Reliability Assessment for Cloud Applications

Dissertation
zur Erlangung des Doktorgrades
Dr. rer. nat.
der Mathematisch-Naturwissenschaftlichen Fakultäten
der Georg-August-Universität zu Göttingen

im PhD Programme in Computer Science (PCS)
der Georg-August University School of Science (GAUSS)

vorgelegt von

Xiaowei Wang
aus Shandong, China

Göttingen, im 2016

Betreuungsausschuss:	Prof. Dr. Jens Grabowski, Georg-August-Universität Göttingen
	Prof. Dr. Xiaoming Fu, Georg-August-Universität Göttingen
Prüfungskommission:	
Referent:	Prof. Dr. Jens Grabowski, Georg-August-Universität Göttingen
Korreferenten:	Prof. Dr. Ramin Yahyapour, Georg-August-Universität Göttingen
Weitere Mitglieder der Prüfungskommission:	Jun.-Prof. Dr. Marcus Baum Georg-August-Universität Göttingen
	Prof. Dr. Carsten Damm, Georg-August-Universität Göttingen
	Prof. Dr. Xiaoming Fu, Georg-August-Universität Göttingen
	Prof. Dr. Dieter Hogrefe, Georg-August-Universität Göttingen

Tag der mündlichen Prüfung:	11. Januar 2017

Abstract

Reliability is a significant quality measurement for computer systems and applications. When cloud computing is becoming mature and pervasive, a variety of applications are deployed on cloud platforms. Influenced by the characteristics of cloud computing, such as server consolidation and virtualization, and reliability requirements, such as redundancies, the deployment structure of cloud applications are complex. A deployment structure usually involves components like services, Virtual Machines (VMs), Physical Servers (PSs), etc. Dependencies of these components make the reliability assessment of cloud applications challenging.

In this thesis, we intend to model cloud applications based on their deployment structures and give an accurate reliability assessment method. To this aim, we propose a DEpendency-Based Reliability Assessment (DEBRA) and accordingly design a framework composed of three functional components: a dependency analyzer for analyzing dependencies between components used by applications and modeling cloud applications with Layered Dependency Graphs (LDGs), a monitor for deriving states of application components, and a reliability analyzer based on DEBRA for assessing the reliability of components as well as applications. Furthermore, we implement the three components and two extra functional components for usage-based testing, which are a fault injector for injecting failures to application components and a tester for testing applications.

We apply our framework to a real-world application and cloud platforms, and conduct case studies. In these case studies, we verify if our framework can assess the cloud application reliability accurately and precisely. We deploy the application with several different structures to a cloud and test the application with usage-based requests. We then use DEBRA and several existing methods to assess the application reliability. The assessment results are compared according to a proposed comparison process. The comparison results show that DEBRA can obtain results of high quality and has several merits regarding modeling cloud applications for reliability assessment.

Acknowledgements

I have been fortunate to work with many persons. Without their kind help, this thesis would never have been possible. First of all, I deeply thank my supervisor Prof. Dr. Jens Grabowski. I feel lucky to have such a nice supervisor as him during the time of pursuing a doctoral degree. It was his constant encouragement, guidance, and support that allowed me to conduct my research under excellent conditions. His valuable suggestions and feedback made this thesis possible.

In addition, I am indebted to my second supervisor Prof. Dr. Xiaoming Fu. He gave me many good advices. I would also like to thank Prof. Ramin Yahyapour for agreeing to be a referee for this thesis. I also thank the remaining defense committee members: Jun.-Prof. Dr. Marcus Baum, Prof. Dr. Carsten Damm, and Prof. Dr. Dieter Hogrefe.

Furthermore, I want to thank all my former and current colleagues at our research group. The discussions and collaborations with them helped me to make this thesis better. I'm especially grateful to Thomas Rings for helping me both in research and in life during the first two years after I joined our group. I also greatly thank Fabian Glaser, Michael Göttsche, Patrick Harms, and Steffen Herbold for proof reading this thesis. Their comments were always helpful to improve this thesis.

Last but not least, I want to thank my family and friends. Their unconditional and considerate support was the biggest motivation for me to finish this work.

Contents

Chapter 1

Introduction

Reliability is one of the crucial non-functional requirements for measuring quality of computer systems quantitatively. With the advance of Information Technology (IT), computer systems become increasingly more complex and suffer more possible failures consequently. Failures may lead to severe losses, e.g., failures of commercial computer systems may cause data loss, reputation loss, revenue loss, etc.; failures of aircrafts may even lead to mortal dangers [1, p. 5]. Failures need to be prevented, identified, corrected, and handled to achieve high reliability. However, higher is not always better, because higher reliability usually means higher cost. The level of reliability must be determined according to cost-benefit analyses, which need reliability to be assessed before systems are provided to users.

Reliability is always one of the major concerns of developers and consumers along with the rapid evolution of distributed computing systems in recent decades. Following cluster computing, peer-to-peer computing, grid computing, and utility computing, cloud computing has been attracting much attention and exerting influences over our daily lives. After several years of development, cloud computing now basically lives up to the promise of providing computing resources and services as utilities [2, 3]. Clouds can provide seemingly infinite resources to consumers by means of resource pooling and rapid elasticity [4]. While employing cloud computing to reduce the purchase and maintenance cost of hardware in traditional 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 cloud computing's adoption [2].

One typical use case scenario of cloud computing is that consumers deploy applications on clouds (i.e., data centers including hardware and software [2]) and provide services to end users. In this scenario, the multiple failure causes, such as hardware (Physical Servers (PSs), network devices, etc.), software (management software, hypervisors, Virtual Machines (VMs), applications, etc.), infrastructure of data centers (e.g., public networks, power supplies, cooling systems, etc.), and different deployment structures make it challenging to accurately assess the reliability of cloud applications.

1.1 Motivation

Rigid failure containment of software failures at the VM level provided by virtualization [5, p. 93] and the service-oriented architecture of cloud computing make it intuitive to model cloud applications with component-based architectures [6, 7], more specifically, in a hierarchical manner [8–10]. Typically, a cloud application is divided into services which are deployed to VMs hosted by PSs. Services, VMs, PSs, and other hardware and software components, e.g., routers and hypervisors, comprise the deployment stack of an application. Ideally, all components of the deployment stack should be considered for reliability analysis. However, with different emphases, many works involve only a part of components, such as only hardware [11], only PSs and VMs [12], only services [13], etc., which are not enough for synthetically assessing the reliability of cloud applications. Therefore, we endeavor to design a reliability assessment method for cloud applications by modeling the whole deployment stack in order to determine if it is sufficient to model specific parts of the deployment stack for reliability assessment or if the whole deployment stack must be considered.

There are also works [10, 14] that consider relatively comprehensive components while overlooking some dependencies between components. Similar to other combined software-hardware systems, components in the deployment stack are also subject to failure propagation, and characteristics of cloud computing aggravate it . For instance, to use PSs efficiently so as to reduce the number of servers required by an organization [15], VMs are often consolidated into one PS whose failures may lead to common cause failures [16] of all VMs on it. Failure propagation caused by the dependency between VMs and PSs is evident and well researched [12]. However, the dependency between services and PSs is nontransparent and often not considered for reliability assessment [17]. "Hardware can fail, and reliability should come from the software" [18] has nearly been a consensus in the context of cloud computing. This requires the tolerance of hardware failures in upper layers of the deployment stack. The most widely used fault tolerance technique is redundancy [19] and in a typical usage scenario of cloud computing, a service is usually deployed with several identical instances as redundancies. But due to server consolidation or limited control of the deployment process, instances of the same service may be deployed on the same PS. In this case, failures of a PS may crash services and then the whole application. Furthermore, redundant instances can be organized in different manners, e.g., one service may require at least one available instance to work and another service may require at least k, $k > 1$, available instances to ensure a certain level of performance, which needs the instances to be configured as k-out-of-n redundancies to each other. Different configurations of Service Instances (SIs) lead to different dependencies between the service and PSs and also different service reliability. The normal redundancy requiring at least one available instance is very often considered in the context of reliability assessment for services and cloud applications [12, 20], while the k-out-of-n redundancy is not. Hence, we propose DEpendency-Based Reliability Assessment (DEBRA) considering the influences of dependencies between components and configurations of redundant SIs to the application's reliability.

DEBRA is intended to be appropriate for both before and after the deployment of cloud applications. Before the deployment, the reliability of components in the deployment stack can be gathered from the cloud provider and DEBRA can be directly utilized to assess the application reliability based on artificial settings. After the deployment, the reliability of components may vary with different settings in different systems. In this case, field reliability needs to be obtained by monitoring and dependencies between components need to be analyzed according to the deployment stack. Therefore, based on DEBRA, we intend to develop a framework which can assess the cloud application reliability not only with artificial information but also by gathering field data and dependencies.

Reliability assessment methods for cloud applications usually have different assumptions and usage scenarios. For example, as aforementioned, methods may model an application with different set of components, and redundancies and dependencies may be considered or not. Therefore, it is challenging to evaluate and compare the quality of different reliability assessment methods. The challenge is twofold. On one hand, proper baselines and metrics are required to determine the quality of methods. On the other hand, methods need to be compared with both the baselines and other methods after the adaption based on unified assumptions. To tackle this problem, we intend to develop a comparison process which provides an approach of getting baselines and a method to determine the quality of reliability assessment methods and compare different methods based on quality metrics.

1.2 Goals and Contributions

Motivated by the above challenges of the reliability assessment for cloud applications, we answer the following research questions in this thesis:

- RQ1: What aspects should be considered when modeling cloud applications for reliability assessment?
- RQ2: How do dependencies affect the accuracy of a reliability assessment?
- RQ3: How can the quality of reliability assessment methods be compared?

Based on the above research questions, the goals of this thesis are as follows: 1) to design a model of cloud applications and assess the reliability of cloud applications considering components in the deployment stack and the dependencies between them, 2) to develop a framework to support the reliability assessment method, and 3) to define criteria and devise a process to evaluate the quality of reliability assessment methods and compare the quality of different reliability assessment methods.

The main contributions of this thesis are:

- DEBRA for component-based cloud applications (Section 4.4). DEBRA combines the reliability of individual components, dependencies, and the configurations of redundant SIs to assess the reliability of services as well as applications.
- A reliability assessment framework (Chapter 4) that includes three components. A dependency analyzer extracts the dependencies between cloud components. A monitor collects state data of cloud components. A reliability analyzer assesses the reliability based on the dependencies and state data.
- An instantiation of the proposed framework and a usage-based reliability testing tool for web-based applications on real-world clouds (Chapter 5). The three components of the framework are implemented for real-world cloud platforms. Besides, a usage-testing reliability tool includes a fault injector to inject failures to components and a tester to create usage-based test cases, send test cases to the application and collect testing results are implemented for web-based applications.
- A case study for evaluating and comparing the quality (accuracies and precisions) of reliability assessment methods for cloud applications (Chapter 6). The comparison process includes three steps: 1) get required accurate baselines for comparison by simulation, 2) assess the reliability based on field data, and 3) evaluate and compare the quality of different methods.

1.3 Structure of the Thesis

The rest of the thesis is organized as follows. In Chapter 2, we introduce basic terms used in this thesis. At first, we present definitions related to reliability engineering (Section 2.1). And then, we introduce the definition, service models, and deployment models of cloud computing, and two open source cloud platform software employed in this thesis (Section 2.2). In Chapter 3, we discuss related work about reliability assessment in the context of cloud computing. Thereby, we consider works about the reliability of cloud hardware (Section 3.1), systems (Section 3.2), and applications (Section 3.3), respectively. In Chapter 4, we introduce a DEBRA-based framework for reliability assessment of cloud applications and details of its components. We describe the dependency analyzer (Section 4.2), the monitor (Section 4.3), and the reliability analyzer based on DEBRA (Section 4.4), respectively. In Chapter 5, we illustrate the implementation of the framework proposed in Chapter 4. Beside the implementation of the components of the framework (Section 5.1, 5.2, and 5.5), we illustrate the implementation of two more components for reliability testing: the fault injector (Section 5.3) and the tester (Section 5.4), respectively. In Chapter 6, we state the case studies conducted for verifying the proposed framework and comparing DEBRA with related methods. At first, we state the setup, the platform, and the application utilized in the case studies (Section 6.1 and 6.2). Then, we state the comparison results (Section 6.3) and discuss them (Section 6.4). In Chapter 7, we answer the research questions listed in Section 1.2 (Section 7.1) and discuss strengths, limitations, and threats to validity of the work proposed in this thesis (Section 7.2). In Chapter 8, we summarize the thesis, and give conclusions (Section 8.1) and possible directions for the future work (Section 8.2).

Chapter 2

Background

In this chapter, we introduce the background knowledge required for this work. In Section 2.1, we provide concepts and definitions about reliability engineering. Afterwards, in Section 2.2, we define cloud computing and its service models and deployment models, and describe the employed cloud platforms in this thesis.

2.1 Reliability Engineering

Reliability engineering was scientifically established in the mid-1950s driven by military efforts [21] and is now indispensable to the quality assurance of products during their lifecycles. It has a broad connotation, but no unified definition. In this thesis, we employ the definition from Fuqua [22, p. 7] that *reliability engineering* is "the technical discipline of estimating, controlling, and managing the probability of failure in devices, equipment, and systems". More specifically, as stated by Kececioglu [23, p. 2], "reliability engineering provides the theoretical and practical tools whereby the probability and capability of parts, components, equipment, products, and systems to perform their required functions for desired periods of time without failure, in specified environments and with a desired confidence, can be specified, predicted, designed in, tested, demonstrated, packaged, transported, stored, installed, and started up, and their performance monitored and fed back to all concerned organizations". Currently, the widely accepted objectives of reliability engineering are:

- to utilize engineering knowledge and techniques to reduce the probability of failures;
- to determine and eliminate the causes of failures;
- to tolerate unhandled failures; and
- to evaluate and predict the reliability [24, p. 2].

To achieve the four objectives, four corresponding methods can be used: *fault prevention*, *fault removal*, *fault tolerance*, and *fault forecasting* [1, p. 19]. Fault prevention is used to prevent faults from being introduced into the system by using, e.g., mature design and development techniques. Fault removal is used to detect and remove faults during development and maintenance by verification and validation. Fault tolerance is used to tolerate faults when occurring by, e.g., redundancy. Fault forecasting is used to estimate and predict the probability of occurrences of faults by evaluation.

Fundamentals of reliability engineering, which focuses on reliability, have been well developed based on the probability and statistical theory. *Reliability*, as defined by the IEEE, is “the ability of a system or component to perform its required functions under stated conditions for a specified period of time” [25]. In the following, we illustrate the mathematical fundamentals of reliability based on the definitions and equations proposed by Lyu [1].

Using probability theory, the reliability at a point of time t can be described as the probability that the *time to failure* T is larger than t. Given an event that an item is reliable till a point of time $T \in [t, t+\Delta t]$, where Δt means a period of time, the probability of the event $P(t \leq T \leq t+\Delta t)$ can be expressed as:

$$P(t \leq T \leq t+\Delta t) = f(t)\Delta t = F(t+\Delta t) - F(t) \tag{2.1.1}$$

where $f(t)$ is the Probability Density Function (PDF) and $F(t)$ is the Cumulative Distribution Function (CDF) of T. Since $T \geq 0$ and $f(t) = dF(t)/dt$, we can derive:

$$F(t) = P(0 \leq T \leq t) = \int_0^t f(x)dx \tag{2.1.2}$$

From (2.1.2), we can calculate the reliability at time t by:

$$R(t) = P(T > t) = 1 - F(t) = \int_t^\infty f(x)dx \tag{2.1.3}$$

Except for itself, reliability can also be measured by:

- ***Failure rate***. As defined by the IEEE, failure rate is "the ratio of the number of failures of a given category to a given unit of measure; for example, failures per unit of time, failures per number of transactions, failures per number of computer runs" [25]. If we take time as the measure, mathematically, the failure rate can be defined as "the probability that a failure per unit time occurs in the interval, say, $[t, t+\Delta t]$, given that a failure has not occurred before t" [1, p. 752]. Then, the failure rate, $h(t)$, can be calculated by:

$$\begin{aligned} h(t) &= \frac{P(t \leq T \leq t+\Delta t | T \geq t)}{\Delta t} = \frac{P((t \leq T \leq t+\Delta t) \cap (T \geq t))}{\Delta t P(T \geq t)} \\ &= \frac{P(t \leq T \leq t+\Delta t)}{\Delta t P(T \geq t)} = \frac{F(t+\Delta t) - F(t)}{\Delta t R(t)} = \frac{f(t)}{R(t)} = \frac{f(t)}{1-F(t)} \end{aligned} \tag{2.1.4}$$

- ***Mean Time To Failures (MTTF)***. MTTF is usually used for measuring the reliability of non-repairable systems or components and defined as "the expected life, or the expected time during which the system will function successfully without maintenance or repair" [1, p. 756]. For reparable systems or components, their reliability can be measured by *Mean Time Between Failures (MTBF)* which is defined as "the expected or observed time between consecutive failures in a system or component" [25]. Based on the definition, the MTTF can be calculated with [1, p. 757]:

$$MTTF = E(T) = \int_0^\infty t f(t) dt \tag{2.1.5}$$

where $E(T)$ is the expected time to failure.

2.1.1 Failure Distributions

In reliability engineering, it's useful to study failure data of an item or component and determine the distribution of the time to failure by providing a PDF. With the PDF, we can then derive the CDF of failures and calculate the reliability, the failure rate, and the MTTF with formulas introduced in Section 2.1. In the following, we introduce two frequently used distributions in the context of reliability engineering using equations proposed by Trivedi [26].

One widely-used distribution is the exponential distribution where the occurrence of failures (time to failure) is modeled by an one-parameter exponential distribution. The PDF and CDF of the one-parameter exponential distribution are:

$$f(t) = \lambda e^{-\lambda t} \tag{2.1.6}$$

$$F(t) = 1 - e^{-\lambda t} \tag{2.1.7}$$

where λ is a constant. Correspondingly, the failure rate, the MTTF, and the reliability are:

$$h(t) = \frac{f(t)}{1-F(t)} = \frac{\lambda e^{-\lambda t}}{1-(1-e^{-\lambda t})} = \frac{\lambda e^{-\lambda t}}{e^{-\lambda t}} = \lambda \tag{2.1.8}$$

$$MTTF = \int_0^\infty t f(t) dt = \int_0^\infty t \lambda e^{-\lambda t} dt = \frac{1}{\lambda} \tag{2.1.9}$$

$$R(t) = 1 - F(t) = 1 - (1 - e^{-\lambda t}) = e^{-\lambda t} \tag{2.1.10}$$

Equation (2.1.8) shows that the failure rate is constant. As failures of software and combined software-hardware systems are often considered random and with constant failure rates, they are often modeled by exponential distributions. The trends over time of the PDF, the CDF, the failure rate, and the reliability of an exponential distribution with $\lambda = 0.9$ are shown in Figure 2.1.

Another significant distribution is the two-parameter Weibull distribution which is the most popular distribution for modeling failures of an individual hardware item [27]. The

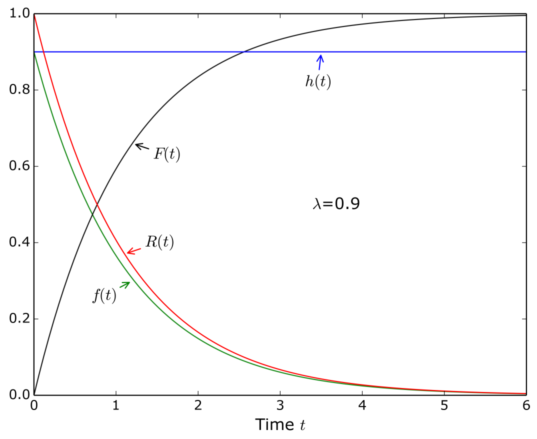


Figure 2.1: An example of the exponential distribution.

PDF and CDF of the two-parameter Weibull distribution are:

$$f(t) = \frac{\beta}{\eta}(\frac{t}{\eta})^{\beta-1}e^{-(\frac{t}{\eta})^{\beta}} \tag{2.1.11}$$

$$F(t) = 1 - e^{-(\frac{t}{\eta})^{\beta}} \tag{2.1.12}$$

where β is the shape parameter and η is the scale parameter. By adjusting β and η, reliability engineers can model the reliability of hardware items in different phases of lifetime.

Under the assumption of the Weibull reliability model, the failure rate, the MTTF, and the reliability of a hardware item are:

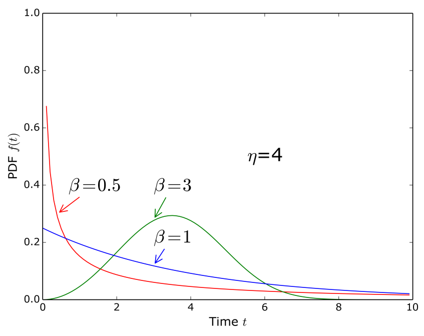
$$h(t) = \frac{f(t)}{1-F(t)} = \frac{\frac{\beta}{\eta}(\frac{t}{\eta})^{\beta-1}e^{-(\frac{t}{\eta})^{\beta}}}{1-1+e^{-(\frac{t}{\eta})^{\beta}}} = \frac{\beta}{\eta}(\frac{t}{\eta})^{\beta-1} \tag{2.1.13}$$

$$MTTF = \int_0^{\infty} tf(t)dt = \eta\Gamma(\frac{1}{\beta}+1) \tag{2.1.14}$$

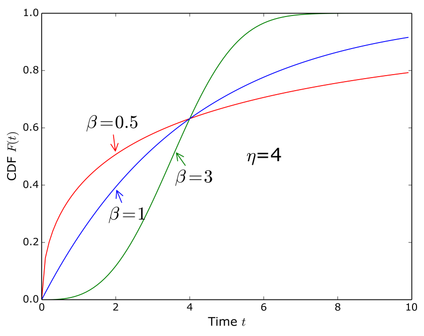
$$R(t) = 1 - F(t) = e^{-(\frac{t}{\eta})^{\beta}} \tag{2.1.15}$$

where $\Gamma(n) = \int_0^{\infty} e^{-x}x^{n-1}dx$. The trends over time of PDFs, CDFs, failure rates, and reliability of Weibull distributions with η=4 and β=0.5, 1, and 3 respectively are shown in Figure 2.2. Figure 2.2 and (2.1.11) to (2.1.15) show that, when β=1, the Weibull distribution reduces to an exponential distribution with $\lambda = 1/\eta$.

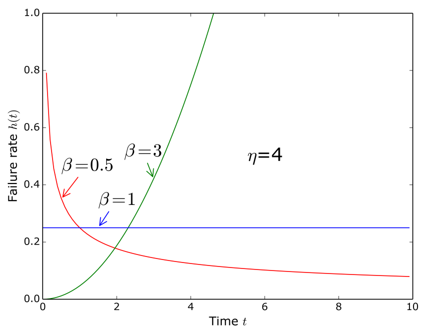
Hardware reliability is time-varying because of infant mortality, random faults, and wear-out during the lifetime. The failure rate of a population of hardware items in different phases



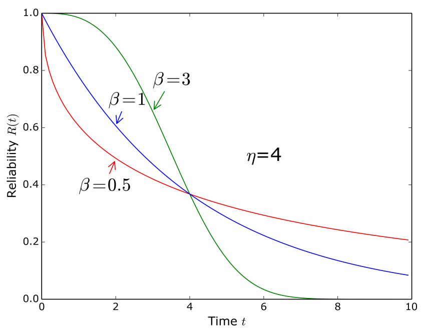
(a) PDFs



(b) CDFs



(c) Failure rate



(d) Reliability

Figure 2.2: PDFs, CDFs, failure rates, and reliability of Weibull distributions.

of the lifetime can also be modeled by Weibull distributions with different parameters. Comprehensively, the lifetime of hardware can be described by a bathtub curve [28], as shown in Figure 2.3.

As Figure 2.3 shows, the hardware lifetime can be divided into three phases from the beginning to the end:

1. ***Infant mortality***. Failures are mostly burn-in, therefore, the failure rate is decreasing with time and $0 < \beta < 1$.

2. ***Useful life***. Failures are mostly arbitrary, therefore, the failure rate is relatively stable and $\beta = 1$. As aforementioned, failures in this phase can be modeled by the exponential distribution with $\lambda = 1/\eta$.

3. ***Wear-out***. Failures are mostly due to wear-out, therefore, the failure rate is increasing with time and $\beta > 1$.

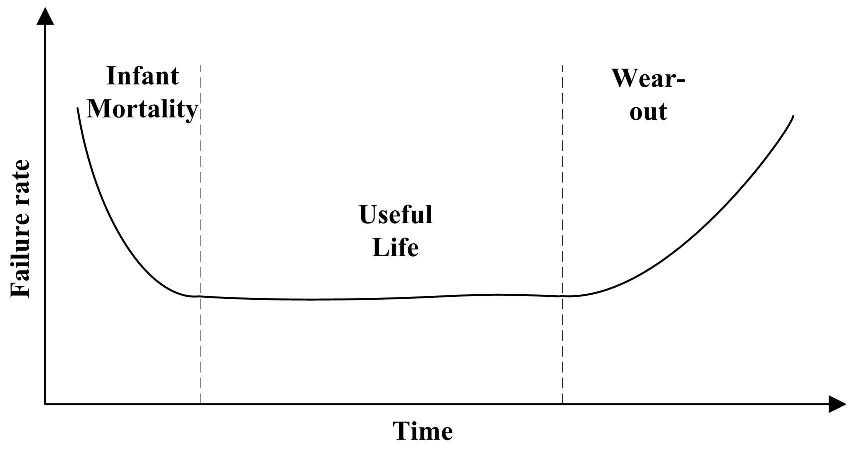


Figure 2.3: An example of bathtub curve [28].

2.1.2 Reliability Modeling

If system components are independent and their reliability are given, system reliability can be assessed using Reliability Block Diagram (RBD) [24] according to the component reliability and the system structure. In the following, we give the modeling and reliability of systems with different structures based on the description from O'Connor and Kleyner [24].

2.1.2.1 Series System

A series system will fail when any one of its components fails. An example of a series system with two components is shown in Figure 2.4.

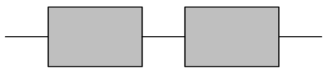


Figure 2.4: An example of series system.

Assuming that the reliability of the two components are R_1 and R_2, respectively, the system reliability (R) can be calculated with $R = R_1R_2$. Similarly, the reliability of a series system with n independent components can be calculated with:

$$R = \prod_{i=1}^{n} R_i \tag{2.1.16}$$

where R_i is the reliability of the ith component.

2.1.2.2 Parallel System

A parallel system succeeds when at least one of its components succeeds. An example of a parallel system with two components is shown in Figure 2.5.

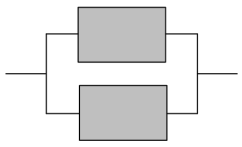


Figure 2.5: An example of parallel system.

Assuming that the reliability of the two components are R_1 and R_2, respectively, the system reliability (R) can be assessed with $R = 1-(1-R_1)(1-R_2)$, and the reliability of a parallel system with n independent components can be assessed with:

$$R = 1 - \prod_{i=1}^{n}(1-R_i) \tag{2.1.17}$$

where R_i is the reliability of the ith component.

2.1.2.3 K-out-of-n System

A k-out-of-n redundant system ($1 \leq k \leq n$) is a system composed of n components, each of which is redundant to others, and the system succeeds only if at least k components succeed. Assume that all components of a k-out-of-n system are independent and identical (with the reliability of R), then the system reliability R_{sys} can be calculated with:

$$R_{sys} = \sum_{i=k}^{n} C_n^i R^i (1-R)^{n-i} \tag{2.1.18}$$

where C_n^i is the number of i-combinations from a set of n elements [1].

2.2 Cloud Computing

In this section, we introduce the basic definition and characteristics of cloud computing and cloud platforms used in this thesis.

Cloud computing has been increasingly more popular in recent years and wildly used as a daily utility. Public clouds provide seemingly infinite virtualized resources (e.g., computing capacity, network bandwidth, storage, etc.) as services via the Internet to users. Different from traditional computing systems, public clouds do not require users to maintain their

[1]The i-combinations from a set of n elements is often denoted by other forms, like $C(n,i)$, $\binom{n}{i}$, etc.

own in-house hardware [4]. Besides, organizations and businesses can also build private or community clouds or cloud platforms using open source cloud software (e.g., OpenStack [29], Helion [30], OpenShift [31], Cloudify [32]). A survey report from RightScale in 2016 [33] shows that, 95 percent of their respondents from various organizations are utilizing cloud infrastructure for deploying and maintaining applications. And the market size of cloud computing, as discussed in a report from the U.S. Department of Commerce in 2016 [34], is from dozens to hundreds of billions over the next years. Nearly all big IT companies are providing cloud services, such as Amazon Web Services (AWS) [35], Google Compute Engine [36], Microsoft Azure [37], etc. In the meantime, cloud computing affects our daily life in many ways. For instance, iCloud [38] is used by most iPhone users to manage files.

Cloud computing, as a term in the context of utility computing, was firstly put forward by Eric Schmidt from Google in 2006 [39]. At nearly the same time, Amazon announced its Elastic Compute Cloud (EC2) service [40]. Till now, there are several definitions of cloud computing rather than a unanimous one. In this thesis, we use the definition from the National Institute of Standards and Technology (NIST) of America, which defines cloud computing as "*a model for enabling ubiquitous, convenient, on demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*" [4].

From the definition, the five essential attributes of cloud computing are:

- **On-demand self-service**. Users can utilize cloud capabilities on their own according to individual requirements, without professional assistance of service providers.
- **Broad network access**. Capabilities can be used via the network and be accessed by various client devices, such as laptops, mobile phones, servers, etc.
- **Resource pooling**. Computing resources, e.g., storage, processing and network bandwidth, are transformed into pools, which make underlying resources transparent for users. Resource pools are assigned and reassigned dynamically by cloud systems based on the user demand.
- **Rapid elasticity**. Capabilities can be scaled by provisioning and recycling elastically according to the consumer's demand, and are seemingly infinite to the consumer.
- **Measured service**. Resources are managed and used in a manner of pay-per-use at an abstraction level [4].

With the support of hardware (PSs, power systems, cooling systems, etc.) and kernel software, cloud computing systems can be deployed to provide three kinds of service: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [4] from bottom to top, as shown in Figure 2.6. IaaS provides fundamental computing resources, e.g., compute cores, memories, storage spaces, and network bandwidth, as services to consumers. Consumers can utilize these resources to install operating systems, store data and deploy applications. *PaaS* provides the runtime environment and tools, e.g., programming languages, monitoring tools, scaling tools, and security tools, as services for consumers to deploy and manage applications. Consumers have the authority for the environment and tools, but neither for fundamental resources nor for operating systems. And *SaaS* provides various applications, e.g., data processing applications, business management applications, and application management applications, as web services to consumers. Consumers can only use the services without control of applications or the underlying platform and infrastructure [4].

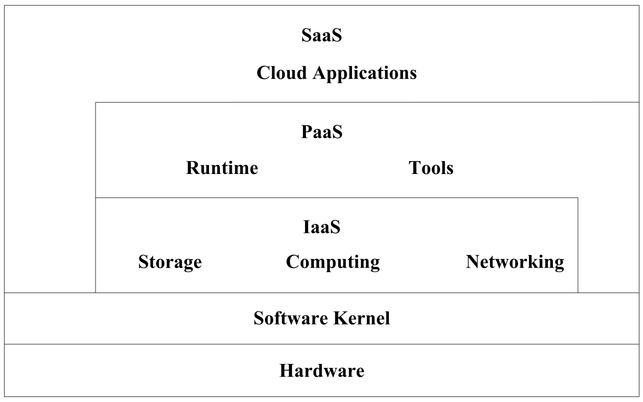


Figure 2.6: Cloud ontology structure, adapted from [41].

Cloud computing systems can be deployed in four deployment models: *private cloud*, *community cloud*, *public cloud*, and *hybrid cloud*. A *private cloud* is used by a specific organization and may be possessed and managed by the organization and/or other parties. A *community cloud* is used only by a group of users who have the same requirement, for instance, real-time jobs, data intensive applications, scientific research, etc. It is possibly owned and managed by the community or other providers; A *public cloud* is used by public users via the Internet. It is usually owned and managed by a company, and can also be administrated by other kinds of organizations; A *hybrid cloud* is a combination of at least two interoperable clouds of different types. It can be owned and managed by the providers of the integrated clouds or the provider of the unified interface [4].

2.2.1 OpenStack

OpenStack [29] is one of the most popular open source cloud operating systems for deploying and managing cloud platforms. Users of OpenStack include several large IT enterprises and organizations in different areas (mainly IT, telecommunications and academic research [42]) and countries, like Yahoo, Cisco, PayPal, and Purdue University in America, Suning Cloud Commerce in China, Cybera in Canada, etc. The usage of OpenStack ranges from private clouds to public clouds and also hybrid clouds. Using OpenStack, users can easily build their own cloud platforms and can customize functions based on the open source code.

OpenStack consists of six core components: Nova, Neutron, Swift, Cinder, Keystone, and Glance. Nova is the compute service that is responsible for the lifecycle management of VMs, such as spawning, migrating, terminating VMs, etc. Users can access the VMs via Secure SHell (SSH) commands using key files. OpenStack supports several types of hypervisors of VMs, such as Kernel-based Virtual Machine (KVM), Hyper-V, etc. [43]. Neutron provides network services to other components and enables users to configure network topologies and devices for their applications. Swift is the object storage service that can be used to reliably store unstructured data, such as pictures, videos, and documents, etc. Cinder is the block storage service that can be used to create and manage blocks (volumes). Keystone is the identity service for managing user information and service endpoints. Glance is the image service providing registration, query, and retrieval of images' metadata. The core components provide typical services of a IaaS cloud, i.e., computing, storage, and networking. Besides, OpenStack provides abundant REpresentational State Transfer (REST) Application Program Interfaces (APIs) for users to access components, which "have become the standard for enterprise IaaS" [42].

2.2.2 Cloudify

Cloudify [32] is an open source cloud orchestration (PaaS) software which is, since version 3.0, based on the Topology and Orchestration Specification for Cloud Applications (TOSCA) [44] standard. It helps users to deploy, monitor, and scale applications on private clouds, such as OpenStack clouds, as well as on public clouds, such as AWS, Microsoft Azure, etc., and even on a bare metal environment.

Cloudify divides an application into services. In versions 2.x, Cloudify uses Groovy files to describe applications, services, and dependencies between services. In versions 3.x, attributes of the services of an application and the relationships between these services are defined in a blueprint file which is based on the YAML [45] Domain Specific Language (DSL). Services, VMs, floating IPs, security groups [46], etc., can all be defined as nodes with properties. By characterizing nodes, Cloudify is able to control deployment details of the underlying cloud infrastructure. Besides, lifecycle operations of services, such as installing, starting, and stopping, can also be defined by several kinds of plug-ins in blueprints. Similar to OpenStack, Cloudify also provides REST APIs for above elements and functions, such as the blueprints API, the deployments API, the node instances API.

Chapter 3

Related Work

Reliability assessment, in the context of cloud computing, contains several aspects: hardware reliability, cloud system reliability, service/application reliability, etc. In this chapter, we introduce existing works in above areas.

3.1 Cloud Hardware Reliability

Many studies analyzed hardware failure characteristics of Cloud Computing Systems (CCSs) and high performance computing systems [47–50]. The reliability of hardware, e.g., PS, Central Processing Unit (CPU), memory, router, Hard Disk Drive (HDD), were analyzed according to the failure and repair logs of PSs. Works in this area focus on the hardware layer (refer to Figure 2.6) while our work in this thesis focus on the whole deployment stack of cloud applications. These works can provide the simulation with practical parameters for PS failures.

About failures of PSs, Garraghan et al. [50] presented an analysis of the Google trace log of 12,532 PSs in 29 days. They found that PS failures fit a Weibull distribution which is the same as the finding by Schroeder and Gibson [48]. They also observed that a small part of all PSs suffered much more failures than others, which is consistent with other works [47, 48].

Schroeder and Gibson [48] gave an analysis of the monitoring data for about ten-year of a high performance computing site (Los Alamos National Laboratory) with 4,750 nodes and another one-year node outage data set of a supercomputing system. They found that the time between failures can be well modeled by a Weibull distribution with a decreasing hazard rate and repair times can be well modeled by a lognormal distribution and, however, mean repair times are quite different for different systems.

Regarding failures of network devices, Gill et al. [49] gave an analysis of the network error logs for over one year of tens of different data centers. They found that data center network reliability is high when about 80% of the links and 60% of the devices, such as switches and routers, have an availability higher than four nines.

About the reliability of the whole data center including several PSs, Wei et al. [51] proposed an RBD-based reliability model for Virtual Data Centers (VDCs). A VDC is defined as a set of clusters connected by network modules. Each cluster contains several PSs, each of which is a backup to others and composed of a number of VMs, a Virtual Machine Monitor (VMM), and a PS. VMs hosted by the same PS are deemed as backups to each other. The VDC and the PS are respectively modeled by an RBD. With the RBD model, the reliability of a PS is evaluated as the probability that the PS and the VMM are reliable and at least one VM hosted by the PS is reliable. And the cluster reliability is assessed as the probability that at least one PS in the cluster succeeds. Finally, the VDC reliability is calculated as the probability that all clusters and network modules succeed. Compared with our reliability model, their model studies the reliability of VDCs built on PSs and does not include services and applications. Besides, they do not consider the case that VMs hosted by different PSs are backups to each other.

3.2 Cloud Computing System Reliability

Many researchers tried to evaluate the reliability of a CCS by modeling and analyzing the process of the CCS of fulfilling users' requests. In this context, the CCS is usually modeled as a set or a network of physical resources (PSs, switches, routers, etc.). Some works [52, 53] also combined the physical resources reliability with the cloud management software reliability to evaluate the CCS reliability. These works focus on the hardware layer and the software kernel layer, sometimes also the IaaS layer, but not the PaaS or SaaS layer (refer to Figure 2.6).

Studies in this area start from Dai et al.'s work [54]. Dai et al. tried to assess the reliability of cloud services, e.g., Amazon EC2 [55], in a way similar to the reliability assessment for Grid systems [56]. They divide the lifetime of a cloud service into two stages: request stage and execution stage, and assess the cloud service reliability as the product of the reliability of the two stages. During the request stage, like in Grid and cluster computing systems, they assume that there is a scheduler in the CCS to serve user requests. Then, considering overflow failures and timeout failures, they calculate the request stage reliability as the probability of satisfying user requests in time and model it with a Markov model as well as the queue theory. During the execution stage, they model a cloud service as a set of subtasks, and a cloud service succeeds only when all subtasks are successful. The execution stage reliability is calculated as the probability that at least one possible set of elements required by the service are available. Elements can be hardware, database, software, and network links, whose failure rates are all assumed constant. Dai et al.'s work focuses on the reliability of the IaaS cloud service itself, and it does not consider the structure and reliability of applications deployed on clouds. In our work, the reliability of cloud components and the CCS is considered known. Dai et al.'s work can be used as a complementary method to provide the reliability of the CCS, which is then considered to be a part of the PS reliability.

Based on Dai et al.'s work [54], Cui et al. [57] proposed a low complexity method for evaluating the cloud service reliability. Different from dividing cloud services into two stages, they simply model a cloud service as a set of subtasks. And they calculate the cloud service reliability as the sum of weighted reliability of all subtasks. They model the reliability of the cloud service in two cases: when node failures are independent, they model the cloud service with an undirected graph; when nodes have correlated failures, they model the cloud service with a Directed Acyclic Graph (DAG). They transform undirected graphs into directed graphs by replacing an undirected edge with two corresponding opposing directed edges and assigning the new edges with the failure probability of the original edge. However, they reduce the complexity of reliability evaluation at the cost of accuracy (in some cases, the absolute error can be more than 2%), which is one of the main concerns of our work. Besides, no application aspects are considered either.

Faragardi et al. [11] proposed Analytical Reliability Model for Reliability Assessment (ARMRA) for 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 like in Dai et al.'s work [54], while both the service and tasks are assumed fully reliable. They calculate the CCS reliability by combining the (exponential) reliability of PSs and links. And they calculate the server reliability as the product of the reliability of memory, hard disk, RAID controller and processor. Under several constraints, including memory, Quality of Service (QoS), task precedence, communication load and task redundancy, the maximum reliability of the CCS is evaluated. Similar to Dai et al.'s work [54] and Cui et al.'s work [57], Faragardi et al. do not consider applications that use cloud services as our work does.

Different from dividing services into subtasks, Lin and Chang [52] model the CCS as a capacitated-flow network of PSs, switches, physical lines, etc., between the cloud and clients. They assume that the capacity of nodes and edges have multiple states due to failures and maintenances. They define the reliability of a CCS as the probability that the CCS can send a number of data units to clients under constraints of data size, transmission time and maintenance cost. Then, they formalize the reliability assessment as the calculation of the probabilities of capacity vectors (paths) that fulfill the above constraints. However, compared with our work, a cloud in Lin and Chang's work is considered as a node providing services without internal details and no services or applications are considered.

A recent work is from Snyder et al. [53] who evaluate the reliability of a cloud with a set of (physical) resources as the probability of hosting a group of VMs. The CCS in this work is modeled as a set of PSs, each of which is abstracted as a 4-bit field, where each bit represents the state of a resource (including CPU, memory, HDD, and network bandwidth). A server is considered to be failed if any one resource fails (with the probability of its Annualized Failure Rate (AFR)). They define the reliability as the probability of the functional state of the CCS. They evaluate the reliability by calculating the probability that the CCS has more (physical) resources than requested for VMs using the non-sequential Monte Carlo Simulation (MCS). Compared to our work, this work has no consideration of cloud management softwares, services or applications.

3.3 Cloud Application Reliability

Many works have been proposed to improve and assess cloud application reliability. Cloud applications in this context are applications deployed on IaaS or PaaS clouds. In this section, we divide works in this area into works for reliability improvement and works for reliability evaluation and prediction, and describe the details.

3.3.1 Reliability Improvement

Works to improve reliability can be systematic or dedicated to specific methods. Specific methods are mainly about fault tolerance, whose two widely used mechanisms are check-pointing and replication/redundancy.

Works about systematic methods focus on system-level frameworks or tools for improving application reliability in a view of the combination of cloud applications and CCSs. Wu et al. [58] proposed a system accounting framework called S^5 based on Maslow's hierarchy of needs [59] for the Quality of Reliability (QoR). In Wu et al.'s work, they introduce generalized service reliability requirements, including existence, availability, capability and usability, and self-healing. S^5 examines the first three attributes of services and provides self-healing functions. The service reliability is improved by recognizing event patterns and predicting the most possible irregular events based on monitoring logs, and then adopting corresponding healing methods. Wu et al. predict the occurrence probability of serious events by analyzing log files of the application, which can be seen as a reliability assessment method, while in our work, we use the reliability of all components to assess and predict the application reliability.

Dudouet et al. [60] proposed a service orchestration framework, to manage dependencies and lifecycles of services used by an application. An orchestrator is proposed to improve reliability by handling "alarms" of violating performance thresholds detected by a monitoring service and enabling rolling upgrades of applications without scheduled maintenance. The design and functions of the service orchestration framework is similar to Cloudify. An alarm in Dudouet et al.'s work is related to the application type, so, the reliability assessment method in their work only focuses on the application and does not consider the whole deployment stack as we do.

Fault tolerance is an important technique to improve reliability and the characteristics of cloud computing, like virtualization and server consolidation, make it more convenient to perform fault tolerance. Zheng et al. [7, 61] proposed a component ranking framework called FTCloud to select significant components and apply different fault tolerance strategies to components according to their rankings. The selection is according to the invocation frequencies of components. And the alternative fault tolerance strategies include recovery block (standby redundancy), N-Version Programming (NVP) ($n/2$-out-of-n system) and parallel (active redundancy). Based on Zheng et al.'s work, Qiu et al. [62] proposed a reliability-based optimization framework called ROCloud to improve the reliability of legacy applications during their migration to clouds. The framework selects significant components according to the reliability impact which is considered as the number of failures where the application and the component failed simultaneously divided by the number of the component failures. Then, the optimal fault tolerance strategy is selected among the three strategies and the strategy of VM restart for the most significant components. The selection is constrained by failure rate, response time, and resource cost. These two works focus on reducing the cost of fault tolerance by applying it to only a part of chosen components. The reliability of components organized in different structures are assessed, however, they only consider service components, such as Amazon EC2 and Amazon Simple Storage Service (S3), rather than specific PSs, VMs and services as we do in our work. Another advance in our work is that we consider the k-out-of-n redundancy as a fault tolerance technique.

Another conceptual framework named Fault Tolerance Manager (FTM) for delivering fault tolerance as a service was proposed by Jhawar et al. [19]. The framework is designed to be a service layer working together with VMs on top of the VMM. FTM collects detailed information of PSs and provides the VM-level fault tolerance to cloud applications according to users' requirements. Jhawar et al. take PSs, hypervisors, VMs, redundancies and checkpointing into consideration, however, they do not separate an application and its hosting VM and they do not consider k-out-of-n redundancies. Besides, FTM is not implemented to the best of our knowledge.

Regarding specific methods for fault tolerance, many works study how to improve the reliability by checkpointing and replication/redundancy, and how to improve the effectiveness of the methods themselves. A recent representative work about checkpointing of cloud service is from Zhou et al. [63]. Zhou et al. proposed a method to reduce the network usage and storage resource consumption for checkpoint images of VMs. Their method chooses the checkpointing storage node based on its reliability, which is modeled by Weibull distributions. Therefore, the reliability assessment method in Zhou et al.'s work is only for PSs which is different from our work which can assess the reliability of not only PSs, but also VMs, services, and applications.

Many other works are about replication/redundancy. Zhao et al. [64] proposed a Low Latency Fault Tolerance (LLFT) middleware for managing the communication and membership of replicated application processes. Replicas of a process form a process group with one primary process and several backups. The LLFT middleware is composed of a low latency messaging protocol, which multicasts messages from the primary to the backups, and a leader-determined membership protocol, which handles the change of the primary and the backups. Similarly, Zhang et al. [65] proposed a Byzantine fault tolerance framework (BFTCloud) as a middleware for voluntary-resource clouds to manage the membership of cloud nodes [66], i.e., PSs. The primary and the replicas for executing a request from cloud modules form a Byzantine Fault Tolerance (BFT) group. BFTCloud selects and updates primaries and backups of BFT groups according to QoS values and priorities of cloud nodes. The two works respectively consider the primary-backup redundancy for application processes and Byzantine fault tolerance for PSs. Both of them do not consider the k-out-of-n redundancy, the reliability of VMs or the application structure while we do.

Recently, Zhou et al. [67] proposed a method to reduce the network resource consumption and the data transfer delay while guaranteeing the cloud service reliability, named OPtimal redundant Virtual Machine Placement (OPVMP). Zhou et al. first try to choose just enough PSs in as less subnets of PSs in a data center as possible for required VMs. Then, they place required VMs in the chosen PSs according to a heuristic algorithm. At last, they recover failed VMs by a recovery strategy minimizing the total network resource consumption. Different from our work, they assume that any two backup VMs are not on the same PS, which may happen in practice and is considered in our work.

Besides, Malik et al. [68, 69] proposed a model for tolerating failures of real time cloud applications based on the reliability of VMs. They assess the VM reliability in a reputation-based manner. The reliability increases if the compute instance, i.e., a VM or a physical processor, returns a correct result, otherwise, the reliability decreases. The increase and decrease are controlled by a designed reliability assessment algorithm. Different from the traditional definition of the reliability employed in our work, reliability in their works is allowed to be larger than one. In addition, they do not separate applications from VMs or PSs while we consider the components in the whole deployment stack of the application separately.

3.3.2 Reliability Assessment and Prediction

Lots of works research the evaluation/assessment and prediction of reliability of cloud services and applications, and are most related to our work in this thesis.

Padmapriya and Rajmohan [70] proposed a conceptual reliability evaluation method for web services. Similar to Wu et al.'s work [58], they also consider the reliability as a generalized property of services, and calculate it as the sum of the availability, the fault tolerance coverage probability, the fault recovery probability, and the service accuracy, i.e., the probability of returning right responses. They assign each of the above metrics with a weight, although it is unclear how to determine these weights. One difficulty of using Padmapriya and Rajmohan's method is to determine the quantitative numbers of above metrics. Padmapriya and Rajmohan's work is different from ours because it uses no service structure or reliability model, but measures reliability vaguely in an abstract way.

Concentrating on the service layer, Banerjee et al. [71, 72] proposed LOg-Based Reliability Assessment (LOBRA) for analyzing a commercial SaaS application's reliability based on access logs. The same as Tian et al.'s work [73], they use Nelson's model [74], which defines the reliability as $R = 1 - f/n$, where f was the number of failed entries/sessions and n is the total number of entries/sessions, to assess the application reliability. They evaluate the application reliability based on the filtered data. They proposed two kinds of data filtration methods: to distinguish the requests whether they are from registered users or from unregistered users and to classify entries according to the effects of its failures. Using these two methods, they measure the log entry-based reliability with $R_{entry} = 1 - f_e/n_e$, where f_e and n_e are the number of failed entries and total number of entries, respectively. And they measure session-based reliability with $R_{session} = 1 - f_s/n_s$, where f_s and n_s are the number of failed sessions and total number of sessions, respectively. Afterwards, they further divide log files into access logs, server logs, business process logs and customer problem report logs, but only access logs are used for reliability analysis [72]. Banerjee et al.'s work focuses on SaaS reliability and considers only service failures but no underlying failures, such as VM failures. Their method can also be applied to evaluate the reliability of common cloud applications as long as access logs are available.

Further considering the composite structure of services, Zheng and Lyu [13] proposed a collaborative reliability prediction method for service-oriented systems in the users' perspective. They predict the service reliability based on failure data of similar service users. First, similarities between users and between services are calculated based on failure probabilities of services observed by users. Secondly, users similar to the target user and services similar to the target service are chosen by identifying a number of most similar users and services. Thirdly, the service failure probability observed by the user is predicted by combining results predicted by using only failure probabilities observed by similar users' and by using only failure probabilities of similar services. At last, the failure probabilities of services are aggregated according to the compositional structure to calculate the reliability of the composite service. Comparing with our work, this work makes a different assumption that failure probabilities of services are possibly unknown before assessing reliability and focuses on the prediction of unknown failure probabilities. Besides, this work assesses service reliability in the users' perspective, which is different from the cloud providers' and consumers' perspective of our work. In the end, the fourth step of predicting the reliability of the composite service can be used to assess the reliability of cloud services/applications under the assumption that reliability of services are known, but without regard to redundancies or underlying structures of the system.

Taking redundancies and data into consideration besides above considerations, Wang et al. [20] proposed SErvice-Based Reliability Assessment (SEBRA), which is a hierarchical reliability model for modeling and evaluating the reliability of service-based software systems. They assess the software reliability by combining the reliability of the workflow, service pools, services, and data. Comparing with Zheng et al.'s work [13], Wang et al.'s work considers more aspects when assessing service reliability, such as service pools. The model can be adapted to evaluate the reliability of cloud applications as composite services, but regardless of VMs or PSs failures which are considered in our work.

Focusing on Common Cause Failures (CCFs), Qiu et al. [12] proposed Hierarchical Correlation Model for Reliability Assessment (HCMRA) for cloud services. HCMRA can also evaluate performance and power consumption of cloud services. A CCF indicates that a PS failure would bring down all VMs on the PS. In their model, the reliability of large online services, e.g., social networking services, is assessed. Meanwhile, 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. And the reliability of the service itself is not considered, instead, they consider only VM and PS failures. Both VM and PS failures are assumed to follow Poisson processes with different constant failure rates and the Markov process is utilized to model the available amount of VMs. The cause of CCF in their work is similar to the deep dependency defined in our work. The difference to our work is that they do not consider the impact of the k-out-of-n redundancy, service reliability or the application structure.

Based on the whole deployment structure of applications, Thanakornworakij et al. [10] proposed High Performance Computing Reliability Assessment (HPCRA) for calculating the PDF and the reliability of high performance applications, specifically, Message Passing Interface (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 during the reliability assessment process. They focus on exploring the impact of correlations of failures to cloud application reliability. However, they do not consider any redundancies as we do.

In the context of deploying applications to VDCs, Zhang et al. [14] proposed a method and a framework to assess the availability of the VDC provisioned for services considering hardware failures and dependencies between virtual components. Zhang et al. define the service availability of a 3-tier web application, which consists of web servers, application servers and database servers, as the probability that "there exists a path from the web server to the database server where every component (physical nodes and links) along the path is available" [14]. The service availability is estimated using the importance sampling technique [75]. Different from our work, on one hand, they do not consider the failures of VMs or the failures of the application itself, on the other hand, the service availability is estimated rather than accurately calculated in their work.

Chapter 4

A Reliability Assessment Framework for Cloud Applications

In this chapter, we introduce a DEBRA-based framework for monitoring, analyzing, and assessing the reliability of cloud applications with DEBRA. An initial version of DEBRA and the framework is proposed in a paper [76]. At first, we introduce the overview of our framework. Afterwards, we describe details of its components and DEBRA.

4.1 Overview

The framework is designed to assess the reliability of cloud applications during both the development and the maintenance phases. To this aim, we develop three functional components for the framework: a dependency analyzer, a monitor, and a reliability analyzer. The dependency analyzer analyzes dependencies between components in the deployment stack of applications and creates a graph named Layered Dependency Graph (LDG) to represent the application deployment structure. 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 with DEBRA. During the development phase of an application, the reliability of components obtained by testing or manually setting, and assumed dependencies can be used by the reliability analyzer to predict the reliability of the application with certain deployment structures. In this case, the reliability analyzer can work solely without the dependency analyzer or the monitor. And, during the maintenance phase, the application is deployed on clouds. The field failure data can be obtained by the monitor and the LDG can be built by the dependency analyzer. In this case, the reliability analyzer works collaboratively with the dependency analyzer and the monitor and uses field data to assess the application reliability. The framework structure in this scenario is shown in Figure 4.1, where the deployment stack of cloud applications is divided into three layers: the application layer consisting of services, the VM layer, and the PS layer.

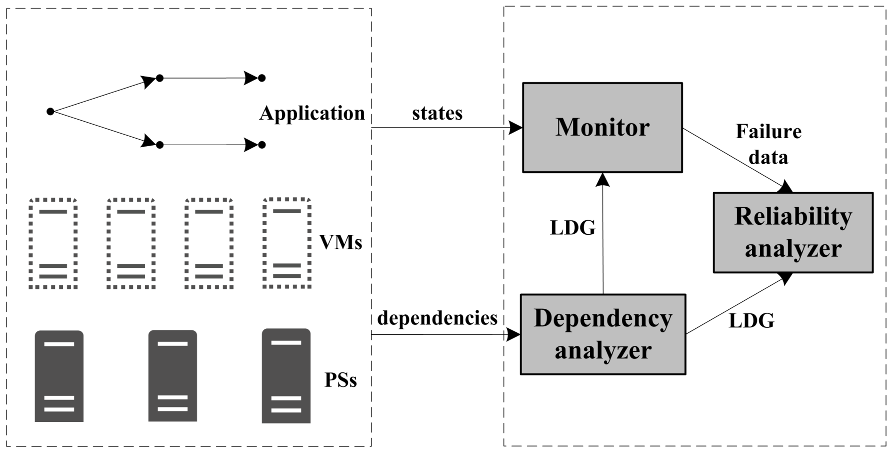


Figure 4.1: Deployment stack of cloud applications and overview of the framework.

4.2 Dependency Analyzer

The dependency analyzer is designed to collect components and the dependencies among them from the cloud where the application is deployed and to generate an LDG. 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 to fulfill its function is defined as the *predecessor* and the needed component is defined as the *successor*. The application components include services, SIs, VMs, and PSs. In the following, we introduce how the dependency analyzer models the relationships between the above components.

The dependency analyzer models a cloud application as a composition of several services, each of which has n ($n \geq 1$) SIs and every SI is deployed on one VM. SIs of a service are generally organized as a k-out-of-n system (Section 2.1.2.3). When $k = 1$, which means that the service needs at least one SI to succeed, SIs are organized as a normal parallel system (Section 2.1.2.2). When $k > 1$, which usually means that the service needs more than one SIs to ensure its performance, SIs are organized as a k-out-of-n system (Section 2.1.2.3). Therefore, a service is seen as an abstract component, which is instantiated, exists in the form of a k-out-of-n system of SIs. The dependency between two services are defined as a **function dependency** which means that a service needs another service for its full function. For example, a website needs a database to store user information. Furthermore, function dependencies are divided into two types: *sequence function dependency* and *choice function dependency*. A sequence function dependency is a function dependency with a weight of $w = 1$ and a choice function dependency is a function dependency with a weight of $0 < w < 1$, and the weights of all choice function dependencies with the same predecessor sum up to 1. Besides, we assume that there are no cycles or triangles of function dependencies in the application structure.

In cloud systems, we assume that SIs are deployed in VMs which are deployed on PSs. Meanwhile, 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 thesis, a **deployment dependency** is formally defined as the dependency between two components when one component is deployed on another one. A PS can host more than one VM and a VM can host more than one SIs. In this thesis, for simplicity, we assume that a PS can host several VMs while one VM can hold only one SI. Under this assumption, VMs hosting SIs of the same service may be deployed on the same PS, which makes the PS a possible Single Point of Failure (SPoF) of the application. In this case, we define the dependency between the SIs as a **deep dependency**. The deep dependency is similar to the reason of common cause failures introduced by Qiu et al. [12], which are simultaneous failures of all VMs on the same PSs induced by failures of the PS.

Based on function dependencies and deployment dependencies, the dependency analyzer uses an LDG to model components and their dependencies. In this thesis, an LDG is formally defined as a triangle-free DAG $G(V,E)$, where V is the set of components and E is the set of dependencies between components. A component is defined as $v(t,m,k,n)$, where t is the type of the component, which can be one of *PS*, *VM*, *SI* and *service*, m is the name of the component, and k and n are for services which have n SIs and need at least k of them to succeed. A dependency is defined as $e(p,s,d,w)$, where p and s are the *predecessor* and *successor*, respectively, d is the type, and w is the weight of the dependency. Dependencies are transitive, i.e., the successor of a component's successor is also the component's successor and the same for predecessors. The type d can be *function* or *deployment* denoting the function dependency or the deployment dependency, respectively. The sum of weights of function dependencies with the same predecessor is 1 and the weight of every deployment dependency is 1. An example of the LDG is shown in Figure 4.2, where solid arrows represent function dependencies, dashed arrows represent deployment dependencies, and all arrows without weights represent dependencies with a weight of 1.

An LDG is composed of four layers from bottom to top: the PS layer, the VM layer, the SI layer, and the service layer. The four types of components: PSs, VMs, SIs, and services, are included in the corresponding layer. As shown in Figure 4.2, there are two choice function dependencies with weights w_1 and w_2, and one sequence function dependency in the services layer.

Base on the LDG model, we define the *sub-LDG (sLDG)* of a service se, $G(se)$, as the subgraph of the LDG induced by the vertex set containing the service and its successors. For example, for the rightmost service in Figure 4.2, the vertex set contains the service itself, its two SIs, the two VMs hosting the SIs and the two PSs hosting the two VMs. The sLDG of the service is shown in Figure 4.3 as the right bold part. Furthermore, we define the *sLDG* of a set of PSs $\{ps_1, ps_2, ..., ps_n\}$, $G(\{ps_1, ps_2, ..., ps_n\})$ where ps_i $(i = 1,2,...,n)$ is the ith PS, as the subgraph of the LDG induced by the vertex set containing all PSs in $\{ps_1, ps_2, ..., ps_n\}$ and their predecessors. The sLDG of the leftmost two PSs is shown in Figure 4.3 as the left bold part.

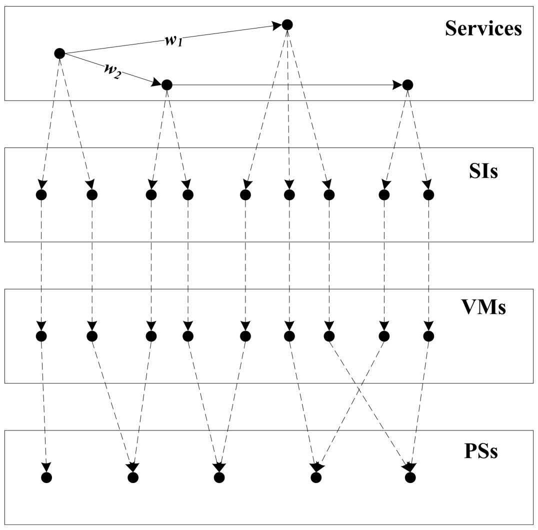


Figure 4.2: An example LDG.

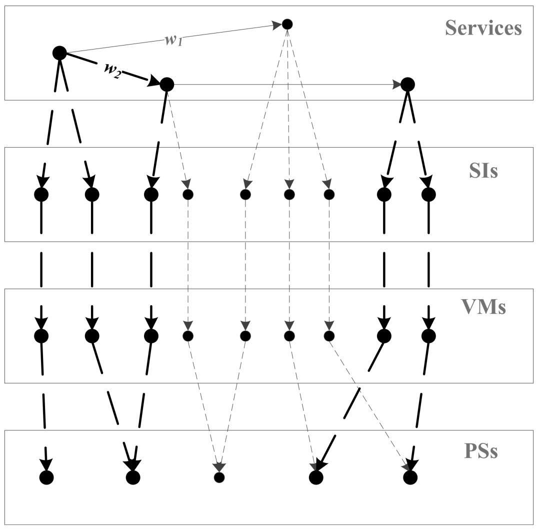


Figure 4.3: The sLDGs of a service and a set of PSs.

4.3 Monitor

The monitor is designed for monitoring and logging states of PSs, VMs, and SIs after the deployment of an application. State information of these components are gathered and logged in log files once per time unit. A time unit can be a minute, an hour or a day, etc. We assume that a component has only one state during a time unit, either *success* or *failure*.

The success state means that the component functions without failures during the time unit and the failure state means that the component suffers at least one failure. In this thesis, a component succeeds or is successful represents that the component is in its success state and correspondingly, a component fails represents that the component is in its failure state. To distinguish the two states, we define failure scopes of above components as follows:

- for a PS, possible failures are that the PS crashes, network failures that make a PS unreachable, and cloud management software failures, especially hypervisor failures, that make a PS unable to host VMs;
- for a VM, possible failures are that the VM crashes, (virtualized) network failures that make a VM unreachable or unable to host SIs; and
- for an SI, possible failures are the corresponding service process crashes or Operating System (OS) failures that make an SI inactive.

As service components are abstract and instantiated to SIs, possible failures of a service are that the number of its successful SIs is less than required. Therefore, the state of a service during a time unit is determined by both the states and the organization of its SIs. For example, suppose a service has four SIs organized as a 2-out-of-4 system, then if more than two SIs are in a failure state, the service is in failure state. The states of services are determined by the reliability analyzer based on the SI states obtained by the monitor and the LDG generated by the dependency analyzer.

4.4 Reliability Analyzer

The reliability analyzer is responsible for assessing the reliability of the application and all components in the LDG according to the dependencies obtained from the dependency analyzer and the field failure data obtained from the monitor. In this section, we first introduce the assumptions made for the reliability assessment. Afterwards, we describe DEBRA for components and the application.

4.4.1 Assumptions

First of all, because of the reliability assessment methods for different components are distinct, we divide components in LDGs into two types: service components and non-service components including PSs, VMs and SIs. Then, before making assumptions, we need to clarify the following terms:

- *Inner reliability (r)*: for a non-service component, similar to the definition of reliability (Section 2.1), the inner reliability is defined as the ability to perform its required functions without need of other components under stated conditions for a specified period of time; for a service, the inner reliability is defined as the ability to perform its required functions without need of its service successors, i.e., only SIs, VMs, and PSs are considered when assessing the inner reliability of services, under stated conditions for a specified period of time.

- *Inner failure*: the failure of a component that is caused by itself rather than its successors. For services, inner failures are the failures which are caused by its non-service successors. Corresponding to the inner reliability, the rate of inner failures is defined as the *inner failure rate*.

The following assumptions are made for reliability assessment methods in this thesis:

- A1: component failures are fail-stop, which means that a component suffering a failure will stop working and the failure can be detected;
- A2: inner failures of non-service components are independent;
- A3: the same type of non-service components have the same inner reliability;
- A4: the reliability of a component is determined by both its inner reliability and the reliability of its successors;
- A5: inner reliability and reliability are assessed for one time unit, during which the state of every component and the application can only be success or failure while the probability of a component's success state equals its inner reliability; and
- A6: SIs and VMs are mapped one-to-one which indicates that only one SI can be deployed on one VM.

Regarding the assumption A5, VMs and SIs are software components, and the occurrence of their failures are assumed to follow exponential distributions (Section 2.1.1), therefore, their inner reliability in a time unit can be calculated by formula (2.1.10) by with $t = 1$, i.e.:

$$r = e^{-\lambda} \tag{4.4.1}$$

As introduced in Section 4.3, a PS may fail because of several kinds of hardware and software failures. When software failures are in the majority, failures of a PS can be approximated by an exponential distribution and the inner reliability of the PS can be calculated with (4.4.1). When hardware failures are as many as software failures, failures of a PS can be seen as aggregated failures and the occurrence of them can be modeled with a Poisson process [77], which leads to a constant failure rate and the calculation of the inner reliability of the PS with (4.4.1). When hardware failures dominate, as introduced in Section 2.1.1, failures of a PS can be modeled by a Weibull distribution and the inner reliability of PSs in a time unit can be calculated by (2.1.15) with $t = 1$, i.e.:

$$r = e^{-(\frac{1}{\eta})^{\beta}} \tag{4.4.2}$$

where β and η can be adjusted to control the failure rate. In the rest of this thesis, similar to some existing works [12, 54, 77], we assume that failure of a PS follows an exponential distribution and use (4.4.1) to calculate the PS inner reliability.

Regarding the assumption A6, in practice, SIs of two or more services are possibly deployed on one VM, in this case, SIs and services can be integrated by reducing the application structure and updating the (inner) reliability according to the reduced structure. In this thesis, we only consider the reduction of sequence function dependencies, since the reduction of choice function dependencies may lead to triangles or significantly change the application structure. We assume that if two different SIs, i.e., two SIs of two different services, are deployed on one VM, the corresponding two services should meet three conditions for a reduction: *a*) one service is directly dependent on another with a sequence function dependency, *b*) the two services have the same amount, e.g., n, of SIs and *c*) the $2n$ SIs of the two services are deployed on n VMs with each VM hosting two different SIs. The condition a means that one service is a direct successor of another. For the condition c, suppose two services both have three SIs and one service is dependent on another with a sequence function dependency, some possible deployments of their SIs are shown in Figure 4.4. The deployment in Figure 4.4(a) satisfies the condition c while the deployments in Figure 4.4(b) to 4.4(d) violates because either the number of VMs is wrong (Figure 4.4(c) and Figure 4.4(d)) or SIs of the same service are deployed on the same VM (Figure 4.4(b) and Figure 4.4(c)).

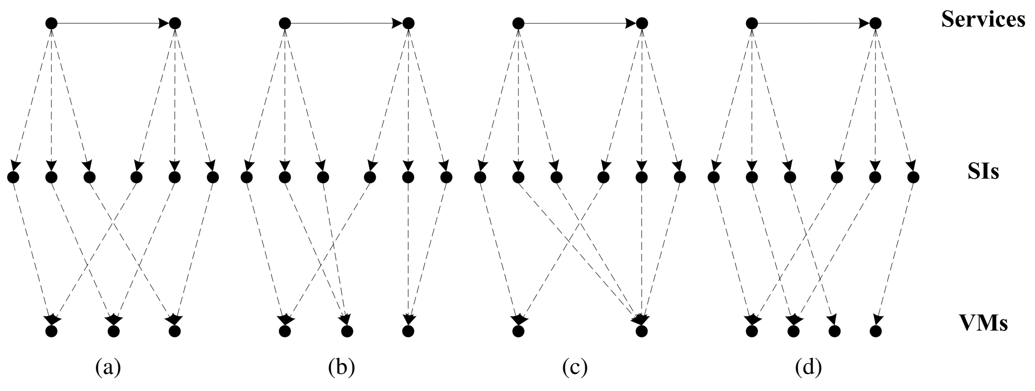


Figure 4.4: Deployment examples of two sequential services.

Conditions a to c guarantee that two services meeting them can be reduced to one service. For example, the deployment in Figure 4.4(a) can be reduced into the deployment in Figure 4.5. SIs on the same VM are integrated into a composition instance and the two services are integrated into a composition service. Suppose the inner reliability of SIs of the two services in Figure 4.4(a) is r_1 and r_2, respectively. Then, the inner reliability r of the composition instance is calculated by $r = r_1 r_2$ according to the sequence function dependency between the two services.

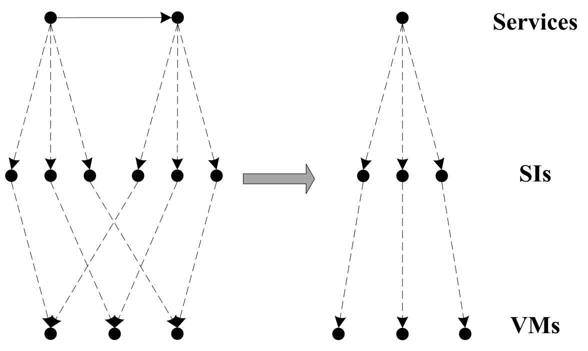


Figure 4.5: The reduction of two sequential services.

4.4.2 Non-service Component Reliability

In this section, we describe the reliability assessment methods for non-service components, i.e., PSs, VMs and SIs.

In an LDG, PSs are in the bottom layer and not dependent on any further components. And as assumed in Section 4.4.1, the reliability of a component is determined by its inner reliability and the reliability of its successors. As PSs do not have successors, the reliability of a PS, R_{ps}, is fully determined by its inner reliability r_{ps} whose calculation is described in Section 4.4.

As introduced in Section 4.4, VMs and SIs are software components, and their inner reliability r_{vm} and r_{si} are calculated with (4.4.1). Let λ_{vm} and λ_{si} be the inner failure rate of VMs and SIs, respectively, then we have:

$$r_{vm} = e^{-\lambda_{vm}} \tag{4.4.3}$$

$$r_{si} = e^{-\lambda_{si}} \tag{4.4.4}$$

$$\tag{4.4.5}$$

Meanwhile, a VM has only one successor which is the PS hosting it. Therefore, a VM and the hosting PS form a two-component series system (Section 2.1.2.1), and the reliability of a VM R_{vm} is calculated by:

$$R_{vm} = r_{vm}r_{ps} = e^{-(\lambda_{vm}+\lambda_{ps})} \tag{4.4.6}$$

where r_{ps} is the (inner) reliability of the PS hosting the VM and λ_{ps} is the failure rate of the PS.

Regarding SIs, each of them is hosted by a VM which is hosted by a PS. An SI, the hosting VM and the hosting PS form a three-component series system, and the reliability of an SI R_{si} is calculated by:

$$R_{si} = r_{si} r_{vm} r_{ps} = e^{-(\lambda_{si}+\lambda_{vm}+\lambda_{ps})} \tag{4.4.7}$$

where r_{si} is the inner reliability of the SI.

Given the one-to-one mapping between SIs and VMs, we define the composition of an SI and the VM hosting it as a composite component named SI-VM (SV) to represent the series system of the SI and the VM. Consequently, the inner reliability of an SV r_{sv} is calculated by:

$$r_{sv} = r_{si} r_{vm} = e^{-(\lambda_{si}+\lambda_{vm})} \tag{4.4.8}$$

Furthermore, an SV and the PS hosting the corresponding VM form a series system and the reliability of an SV R_{sv} can be calculated with:

$$R_{sv} = r_{sv} r_{ps} = r_{si} r_{vm} r_{ps} = R_{si} \tag{4.4.9}$$

4.4.3 Service Inner Reliability

In this section, we introduce the assessment method for the inner reliability of services.

A service is usually deployed with several SVs for fault tolerance, each of which is deployed on a PS. As introduced in Section 4.2, n redundant SVs of a service can be generally modeled as a k-out-of-n system where k is the least number of SVs required by the service to succeed. And, as defined in Section 4.3, the inner reliability of a service is the probability that at least k SVs succeed. If all SVs of a service are independent with each other, the inner reliability of the service can be calculated using formula (2.1.18):

$$r_{se} = \sum_{i=k}^{n} C_n^i (R_{sv})^i (1-R_{sv})^{n-i} \tag{4.4.10}$$

where R_{sv} is the reliability of the SV, $1 \le k \le n$, and C_n^i is the number of i-combinations from a set of n elements.

However, the assumption that all SVs of a service are independent does not hold when two or more SVs of the same service are deployed on the same PS, because failures of these SVs are dependent on the same PS's failures. In this case, deep dependencies must be taken into consideration to calculate the service inner reliability.

For a service whose n SVs are hosted by m PSs and configured as a k-out-of-n system, we assume that n_l $(l = 1,2,...,m)$ SVs are hosted by the lth PS. In this case, (4.4.10) can be adapted to:

$$r_{se} = \sum_{i=k}^{n} P_{v_i} \tag{4.4.11}$$

where P_{v_i} is the probability of the event that i out of n SVs succeed which is represented by v_i. 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 4.6, where dashed arrows are deployment dependencies. If the service requires at least four SVs to succeed, i.e., the six SVs are configured as a 4-out-of-6 system, we have $k = 4$, $n = 6$, $m = 3$, $n_1 = 1$, $n_2 = 3$, and $n_3 = 2$. For this service, when $i = 4,5,6$, there are respectively C_6^i possibilities that i out of six SVs succeed.

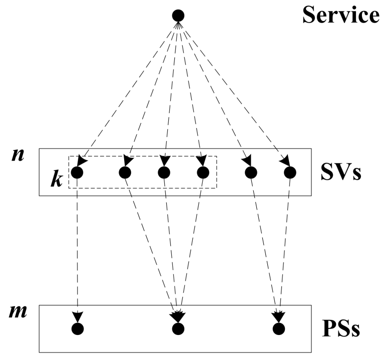


Figure 4.6: An example deployment structure of a service.

If we assume that there are d_i scenarios that i ($i = k, k+1, ..., n$) SVs succeed, P_{v_i} can be calculated by summing up the probability of each scenario:

$$P_{v_i} = \sum_{j=1}^{d_i} P_{c_j} \tag{4.4.12}$$

where P_{c_j} is the probability that the jth scenario, c_j, occurs. For the service shown in Figure 4.6, one scenario where four out of six SVs succeed is shown in Figure 4.7.

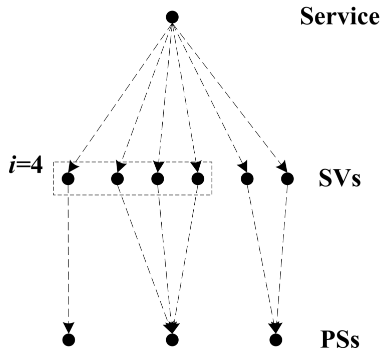


Figure 4.7: A scenario that four out of six SVs succeed.

For the jth scenario that i SVs succeed, suppose the number of successful SVs on the lth

PS is s_{jl} ($l = 1, 2, ..., m$). Then, s_{jl} should meet:

$$0 \leq s_{jl} \leq n_l \tag{4.4.13}$$

$$\sum_{l=1}^{m} s_{jl} = i \tag{4.4.14}$$

where (4.4.13) means that the number of successful SVs on a PS, s_{jl}, can range from zero to the number of total hosted SVs, n_l, and (4.4.14) means that the number of successful SVs on each PS sums up to the number of total successful SVs.

Before calculating the probability of the jth scenario P_{c_j}, we need to find out all the d_i scenarios that i out of n SVs succeed. If we define a scenario as a sequence $(s_1, s_2, ..., s_m)$, 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 (4.4.13). As defined by Bona [78, p. 89], 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 [79], 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 (4.4.13) with $l = 1, 2, ..., m$. To solve the problem, we employ the generalized algorithm for restricted weak composition generation proposed by Page [79]. 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 is a queue which contains all restricted weak compositions of the integer. In our case, the restricted set of s_l is $\{0, 1, ..., n_l\}$, the integer is the number of successful SVs, i, and the number of parts of the restricted weak composition is the number of PSs, m. Let's take the service in Figure 4.6 as an example. The restricted sets are $\{0, 1\}$, $\{0, 1, 2, 3\}$, and $\{0, 1, 2\}$, the integer i is 4, 5 or 6, and the number of parts is 3. When i=4, the output of the algorithm would be $((1, 1, 2), (1, 2, 1), (1, 3, 0), (0, 2, 2), (0, 3, 1))$, which indicates that the number of possible scenarios of four successful SVs is 5, i.e., $d_4 = 5$. The scenario c_3 corresponding to the weak composition $(1, 3, 0)$ is shown in Figure 4.8, where successful SVs are connected with the service and PSs with dashed bold arrows.

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:

$$P_{c_j} = \prod_{l=1}^{m} P_{u_{jl}} \tag{4.4.15}$$

where $P_{u_{jl}}$ is the probability that s_{jl} SVs on the lth PS succeed in the j scenario. As the probability of the success state of a component is its inner reliability, $P_{u_{jl}}$ can be calculated

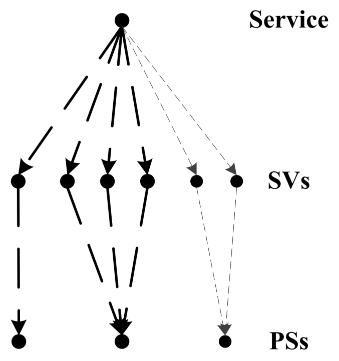


Figure 4.8: The scenario corresponding to the weak composition $(1,3,0)$.

according to the inner reliability of SVs and PSs. When $s_{jl} = 0$ representing that no SVs on the lth PS succeeds, $P_{u_{jl}}$ is the sum of two probabilities which are 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, $r_{ps}(1 - r_{sv})^{n_l}$, therefore, we have:

$$P_{u_{jl}} = (1 - r_{ps}) + r_{ps}(1 - r_{sv})^{n_l},\ s_{jl} = 0 \tag{4.4.16}$$

When $s_{jl} > 0$ representing that s_{jl} SVs on the lth PS succeed, $P_{u_{jl}}$ is the probability that the lth PS succeeds, r_{ps}, 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 have:

$$P_{u_{jl}} = r_{ps}(r_{sv})^{s_{jl}}(1 - r_{sv})^{n_l - s_{jl}},\ s_{jl} > 0 \tag{4.4.17}$$

In total, we get:

$$P_{u_{jl}} = \begin{cases} (1 - r_{ps}) + r_{ps}(1 - r_{sv})^{n_l}, & \text{if } s_{jl} = 0 \\ r_{ps}(r_{sv})^{s_{jl}}(1 - r_{sv})^{n_l - s_{jl}}, & \text{if } s_{jl} > 0 \end{cases} \tag{4.4.18}$$

Taking the third scenario c_3 in Figure 4.8 as an example, we have $s_{31} = 1$, $s_{32} = 3$, and $s_{33} = 0$, then:

$$\begin{aligned} P_{u_{31}} &= r_{ps} r_{sv} \\ P_{u_{32}} &= r_{ps}(r_{sv})^3 \\ P_{u_{33}} &= 1 - r_{ps} + r_{ps}(1 - r_{sv})^2 \end{aligned} \tag{4.4.19}$$

At last, combining (4.4.11), (4.4.12), and (4.4.15), we calculate the inner reliability of a service with:

$$r_{se} = \sum_{i=k}^{n} \sum_{j=1}^{d_i} \prod_{l=1}^{m} P_{u_{jl}} \tag{4.4.20}$$

In the following, we employ the service in Figure 4.6 to show how the inner reliability of a service can be calculated. Suppose the inner reliability of PSs and SVs are $r_{ps} = 0.9$ and $r_{sv} = 0.8$, respectively. As the six SVs are configured as a 4-out-of-6 system, inputs of the algorithm proposed by Page [79] are the integers 4, 5, and 6, the number of parts is 3, and the restricted sets are $\{0,1\}$, $\{0,1,2,3\}$, and $\{0,1,2\}$. The generated restricted compositions are shown in Table 4.1, where each column shows the list of scenarios when i SVs succeed with c_j $(j = 1,2,3,4,5)$ representing the jth scenario.

Table 4.1: Generated compositions.

$i = 4$	$i = 5$	$i = 6$
$c_1 = (1,1,2)$	$c_1 = (1,2,2)$	$c_1 = (1,3,2)$
$c_2 = (1,2,1)$	$c_2 = (1,3,1)$	
$c_3 = (1,3,0)$	$c_3 = (0,3,2)$	
$c_4 = (0,2,2)$		
$c_5 = (0,3,1)$		

For the scenario $c_3 = (1,3,0)$, by substituting $r_{ps} = 0.9$ and $r_{sv} = 0.8$ into (4.4.19), we get:

$$\begin{aligned} P_{u_{31}} &= r_{ps} r_{sv} = 0.72 \\ P_{u_{32}} &= r_{ps} (r_{sv})^3 = 0.4608 \\ P_{u_{33}} &= 1 - r_{ps} + r_{ps}(1 - r_{sv})^2 = 0.136 \end{aligned}$$

then, according to (4.4.15), we get P_{c_3} by:

$$\begin{aligned} P_{c_3} &= \prod_{l=1}^{3} P_{u_{3l}} \\ &= P_{u_{31}} P_{u_{32}} P_{u_{33}} \\ &= 0.72 * 0.4608 * 0.136 \\ &= 0.045121536 \end{aligned}$$

Similarly, for the other scenarios, we can get:

$$\begin{aligned}P_{c_1} =&0.035831808\\P_{c_2} =&0.071663616\\P_{c_4} =&0.055738368\\P_{c_5} =&0.037158912\end{aligned}$$

Then, substituting $d_4 = 5$ and values of P_{c_j} $(j = 1,2,3,4,5)$ into (4.4.12), we get:

$$\begin{aligned}P_{v_4} =&\sum_{j=1}^{5} P_{c_j}\\=&P_{c_1} + P_{c_2} + P_{c_3} + P_{c_4} + P_{c_5}\\=&0.24551424\end{aligned}$$

Via the same process, we can get $P_{v_5} = 0.313196544$ and $P_{v_6} = 0.191102976$, and finally get the inner reliability of the service according to (4.4.11):

$$\begin{aligned}r_{se} =&\sum_{i=4}^{6} P_{v_i}\\=&P_{v_4} + P_{v_5} + P_{v_6}\\=&0.24551424 + 0.313196544 + 0.191102976\\=&0.74981376\end{aligned}$$

4.4.4 Service Reliability

In this section, we introduce the reliability assessment method of services. To this aim, we need to consider function dependencies and possibly existing deep dependencies between SIs, beside its inner reliability.

Regarding the function dependencies, Wang et al. [80] proposed four basic architectural styles for modeling software architecture: *batch-sequential* or *sequence*, *parallel* or *pipe-filter*, *fault tolerance*, and *call-and-return* or *loop*, which are widely used to model compositional software [13, 20, 81]. An extra style, *choice*, is usually used as an supplement to the above styles. Suppose service successors of a service are independent and form one of the above styles, the above styles and the calculation of their reliability are introduced according to the definition of Wang et al. [80]:

- *Sequence*: service successors are executed in sequence. If two services are executed in sequence, then the first executed one is the predecessor and the later one is the successor. The sequence style fails if any one successor fails. According to formula (2.1.16), the sequence style's reliability R_{seq} is:

$$R_{seq} = \prod_{i=1}^{n} R_{suc_i} \tag{4.4.21}$$

where R_{suc_i} is the reliability of the ith successor and n is the number of successors.

- *Pipe-filter*: service successors are identical and executed simultaneously to improve performance. In this case, identical services can be considered as SIs of the same service. Therefore, the reliability of a pipe-filter style equals the inner reliability of a service whose SIs are configured as a n-out-of-n system where n is the number of SIs.
- *Fault tolerance*: service successors are identical and at least one succeeds while others are backups. The same as for the pipe-filter style, identical services can also be considered as SIs of the same service, but the difference is that the reliability of a fault tolerance style equals the inner reliability of a service whose SIs are configured as a 1-out-of-n system where n is the number of SIs.
- *Loop*: a service successor is executed several times to fulfill another service's function. The fulfilled service is the predecessor and the looped service is the successor. The loop style fails if the successor fails at least once during its execution. If we assume that the service is expected to execute n times during a time unit, then the reliability R_{loo} of a loop style is:

$$R_{loo} = (R_{suc})^n \tag{4.4.22}$$

  where R_{suc} is the looped successor's reliability.
- *Choice*: each service successor is chosen to execute with a probability. The service calling other services is the predecessor and the called services are successors. The choice style fails if the chosen successor fails. Assuming that the ith service out of n successors is chosen with a probability of w_i and $\sum_{i=1}^{n} w_i = 1$ where n is the total number of successors, the reliability of a choice style R_{cho} is:

$$R_{cho} = \sum_{i=1}^{n} w_i R_{suc_i} \tag{4.4.23}$$

  where R_{suc_i} is the reliability of the ith successor

As introduced above, sequence and choice styles are enough to model function dependencies because other styles are either modeled in lower layers of LDGs or reduced to the two styles. The two kinds of function dependencies defined in Section 4.2: sequence function dependency and choice function dependency correspond to the sequence and choice style, respectively.

Based on the reliability assessment method of the choice and sequence style, the reliability of a style consisting of several styles can be calculated by reducing the style into a sequence style. Similar to the failure probability composition proposed by Zheng and Lyu [13], the reduction is achieved by replacing every style with a virtual service with the same reliability. And according to (4.4.21), the reliability of the original style can be expressed as $R = \prod r_i$, where r_i is the inner reliability of the ith service successor in the final sequence style. An example of the reduction is shown in Figure 4.9.

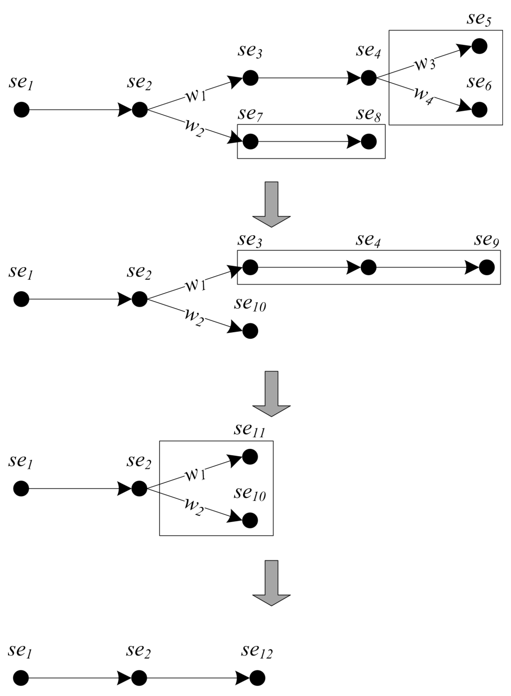


Figure 4.9: An example of style reduction.

In the first step, the choice style formed by se_5 and se_6 is reduced and replaced by a virtual service se_9 and the sequence style of se_7 and se_8 is replaced by se_{10}. Afterwards, the sequence style of se_3, se_4, and se_9 is replaced by se_{11}. In the last step, the choice style of se_{10} and se_{11} is replaced by se_{12}. Let the inner reliability of se_i be r_i $(i = 1, 2, ..., 12)$ and suppose r_1 to r_8 are known. According to (4.4.21) and (4.4.23), we have:

$$\begin{aligned}
r_9 &= w_3 r_5 + w_4 r_6 \\
r_{10} &= r_7 r_8 \\
r_{11} &= r_3 r_4 r_9 \\
r_{12} &= w_1 r_{11} + w_2 r_{10}
\end{aligned}$$

And the reliability of the style R_{sty} is:

$$R_{sty} = r_1 r_2 r_{12} \tag{4.4.24}$$

If services are independent from each other, i.e., irrespective of PSs or deep dependencies, the service's reliability R'_{se} can be calculated according to the style of its successors:

$$R'_{se} = r'_{se} R_{sty} \tag{4.4.25}$$

where r'_{se} is the inner reliability of the service without regard to PSs and R_{sty} is the reliability of the style formed by the service successors of the service. For example, suppose a service se_1 depends on two services se_2 and se_3 which form a choice style, as shown in Figure 4.10.

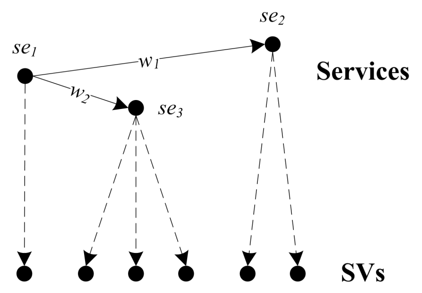


Figure 4.10: An example of choice style.

Services se_1, se_2, and se_3 have respectively one, two, and three SVs and are respectively configured as a 1-out-of-1 system, a 1-out-of-2 system, and a 2-out-of-3 system. Let the inner reliability of SVs be r_{sv} and weights of the two choice function dependencies be respectively w_1 and w_2, then, r'_{se_1}, r'_{se_2}, and r'_{se_3} can be assessed by:

$$r'_{se_1} = r_{sv} \tag{4.4.26}$$

$$r'_{se_2} = \sum_{i=1}^{2} C_2^i (r_{sv})^i (1 - r_{sv})^{2-i} \tag{4.4.27}$$

$$r'_{se_3} = \sum_{i=2}^{3} C_3^i (r_{sv})^i (1 - r_{sv})^{3-i} \tag{4.4.28}$$

and according to (4.4.23), the reliability of the choice style formed by se_2 and se_3, R_{sty}, is:

$$R_{sty} = w_1 r'_{se_2} + w_2 r'_{se_3} \tag{4.4.29}$$

then, according to (4.4.25), the reliability of se_1, R'_{se_1}, is:

$$R'_{se_1} = r'_{se_1}(w_1 r'_{se_2} + w_2 r'_{se_3}) \tag{4.4.30}$$

However, taking the whole LDG including PSs into consideration, we can not substitute the service inner reliability obtained in Section 4.4.3 into (4.4.25) to calculate the service reliability, since deep dependencies may exist between SVs of different services while only deep dependencies among SVs of the same service are considered in Section 4.4.3.

In addition, whenever a service succeeds, there must be a non-empty set of successful PSs, i.e., all PSs in a non-empty subset of the set of PSs in the sLDG (Section 4.2) of the service succeed, but not vice versa, i.e., when all PSs in a non-empty subset of the set of PSs in the sLDG of the service succeed, the service does not necessarily succeed even if all SVs hosted by the subset of PSs succeed. The services in Figure 4.10 can serve as an example. Suppose the sLDG of se_1, which includes all types of components, is shown in Figure 4.11. Then, the set of PSs in the sLDG is $\{ps_1, ps_2, ps_3\}$. We choose one of its non-empty subsets, e.g., $\{ps_1\}$, whose sLDG is marked in bold in Figure 4.11, and assume that all SVs hosted by the PSs in the subset succeed. Then, we can see in Figure 4.11 that, only one SV of se_3 succeeds while se_3 requires at least two SVs to succeed and no SV of se_2 succeeds. Hence, both se_2 and se_3 fail and consequently, se_1 fails.

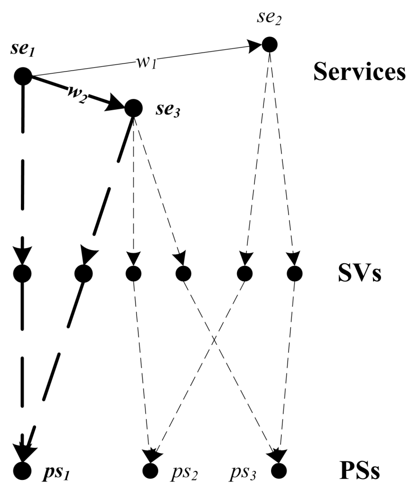


Figure 4.11: An example sLDG of service se_1.

Therefore, we separate all scenarios where a service possibly succeeds according to the subsets of the set of PSs in the service's sLDG. In each scenario, the service succeeds with a certain probability when the corresponding subset of PSs are considered successful. The probability can be 0, as the above example shows. Then, the service reliability can be represented as the sum of all probabilities, i.e.:

$$R_{se} = \sum_{i=1}^{n} P_{ps_i} R'_{se_i} \tag{4.4.31}$$

where n is the number of non-empty subsets of the set of PSs included in the service's sLDG, P_{ps_i} is the probability that all PSs in the ith subset succeed, and R'_{se_i} is the probability that se_i succeeds under the condition that all PSs in the ith subset succeed. Since PSs are independent with each other, P_{ps_i} can be calculated as the product of the reliability of PSs in the ith subset and the "unreliability", i.e., $1 - reliability$, of the remaining PSs in the set of PSs. And R'_{se_i} can be calculated by (4.4.25) where r'_{se} is calculated by adapting the service inner reliability assessment process proposed in Section 4.4.3.

With the sLDG in Figure 4.11 as an example, the reliability assessment process for a service is as follows:

1. get the sLDG $G(se)$ of the service, e.g., the sLDG of se_1 in Figure 4.11;
2. if only one service is included in $G(se)$, use the service inner reliability assessment method proposed in Section 4.4.3 to calculate the service reliability because the reliability of a service equals its inner reliability when the service has no service successors, and finish. Otherwise, perform the following steps;
3. get the set U of PSs included in $G(se)$. In the example, $U = \{ps_1, ps_2, ps_3\}$;
4. get all the non-empty subsets of U. Suppose totally b PSs are in U. As a set with n elements has 2^n subsets, there are totally $2^b - 1$ non-empty subsets of U;
5. for each non-empty subset U_h, $1 \le h \le 2^b - 1$, perform the following steps:
   a) calculate the probability P_{ps_h} that PSs in U_h succeed while all other PSs in U fail, with:
   $$P_{ps_h} = \prod_{ps \in U_h} R_{ps} \prod_{ps \in U/U_h} (1 - R_{ps}) \tag{4.4.32}$$
   where R_{ps} is the reliability of PSs;
   b) get the sLDG of U_h, $G(U_h)$, from $G(se)$, e.g., $G(\{ps_1\})$ is the bold part in Figure 4.11;
   c) calculate the probability R'_{se_h} that the service succeeds when only PSs in U_h succeed by (4.4.25):
   $$R'_{se_h} = r'_h R_{sty_h} \tag{4.4.33}$$
   where r'_h is the inner reliability of the service when PSs in U_h succeed and R_{sty_h} is the reliability of the style in $G(U_h)$. R_{sty_h} can be obtained by the style reduction introduced before. For example, suppose $U_1 = \{ps_1\}$, then $R_{sty_1} =$

$w_1 r'_{se_2} + w_2 r'_{se_3} = 0$, where $r'_{se_2} = 0$ because ps_2 and ps_3 are not in U_h, and $r'_{se_3} = 0$ because at most one SV of se_3 possibly succeed while se_3 needs at least two SVs to succeed. For each service in $G(U_h)$, the inner reliability can be calculated by adapting the inner reliability assessment process introduced in Section 4.4.3, specifically, adapting (4.4.20). The adapted part here is the calculation of $P_{u_{jl}}$ in (4.4.18), because PSs in $G(U_h)$ are assumed successful, i.e., with inner reliability of 1. Suppose $P'_{u_{jl}}$ is the probability that s_{jl} SVs on the lth PS included in U_h succeed in the jth scenario that i SVs succeed. Then, the inner reliability r'_{se_f} of the fth service, se_f, in $G(U_h)$, is:

$$r'_{se_f} = \sum_{i=k_f}^{n'_f} \sum_{j=1}^{d_i} \prod_{l=1}^{m_h} P'_{u_{jl}} \tag{4.4.34}$$

where k_f is the least number of required successful SVs and n'_f is the total number of SVs of se_f in $G(U_h)$, m_h is the number of PSs in U_h, d_i is the number of scenarios where i SVs of se_f hosted by the m_h PSs succeed. Provided the inner reliability of PSs in U_h is 1, (4.4.18) is adapted, by replacing r_{ps} with 1, into:

$$P'_{u_{jl}} = \begin{cases} (1-r_{sv})^{n_{hl}}, & \text{if } s_{jl} = 0 \\ (r_{sv})^{s_{jl}}(1-r_{sv})^{n_{hl}-s_{jl}}, & \text{if } s_{jl} > 0 \end{cases} \tag{4.4.35}$$

where r_{sv} is the inner reliability of SVs, n_{hl} is total the number of SVs hosted by the lth PS in U_h and s_{jl} is the number of succeeded SVs on the lth PS in U_h. The same as in Section 4.4.3, s_{jl} ($l = 1, 2, ..., m_h$) is obtained via the generalized algorithm for restricted weak composition generation proposed by Page [79], but with different inputs. For example, for the service se_3 in $G(\{ps_1\})$ shown in Figure 4.11, the integers are 2 and 3 as its SVs are configured as a 2-out-of-3 system, the number of parts is 1 as there is only one PS, and the restricted set is $\{0, 1\}$ as there is only one SV of service se_3 on the PS; and

6. calculate the service reliability by (4.4.31):

$$R_{se} = \sum_{h=1}^{2^b-1} P_{ps_h} R'_{se_h} \tag{4.4.36}$$

In the following, we go through the whole process for service se_1 with the sLDG shown in Figure 4.11 to describe how to assess the reliability of a service. Suppose se_1, se_2, and se_3 are respectively configured as a 1-out-of-1 system, a 1-out-of-2 system, and a 2-out-of-3 system, the inner reliability of PSs and SVs are $r_{ps} = 0.9$ and $r_{sv} = 0.8$, and $w_1 = w_2 = 0.5$.

1. The sLDG $G(se_1)$ of se_1 is shown in Figure 4.11;

2. Three services se_1, se_2, and se_3 are included in $G(se_1)$, continue;
3. We can get the set of PSs from Figure 4.11, $U = \{ps_1, ps_2, ps_3\}$;
4. U contains three PSs, correspondingly, the $2^3 - 1$ non-empty subsets of U are $U_1 = \{ps_1\}$, $U_2 = \{ps_2\}$, $U_3 = \{ps_3\}$, $U_4 = \{ps_1, ps_2\}$, $U_5 = \{ps_1, ps_3\}$, $U_6 = \{ps_2, ps_3\}$, and $U_7 = \{ps_1, ps_2, ps_3\}$;
5. For the above non-empty subsets, we employ $U_4 = \{ps_1, ps_2\}$ as an example to go through the steps $5a$ to $5c$.
   a) According to (4.4.32):

$$\begin{aligned} P_{ps_4} = & \prod_{ps \in U_4} R_{ps} \prod_{ps \in U/U_h} (1 - R_{ps}) \\ = & r_{ps_1} r_{ps_2} (1 - r_{ps_3}) \\ = & 0.9 * 0.9 * (1 - 0.9) \\ = & 0.081 \end{aligned}$$

   b) The $G(U_4)$ obtained from $G(se_1)$ is shown in Figure 4.12 as the bold part.

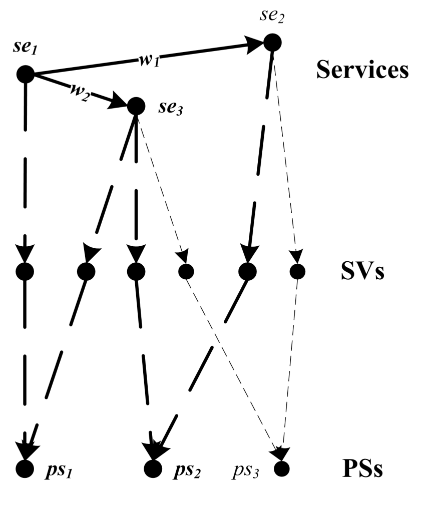


Figure 4.12: The sLDG of U_4.

   c) For U_4, we have:

$$R'_{se_4} = r'_4 R_{sty_4} \tag{4.4.37}$$

   where r'_4 and R_{sty_4} are calculated respectively by (4.4.34) and the style reduction.

   At first, we calculate r'_{se_2}, r'_{se_3}, and r'_4, i.e., r'_{se_1}, by (4.4.34). As shown in Figure 4.12, the number of SVs for se_1, se_2, and se_3 are respectively one, one, and two, and they are respectively configured as a 1-out-of-1 system, a 1-out-of-2 system, and a 2-out-of-3 system. So, the inputs and outputs of the generalized algorithm for restricted weak composition generation (Section 4.4.3) for above services are as shown in Table 4.2.

Table 4.2: Inputs and outputs of the generalized algorithm for restricted weak composition generation for services in $G(U_4)$.

		se_1	se_2	se_3
inputs	integer	1	1	2
	number of parts	1	1	2
	restricted sets	$\{0,1\}$	$\{0,1\}$	$\{0,1\}$, $\{0,1\}$
output weak compositions		(1)	(1)	$(1,1)$

The column for se_3 shows that se_3 has one weak composition $(1,1)$ which corresponds to the scenario that one SV on each of the two PSs succeeds, so, the number of scenarios is $d_2 = 1$, the number of all SVs on each PS is respectively $n_{41} = 1$ and $n_{42} = 1$, and the number of successful SVs on each PS is respectively $s_{11} = 1$ and $s_{12} = 1$. Besides, the number of PSs is $m_4 = 2$. Therefore, according to (4.4.35), $P'_{u_{11}}$ and $P'_{u_{12}}$ for se_2 are:

$$\begin{aligned} P'_{u_{11}} = P'_{u_{12}} =& (r_{sv})^{s_{11}}(1-r_{sv})^{n_{41}-s_{11}} \\ =& r_{sv} = 0.8 \end{aligned}$$

Since the SVs of the service se_3 are configured as a 2-out-of-3 system, i.e., $k_3 = 2$, and there are totally two SVs of se_3 included in $G(U_4)$, i.e., $n'_3 = 2$, according to (4.4.34), we have:

$$\begin{aligned} r'_{se_3} =& \sum_{i=k_3}^{n'_3} \sum_{j=1}^{d_2} \prod_{l=1}^{m_4} P'_{u_{1l}} \\ =& \sum_{i=2}^{2} \sum_{j=1}^{1} \prod_{l=1}^{2} P'_{u_{1l}} \\ =& P'_{u_{11}} P'_{u_{12}} \\ =& 0.8 * 0.8 = 0.64 \end{aligned}$$

Similarly, for se_1, we have $d_1 = 1$ and:

$$\begin{aligned}
P'_{u_{11}} =& r_{sv} \\
r'_4 =& r'_{se_1} \\
=& \sum_{i=1}^{1} \sum_{j=1}^{d_1} \prod_{l=1}^{1} P'_{u_{1l}} \\
=& r_{sv} = 0.8
\end{aligned}$$

And for se_2, we have $d_1 = 1$, and:

$$\begin{aligned}
P'_{u_{11}} =& r_{sv} \\
r'_{se_2} =& \sum_{i=1}^{1} \sum_{j=1}^{d_1} \prod_{l=1}^{1} P'_{u_{1l}} \\
=& r_{sv} = 0.8
\end{aligned}$$

Afterwards, we calculate R_{sty_4} by reducing the style in Figure 4.12. According to the proposed style reduction process, this style can be reduced as shown in Figure 4.13, where the choice style formed by se_2 and se_3 is replaced by a virtual service se_4.

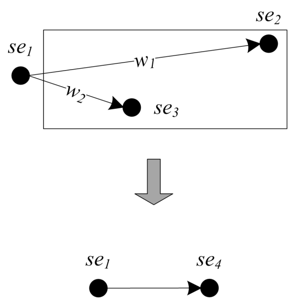


Figure 4.13: The reduction of the style in Figure 4.12.

Therefore, R_{sty_4} is calculated by:

$$\begin{aligned}
R_{sty_4} =& r_{se_4} \\
=& w_1 r'_{se_2} + w_2 r'_{se_3} \\
=& 0.5 * 0.64 + 0.5 * 0.8 \\
=& 0.72
\end{aligned}$$

Finally, we calculate R'_{se_4} by (4.4.33):

$$\begin{aligned}R'_{se_4} =& r'_4 R_{sty_4}\\ =&0.8 * 0.72\\ =&0.576\end{aligned}$$

6. By repeating the steps $5a$ to $5c$ for the other subsets, we get $P_{ps_1} = 0.009$, $P_{ps_2} = 0.009$, $P_{ps_3} = 0.009$, $P_{ps_5} = 0.081$, $P_{ps_6} = 0.081$, and $P_{ps_7} = 0.729$, and $R'_{se_1} = 0$, $R'_{se_2} = 0$, $R'_{se_3} = 0$, $R'_{se_5} = 0.576$, $R'_{se_6} = 0$, and $R'_{se_7} = 0.7424$. According to (4.4.36), we can get R_{se_1} by:

$$\begin{aligned}R_{se_1} =& \sum_{h=1}^{7} P_{ps_h} R'_{se_h}\\ =&0.6345216\end{aligned}$$

4.4.5 Application Reliability

Assuming the application has only one service that has no predecessors, given the inner reliability of each component and the deployment structure, the application reliability equals the reliability of the service without predecessors and can be assessed by the process proposed in Section 4.4.4.

Chapter 5

Implementation

To verify our framework, we implemented a prototype of the framework proposed in Chapter 4, including the three basic components. Furthermore, we implemented two more components which are necessary for testing applications. The prototype was implemented in Java for web-based applications deployed with Cloudify [32] on OpenStack [29] clouds.

5.1 Implementation of Dependency Analyzer

The dependency analyzer was implemented as an LDG generator to create the LDG of the application.

To obtain information of components and dependencies, the dependency analyzer calls three analyzers one after another, which are a Function Dependency Analyzer (FDA) and two Deployment Dependency Analyzers (DDAs). The process of generating the LDG of an application is shown in Figure 5.1. In the first step, the dependency analyzer creates an empty DAG as the LDG and calls the FDA to add service vertices and function dependency edges to the LDG. Afterwards, the dependency analyzer calls the first DDA to add SI vertices, VM vertices, edges of deployment dependencies between services and SIs and between SIs and VMs to the LDG. In the last step, the dependency analyzer calls the second DDA to add the PS vertices and edges of deployment dependencies between VMs and PSs to the LDG.

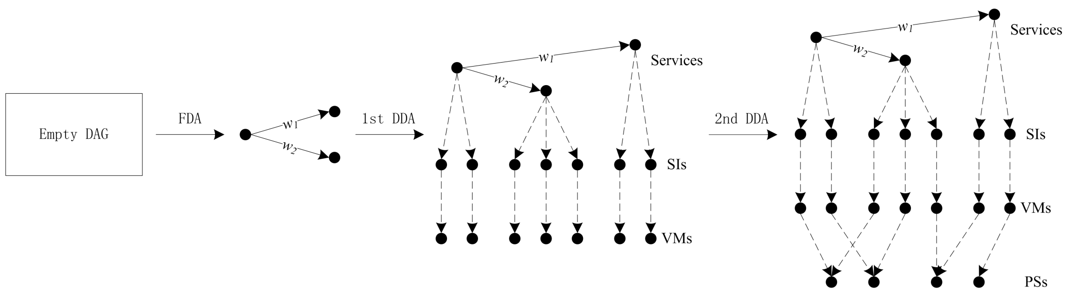


Figure 5.1: The process of generating an LDG.

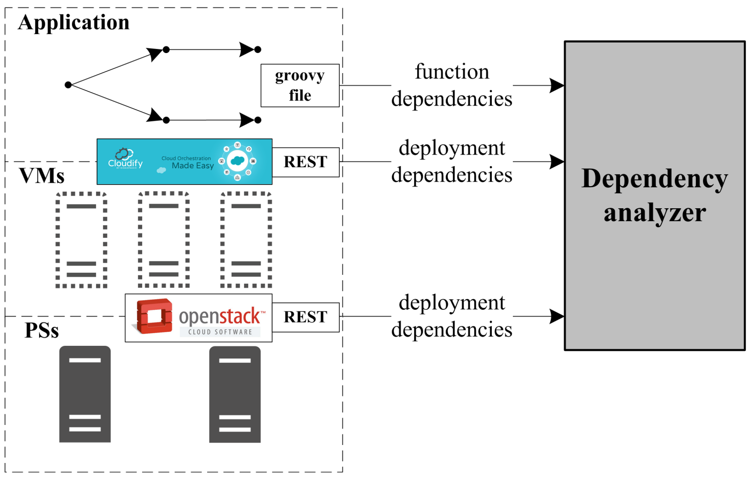
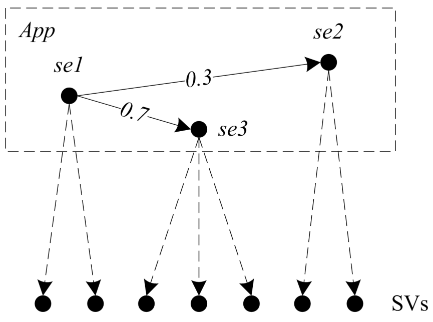


Figure 5.2: The overview of the implementation of the dependency analyzer.

An overview of the implementation architecture of the dependency analyzer is shown in Figure 5.2.

The FDA analyzes services and function dependencies between services according to the user defined application descriptor file [82]. The application descriptor file is written in Groovy [83] and used by Cloudify 2.6 and 2.7 for describing applications. The application descriptor file contains the name of the application and describes each service with two properties: a *name* and a *dependsOn* representing its interdependencies with other services. However, YAML-based blueprints instead of Groovy-based descriptor files are used by Cloudify to describe applications since version 3.0. Compared to Groovy-based files, it's not so intuitive or flexible for users to assign weights in YAML-based blueprints. So, we still use Groovy-based files for describing the structure of applications in our implementation and add two more properties, *redundancy* and *weight*, to service properties. The *redundancy* property represents the configuration of a service's SIs and the *weight* property indicates the weights of function dependencies between the services and its successors. The *name* property and the *redundancy* property are requisite and the *dependsOn* property and the *weight* property are optional since a service possibly has no successors. The four service properties are utilized by the function dependency analyzer to create vertices of services and edges of function dependencies. An example of the structure of an application and the corresponding descriptor file are shown in Figure 5.3. As Figure 5.3(a) shows, the application consists of three services: *se1*, *se2* and *se3*, who have respectively two, three and two SVs. The descriptor file of the application is shown in Figure 5.3(b), where we can see that, the SVs of the services are respectively configured as a 1-out-of-2 system, a 1-out-of-2 system, and a 2-out-of-3 system. Based on the descriptor file, for example, the created vertex of $se1$ is $v(service, se1, 1, 2)$ and the created edge between $se1$ and $se2$ is $e(se1, se2, function, 0.3)$.

The first DDA collects information of vertices and edges from Cloudify via the REST



(a) The structure of *App*

```
application {
      name="App"
      service {
            name = "se1"
            dependsOn = ["se2", "se3"]
            weight = ["0.3", "0.7"]
            redundancy = ["1", "2"]
      }
      service {
            name = "se2"
            redundancy = ["1", "2"]
      }
      service {
            name = "se3"
            redundancy = ["2", "3"]
      }
}
```

(b) The groovy-based descriptor file of *App*

Figure 5.3: An application and its descriptor file.

API, as shown in Figure 5.2. In Cloudify 3.1, an application is described by an YAML-based blueprint, which illustrates an application as a set of nodes and their relationships. Nodes are divided into several types, e.g., services, VMs, floating Internet Protocols (IPs), etc. And relationships are used to describe dependencies between nodes and determine the sequence of the creation of nodes when deploying an application. Nodes and relationships are instantiated into concrete instances during the deployment of the application. Based on the blueprint, an application can be deployed with a unique deployment ID, which is essential for identifying the application when collecting components used by the application. Among the REST APIs provided by Cloudify, we utilize the node instances API to get names of service, SIs, and VMs, as well as the deployment dependences between them. The analyzing process is shown in Algorithm 5.1. The algorithm takes the LDG updated by the FDA, the deployment IDentifier (ID) of the application, the list of services included in the LDG, and the IP and port of the Cloudify manager as inputs and outputs the updated LDG. In the first step, a node instances client is initialized by the IP and the port of the Cloudify manager. Afterwards, the DDA gets names of SIs of services in the LDG. Then, for each SI of a service, the DDA gets the corresponding VM. And, the DDA adds the vertices of SIs and VMs, and edges between services and their SIs as well as between SIs and their hosting VMs, to the LDG. In the last step, the updated LDG is returned to the dependency analyzer.

Based on the application deployment information and the VMs obtained by the first DDA, the second DDA updates the LDG with the details of PS vertices and the dependencies between VMs and PSs derived via the OpenStack REST API, as shown in Figure 5.2. The OpenStack REST API is utilized with the help of Apache jclouds [84]. For each VM, the name of its hosting PS is obtained according to its name.

Algorithm 5.1: Algorithm for analyzing deployment dependencies between services, SIs, and VMs

```
Input: the LDG $ldg$, deployment ID of the application $depId$, list of services $seList$,
       Cloudify manager IP $ip$ and port $port$
Output: The updated LDG
// initialize the node instances client by $ip$ and $port$
1  $nic \leftarrow new\ NodeInstancesClient(ip, port)$
2  foreach $se$ in $seList$ do
       // get the list of service instances
3      $siList \leftarrow nic.getNodeInstances(se, depId)$
4      foreach $si$ in $siList$ do
           // get the successor VM
5          $vm \leftarrow nic.getVM(si)$
6          $ldg.add(new\ Vertex(SI, si))$
7          $ldg.add(new\ Vertex(VM, vm))$
8          $ldg.add(new\ Edge(si, vm))$
9          $ldg.add(new\ Edge(se, si))$
10     end
11 end
12 return $ldg$
```

5.2 Implementation of Monitor

As described in Section 4.3, the monitor is responsible for monitoring PSs, VMs, and SIs and logging their states into log files every time unit. Figure 5.4 shows the implementation of the monitor, where states of PSs including the hypervisor state and the heartbeat are obtained via Ganglia, states of VMs are obtained via OpenStack REST API, and states of SIs are obtained via SSH commands.

To obtain the component states, the monitor needs metrics to tell the difference between success and failure states. According to the failures defined in Section 4.3, the monitoring methods and measures are implemented as follows.

PSs have two kinds of failures effects: unreachable and unable to host VMs. Based on this, PSs are monitored with two metrics: heartbeat and hypervisor state. Both metrics are obtained every time unit from Ganglia via an API [85]. Heartbeats are sent by Ganglia daemons on PSs to the Ganglia meta daemon in a time interval whose default value is 20 seconds. If no heartbeat from a PS is received in a multiple of the time interval, the PS is considered unreachable. The multiple of the time interval is the metric or threshold for the heartbeat. We use twice time intervals, i.e., 40 seconds, as the threshold, since the cloud system utilized by our implementation is small and not very complicated. For a specific

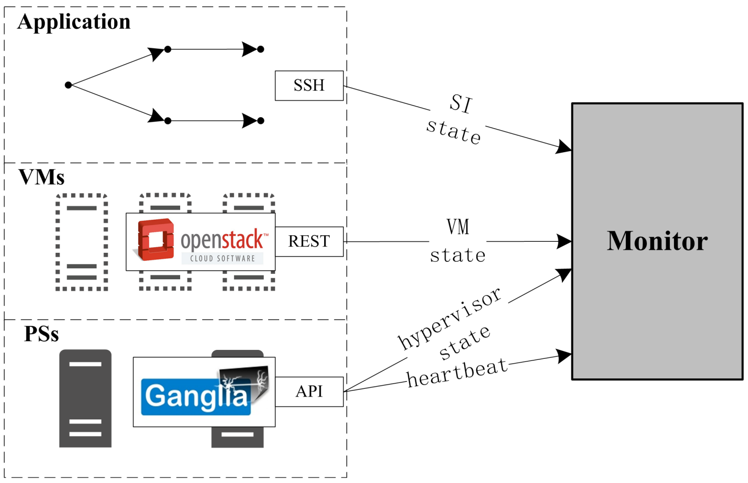


Figure 5.4: Implementation of the Monitor.

cloud platform, the threshold can be customized according to certain requirements. As introduced by Massie et al. [86], using four time intervals as the threshold to determine whether a PS is reachable can report the crashes in time without too many false positives. The monitor gets the time after the last heartbeat of each PS and compares it with the threshold. If the time is larger than the threshold, the PS is considered unreachable. The hypervisor in our implementation is KVM which is only running on compute nodes of the OpenStack cloud. The state of KVM is collected from each PS included in the LDG by a bash script and submitted to Ganglia in a time interval. If KVM is running, the hypervisor is considered alive and the PS is considered able to host VMs. When a PS is reachable and the hypervisor is running, it's in the success state, otherwise, it's in the failure state.

For VMs, possible failure effects are unreachable and unable to host SIs, based on which, VMs are determined to be failed or not by their states in OpenStack. The VM state can be one of *building, active, stopped, resized, deleted, error, suspended, paused*, etc. A VM with all above states except *active* is seen as failed. VM states are obtained with Apache jclouds via the REST APIs provided by OpenStack.

For SIs, the possible failure effect is that the corresponding process is inactive, based on which, SIs are monitored according to the process state. If the process is running and the running state can be obtained, i.e., the OS is working, the SI is in a success state, otherwise, the SI is failed. The SI state is obtained via SSH commands with the key file provided by OpenStack for the VM hosting the SI. States of SIs of the same service are logged in the same log file.

5.3 Implementation of Fault Injector

The fault injector is used in our case studies for simulating component failures at the beginning of every time unit before the start of the monitor. It is responsible for injecting failures to and recovering PSs, VMs and SIs according to the failure effects of components and the implementation of the monitor. The probability of injecting failures to a component is determined by the inner reliability of the component and all injections and recoveries are logged in log files. For a component, the fault injection means to set the component into the failure state if the component is determined to be failed in a time unit, otherwise, recover it from the failure state if necessary. Afterwards, the fault injector informs the monitor about the state of the component. The fault injector implementation for each kind of components is introduced below.

The failure effect of a PS is unreachable or unable to host VMs. Based on the fail-stop assumption in Section 4.4.1, the failure state has only one consequence that both the PS and VMs on it crash. An intuitive way to inject failures to a PS is to directly suspend it by remote commands which requires root privileges and the support of Wake-on-LANs (WoLs) for recovering physical machines. However, it is problematic when other users are also using the cloud, which is a typical usage scenario of IaaS clouds. As a result, we implement the fault injector in a simulative way which crashes PS by suspending VMs that are hosted by the PS and also used by the application. Afterwards, the fault injector informs the monitor to mark the PS as failed. The suspension and recovery of VMs, corresponding to the *suspend* and *resume* operation in the OpenStack cloud, are implemented via Apache jclouds.

VMs are monitored by their states in the OpenStack cloud introduced in Section 5.2. VMs are hosted by PSs, so, the failure injection and recovery of a VM work only if the hosting PS is in the success state. The failure injection and recovery of VMs are implemented in the same way as above.

The only failure effect of an SI is the inactiveness of its process. Similar to VMs, the failure injection and recovery of an SI also work only if the hosting VM is in the success state. The failure injection and recovery of an SI are corresponding to the *kill* and *restart* operation of the process of the SI, which are both implemented via SSH remote commands with the key file provided by OpenStack for the hosting VM.

5.4 Implementation of Tester

The tester is designed to test the application in each time unit after the fault injection. It first generates requests, and then sends the requests to the application, and at last, records the responses from the application. The tester in our framework is implemented only for web-based applications, specifically, websites.

Requests in our implementation are generated and sent to the application by Selenium (version 2.44) [87]. Selenium is an open source testing framework for web applications.

It automates the testing of web applications against different web browsers. Selenium provides an Integrated Development Environment (IDE) for Firefox [88] to record and replay user actions. The recorded actions can also be exported as test cases. A test case in the Selenium IDE involves several steps which are organized as an eXtensible HyperText Markup Language (XHTML) [89] table of three columns and several rows, which is written in a HyperText Markup Language (HTML) file. The Selenium IDE is only for Firefox, so, for generality, we use recorded session-based usage data of the application by AutoQUEST [90] rather than the Selenium IDE to generate test cases in the same format with test cases for the Selenium IDE. To translate sessions to Selenium test cases, we implement a command as a HTML plug-in of AutoQUEST. An example of the generated Selenium test cases is shown in Table 5.1. The first row shows the test case name and the following rows describe the steps/events of the test case including the command, target, and value from left to right. The *command* represents the event type, such as *click* means clicking a link in a website. Every test case starts from an *open* command to ensure that every test case can be executed alone. The *open* command opens a web page in the browser. The *target* represents a partial Uniform Resource Locator (URL) of the web page to be opened or the location of the target of the event which is represented by means of XML Path Language (XPath) [91]. For example, the target of the command *open* in Table 5.1 is "/lectures/software-testing-ws2012" which is the partial URL of the website without the IP address or port, both of which are in the head of the HTML file. And the location of the target of the command *type* in Table 5.1 is a part of the XPath of the target, ”//div/input[@id=’edit-search-block-form–2’]”, which means that a user inputs contents into a search box. The *value* means the content of the event, such as the value of the command *type* means that the user inputs "software testing" into the search box.

Table 5.1: An example test case of Selenium.

session-690		
open	/lectures/software-testing-ws2012	
waitForElementPresent	//div/input[@id=’edit-search-block-form–2’]	
click	//div/input[@id=’edit-search-block-form–2’]	
type	//div/input[@id=’edit-search-block-form–2’]	software testing
waitForElementPresent	//div[@id=’edit-actions’]/input[@id=’edit-submit’]	
click	//div[@id=’edit-actions’]/input[@id=’edit-submit’]	

A request in our implementation can be an event or a test case for the Selenium IDE including several events. Generated test cases are sent by Selenium to the website according to its URL and the responses are then logged into files.

5.5 Implementation of Reliability Analyzer

The reliability analyzer is designed to calculate the inner reliability and the reliability of components and applications according to the monitoring log files.

The monitoring log files of PSs, VMs, and SIs record the inner failures of components as well as the failures that are caused by the component's successors. For instance, the monitoring log file of a VM records the failures of the VM itself and also the failures of the VM that caused by the PS hosting it. To tackle this problem, we extract the inner failures of a VM or SI by comparing the failures of the component and its direct successor in each time unit. For example, suppose a failure of a VM is recorded in a time unit. If the hosting PS of the VM does not fail in this time unit, this failure is an inner failure of the VM, otherwise, this failure is not an inner failure of the VM. After the extraction, the inner reliability of PSs, VMs, and SIs is calculated respectively by:

$$r = 1 - \frac{N_f}{N_T} \tag{5.5.1}$$

where N_f and N_T are the number of the inner failures of the component and the number of entries of the monitoring log file of the component, which is also the number of time units for testing.

Afterwards, the reliability of services and application is calculated according to the inner reliability of PSs, VMs, and SIs by the method proposed in Section 4.4, where the generalized algorithm for restricted weak composition generation is implemented in Java according to the source code provided by Page [79]. All extracted failure data and assessed (inner) reliability are logged into report files (an example is shown in Appendix A.2.6).

Chapter 6

Case Studies

We conducted experiments to validate our framework and compare DEBRA with the existing representative methods. First, we introduce the basic setup of the experiments, including hypotheses, metrics, parameters, and the comparison process. Secondly, we describe the employed cloud platform, application and the deployment structures of the application. Afterwards, we show how we use our framework to test the application and use both DEBRA and other methods to evaluate the reliability of the application, and how we compare the results of all methods. At last, we discuss the results.

6.1 Setup

In this section, we introduce the setup of the case studies.

6.1.1 Hypotheses and Metrics

To verify our framework, we are going to test the following hypotheses:

- H1: The framework can assess the reliability of services and the application.
- H2: DEBRA can assess the application reliability when deployment structures include deep dependencies or not.
- H3: DEBRA gets more accurate results than existing methods.
- H4: DEBRA gets more precise results than existing methods with a small size of data.

For the hypothesis H1, we will accept it if quantitative reliability results of services and the application can be obtained by our framework. As to the hypothesis H2, we design or adapt several different deployment structures considering different types of redundancies and deep dependencies. Then, we check if DEBRA can get the reliability of the application deployed with these deployments. To test H3, at first, we choose five existing reliability assessment methods from related works and use them to assess the application reliability with different deployments. Afterwards, we test if the results of existing methods for different deployments are significantly different from DEBRA. If a method is significantly different from DEBRA, we compare their accuracies, otherwise, we test H4 to compare their precisions. The accuracy is measured by the Mean Absolute Error (MAE). For the hypothesis H4, if a method is not significantly different from DEBRA, we use the assessment results of the methods for a number of rounds to compare their precisions, otherwise, we don't test it. The precision is evaluated by the Standard Deviation (SD) (s_{eva}). The smaller the MAE or the s_{eva} is, the better the accuracy or the precision is. MAE and s_{eva} are calculated, respectively, by:

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |R_{eva_i} - R_{ref}| \tag{6.1.1}$$

$$s_{eva} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n} (R_{eva_i} - \overline{R_{eva}})^2} \tag{6.1.2}$$

with:

$$\overline{R_{eva}} = \frac{1}{n}\sum_{i=1}^{n} R_{eva_i} \tag{6.1.3}$$

where n is the number of test rounds, R_{eva_i} is the ith round's evaluated application reliability and $\overline{R_{eva}}$ is the mean of all rounds' results; R_{ref} is the baseline as the referential reliability of the application obtained by simulation. Details about R_{ref} will be introduced in Section 6.3.1.

6.1.2 Parameters and Comparison Process

We assume that the inner failure rate of all components are known and components of the same type have an identical inner failure rate. According to a Google report [92], a new cluster of 1,800 PSs can suffer from as many as 1,000 failures in the first year, which means an Annualized Failure Rate (AFR) of about 55.56%. And Vishwanath et al. observed an AFR of around 8% for data center machines [47]. As discussed in related works [47, 93], the infant mortality phase of PSs usually lasts for about one year, afterwards, PSs are in their useful life with lower and relatively constant failure rates which corresponds to an exponential reliability model. Therefore, it's reasonable to assume an AFR between 8% and 55.56%, corresponding to a daily failure rate between around 0.0002 and 0.0022. And we assume that the daily (inner) failure rate of a PS is 0.001. The inner failure rate of software components can be significantly different in practice [50] because of different developers or companies, so, we assume that all VMs and SIs have the same daily inner failure rate of 0.03 which can be easily adjusted in our case studies.

Based on the numeric parameters, assessment results of different methods can be compared quantitatively. The comparison process involves three steps, which are shown in

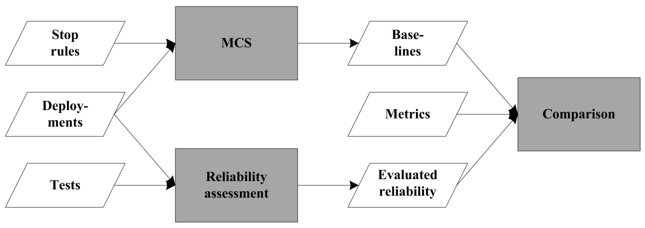


Figure 6.1: The comparison process.

Figure 6.1:

1. In the first step, we design several deployment structures of an application and, for each deployment, we obtain the baseline for comparison by the Monte Carlo Simulation (MCS) with certain stop rules which can be adjusted according to accuracy requirements.

2. In the second step, we deploy the application to a real-world cloud platform with different structures, test the application with each deployment and assess the reliability of the application. The assessment consists of a number of rounds, each of which is divided into several time units 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 only one state during a time unit, which is either success or failure, as introduced in Section 4.4.1. 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 according to its inner reliability, it will stay in the success state, or be recovered from the failure state to the success state. Otherwise, if a component is decided to be failed, it will keep the failure state or be injected with a failure if it is in the success state. When a component is in its failure state, all its predecessors (except service components) will also be injected with failures. 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 in log files. When all time units finish, the reliability of services and the application for this round is calculated by the reliability analyzer and other methods. When all rounds finish, the evaluated reliability is calculated by formula (6.1.3).

3. In the third step, based on baselines and evaluated reliability of each deployment, we compare different methods according to metrics proposed in Section 6.1.1.

6.1.3 Existing Methods for Comparison

Most of the existing methods for assessing cloud service/application reliability use only partial information of the whole deployment stack, such as only the application failures or PS failures. To the best of our knowledge, there is only one method (HPCRA [10]) that uses the whole deployment stack including services, VMs, PSs, and their relationships to evaluate the cloud application reliability (denoted by R_{app}). Based on the classification in Chapter 3 and layers of components in the LDG (from top to bottom) a method considers, we choose and adapt the following representative methods from existing methods to compare them with DEBRA:

- LOBRA, is a log-based reliability assessment method proposed by Banerjee et al. [71, 72]. LOBRA assesses the application reliability based on the application failure rate without regard to its deployment structure or components failure rates. LOBRA is proposed to assess the reliability of SaaS applications using their access logs, and the application reliability is calculated by:

$$R_{app} = 1 - \frac{f_e}{n_e} \tag{6.1.4}$$

  where f_e and n_e are the number of failed and total log entries, respectively. In our framework, we use the log files generated by the monitor as access log files to assess every component's reliability, at the same time, the testing results (application failures) are also logged. As a result, LOBRA can directly be used to assess the application reliability using the testing log files from our framework.

- SEBRA, is a service-based reliability assessment method adapted based on the method proposed by Wang et al. [20]. SEBRA evaluates the application reliability based on service component failure rates and the application structure irrespective of VMs or PSs. Wang et al. [20] evaluated the reliability of an N-redundant service (R_{se}) and a composite service (application) consisting of a series of K services (R_{app}), respectively, by:

$$R_{se} = 1 - \prod_{i=1}^{N}(1 - R_{si_i}) \tag{6.1.5}$$

$$R_{app} = \prod_{i=1}^{K} e^{-w_i \lambda_i} \tag{6.1.6}$$

  where λ_i is the failure rate of the ith service and w_i is the probability that the ith service is required by the application. If $w_i = 1$ for every $i = 0, 1, ..., N-1$, then (6.1.6) turns into $R_{app} = \prod_{i=1}^{K} R_{se_i}$. With our framework, the reliability of services can be evaluated by formula (5.5.1) from monitoring data, and then can be used by SEBRA in (6.1.5) and use (6.1.6) to calculate the application reliability.

- HCMRA, is a hierarchical correlation model proposed by Qiu et al. [12]. HCMRA assesses the cloud service reliability based on the reliability of VMs and PSs. Qiu et al. define the service reliability as the probability that at least one VM used by the service is available. When a service uses K ($K \geq 1$) PSs, each of which hosts N_m ($m = 1, ..., K$) VMs, the states of every PS and its hosted VMs are modeled by a Markov process, using the VMs inner failure rates (λ_{vm}) and the PS inner failure rate (λ_{ps}) as well as their repair rates. They assume that VMs on the same PS have the same inner failure rate and repair rate. In this thesis, as introduced in Section 6.1.2, failed components will be repaired in the next time unit if it's determined successful. Therefore, the repair rates of VMs and the PS can be represented by $1 - \lambda_{vm}$ and $1 - \lambda_{ps}$, respectively, and the service reliability is evaluated by:

$$R_{se} = 1 - \prod_{m=1}^{K} p_m(0) \tag{6.1.7}$$

with:

$$p_m(0) = (\frac{\lambda_{vm}}{1-\lambda_{vm}})^{N_m}\pi_{N_m} + (1+\frac{\lambda_{vm}}{1-\lambda_{vm}})^{N_m}\frac{\lambda_{ps}}{1-\lambda_{ps}}\pi_{N_m} \tag{6.1.8}$$

$$\pi_{N_m} = [\sum_{n=0}^{N_m} C_{N_m}^{n}(\frac{\lambda_{vm}}{1-\lambda_{vm}})^n(1+\frac{\lambda_{ps}}{1-\lambda_{ps}})]^{-1} \tag{6.1.9}$$

where $p_m(0)$ is the probability that no VM on the mth PS is available, and π_{N_m} is the steady probability that all N_m VMs on the mth PS are available. For example, the reliability of a service using two VMs on one PS will be $1 - [\lambda_{vm}^2(1-\lambda_{ps}) + \lambda_{ps})]$ according to (6.1.7), (6.1.8), and (6.1.9), with $K = 1$ and $N_1 = 2$. HCMRA was designed only for the cloud service reliability rather than the application reliability (i.e., no application structure was considered). For comparison, we use the service reliability obtained by HCMRA as the input to DEBRA for calculating the application reliability.

- ARMRA, is an analytical reliability model for CCSs proposed by Faragardi et al. [11]. ARMRA evaluates cloud service reliability based on only PSs failure rates. Faragardi et al. assume that services are perfect, i.e., fully reliable. The cloud service reliability is calculated as the product of the reliability of every PS and the physical network link included in the system. Besides, the PS failure rate is the sum of failure rates of the memory, hard disks, the RAID controller and processors. As assumed in Section 4.3, we include physical network failures in PS failures, and the PS failure rate is considered given. Thus, the application reliability can be calculated with ARMRA by:

$$R_{app} = \prod_{i=1}^{n} R_{ps_i} \tag{6.1.10}$$

where R_{ps_i} is the reliability of the ith PS and n is the number of PSs used by the application.

- HPCRA, is adapted according to the assumptions in Section 4.4 based on a high performance computing application reliability assessment method proposed by Thanakornworakij et al. [10]. HPCRA assesses the cloud application reliability based on all components' reliability while it focuses on the correlation of failures among software and/or hardware components and does not consider redundancies. However, correlated failures of PSs are rare and only obvious due to network failures [48], and correlated failures of services without load sharing are not as significant for normal cloud applications as for high performance applications. As a result, HPCRA can be used to calculate the application reliability by:

$$R_{app} = \prod_{i=1}^{n} r_i \tag{6.1.11}$$

which means that the application reliability is the product of the inner reliability (r_i) of all n components. As mentioned by Thanakornworakij et al. [10], their method was not for applications with redundancies. But we try to utilize HPCRA for all scenarios for comparison, as redundancy is a normal fault tolerance method for cloud applications.

6.2 Platform and Application

The cloud platform for deploying, managing, and testing applications is a Cloudify (Version 3.1) PaaS platform on top of an OpenStack (Juno) IaaS cloud. The platform structure is shown in Figure 6.2. The OpenStack cloud consists of a controller node, a network node and five identical compute nodes. Physical specifications of all nodes, including OS, CPU, memory, and storage capacity, are listed in Table 6.1.

Table 6.1: Physical specifications of OpenStack cloud nodes.

Role	OS	CPU	Memory	Storage
Controller	Ubuntu 14.04 64 bit	Intel Core 2 Duo, 3 GHZ	3.4 GB	38 GB HDD
Network	Ubuntu 14.04 64 bit	Intel Core 2 Duo, 3 GHZ	3.4 GB	38 GB HDD
Compute	Ubuntu 14.04, 64 bit	Intel Core i7, 3.4 GHZ	32 GB	500 GB HDD 512 GB SSD

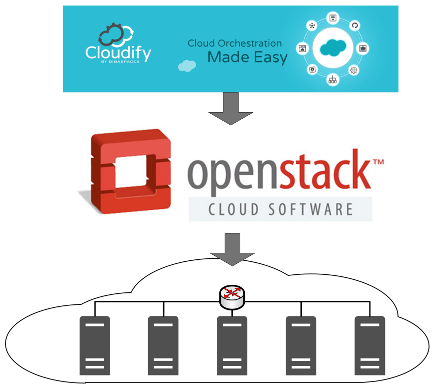


Figure 6.2: The structure of the cloud platform.

The application under test in this case study is the website of the Software Engineering for Distributed Systems group at the Institute of Compute Science, University of Goettingen [94]. The website is composed of three components: an Apache [95] load balancer, an Apache HTTP server hosting a Drupal [96] website, and a MySQL [97] database.

With the component-based application, we test if DEBRA can assess the application reliability with different deployment structures and explore the impact of deployment structures to the application reliability. To this aim, six deployment structures are designed considering resource limits, dependency, redundancy, and practical significances: *Minimal*, *Simple*, *Average-1*, *Average-2*, *Medium*, *Complex* and *K-out-of-n*. The deployment structures are shown in Figure 6.3, and details of the deployments are as follows:

- The *Minimal* deployment has all three components hosted in one VM to evaluate the reliability that can be achieved with minimal resources, as Figure 6.3(a) shows.
- The *Simple* deployment assigns one VM to each website component to evaluate the reliability of the most intuitive deployment structure. To eliminate the impact of dependencies, three VMs are hosted by three different PSs, as Figure 6.3(b) shows.
- The *Average-1* deployment is adapted from the application structure proposed by Thanakornworakij et al. [10]. The structure is originally for typical MPI applications. The application proposed by Thanakornworakij et al. was deployed to nine VMs evenly hosted by three PSs. The authors assumed that the application fails if more than one component (including PSs, hypervisors, VMs, and application instances) fails. Average-1 can also be seen as an application working under heavy workload. To adopt this structure, we view the website as a service and deploy it as a 9-out-of-9 system, as Figure 6.3(c) shows. Average-1 is also used to check if our framework can assess the reliability of a n-out-of-n service.
- The *Average-2* deployment is adapted from a structure proposed by Qiu et al. [12], where a service uses six VMs evenly distributed on three PSs. The service fails if all

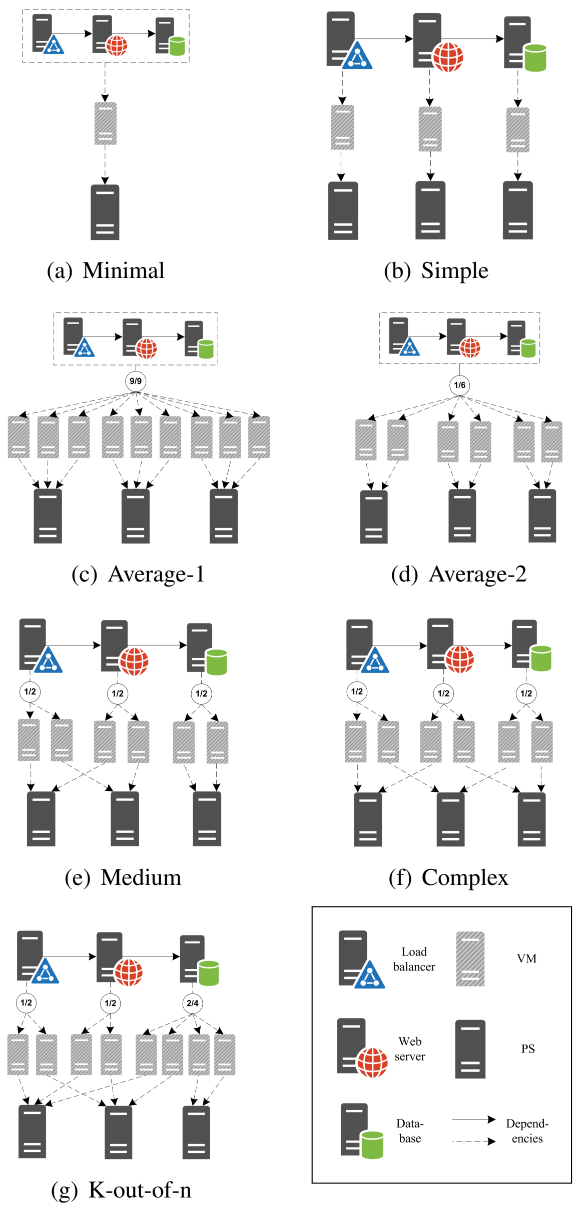


Figure 6.3: Application deployment structures.

SIs fail. The same to Average-1, we treat our application as a service and deploy it to six VMs, as Figure 6.3(d) shows.

- The *Medium* deployment is employed to improve the reliability of the Simple deployment by redundancy. To this aim, we deploy each website component with two SIs which are configured as 1-out-of-2 systems. Practically, redundant SIs of the same service could possibly be deployed on the same PS, when, e.g., not enough PSs are available. Meanwhile, the two instances on the same PS in this case are used to verify the impact of deep dependencies between SIs of the same service, as Figure 6.3(e) shows.
- The *Complex* deployment has exactly the same amount of SIs as the Medium but with a different distribution of the VMs, where SIs of the same service are on different PS. Complex is a common deployment in practice. We use it to evaluate the impact of deep dependencies between SIs of different services, as Figure 6.3(f) shows.
- The *K-out-of-n* deployment is also based on Medium. The difference is that two more MySQL instances are added, and the four MySQL instances are organized as a 2-out-of-4 system, as Figure 6.3(g) shows. Practically, this deployment is used for websites with massive throughput of data. This deployment is employed to verify the impact of the k-out-of-n redundancy to the application reliability, and also to check if DEBRA can assess the reliability of an application with k-out-of-n services.

6.3 Results

In this section, we introduce the results of baselines, the assessment, and the comparison.

6.3.1 Baselines

To compare results of different reliability assessment methods, we need an accurate enough baseline for each deployment. In this section, we utilize the MCS to get the baselines of deployments. The MCS of each deployment involves a number of rounds, and within each round, the application state is either *success* or $failure$, corresponding to the reliability of 1 or 0, respectively. For an n-round MCS, according to (6.1.3), the simulated reliability ($\overline{R_{sim}}$) of the application is the mean ($\bar{\mu}_n$) of all rounds' simulated reliability:

$$\overline{R_{sim}} = \bar{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} R_{sim_i} \tag{6.3.1}$$

where R_{sim_i} is the simulated application reliability of the ith round and can only be 0 or 1.

The number of rounds is determined by the stop rule of the MCS. As "accurate enough" cannot be used as a mathematical constraint, we define an "accurate enough" baseline mathematically as a baseline meets conditions *a*) within the 99% confidence interval of the application reliability and *b*) with at least five significant figures after the decimal point.

For the condition *a*, the 99% confidence interval of *n*-round MCS results is of the form [98, p. 19]:

$$\overline{R_{sim}} \pm 2.58 s_{sim}/\sqrt{n} \tag{6.3.2}$$

where s_{sim} is the SD of simulation results, with:

$$s_{sim} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(R_{sim_i} - \overline{R_{sim}})^2} \tag{6.3.3}$$

Equation (6.3.2) indicates that, with a 99% confidence level, $\overline{R_{sim}}$ and the real application reliability R_{rea} meet:

$$|R_{rea} - \overline{R_{sim}}| <= 2.58 s_{sim}/\sqrt{n} \tag{6.3.4}$$

Regarding the condition *b*, to get a five-significant-figure estimate (we assume that every deployment's reliability ranges from 0.1 to 1.0), $\overline{R_{sim}}$ and R_{rea} should meet [99, p. 4]:

$$|R_{rea} - \overline{R_{sim}}| < 5 \times 10^{-6} \tag{6.3.5}$$

Combining (6.3.4) and (6.3.5), the baseline will be accurate enough when:

$$2.58 s_{sim}/\sqrt{n} < 5 \times 10^{-6} \tag{6.3.6}$$

which is used as the stop rule of the MCS for baselines.

For calculating s_{sim}, Owen [98, p. 21] gives a one-pass algorithm, which defines $S_n = \sum_{i=1}^{n}(R_{sim_i} - \overline{R_{sim}})^2$ and $s_{sim}^2 = S_n/(n-1)$, and starts with $\overline{R_{sim}} = R_{sim_1}$ and $S_1 = 0$. Using δ_i and $\bar{\mu}_i$ for intermediate results, the algorithm makes the following updates:

$$\begin{aligned} \delta_i &= R_{sim_i} - \bar{\mu}_{i-1} \\ \bar{\mu}_i &= \bar{\mu}_{i-1} + \frac{1}{i}\delta_i \\ S_i &= S_{i-1} + \frac{i-1}{i}\delta_i^2 \end{aligned} \tag{6.3.7}$$

where $i = 2, 3, ..., n$ and n is the number of rounds. As R_{sim_i} can only be 0 or 1, we simplify the update of $\bar{\mu}_i$ in the original algorithm according to (6.3.1):

$$\bar{\mu}_i = \frac{\sum_{j=1}^{i} R_{sim_j}}{i} \tag{6.3.8}$$

The proof of the correctness of the adapted algorithm is in Appendix A.1. Then, based on the adapted algorithm and the stop rule, we design the MCS algorithm for baselines as shown in Algorithm 6.1. For each deployment, we initialize variables in the first round. In the following rounds, we use the MCS to simulate the state of the application and update δ, R_{sim} and S according to the simplified algorithm and check if s_{sim} satisfies the stop rule. The algorithm of simulating the application state in each round is shown in Algorithm 6.2. At the beginning, the algorithm determines the state of non-service components, i.e., PSs, VMs, and SIs, according to their inner reliability. Then, the algorithm determines the states of VMs according to their hosting PSs. Afterwards, the algorithm determines the states of SIs and according to the deployment structure, determines the states of services. At the end, the algorithm returns the state of the application.

Using Algorithm 6.1 and Algorithm 6.2, we get that baselines of the Minimal, Simple, Average-1, Average-2, Medium, Complex, and K-out-of-n deployments: 0.886037, 0.832768, 0.338578, 0.999999, 0.988659, 0.989542, 0.992138, respectively. To visually show the convergence of the algorithm, 100 data points from the first 100,000 simulation rounds for each deployment are sampled and plotted in Figure 6.4.

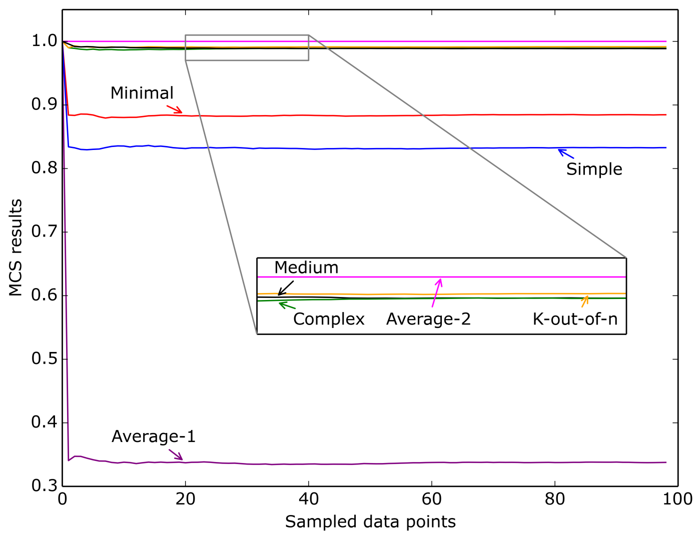


Figure 6.4: Simulated application reliability for all deployments.

Baselines and Figure 6.4 suggest that:

1. The Average-2 deployment is most reliable, with a reliability of six nines, and the Average-1 deployment is least reliable with a reliability of 0.338578. However, Average-2 deploys all services together in every VM, which requires larger (i.e., more expensive) VMs than other deployments, like Simple. If the three services are deployed separately on smaller VMs like the Complex deployment, the application

Algorithm 6.1: MCS algorithm for baselines

```
Input: deployment $dep$, the target error $err_{tgt}$ as the stop rule
Output: Baseline for the deployment
// initialization
$round \leftarrow 1$
$mcsb \leftarrow$ new MCSBaseline(dep)
$error \leftarrow 1$
$\delta \leftarrow 0$
$n_{suc} \leftarrow 0$
// the first round
$success \leftarrow mcsb.simulate()$ // see Algorithm 6.2
$\overline{R_{sim}} \leftarrow success$
$n_{suc} \leftarrow n_{suc} + success$
$S \leftarrow 0$
$s_{sim} \leftarrow 0$
$round \leftarrow round + 1$
// round 2,3,...
while $error > err_{tgt}$ do
    $success \leftarrow mcsb.simulate()$
    $n_{suc} \leftarrow n_{suc} + success$
    // update $\delta$, $\overline{R_{sim}}$, and $S$
    $\delta \leftarrow success - \overline{R_{sim}}$
    $\overline{R_{sim}} \leftarrow n_{suc}/round$
    $S \leftarrow S + ((round - 1)/round) * \delta^2$
    // calculate $s_{sim}$ and $error$
    $s_{sim} \leftarrow \sqrt{S/(round - 1)}$
    if $S \neq 0$ then
        $error \leftarrow 2.58 * s_{sim}/\sqrt{round}$
    end
    $round \leftarrow round + 1$
end
return $\overline{R_{sim}}$
```

Algorithm 6.2: Simulation algorithm for application states

```
Input: deployment $dep$, list of non-service components $comList$, the LDG $ldg$, VM list
       $vmList$, services list $seList$, inner reliability of non-service components $rMap$
Output: state/reliability of the application, 0 or 1
   // initialize the application state
1  $succeeded \leftarrow true$
   // determine the state $st_{com}$ of all PSs, VMs, and SIs
   // $st_{com}$ can be one of $st_{ps}$, $st_{vm} or st_{si}$
2  foreach component com in comList do
3      create a random double $d_{com} \in [0,1)$
4      $r_{com} \leftarrow rMap.get(com)$
5      if $d_{com} > r_{com}$ then
6          $st_{com} \leftarrow false$
7      else
8          $st_{com} \leftarrow true$
9      end
10 end
   // renew the state of every VM $st_{vm}$
   // according to the corresponding PS state $st_{ps}$
11 foreach vm in vmList do
12     get the corresponding PS $ps$ from $ldg$
13     if $st_{ps} = false$ then
14         $st_{vm} \leftarrow false$
15     end
16 end
17 foreach se in seList do
       // initialize the number of failed SIs
18     $n_{fsi} \leftarrow 0$
19     get the corresponding instance list $siList$
20     foreach si in siList do
21         get the corresponding VM $vm$ from $ldg$
           // renew the service instance state $st_{si}$
           // according to its corresponding VMs state
22         if $st_{vm} = false$ then $st_{si} \leftarrow false$
           // count the number of failed instances $n_{fsi}$
23         if $st_{si} = false$ then
24             $n_{fsi} \leftarrow n_{fsi} + 1$
25         end
26     end
       // simplified process of determining the application state
27     if se is a k-out-of-n-ed service then
28         if $n_{fsi} > (n-k)$ then $succeeded \leftarrow false$
29     else if $n_{fsi} = siList.size()$ then
30         $succeeded \leftarrow false$
31     end
32 end
33 return $succeeded$ ? 1 : 0
```

reliability will decrease by $0.999999 - 0.989542 = 0.010457$, at the same time, the cost of VMs will also be reduced in the light of VM prices.

2. The Minimal deployment is more reliable than the Simple deployment, even though the later uses more VMs and PSs. It indicates that increasing the amount of components without regard to redundancies will decrease the system reliability, if no performance issues are considered.
3. Comparing to the Simple deployment, both the Medium and Complex deployments have two VMs for every service and have higher reliability. It indicates that redundancy can improve the application reliability even with the same amount of physical resources. The degree of improvement is different because of the different allocation of redundant VMs on PSs.
4. The reliability of the Medium deployment is lower than the Complex deployment, which is lower than the K-out-of-n deployment. It suggests that 1) deep dependency decreases the application reliability, i.e., optimizing the deployment structure can improve the application reliability, and 2) assigning more VMs on different PSs to a service will increase its reliability and hence the application reliability.
5. If we use the R_{ref} of the Medium, Complex, and K-out-of-n deployments for comparing, by removing deep dependencies between instances of the same service, the Complex deployment is about 0.09% more reliable than the Medium deployment. The improvement may be insignificant, but no further resources are required and, above all, the SPoF is eliminated in the Complex deployment.

6.3.2 Assessment Results

In Section 6.3.1, we showed how we determined the R_{ref} for each deployment by using the MCS. In this section, we will introduce the process of reliability assessment and the results of all methods for all deployments.

For each deployment, we test the application for 30 25-hour rounds with every minute as a time unit (1,500 time units in each round) and collect the monitoring data with our framework. Fault injections, state information of all components, and testing results are recorded in files. Based on the log files, the reliability analyzer assesses the reliability of all components as well as the application and creates report files. The report file of the first test round of the Simple deployment is shown in Figure 6.5 (more examples of log files can be found in Appendix A.2). A report file is constituted by four parts:

- Setup: includes inner failure rates (denoted by *IFR*) and corresponding inner reliability (denoted by *IRE*) of components, based on which the reliability (denoted by *RE*) of the components as well as the application is assessed by the reliability analyzer;
- Fault injector: includes the total amount of injections, the injected inner failure rate and the corresponding injected inner reliability of each component. Furthermore, the

```
Setup
--------------------------------------------------------------
                IFR             IRE              Assessed RE
--------------------------------------------------------------
ps_1            0.00100000      0.99900050       0.99900050
ps_2            0.00100000      0.99900050       0.99900050
ps_3            0.00100000      0.99900050       0.99900050
apache_vm_1     0.03000000      0.97044553       0.96947557
drupal_vm_1     0.03000000      0.97044553       0.96947557
mysql_vm_1      0.03000000      0.97044553       0.96947557
----apache_1    0.03000000      0.97044553       0.94082324
apache          0.06100000      0.94082324       0.83276816
----drupal_1    0.03000000      0.97044553       0.94082324
drupal          0.06100000      0.94082324       0.88514837
----mysql_1     0.03000000      0.97044553       0.94082324
mysql           0.06100000      0.94082324       0.94082324
Application                                      0.83276816

Fault Injector: 1500 injections
--------------------------------------------------------------
                Injected IFR    Injected IRE     Assessed RE
--------------------------------------------------------------
ps_1            0.00133422      0.99866667       0.99866667
ps_2            0.00000000      1.00000000       1.00000000
ps_3            0.00066689      0.99933333       0.99933333
apache_vm_1     0.03252319      0.96800000       0.96670933
drupal_vm_1     0.03528181      0.96533333       0.96533333
mysql_vm_1      0.02977216      0.97066667       0.97001956
----apache_1    0.03183472      0.96866667       0.93641911
apache          0.06569214      0.93641911       0.83298745
----drupal_1    0.02771384      0.97266667       0.93894756
drupal          0.06299565      0.93894756       0.88954555
----mysql_1     0.02360987      0.97666667       0.94738577
mysql           0.05404891      0.94738577       0.94738577
Application                                      0.83298745

Monitor: 1500 logs
--------------------------------------------------------------
                Monitored RE                     Assessed RE
--------------------------------------------------------------
ps_1            1-2/1500=0.99866667              0.99866667
ps_2            1-0/1500=1.00000000              1.00000000
ps_3            1-1/1500=0.99933333              0.99933333
apache_vm_1     1-50/1500=0.96666667             0.96670933
drupal_vm_1     1-52/1500=0.96533333             0.96533333
mysql_vm_1      1-45/1500=0.97000000             0.97001956
----apache_1    1-95/1500=0.93666667             0.93641911
apache          1-95/1500=0.93666667 (IRE)       0.83298745
----drupal_1    1-91/1500=0.93933333             0.93894756
drupal          1-91/1500=0.93933333 (IRE)       0.88954555
----mysql_1     1-77/1500=0.94866667             0.94738577
mysql           1-77/1500=0.94866667 (IRE)       0.94738577

Test: 15000 requests
--------------------------------------------------------------
                Tested RE                        Assessed RE
--------------------------------------------------------------
Application     1-2510/15000=0.83266667          0.83298745
```

Figure 6.5: The analysis report of the first test round of the Simple deployment.

reliability is assessed by DEBRA based on the injected inner failure rate and the inner reliability;

- Monitor: includes the total amount of monitoring log entries and the amount of monitored failures and the assessed reliability of components; and
- Test: includes the total amount of requests sent by the tester, the amount of failed tests and the tested reliability of the application.

The reliability of each round of each deployment is evaluated by all other methods based on the report files. The assessment results for each deployment are shown in table 6.2 and the results for all rounds are shown in Figure 6.6 to 6.12 (*REFER* denotes the baseline). Table 6.2 and Figure 6.6 to 6.12 show that:

- For the Minimal deployment, ARMRA gets results very close to 1.0 (with $\overline{R_{eva}} = 0.99880000$), HCMRA gets higher results and SEBRA gets lower results than the baseline while DEBRA and HPCRA get the same results which are very close to LOBRA's results and fluctuate around the baseline.
- For the Simple deployment, similar to the Minimal deployment, ARMRA gets results very close to 1.0 (with $\overline{R_{eva}} = 0.99722500$), HCMRA gets higher results than the baseline while DEBRA and HPCRA get the same results which are close to LOBRA's and SEBRA's results and fluctuate around the baseline.
- For the Average-1 deployment, SEBRA, ARMRA, and HCMRA get close results to each other which are 1.0 or close to 1.0 and much higher than the baseline while DEBRA gets the same results as HPCRA which are close to LOBRA's results and fluctuate around the baseline.
- For the Average-2 deployment, HPCRA gets much lower results than the baseline, ARMRA gets results slightly lower than the baseline and slightly higher than the HCMRA's results while DEBRA ($R_{eva} = 0.99999751$), LOBRA ($R_{eva} = 1.00000000$) and SEBRA ($R_{eva} = 1.00000000$) get nearly the same results to each other and as the baseline (0.999999).
- For the Medium deployment, HPCRA gets much lower results than all other methods and the baseline, ARMRA and HCMRA get close results which are higher than the baseline, DEBRA, LOBRA, SEBRA, and HCMRA get close results which are also close to the baseline.
- For the Complex deployment, HPCRA gets much lower results than all other methods and the baseline, ARMRA and HCMRA get close results which are higher than the baseline while DEBRA, LOBRA, and SEBRA get close results which fluctuate around the baseline.
- For the K-out-of-n deployment, HPCRA gets much lower results than all other methods and the baseline, ARMRA and HCMRA get close results which are higher than the baseline while SEBRA gets slightly higher results than the baseline and DEBRA and LOBRA get close results which fluctuate around the baseline.

Table 6.2: Assessment results for deployments.

Deployment	R_{ref}	$\overline{R_{eva}}$					
		DEBRA	LOBRA	SEBRA	HCMRA	ARMRA	HPCRA
Minimal	0.886037	0.88473215	0.88440000	0.82954334	0.96801919	0.99880000	0.88473215
Simple	0.832768	0.83376388	0.83322222	0.83364761	0.90966612	0.99722500	0.83376388
Average-1	0.338578	0.33660991	0.33606667	1.00000000	0.99675905	0.99684805	0.33660991
Average-2	0.999999	0.99999751	1.00000000	1.00000000	0.99463018	0.99729121	0.48677148
Medium	0.988659	0.98949129	0.98895556	0.98946678	0.99622572	0.99702557	0.69480509
Complex	0.989542	0.98979754	0.98995556	0.98978590	0.99719622	0.99698074	0.69896738
K-out-of-n	0.992138	0.99241110	0.99202222	0.99324609	0.99732751	0.99753518	0.62231928

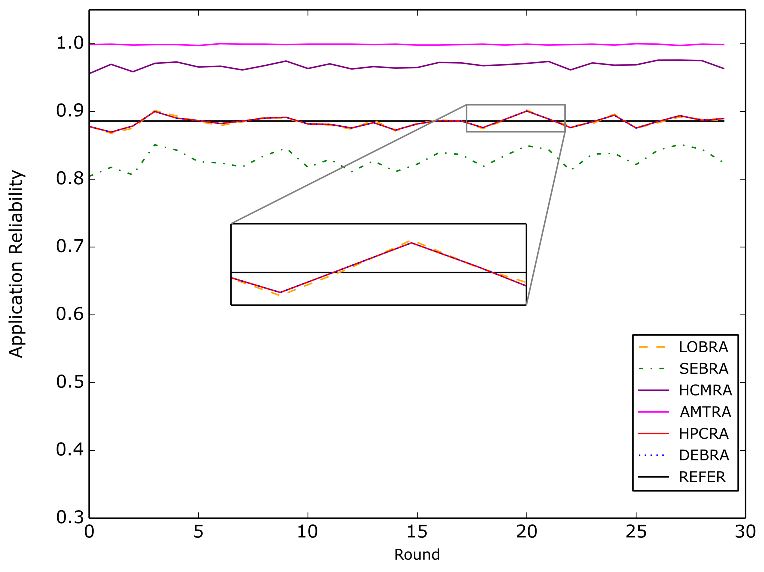


Figure 6.6: Comparison results for the Minimal deployment.

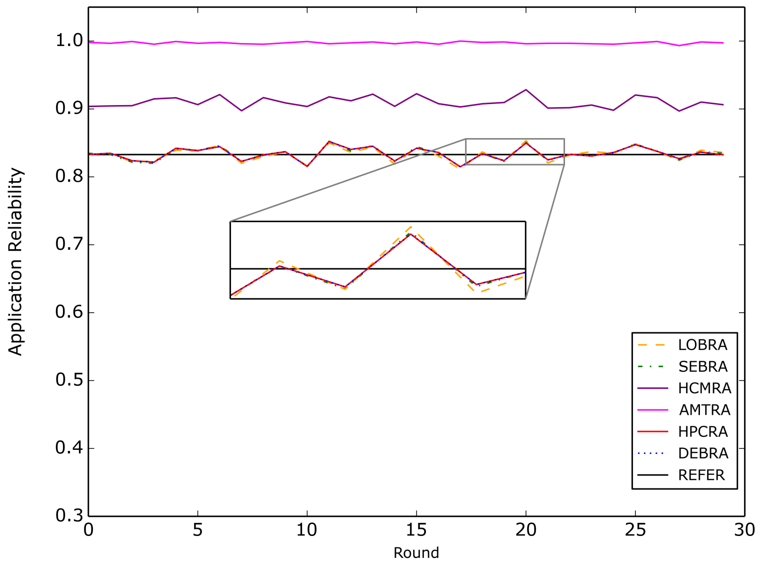


Figure 6.7: Comparison results for the Simple deployment.

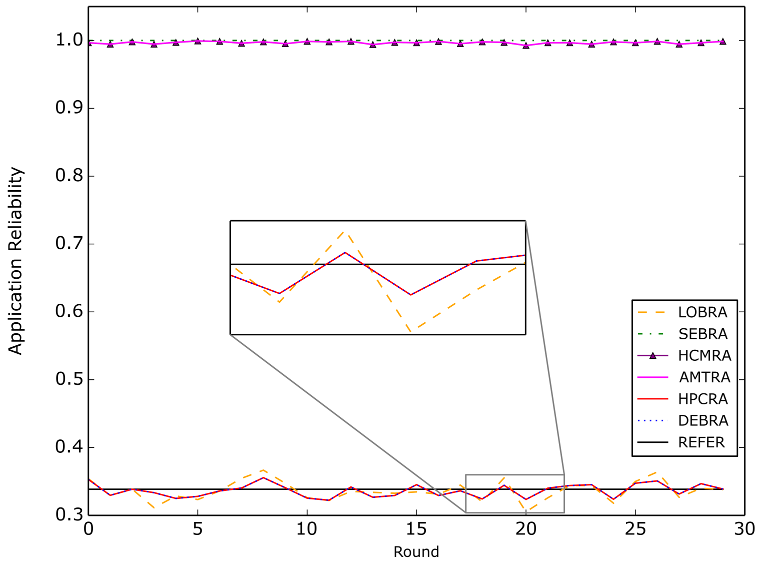


Figure 6.8: Comparison results for the Average-1 deployment.

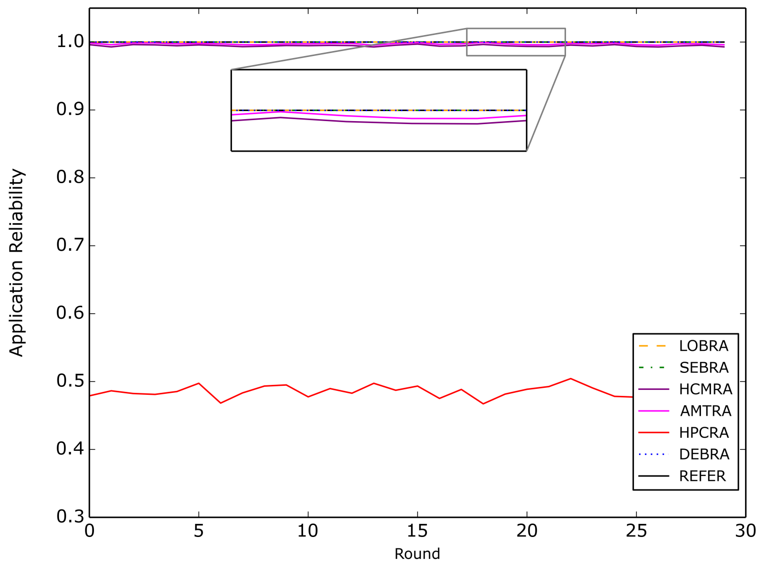


Figure 6.9: Comparison results for the Average-2 deployment.

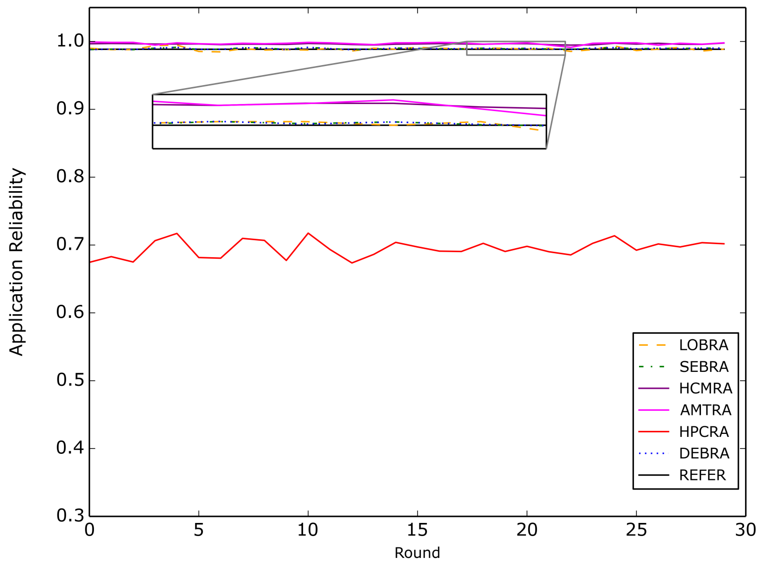


Figure 6.10: Comparison results for the Medium deployment.

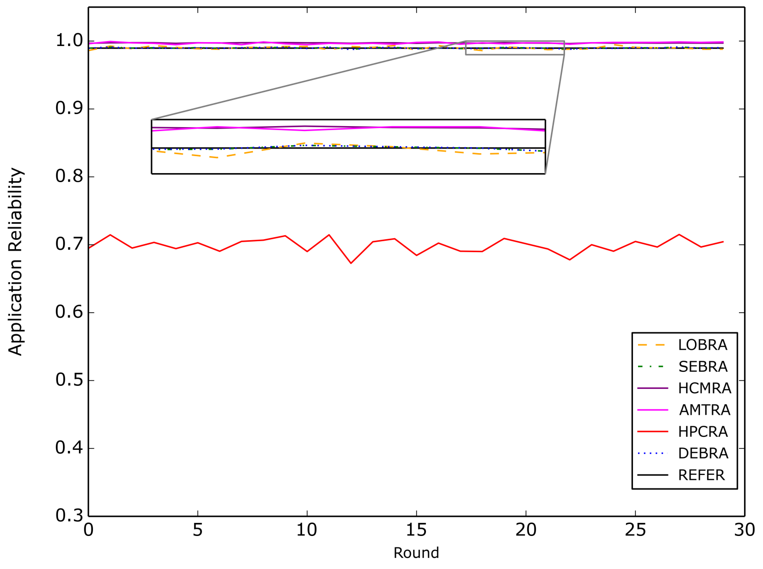


Figure 6.11: Comparison results for the Complex deployment.

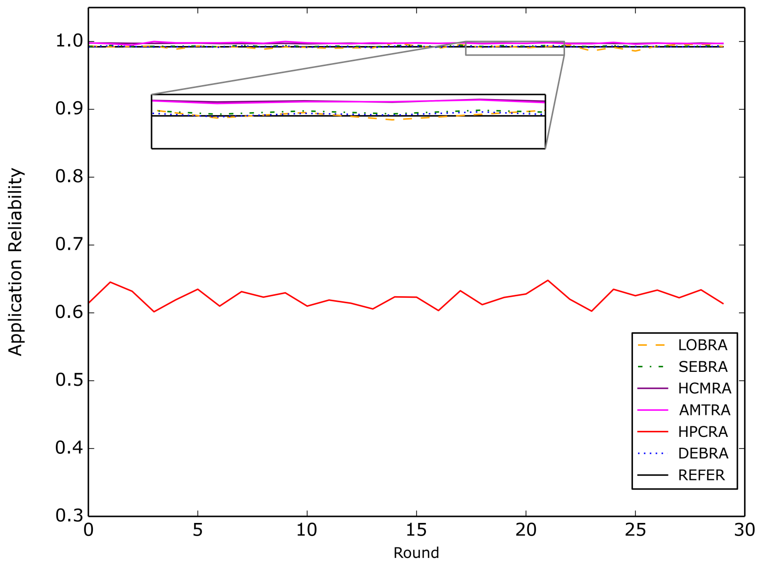


Figure 6.12: Comparison results for the K-out-of-n deployment.

6.3.3 Comparison

In this section, based on the baselines and the assessment results of each method for each deployment, we compare the results of DEBRA with the aforementioned methods in Section 6.1.3.

For all methods corresponding to each deployment, we employ the two-tailed Wilcoxon signed-rank test [100] to test if the results of other methods are significantly different from the results of DEBRA. The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test method that applies to situations where the data cannot be assumed to follow the normal distribution. For n pairs of values, the Wilcoxon signed-rank test arranges the absolute differences of the two values in all pairs in ascending order and assigns ranks to these differences. At the beginning, the ranks ranges from 1 to n. When a number of differences equal to each other, their ranks form a group of ties [101]. Then, the ranks in the same tie group will be updated. Suppose a tie group corresponds to m differences with ranks $rk_1, rk_2, ..., rk_m$, all the differences will be assigned with the same rank $\frac{\sum_{i=1}^{m} rk_i}{m}$. For example, if a tie group corresponds to three ranks, e.g., 1, 2, and 3, the corresponding three differences will be assigned with the same rank of $(1+2+3)/3 = 2$. To these ranks, the Wilcoxon signed-rank test gives them differences signs, i.e., negative signs to negative differences and positive signs to positive differences. The Wilcoxon signed-rank test assumes that "the sum of all positive ranks and the sum of all negative ranks should not differ greatly" [101, p. 43]. In our case, for each method, this assumption corresponds to the null hypothesis H_0: "results are not significantly different from results of DEBRA", i.e.:

$$\boldsymbol{H_0}\textbf{:}\ \mu_d = 0$$

where μ_d is the mean difference between results of the method under test and DEBRA, e.g., for LOBRA:

$$\mu_d = \frac{1}{n}\sum_{i=1}^{n}(R_{LOB_i} - R_{DEB_i}) \tag{6.3.9}$$

where n is the number of rounds, i.e., 30, and R_{LOB_i} and R_{DEB_i} are the evaluated reliability with LOBRA and DEBRA for the ith round, respectively.

According to the method described by Montgomery and Runger [100, p. 583], we calculate the test statistic w-value of each test by:

$$w = min(w^+, w^-) \tag{6.3.10}$$

where w^+ and w^- are the sum of the positive ranks and the absolute values of negative ranks, respectively. And when the sample size is larger than 20, w^+ (or w^-) can be assumed to approximately follow a normal distribution whose mean μ_w and SD δ_w can be calculated, according to [100, p. 583] and [102, p. 42], by:

$$\mu_w = \frac{n(n+1)}{4} \tag{6.3.11}$$

$$\delta_w = \sqrt{\frac{n(n+1)(2n+1) - \sum_{i=1}^{m} t_i(t_i-1)(t_i+1)/2}{24}} \tag{6.3.12}$$

where m is the number of group of ties and t_i is the number of ranks in the ith tie group. Then, the test of the null hypothesis H_0 can be based on the statistic:

$$z = \frac{w - \mu_w}{\delta_w} \tag{6.3.13}$$

where z is called z-score. Calculated w-values and z-scores are shown in Table 6.3.

Table 6.3: w-values and z-scores of Wilcoxon tests.

Deployment		LOBRA	SEBRA	HCMRA	ARMRA	HPCRA
Minimal	w-value	173	0	0	0	–[†]
	z-score	-1.2238	-4.7821	-4.7821	-4.7821	
Simple	w-value	176	207	0	0	–
	z-score	-1.1621	-0.5245	-4.7821	-4.7821	
Average-1	w-value	227	0	0	0	–
	z-score	-0.1131	-4.7821	-4.7821	-4.7821	
Average-2	w-value	0	0	0	0	0
	z-score	-4.7825	-4.7825	0.6174	-4.7821	-4.7821
Medium	w-value	217	28	0	0	0
	z-score	-0.3188	-4.2062	-4.7821	-4.7821	-4.7821
Complex	w-value	219	204	0	0	0
	z-score	-0.2777	-0.5862	-4.7821	-4.7821	-4.7821
K-out-of-n	w-value	194	0	0	0	0
	z-score	-0.7919	-4.7821	-4.7821	-4.7821	-4.7821

[†] DEBRA gets the same results with HPCRA for Minimal, Simple, and Average-1, so, there are no w-values or z-scores for HPCRA.

Based on z-scores, the p-value and effect size (correlation coefficient e) of each test are calculated by:

$$p = 2 * \Phi(-|z|) \tag{6.3.14}$$

$$e = \frac{z}{\sqrt{2n}} \tag{6.3.15}$$

where Φ is the CDF of the standard normal distribution. Results are shown in Table 6.4.

Table 6.4: p-values and effect sizes.

Deployment		LOBRA	SEBRA	HCMRA	ARMRA	HPCRA
Minimal	p-value	**0.2210**	< 0.0001	< 0.0001	< 0.0001	–
	effect size	**0.1580**	0.6174	0.6174	0.6174	
Simple	p-value	**0.2452**	**0.6071**	< 0.0001	< 0.0001	–
	effect size	**0.1500**	**0.0677**	0.6174	0.6174	
Average-1	p-value	**0.9099**	< 0.0001	< 0.0001	< 0.0001	–
	effect size	**0.0146**	0.6174	0.6174	0.6174	
Average-2	p-value	< 0.0001	< 0.0001	< 0.0001	< 0.0001	< 0.0001
	effect size	0.6174	0.6174	0.6174	0.6174	0.6174
Medium	p-value	**0.7499**	< 0.0001	< 0.0001	< 0.0001	< 0.0001
	effect size	**0.0412**	0.5430	0.6174	0.6174	0.6174
Complex	p-value	**0.7813**	**0.5577**	< 0.0001	< 0.0001	< 0.0001
	effect size	**0.0358**	**0.0757**	0.6174	0.6174	0.6174
K-out-of-n	p-value	**0.4284**	< 0.0001	< 0.0001	< 0.0001	< 0.0001
	effect size	**0.1022**	0.6174	0.6174	0.6174	0.6174

According to Cohen's standard [103], 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 see from Table 6.4 that:

1. Results of LOBRA for all deployments except Average-2 are not significantly different from DEBRA, with $p = 0.2210$, 0.2452, 0.9099, 0.5716, 0.7813, and 0.4284 for the Minimal, Simple, Average-1, Medium, Complex, and K-out-of-n deployments, respectively. Besides, Section 6.3.1 shows that, the baseline for Average-2 is 0.999999, which suggests an extremely reliable deployment and it's too short time for rounds with 1,500 time units to see an application failure. And results in Section 6.3.2 show that LOBRA gets the reliability of 1.000000 for all rounds while DEBRA does not. In addition, the assessment results and $\overline{R_{eva}}$ (0.99999751) of DEBRA for Average-2 is also very close to 1.0, we believe that LOBRA and DEBRA will not be significantly different with a larger data size. As a result, we do not reject H_0 for LOBRA.

2. Results of SEBRA for the Minimal, Average-1, Medium, and K-out-of-n deployments are significantly different from DEBRA, with $p < 0.0001$, and large effect sizes of 0.6174, 0.6174, 0.5430, and 0.6174, respectively, so, we can reject H_0 for above deployments. Besides, results of SEBRA for the Simple and Complex deployments are not significantly different from DEBRA, with $p = 0.6071$ and 0.5577, respectively, we cannot reject H_0 for Simple or Complex. And due to the same reason as LOBRA, we do not reject H_0 for Average-2.

3. For HCMRA and ARMRA, we can reject H_0 for all deployments because results of them are significantly different from DEBRA, with $p < 0.0001$ and a large effect size of 0.6174.

4. As to HPCRA, on one hand, we cannot reject H_0 for the Minimal, Simple, and Average-1 deployments, because DEBRA can be simplified to HPCRA for these deployments and gets the same results as HPCRA. On the other hand, we can reject H_0 for the Average-2, Medium, Complex, and K-out-of-n deployments because HPCRA gets significantly different results from DEBRA, with $p < 0.0001$ and a large effect size of 0.6174.

We have tested whether a method is significantly different from DEBRA, however, different or not can not help us determine which method is more accurate or precise. To further compare the accuracy and precision of different methods for different deployments, we calculate the MAEs and SDs of all methods with (6.1.1) and (6.1.2), which are shown in Table 6.5 with all non-significantly different results in bold.

Table 6.5: MAE and SD of results.

Deployment	R_{ref}	metrics	DEBRA	LOBRA	SEBRA	HCMRA	ARMRA	HPCRA
Minimal	0.886037	MAE	**0.001305**	**0.001637**	0.056494	0.081982	0.112763	0.001305
		SD	**0.007544**	**0.008230**	0.013788	0.005170	0.000709	0.007544
Simple	0.832768	MAE	**0.000996**	**0.000454**	**0.000880**	0.076898	0.164457	0.000996
		SD	**0.009618**	**0.010579**	**0.010133**	0.008268	0.001593	0.009618
Average-1	0.338578	MAE	**0.001968**	**0.002511**	0.661422	0.658181	0.658270	0.001968
		SD	**0.009665**	**0.014663**	< 0.000001	0.001696	0.001693	0.009665
Average-2	0.999999	MAE	0.000001	0.000001	0.000001	0.005369	0.002708	0.513228
		SD	< 0.000001	< 0.000001	< 0.000001	0.001204	0.001145	0.010085
Medium	0.988659	MAE	**0.000047**	**0.000297**	0.000808	0.007567	0.008367	0.293854
		SD	**0.001114**	**0.002443**	0.001007	0.000746	0.001663	0.012575
Complex	0.989542	MAE	**0.000256**	**0.000414**	**0.000244**	0.007654	0.007439	0.290575
		SD	**0.000858**	**0.002329**	**0.000844**	0.000361	0.001328	0.010527
K-out-of-n	0.992138	MAE	**0.000273**	**0.000116**	0.001108	0.005190	0.005397	0.369819
		SD	**0.000755**	**0.002641**	0.002641	0.000371	0.001051	0.012058

Combining Table 6.4 and 6.5, for 30 rounds of 1,500 time units of simulation, we can conclude that:

1. Results of LOBRA for Average-2 are not significantly different DEBRA while MAEs of both DEBRA and LOBRA are 0.000001, based on which we cannot tell which method is more accurate. Besides, the results of LOBRA for deployments except for Average-2 are not significantly different from DEBRA, while the SDs of LOBRA are larger than that of DEBRA, especially for Medium, Complex, and K-out-of-n, which manifests that DEBRA is more precise than LOBRA for these deployments.

2. Results of SEBRA for Minimal, Average-1, Medium, and K-out-of-n are significantly different from DEBRA, while MAEs are larger than that of DEBRA. So, DEBRA is more accurate than SEBRA for Minimal, Average-1, Medium, and K-out-of-n. Besides, results of SEBRA for Simple, Average-2, and Complex are not significantly different from DEBRA, however, as to SDs, the differences are too small (around 5%, the same and around 2% for Simple, Average-2, and Complex, respectively) to make conclusions.

3. For HCMRA and ARMRA, results are significantly different from DEBRA for all deployments and MAEs are always larger than DEBRA, which indicates that DEBRA are more accurate than HCMRA and ARMRA for all deployments.

4. As to HPCRA, results are the same as results of DEBRA for the Minimal, Simple, and Average-1 deployments, so, HPCRA is the same accurate and precise as DEBRA for these deployments. Besides, results of HPCRA for Average-2, Medium, Complex, and K-out-of-n are significantly different from DEBRA with much larger MAEs, which means that DEBRA is more accurate than HPCRA for these deployments.

6.4 Discussion

In this section, we discuss the results of the case study shown in Section 6.3.2 and determine if we accept or reject the hypotheses presented in Section 6.1.1.

The determination of hypotheses proposed in Section 6.1.1 is as following:

- *H1: The framework can assess the reliability of services and the application.* Based on fault injection, we simulated the application state for 30 25-hour rounds with each minute as a time unit. The report files created by our framework show the (inner) reliability of components including services and the application reliability, i.e., the reliability of the Apache load balancer service, assessed based on both the setup parameters and monitoring logs. Therefore, we accept this hypothesis.

- *H2: DEBRA can assess the application reliability when deployment structures include deep dependencies or not.* We designed different deployment structures for the chosen application. Some deployment structures include deep dependencies between SIs of the same service, e.g., the Average-1, Average-2, Medium, and K-out-of-n deployments and some deployment structures include deep dependencies between SIs of different services, e.g., the Medium, Complex, K-out-of-n deployments. For deployments with both types of deep dependencies, DEBRA can assess the reliability of the application which are shown in Table 6.2 and Figure 6.6 to Figure 6.12. Therefore, we accept this hypothesis.
- *H3: DEBRA gets more accurate results than existing methods.* According to Section 6.3.2, with a 0.05 significance level, DEBRA results are: not significantly different from results of LOBRA for all deployments; significantly different from and more accurate than SEBRA results for Minimal, Average-1, Medium and K-out-of-n, and not significantly different for Simple, Average-2, and Complex; significantly different from and more accurate than results of HCMRA and ARMRA for all deployments; significantly different from and more accurate than HPCRA results for Average-2, Medium, Complex and K-out-of-n, and not significantly different for Minimal, Simple, and Average-1. Therefore, we reject this hypothesis because it holds only under above conditions for different methods.
- *H4: DEBRA gets more precise results than existing methods with a small size of data.* According to Section 6.3.2, DEBRA results are: more precise for Minimal, Simple, Medium, Complex, and K-out-of-n, and not less precise for Average-2 than the results of LOBRA; more accurate for Minimal, Average-1, Medium, and K-out-of-n, not less precise for Average-2, Simple and Complex than the results of SEBRA; the same for Minimal, Simple and Average-1 as HPCRA and more precise for Average-2, Medium, Complex, and K-out-of-n than the results of HPCRA. Therefore, we reject this hypothesis because it holds only under above conditions for different methods.

According to the determination of H3 and H4, we can conclude that, DEBRA is the same accurate as LOBRA and more precise than it for 30 rounds of 1,500 time units of simulation, and more accurate and precise than other methods in most cases.

Chapter 7

Discussion

In this chapter, we discuss the answers to the research questions posed in Section 1.2 and strengths and limitations of DEBRA, the framework and its implementation based on the case studies.

7.1 Answers to Research Questions

RQ1: What aspects should be considered when modeling cloud applications for reliability assessment? We addressed this question in case studies by comparing the results of different methods that consider different sets of components in the deployment stack of cloud applications. The results show that, by considering all components in the deployment stack and dependencies between them, DEBRA obtained non-significantly different results from a method, which considers only the failures of the application that emerge in the access log files, for all the tested deployment structures with a 0.05 significance level, while the results of DEBRA and the above method are more accurate than the results of other methods. Therefore, the answer to this question is that, considering an application as a whole and assessing its reliability according to application failures without regard to underlying structure details is sufficient for the reliability assessment of cloud applications.

RQ2: How do dependencies affect the accuracy of a reliability assessment? To address this question, we explored the influences of dependencies introduced by the layered structure of cloud applications and different kinds of redundancies to the accuracies of reliability assessment methods in our case studies. The answer to this question is twofold. On one hand, the answer to the first research question partially applies to this question. If the only goal of a reliability assessment method is to obtain the application reliability, dependencies have no significant influences to the accuracy because methods considering no dependencies can also obtain the same accurate results. On the other hand, if a reliability assessment method is used as the basis to improve the application reliability, which requires the ability of predicting the reliability when the application deployment structure varies, involving dependencies in the model will improve the accuracy. Methods that do not consider the underlying structures of the deployment stack will cover up the failures of underlying components, which is not appropriate for improving the application reliability.

RQ3: How can the quality of reliability assessment methods be compared? To tackle this question, we proposed a process to compare the accuracy and the precision of different methods in our case studies. Before the comparison, we generated baselines of certain accuracies. In practice, accurate baselines are difficult to obtain because too much data will be required. For example, to get a baseline with at least five significant figures after the decimal point for the Complex deployment in Section 6.3.1, the MCS performs more than 2.7 billion rounds of simulation. If one simulation round corresponds to one day as a time unit, the practical data would need much more time than acceptable. But for testing, when failure rates of components are assumed, the MCS performs well for getting accurate baselines, e.g., 81 minutes for the above 2.7 billion rounds. When baselines are available, based on whether results of two different methods are significantly different or not, we utilized MAE to compare the accuracy and the SD to compare the precision. And the difference between methods are tested by the Wilcoxon signed-rank test. Judging from the comparison results, one possible answer to this question is: using the MCS to obtain baselines, using the Wilcoxon signed-rank test to determine if results of different methods are significantly different and using the MAEs and the SDs as metrics to compare the quality of different methods.

7.2 Strengths and Limitations

DEBRA has several strengths comparing with existing methods while our framework and the implementation have inevitable limitations.

7.2.1 Strengths of DEBRA

Based on the introduction in Chapter 3 and Section 6.1.3 and the comparison in Section 6.3, we compare DEBRA with the chosen existing methods regarding merits during the reliability assessment to show the strengths of DEBRA. We compared the following features:

- *Normal redundancy*: whether a method considers 1-out-of-n redundancies when assessing the service/application reliability. DEBRA models cloud applications with LDGs which generalize normal redundancies into k-out-of-n ($k = 1$) redundancies. LOBRA uses the access logs to assess the application reliability, regardless of any structure of the application including any kind of redundancies. SEBRA considers services providing the same functionalities as redundancies to each other with priorities. HCMRA considers the scenario that redundant service instances are deployed on a number of PSs. ARMRA calculates the system (application) reliability with the production of hardware and network links regardless of any kind of redundancies. HPCRA is for high performance applications (specifically, MPI applications) deployed with several processes which are organized as a n-out-of-n redundant system, so, HPCRA does not consider normal redundancies.

- *K-out-of-n redundancy*: whether a method considers k-out-of-n ($1 \leq k \leq n$) redundancies when assessing the service/application reliability. DEBRA uses LDGs to generalize all redundancies including k-out-of-n ($1 < k \leq n$) into k-out-of-n ($1 \leq k \leq n$) redundancies. LOBRA, as aforementioned, does not consider k-out-of-n redundancy. SEBRA considers that all redundant services are organized as a normal 1-out-of-n redundancy system. HCMRA considers the service reliability as the probability that at least one VM used by the service is available, which limits the method to $k = 1$. ARMRA, as discussed above, does not consider any kind of redundancies. HPCRA, as discussed above, considers only n-out-of-n redundancies rather than k-out-of-n redundancies with $\forall k \in (1, n]$.
- *Deep dependency*: whether a method considers deep dependencies when assessing the service/application reliability. DEBRA considers deep dependencies between SIs of the same service and of different services. LOBRA, as aforementioned, does not consider deep dependency. SEBRA is designed for SaaS applications without considering VMs, PSs or deep dependencies. HCMRA explores one case of deep dependency that VMs used by a service are deployed on the same PS, and as HCMRA is proposed for one single service, it does not consider the deep dependencies between different services. ARMRA considers only the hardware reliability (such as reliability of hard disks, processors, etc.) and assumes that PSs are independent, as a result, no dependency is considered. HPCRA considers dependencies between software or between hardware, and assumes no dependencies between software and hardware, which indicates that no deep dependencies are considered.
- *Service reliability*: whether a method is able to assess service (as a part of an application) reliability. DEBRA assesses the service reliability based on the sLDG of the service. LOBRA only assesses the application reliability. SEBRA assumes that service reliability is published by providers and can assess the service reliability perceived by consumers. HCMRA is designed for large online services in IaaS clouds. ARMRA assumes that services are fully reliable and assesses service reliability with only hardware reliability. HPCRA assesses reliability of MPI applications with several processes (organized as a n-out-of-n redundant system) on several VMs, so, an application in this context equals a service in our model and HPCRA is able to assess service reliability when service instances are organized as a n-out-of-n system.
- *Application reliability*: whether a method is able to assess the application (as a composite service or a set of different services) reliability. DEBRA considers the application reliability as the reliability of the service that has no predecessors and assesses the application reliability in the same way as assessing the service reliability. the LOBRA is able to assess the application reliability via access logs. SEBRA can be used to assess SaaS application reliability. HCMRA is for one single service which

can be an application, but if we consider an application as a set of services, then HCMRA is not for applications. ARMRA treats a CCS as a set of services each of which is composed of several tasks, so, the assessed CCS reliability can be seen as the application reliability. HPCRA, as discussed above, can assess the reliability of services with n-out-of-n instances rather than applications.

Regarding the above features, the comparison results are shown in Table 7.1 ("✓" means "with the merit" and blank means "without the merit"). Generally speaking, the most significant strength of DEBRA is the applicability for diverse deployment structures that includes different types of redundancies, especially the k-out-of-n redundancy, and deep dependencies. To the best of our knowledge, DEBRA is the only one that considers the k-out-of-n ($k > 1$) redundancy for the reliability assessment of cloud applications. Besides, DEBRA models deep dependencies between SIs of the same service and different services, which are not considered adequately in existing works.

Table 7.1: Comparison of DEBRA with existing methods.

Feature	DEBRA	LOBRA	SEBRA	HCMRA	ARMRA	HPCRA
Normal redundancy	✓		✓	✓		
K-out-of-n redundancy	✓					
Deep dependency	✓			✓		
Service reliability	✓		✓	✓	✓	✓
Application reliability	✓	✓	✓		✓	

7.2.2 Limitations of the Framework and Implementation

Our framework and its implementation have some limitations. At first, our framework works ideally when the application structure and reliability information of components included in the deployment stack of cloud applications are available. This means that our framework works best for private IaaS cloud customers, since reliability information of PSs, cloud management software, etc., are usually not accessible for public cloud customers. Nonetheless, our framework can also work with only the reliability of components in upper layers, such as only service instances. In this case, the reliability assessment method in our framework degenerates to other methods, e.g., the method proposed by Wang et al. [20] if the reliability of PSs is not available. Secondly, the implementation of our framework is only for web-based cloud applications, more specifically, websites deployed with Cloudify on top of OpenStack clouds. Testing methods of other types of applications, such as storage applications without need of browsers, are not implemented. However, we believe that the support of other cloud platforms and other types of applications can be easily added.

In the proposed reliability assessment method, we assume that failures of PSs, VMs, and SIs are independent in consideration of the trivial influence of the correlation of failures to the application reliability. Besides, some deployment scenarios of cloud applications, such as with containers, are not considered in the proposed framework and method.

7.2.3 Threats to Validity

We compare different reliability assessment methods with testing data obtained by manually setting reliability information of components rather than real-world monitoring data. This is because the failure rate of PSs and VMs are much lower than usual applications and the same size of data as one round of the testing data we used in our case study will take about four years (1,500 time units) to obtain.

In our case studies, we do not consider the cloud applications with choice function dependencies which are modeled in our framework, we used some educated guesses for the failure rates of VMs and SIs, and we compared chosen methods with DEBRA based on the monitoring field data from our framework, which is problematic if the monitor did not get the states of components correctly.

Chapter 8

Conclusion

In the last chapter, we summarize the thesis and its contributions and then state possible directions of extending DEBRA and the framework proposed in this thesis.

8.1 Summary

In this thesis, we aim to model cloud applications and assess their reliability. To achieve this purpose, we proposed DEBRA and accordingly design a framework including a dependency analyzer for analyzing dependencies between components, a monitor for obtaining failure data of components and a reliability analyzer based on DEBRA. In the framework, components of cloud applications including PSs, VMs, SIs, and services are modeled with LDGs considering different kinds of redundancies and dependencies. Compared to existing methods, we considered *k*-out-of-*n* redundancies which have not been well considered to the best of our knowledge, and we also considered deep dependencies which have only partially considered in existing works.

We implemented not only the above three components of our framework but also two components for testing web-based cloud applications deployed with Cloudify on OpenStack IaaS clouds. The two additional components are a fault injector and a tester. We designed the fault injector to inject failures to components of cloud applications. And we implemented the tester to test web-based cloud applications with usage data.

To verify our framework and compare DEBRA with existing methods, we proposed a process for comparing accuracies and precisions of different methods and conducted case studies with different kinds of deployments of cloud applications. In the case studies, we explored the influences of redundancies and deep dependencies to the application reliability and compared the accuracies and precisions of DEBRAs with existing methods. The results show that, regarding the accuracy, DEBRA performs better than other model-based methods. And DEBRA can get more precise results than existing methods when only a small size of data are available. The results also show that, if no requirement of improving the application reliability, the underlying deployment structures including, e.g., different kinds of redundancy configurations of SIs and the deep dependencies, are not necessary to be considered for obtaining the accurate reliability of cloud applications.

8.2 Outlook

Our work in this thesis can be extended in the following directions.

The LDG model of cloud applications can be extended by involving more components and more complex structures. One possible component is the hypervisor that can be separated from the integrated component PS in our model and can be further divided into different types. In this case, more than one type of hypervisors can be installed and modeled on a single PS. Another possible kind of components are network devices. In our model, they are also integrated into PSs. If separated, network devices and PSs can then be modeled by an extra correlation model. However, this may lead to a much more complex model. An example structure that can be added to the LDG is the triangle of function dependencies which can be used to model more complex applications. Furthermore, the LDG can also be improved by supporting dynamic changes of the deployment of applications. Elasticity is one of the significant characteristics of cloud computing which enable the scaling of cloud applications. When instances of a service or an application are scaled, the LDG will also be changed. In the current implementation of our framework, the LDG must be fixed before the application is deployed and unable to be changed during the running process of the framework because, we think, the support of elasticity may also be achieved by restarting our framework.

The dependency analyzer can also be updated. Currently, the dependencies between services are gathered from user-defined Groovy files. On one hand, more file formats can be added to the dependency analyzer, such as XML and YAML. On the other hand, automatically analyzing the blueprints used to describe applications in Cloudify to get deployments of applications will make our framework more user-friendly.

The fault injector currently does not support service-level fault injection. Though this function is not necessary currently and can be achieved by injecting failures to instances of a service, it would be convenient if the framework can automatically kill a service according to the redundancy configuration, e.g., k-out-of-n, of its instances.

The current implementation of our framework supports only websites. Supports of other types of applications, such as applications with Graphical User Interfaces (GUIs) are also interesting for monitoring and usage-based testing.

The correlations of failures of the same type of components, such as PSs, VMs, and SIs, can be further considered in DEBRA.

Bibliography

[1] M. R. Lyu *et al.*, *Handbook of software reliability engineering*. IEEE computer society press CA, 1996, vol. 222.

[2] 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.

[3] 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.

[4] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

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

[6] R. Buyya, R. Ranjan, and R. N. Calheiros, "Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2010, pp. 13–31.

[7] Zheng, Zibin and Zhou, Tom Chao and Lyu, Michael R and King, Irwin, "Component ranking for fault-tolerant cloud applications," *Services Computing, IEEE Transactions on*, vol. 5, no. 4, pp. 540–550, 2012.

[8] 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.

[9] 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.

[10] 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.

[11] 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," *2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)*, pp. 315–321, jun 2013.

[12] 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.

[13] Z. Zheng and M. R. Lyu, "Collaborative reliability prediction of service-oriented systems," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, vol. 1. New York, New York, USA: ACM Press, 2010, p. 35.

[14] Q. Zhang, M. F. Zhani, M. Jabri, and R. Boutaba, "Venice: Reliable virtual data center embedding in clouds," in *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 2014, pp. 289–297.

[15] B. Speitkamp and M. Bichler, "A mathematical programming approach for server consolidation problems in virtualized data centers," *IEEE Transactions on services computing*, vol. 3, no. 4, pp. 266–278, 2010.

[16] A. Birolini, *Reliability engineering*. Springer, 2007, vol. 5.

[17] B. Ford, "Icebergs in the clouds: the other risks of cloud computing," in *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '12)*, mar 2012, p. 6.

[18] Y. Sharma, B. Javadi, W. Si, and D. Sun, "Reliability and energy efficiency in cloud computing systems: Survey and taxonomy," *Journal of Network and Computer Applications*, vol. 74, pp. 66–85, 2016.

[19] R. Jhawar, V. Piuri, and M. Santambrogio, "Fault tolerance management in cloud computing: A system-level perspective," *IEEE Systems Journal*, vol. 7, no. 2, pp. 288–297, 2013.

[20] 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.

[21] E. Zio, "Reliability engineering: Old problems and new challenges," *Reliability Engineering & System Safety*, vol. 94, no. 2, pp. 125–141, 2009.

[22] N. Fuqua, *Reliability engineering for electronic design*. CRC Press, 1987, vol. 34.

[23] D. Kececioglu, *Reliability engineering handbook (vol. 1)*. Prentice-Hall, Inc., 1991.

[24] P. O'Connor and A. Kleyner, *Practical reliability engineering*. John Wiley & Sons, 2011.

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

[26] K. S. Trivedi, *Probability & statistics with reliability, queuing and computer science applications*. John Wiley & Sons, 2008.

[27] J. I. McCool, *Using the Weibull distribution: reliability, modeling and inference*. John Wiley & Sons, 2012, vol. 950.

[28] G.-A. Klutke, P. C. Kiessler, and M. Wortman, "A critical look at the bathtub curve," *IEEE Transactions on Reliability*, vol. 52, no. 1, pp. 125–129, 2003.

[29] OpenStack. Retrieved 12/2016. [Online]. Available: https://www.openstack.org/

[30] HP Helion. Retrieved 12/2016. [Online]. Available: http://www8.hp.com/us/en/cloud/helion-overview.html

[31] OpenShift. Retrieved 12/2016. [Online]. Available: https://www.openshift.com/

[32] Cloudify. Retrieved 12/2016. [Online]. Available: http://www.getcloudify.org/

[33] State of the cloud report. Retrieved 03/2016. [Online]. Available: http://assets.rightscale.com/uploads/pdfs/RightScale-2016-State-of-the-Cloud-Report.pdf

[34] B. Larkin and M. Rose, "2015 top markets report cloud computing," retrieved 09/2016. [Online]. Available: http://trade.gov/topmarkets/pdf/Cloud_Computing_Top_Markets_Report.pdf

[35] Amazon AWS. Retrieved 12/2016. [Online]. Available: https://aws.amazon.com/

[36] Google Compute Engine. Retrieved 12/2016. [Online]. Available: https://cloud.google.com/compute/

[37] Microsoft Azure. Retrieved 12/2016. [Online]. Available: https://azure.microsoft.com/en-gb/

[38] iCloud. Retrieved 03/2016. [Online]. Available: https://www.icloud.com/

[39] Conversation with eric schmidt hosted by danny sullivan. Retrieved 03/2016. [Online]. Available: http://www.google.com/press/podium/ses2006.html

[40] Amazon EC2 beta announcement. Retrieved 06/2016. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/

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

[42] OpenStack User Survey. Retrieved 09/2016. [Online]. Available: https://www.openstack.org/assets/survey/April-2016-User-Survey-Report.pdf

[43] OpenStack Hypervisors. Retrieved 12/2016. [Online]. Available: http://docs.openstack.org/newton/config-reference/compute/hypervisors.html

[44] TOSCA. Retrieved 09/2016. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf

[45] YAML. Retrieved 12/2016. [Online]. Available: http://www.yaml.org/

[46] Security group. Retrieved 09/2016. [Online]. Available: http://docs.openstack.org/user-guide/cli-nova-configure-access-security-for-instances.html

[47] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proceedings of the 1st ACM symposium on Cloud computing.* ACM, 2010, pp. 193–204.

[48] 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.

[49] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, p. 350, oct 2011.

[50] P. Garraghan, P. Townend, and J. Xu, "An empirical failure-analysis of a large-scale cloud computing environment," in *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on.* IEEE, 2014, pp. 113–120.

[51] B. Wei, C. Lin, and X. Kong, "Dependability modeling and analysis for the virtual data center of cloud computing," in *2011 IEEE International Conference on High Performance Computing and Communications.* IEEE, Sep 2011, pp. 784–789.

[52] Y.-K. Lin and P.-C. Chang, "Evaluation of system reliability for a cloud computing system with imperfect nodes," *Systems Engineering*, vol. 15, no. 1, pp. 83–94, 2012.

[53] B. Snyder, J. Ringenberg, R. Green, V. Devabhaktuni, and M. Alam, "Evaluation and design of highly reliable and highly utilized cloud computing systems," *Journal of Cloud Computing*, vol. 4, no. 1, pp. 1–16, 2015.

[54] Y.-s. Dai, B. Yang, J. Dongarra, and G. Zhang, "Cloud service reliability: Modeling and analysis," in *15th IEEE Pacific Rim International Symposium on Dependable Computing*, 2009, pp. 1–17.

[55] Amazon EC2. Retrieved 12/2016. [Online]. Available: https://aws.amazon.com/ec2

[56] Y.-s. Dai, Y. Pan, and X. Zou, "A hierarchical modeling and analysis for grid service reliability," *IEEE Transactions on Computers*, vol. 56, no. 5, pp. 681–691, may 2007.

[57] H. Cui, Y. Li, J. Chen, and Y. Liu, "Methods with low complexity for evaluating cloud service reliability," in *Wireless Personal Multimedia Communications (WPMC), 2013 16th International Symposium on*. IEEE, 2013, pp. 1–5.

[58] Z. Wu, N. Chu, and P. Su, "Improving cloud service reliability–a system accounting approach," in *Services Computing (SCC), 2012 IEEE Ninth International Conference on*. IEEE, 2012, pp. 90–97.

[59] A. H. Maslow, "A theory of human motivation." *Psychological review*, vol. 50, no. 4, p. 370, 1943.

[60] F. Dudouet, A. Edmonds, and M. Erne, "Reliable cloud-applications: an implementation through service orchestration," in *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*. ACM, 2015, pp. 1–6.

[61] Z. Zheng, T. C. Zhou, M. R. Lyu, and I. King, "FTCloud: A component ranking framework for fault-tolerant cloud applications," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, nov 2010, pp. 398–407.

[62] W. Qiu, Z. Zheng, X. Wang, X. Yang, and M. R. Lyu, "Reliability-based design optimization for cloud migration," *Services Computing, IEEE Transactions on*, vol. 7, no. 2, pp. 223–236, 2014.

[63] A. Zhou, S. Wang, Z. Zheng, C.-H. Hsu, M. Lyu, and F. Yang, "On cloud service reliability enhancement with optimal resource usage," *IEEE Transactions on Cloud Computing*, vol. 7161, 2014.

[64] W. Zhao, P. Melliar-Smith, and L. E. Moser, "Fault tolerance middleware for cloud computing," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE, 2010, pp. 67–74.

[65] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[66] Y. Zhang, Z. Zheng, and M. R. Lyu, "BFTCloud: A byzantine fault tolerance framework for voluntary-resource cloud computing," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 444–451.

[67] A. Zhou, S. Wang, B. Cheng, Z. Zheng, F. Yang, R. Chang, M. Lyu, and R. Buyya, "Cloud service reliability enhancement via virtual machine placement optimization," *IEEE Transactions on Service Computing*, 2016.

[68] S. Malik and F. Huet, "Adaptive fault tolerance in real time cloud computing," in *Services (SERVICES), 2011 IEEE World Congress on*. IEEE, 2011, pp. 280–287.

[69] S. Malik, F. Huet, and D. Caromel, "Reliability aware scheduling in cloud computing," in *Internet Technology And Secured Transactions, 2012 International Conference for*. IEEE, 2012, pp. 194–200.

[70] N. Padmapriya and R. Rajmohan, "Reliability evaluation suite for cloud services," in *Computing Communication & Networking Technologies (ICCCNT), 2012 Third International Conference on*. IEEE, 2012, pp. 1–6.

[71] S. Banerjee, H. Srikanth, and B. Cukic, "Log-based reliability analysis of Software as a Service (SaaS)," *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pp. 239–248, nov 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5635046

[72] Banerjee, Sean and Srikanth, Hema and Cukic, Bojan, "Challenges for creating highly dependable service based systems," in *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. IEEE, Mar 2011, pp. 264–273.

[73] J. Tian, S. Rudraraju, and Z. Li, "Evaluating web software reliability based on workload and failure data extracted from server logs," *Software Engineering, IEEE Transactions on*, vol. 30, no. 11, pp. 754–769, 2004.

[74] E. Nelson, "Estimating software reliability from test data," *Microelectronics Reliability*, vol. 17, no. 1, pp. 67–73, 1978.

[75] S. Au and J. L. Beck, "A new adaptive importance sampling scheme for reliability calculations," *Structural safety*, vol. 21, no. 2, pp. 135–158, 1999.

[76] X. Wang and J. Grabowski, "A reliability assessment framework for cloud applications," *CLOUD COMPUTING 2015*, p. 142, 2015.

[77] R. Ghosh, F. Longo, F. Frattini, S. Russo, and K. S. Trivedi, "Scalable analytics for iaas cloud availability," *Cloud Computing, IEEE Transactions on*, vol. 2, no. 1, pp. 57–70, 2014.

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

[79] 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.

[80] 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.

[81] T.-T. Pham and X. Defago, "Reliability prediction for component-based software systems with architectural-level fault tolerance mechanisms," in *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*. IEEE, 2013, pp. 11–20.

[82] Recipe. Retrieved 09/2016. [Online]. Available: http://getcloudify.org/guide/2.7/developing/application_recipe.html

[83] Groovy. Retrieved 09/2016. [Online]. Available: http://groovy-lang.org/

[84] Apache jclouds. Retrieved 12/2016. [Online]. Available: https://jclouds.apache.org/

[85] Ganglia API. Retrieved 12/2016. [Online]. Available: https://github.com/guardian/ganglia-api

[86] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[87] Selenium. Retrieved 12/2016. [Online]. Available: http://www.seleniumhq.org/

[88] Firefox. Retrieved 12/2016. [Online]. Available: https://www.mozilla.org/en-US/firefox/new/

[89] XHTML. Retrieved 09/2016. [Online]. Available: https://www.w3.org/TR/xhtml1/

[90] S. Herbold and P. Harms, "Autoquest–automated quality engineering of event-driven software," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 134–139.

[91] XPath. Retrieved 09/2016. [Online]. Available: https://www.w3.org/TR/xpath/

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

[93] W. E. Smith, K. S. Trivedi, L. A. Tomek, and J. Ackaret, "Availability analysis of blade server systems," *IBM Systems Journal*, vol. 47, no. 4, pp. 621–640, 2008.

[94] Software Engineering for Distributed Systems Group. Retrieved 12/2016. [Online]. Available: https://www.swe.informatik.uni-goettingen.de/

[95] Apache HTTP Server. Retrieved 12/2016. [Online]. Available: https://httpd.apache.org/

[96] Drupal. Retrieved 12/2016. [Online]. Available: https://www.drupal.org/

[97] MySQL. Retrieved 12/2016. [Online]. Available: https://www.mysql.com/

[98] A. B. Owen, *Monte Carlo theory, methods and examples*, 2013.

[99] N. J. Higham, *Accuracy and stability of numerical algorithms*. Siam, 2002.

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

[101] P. Sprent and N. C. Smeeton, *Applied nonparametric statistical methods, 3rd edition*. CRC Press, 2001.

[102] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods*. John Wiley & Sons, 2013.

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

List of Figures

List of Tables

Acronyms

AFR	Annualized Failure Rate
API	Application Program Interface
ARMRA	Analytical Reliability Model for Reliability Assessment
AWS	Amazon Web Services
BFT	Byzantine Fault Tolerance
CCF	Common Cause Failure
CCS	Cloud Computing System
CDF	Cumulative Distribution Function
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DDA	Deployment Dependency Analyzer
DEBRA	DEpendency-Based Reliability Assessment
DSL	Domain Specific Language
EC2	Elastic Compute Cloud
FDA	Function Dependency Analyzer
FTM	Fault Tolerance Manager
GUI	Graphical User Interface
HCMRA	Hierarchical Correlation Model for Reliability Assessment
HDD	Hard Disk Drive
HPCRA	High Performance Computing Reliability Assessment
HTML	HyperText Markup Language

HTTP	HyperText Transfer Protocol
IaaS	Infrastructure as a Service
ID	IDentifier
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IT	Information Technology
KVM	Kernel-based Virtual Machine
LDG	Layered Dependency Graph
LLFT	Low Latency Fault Tolerance
LOBRA	LOg-Based Reliability Assessment
MAE	Mean Absolute Error
MCS	Monte Carlo Simulation
MPI	Message Passing Interface
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failures
NIST	National Institute of Standards and Technology
NVP	N-Version Programming
OPVMP	OPtimal redundant Virtual Machine Placement
OS	Operating System
PaaS	Platform as a Service
PDF	Probability Density Function
PS	Physical Server
QoR	Quality of Reliability
QoS	Quality of Service

RBD	Reliability Block Diagram
REST	REpresentational State Transfer
S3	Simple Storage Service
SaaS	Software as a Service
SD	Standard Deviation
SEBRA	SErvice-Based Reliability Assessment
SI	Service Instance
sLDG	sub-LDG
SPoF	Single Point of Failure
SSH	Secure SHell
SV	SI-VM
TOSCA	Topology and Orchestration Specification for Cloud Applications
URL	Uniform Resource Locator
VDC	Virtual Data Centers
VM	Virtual Machine
VMM	Virtual Machine Monitor
WoL	Wake-on-LAN
XHTML	eXtensible HyperText Markup Language
XML	eXtensible Markup Language
XPath	XML Path Language

Chapter A

Appendices

A.1 Correctness of the Adapted Algorithm for Calculating the Standard Variance of Monte Carlo Simulation Results

Given:

$$\delta_i = R_{sim_i} - \bar{\mu}_{i-1}$$

$$\bar{\mu}_i = \sum_{j=1}^{i} R_{sim_j}/i$$

$$S_i = S_{i-1} + \frac{i-1}{i}\delta_i^2$$

prove that:

$$\bar{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} R_{sim_i}$$

$$S_n = \sum_{i=1}^{n} (R_{sim_i} - R_{sim})^2$$

Proof 1: *Apparently, we have:*

$$\bar{\mu}_i = \sum_{j=1}^{i} R_{sim_j}/i$$

$$\Rightarrow \bar{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} R_{sim_i}$$

■

Proof 2: First, we can prove that:

$$
\begin{aligned}
&\sum_{i=1}^{n}(R_{sim_i}-\bar{\mu}_n)^2-\sum_{i=1}^{n-1}(R_{sim_i}-\bar{\mu}_{n-1})^2\\
=&(R_{sim_n}-\bar{\mu}_n)^2+\sum_{i=1}^{n-1}((R_{sim_i}-\bar{\mu}_n)^2-(R_{sim_i}-\bar{\mu}_{n-1})^2)\\
=&(R_{sim_n}-\bar{\mu}_n)^2+\sum_{i=1}^{n-1}(R_{sim_i}-\bar{\mu}_n+R_{sim_i}-\bar{\mu}_{n-1})(R_{sim_i}-\bar{\mu}_n-R_{sim_i}+\bar{\mu}_{n-1})\\
=&(R_{sim_n}-\bar{\mu}_n)^2+\sum_{i=1}^{n-1}(2R_{sim_i}-\bar{\mu}_{n-1}-\bar{\mu}_n)(\bar{\mu}_{n-1}-\bar{\mu}_n)\\
=&(R_{sim_n}-\bar{\mu}_n)^2+(2\sum_{i=1}^{n-1}R_{sim_i}-(n-1)\bar{\mu}_{n-1}-(n-1)\bar{\mu}_n)(\bar{\mu}_{n-1}-\bar{\mu}_n)\\
=&(R_{sim_n}-\bar{\mu}_n)^2+(2\sum_{i=1}^{n-1}R_{sim_i}-\sum_{i=1}^{n-1}R_{sim_i}-\frac{n-1}{n}\sum_{i=1}^{n}R_{sim_i})(\bar{\mu}_{n-1}-\bar{\mu}_n)\\
=&(R_{sim_n}-\bar{\mu}_n)^2+(\sum_{i=1}^{n-1}R_{sim_i}-\frac{n-1}{n}\sum_{i=1}^{n}R_{sim_i})(\bar{\mu}_{n-1}-\bar{\mu}_n)\\
=&(R_{sim_n}-\bar{\mu}_n)^2+(\sum_{i=1}^{n-1}R_{sim_i}-\frac{n-1}{n}\sum_{i=1}^{n-1}R_{sim_i}-\frac{n-1}{n}R_{sim_n})(\bar{\mu}_{n-1}-\bar{\mu}_n)\\
=&(R_{sim_n}-\bar{\mu}_n)^2+(\frac{1}{n}\sum_{i=1}^{n-1}R_{sim_i}+\frac{1}{n}R_{sim_n}-R_{sim_n})(\bar{\mu}_{n-1}-\bar{\mu}_n)\\
=&(R_{sim_n}-\bar{\mu}_n)^2+(\bar{\mu}_n-R_{sim_n})(\bar{\mu}_{n-1}-\bar{\mu}_n)\\
=&(R_{sim_n}-\bar{\mu}_n)(R_{sim_n}-\bar{\mu}_n-\bar{\mu}_{n-1}+\bar{\mu}_n)\\
=&(R_{sim_n}-\bar{\mu}_n)(R_{sim_n}-\bar{\mu}_{n-1})
\end{aligned}
$$

then, we have:

$$
\left.\begin{array}{l}\delta_i = R_{sim_i} - \bar{\mu}_{i-1}\\ S_i = S_{i-1} + \frac{i-1}{i}\delta_i^2\end{array}\right\} \Rightarrow S_n - S_{n-1} = \frac{n-1}{n}\delta_n^2
$$

$$
\begin{aligned}
&= \frac{n-1}{n}(R_{sim_n} - \bar{\mu}_{n-1})^2\\
&= (\frac{n-1}{n}R_{sim_n} - \frac{n-1}{n}\bar{\mu}_{n-1})(R_{sim_n} - \bar{\mu}_{n-1})\\
&= (\frac{n-1}{n}R_{sim_n} - \frac{1}{n}\sum_{i=1}^{n-1}R_{sim_i})(R_{sim_n} - \bar{\mu}_{n-1})\\
&= (R_{sim_n} - (\frac{1}{n}\sum_{i=1}^{n-1}R_{sim_i} + \frac{1}{n}R_{sim_n}))(R_{sim_n} - \bar{\mu}_{n-1})\\
&= (R_{sim_n} - \bar{\mu}_n)(R_{sim_n} - \bar{\mu}_{n-1})
\end{aligned}
$$

Therefore, we have:

$$
\begin{aligned}
S_n - S_{n-1} &= \sum_{i=1}^{n}(R_{sim_i} - \bar{\mu}_n)^2 - \sum_{i=1}^{n-1}(R_{sim_i} - \bar{\mu}_{n-1})^2\\
\Rightarrow S_n &= \sum_{i=1}^{n}(R_{sim_i} - \bar{\mu}_n)^2
\end{aligned}
$$

■

A.2 Examples of Log Files

For each test round of each deployment, fault injections information from the fault injector, state information of PSs, VMs and services from the monitor, testing results from the test case generator, analysis reports created by the reliability analyzer are logged in separate files. In the following sections, we give examples of all kinds of log files of the first test round of the Medium deployment. In all example files, component names are modified and timestamps are removed.

A.2.1 An Example of Injection Logs

The following is an example of injection logs, which shows the header and the first 20 injections of the injection log file. The header includes three parts: the LDG, the list of components in the LDG and the failure rate of each component. For each injection, the injection number, the target components and actions are logged.

```
INFO: Graph is: Graph[apache_vm_1=ps_1, drupal_vm_1=ps_2, mysql_vm_1=ps_1, apache_vm_2=ps_3, drupal_vm_2=ps_2, mysql_vm_2=ps_3,
apache_1=apache_vm_1, drupal_1=drupal_vm_1, mysql_1=mysql_vm_1, apache_2=apache_vm_2, drupal_2=drupal_vm_2, mysql_2=mysql_vm_2,
apache=(apache_1; apache_2; drupal), drupal=(drupal_1; drupal_2; mysql), mysql=(mysql_1; mysql_2)]

INFO: Objects are:
[ps_1, ps_2, ps_3, apache_vm_1, drupal_vm_1, mysql_vm_1, apache_vm_2, drupal_vm_2, mysql_vm_2, apache_1, drupal_1, mysql_1,
apache_2, drupal_2, mysql_2]

INFO: Failure Rate Map is:
[drupal_vm_1=0.03, drupal_vm_2=0.03, apache_1=0.03, ps_1=0.001, apache_2=0.03, mysql_1=0.03, mysql_2=0.03, drupal_1=0.03,
drupal_2=0.03, mysql_vm_2=0.03, mysql_vm_1=0.03, apache_vm_2=0.03, apache_vm_1=0.03, ps_2=0.001, ps_3=0.001]

INFO: Injection-0
INFO: Injection-1
INFO: Injection-2
INFO: Injection-3
INFO: Injection-4
INFO: Injected failure to SI: drupal_1 KILLED.
INFO: Injected failure to SI: drupal_2 KILLED.
INFO: Injection-5
INFO: Injection-6
INFO: Injection-7
INFO: Injection-8
INFO: Injection-9
INFO: Injected failure to VM: apache_vm_2 SUSPENDED.
INFO:       Suspended SI : apache_2
INFO: Injection-10
INFO: Injection-11
INFO: Injected failure to VM: drupal_vm_1 SUSPENDED.
INFO:       Suspended SI : drupal_1
INFO: Injection-12
INFO: Injected failure to SI: drupal_2 KILLED.
INFO: Injection-13
INFO: Injection-14
INFO: Injection-15
INFO: Injected failure to SI: mysql_1 KILLED.
INFO: Injection-16
INFO: Injected failure to VM: mysql_vm_1 SUSPENDED.
INFO:       SI : mysql_1 ALREADY FAILED.
INFO: Injection-17
INFO: Injected failure to VM: apache_vm_1 SUSPENDED.
INFO:       Suspended SI : apache_1
INFO: Injection-18
INFO: Injection-19
INFO: Injection-20
```

A.2.2 An Example of Monitoring Logs of PSs

A PS has two states: alive and not alive. The following is the monitoring information of PSs during the first 20 injections:

```
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
```

```
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
INFO: Server ps_1 is alive.true
INFO: Server ps_2 is alive.true
INFO: Server ps_3 is alive.true
```

A.2.3 An Example of Monitoring Logs of VMs

A VM also has two states: alive and not alive. The following is the monitoring information of VMs during the first 20 injections:

```
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
```

```
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
SEVERE: VM apache_vm_2 is NOT alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
SEVERE: VM drupal_vm_1 is NOT alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
SEVERE: VM mysql_vm_1 is NOT alive.
INFO: VM mysql_vm_2 is alive.
SEVERE: VM apache_vm_1 is NOT alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
```

```
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
INFO: VM apache_vm_1 is alive.
INFO: VM apache_vm_2 is alive.
INFO: VM drupal_vm_1 is alive.
INFO: VM drupal_vm_2 is alive.
INFO: VM mysql_vm_1 is alive.
INFO: VM mysql_vm_2 is alive.
```

A.2.4 An Example of Monitoring Logs of Services

The same to PSs and VMs, SIs and services have two states: alive and not alive. The following is the monitoring information of the Apache service during the first 20 injections, which firstly shows the states of all SIs of the service and then the state of the service for each injection.

```
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
SEVERE:      INSTANCE apache_2 is NOT alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
SEVERE:      INSTANCE apache_1 is NOT alive.
INFO:      INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:      INSTANCE apache_1 is alive.
```

```
INFO:       INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
INFO:       INSTANCE apache_1 is alive.
INFO:       INSTANCE apache_2 is alive.
INFO: SERVICE apache is alive.
```

A.2.5 An Example of Testing Result Logs

A testing result may be succeeded or failed. Beside of the testing result, the command, target and value of the testing action are also logged. In the following example, commands are all "open", targets are all the home page of the website and values are all void. The following is the testing results during the first 20 injections.

```
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
SEVERE: Event failed. Command: open-Target: /-Value:
SEVERE: Event failed. Command: open-Target: /-Value:
SEVERE: Event failed. Command: open-Target: /-Value:
SEVERE: Event failed. Command: open-Target: /-Value:
SEVERE: Event failed. Command: open-Target: /-Value:
SEVERE: Event failed. Command: open-Target: /-Value:
SEVERE: Event failed. Command: open-Target: /-Value:
SEVERE: Event failed. Command: open-Target: /-Value:
SEVERE: Event failed. Command: open-Target: /-Value:
SEVERE: Event failed. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
```

```
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
```

```
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
INFO: Event succeeded. Command: open-Target: /-Value:
```

A.2.6 An Example of the Analysis Report

The following is the analysis report for the first test round of the Medium deployment.

```
Setup
-----------------------------------------------------------------
                Inner Failure   Inner Reli-       Assessed Reli-
                Rate (IFR)      ability (IRE)     ability (RE)
-----------------------------------------------------------------
ps_1            0.00100000      0.99900050        0.99900050
ps_2            0.00100000      0.99900050        0.99900050
```

```
ps_3            0.00100000      0.99900050        0.99900050
apache_vm_1     0.03000000      0.97044553        0.96947557
apache_vm_2     0.03000000      0.97044553        0.96947557
drupal_vm_1     0.03000000      0.97044553        0.96947557
drupal_vm_2     0.03000000      0.97044553        0.96947557
mysql_vm_1      0.03000000      0.97044553        0.96947557
mysql_vm_2      0.03000000      0.97044553        0.96947557
----apache_1    0.03000000      0.97044553        0.94082324
----apache_2    0.03000000      0.97044553        0.94082324
apache          0.00350803      0.99649811        0.98953804
----drupal_1    0.03000000      0.97044553        0.94082324
----drupal_2    0.03000000      0.97044553        0.94082324
drupal          0.00439713      0.99561252        0.99300849
----mysql_1     0.03000000      0.97044553        0.94082324
----mysql_2     0.03000000      0.97044553        0.94082324
mysql           0.00350803      0.99649811        0.99649811
Application                                       0.98953804

Fault Injector: 1500 injections
-----------------------------------------------------------------
                Injected IFR    Injected IRE      Assessed RE
-----------------------------------------------------------------
ps_1            0.00000000      1.00000000        1.00000000
ps_2            0.00000000      1.00000000        1.00000000
ps_3            0.00066689      0.99933333        0.99933333
apache_vm_1     0.03390155      0.96666667        0.96666667
apache_vm_2     0.03597266      0.96466667        0.96402356
drupal_vm_1     0.03114673      0.96933333        0.96933333
drupal_vm_2     0.03183472      0.96866667        0.96866667
mysql_vm_1      0.03252319      0.96800000        0.96800000
mysql_vm_2      0.03459144      0.96600000        0.96535600
----apache_1    0.02429269      0.97600000        0.94346667
----apache_2    0.03114673      0.96933333        0.93446017
apache          0.00371207      0.99629481        0.98793148
----drupal_1    0.03874083      0.96200000        0.93249867
----drupal_2    0.03943407      0.96133333        0.93121156
drupal          0.00465413      0.99535669        0.99157049
----mysql_1     0.03045921      0.97000000        0.93896000
----mysql_2     0.02908558      0.97133333        0.93768246
mysql           0.00381112      0.99619614        0.99619614
Application                                       0.98793148

Monitor: 1500 logs
-----------------------------------------------------------------
                Monitored RE                      Assessed RE
-----------------------------------------------------------------
ps_1            1-0/1500=1.00000000               1.00000000
ps_2            1-0/1500=1.00000000               1.00000000
ps_3            1-1/1500=0.99933333               0.99933333
apache_vm_1     1-50/1500=0.96666667              0.96666667
apache_vm_2     1-54/1500=0.96400000              0.96402356
drupal_vm_1     1-46/1500=0.96933333              0.96933333
drupal_vm_2     1-47/1500=0.96866667              0.96866667
mysql_vm_1      1-48/1500=0.96800000              0.96800000
mysql_vm_2      1-52/1500=0.96533333              0.96535600
----apache_1    1-84/1500=0.94400000              0.94346667
----apache_2    1-98/1500=0.93466667              0.93446017
apache          1-6/1500=0.99600000 (IRE)         0.98793148
----drupal_1    1-102/1500=0.93200000             0.93249867
----drupal_2    1-101/1500=0.93266667             0.93121156
drupal          1-5/1500=0.99666667 (IRE)         0.99157049
----mysql_1     1-92/1500=0.93866667              0.93896000
----mysql_2     1-92/1500=0.93866667              0.93768246
mysql           1-4/1500=0.99733333 (IRE)         0.99619614
Application                                       0.98793148

Test: 15000 requests
-----------------------------------------------------------------
                Tested RE                         Assessed RE
-----------------------------------------------------------------
Application     1-150/15000=0.99000000            0.98793148
```