Mapping Non-Functional Requirements to Cloud Applications

David Villegas and S. Masoud Sadjadi
School of Computing and Information Sciences
Florida International University
Miami, Florida
{dvill013, sadjadi}@cs.fiu.edu

***Abstract*—Cloud computing represents a solution for applications with high scalability needs where usage patterns, and therefore resource requirements, may fluctuate based on external circumstances such as exposure or trending. However, in order to take advantage of the cloud's benefits, software engineers need to be able to express the application's needs in quantifiable terms. Additionally, cloud providers have to understand such requirements and offer methods to acquire the necessary infrastructure to fulfill the users' expectations. In this paper, we discuss the design and implementation of an Infrastructure as a Service cloud manager such that non-functional requirements determined during the requirements analysis phase can be mapped to properties for a group of Virtual Appliances running the application. The discussed management system ensures that expected Quality of Service is maintained during execution and can be considered during different development phases.**

I. Introduction

The emergence of cloud computing responds to a increasing trend in web application emergence and utilization. The wide adoption of Internet has resulted in systems that need to accomodate millions of users [1] and provide capabilities that until now were only required by critical, high availability or high throughput software. The practice of Software Engineering provides methodologies to ensure such characteristics are met, but it is necessary to review how they fit in this new paradigm. In this paper, we explore the applicability of traditional processes to the incipient field of cloud computing from the perspective of our research in an Infrastructure as a Service (IaaS) cloud manager.

Internet has resulted in rapid cycles of software development, deployment and consumption by users. The rising number of subscribers, better network connectivity and bandwidth, and the growing connectedness between users have created new dynamics where applications can be rapidly discovered and consumed. However, the benefits produced by these circumstances are hindered when expected requirements are not met. Nowadays, cloud computing is often employed as a solution to this problem. Capabilities such as pay-per-use, scalability or elastic provisioning of resources can help to overcome these new challenges. Nevertheless, application developers need to recognize how to apply Software Engineering methods to the cloud in order to successfully map their needs to fulfill service expectations.

There are two interrelated points that we believe have to be considered to successfully make use of clouds to develop applications that respond to the new demands generated in this field. First, developers must understand which non-functional requirements take renewed importance in cloud applications so that they can be accounted for during the requirements analysis phase. Second, cloud providers need to define better guarantees for their services, so developers can design their systems accordingly. We believe that providing a solution to these problems will result in a more dependable use of clouds to confront the new challenges of this era.

In this paper we consider the concept of *Distributed Ensembles of Virtual Appliances* (DEVAs), introduced in [2], as a model to represent complex systems with Quality of Service (QoS) guarantees. We discuss how a software architecture can be mapped to a DEVA, and how through the use of performance modeling and prediction we can make certain assurances about its behavior in the cloud in order to address its non-functional requirements. We finally present a case study were we demonstrate the feasibility of our approach to model the expected number of requests per second and response time of a web application hosted in the cloud.

II. Background

We define a cloud application as any software that runs on a distributed system that complies with the definition of a *cloud*. Such systems ([3], [4]) possess certain common capabilities such as on-demand provisioning, resource elasticity or pay-per-use billing model. Therefore, cloud applications can be deployed on remote resources with a minimal cost, and scaled dynamically when user demand grows.

We consider three main actors in our scenario: application users, application providers, and cloud providers. In this proposed division, application providers also have the role of cloud users, even though in certain cases it would be possible that application and cloud providers are the same individual or organization. The cloud is usually divided in Software, Platform and Infrastructure as a Service [3] —SaaS, PaaS and IaaS respectively. Application providers are in charge of implementing the SaaS layer, while the PaaS and IaaS layers are supplied by cloud providers.

A DEVA [2] is a group of Virtual Appliances and virtual network devices, where individual and composite policies can be defined for elements. Virtual Appliances [5] are Virtual

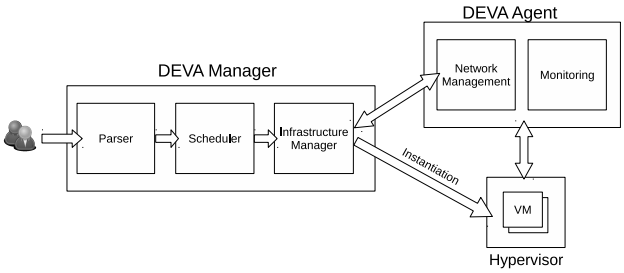

Fig. 1. General architecture

Machines with specific functions, usually containing a particular software and configuration; for simplicity, we'll use the more general term VM to refer to them. Figure 1 illustrates the architecture of the DEVA Manager. A user sends a specification for a list of VMs and their associated QoS requirements, which may consist of CPU, memory and required software for individual VMs, and network bandwidth and latency for the network links. Then, the Manager instantiates the ensemble across heterogeneous resources, which may be located in different administrative domains. A group of agents monitors each VM's behavior and provides the requested QoS and network isolation.

III. Requirement analysis and architectural design

In Software Engineering, the requirements analysis phase is in charge of determining the functional and non-functional requirements of the system based on the client's needs. In particular, non-functional requirements [6] describe the characteristics of the system not related to its functionality. These requirements shape the architecture of the system during the design phase.

In this paper we target a class of applications that are specially suited to be hosted in the cloud and have a prevalent set of non-functional requirements. Identifying them allows developers to ensure that they are addressed during the requirement analysis phase, and establish a set of requisites that must be met by cloud providers in order to quantify their service and ascertain the application goals are met successfully. We enumerate the most salient ones next.

Response time

This requirement describes how much time it takes from the moment a user sends a request to the system, until a complete response is provided. In web applications, this comprehends request transmission and processing, and response transmission. The factors that account for it are resource capabilities —processing power, memory, disk, network latency and bandwidth— and the load produced by other processes running in the server or the number of concurrent requests. For complex requests, this may also involve calls to external systems, or to other subsystems, in which case the host's internal network characteristics and other resources' load may be taken into account.

Uptime

The total time the service is available. It may be expressed as a percentage. When considering this requirement, it is necessary to take into account the provider's own uptime. For example, if a provider has an uptime of 99.5%, it would be impossible to deploy an application with a higher uptime. Other factors involve the recoverability of the system (*i.e.*, how much time it takes to restart the service after a failure happens).

Requests per unit of time

This requirement describes the number of requests the system can handle successfully per unit of time, and can also be referred to as the system's throughput. Resource allocation and usage has an impact in this parameter. Additionally, the number of requests can have an impact in the response time requirement (*i.e.,* a high number of requests will result in a deterioration of the overall response time).

Fault tolerance

One of the system's properties is how it can withstand errors, either hardware or software-based. In the case of cloud, non-software errors can be generated either at the physical or the virtual machines hosting the service. While the first case is usually out of the developer's control, virtual machine faults can be handled by different means, for example by spawning new instances, or having backup VMs to respond to failures.

Security

Security is another requirement that can be applied to the cloud provider or to the developed system. In the first case, the application developer is under the provider's security measures such as physical infrastructure access policies or network isolation mechanisms. Alternatively, security in the instantiated VMs must be handled by the cloud user.

Operational cost

In traditional systems, hardware was determined based on the application's initial requirements. Changes in requirements would typically result in costly upgrades involving the acquisition of new physical machines and installation and configuration of the application to run on them. In cloud systems, resources can be upgraded almost instantaneously, meaning that cost can be considered a changing variable. This allows defining tradeoffs to architectural (static) and operational (dynamic) behavior.

During the requirements analysis, it is the job of the software engineer to give appropriate values to each of the non-functional requirements according to the user's expectations. Each of these parameters needs to be reflected in one or more architectural decisions and tradeoffs.

IV. Mapping Requirements to DEVAs

The original implementation of the DEVA Manager accepts three types of parameters: nodes (VMs and virtual network devices), edges between nodes, and element annotations. Basic annotations describe node and edge characteristics such as VM processor power or memory size, and link bandwidth and latency, respectively.

An application developer could map the assigned non-functional requirement values to any of the discussed DEVA parameters in order to ensure the application's operational guarantees. For example, the number of desired requests per second would influence the assigned latency for the links between VMs; alternatively, the targeted response time could translate to a minimum processing power for the VMs in the ensemble.

However, this process is complicated and error-prone: the relationship between non-functional requirements and low level values is in many cases difficult to determine, and many factors can take part in the fulfillment of one individual parameter. Thus, we propose an extension to our system where non-functional requirements can be directly mapped to the execution platform, not only during the deployment phase, but also along the whole design process.

Our proposed approach in this paper extends DEVA annotations with new high-level values that correspond to non-functional requirements. An underlying application-dependent model is in charge of translating high-level parameters to low-level ones, and performing the required operations on the appropriate elements. We describe our system design next, and discuss its implementation.

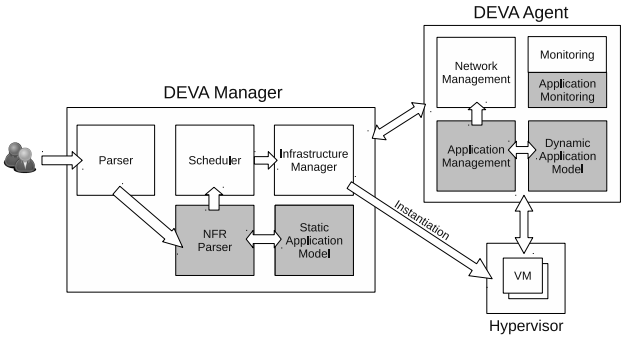

Fig. 2. Extended architecture

A. Processing annotations for DEVAs

Once the user defines a set of nodes, edges and annotations, a request is sent to the DEVA manager, which parses it and assigns the virtual resources to physical hosts that can comply with the requested QoS parameters. At each host, a number of VMs and virtual network links are created so that the ensemble appears as an isolated group of machines with dedicated resources. Additionally, a set of DEVA Agents are in charge of monitoring the infrastructure usage and ensuring global and individual QoS.

Requirements in the request are realized in two phases: first, the scheduling module of the manager chooses the target resources so that the ensemble requirements can be met. This implies selecting hosts with enough free memory and CPU for the VMs and ensuring the underlying physical network will be able to support the requested bandwidth and latency. Second, control measures are applied to constraint resource usage so that shared requests don't interfere among them. VM resources are adjusted by a Virtual Machine Monitor such as Xen or VMWare running in the target host, while network resources are controlled by a DEVA Agent. Agents apply shaping and monitoring to manage network QoS.

In order to implement high-level annotations representing non-functional requirements, we need to extend the DEVA manager and agents so that the new parameters can be translated into the existing ones. Figure 2 shows the system architecture with the new components in a darker shade.

First, the manager needs to translate non-functional requirements into values that can be considered as part of the scheduling process. We provide a non-functional requirements (NFR) Parser that is in charge of converting high-level values to low-level ones. For this, a Static Application Model is employed. Such model is dependent on the type of application, and can be defined either theoretically or empirically. We define a global annotation for the request, where the user can specify the application type. The value of this annotation will determine the model to use in the scheduler.

The Non-Functional Requirements Parser generates a set of requirements for nodes and connections based on the translated parameters, and these are fed to the scheduler, which takes them into account to generate a list of physical machine candidates. Each of the candidates is assigned a number of VMs to host. Finally, the Infrastructure manager, implemented in our system by OpenNebula [7], sends instantiation requests to Hypervisors and the DEVA Agents in charge of the dynamic behavior of the application.

After VMs are instantiated, DEVA Agents create a virtual network in the hosting machines. In our proposed architecture, we extend the system by adding three new components in the agents: First, we define an additional monitoring module with application dependent rules. While the basic component reads values such as CPU, memory and bandwidth usage, new values need to be contemplated in order to track non-functional requirements compliance. Examples of this are requests per second for a web server or database transactions for a database server. The application-dependent monitoring module can be extended based on different applications. All agents send the monitored data to the DEVA Manager, where the data is aggregated to determine high-level actions.

The second change in the DEVA Agents consists in an Application Management module similar to the existing Network Management component. While the later one is in charge of determining low-level actions to maintain network QoS, the new subsystem needs to consider high-level requirements and send them as an input to the other module. The third modification of the agent, the Dynamic Application Model, provides

the mapping based on a model of the application's behavior. Contrarily to the Non-Functional Requirements Parser and the Static Application Model, the components in the agent can also consider the runtime state of the application.

B. Model-based translation of non-functional requirements

There are two modules with the task of translating non-functional —high level— to infrastructure or low-level requirements. As stated in the last section, the first one considers the static behavior of the application and provides the necessary criteria to the scheduler, while the second one takes into account the dynamic state of the application. There are different approaches in the literature to modeling application performance such as [8] or [9], which can be divided into the categories of empirical, theoretical and on-line simulation models.

The first category corresponds to those models created from the application's off-line execution. Requirements can be inferred by observing the behavior of the system under different conditions and creating a model that can be later used to obtain approximate parameters to provide to the underlying management system. These models are usually measured by treating the application as a black-box (*i.e.*, without employing any knowledge of the internal implementation or design).

The second category consists of creating a mathematical model to relate the application's characteristics to its requirements. In this case, knowledge about the internal implementation is used to quantify the application's behavior based on available resources.

Finally, some models perform on-line (runtime) simulation of the application in order to find its behavior for a certain input. Simulations can be either event-based, for which an abstract model is created to approximate the behavior under certain conditions, or real-time, where a part or the whole application is executed to predict how the real system would behave.

Our system does not make any assumptions about the models used to map non-functional requirements to low-level ones. In fact, any of these could be employed either for the static or the dynamic modules in the manager and the agents. The basic prerequisite is that the used model understands the application's requirements and is able to determine a set of values that can be expressed via DEVA annotations. Some models may consider individual components of the system separately, while others contemplate complex relations between modules and how changes in one may affect others.

C. Non-functional requirements fulfillment

The modules added to the system allow the translation of non-functional requirements to low-level ones by using an application model. However, the DEVA Manager and agents need to perform the appropriate actions in order to fulfill the requested requirements. We classify these actions in two areas: resource allocation, and resource control. These categories also correspond to static and dynamic management, respectively.

The first type of actions is decided and enforced by the DEVA Manager based on the initial ensemble request and the model mapping. After parsing the user's input, non-functional requirements are translated into a set of low-level QoS values, which can be in turn used by the scheduler component to assign virtual elements to physical infrastructure. In our implementation in [2], the scheduler executes a metaheuristic algorithm to iteratively choose a near optimal placement of the requested DEVA in the available resources. This mapping would ensure that non-functional requirements are met by employing the appropriate infrastructure. Additionally, the DEVA Manager sends a description of the requested network links to each agent. Agents perform traffic isolation and shaping operations on the physical links to multiplex their usage among ensemble members, and when needed, create tunnels between physical hosts in different domains to build a virtual overlay network.

However, static allocation is not enough to respond to the runtime behavior of the application. While some values can be applied during the instantiation phase, most of the non-functional requirements need to be considered in terms of the application's dynamic usage. The DEVA agent is in charge of monitoring the system's running state and execute the appropriate control mechanisms. In many cases, these actions have associated trade-offs which need to be considered. Examples of control mechanisms run by the agents are dynamic bandwidth or CPU adjustment, provisioning of additional VM instances or VM migration.

V. Experimental Validation

In order to validate the proposed architecture, we have implemented a prototype extending the original DEVA Manager and agent. There are two main goals for this section:

1) Demonstrate the feasibility of translating high-level, non-functional requirements into a deployed ensemble of VMs.
2) Show how high-level QoS requirements are met during a DEVA lifecycle.

The experiment includes provisioning a test application through the DEVA Manager in order to determine how a set of non-functional requirements defined through the requirements analysis phase can be fulfilled during runtime. We have developed a three-tiered web application to illustrate the process.

A. The Chirper Application

In our test scenario, an fictitious company wants to develop an internal messaging systems so that their employees can communicate without having to use third party applications. They decide to deploy this solution in their private cloud so that they can take advantage of their in-house resources. The application, which we call *Chirper*, stores profile information for users, and enables them to post short messages to a common virtual board and query others' messages.

The application has two main components: the first one is a web server running the CherryPy[1] python web server;

[1] http://www.cherrypy.org

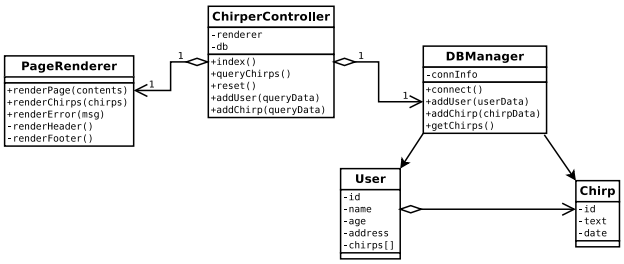

Fig. 3. Chirper class diagram

the second is a PostgreSQL[2] database with the users and messages information. The application will be accessed from the company's intranet.

As the first step, we perform the requirements elicitation to come up with the different functional and non-functional requirements. In this case, users need to be able to register in the application through a form, and then query either a specific user entries or last 50 messages in the database. We focus on a subset of the typical non-functional requirements explained in Section III: after exploring their users' behavior, our fictitious company estimates that the application should be able to respond to a peak of 40 requests per second, and that any request should be served in less than 500 milliseconds through the internal network.

In the second step, we define the application's architecture and implement it. In our approach, we follow the Model-View-Controller (MVC) architecture: a front-end interface where the user can interact with the system, a controller to submit and request data to the database, and the database layer itself. Figure 3 shows a class diagram of the system. The application receives requests through different URLs, which are mapped by CherryPy to the appropriate functions in the Chirper-Controller object. This class handles each request separately, performing input validation, then retrieving the requested information by calling the DBManager class, and finally rendering the response through the PageRenderer instance. The DBManager uses the SQLObject[3] Object-Relational Mapping (ORM) library to access the PostgreSQL database and perform selection and insertion operations. Finally, the PageRenderer class has methods to produce HTML code to return to the user.

In order to simplify scalability and be able to assign physical resources separately to each of the components, the web and database servers are deployed as different Virtual Appliances. Each of the VMs runs CentOS 5.3 and the software requirements needed by the application, which consist of Python 2.4 and CherryPy 3.2 for the Web appliance, and PostgreSQL 8.4 for the Database appliance. When the VMs are provisioned in the cloud, the DEVA Manager is able to define the resource allocation by sending commands to the VM Hypervisor and the DEVA agents. There are three main parameters that can be configurable once the VMs are instantiated in the infrastructure: maximum CPU share assigned to a VM, amount of memory, and bandwidth allocation between pairs of VMs.

[2]http://www.postgresql.org
[3]http://www.sqlobject.org

B. Performance Modeling

Once the application complies with the specified functional requirements, a model is created to account for the expected performance. In this example, a simple black-box model is defined by benchmarking the application externally. We deploy both appliances in the private cloud, consisting of a cluster of machines with Pentium 4 processors and 1 GB of memory running the Xen 3.0 hypervisor, and a third VM to act as a user. Physical machines are interconnected with 1 Gbps ethernet links and a dedicated network switch.

Initially, each VM is assigned a quota of 100% of the CPU, 1GB of disk, and 768 Mb of RAM. We run the Apache Benchmark tool to send 1000 requests with a level of concurrence of 10 to the service for each tested configuration. CherryPy is set up to spawn 10 serving threads without page caching mechanisms. We measure the request time and number of requests served per second for the operation of querying the last 50 messages in the database. We consider CPU and bandwidth as the VMs' configuration parameters. Memory allocation was discarded since the application doesn't require a high amount of main memory and consequently its performance doesn't depend on this parameter (our tests demonstrated 40 Mb were enough for the application to function at maximum capacity).

In the first set of runs, we calculate the application's behavior depending on the CPU quota. After running the tests, we determined that the Database appliance is not CPU bound, and therefore, there is no difference in performance with different values. Figure 4 shows the number of handled requests per second and the milliseconds taken for each request when the CPU allocation for the Web appliance is changed from 25% to 100% in intervals of 25%. As the figure shows, the number of served requests per second is directly proportional to the CPU allocation, while the time taken to respond to each request decreases with a higher CPU quota.

As the second set of measurements, we explore the application's behavior in relation to the allocated bandwidth. There are two links considered in this benchmark: the incoming connection to the Web appliance, and the private connection linking it to the database. Each of them can be constrained and isolated independently by the DEVA agents in the hosting machines. By doing this, each DEVA can perform independently of the rest, and network traffic from different applications is separated so that different VMs can multiplex the physical channel. We test the application with symmetric network assignments —*i.e.* same incoming and outgoing rate— from 100 Kb/s to 500 Kb/s with increments of 100 Kb/s. Figure 5 shows the results in requests per seconds and milliseconds per request for the incoming link (*in*) and the private one connecting both VMs (*priv*).

As it can be observed, the number of served requests per second depends on the available bandwidth, up to approximately 550 Kb/s for the incoming link (not shown in the graph)

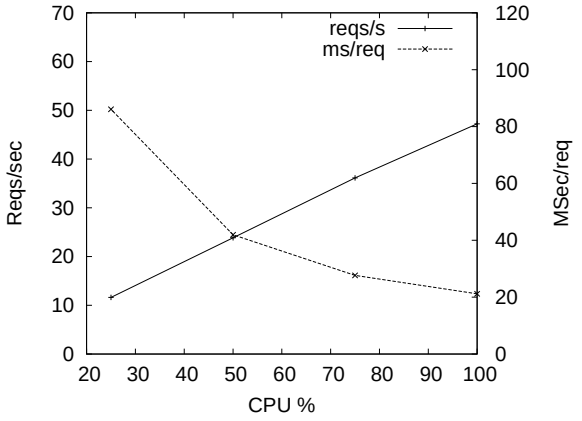

Fig. 4. Application model according to Web Appliance CPU allocation

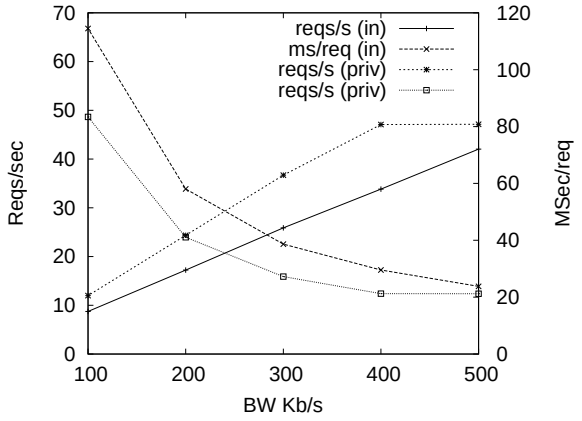

Fig. 5. Application model according to Incoming and Private bandwidth allocation

and 400 Kb/s for the private link. Lower bandwidth results in reduced requests per second and higher response time.

C. Integrating the model with the DEVA manager

The final step consists of integrating the experimental performance model into the DEVA manager and making the appropriate changes so that user requests can specify high-level requirements as parameters. We add the logic to translate such submission into low level parameters by employing the models and specifying an additional global parameter, *applicationType='Chirper'* so that the DEVA Manager knows which model to apply.

After that, we test the system by sending a request to the DEVA Manager specifying the two described Virtual Appliances and the desired non-functional requirements of 40 requests per second and 500 ms of maximum response time. The manager identifies this request to instantiate the *Chirper* application, and translates the requirements to 75% of CPU for the Web appliance and 25% of CPU for the Database appliance, 64 Mb or RAM for each VM, 500 Kb/s for incoming bandwidth and 400 Kb/s for private bandwidth. Finally, it decides that both VMs can be assigned to a single physical machine and provisions them accordingly.

VI. Conclusions and Future Work

As the cloud becomes more mainstream as a method to host applications, developers will need to consider how different providers —or in-house solutions— will be able to fulfill the final users' needs. Similarly, providers need to be able to give reliable guarantees for the Quality of Service of software deployed on their infrastruture. In this paper, we addressed this problem from both the developer's and cloud provider's perspectives. We showed how an example application with concrete requirements can be developed and deployed in a cloud manager that takes high-level non-functional requirements into consideration.

However, there are still many issues to address in order to determine how software can be successfully deployed in clouds: additional non-functional requirements such as fault-tolerance, execution cost or security need to be considered, and improved models that are able to predict applications' performance considering different parameters such as processor, memory, network and disk usage have to be developed.

VII. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. OISE-0730065.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," Feb 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html

[2] D. Villegas and S. Sadjadi, "DEVA: Distributed ensembles of virtual appliances in the cloud," in *Proceedings of the 17th Euro-Par Conference (Euro-Par 2011)*, 2011.

[3] L. Youseff, M. Butrico, and D. Da Silva, "Toward a unified ontology of cloud computing," in *Grid Computing Environments Workshop, 2008. GCE '08*, 2008, pp. 1 –10.

[4] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, pp. 7–18, 2010, 10.1007/s13174-010-0007-6.

[5] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum, "Virtual appliances for deploying and maintaining software," in *Proceedings of the 17th USENIX conference on System administration*. Berkeley, CA, USA: USENIX Association, 2003, pp. 181–194.

[6] L. Chung and J. do Prado Leite, "On non-functional requirements in software engineering," in *Conceptual Modeling: Foundations and Applications*, ser. Lecture Notes in Computer Science, A. Borgida, V. Chaudhri, P. Giorgini, and E. Yu, Eds. Springer Berlin / Heidelberg, 2009, vol. 5600, pp. 363–379.

[7] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, "Virtual infrastructure management in private and hybrid clouds," *IEEE Internet Computing*, vol. 13, pp. 14–22, 2009.

[8] C. Stewart and K. Shen, "Performance modeling and system management for multi-component online services," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 71–84.

[9] S. Sadjadi, S. Shimizu, J. Figueroa, R. Rangaswami, J. Delgado, H. Duran, and X. Collazo-Mojica, "A modeling approach for estimating execution time of long-running scientific applications," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1 –8.