A Deployment Platform for Dynamically Scaling Applications in The Cloud

Rui Han, Li Guo, Yike Guo*, Sijin He
Department of Computing
Imperial College London
London, UK
{r.han10, liguo, y.guo, sijin.he07}@imperial.ac.uk

Abstract— **Simplifying the process of deploying applications is almost essential in the cloud. However, existing techniques can automate applications' initial deployment but have not yet adequately addressed their dynamic scaling problems. In this paper, a deployment platform to enable a novel dynamic scaling technique is introduced. This platform employs: (i) an extensible specification that describes all aspects of applications; (ii) a flexible analytical model that determines how many servers to be deployed for an application in each scaling. The platform's ability to handle dynamic workloads and to scale applications quickly enough to maintain the response time target is demonstrated.**

Keywords- applications; cloud computing; deployment; dynamic scaling; platform

I. INTRODUCTION

With the significant progress in Information and Communications Technology, cloud computing allows thousands of individuals and small enterprises to produce and providing applications in a way that only large corporations could manage in the past [1]. In this context, irresistible trends promoted by leading cloud enterprises such as Amazon[2] encourage people to deliver their services in the cloud. Typically, a service is implemented as a multi-tier application composed of a series of servers running in VMs (virtual machines) and interacting across the network. However, since cloud providers only provide standalone VM images (e.g., Amazon EC2 [2]), service providers have to manually conduct a series of deployment tasks before they can deliver services in the cloud, which incur three problems. First of all, the deployment process is time-consuming process, in which a lot of time is wasted in tedious tasks such as installing, configuring and integrating applications. Secondly, the complexity nature of these tasks makes them error-prone. Finally, professional knowledge is required and external consultant hours for domain experts and solution architects are often expensive.

Recent work has tried to simplify the deployment process by providing pre-defined packages capable of being automatically deployed (e.g., RightScale's [3] and 3Tera [4]). Although existing techniques [3-8] can serve well for automating applications' initial deployment, they still remain deployed systems' dynamic scaling for human interventions. This manual redeployment requires services to be put offline and this is sometimes unaffordable for the service end users.

In this paper, we tackle the above challenges by proposing a platform that enables the agile dynamic scaling of cloud applications. This *dynamic scaling* denotes a process that responds to users' changing requirements and automatically scales applications without having to shut down the delivered services. The paper features three key elements:

• an extensible specification is introduced to explicitly describe all aspects of an application, including all configurations of its servers and their linking relationships.

• a flexible analytical model is utilised to capture the behaviour of various types of tiers in an application. A worst-case scenario is used to ensure the deployed application has sufficient resources even at peak workload.

• a successful trial of the complete platform is conducted using an open source application.

The rest of this paper is organized as follows: Section II presents some basic concepts and discusses motivation of this work and its related work. Section III then introduces our platform and explains how the dynamic scaling is realised. Section IV's experiments evaluate the platform's effectiveness and finally, Section V summarizes the work.

II. MOTIVATION AND RELATED WORK

This section first introduces a concrete example illustrating the problems we want to solve, followed by brief discussing on related work.

A. Example Scenarios and Problem Analysis

This section illustrates three problems that must be stepwise solved in achieving the dynamic scaling using an online bookstore application. When a service provider initially deploys this application, only one Tomcat web server and one MySQL database server are needed to support a small amount of customers, as shown in Figure 1(a). When the business is growing larger, this application is scaled up, as depicted by Figure 1(b). In the scaling, the Tomcat is expanded as several Apache web servers and a cluster of Tomcat application servers. These two types of servers are designed to handle static and dynamic user requests, respectively. Similarly, a number of MySQL Slavers are added to handle "read" requests so as to alleviate MySQL Master's load. In addition, Varnish and Memcache cache servers are utilized to accelerate user access speed. Two HAProxy are used to distribute.

* Please direct your enquiries to the communication author Professor Yike Guo.

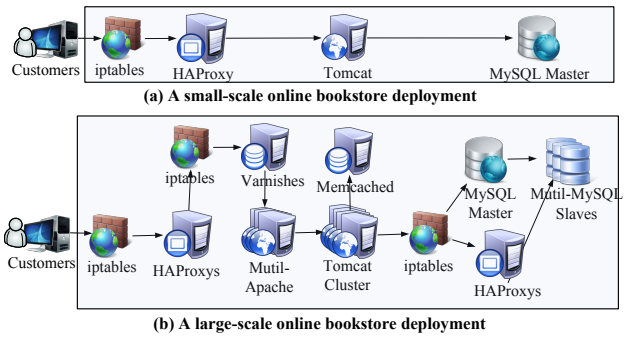


Figure 1. Example small and large-scale online bookstore applications.

Problem 1: Server specification. To automate the above scaling process, a specification must be proposed to describe all information of each single server including its VM configuration, user settings and linking with other servers. Based on this specification, basic deployment activities such as adding or removing should be automatically executed.

Problem 2: How much to scale. In Figure 1's online bookstore, when the number of customers increases, a capacity estimation must be triggered to ensure enough servers are deployed to meet the response time specified in the service-level agreements (SLAs).

Problem 3: How to scale. In the scaling, the deploying of servers must be implemented in right order. For example, in Figure 1(a), the HAProxy must be deployed later than the Tomcat to connect it; in Figure 1(b), multiple Tomcats can be deployed in parallel to form a cluster. In addition, end users' interface such as HAProxys in Figure 1's two examples must be kept open and unchanged to ensure that end users can always get access to the service.

B. *Discussions on Related Work*

In cloud computing, a representative technique in simplifying deployment is proposed by RightScale [3]. It integrates applications with VM images to generate server templates that can be automatically deployed. In addition, some other enterprises such as 3Tera [4] provide visual user portals to facilitate the design of deployment plans.

In addition, researchers have proposed a number of approaches to facilitate deployment in the cloud. Konstantinou et al. [5] introduce a model-based architecture using virtual solution models to provide abstract deployment plans that are platform-independent. When a VSM is bound to a cloud platform, it can be transformed into an executable deployment plan. Chieu et al [6] present a cloud provisioning system that preloads applications in VMs to generate basic application images. This system allows users to specify complex deployment scenarios by constructing these application images. In addition, Xabriel et al. use a meta model based approach to automatic applications' initial deployment and their approach also allows static deployment modifications at the design time [7]. Hughes et al propose a framework to support individual applications' self-management, including setup, configuration, recovery and scaling up and down [8].

However, to the best of our knowledge, although previous investigations of this sort have solved applications' initial deployment satisfactorily, they only support single applications' dynamic scaling. This means any scaling of the whole deployment still needs human interventions. The deployment platform proposed in this work, therefore, attempts to solve this problem by supporting both applications' initial deployment and dynamic scaling.

Furthermore, some platforms are proposed to manage the deployment of specific types of applications supported by the provided platforms. Microsoft Windows Azure [9] and Google App Engine [10] assist developers to conveniently create, deploy and scale their applications. These applications are required to be developed with only the supported languages, such as Python or Java in Google App Engine. In addition, Aneka Clouds provides three different programming models, i.e., Task, Thread, and MapReduce, to support the deployment of three types of applications [11]. Inheriting and developing from these techniques, this work aims at offering service providers a general platform to deploy applications with no restrictions.

III. THE PLATFORM FOR DYNAMICALLY SCALING APPLICATIONS

At the beginning of this section, we give an overview of the platform. In the following three sections, Section III.B explains the application specification (problem 1). Section III.C presents the capacity planning technique (problem 2). Finally, Section III.C gives the principles in implementing the dynamic scaling (problem 3).

A. *The Platform Overview*

The proposed platform acts as middleware between service providers and cloud providers, as shown in Figure 2. At the client side, the *Application Deployment Portal* assists service providers to define deployment specifications and executes deployment on their behalf. At the server side, the five service components supports the dynamic scaling.

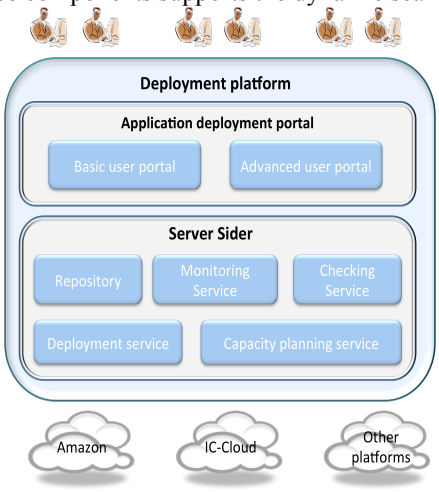


Figure 2. The architecture of the platform

The *Repository* contains a library of single servers and whole deployment templates including all necessary servers

for an application. These servers and templates are pre-designed by domain experts and solution architects based on best practice. The *Repository* also provides convenient registration mechanisms. For example, if service providers want to use JBoss web server but cannot find it in the *Repository*. They can register it in the platform by completing a series of standard registration procedures, after which they can even package the JBoss web server into a deployment template. This newly registered JBoss server can then be dynamically scaled in the same way as the pre-defined Tomcat server.

The *Monitoring Service* has two functionalities. First, it monitors the execution of an application by attaching a monitoring tool to its entry server (e.g., Figure 1(a)'s HAProxy). This tool examines the per-request response time over a finite interval (e.g., 30 seconds) and trigger a dynamic scaling if the observed response time exceeds the required one. Secondly, the *Monitoring Service* allocates each server a monitor. This monitor keeps the application execution records, which are used to analyses data needed in the capacity planning. For example, the mean and variation of an server's service time per request.

In addition, the *Checking Service* exams all the information in deployment specifications and alerts service providers if any syntax and semantic error (e.g., two HAProxys are connected) is found. The *Checking Service* is implemented as an expert system, which utilizes validation rules to detect errors. Drools Expert [28] is used as a rule engine to reason these rules in our platform. In addition, the *Deployment Service* interprets deployment specifications and automatically executes the basic deployment activities upon multiple cloud platforms such as Amazon. The *Capacity Planning Service* employs the analytical modelling technique to transform high level QoS requirements into the number of servers to be deployed.

B. Dynamic Scaling Basis: Server Specification

A XML based specification is designed to accommodate all aspects of a single server based on according to the Open Virtualization Format (OVF) open standard [12]. Figure 3 shows an example specification of a Tomcat server, which has four sections. The first section lists basic information of this server and the following sections are used by the *Deployment Service* to *add*, *modify* or *remove* this server. For instance, in the *adding* activity, the *Deployment Service* first produces a Tomcat VM image as specified in the VM Configuration Section. Three of this Tomcat's user settings are then configured using data in the user setting section. Finally, the Tomcat is linked to the MySQL Master according to the "output-server-id" specified to the linking section.

In a multi-tier application, server are categorized into different tiers based on the services that they can provide, as listed in Figure 4: The Storage tier is used for managing data; the Service tier is responsible for delivering services to end users; and two LB (Load Balance) tiers distribute requests to Service or Storage tier. In addition, Service and Storage tier can have multiple sub tiers. In a m-tier applications, tiers of servers are numbered consecutively (from 1 to m) according to these servers' tier types: Storage tier, the LB tier above Storage tier, the Service tier, the LB tier above Service tier. For instance, in Figure 1(a)'s example, the tier number of MySQL, Apache and Tomcat servers are 1, 2 and 3, respectively.

```
<Section xsi:type="ovf:BasicInformationSection_Type">
        <Info>Basic information of the server</Info>
        <Server-Id>Tomcat Server One</Server-Id >
        <Description>Tomcat application server</ Description>
        <Owner>IC-Cloud</Owner >
        <Tier-id>4</ Tier-id>
        <availability-zone>us-east-1a</availability-zone>
        <pricing-model>pay-as-you-go</pricing-model>
</Section>
<Section xsi:type="ovf:VMConfigurationSection_Type">
        <Info>Server's VM configuration</Info>
        <Instance-type>High-CPU Medium</Instance-type>
        <CPU-cores>2</CPU-cores>
        <Memory>1.7GB</Memory>
        <Storage>350GB</Storage>
        <Operating-system>Linux-Ubuntu</Operating-system>
</Section>
<Section xsi:type="ovf:UserSettingSection_Type">
        <Info>Users' configuration settings for the server</Info>
        <User-name>ic-cloud</User-name>
        <Password>ic-cloud</Password>
        <Port-number>8080</Port-number>
</Section>
<Section xsi:type="ovf:LinkingSection_Type">
        <Info>Server's link information</Info>
        <Output-server-id>MySQL Master One</ Output-server-id>
        <Link-purpose>Obtain data</Link-purpose>
</Section>
```

Figure 3. A fragment of Tomcat server specification

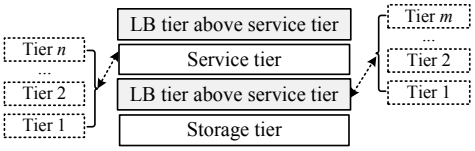


Figure 4. The four types of tiers in an application.

C. How Much to Scale: Queueing Model Based Capacity Estimation

To address the issue of how many servers to be deployed at each tier of an application in a dynamic scaling, we construct an analytical model of this application. In our platform, the *Capacity Planning Service* takes a deployment's required response time, its request arrival rate and service demand of each request as inputs, and conducts the capacity estimation in four steps.

In the first step, an application is modelled as a network of queues, in which each queue represents a tier in the deployment and the queue from a tier feeds its adjacent tier. Consider an application that comprises m tiers, where m-2 tiers belong to the Service or Storage tier and the remaining *2* tiers belong to the LB tier. Observe that servers belonging to the first m-2 tiers typically occupy most of the response time, the required end-to-end response time *r* is broken down into the per-tier response time of these m-2 tiers using offline profiling:

$$R = \sum_{i=1}^{m-2} r_i. \quad (1)$$

In the second step, a single server at tier i ($1 \le i \le m$-2) is modelled as a G/G/1 queueing model. This model can handle arbitrary arrival distribution and service time distribution. This makes the model comprehensive to capture behaviours of various types of applications belonging to the Service and Storage tiers such as Apache and MySQL. This G/G/1 model is then solved to estimate the application's capacity [13]:

$$X_i \geq \left[\overline{s_i} + \frac{\sigma_{\lambda_i}^2 + \sigma_{s_i}^2}{2(r_i - \overline{s_i})} \right]^{-1}. \quad (2)$$

Where $\overline{s_i}$ is the mean service time and $\sigma_{\lambda_i}^2$ and $\sigma_{s_i}^2$ are the variances of the inter-arrival time and service time, respectively. These values of the application can be monitored online by its attached monitor in our platform. The required per-tier response time r_i is known. By substituting these values into equation 2, this single server's capacity X_i, namely the lower bound request arrival rate it can handle, can be obtained.

Assumed that, the whole application is stable and thus the job flow of tier i is balanced, i.e., the number of requests arrives at this tier is equal to the number of jobs it can handle during a finite observation period. Once the capacity of a single server at tier i is known, the number of servers needed at this tier under the peak arrival rate can be calculated according to the operational law [14]:

$$n_i' = \frac{V_i \lambda}{X_i}. \quad (3)$$

Where λ is the peak request rate (i.e., the high percentile of the arrival rate distribution) and V_i is the average number of requests at tier i for an incoming request. Observe that V_i is usually more than one. For instance, a single search request at an online bookstore might trigger multiple requests to Apache web servers and MySQL databases. The above values can be estimated using online measurements. For convenience, equation 3 assumes that all applications at a tier are homogeneous and they share the incoming requests of this tier. Relaxation of this assumption is possible and it is not discussed in this work due to space constraints.

The *Capacity Planning Service* also considers servers' replication constraints. For instance, each application typically has at most two MySQL Masters servers, so MySQL Master's degree of replication d is 2. The sever number n_i' of tier i ($1 \le i \le m$-2), therefore, is corrected: $n_i = \min(n_i', d_i)$, which means this tier can deploy no more than $\min(n_i', d_i)$ applications. Furthermore, using the server numbers of the first m-2 tiers, these numbers of the remaining 2 tiers can then be determined. For example, in Figure 1(b), every 10 Apache web servers need one HAProxy load balancer.

In the final step, the total deployment cost is calculated. In the cloud, applications are usually priced in Pay-as-you-go model and an application's cost is usually decided by its instance type. For example, in Amazon EC2, a large instance of MySQL application is priced 12.6 cents/hour and this application is charged 87.2 cents/hour if its instance type is extra large. The total cost of a deployment with m tiers is the summer of each tier's cost:

$$C = \sum_{k=1}^{m} n_k c_k \leq C_{budget}. \quad (4)$$

Where n_k is tier k's server number and c_k is the cost of a single server at tier k ($1 \le k \le n+m$). In the capacity planning, the total cost C should be less than service providers' budget C_{budget}.

D. How to Scale: The Dynamic Scaling Algorithm

When the number of server to be deployed at each tier of a deployment is obtained, the *Deployment Service* conducts the automatic scaling to deploy all these servers. Figure 5 shows the dynamic scaling algorithm, which follows two principles: Servers at the same tier can be deployed in parallel (line 8 and 12); The scaling sequence of servers in different tiers is arranged in ascending order according to servers' tier numbers (line 7 and 11).

This algorithm will continuously running before the application is terminated, so we only analyse the time complexity of each scaling. Both scaling up and down (line 7, 8 and line 10, 11) have m cycles and each cycle can be completed in constant time, so time complexity of each scaling is O(m), where m is the application's tier number.

The dynamic scaling Algorithm
Input: All servers of a m-tier application and service providers' QoS requirements.
1. **Begin**
2. **while** (the application is not terminated)
3. Monitor the arrival rate λ;
4. Let λ' be the last observed arrival time;
5. **if** $\lambda < \lambda'$, **then** // the request rate increases.
6. Conduct the capacity estimation for scaling up;
7. **for** (i=1; $i \le m$; i= i+1)
8. Simultaneously add each server at tier i;
9. **else if** $\lambda > \lambda'$, **then** // the request rate decreases.
10. Conduct the capacity estimation for scaling down;
11. **for** (i=1; $i \le m$; i= i+1)
12. Simultaneously remove each server from tier i;
13. **End**

Figure 5. The dynamic scaling algorithm

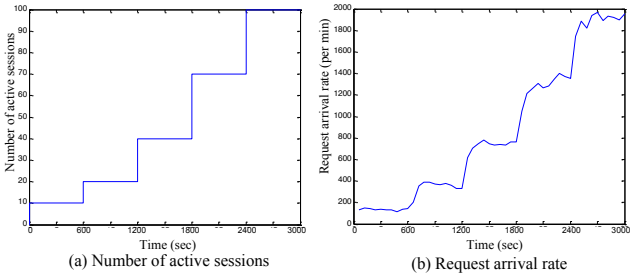
IV. EXPERIMENT EVALUATIONS

In this section, we did experiment by testing Figure 1's online bookstore example. The experiment is designed to prove our platform's ability to dynamically scale an application under changing workloads to maintain its response time target. The experiment assumes that the required end-to-end average response time is less than 1 second. This response time is broken down into the per-tier response time, which is 10, 50, 10 and 30% for the tiers of Apache, Tomcat, MySQL Master and MySQL Slaver, respectively. In addition, the workload is represented by the number of active sessions.

In the experiment, we test five active (simultaneous) session numbers: 10, 20, 40, 70 and 100, which denote the workload increases from low to high, as shown in Figure 6(a). The first workload is generated at time=0 second and it lasts 600 seconds. We observe the deployment once every 60 seconds and Figure 6(b) displays the 10 request arrival rates (i.e., the number of arrival requests per second) during this period. Similarly, the other four workloads are generated at

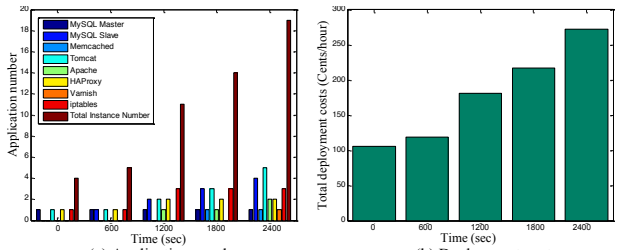
time = 600, 1200, 1800, 2400 sec, respectively and Figure 6(b) shows that the request arrival rates increase together with these workloads.

Figure 7(a) lists the eight types of servers' numbers and the total server number under the five workloads. For the first workload (active session number is 10), the online bookstore service is initially deployed with one MySQL Master, Tomcat, HAProxy and iptables, respectively. When the active session is increased to 20 at time = 600s and saturates the MySQL Master, a dynamic scaling is triggered and one MySQL Slaver is deployed. When the active session is increased at time=1200, 1800 and 2400, the above cycle repeats. Figure 7(b) shows that the deployment cost increases as the deployment is scaled up.



(a) Number of active sessions (b) Request arrival rate

Figure 6. The number of active sessions and request arrival rate



(a) Application number (b) Deployment cost

Figure 7. The application number and deployment cost

From service users' perspective, Figure 8 demonstrates the fluctuation of the observed average response time. This shows that the SLA is violated whenever the active session is increased. This is due to the fact that a dynamic scaling is difficult to be completed on-the-fly and servers need some time to be deployed. Especially, MySQL Slavers take time to replicate with the MySQL Master. The experiment result shows that after each dynamic scaling, the delivered service can meet the response time target, thus proving the effectiveness of the capacity planning. The result also illustrates that these scaling processes can be completed in a short time, usually within 2 or 3 minutes.

V. CONCLUSIONS

In this paper, we argued that dynamic scaling of applications raises new challenges that are not adequately addressed by existing work on application deployment. We introduced a platform that enables the novel dynamic scaling technique for cloud applications. This platform employs: (i) an extensible specification that precisely describes all information of a deployment, and (ii) a flexible capacity planning using queuing model to compute how many applications to be deployed at each tier of the deployment. The platform has been currently implemented as a service of the IC Cloud workstation and an experiment evaluation using a practical application has been conducted. The experiment shows the platform's ability in quickly scaling a deployment to maintain its response time target under dynamic workloads.

In the future, we plan to develop intelligent capacity estimation techniques to deal with complex workloads and optimise the cost/performance ratio of deployed applications.

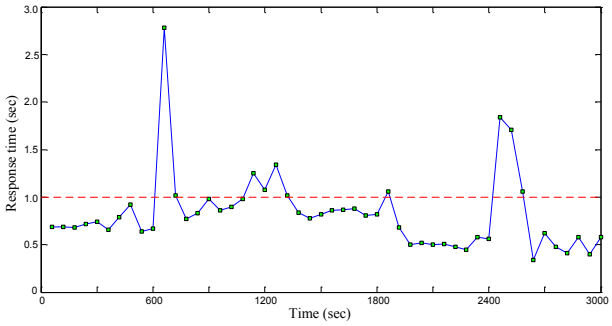


Figure 8. The observed end-to-end average response time

REFERENCES

[1] R. Buyya, C. S. Yeo, S. Venugopal et al., "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," Future Generation Computer Systems, vol. 25, no. 6, pp. 599-616, 2009.

[2] "Amazon EC2," http://aws.amazon.com/ec2/.

[3] "RightScale "; http://www.rightscale.com/.

[4] "CA 3Tera," http://www.3tera.com/.

[5] A. V. Konstantinou, T. Eilam, M. Kalantar et al., "An architecture for virtual solution composition and deployment in infrastructure clouds.", VTDC'09, 2009, pp. 9-18.

[6] T. Chieu, A. Karve, A. Mohindra et al., "Simplifying solution deployment on a Cloud through composite appliances." IPDPSW'10, 2010, pp. 1-5.

[7] X. J. Collazo-Mojica, and S. M. Sadjadi, "A Metamodel for Distributed Ensembles of Virtual Appliances." SEKE'11, 2011.

[8] G. Hughes, D. Al-Jumeily, and A. Hussain, "A Declarative Language Framework for Cloud Computing Management."DESE'09, 2009, pp. 279-284.

[9] "Microsoft Windows Azure," http://www.microsoft.com/windowsazure/.

[10] "Google App Engine," http://code.google.com/appengine/.

[11] R. Buyya and K. Sukumar, "Platforms for Building and Deploying Applications for Cloud Computing", CoRR'11, 2011, pp. 6-11.

[12] "VMware OVF," http://www.vmware.com/appliances/getting-started/learn/ovf.html.

[13] Robert B. Cooper, Introduction to Queueing Theory, Second ed., North Holland, Oxford: New York,1990.

[14] R. Jain, The art of computer systems performance analysis,1989 edition, John Wiley & Sons, New York, 1991.