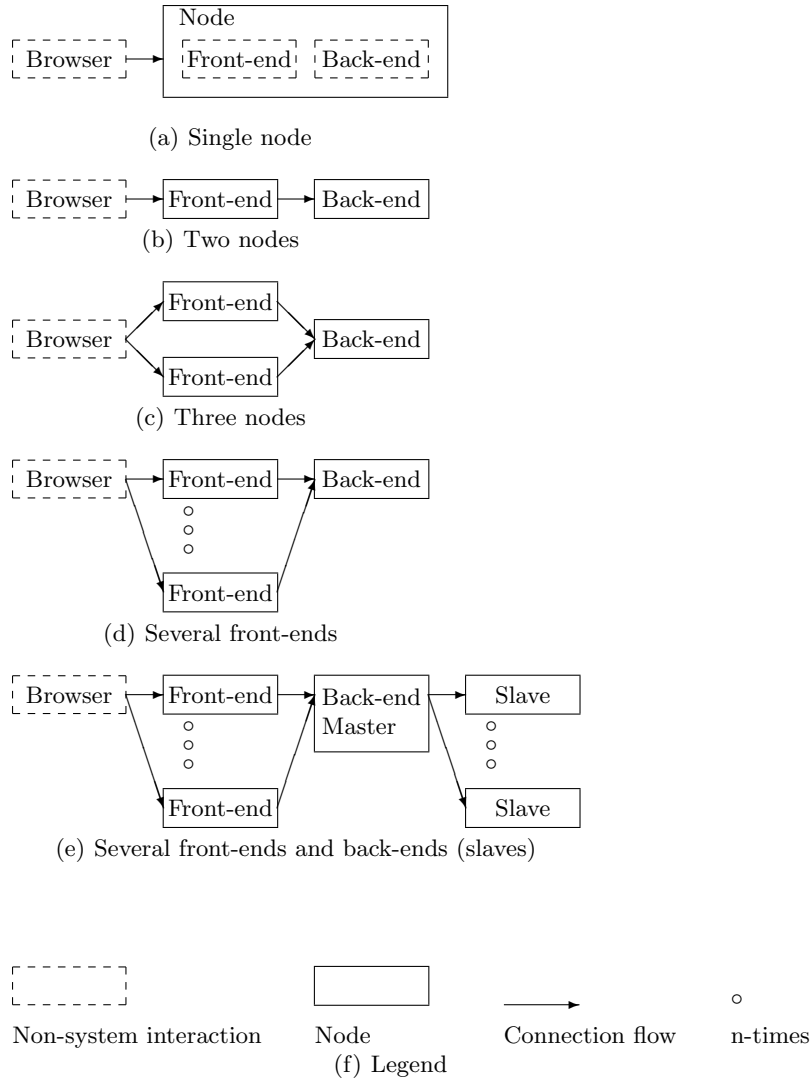|			
Non-system interaction	Node	Connection flow	n-times

(f) Legend

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, 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) [1] 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:

– **Complexity**: The first challenge we encountered was to simply authenticate and communicate with the cloud. The two providers we tested against 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, 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. 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. Or availability decreases and results in decrease of service uptime damaging revenue.
– **Reproducibility**: The scripts provisioned nodes based on command-line arguments and did not persist the designed topology in any way, this made topologies cumbersome to reproduce.
– **Shareable**: Since the scripts did not remember a given setup it was impossible to share topologies "as is" between coworkers. It is important that topologies can be shared because direct input from individuals with different areas of competence can increase quality.
– **Robustness**: There were several ways the scripts could fail and most errors were ignored. Transactional behaviors were non-existent.
– **Run-time dependency**: The scripts were developed to fulfill a complete deployment, and to do this it proved important to temporally save run-time specific metadata. This was crucial data needed to connect front-end nodes with the back-end node.

Envision. 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 [7] to justify *multicloud* differences and at the same time base this on a human readable lexical format to resolve *reproducibility* and make it *shareable.*

The concept and principle of CloudML is to be an easier and more reliable path into cloud computing for IT-driven businesses of variable sizes. 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 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.

3 Contribution

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 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 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. Challenges addressed.

We describe the meta model in FIG. 2 and introduce CloudML by using a scenario where "Alice" is provisioning the *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 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
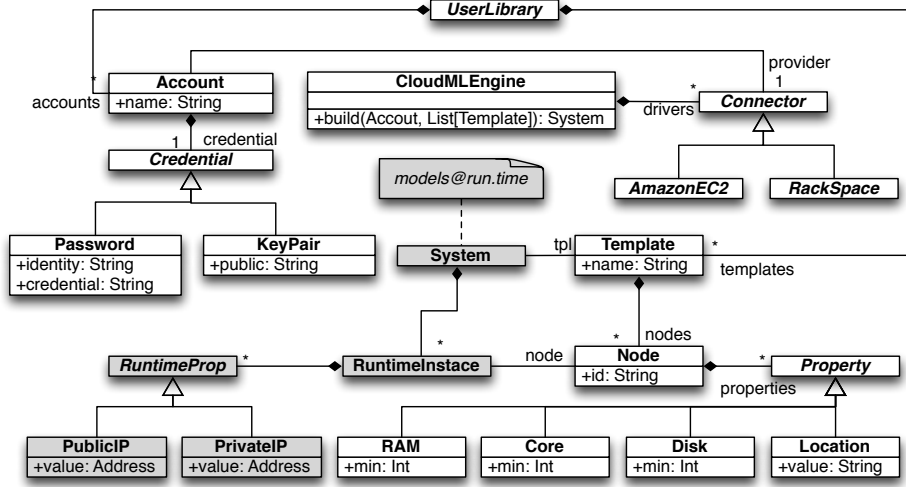
Fig. 2. Architecture of CloudML

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 [4] (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*.

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 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.

Full deployment is planed for next version of CloudML.
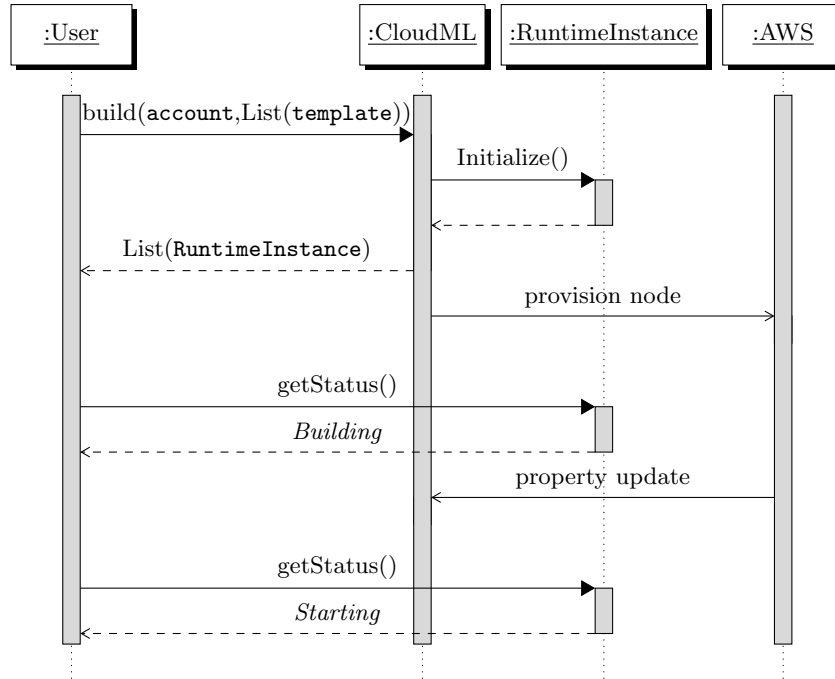
4 Validation & Experiments

Fig. 3. Sequence diagram of CloudML.

To validate how CloudML addressed the challenges from TABLE. 1 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 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 a list of `Node` from the model. The 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 (*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 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.

```
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

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 *M@RT*. Reservoir [11] is another project that also aim at *(iv)*multicloud. The other goals of this project is to leverage scalability in single providers and support built-in *Business Service Management* (BSM), important topics but not directly related to our goals. CloudML stands out from Reservoir in the same way as mOSAIC. Vega framework [5] is a deployment framework aiming at full cloud deployments of multi-tier topologies, they also follow a *(v)*model-based approach. The main difference between CloudML and Vega are support of multicloud provisioning.

There are also distinct technologies that bear similarities to CloudML. None of AWS CloudFormation and CA Applogic are *(i)*model-driven. Others are plain APIs supporting *(ii)*multicloud such as libcloud, jclouds and DeltaCloud. The last group are projects that aim specifically at deployment, making *Infrastructure-as-a-Service* (IaaS) work as *Platform-as-a-Service* (PaaS) like AWS Beanstalk and SimpleCloud. The downside about the technical projects are their inability to solve all of the challenges that CloudML aims to address, but since these