On the Relatively Small Impact of Deep Dependencies on Cloud Application Reliability

Xiaowei Wang, Fabian Glaser, Steffen Herbold, Jens Grabowski
Institute of Computer Science, University of Goettingen, Germany
Email: {*xiaowei.wang, fabian.glaser, herbold, grabowski*}@cs.uni-goettingen.de

Abstract—**Reliability is one of the key concerns of both cloud providers and consumers, who require accurate reliability evaluation methods to develop, deploy, and maintain cloud applications. However, few works assess the reliability of cloud applications considering deep dependencies in the deployment stack. To explore the impact of deep dependencies to the reliability assessment, in this paper, we propose a layered dependency graph-based reliability assessment method for cloud applications. By introducing the inner reliability of cloud components, and combining the deep dependencies between services and physical servers, the method can assess the reliability of all components as well as the application. We implement a framework for the method and compare the assessment results of our approach against existing methods. The results show that deep dependencies have few impacts on the accuracy while can improve the precision of reliability assessment methods because the failure rate of physical servers is much lower than software components.**

Index Terms—**cloud application; reliability assessment; dependencies**

I. Introduction

After the development during the last decade, cloud computing basically lives up to the promise of providing computing resources and services as utilities [1] [2]. It provides seemingly infinite resources to consumers by means of resource pooling and rapid elasticity [3]. While employing cloud computing to reduce the purchase and maintenance cost of hardware in traditional Information Technology (IT) industry, consumers also expect at least as reliable services as provided by in-house systems. Actually, reliability-related issues are among the top obstacles for the adoption of cloud computing [1].

One typical use case scenario of cloud computing is that consumers deploy applications on clouds and provide services to end users. In this scenario, an application is usually divided into services which are deployed to Virtual Machines (VMs) hosted by Physical Servers (PSs). The deployment stack of an application comprises Services, VMs, PSs, and other hardware and software components, e.g., routers and hypervisors. Rigid failure containment of software failures at the VM level provided by virtualization [4] and the service-oriented architecture of cloud computing make it intuitive to model cloud applications with component-based architectures [5] [6], more specifically, in a hierarchical manner [7] [8] [9].

Components in the deployment stack are subject to failure propagation. For instance, when several VMs are consolidated into one PS, failures of the PS will lead to failures of all these VMs. Failure propagation caused by the dependency between VMs and PSs is evident and well researched [10]. However, the *deep dependency* [11] between services and PSs is nontransparent and often not considered for reliability analysis [12]. With deep dependencies we refer to indirect dependencies that go deep into the deployment stack, i.e., on the virtual and physical infrastructure. A service is usually deployed with several identical Service Instances (SIs) as redundancies to tolerant PS failures. But due to server consolidation or limited control of the deployment process, SIs of the same service may be deployed on one PS. In this case, failures of the PS may crash the service and possibly the whole application. Furthermore, redundant SIs can be configured in different ways, e.g., one service may require at least one SI to succeed and another service may require at least k, $k > 1$, SIs to ensure a certain level of performance. Different configurations of SIs lead to different deep dependencies. The normal redundancy requiring at least one available SI is very often considered in the context of reliability assessment for services and cloud applications [10] [13], while the k-out-of-n redundancy is not. In this paper, we intend to determine if considering deep dependencies can improve the quality of reliability assessment methods and, if so, explore the degree of the impact.

To tackle the challenge, we propose a dependency-based reliability assessment method for cloud applications. The main contributions of this paper are:

- a reliability assessment method for cloud services and applications considering deep dependencies;
- a process for comparing the quality of reliability assessment methods for cloud applications; and
- a comparison of qualities of different reliability assessment methods.

The rest of the paper is organized as follows. Section II demonstrates related works. Section III describes the reliability assessment method. Section IV illustrates a framework and the comparison with other methods. Section V concludes this paper.

2159-6190/17 $31.00 © 2017 IEEE
DOI 10.1109/CLOUD.2017.73

II. Related Work

Many works research the assessment and prediction of the reliability of cloud services and applications. Assessment refers to the analysis of the reliability based on already existing deployments, e.g., due to measurements. Prediction of reliability deals with the problem to forecast the expected reliability for based on the planned deployment stack.

According to the considered components in these works, we introduce representative ones in the following.

Considering only failures of hardware, Faragardi et al. [14] proposed an Analytical Reliability Model (denoted by ARM) for Cloud Computing Systems (CCSs) which are modeled as a set of linked PSs with resources of memory, storage space, computation power, and network bandwidth. They divide a cloud service into tasks and assume that both the service and tasks are fully reliable. They calculate the CCS reliability by combining the reliability of PSs and links, and calculate the PS reliability as the product of the reliability of memory, hard disk, RAID controller, and processor. Under several constraints including memory, quality of service, task precedence, communication load, and task redundancy, the maximum reliability of the CCS is evaluated. Comparing with our work, Faragardi et al. do not consider failures of VMs or SIs.

Focusing on failures of VMs and PSs, Qiu et al. [10] proposed a Hierarchical Correlation Model (denoted by HCM) for evaluating cloud services' reliability, performance, and power consumption. HCM assesses the reliability of large online services, e.g., social networking services. To connect the service reliability with performance and power consumption, they define the service reliability as the probability that at least one VM used by the service is available. They consider only failures of VMs and PSs and utilize a Markov process to model the amount of VMs. The difference to our work is that they do not consider the k-out-of-n redundancy or service reliability.

Taking services, SIs and data into consideration, Wang et al. [13] proposed a Hierarchical Reliability Model (denoted by HRM) for modeling and evaluating the reliability of service-based software systems. HRM assesses the software reliability by combining the reliability of the workflow, service pools, services and data. HRM can be adapted to evaluate the reliability of a cloud application by modeling it as composite services, but no failures of VMs or PSs are considered as our work does due to the design goal of HRM.

Concentrating on the failures of the application itself, Banerjee et al. [15] proposed an Log-Based Reliability Analysis (LBRA) of a commercial Service as a Service (SaaS) application. They define the reliability as $R = 1 - f/n$, where f is the number of failed entries/sessions and n is the total number of entries/sessions, to assess the application reliability. Banerjee et al. focus on SaaS reliability and consider only application failures but no underlying failures, such as VM failures. LBRA can also be applied to evaluate the reliability of common cloud applications as long as access logs are available.

Based on the whole deployment stack of an application, Thanakornworakij et al. [9] proposed a High Performance Applications Reliability Model (HPARM) specifically for MPI applications deployed on cloud systems. They consider the reliability of application-related components, such as SIs, VMs, hypervisors and PSs, and relationships of these components. They focus on exploring the impact of correlations of failures to the cloud application reliability. However, correlated failures of PSs are rare and only obvious due to network failures [16], and correlated failures of services without load sharing are not as significant for normal cloud applications as for high performance applications. Besides, they do not consider redundancies as we do in our work.

Comparing with the existing works, our method considers PS failures, VM failures, service failures, and also the dependencies between them and can assess the reliability of components and the application.

III. Dependency-based Reliability Assessment for Cloud Applications

In this paper, *reliability* is defined as "the ability of a system or component to perform its required functions under stated conditions for a specified period of time" [17]. To assess the reliability of cloud applications, we model dependencies between the components in the deployment stack with Layered Dependency Graphs (LDGs). Based on the LDG of an application, we assess the reliability of services and also the application with our DEpendency-Based Reliability Assessment (DEBRA) method.

A. Layered Dependency Graph

A *dependency* is defined as the relationship between two components that one component requires another one to fulfill its function. The component that needs another one is the *predecessor* and the required component is the *successor*. Dependencies are transitive, i.e., the successor of a component's successor is also the component's successor and the same for predecessors. Application components include services, SIs, VMs and PSs. In the following, we introduce how the dependencies between components are modeled.

We model a cloud application as a set of services, each of which has at least one SI and every SI is deployed on one VM. SIs of a service are generally organized as a k-out-of-n system ($k \geq 1$), which means that the service has n SIs and requires at least k SIs to succeed. The dependency between two services are defined as a *function dependency* which means that a service needs

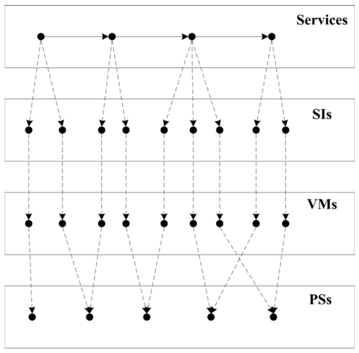


Fig. 1. An example of the LDG

another service for its full function. Function dependencies can be modeled by architectural styles proposed by Wang et al. [18]. In this paper, we use a sequence of services to illustrate our method. SIs are deployed on VMs which are deployed on PSs and a service can also be seen as deployed through its SIs. Therefore, we define the dependencies between services and SIs, between SIs and VMs, and between VMs and PSs as *deployment dependencies*. In this paper, a *deployment dependency* is defined as the dependency between two components when one component is deployed on another one. Besides, we assume that a PS can host several VMs while one VM can hold only one SI as in Thanakornworakij's work [9]. Under this assumption, SIs and VMs are one-to-one mapped and VMs hosting SIs of the same service may be deployed on the same PS.

Based on the above assumptions, an example of the LDG is shown in Figure 1, where solid arrows represent function dependencies and dashed arrows represent deployment dependencies. As Figure 1 shows, an LDG is composed of four layers from bottom to top: the PS layer, the VM layer, the SI layer, and the service layer. PSs, SIs, VMs, and services are in the corresponding layer.

B. Reliability Assessment

Based on the definition of reliability, we further define the *inner reliability* of a PS, VM or SI as the reliability without need of other components and the inner reliability of a service as the reliability without need of its service successors. Correspondingly, we define the failures of a component itself as its *inner failures* and the rate of inner failures as its *inner failure rate*. The following assumptions are made for reliability assessment:

- Component failures are fail-stop, which means that a component suffering a failure will stop working and the failure can be detected.
- Inner failures of non-service components are independent and follow exponential distributions.
- The same type of non-service components have the same inner reliability.

We define the reliability (denoted by R) of a cloud component as the product of its inner reliability r and

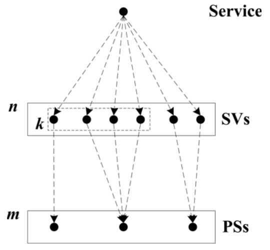


Fig. 2. An example structure of a service

the reliability R_i of components it depends on [19]. Then, the reliability of a component can be calculated by:

$$R = r\prod_{i=1}^{n} R_i \tag{1}$$

where n is the number of components it depends on.

PS reliability. In a LDG, PSs are in the bottom layer and have no successors. As a result, the reliability of a PS R_{ps} is fully determined by its inner reliability r_{ps}. And based on the exponential distribution, the PS reliability can be calculated with:

$$R_{ps} = r_{ps} = e^{-\lambda_{ps}t} \tag{2}$$

where λ_{ps} is the inner failure rate of the PS.

SI-VM (SV) reliability. Due to the one-to-one mapping between SIs and VMs, we define an abstract component SV as the composite of an SI and the VM hosting it. An SV has only one successor which is the PS hosting it, therefore, the inner reliability r_{sv} and the reliability R_{sv} of an SV can be calculated respectively by:

$$r_{sv} = e^{-\lambda_{sv}t} \tag{3}$$

$$R_{sv} = r_{sv}R_{ps} = e^{(\lambda_{sv}+\lambda_{ps})t} \tag{4}$$

where λ_{sv} is the inner failure rate of the SV and $\lambda_{sv} = \lambda_{vm} + \lambda_{si}$, where λ_{vm} and λ_{si} are respectively the inner failure rate of the VM and SI compositing the SV.

Service reliability. *When a service depends on no other services, i.e., has no service successors, its reliability equals its inner reliability*. Suppose a service has no service successors while it has n SVs and requires at least k ($k \geq 1$) of them to succeed. We assume that the n SVs are hosted by m PSs with n_l ($l = 1, 2, ..., m$) SVs hosted by the lth PS. Then, the inner reliability of the service r_{se} is the probability that k or more SVs succeed and can be calculated by:

$$r_{se} = \sum_{i=k}^{n} P_{v_i} \tag{5}$$

where P_{v_i} is the probability of the event v_i that i out of n SVs succeed. For example, suppose a service has six SVs deployed on three PSs and one, three and two SVs are deployed on the first, second and third PS, as shown in Figure 2. If the service requires at least four SVs to succeed, we have $k = 4$, $n = 6$, $m = 3$ and $n_1 = 1$, $n_2 = 3$,

$n_3 = 2$. When $i \geq k$, there are a number of possibilities to choose i out of six SVs. Based on the distribution of successful SVs on PSs, we separate these possibilities into scenarios. Suppose there are d_i scenarios that i ($i \geq k$) SVs succeed, then P_{v_i} can be calculated by summing up the probability of each scenario:

$$P_{v_i} = \sum_{j=1}^{d_i} P_{c_j} \tag{6}$$

where P_{c_j} is the probability that the jth scenario c_j occurs. For c_j, suppose the number of successful SVs on the lth PS is s_l. Then, $c_j = (s_1, s_2, ..., s_m)$ and s_l should meet:

$$0 \leq s_l \leq n_l \tag{7}$$

$$\sum_{l=1}^{m} s_l = i \tag{8}$$

where (7) indicates that the number of successful SVs on a PS, s_l, can range from zero to the number of total hosted SVs n_l, and (8) indicates that the number of successful SVs on each PS sums up to the number of total successful SVs.

Before calculating P_{c_j}, we need to find out all the d_i scenarios that i out of n SVs succeed. As $c_j = (s_1, s_2, ..., s_m)$, $1 \leq j \leq d_i$, the problem of how to find out the d_i scenarios can be formally described as how to distribute i SVs to m PSs, or how to find out the restricted weak compositions of the integer i into m parts, with every part s_l restricted by (7). As defined by Bona [20], a *weak composition* of an integer n is "a sequence $(a_1, a_2, ..., a_k)$ of integers fulfilling $a_i \geq 0$ for all i, and $(a_1 + a_2 + ... + a_k) = n$" and when "the a_i are positive for all $i \in [k]$, the sequence $(a_1, a_2, ..., a_k)$ is a *composition* of n". And as defined by Page [21], a *restricted weak composition* is a subset of the set of weak compositions. "Restricted" means that there is a limited (restricted) range of values for each a_i. For our problem, when i out of n SVs succeed, the integer is i, and a restricted weak composition of i is of the form $(s_1, s_2, ..., s_m)$ where s_l is restricted by (7) with $l = 1, 2, ..., m$. To solve this problem, we employ the generalized algorithm for restricted weak composition generation proposed by Page [21]. The inputs of the algorithm are the restricted set of values for each element in the sequence representing weak compositions, the integer, and the number of parts of restricted weak compositions, and the output of the algorithm is a queue which contains all restricted weak compositions of the integer.

By the above algorithm, we can get the number of successful SVs s_{jl} ($l = 1, 2, ..., m$) on the lth PS in the jth scenario and calculate P_{c_j} with:

$$P_{c_j} = \prod_{l=1}^{m} P_{u_{jl}} \tag{9}$$

where $P_{u_{jl}}$ is the probability that s_{jl} SVs on the lth PS succeed in the jth scenario and can be calculated

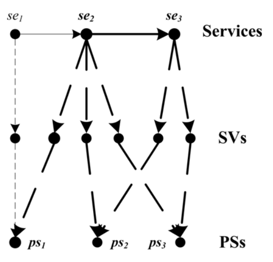


Fig. 3. sLDG of service se_2

according to the inner reliability of SVs and PSs. When $s_{jl} = 0$, $P_{u_{jl}}$ is the sum of the probability that the PS fails, $1 - r_{ps}$, and the probability that the PS succeeds while all n_l SVs hosted by it fail, $r_{ps}(1 - r_{sv})^{n_l}$. When $s_{jl} > 0$, $P_{u_{jl}}$ is the probability that the lth PS succeeds, r_{ps}, while s_{jl} out of n_l SVs hosted by the PS succeed, $(r_{sv})^{s_{jl}}$, and $(n_l - s_{jl})$ SVs hosted by the PS fail, $(1 - r_{sv})^{n_l - s_{jl}}$. Therefore, we get:

$$P_{u_{jl}} = \begin{cases} (1 - r_{ps}) + r_{ps}(1 - r_{sv})^{n_l}, & \text{if } s_{jl} = 0 \\ r_{ps}(r_{sv})^{s_{jl}}(1 - r_{sv})^{n_l - s_{jl}}, & \text{if } s_{jl} > 0 \end{cases} \tag{10}$$

Finally, combining (5), (6), (9), and (10), we can calculate the inner reliability of a service with:

$$r_{se} = \sum_{i=k}^{n} \sum_{j=1}^{d_i} \prod_{l=1}^{m} P_{u_{jl}} \tag{11}$$

When a service has service successors, its reliability can be calculated according to Formula (1). Based on the LDG model, we define the *sub-layered dependency graph (sLDG)* $G(se)$ of a service se as the subgraph of the LDG induced by the vertex set containing the service and its successors. An example of sLDG is shown in Figure 3, where an application consists of three services and $G(se_2)$ is bold. Whenever a service succeeds, there must be a number of successful PSs which form a non-empty subset of the set of PSs included in the service's sLDG. Therefore, we separate all scenarios where a service possibly succeeds according to the subset of PSs it needs. Then, the service reliability can be represented as the sum of the probability of each scenario:

$$R_{se} = \sum_{h=1}^{n} P_{ps_h} R'_{se_h} \tag{12}$$

where n is the number of non-empty subsets, P_{ps_h} is the probability that all PSs in the hth subset succeed, and R'_{se_h} is the probability that the service succeeds under the condition that all PSs in the hth subset succeed. Suppose U is the set of b PSs included in the sLDG of a service, then there are totally $2^b - 1$ non-empty subsets of U. For each non-empty subset U_h, $1 \leq h \leq 2^b - 1$, since PSs are independent with each other, P_{ps_h} can be calculated with:

$$P_{ps_h} = \prod_{ps \in U_h} R_{ps} \prod_{ps \in U/U_h} (1 - R_{ps}) \tag{13}$$

And R_{se_h} can be calculated as the product of the inner reliability of all services in the sLDG:

$$R_{se_h} = \prod_{l=1}^{q} r_{hl} \tag{14}$$

where q is the number of services in $G(se)$, r_{hi} is the inner reliability services when only PSs in U_h succeed. Suppose the ith service has n SVs while n' of them are hosted by the PSs in U_h and the service requires at least k SVs to succeed. Then, r_{hi} can be calculated with:

$$r_{hl} = \begin{cases} \sum_{i=k}^{n'} C_{n'}^{i} (r_{sv})^i (1 - r_{sv})^{n'-i}, & \text{if } n' \geq k \\ 0 & \text{if } n' < k \end{cases} \tag{15}$$

Finally, the service reliability is calculated by (12).

Application reliability. Assuming the application has only one service that has no predecessors, given the inner reliability of each component and the deployment stack, the application reliability can be assessed by the above process for the services with service successors.

IV. Evaluation

In this section, we introduce a reliability assessment a framework based on DEBRA and compare the quality of DEBRA against other representative methods.

A conceptual model of the framework was proposed in [19]. Within this work, we add two more components to the framework, provide a full implementation and extensively evaluate the model. The framework consists of five components: a dependency analyzer, a monitor, a reliability analyzer, a fault injector, and a tester. An overview of the framework is shown in Figure 4, where the deployment stack of a cloud application is divided into an application layer consisting of services, a VM layer, and a PS layer. The dependency analyzer analyzes dependencies between components in the deployment stack and creates a LDG. The monitor gathers states, particularly failures, of the components included in the LDG. The reliability analyzer assesses the reliability of the components and the application based on both the LDG and the reliability of components. The fault injector injects faults to components and also recovers them from the failure state. The tester generates and sends requests to the application.

We implemented the framework in Java for web applications deployed with Cloudify [22] on an OpenStack [23] cloud. PSs in the cloud are monitored by Ganglia [24]. The dependency analyzer, monitor, fault injector are implemented based on the REST APIs of Ganglia, Cloudify, and OpenStack. The tester is implemented using Selenium [25]. And the generalized algorithm for restricted weak composition generation used by the reliability analyzer is implemented according to the source code provided by Page [21]. The application under test is the website of the Software Engineering for Distributed Systems group at the Institute of Compute Science, University of Goettingen [26]. The website consists of three components: a load balancer dependent on a web server which depends on a database. Based on the layers of components in the LDG (from top to bottom) considered by a method, we choose and adapt the following methods to compare them with DEBRA:

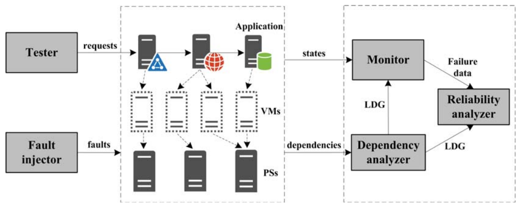


Fig. 4. Framework overview

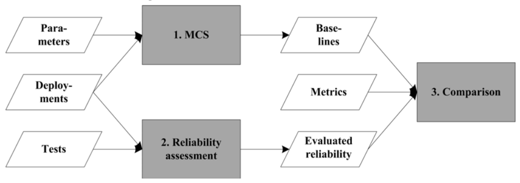
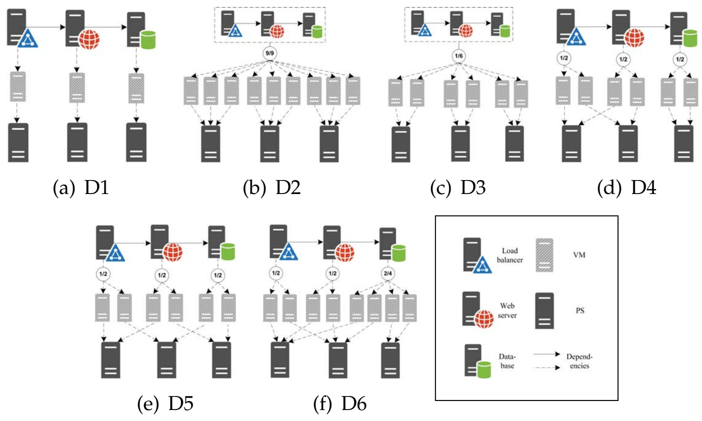


Fig. 5. The comparison process

- LBRA proposed by Banerjee et al. [15] assesses the application reliability based on the application failure rate without regard to its deployment structure or components failure rates, by $R_{app} = 1 - \frac{f_e}{n_e}$, where f_e and n_e are the number of failed and total log entries, respectively.
- HRM proposed by Wang et al. [13] evaluates the reliability of an N-redundant service, R_{se}, with $R_{se} = 1 - \prod_{i=1}^{N} (1 - R_{si_i})$ and a composite service, i.e., application, consisting of a series of K services by $R_{app} = \prod_{i=1}^{K} e^{-w_i \lambda_i}$, where λ_i is the failure rate of the ith service and w_i is the probability that the ith service is required.
- HCM proposed by Qiu et al. [10] evaluates the service reliability by $R_{se} = 1 - \prod_{m=1}^{K} p_m(0)$, where K is the number of PSs and $p_m(0)$ is the probability that no VM on the mth PS is available. HCM was designed only for the cloud service reliability rather than the application reliability, i.e., no application structure is considered. For comparison, we use the service reliability obtained by HCM as the input to DEBRA for calculating the application reliability.
- ARM proposed by Faragardi et al. [14] evaluates the cloud service reliability as the product of the reliability of every PS and the physical network link by $R_{app} = \prod_{i=1}^{n} R_{ps_i}$, where R_{ps_i} is the reliability of the ith PS and n is the number of PSs.
- HPARM proposed by Thanakornworakij et al. [9] is adapted to calculate the application reliability by $R_{app} = \prod_{i=1}^{n} r_i$, which means that the application



(a) D1 (b) D2 (c) D3 (d) D4 (e) D5 (f) D6

Fig. 6. Application deployment structures

reliability is the product of the inner reliability r_i of all n components in the deployment stack. HPARM is not developed for applications with redundancies.

A. Comparison Process

The comparison process involves three steps as shown in Figure 5. In the first step, we design several deployment structures of the application and, for each deployment, we obtain the baseline for comparison by a Mento Carlo Simulation (MCS) with the inner reliability of components. We assume that the daily (inner) failure rate of a PS is 0.001 while all VMs and SIs have the same daily inner failure rate of 0.03. The failure rate for the PS is based on results from previous research, which report failure rates between 0.0002 [27] and 0.022 [28]. The failure rate for the SI comes from measurements of the failure rate of a local Web service that provides our group Web site.

The employed deployment structures are shown in Figure 6. $D1$ assigns one SV to each service to evaluate the reliability of the most intuitive deployment structure. $D2$ is adapted from the structure proposed by Thanakornworakij et al. [9], which is originally designed for typical MPI applications. The application fails if more than one component fails. To adopt $D2$, we consider our application as a service and deploy it as a 9-out-of-9 system. $D3$ is adapted from a structure proposed by Qiu et al. [10], where a service uses six SVs evenly distributed on three PSs. The service fails if all SVs fail. The same to $D2$, we treat our application as a service and deploy it to six SVs. $D4$ is employed to improve the reliability of $D1$ by redundancy. To this aim, we deploy each service with two SVs as a 1-out-of-2 systems. $D5$ has exactly the same amount of SVs as $D4$, but with a different distribution of the SVs, where SVs of the same service are on different PS. $D6$ is also based on $D4$. The difference is that two more database SVs are added, and the four database SVs are organized as a 2-out-of-4 system.

In the second step, we deploy the application to the OpenStack cloud with $D1$ to $D6$, test the application, and assess its reliability. The assessment consists of 30 rounds, each of which is divided into 1,500 time units, each of which corresponds to one day in practice. A time unit starts with the determination of states of PSs, VMs, and SIs. Each component must be in success or failure state during a time unit. The probability of the success state of a component equals its inner reliability. After the determination, the time unit continues with the change of the states of components. If a component is determined to be successful, it will stay in the success state, or be recovered from the failure state to the success state. Otherwise, the component will keep the failure state or be injected with a fault if it is in the success state. When a component is in failure state, all its predecessors (except service components) will also be injected with faults. After the change of states, the time unit proceeds with testing the application with usage-based requests. On one hand, the tester sends ten requests to the application (for eliminating random errors) and logs the responses. On the other hand, the monitor logs the states of all components. When all time units finish, the reliability of services and the application for this round is calculated by DEBRA and other methods. When all rounds finish, the evaluated reliability $\overline{R_{eva}}$ is calculated with:

$$\overline{R_{eva}} = \frac{1}{n}\sum_{i=1}^{n} R_{eva_i} \tag{16}$$

where n is the number of rounds, R_{eva_i} is the ith round's application reliability.

In the third step, we compare the quality of methods with Mean Absolute Errors (MAEs) and Standard Deviations (SDs). First, we test if the results of a method are significantly different from the results of DEBRA. If a method is significantly different from DEBRA, we compare their accuracies measured by MAE; otherwise, we compare their precisions measured by SD. The smaller the MAE or the SD is, the better the accuracy or the

TABLE I
p-VALUES AND EFFECT SIZES

Dep.		LBRA	HRM	HCM	ARM	HPARM
D1	p	**0.245**	**0.607**	< 0.001	< 0.001	–
	ES	**0.150**	**0.068**	0.617	0.617	
D2	p	**0.910**	< 0.001	< 0.001	< 0.001	–
	ES	**0.015**	0.617	0.617	0.617	
D3	p	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
	ES	0.617	0.617	0.617	0.617	0.617
D4	p	**0.750**	< 0.001	< 0.001	< 0.001	< 0.001
	ES	**0.041**	0.543	0.617	0.617	0.617
D5	p	**0.781**	**0.558**	< 0.001	< 0.001	< 0.001
	ES	**0.036**	**0.076**	0.617	0.617	0.617
D6	p	**0.428**	< 0.001	< 0.001	< 0.001	< 0.001
	ES	**0.102**	0.617	0.617	0.617	0.617

precision is. MAE and SD are calculated respectively by:

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |R_{eva_i} - R_{ref}| \tag{17}$$

$$SD = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n} (R_{eva_i} - \overline{R_{eva}})^2} \tag{18}$$

where R_{ref} is the baseline obtained by MCS.

B. Comparison Results

With MCS, we get the baselines for $D1$ to $D6$: 0.83277, 0.33858, 0.99999, 0.98866, 0.98954, and 0.99214. For all methods to each deployment, we employ the two-tailed Wilcoxon signed-rank test [29] to test if their results are significantly different from the results of DEBRA, which corresponds to the null hypothesis H_0: "results are not significantly different from results of DEBRA", i.e., $\mu_d = 0$, where μ_d is the mean difference of the results of two methods, e.g., for LBRA:

$$\mu_d = \frac{1}{n}\sum_{i=1}^{n} (R_{LOB_i} - R_{DEB_i}) \tag{19}$$

where n is the number of rounds, and R_{LOB_i} and R_{DEB_i} are the evaluated reliability with LBRA and DEBRA for the ith round, respectively.

The p-value and Effect Size (ES) (i.e., correlation coefficient e) are shown in Table I. According to the Cohen's standard [30], small, medium, and large values of the correlation coefficient are 0.1, 0.3, and 0.5, respectively. With a 0.05 significance level, we can conclude that:

1) Results of LBRA for all deployments except $D3$ are not significantly different from DEBRA. The baseline for $D3$ is 0.99999, therefore, it's too short for a round with 1,500 time units to see an application failure. As a result, we conclude that results of LBRA and DEBRA are not significantly different.
2) Results of HRM for $D2$, $D4$, and $D6$ are significantly different from DEBRA, with $p < 0.001$, and large Effect Sizes (ESs) of 0.617, 0.543, and 0.617, respectively. Besides, results of HRM for $D1$, $D3$ (the same reason as LBRA), and $D5$ are not significantly different from DEBRA.
3) For HCM and ARM, results are significantly different from DEBRA, with $p < 0.001$ and a large ES of 0.617.
4) As to HPARM, for $D1$ and $D3$, DEBRA gets the same results as HPARM, and for $D3$ to $D6$, HPARM gets significantly different results from DEBRA, with $p < 0.001$ and a large ES of 0.617.

To compare the accuracy and precision of different methods, we calculate the MAEs and SDs, as shown in Table II, where all non-significantly different results are bold. Combining Table I and II, we can conclude that:

1) Results of LBRA are not significantly different from DEBRA, while the SDs of LBRA are larger (more than twice for $D4$ to $D6$) than that of DEBRA for all deployments, which manifests that considering deep dependencies can not significantly improve the accuracy because the failure rate of PSs are much lower than software components, and further considering the whole deployment stack can improve the precision.
2) Results of HRM for $D2$, $D4$, and $D6$ are significantly different from DEBRA, while MAEs are larger than that of DEBRA. So, DEBRA is more accurate than HRM for $D2$, $D4$, and $D6$. Besides, results of HRM for $D1$, $D3$, and $D5$ are not significantly different from DEBRA, and regarding SDs, the differences are too small (less than around 5%) to make conclusions. Therefore, further considering PS beside software components can improve the accuracy of the results at least in some cases.
3) Results of HCM and ARM are significantly different from DEBRA with larger MAEs for all deployments, which indicates that considering only PS and VM failures or only PS failures can not get accurate results.
4) Results of HPARM are the same as DEBRA for $D1$ and $D2$, so, HPARM is the same accurate and precise as DEBRA for them. Besides, results of HPARM for $D3$ to $D6$ are significantly different from DEBRA with much larger MAEs, which means that not considering redundancies, even involving all components, will get very inaccurate results while redundancies are present in the deployment stack.

In total, we conclude that deep dependencies have no significant influences to the accuracy and considering the application failures without regard to underlying structure details is sufficient to get accurate reliability assessment results at the cost of precision. Besides, modeling only a part of components or leaving out redundancies will lead to inaccurate results.

V. Conclusion

In this paper, we evaluated the influence of deep dependencies to the quality of reliability assessment methods. We first present DEBRA for assessing cloud

TABLE II
MAEs and SDs of results

Dep.	metrics	DEBRA	LBRA	HRM	HCM	ARM	HPARM
D1	MAE	**0.00010**	**0.00045**	**0.00088**	0.07690	0.16446	**0.00010**
	SD	**0.00962**	**0.01058**	**0.01013**	0.00827	0.00159	**0.00962**
D2	MAE	**0.00197**	**0.00251**	0.66142	0.65818	0.65827	**0.00197**
	SD	**0.00967**	**0.01466**	< 0.00001	0.00170	0.00169	**0.00967**
D3	MAE	< 0.00001	< 0.00001	< 0.00001	0.00537	0.00271	0.51323
	SD	< 0.00001	< 0.00001	< 0.00001	0.00120	0.00115	0.01009
D4	MAE	< 0.00001	**0.00030**	0.00081	0.00757	0.00837	0.29385
	SD	**0.00111**	**0.00244**	0.00101	0.00075	0.00166	0.01258
D5	MAE	**0.00026**	**0.00041**	**0.00024**	0.00765	0.00744	0.29058
	SD	**0.00086**	**0.00233**	**0.00084**	0.00036	0.00133	0.01053
D6	MAE	**0.00027**	**0.00012**	0.00111	0.00519	0.00540	0.36982
	SD	**0.00076**	**0.00264**	0.00264	0.00037	0.00105	0.01206

application reliability taking deep dependencies between components into account. Then, we implemented a framework based on DEBRA and conducted an experiment to compare the quality of DEBRA against other representative methods. The results show that deep dependencies hardly influence the accuracy of reliability methods because of the much lower failure rate of PSs than software components, and DEBRA can get more precise results because of the synthetic consideration of components and dependencies. In the future, we are going to utilize DEBRA for reliability improvement based on the assessment results.

References

[1] M. Armbrust, I. Stoica, M. Zaharia, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, and A. Rabkin, "Above the Clouds: A Berkeley View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, p. 50, apr 2010.

[2] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "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, jun 2009.

[3] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[4] E. Bauer and R. Adams, *Reliability and availability of cloud computing*. John Wiley & Sons, 2012.

[5] R. Buyya, R. Ranjan, and R. N. Calheiros, "InterCloud : Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services," in *Proceedings of the 10th international conference on Algorithms and Architectures for Parallel Processing*, vol. 6081, no. 1, 2010, pp. 13–31.

[6] Z. Zheng, T. C. Zhou, M. R. Lyu, and I. King, "Component ranking for fault-tolerant cloud applications," *Services Computing, IEEE Transactions on*, vol. 5, no. 4, pp. 540–550, 2012.

[7] R. Wang, Y. Zhang, S. Liu, L. Wu, and X. Meng, "A Dependency-Aware Hierarchical Service Model for SaaS and Cloud Services," in *2011 IEEE International Conference on Services Computing*. IEEE, Jul 2011, pp. 480–487.

[8] S. Malkowski, Y. Kanemasa, H. Chen, M. Yamamoto, Q. Wang, D. Jayasinghe, C. Pu, and M. Kawaba, "Challenges and opportunities in consolidation at high resource utilization: Non-monotonic response time variations in n-tier applications," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 162–169.

[9] T. Thanakornworakij, R. F. Nassar, C. Leangsuksun, and M. Păun, "A reliability model for cloud computing for high performance computing applications," in *Euro-Par 2012: Parallel Processing Workshops*. Springer, 2012, pp. 474–483.

[10] X. Qiu, Y. Dai, Y. Xiang, and L. Xing, "A Hierarchical Correlation Model for Evaluating Reliability, Performance, and Power Consumption of a Cloud Service," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2015.

[11] A. Mohammad, B. John, and T. Ingo, "An analysis of the cloud computing security problem," in *APSEC 2010 Proceeding-Cloud Workshop*, 2010.

[12] B. Ford, "Icebergs in the clouds: The other risks of cloud computing." in *HotCloud*, 2012.

[13] L. Wang, X. Bai, L. Zhou, and Y. Chen, "A Hierarchical Reliability Model of Service-Based Software System," in *2009 33rd Annual IEEE International Computer Software and Applications Conference*. IEEE, 2009, pp. 199–208.

[14] H. R. Faragardi, R. Shojaee, H. Tabani, and A. Rajabi, "An analytical model to evaluate reliability of cloud computing systems in the presence of QoS requirements," *IEEE/ACIS 12th International Conference on Computer and Information Science*, pp. 315–321, 2013.

[15] S. Banerjee, H. Srikanth, and B. Cukic, "Log-Based Reliability Analysis of Software as a Service (SaaS)," *IEEE 21st International Symposium on Software Reliability Engineering*, pp. 239–248, 2010.

[16] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *Dependable and Secure Computing, IEEE Transactions on*, vol. 7, no. 4, pp. 337–350, 2010.

[17] IEEE Standards Board, "Systems and software engineering – vocabulary," *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, Dec 2010.

[18] W.-L. Wang, D. Pan, and M.-H. Chen, "Architecture-based software reliability modeling," *Journal of Systems and Software*, vol. 79, no. 1, pp. 132–146, 2006.

[19] X. Wang and J. Grabowski, "A reliability assessment framework for cloud applications," *CLOUD COMPUTING 2015*, pp. 127–130, 2015.

[20] M. Bóna, *A walk through combinatorics: an introduction to enumeration and graph theory*. World scientific, 2011.

[21] D. R. Page, "Generalized algorithm for restricted weak composition generation," *Journal of Mathematical Modelling and Algorithms in Operations Research*, vol. 12, no. 4, pp. 345–372, 2013.

[22] (2017, Jan.) Cloudify. [Online]. Available: http://www.getcloudify.org/

[23] (2017, Jan.) OpenStack. [Online]. Available: https://www.openstack.org/

[24] (2017, Jan.) Ganglia. [Online]. Available: http://ganglia.sourceforge.net/

[25] (2017, Jan.) Selenium. [Online]. Available: http://www.seleniumhq.org/

[26] (2017, Jan.) Software Engineering for Distributed Systems Group. [Online]. Available: https://www.swe.informatik.uni-goettingen.de/

[27] J. Dean, "Software engineering advice from building large-scale distributed systems," *CS295 Lecture at Stanford University, July*, 2007.

[28] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*. New York, New York, USA: ACM Press, 2010, p. 193.

[29] D. C. Montgomery and G. C. Runger, *Applied statistics and probability for engineers*. John Wiley & Sons, 2010.

[30] J. Cohen, "A power primer." *Psychological bulletin*, vol. 112, no. 1, p. 155, 1992.