WIND OF CHANGE: FROM VENDOR LOCK-IN TO THE META CLOUD

BENJAMIN SATZGER, WALDEMAR HUMMER, CHRISTIAN INZINGER, PHILIPP LEITNER, AND SCHAHRAM DUSTDAR

ABSTRACT. The emergence of more and more cloud offerings from a multitude of service providers calls for a meta cloud, which smoothens the jagged cloud landscape. We discuss our proposal for such a meta cloud, and explain how it solves the lock-in problems that current users of public and hybrid clouds face.

The cloud computing paradigm has found widespread adoption throughout the last years. The reason for the success of cloud computing is the possibility to use services on-demand with a pay as you go pricing model, which proved to be convenient in many respects. Because of low costs and high flexibility, migrating to the cloud is indeed compelling. Despite the obvious advantages of cloud computing, many companies hesitate to “move into the cloud”, mainly because of concerns related to availability of service, data lock-in, and legal uncertainties [1]. Lock-in is particularly problematic for the following reasons. Firstly, even though availability of public clouds is generally high, eventual outages still occur [3]. If this is the case, businesses locked into such a cloud are essentially at a standstill until the cloud is back online. Secondly, public cloud providers generally do not guarantee particular service level agreements [4], i.e., businesses locked into a cloud have no guarantees that this cloud will continue to provide the required Quality of Service (QoS) tomorrow. Thirdly, the terms of service of most public cloud providers allow the provider to unilaterally change pricing of their service at any time. Hence, a business locked in a cloud has no mid- or long-term control over their own IT costs.

At the core of all of these problems, we can identify a need for businesses to permanently monitor the cloud they are using, and to be able to rapidly “change horses”, i.e., migrate to a different cloud if monitoring discovers problems or estimations foresee issues in the future. However, migration is currently far from trivial. A plethora of cloud providers is flooding the market with a confusing body of services, such as compute services e.g., Amazon Elastic Compute Cloud (EC2) and VMware vCloud, or key-value stores, e.g., Amazon Simple Storage Service (S3). Evidently, some of these services are conceptually comparable to each other, others are vastly different, but all of them are, ultimately, technically incompatible and follow no standards but
their own. To further complicate the situation, many companies are not (only) building on public clouds for their cloud computing needs, but combine public offerings with their own private cloud, leading to so-called hybrid cloud setups [5].

In this paper we introduce the concept of a meta cloud consisting of a combination of design time and runtime components. Meta cloud abstracts away from technical incompatibilities of existing offerings, thus mitigating vendor lock-in. It helps to find the right set of cloud services for a particular use case, and supports an applications initial deployment and runtime migration.

A Typical Cloud Computing Use Case

The BBC has recently reported\(^1\) on their technology strategy for operating the Web portal of the 2012 Olympic Games. An event of this dimension requires an enormously efficient and reliable infrastructure and the cloud computing paradigm provides flexibility and elasticity that is needed for such a scenario. It allows to handle short-term usage spikes without the need to have respective dedicated resources available all the time. The problem is, however, that once an application has been developed based on cloud services of one particular provider using its specific API it is bound to the provider; deploying the application on another cloud would usually require to completely redesign and rewrite it. This vendor lock-in leads to strong dependence on the cloud service operator. In the example of the sports portal, besides the ability to scale applications up and down by dynamically allocating/releasing resources, additional aspects such as resource costs and regional communication bandwidth and latency have to be considered.

Let us assume the sports betting portal application is based on a load balancer that forwards HTTP requests to a number of computing nodes hosting a web application, which allows users to submit a bet. Bet records are put into a message queue and subsequently stored into a relational database. Using Amazon AWS cloud services, this scenario can be realized using EC2 to host applications, SQS as cloud message queue, and finally RDS as database system. Instead of being bound to one cloud operator, the betting application should be hosted on the optimal cloud environment.

In order to leverage a more diverse cloud landscape, to support flexibility, and to avoid vendor lock-in, the following main goals need to be achieved by the meta cloud.

1. Find the optimal combination of cloud services for a certain application with regard to QoS for the users and price for hosting.
2. Develop a cloud-based application once, then run it anywhere, including support for runtime migration.

\(^1\)http://www.bbc.co.uk/blogs/bbcinternet/2012/04/sports_dynamic_semantic.html
Lately, the meta cloud idea has already received some attention and several approaches try to tackle at least parts of the problem. In the next section we will briefly discuss currently available solutions.

**The Current Weather in the (Meta) Cloud**

Firstly, **standardized programming APIs** are required to enable the development of cloud-neutral applications, which are not hardwired to any single provider or cloud service. Cloud provider abstraction libraries such as libcloud\(^2\), fog\(^3\), and jclouds\(^4\) provide unified APIs for accessing cloud products of different vendors. Using these libraries, developers are relieved of technological vendor lock-in, as they can switch cloud providers for their applications with relatively low overhead.

As a second ingredient, the meta cloud makes use of **resource templates** to define concrete features that the application requires from the cloud. For instance, an application needs to be able to specify that it requires a given number of computing resources, internet access, and database storage. Some current tools and initiatives (e.g., Amazon’s CloudFormation\(^5\), or the upcoming TOSCA specification\(^6\)) are working towards similar goals, and can be adapted to provide these required features for the meta cloud.

In addition to resource templates, the automated formation and provisioning of cloud applications, as envisioned in the meta cloud, also depends on sophisticated features to actually deploy and install applications automatically. Predictable and controlled **application deployment** is a central issue for cost-effective and efficient deployments in the cloud, and even more so for the meta cloud. Several application provisioning solutions exist today, enabling the declarative specification of deployment artifacts and dependencies to allow for repeatable and managed resource provisioning. Notable examples include Opscode Chef\(^7\), Puppet\(^8\), and juju\(^9\).

At runtime, an important aspect of the meta cloud is **application monitoring**. Monitoring enables meta cloud to decide whether new instance of the application should be provisioned, or whether parts of the application need to be migrated. Currently, various vendors are providing tools for cloud monitoring, ranging from system-level monitoring (e.g., CPU, bandwidth), application-level monitoring (e.g.,

---

\(^2\)http://libcloud.apache.org/
\(^3\)http://fog.io/
\(^4\)http://www.jclouds.org/
\(^5\)http://aws.amazon.com/cloudformation/
\(^6\)https://www.oasis-open.org/committees/tosca/
\(^7\)http://www.opscode.com/chef/
\(^8\)http://puppetlabs.com/
\(^9\)http://juju.ubuntu.com
Amazon’s CloudWatch\(^{10}\) up to monitoring of service level agreements (e.g., monitis\(^{11}\)). However, the meta cloud requires more sophisticated monitoring techniques, and in particular approaches for making automated provisioning decision at runtime based on context and location of current users of the applications.

**Inside the Meta Cloud**

To some extent, the meta cloud can be realized based on a combination of existing tools and concepts, part of which are presented in the previous section. The main components of the meta cloud, depicted in Figure 1, are described in the following and their interplay is illustrated using the previously introduced sports betting portal example. The components of the meta cloud can be distinguished whether they are mainly important for cloud software engineers during development time or whether they perform tasks during runtime.

*Meta Cloud API.* The meta cloud API provides a unified programming interface to abstract from the differences among provider API implementations. For customers, the use of the meta cloud API prevents their application from being hard-wired to a specific cloud service offering. The meta cloud API can be built upon available cloud provider abstraction APIs, like libcloud, fog, and jclouds, as previously mentioned. While these mostly deal with key-value stores and compute services, in principle all

---

\(^{10}\)http://aws.amazon.com/cloudwatch/

\(^{11}\)http://portal.monitis.com/index.php/cloud-monitoring
services can be covered that are abstract enough to be offered by more than one provider and whose specific APIs do not differ too much, conceptually.

*Resource Templates.* Developers describe the cloud services necessary to run an application using resource templates. They allow to specify service types with additional properties, and a graph model is used to express the interrelation and functional dependencies between services. The meta cloud resource templates are created using a simple domain-specific language (DSL), allowing for the concise specification of required resources. The resource definitions is based on a hierarchical composition model, allowing for the creation of configurable and reusable template components, enabling developers and their teams to share and reuse common resource templates in different projects. Using the DSL, developers model their application components and their basic runtime requirements, such as (provider-independently normalized) CPU, memory, and I/O capacities, as well as dependencies and weighted communication relations between these components. The weighted component relations are used by the provisioning strategy to determine the optimal deployment configuration for the application. Moreover, resource templates allow for the definition of constraints based on costs, component proximity, and geographical distribution.

*Migration/Deployment Recipes.* Deployment recipes are an important ingredient for automation in the meta cloud infrastructure. The recipes allow for controlled deployment of the application including installation of packages, starting of required services, managing package and application parameters, and establishing links between related components. Automation tools such as Chef provide an extensive set of functionalities, which are directly integrated into the meta cloud environment. Migration recipes go one step further and describe how to migrate an application during runtime, e.g., migrating storage functionality from one service provider to another. Recipes only describe initial deployment and migration, the actual process is executed by the provisioning strategy and the meta cloud proxy using aforementioned automation tools.

*Meta Cloud Proxy.* The meta cloud provides proxy objects, which are deployed with the application and run on the provisioned cloud resources. They serve as mediator between the application and the cloud provider. These proxies expose the meta cloud API to the application, transform application requests into cloud provider specific requests, and forward them to the respective cloud services. The proxy provides means to execute deployment and migration recipes triggered by the meta cloud’s provisioning strategy. Moreover, proxy objects send QoS statistics to the resource monitoring component running within the meta cloud. The data are gained by intercepting calls of the application to the underlying cloud services and measuring their processing time, or by executing short benchmark programs. Applications can
also define and monitor custom QoS metrics that are sent to the resource monitoring component via the proxy object to enable advanced, application-specific management strategies. To avoid high load and computational bottlenecks, the communication between proxies and the meta cloud is kept at a minimum. Proxies do not run inside the meta cloud, and regular service calls from the application to the proxy are not routed through the meta cloud, either.

Resource Monitoring. The resource monitoring component is responsible for receiving data collected by meta cloud proxies about the resources they are using, on application’s request. These data are filtered and preprocessed, and then stored to the knowledge base for further processing. This helps to generate comprehensive QoS information of cloud service providers and the particular services they are providing, including response time, availability, and more service specific quality statements.

Provisioning Strategy. The main task of the provisioning strategy component is to match an application’s cloud service requirements to actual cloud service providers. It is able to find and rank cloud services based on data in the knowledge base. The initial deployment decision is based on the resource templates, specifying the resource requirements of an application, together with QoS and pricing information about service providers. The result is a ranked list of possible combinations of cloud services regarding expected QoS and costs. At runtime, the component is able to reason about whether migration of a resource to another resource provider is beneficial based on new insights into the application’s behavior and updated cloud provider QoS or pricing data. Reasoning about migrating additionally involves calculating migration costs. Decisions of the provisioning strategy result in executing customer defined deployment or migration scripts.

Knowledge Base. The knowledge base serves as store for data about cloud provider services, their pricing and QoS, and information necessary to estimate migration costs. Customer provided resource templates and migration/deployment recipes are stored in the knowledge base as well. Also, the knowledge base indicates which cloud providers are eligible for a certain customer. These usually comprise all providers the customer has an account with and providers that offer possibilities to create (sub) accounts on the fly. A number of different information sources contribute to the knowledge base: meta cloud proxies regularly send data about application behavior and cloud service QoS. Pricing and capabilities of cloud service providers may be either added manually or by crawling techniques able to get this information automatically, like [2].

A Meta Cloud Use Case. Let us come back to the previously introduced sports application use case. A meta cloud compliant variant of this application accesses cloud services using the meta cloud API, and does not directly talk to the cloud
provider specific service APIs. For the particular case this means the application
does not depend on Amazon’s EC2, SQS, or RDS service APIs, but on meta cloud’s
compute, message queue, and relational database service APIs.

For initial deployment the developer submits the application’s resource template to
the meta cloud. It specifies not only the three types of cloud services needed for run-
ing the sports application, but also their necessary properties and how they depend
on each other. For compute resources, for instance, CPU, RAM, and disk space can
be specified, according to terminology defined by the meta cloud resource template
DSL. Each resource can be named in the template, which allows for referencing dur-
ing deployment, runtime, and migration. The resource template specification should
also contain interdependencies, like the direct connection between the web service
compute instances and the message queue service.

The rich information provided by resource templates helps provisioning strategy
component to make profound decisions about cloud service ranking. The working
principle for initial deployment can be explained by web search analogy, in which
resource templates are queries, cloud service provider QoS and pricing information
represent indexed documents. Algorithmic aspects of the actual ranking are beyond
the scope of this article. If some resources in the resource graph are only loosely
coupled, then it is more likely that resources from different cloud providers may
be selected for a single application. In our use case, however, we assume that the
provisioning strategy ranks the respective Amazon cloud services first, and that the
customer follows this recommendation.

After the resources are determined the application together with an instance of
the meta cloud proxy is deployed, according to customer provided recipes. During
runtime, the meta cloud proxy mediates between the application components and
the Amazon cloud resources, and sends monitoring data to the resource monitoring
component running within the meta cloud.

Monitoring data is used to refine the application’s resource template and the
provider’s overall QoS values, both stored in the knowledge base. This updated in-
fomation is regularly checked by the provisioning strategy component, which might
trigger a migration. Frontend nodes could be migrated to other providers to place
them closer to the application’s users, for example. Another reason for a migration
can be updated pricing data. After a price cut by Rackspace services may migrate to
their cloud offerings. For making these decisions, potential migration costs regarding
time and money need to be taken into account by the provisioning strategy com-
ponent. The actual migration is performed based on customer provided migration
recipes.
Working on the meta cloud, we face the following main technical challenges. Data describing services of different cloud providers must be collected and processed, such that their QoS properties can be compared and ranked in a normalized, provider-independent fashion. While solutions for deployment in the cloud are relatively mature, migration of applications is not as well supported. Here it is particularly important to find the balance between migration facilities provided by the meta cloud and the application. Cloud-centric migration gives responsibility for most migration aspects to the meta cloud infrastructure, leading to issues with application-specific intricacies, while in application-centric migration, meta cloud only triggers the migration process, leaving its execution mostly to the application. We argue that the migration process should be controlled by the meta cloud but offers many interception points for applications to influence the process at all stages. The provisioning strategy, the most integrative component, which derives strategies mainly based on input from runtime monitoring and resource templates, and effects them by executing migration/deployment recipes, requires further research by combining approaches from the information retrieval and autonomic computing fields.

The meta cloud can help to mitigate vendor lock-in and promises transparent use of cloud computing services. Most of the basic technologies necessary to realize the meta cloud are already out there, yet lack integration. The integration of these state-of-the-art tools promises a huge leap towards the meta cloud, though. For avoiding meta cloud lock-in it is critical that the community drives the ideas, to create a truly open meta cloud with added value for all customers with broad support for different providers and implementation technologies.

References

Benjamin Satzger: is an assistant professor at the Distributed Systems Group at the Vienna University of Technology (TU Wien). Contact him at satzger@dsg.tuwien.ac.at.

Waldemar Hummer: is a PhD student at the Distributed Systems Group at TU Wien. Contact him at hummer@dsg.tuwien.ac.at.

Christian Inzinger: is a PhD student at the Distributed Systems Group at TU Wien. Contact him at inzinger@dsg.tuwien.ac.at.

Philipp Leitner: is a postdoctoral researcher in computer science at the Distributed Systems Group TU Wien. Contact him at leitner@dsg.tuwien.ac.at.

Schahram Dustdar: is a full professor of computer science and head of the Distributed Systems Group at TU Wien. Dustdar is an ACM Distinguished Scientist. Contact him at dustdar@dsg.tuwien.ac.at.