Towards Reliable Application Deployment in the Cloud

Ruichuan Chen[†], Istemi Ekin Akkus[†], Bimal Viswanath[§], Ivica Rimac[†], Volker Hilt[†]
[†]Nokia Bell Labs [§]University of Chicago

ABSTRACT

A common practice to increase the reliability of a cloud application is to deploy redundant instances. Unfortunately, such redundancy efforts can be undermined if the application's instances share common dependencies. This paper presents reCloud, a novel system that can efficiently find a reliable deployment plan for cloud applications. reCloud considers and avoids common dependencies shared across application instances that may lead to correlated failures, and works with applications that even have complex internal structures. reCloud utilizes various pieces of available dependency information (e.g., hardware, software and/or network dependencies) about the cloud infrastructure to quantitatively assess the reliability of the application's deployment plan with rigorous error bounds. This assessment further enables reCloud to find a deployment plan that balances between reliability and other criteria such as application performance and resource utilization. We implemented a fully functional system. The experimental results show that, even in a large cloud environment with more than 27K hosts, reCloud needs only 30 seconds to find a deployment plan that is one order of magnitude more reliable than the common practice.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; **Reliability**; **Availability**; **Redundancy**; • **Networks → Data center networks**;

KEYWORDS

Cloud reliability, dagger sampling, simulated annealing, network transformations

ACM Reference Format:
Ruichuan Chen[†], Istemi Ekin Akkus[†], Bimal Viswanath[§], Ivica Rimac[†], Volker Hilt[†]. 2017. Towards Reliable Application Deployment in the Cloud. In *Proceedings of CoNEXT '17*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3143361.3143388

1 INTRODUCTION

Many developers are moving their applications from self-maintained infrastructure to the cloud for increased reliability. One common practice to increase the reliability of a cloud application is to use redundancy techniques, whereby a developer deploys multiple instances of her application and requests each instance to be deployed into different regions [37]. The intent is that some instances will survive the failures of others, so that the application can continue to function properly.

However, cloud service outages are still common even with redundancy techniques. A recent study [32] shows that 25 out of today's 32 popular cloud applications have more than 8.8 hours of annual downtime. The redundancy efforts can be undermined if all (or many) application instances fail simultaneously due to shared dependencies [26, 80]. For example, a recent power disruption at GitHub's primary data center caused a cascading failure leading to a service outage, and in turn affected many GitHub-dependent applications [29]. As another example, an error in Amazon's storage service caused cascading failures across multiple EC2 instances, triggering correlated failures of the applications deployed in the affected EC2 instances [6]. A similar incident recently also happened to Microsoft Azure where a power event affected Azure's storage tier, which in turn affected the services deployed in 26 out of the 28 data center regions [55]. In these examples, the power supply and the storage service were the shared dependencies that caused correlated failures, undermining applications' redundancy efforts.

Correlated failures are common. For example, Google reported that close to 37% of failures in its storage systems were correlated [27]. There are a large number of mechanisms to detect, localize and alleviate correlated failures [8, 17, 35, 42, 47, 50, 59, 78]. These mechanisms, however, are applicable only after failures occur. It would be better to take proactive actions to prevent such correlated failures.

To our knowledge, INDaaS [80] is the first, and also the state-of-the-art, system that attempts to proactively prevent correlated failures of cloud applications. INDaaS compares the reliability of an application's given deployment plans, and selects the most reliable plan for deployment. Here, each deployment plan specifies where the application instances should be deployed in the cloud. INDaaS, however, has four technical shortcomings that make it unsuitable for practice. First, INDaaS considers an application's potential deployment plans according to their *qualitative* or *relative* metrics, and does not produce a quantitative assessment of their reliability (which is however required for service quality auditing and compliance). Second, INDaaS compares the reliability of only *given* deployment plans, with no capability of searching for these deployment plans in the first place. Third, INDaaS treats an application as a monolithic entity, and does not consider the application's internal structures. Fourth, INDaaS scales poorly in large cloud environments (on the order of hours [80]).

In this paper, we design a novel system, reCloud, for cloud providers to address the shortcomings of prior systems. reCloud can efficiently find a deployment plan for a cloud application that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CoNEXT '17, December 12–15, 2017, Incheon, Korea
© 2017 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5422-6/17/12…$15.00
https://doi.org/10.1145/3143361.3143388

fulfills the developer's requirements even in a large cloud environment. reCloud considers various common dependencies which may lead to correlated failures, and incorporates any dependency information about the cloud infrastructure, such as the states of the hosts and switches, as well as their dependencies such as power supplies, cooling systems, software and firmware. With this information, reCloud can quantitatively assess the reliability of a cloud application's deployment plan with rigorous error bounds.

There are a few main challenges for enabling such a system. First, given an application's deployment plan, it is an NP-hard problem to quantitatively assess the reliability of the application [9]. Our reCloud system enables an efficient and quantitative reliability assessment based on a high-performance approximate approach — *dagger sampling* [45]. Furthermore, reCloud gives a rigorous bound on the approximation error of its reliability assessment. This bound is crucial because it helps both application developer and cloud provider gain confidence on the accuracy of reCloud's reliability assessment.

Second, it is non-trivial to find a reliable deployment plan that fulfills an application developer's reliability requirements (e.g., at least 4 out of the application's 5 redundant instances are required to be alive). There has been surprisingly little prior work regarding this reliable deployment search. One naïve approach would be to generate all possible deployment plans, assess them, and select the best one that fulfills the requirements. This approach, however, does not scale to the size of today's cloud infrastructure. reCloud systematically explores the huge space of all deployment plans based on *simulated annealing* [15, 43] and *network transformations* [60], and can quickly converge on a reliable deployment plan.

Third, it is important for reCloud to consider correlated failures and integrate various pieces of dependency information, whenever available, into the reliability assessment. While hosts and switches as well as their connectivity comprise a significant part of an application deployment's reliability, they may share common dependencies (e.g., power supplies, cooling systems, and software) which can bring down the hosts and switches in a correlated fashion. reCloud considers common dependencies, and integrates various pieces of available dependency information seamlessly with no system changes.

Fourth, another challenge is that cloud applications can be complex with components requiring connectivity among themselves (e.g., frontend servers to backend databases, or more complex microservices [48]). As a result, the reliability assessment of an application deployment needs to consider the internal structures of the application (e.g., frontend servers should be reachable from the Internet, while frontend servers and backend databases should be mutually reachable; or, some microservice components should be well interconnected). While prior systems treat an application as a monolithic entity, reCloud can cover application's complex internal structures to assess the reliability of an application deployment.

Finally, high reliability is only one objective when it comes to deploying an application into the cloud. Other objectives routinely aimed for are: 1) application performance for developers, and 2) resource utilization for cloud providers. As a result, the ability to support the combination of multiple objectives becomes crucial. reCloud incorporates such additional objectives while searching for a reliable deployment plan, therefore both application developers and cloud providers can make informed decisions about the potential trade-offs.

We implemented reCloud with all the aforementioned capabilities. The experimental results show that, even in a large cloud environment with more than 27K hosts, reCloud needs only 30 seconds to find a deployment plan that is typically one order of magnitude more reliable than the common practice. Altogether, reCloud makes the following contributions:

- It is, to our knowledge, the *first* system which: 1) proactively finds a cloud application's reliable deployment plan that fulfills the application developer's reliability requirements, and 2) quantitatively assesses the reliability of an application's deployment plan with rigorous error bounds.
- It considers common dependencies which may lead to correlated failures, and incorporates available dependency information of the cloud infrastructure, as well as the internal structures of a cloud application, into reliability assessment.
- It is able to find an application's deployment plan that balances between reliability and other criteria such as application performance and resource utilization.
- It scales well even in a large cloud environment.

The rest of this paper is organized as follows. The next section gives an overview of reCloud. Section 3 elaborates upon the design details of reCloud. We present the evaluation results in Section 4, and describe the related work in Section 5. Finally, we conclude the paper in Section 6.

2 RECLOUD OVERVIEW

This section gives an overview of reCloud, including its fault model, scenario and high-level workflow.

2.1 Fault Model

reCloud considers three types of infrastructure components: hardware components (e.g., servers, switches, power supplies, and cooling systems), software components (e.g., software and firmware deployed at hardware components), and network components (e.g., network connectivity across hardware components). These components are the most common causes of various failures in the cloud [28, 32, 78]. Infrastructure components may also share common dependencies which can lead to correlated failures. In other words, if the common dependencies fail for whatever reason, all components relying on these dependencies will fail simultaneously. Furthermore, each component is in one of the following two states: alive or failed. Partially failed components are treated as failed.

Today, cloud providers normally use cloud management platforms to manage their infrastructure. Some examples include the platforms from VMware [74], Cisco [18], Embotics [24] and OpenNebula [57]. These platforms provide a rich set of management features, such as monitoring the states and dependencies of infrastructure components, and how they are connected in the cloud.

In addition to commercial cloud management platforms, there are also a large number of tools [2, 8, 11, 16, 17, 21–23, 36, 40–42, 44, 54, 56, 59, 71, 82] that can be used to acquire the dependency information about various cloud infrastructure components. For example, as suggested in [80], HardwareLister [36] can obtain the

detailed hardware configurations (e.g., CPU/memory/mainboard configuration, firmware version, etc.); apt-rdepends [21] can recursively extract the dependencies of software packages and libraries; NSDMiner [54, 56, 59] can identify the network dependencies by passively monitoring and analyzing the network traffic.

With the aforementioned information acquired by cloud management platforms or specialized tools, cloud providers can measure each infrastructure component's downtime within a time window [7, 51], and in turn, each component's *failure probability* $p = downtime/windowLength$ [28, 72]. This can be represented as the annual failure rate. Such an approach has already been realized to acquire the failure probability of hardware and network components [28, 72]. Arguably, it may not be trivial to acquire the failure probability of software components. Nevertheless, such software failure probability could be monitored, or simulated using the FIFL framework via fault injections [82], or estimated using the publicly-available CVSS scores [25] similar to [38, 58, 81]. Measurement studies on various components' failure probabilities have been performed in real-world systems [28, 32, 52, 61, 67, 72, 73].

Notice. As discussed in §3.4, reCloud works with limited dependency information (e.g., only network dependencies), and also works with limited or even no failure probabilities. In addition, as shown in §4, reCloud can quickly adapt to varying system conditions collected at (near) real-time. Our contribution in this paper is not how best to acquire a fault model in the cloud, but how best to exploit it to find a reliable deployment plan for a cloud application.

2.2 Scenario and System Workflow

In this paper, we focus on deploying an application into a cloud data center. At a high-level, reCloud works as follows. The application developer first specifies her reliability requirements to the cloud provider, including the following four parameters:

- N: the total number of application instances to be deployed for redundancy.
- K: the minimum number of deployed instances that need to be alive in order to meet business needs.
- $R_{desired}$: the desired reliability score, which is defined as the desired probability that at least K out of N deployed instances are alive, and can be decided based on the application developer's expected deployment cost. An alternative way could be to allow an application developer to specify the acceptable annual service downtime which can then be translated to $R_{desired}$.
- T_{max}: the maximum amount of time to be spent on searching for a reliable deployment plan that fulfills the application developer's requirements. For practicality, this should be within minutes, not hours as in prior systems [80].

After receiving the reliability requirements, the cloud provider starts with a random deployment plan and uses any available dependency information (e.g., hardware, software and/or network dependencies) to assess the deployment plan's reliability. Here, a deployment plan specifies which hosts the application instances should be deployed onto.

If the desired reliability score $R_{desired}$ can be satisfied by the current deployment plan, the cloud provider deploys the application accordingly. If not, the cloud provider continues the search by generating a different deployment plan and then assessing its reliability to check whether it satisfies the desired reliability score. If the cloud provider cannot find such a deployment plan within the maximum search time T_{max}, the cloud provider informs the application developer that her current reliability requirements cannot be fulfilled.

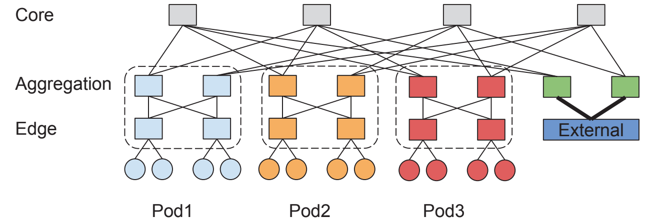


Figure 1: Fat-tree ($k = 4$) with external connectivity.

The above K-of-N redundancy deployment represents a simple scenario, where an application can work without requiring connectivity among its components. We first use this scenario to describe our system design. We will later describe how to handle sophisticated cases where applications have complex internal structures.

3 RECLOUD DESIGN

In this section, we first briefly describe a cloud data center's example architecture for ease of the following presentation (§3.1). Then, we propose two cooperative techniques for reliable application deployment in the cloud: the quantitative reliability assessment of an application deployment (§3.2), and the proactive search for an application's reliable deployment plan (§3.3). Finally, we discuss how to deal with a situation where we can only acquire limited dependency information (§3.4).

3.1 Data Center Example: Fat-Tree

There are a large number of data center architectures [1, 3, 4, 31, 33, 34, 49, 50, 53, 70, 77]. Although reCloud is general and works with any of these architectures (see §3.2), we use the classic fat-tree [3] as an example to demonstrate our system design.

Figure 1 shows a fat-tree consisting of a number of hosts (circles) and k-port switches (rectangles). Switches peering with external entities are called *border switches*. We apply Google's approach as an example to manage fat-tree's external connectivity via using a dedicated pod [69]; therefore, the switches in that pod are border switches. This architecture provides the full external bandwidth to all pods in the data center.

3.2 Reliability Assessment of a Deployment

Suppose an application developer wants to deploy her application into a data center operated by a cloud provider. For redundancy, the developer requests the cloud provider to deploy N instances of the application onto some hosts in the data center, and requires at least K deployed instances to be alive (i.e., reachable from any of the border switches used for external connectivity). Hosts and switches may fail due to hardware, software and/or network failures.

For descriptive clarity, we first consider only hosts and switches, as well as their connectivity. They can certainly share various types

Table 1: Illustrative example of failure state generation. For clarity, we consider only hosts and switches here, and will consider other dependencies later.

	Round 1	Round 2	⋯	Round X
Host 1	Alive	Failed	⋯	Alive
⋯	⋯	⋯	⋯	⋯
Host h	Alive	Alive	⋯	Failed
Switch 1	Failed	Failed	⋯	Alive
⋯	⋯	⋯	⋯	⋯
Switch s	Alive	Alive	⋯	Alive

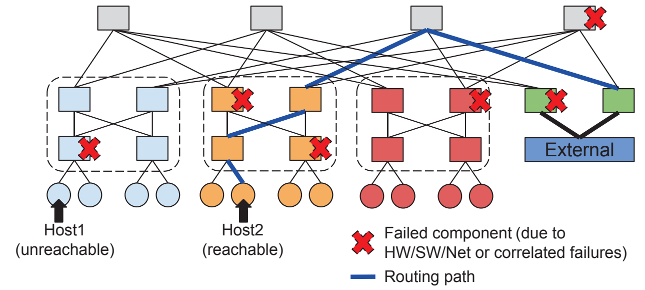


Figure 2: Route-and-check in one round for a deployment plan using (Host1, Host2) with $N = 2$ and $K = 1$. Red crosses mark failed components (e.g., due to hardware, software, network, or correlated failures). This plan is considered 'reliable' in this round, because at least one host (i.e., Host2) is reachable from a border switch.

of common dependencies (e.g., power supplies, cooling systems, software and firmware) which may lead to correlated failures. In §3.2.3, we will describe how such dependencies can be integrated in a correlated fashion.

3.2.1 Strawman Design using Monte-Carlo Sampling

Assessing the reliability of an application deployment (i.e., calculating how likely the application instances are reachable from any of the border switches) can be reduced to a classic two-terminal reliability problem. However, this problem is NP-hard [9], forcing us to use an approximate approach, e.g., Monte-Carlo sampling (as used in the state-of-the-art INDaaS system [80]). Next, we describe our strawman design based on Monte-Carlo sampling, and explain its practical limitations. Then, we will describe how to enhance the strawman design to practically assess the reliability of an application deployment plan.

Step 1: Generate failure states for infrastructure components. We run many Monte-Carlo sampling rounds to generate the failure states of the infrastructure components. Specifically, in each round, we randomly generate the failure state of each component according to this component's failure probability (defined in §2.1). Suppose a component fails with probability p. A uniformly random number $r \in [0, 1)$ can be generated to decide whether this component is set to be 'failed' in a round: if $r < p$, the component is 'failed'; otherwise, it is 'alive'. Repeating this process, we can randomly generate the failure states for each component across all rounds. This process produces a table similar to Table 1, where each column represents one round and each row represents the failure states of one component across all rounds.

Step 2: Route and check reachability. Suppose the cloud provider follows a given deployment plan to deploy application instances onto N specific hosts. For each such host, we can run the routing protocol[1] to check whether this host is reachable from any of the border switches (given the failure states of infrastructure components). This implicitly considers some correlated failures, e.g., an edge/ToR switch failure makes all hosts under that switch unreachable. In each round, if at least K hosts are reachable, then this deployment plan is considered reliable for this round (see Figure 2). If the deployment plan is reliable in Y out of the total X rounds, then the reliability score of this deployment plan is $R = Y/X$. As we will show in the evaluation, this "route-and-check" process is very efficient. To work with another data center architecture instead of fat-tree, we only need to change this step's routing protocol to the one used in that architecture accordingly.

[1]For example, the fat-tree routing protocol in a fat-tree data center.

Note that, the "route-and-check" process can be performed in parallel via MapReduce [20]: A master node distributes portions of rounds to worker nodes. Each worker node performs the "route-and-check" for the assigned rounds. The master node then gathers the results from each worker node to compute the overall reliability score according to the results of all rounds.

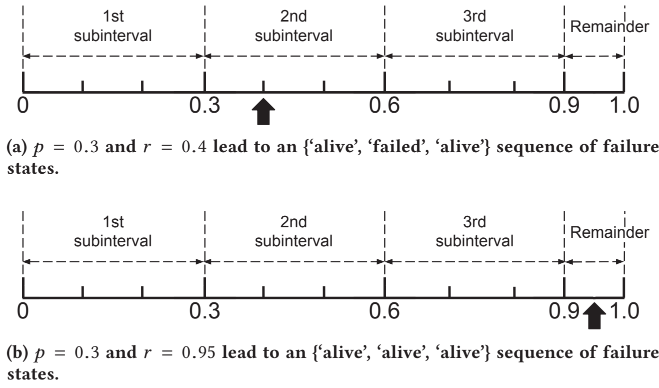
Problems in the Strawman Design. The design based on Monte-Carlo sampling is straightforward, but it requires an individual failure state generation for each infrastructure component in each round. In other words, there will be $C \times X$ failure state generations with C components and X rounds.

Generating so many failure states can be expensive for two reasons. First, today's data centers have an increasingly large number of infrastructure components including hardware, software and network components. For example, an Amazon data center contains tens of thousands of infrastructure components [19]. Second, individual components in a data center are fairly reliable (e.g., with low annual failure probabilities of 1%) [28, 32, 52, 61, 67, 72, 73], so that a prohibitively large number of sampling rounds are required to calculate an accurate reliability score for an application deployment plan. For these reasons, Monte-Carlo sampling is unsuitable for assessing the reliability of a deployment plan, especially in large data centers (more in §4.2.1).

3.2.2 Practical Design using Dagger Sampling

reCloud derives its practicality from using *dagger sampling* [45] as the basis of generating the failure states for infrastructure components. Dagger sampling is a highly efficient technique proposed to specifically handle two-state variables and low-probability events. It is well-suited to our scenario because each component has two states (i.e., 'failed' or 'alive') and these components fail with low probabilities. Next, we describe the dagger sampling and how we adapt it to assess the reliability of an application deployment.

Dagger Sampling. Suppose an infrastructure component fails with probability p. Let s be the largest integer not larger than $1/p$ (i.e., $s = \lfloor 1/p \rfloor$). Then, the interval [0,1) can be divided into s subintervals each of length p, plus a remainder section if $s \times p < 1$.



(a) $p = 0.3$ and $r = 0.4$ lead to an {'alive', 'failed', 'alive'} sequence of failure states.

(b) $p = 0.3$ and $r = 0.95$ lead to an {'alive', 'alive', 'alive'} sequence of failure states.

Figure 3: Illustrative examples of dagger sampling.

Figure 3a shows an example with $p = 0.3$. There are 3 subintervals each of length 0.3, plus a remainder section of length 0.1.

During the failure state generation, each of the s subintervals corresponds to one sampling round for an infrastructure component, thus producing s rounds. If a uniformly-generated random number $r \in [0, 1)$ falls within the i-th subinterval, the component is set to be 'failed' in the i-th round and 'alive' in all other $s - 1$ rounds. If r falls within the remainder section, the component is set to be 'alive' in all s rounds. Note that, there is no bias introduced by the presence or absence of a remainder section, i.e., the expected ratio of the 'failed' rounds across all rounds is still p [45, 63].

This dagger sampling process can be best illustrated with examples in Figure 3. For a component with failure probability $p = 0.3$, the largest integer not larger than $1/p$ is $s = 3$; therefore, there are 3 subintervals. Figure 3a shows that, if the generated random number is $r = 0.4$, which falls within the second subinterval, then the component is set to be 'failed' in the second round and 'alive' in other rounds. That means, the component's failure states for the 3 rounds will be {'alive', 'failed', 'alive'}. Similarly, in Figure 3b, if the random number is $r = 0.95$, which falls within the remainder section, then the component's failure states for the 3 rounds will all be 'alive'.

Generating the failure states of a component via dagger sampling is much more efficient than via Monte-Carlo sampling, especially when a component's failure probability p is small. The reason is that a *single* random number r can be used to determine a component's failure states for a large number of rounds (i.e., s rounds). To generate a component's failure states for a new s number of rounds, one needs to generate another random number. We call each segment of s rounds a *dagger cycle*, where s represents the *dagger cycle length*.

Note that, dagger sampling only fails a component in one or zero round within a dagger cycle. In reality, a component can certainly fail in multiple rounds within a dagger cycle although rarely (because each component is fairly reliable [28, 32, 52, 61, 67, 72, 73]). Dagger sampling uses a single random number to decide a component's failure states for many rounds, but still keeps the statistical properties of Monte-Carlo sampling [45, 63].

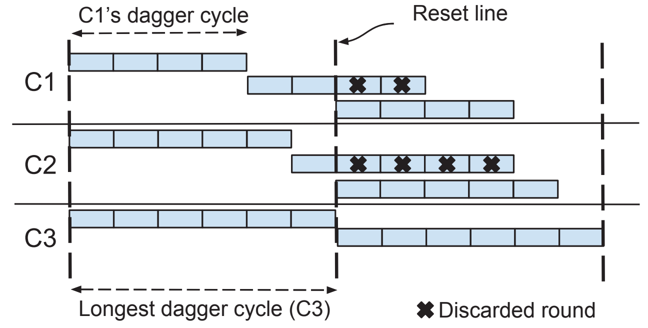


Figure 4: Extended dagger sampling with dagger cycle reset at the end of the longest dagger cycle (i.e., C3).

Note also that, the failure probability of a component may vary during its lifetime, normally following a "bathtub curve" with more failures at the beginning and the end of its lifecycle [66, 79]. reCloud can adjust p quickly to handle such varying failure probabilities whenever they are available.

Although dagger sampling can efficiently generate failure states for each individual component, a realistic data center may have various components with different failure probabilities. As a result, these components may have different dagger cycle lengths. The original dagger sampling needs to be extended to accommodate these cases.

Extended Dagger Sampling. To handle infrastructure components with different dagger cycle lengths, we still use the original dagger sampling technique to independently generate the failure states of each individual component. However, we reset all components' dagger cycles at the end of the longest dagger cycle, as suggested in [63].

Suppose there are three components C_1, C_2 and C_3. They fail with probabilities of p_1, p_2 and p_3. Without loss of generality, we assume $p_1 > p_2 > p_3$. Therefore, their dagger cycle lengths are $s_1 = \lfloor 1/p_1 \rfloor$, $s_2 = \lfloor 1/p_2 \rfloor$ and $s_3 = \lfloor 1/p_3 \rfloor$, where $s_1 \leq s_2 \leq s_3$. In this setup, the component C_3 has the longest dagger cycle length s_3. As shown in Figure 4, we apply the original dagger sampling to generate the failure states of each component independently. For each component, its respective dagger cycles are concatenated. We then truncate the dagger cycles of the components C_1 and C_2 at the end of every s_3 rounds regardless of whether the dagger cycles of C_1 and C_2 are complete. This extended dagger sampling enables us to generate failure states for infrastructure components with different failure probabilities over many such cycles, without a bias [63].

Practical Design with Extended Dagger Sampling. Recall that, for assessing the reliability of a deployment plan in the strawman design, we first generate the failure states for all the infrastructure components across many rounds, and then run the routing protocol to check how many instances of the application can be reached from any of the border switches. If a desired number of instances are reachable in a round, the application deployment is considered reliable in this round.

To generate the failure states of infrastructure components, the strawman design uses Monte-Carlo sampling, which becomes expensive due to the large numbers of components and sampling

rounds. We replace it with the extended dagger sampling, which is much more efficient as we will also show in the evaluation. The rest of the design remains the same.

Accuracy of Reliability Assessment. So far, we have described how reCloud can assess the reliability of a deployment plan, and generate a reliability score for it. We now describe a method to estimate the accuracy of our reliability assessment based on extended dagger sampling.

Suppose we use n rounds to assess a deployment plan. reCloud produces a result list $L = \{d_1, d_2, \cdots, d_n\}$, where $d_i = 1$ represents the deployment plan is reliable in the i-th round and $d_i = 0$ represents it is unreliable in that round. Then, the reliability score of this deployment plan is assessed as:

$$R = \frac{\sum_{i=1}^{n} d_i}{n} \tag{1}$$

Due to the variance reduction effect of dagger sampling, the variance V of the above assessed reliability score R can be *conservatively* estimated as [45]:

$$V = \frac{\mathrm{Var}[L]}{n} \tag{2}$$

Here, $\mathrm{Var}[L]$ denotes the variance of the result list L. The variance reduction effect of dagger sampling indicates that dagger sampling converges faster than the Monte-Carlo sampling. In other words, with the same number of rounds, dagger sampling-based approach produces more accurate and stable reliability scores than the Monte-Carlo-based approach; and, to achieve the same level of assessment accuracy, dagger sampling-based approach needs fewer rounds.

The central limit theorem [76] indicates that the reliability score R follows a normal distribution; therefore, the 95% confidence interval width of this reliability score can be calculated as [75]:

$$CIW_{95\%} = 4 \times \sqrt{V} \tag{3}$$

This confidence interval width is a rigorous accuracy measure that indicates a range within which the reCloud-produced reliability score differs from the ground-truth reliability score. Such a measure is critical because it gives both cloud providers and application developers a quantitative understanding of the accuracy of our reliability assessment.

3.2.3 Additional Dependencies

So far, we have considered only the hosts, switches and the connectivity among them. These components, however, may share additional common dependencies which may also be captured by the cloud management platforms or specialized tools (see §2.1). For example, hosts and switches may depend on the same power supplies, cooling systems, software and firmware. Failures in these shared dependencies may lead to correlated failures at hosts and switches, and bring down the associated application instances simultaneously, similar to the recent failure event where the power disruption at GitHub's primary data center affected many GitHub-dependent applications [29]. Therefore, it is important to consider additional dependencies during the reliability assessment of a deployment plan.

reCloud automatically constructs a fault tree [62] for each host/switch's dependencies whenever they are available, similar to [62, 80]. Figure 5 shows an example of how such a fault tree is built. Suppose a host runs an operating system and some software

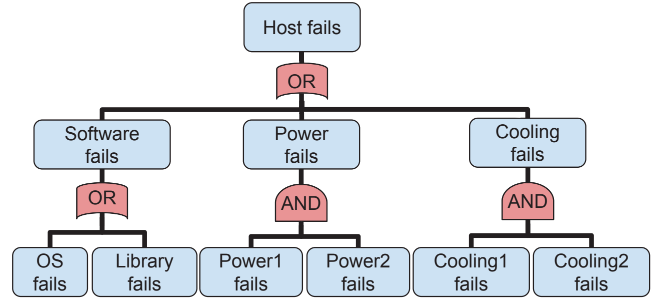


Figure 5: Example of a host's fault tree. Multiple hosts' fault trees can be connected if they share dependencies.

library. Besides, the host has two redundant power supplies, and the rack containing this host has two redundant cooling systems. To build this host's fault tree, the top node is labeled "host fails". The top node has three child nodes: "software fails", "power fails" and "cooling fails". There is a logical *OR* gate connecting the top node to its three child nodes. In addition, the "software fails" connects to the "OS fails" and "library fails" with an *OR* gate, the "power fails" connects to the two power supplies with an *AND* gate, and the "cooling fails" connects to the two cooling systems also with an *AND* gate. The resulting fault tree represents: 1) the host fails if the software, the power *or* the cooling fails, 2) the software fails if the operating system *or* the library fails, 3) the power fails only if *both* power supplies fail, and 4) the cooling fails only if *both* cooling systems fail.

reCloud constructs fault trees for the additional dependencies of all hosts and switches. Multiple fault trees are connected if they share common dependencies. Using fault trees allows reCloud to incorporate various types of dependencies (e.g., hardware, software and network dependencies) with complex logical relationships. Such flexibility becomes important when emerging tools can help cloud providers know more about their infrastructure.

After building the fault trees for each host/switch's dependencies, we apply the extended dagger sampling to generate the failure states of all these dependencies for many rounds. In each round, during the "route-and-check" step, we first reason about whether each host or switch fails according to their own failure states, as well as the states of these dependencies in their fault trees and the logical relationships among these dependencies. We then filter out the failed hosts and switches for that round. Besides the reasoning and filtering, there are no additional changes.

3.2.4 Complex Application Structures

We have described reCloud's reliability assessment assuming an application only requires at least K out of its N instances to be reachable from any of the border switches. Each application instance can function well without interactions with others. In reality, however, an application can be complex and may require connectivity among its components, each of which may have multiple instances. While prior systems treat an application as a monolithic entity, we describe how reCloud integrates complex application structures into reliability assessment.

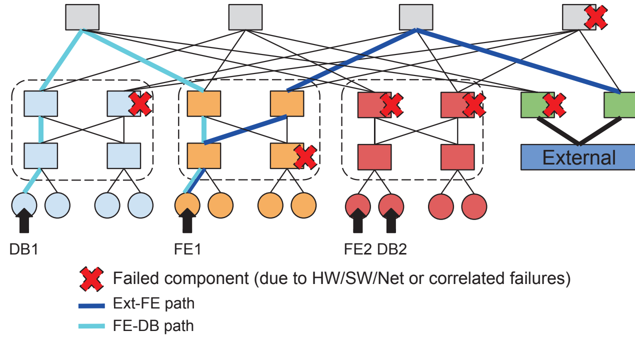


Figure 6: Route-and-check in one round for a deployment plan requiring connectivity between frontend servers (FE) and backend databases (DB), with $N_{FE} = 2$, $N_{DB} = 2$, $K_{FE,Ext} = 1$ and $K_{DB,FE} = 1$. Red crosses mark failed components (e.g., due to hardware, software, network or correlated failures). This plan is considered 'reliable' in this round, because at least one frontend (FE1) is reachable from a border switch and it can also reach at least one database (DB1).

The idea is to modify our reliability assessment: instead of only considering if the application instances can be reached from border switches, the "route-and-check" step in §3.2.1 checks whether the connectivity among application components is preserved according to the developer's requirements. This enables reCloud to assess the reliability of a deployment plan for an application with complex internal structures.

To enable this functionality, the reliability requirements specified by the developer are now defined for an application's each component C_i, as follows:

- N_{C_i}: the total number of instances for component C_i to be deployed for redundancy, and
- K_{C_i,C_j} for each component C_j: the minimum number of deployed instances of component C_i that need to be reachable from component C_j. Here, C_j can be another application component or a border switch used for external connectivity.

Figure 6 shows an example. Suppose a developer deploys a two-layer application, requiring at least 1 of the application's 2 frontend servers to be alive (i.e., reachable from the border switches), and at least 1 of its 2 backend database servers to be reachable from the alive frontend servers (which are also reachable from the border switches). To assess the reliability of a deployment plan, reCloud generates the failure states of infrastructure components for many rounds using the extended dagger sampling with no changes. However, as shown in Figure 6, the deployment plan in a round is considered reliable only if the "route-and-check" succeeds: 1) between the border switches and at least one frontend host, and 2) between the border-switch-reachable frontend host(s) and at least one backend database host.

It is worth mentioning that, this technique can also be used to deal with the increasingly popular "microservices-based" cloud application [48], which may consist of tens or even hundreds of components with complex communication patterns. Our technique to handle complex application structures allows the developer to specify the reliability requirements of each component, so that reCloud can incorporate these requirements into the reliability assessment process.

3.3 Reliable Deployment Search

With our reliability assessment technique, we now present how reCloud searches for a reliable deployment plan. This ability is completely missing in the state-of-the-art INDaaS system [80].

Suppose there are H hosts in a data center, and for reliability, an application developer requests the cloud provider to choose N hosts to deploy her application instances. In total, there are $\binom{H}{N}$ deployment plans without considering any instances on the same host. Even with additional heuristics like "do not use multiple hosts from the same rack", the number of potential deployment plans is still huge, especially in a large data center. Choosing hosts for a deployment to achieve high reliability is a typical NP-hard combinatorial optimization problem.

3.3.1 Search via Simulated Annealing and Network Transformations

To accelerate the search for a reliable deployment plan in a huge space to fulfill the developer's requirements, we first employ *simulated annealing* [15, 43]. Its original idea is that, as we randomly explore the huge space of potential deployment plans, we accept not only more reliable deployment plans, but also less reliable deployment plans with slowly decreasing probability. By not always discarding less reliable deployment plans during the search, we avoid being trapped in a local optimum, and can conduct a more extensive search for the global optimum.

However, a direct adoption of the classic simulated annealing does not work since its probability setting for accepting less reliable deployment plans fits badly in our scenario (see §3.3.2). In addition, data centers are normally designed to create network symmetry which we can utilize to accelerate the search. We will describe how to use the *network transformations* technique [60] to exploit network symmetry and then adjust the system parameters, to enhance simulated annealing to support our reliable deployment search.

Recall from §2.2 that, an application developer specifies to the cloud provider that she wants to deploy N application instances and requires at least K of them to be alive. The developer also specifies the desired reliability score $R_{desired}$ and the maximum search time T_{max}. The cloud provider then performs the following 6 steps to search for a reliable deployment plan:

Step 1: Generate an initial deployment plan. The cloud provider randomly selects N hosts for the application's instances. This selection can use any additional heuristics such as "no hosts from the same rack or pod".

Step 2: Assess the reliability of the initial deployment plan. The cloud provider treats the initial deployment plan as the current plan, and computes its reliability score $R_{current}$ (see §3.2).

Step 3: Generate a neighboring deployment plan, and check its equivalence with respect to network symmetry. The cloud provider generates a neighboring deployment plan by randomly replacing one host used in the current deployment plan by a new, randomly chosen host.

The cloud provider then applies the network transformations technique [60] to simplify the representations of the two networks

involved in the current and the neighboring deployment plans, respectively. With the simplified networks, the cloud provider checks whether the neighboring deployment plan is equivalent to the current plan with respect to both the network symmetry and the component failure probabilities (when available) between the two deployment plans. If they are equivalent, the cloud provider repeats this step to generate another neighboring deployment plan. Using the network transformations technique to exploit network symmetry avoids unnecessary, redundant operations and accelerates the search for a reliable deployment plan.

Note that, while data centers are normally designed to create network symmetry, the infrastructure components may fail with various different probabilities. The network transformations technique works best when components of the same type fail with the same or similar probability, which is usually the case in data centers. However, if components of the same type fail with very different probabilities, they are logically treated as of different types in our equivalence checking.

Step 4: Assess the reliability of the neighboring deployment plan. The cloud provider computes the reliability score $R_{neighbor}$ of the neighboring deployment plan (see §3.2).

Step 5: Decide whether to accept the neighboring deployment plan. If $R_{neighbor} \geq R_{current}$, the cloud provider accepts the neighboring deployment plan as the new current plan. This plan is then used as the basis in the next iteration. If $R_{neighbor} < R_{current}$, the cloud provider can still accept the neighboring plan as the new current plan with our specially-designed acceptance probability (see §3.3.2). Note that, the acceptance probability setting in the classic simulated annealing fits badly in our scenario, as we will discuss in §3.3.2.

Step 6: Repeat or terminate. The cloud provider repeats the Steps 3-5, during which if the desired reliability score is satisfied (i.e., $R_{desired} \leq R_{current}$), the cloud provider reports the current deployment plan to the developer. Otherwise, if the maximum search time T_{max} has elapsed, the cloud provider informs the developer that her requirements cannot be fulfilled.

Guarantee. With simulated annealing, the probability that the cloud provider terminates with the most reliable deployment plan approaches 1.0 with the increased number of annealing iterations [30]. In practice, reCloud can find a reliable deployment plan very quickly (see §4).

3.3.2 Acceptance Probability

In Step 5, the cloud provider decides whether to accept a newly-generated neighboring deployment plan. With some probability, a neighboring plan with a lower reliability score than the current plan is accepted. Adjusting this acceptance probability properly is crucial to the reliable deployment search. We denote this probability as Pr[accept]. In simulated annealing, this probability is generally set as [15, 43]:

$$\Pr[\text{accept}] = \exp\Big(-\Big|\frac{\Delta}{t}\Big|\Big) \tag{4}$$

Here, Δ denotes the difference between the reliability scores of the current deployment plan and the neighboring plan — the bigger the difference, the more unreliable the neighboring plan than the current plan, leading to a lower chance of accepting the neighboring plan. t denotes the annealing temperature which slowly decreases during the course of the annealing process — the higher the temperature, the higher the chance of accepting a less reliable neighboring plan. The appropriate settings of Δ and t allow reCloud to explore the huge search space efficiently (with the help of Step 3, i.e., network transformations), but the classic practice of setting these parameters does not fit in our scenario.

Setting Δ. Let $R_{current}$ and $R_{neighbor}$ denote the reliability scores of the current and the neighboring deployment plans. In classic simulated annealing, the difference Δ between the two reliability scores is normally set as the absolute value of the difference (i.e., $\Delta = |R_{current} - R_{neighbor}|$). For example, if $R_{current} = 0.999$ and $R_{neighbor} = 0.99$, then Δ is only 0.009.

However, in terms of reliability, these reliability scores indicate the current deployment plan is one order of magnitude more reliable than the neighboring plan. To reflect this effect, we adjust the classic simulated annealing to amplify this difference, and set Δ to:

$$\Delta = \Big|\log\Big(\frac{1 - R_{neighbor}}{1 - R_{current}}\Big)\Big| \tag{5}$$

Here, the log() operator is used to enable a broader search for the global optimum. In the prior example, the difference Δ between the two reliability scores is now $\Delta = |\log(\frac{1-0.99}{1-0.999})| = \log(10) > 0.009$. According to Equation 4, a bigger Δ means a lower chance of accepting a less reliable neighboring deployment plan.

Setting t. t represents the annealing temperature during the search for a reliable deployment plan. Specifically, we set t to:

$$t = \frac{T_{max} - T_{elapsed}}{T_{max}} \tag{6}$$

Here, T_{max} denotes the maximum search time before the search for a reliable deployment plan gets terminated and is specified by the developer. $T_{elapsed}$ denotes the search time elapsed since the beginning of the search. As $T_{elapsed}$ increases during the search, the annealing temperature t gradually decreases.

At the beginning of the search, the annealing temperature is higher, leading to a higher chance of accepting a less reliable neighboring plan, so that a more extensive search for the most reliable deployment plan can be conducted. Towards the end of the search, the annealing temperature gets lower, producing a lower chance of accepting a less reliable deployment plan.

3.3.3 Multi-Objective Optimization

Optimizing only reliability during the search would result in a deployment plan where all application instances or application components are placed distant from each other. In reality, however, some application components may need to be co-located as they frequently interact with each other, and application deployments may need to adapt to the changing conditions of infrastructure. Therefore, high reliability is only one objective when deploying a cloud application. Other important objectives include: 1) application performance for developers, and 2) resource utilization for cloud providers. The ability to combine multiple objectives becomes important in practice.

Rather than considering only the reliability score to search for a reliable deployment plan, reCloud can generate a holistic measure M by combining: 1) the reliability score of a deployment plan, and 2)

the utility score of the deployment plan produced by the complimentary techniques optimizing other objectives (e.g., [5, 10, 13, 39, 46]):

$$M = a \times reliability + b \times utility \quad (7)$$

Two examples of the utility score are the bandwidth usage across the hosts used in a deployment plan, and the resource utilization of such hosts. How to weigh the reliability and utility scores (i.e., a and b in Equation 7) depends on the application domain. For example, we treat them equally in the evaluation (see §4.2.2). We note that the system resource utilization may vary during the search process. However, as shown in §4, reCloud's high efficiency enables it to quickly adapt to varying conditions collected at (near) real-time.

As a result, during the reliable deployment search, instead of using only the reliability score, reCloud uses this holistic measure to evolve neighboring deployment plans and determine whether to accept them; otherwise, reCloud terminates if the application developer's holistic requirements cannot be fulfilled when the maximum search time has elapsed. During this search process, reCloud can also quickly discard any generated deployment plans that do not satisfy resource constraints, if any. Altogether, reCloud incorporates additional objectives while searching for a reliable deployment plan, enabling both the application developers and the cloud providers to make informed decisions.

3.4 Limited Dependency Information

As described in §2.1, cloud providers can use cloud management platforms or specialized tools to acquire the dependency information and the failure probabilities of infrastructure components (e.g., hardware, software and network components). Such dependency information, however, is not always available. reCloud works with limited dependency information (e.g., only network dependencies), but provides a more complete reliability assessment if more dependency information is available.

Similarly, the failure probabilities of some components (e.g., software components) are not always available. reCloud works with limited or even no failure probabilities by assigning each component a failure probability (e.g., a default value, or a value decided by an analytic hierarchy process [65]). In doing so, even with no failure probabilities, reCloud still provides the critical feature to find a deployment plan to avoid shared dependencies (but without a quantitative assessment).

4 EVALUATION

We implemented the complete reCloud system with all the functionality described in §3. In total, our implementation, including a distributed execution engine, consists of 5.3K lines of Java code. We omit the details due to space limit.

4.1 Evaluation Setup

Infrastructure. While reCloud is general and works with any data center topologies (see §3.1 and §3.2), we notice that the fat-tree-alike topologies have been widely used in real-world data centers [69]. Therefore, we generate four fat-tree topologies as representative examples to demonstrate today's data centers from tiny scale to large scale [19], and run reCloud on these topologies. Table 2 details these data center topologies. As described in §3.1, we use Google's approach [69] to manage a data center's external connectivity. In addition, we add 5 power supplies into each data center as additional dependencies, and assign a power supply in round-robin to each switch, as well as the group of hosts under each edge switch, to maximize the power diversity. These power supplies may produce correlated failures.

Table 2: Data center topologies with external connectivity.

	Tiny	Small	Medium	Large
# ports per switch	8	16	24	48
# core switches	16	64	144	576
# agg switches	28	120	276	1,128
# edge switches	28	120	276	1,128
# border switches	4	8	12	24
# hosts	112	960	3,312	27,072
# power supplies	5	5	5	5

Note that, we take power supplies as a representative example of the additional dependencies. Other types of dependency information (e.g., software dependencies) can be integrated in the same way as power supplies whenever available (see §3.2.3). reCloud does not necessarily require all types of dependency information in order to provide reliable deployment plans (see §3.4).

Failure Probabilities. Various infrastructure components may fail with different probabilities. According to the measurements from real-world systems [28, 32, 52, 61, 67, 72, 73], we apply a realistic setting where each switch fails with a probability following the normal distribution $N(0.008, 0.001)$, and every other component (including power supplies) fails with a probability following the normal distribution $N(0.01, 0.001)$. All failure probabilities are rounded to 4 decimal places. Note that, as discussed in §3.4, reCloud can work with limited or even no failure probabilities, while still providing the critical feature to find deployment plans that avoid common dependencies. The above settings are used to demonstrate the *worst-case* performance of reCloud: without failure probabilities, reCloud's efficiency will only improve because of even faster dagger sampling and network symmetry checking.

Default Settings. We assume that a developer by default requests reCloud to deploy 5 application instances in a large data center (see Table 2), and requires at least 4 of them to be alive (i.e., 4-of-5 redundancy). In addition, by default, the maximum amount of time T_{max} for searching a reliable deployment plan is 30 seconds, and reCloud runs 10^4 "route-and-check" rounds to assess the reliability of each generated deployment plan. The desired reliability score $R_{desired}$ is set to 1.0; therefore, it cannot be satisfied and the search for a reliable deployment plan always terminates at T_{max}. In each experiment, we specify which settings are different from the default, while the remainder stays the same.

Testbed. The servers used in our evaluation are equipped with Intel Xeon quad-core 2.26GHz and 48GB memory.

4.2 Evaluation Results

4.2.1 reCloud vs. INDaaS

To assess the reliability of each generated deployment plan, we need to first produce the failure states for infrastructure components

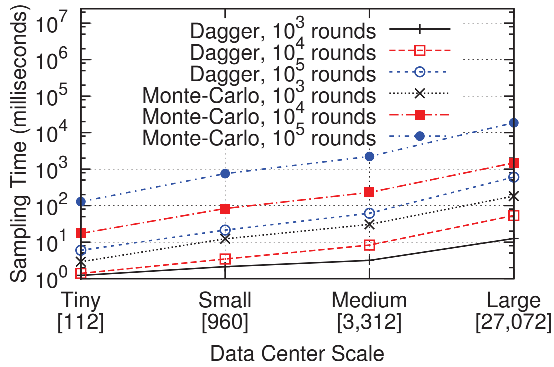


Figure 7: Dagger sampling vs. Monto-Carlo sampling.

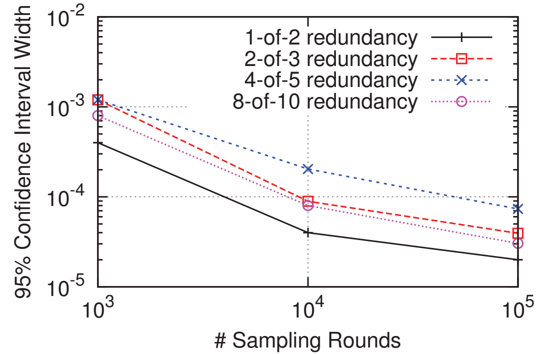


Figure 8: Accuracy of deployment assessment.

across many rounds, and then perform the "route-and-check" process (see §3.2). We evaluated two sampling techniques: Monte-Carlo sampling (used by the state-of-the-art INDaaS system [80]) and extended dagger sampling. Figure 7 shows that dagger sampling is much more efficient than Monte-Carlo sampling across all data center scales, and the benefit of dagger sampling gets larger with the increase of data center scale. For example, in a large data center, dagger sampling is typically more than one order of magnitude faster than Monte-Carlo sampling (e.g., to perform 10^4 sampling rounds, dagger sampling needs only 53 ms, while Monte-Carlo sampling needs 1,487 ms).

More importantly, dagger sampling has a variance reduction effect and converges faster than the Monte-Carlo sampling [45]. That means, with the same number of rounds, dagger sampling-based approach produces more accurate and stable assessment results than the Monte-Carlo-based approach.

Note that, during the reliable deployment search (see §3.3), an individual multi-round sampling is required to assess *each* of the many generated deployment plans. Therefore, compared with dagger sampling, the Monte-Carlo sampling incurs a significant *cumulative* overhead for the reliable deployment search over these many generated deployment plans (e.g., a few hundreds or thousands).

Note also that, INDaaS applies the Monte-Carlo sampling to compare the reliability of given deployment plans without any capability of finding these deployment plans in the first place. Even if we could integrate reCloud's mechanism of searching for a reliable deployment plan (see §3.3) into INDaaS, Figure 7 implies that such an enhanced INDaaS would still be one order of magnitude slower than reCloud.

Accuracy. reCloud not only assesses each generated deployment plan, it also gives rigorous error bounds for the deployment assessment (see §3.2.2). These error bounds indicate a range within which the reCloud-produced reliability score differs from the ground-truth reliability score. Figure 8 shows that, with different K-of-N redundancy settings, the 95% confidence interval width of reCloud's deployment assessment always decreases with the increased number of sampling rounds. Figure 8 indicates that performing 10^4 sampling rounds to assess each deployment plan leads to a 95% confidence interval width at around 10^{-4}, which is normally sufficient for an accurate deployment assessment.

Recall that, the INDaaS system does not quantitatively assess a deployment plan, and does not give error bounds to understand the accuracy. Even if we could make substantial changes to the INDaaS design to support quantitative assessment, our reCloud system is guaranteed to achieve the same level of assessment accuracy but with fewer sampling rounds than such an enhanced INDaaS. This is simply because dagger sampling used in reCloud has a variance reduction effect [45] compared with the Monte-Carlo sampling used in INDaaS.

4.2.2 reCloud vs. Common Practice (with Multi-Objectives)

To our knowledge, there has been little prior work which could search for a cloud application's reliable deployment plan in a quantitative and systematic fashion. Nevertheless, one common practice for reliability is to deploy application instances onto the least-loaded hosts where each host is in a different rack.[2] It, however, lacks the capability of systematically searching for a reliable deployment plan to avoid correlated failures produced by additional common dependencies, e.g., the 5 added power supplies (see §4.1). If we were to make changes to the common practice to avoid common dependencies, this would end up with a system utilizing various heuristics (e.g., no shared edge/aggregation switches, no shared power supplies, and many others) to search for a deployment plan in a huge space. In this experiment, we choose to compare reCloud with an *enhanced* common practice where we run the vanilla common practice 5 times to generate the top-5 non-repeating deployment plans and then pick the plan with the most diversified power supplies.

For comparison, we enable reCloud's *multi-objective optimization* function (see §3.3.3). Specifically, instead of considering only the reliability score to search for a deployment plan, reCloud uses a holistic score combining two factors: 1) the reliability score of the considered deployment plan, and 2) the average workload of the hosts used in this deployment plan. We give the two factors equal weights (see Equation 7).

In practice, a data center's resource utilization (e.g., bandwidth usage) is typically low [12, 64]. To reflect this, we apply a realistic setting where each host has a workload over [0, 1] with the normal distribution $N(0.2, 0.05)$ [12, 64]. We note that workload may vary during reCloud's search process, especially among peak hours.

[2] This common practice was learned from our cloud operator contacts.

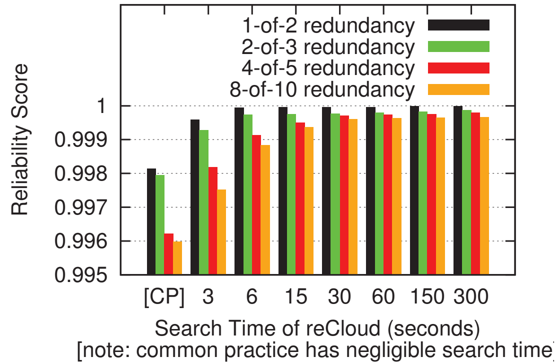


Figure 9: reCloud vs. Enhanced common practice (CP).

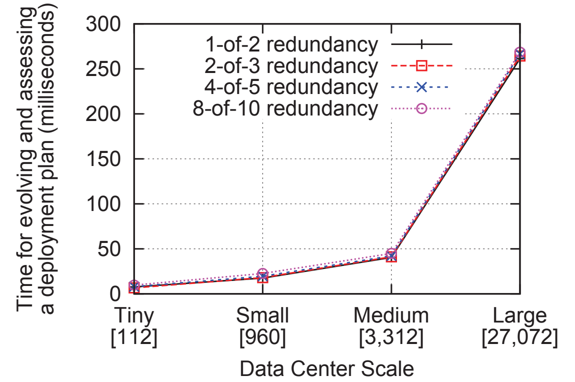


Figure 10: Different redundancy settings.

However, reCloud's efficiency in assessing a deployment plan (see §4.2.3) enables it to quickly adapt to varying workload collected at (near) real-time.

Figure 9 shows that, with different K-of-N redundancy settings, reCloud can always find a deployment plan that is one order of magnitude more reliable than the enhanced common practice (CP). For example, to achieve a 4-of-5 redundancy, the enhanced common practice can find a deployment plan with 99.62% reliability (i.e., 33.3 hours downtime per year), while reCloud can find a deployment plan with 99.97% reliability (i.e., 2.6 hours downtime per year). That said, the deployment plan found by reCloud ensures that, with 99.97% probability, at least 4 out of the 5 deployed instances are alive. Note that, while it is extremely hard, if not impossible, to get the ground-truth reliability of a deployment plan, the accuracy of our reliability assessment ensures that the reliability score produced by reCloud is within very small, rigorous error bounds from the ground truth (see §4.2.1 and Figure 8).

Figure 9 also shows that reCloud can find a reliable deployment plan efficiently even in a large data center, typically within 30 seconds (with 4-of-5 redundancy, this corresponds to checking around 438 generated deployment plans including the ones quickly discarded by the network transformations technique due to network symmetry). This high efficiency of 30-second search time potentially enables reCloud to periodically recalculate the deployment of an existing application to adapt to varying system conditions during service time. In addition, Figure 9 shows that different redundancy deployments achieve different reliability. For example, the 2-of-3 redundancy is more reliable than the 4-of-5 redundancy. The reason is simple: it is harder to achieve the 4-of-5 redundancy which requires at least 4 out of the 5 instances to be alive. Note that, the 4-of-5 redundancy can process more workload than the 2-of-3 redundancy to meet certain business needs.

4.2.3 Application Structures

reCloud evolves and assesses a series of deployment plans to find a reliable plan. Figure 10 shows that, even in a large data center and without the help of the network transformations technique, reCloud can evolve and assess each deployment plan within 270 ms for a single-layer application (i.e., with no cross-layer connectivity). This efficiency enables reCloud to quickly explore many generated deployment plans to search for a reliable plan. Figure 10 also shows that different K-of-N redundancy settings have little impact on the performance. This is because running the routing protocol to check the reliability of any K-of-N deployment plan is very efficient, and the time-consuming part is the context setup for the "route-and-check" process in each round. This indicates that reCloud has the capability to efficiently support highly-redundant application deployments with large K and N values.

Next, we evaluate reCloud for a multi-layer application, whereby the application developer specifies that each layer consists of one application component with 4-of-5 redundancy and the alive component instances in one layer are required to reach the component instances in the next layer. Figure 11 shows that, the number of layers also has little impact on the performance. Similarly, this is because running the routing protocol to check the reachability between any two layers is very efficient, and the time-consuming context setup for the "route-and-check" process in each round needs to be done only once regardless of the number of layers. This enables reCloud to efficiently search for reliable deployment plans for a multi-layer cloud application.

Finally, we evaluate reCloud for a microservices-based cloud application, with 4-of-5 redundancy for each of its components. Suppose this application has X core components which are fully meshed, and in addition, each core component communicates with its respective Y supporting components, denoted as an "X-Y" structure. Figure 11 shows that, reCloud can deal with microservice applications efficiently. Even for an application with a "10-20" structure (i.e., 210 components in total) and without the help of network transformations, reCloud can evolve and assess each deployment plan within 1 second even in a large data center.

4.2.4 Parallel Execution

While reCloud's deployment assessment with 10^4 rounds is quite accurate (see §4.2.1), some application developers may want even higher accuracy, requiring reCloud to run more rounds for each deployment assessment. This further affects the search time of finding a reliable deployment plan. Recall from §3.2 that, running a routing protocol for many rounds to check the reliability of each generated deployment plan can be parallelized via MapReduce. Figure 12 shows that parallel execution enables the deployment assessment

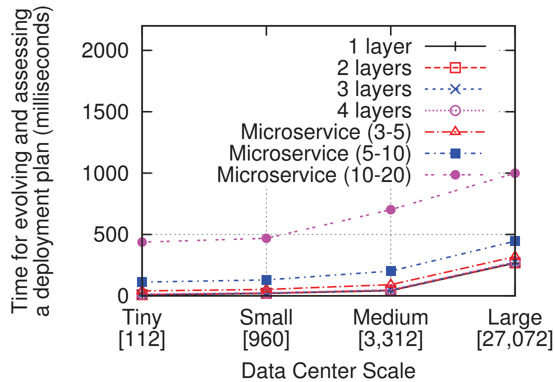


Figure 11: Complex application structures.

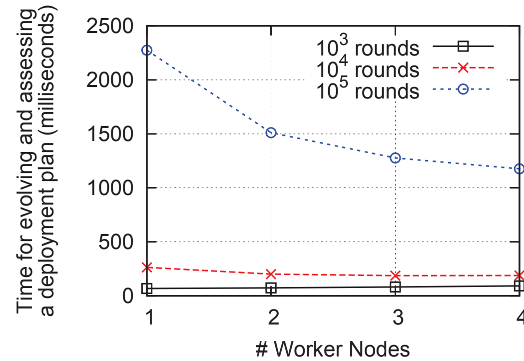


Figure 12: Parallel execution.

with a large number of rounds (e.g., 10^5 rounds), without affecting the performance substantially. Note that, with a smaller number of rounds, the data serialization/transmission/deserialization and the context setup take significant time compared with the efficient "route-and-check" process, thus reducing the gains from parallel execution. Therefore, parallel execution is only beneficial when an extremely high assessment accuracy is required.

5 RELATED WORK

INDaaS [80] is the closest system to reCloud. It compares the reliability of given deployment plans for a cloud application, and selects a reliable plan for deployment. However, it does not quantitatively assess the reliability of a deployment plan, cannot search for deployment plans to fulfill the developers' requirements, cannot consider an application's internal structures, and scales poorly in a large cloud infrastructure. Afterwards, Zhai et al. [81] focus on software dependencies and rank service providers according to their security vulnerabilities, but still cannot search for a reliable deployment plan. On the other hand, recent proposals [5, 13] could find a deployment plan balancing fault tolerance and bandwidth usage; however, they target "worst-case survival" or alike rather than directly model the actual quantitative reliability of a deployment plan, and do not consider applications' internal structures.

NSDMiner [59] and Orion [17] discover network dependencies among infrastructure components in enterprise networks using traffic analysis, so that these dependencies can be used for fault localization. Similarly, Sherlock [8] and Sieve [71] use network dependencies and software dependencies, respectively, for root cause analysis. Besides, NetMedic [42] incorporates the application-specific knowledge with network dependencies to diagnose faults, and the accountable virtual machines [35] allow fault detection and isolation. These systems are beneficial after failures occur whereas reCloud takes a proactive approach to find a reliable deployment plan for cloud applications.

CHARM [83] enables developers to dynamically distribute data replicas across multiple cloud providers. The focus is on data availability with reduced cost rather than application deployment. Bonvin et al. [14] propose a fault-tolerant key value store using fine-grained geographical tags for components (e.g., region, center, rack, and host), where components with dissimilar tags are assumed to fail independently. Shared dependencies across these tags, however, are not considered.

NetPilot [78] aims to minimize the effect of disruptions in the network by differentiating failure types to take different actions. NetPilot only considers network component failures whereas reCloud integrates any available information about the cloud infrastructure to assess the reliability of application deployments.

Zhou et al. [84] propose a placement optimization for cloud services that use primary and backup VMs to minimize the consumed network resources during recovery (i.e., when a backup replaces a primary). However, there is no quantitative reliability assessment, and component failures caused by dependencies are not considered.

Recently, Sedaghat et al. propose DieHard [68] to compute the approximate reliability of a service or a job in a data center, and then to schedule the job with reliability constraints; however, it focuses on only power outages and network component failures, its failure domains used for job scheduling are non-trivial to identify in the presence of common dependencies, and it does not consider internal structures of applications.

6 CONCLUSIONS

This paper presented reCloud, the first system which utilizes any available dependency information (e.g., hardware, software and/or network dependencies) and considers correlated failures to perform quantitative reliability assessment for cloud applications with rigorous error bounds, and it efficiently finds reliable deployment plans for applications that even have complex internal structures. reCloud can find a deployment plan that balances reliability and other criteria such as application performance and resource utilization. Experimental results based on a complete implementation show that, even in a large cloud infrastructure with more than 27K hosts, reCloud needs only 30 seconds to find a deployment plan that is one order of magnitude more reliable than the common practice. This high efficiency can further enable reCloud to periodically recalculate the deployment of any existing application to adapt to varying system conditions during service time.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers and our shepherd, Idilio Drago, for their insightful comments.

REFERENCES

[1] Hussam Abu-Libdeh, Paolo Costa, Antony I. T. Rowstron, Greg O'Shea, and Austin Donnelly. 2010. Symbiotic routing in future data centers. In *SIGCOMM*. 51–62.

[2] Marcos Kawazoe Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance debugging for distributed systems of black boxes. In *SOSP*. 74–89.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *SIGCOMM*. 63–74.

[4] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*. 281–296.

[5] Hyame Assem Alameddine, Sara Ayoubi, and Chadi Assi. 2017. An Efficient Survivable Design With Bandwidth Guarantees for Multi-Tenant Cloud Networks. *IEEE Trans. Network and Service Management* 14, 2 (2017), 357–372.

[6] Amazon. 2012. AWS Service Event Report. (2012). https://aws.amazon.com/message/680342/

[7] Amazon. 2017. AWS Service Health Dashboard. (2017). http://status.aws.amazon.com/

[8] Paramvir Bahl, Ranveer Chandra, Albert G. Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. 2007. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM*. 13–24.

[9] Michael O Ball. 1986. Computational complexity of network reliability analysis: An overview. *IEEE Transactions on Reliability* 35, 3 (1986), 230–239.

[10] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. 2013. Chatty Tenants and the Cloud Network Sharing Problem. In *NSDI*. 171–184.

[11] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*. 259–272.

[12] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network traffic characteristics of data centers in the wild. In *IMC*. 267–280.

[13] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A. Maltz, and Ion Stoica. 2012. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*. 431–442.

[14] Nicolas Bonvin, Thanasis G. Papaioannou, and Karl Aberer. 2010. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *SoCC*. 205–216.

[15] Vladimír Černỳ. 1985. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications* 45, 1 (1985), 41–51.

[16] Mike Y. Chen, Anthony Accardi, Emre Kiciman, David A. Patterson, Armando Fox, and Eric A. Brewer. 2004. Path-Based Failure and Evolution Management. In *NSDI*. 309–322.

[17] Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. 2008. Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions. In *OSDI*. 117–130.

[18] Cisco. 2017. Cisco Cloud and Systems Management. (2017). http://www.cisco.com/c/en/us/products/cloud-systems-management/index.html

[19] Jack Clark. 2014. 5 Numbers That Illustrate the Mind-Bending Size of Amazon's Cloud. (2014). http://www.bloomberg.com/news/2014-11-14/5-numbers-that-illustrate-the-mind-bending-size-of-amazon-s-cloud.html

[20] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*. 137–150.

[21] Debian. 2017. Recursively lists package dependencies. (2017). https://packages.debian.org/jessie/apt-rdepends

[22] Draios. 2017. Sysdig. (2017). https://www.sysdig.org/

[23] John Dunagan, Nicholas J. A. Harvey, Michael B. Jones, Dejan Kostic, Marvin Theimer, and Alec Wolman. 2004. FUSE: Lightweight Guaranteed Distributed Failure Notification. In *OSDI*. 151–166.

[24] Embotics. 2017. Embotics Cloud Management Software. (2017). http://www.embotics.com/

[25] FIRST. 2017. Common Vulnerability Scoring System. (2017). https://www.first.org/cvss

[26] Bryan Ford. 2012. Icebergs in the Clouds: The Other Risks of Cloud Computing. In *HotCloud*.

[27] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in Globally Distributed Storage Systems. In *OSDI*. 61–74.

[28] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM*. 350–361.

[29] GitHub. 2016. GitHub Service Outage Report. (2016). https://github.com/blog/2101-update-on-1-28-service-outage

[30] Vincent Granville, Mirko Krivánek, and Jean-Paul Rasson. 1994. Simulated Annealing: A Proof of Convergence. *IEEE Trans. Pattern Anal. Mach. Intell.* 16, 6 (1994), 652–656.

[31] Albert G. Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: a scalable and flexible data center network. In *SIGCOMM*. 51–62.

[32] Haryadi S. Gunawi, Agung Laksono, Riza O. Suminto, Mingzhe Hao, Jeffry Adityatama, Kurnia J. Eliazar, and Anang D. Satria. 2016. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *SoCC*.

[33] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. 2009. BCube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM*. 63–74.

[34] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. 2008. Dcell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM*. 75–86.

[35] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. 2010. Accountable Virtual Machines. In *OSDI*. 119–134.

[36] HardwareLister. 2017. Hardware Lister. (2017). http://www.ezix.org/project/wiki/HardwareLiSter

[37] Keqiang He, Alexis Fisher, Liang Wang, Aaron Gember, Aditya Akella, and Thomas Ristenpart. 2013. Next stop, the cloud: understanding modern web service deployment in EC2 and azure. In *IMC*. 177–190.

[38] Heqing Huang, Su Zhang, Xinming Ou, Atul Prakash, and Karem A. Sakallah. 2011. Distilling critical attack graph surface iteratively through minimum-cost SAT solving. In *ACSAC*. 31–40.

[39] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable Message Latency in the Cloud. In *SIGCOMM*. 435–448.

[40] Nikolai Joukov, Vasily Tarasov, Joel Ossher, Birgit Pfitzmann, Sergej Chicherin, Marco Pistoia, and Takaaki Tateishi. 2011. Static discovery and remediation of code-embedded resource dependencies. In *IM*. 233–240.

[41] Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseur. 2005. Shrink: a tool for failure diagnosis in IP networks. In *MineNet*. 173–178.

[42] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. 2009. Detailed diagnosis in enterprise networks. In *SIGCOMM*. 243–254.

[43] Scott Kirkpatrick, C. D. Gelatt, and Mario P Vecchi. 1983. Optimization by simulated annealing. *Science* 220, 4598 (1983), 671–680.

[44] Ramana Rao Kompella, Jennifer Yates, Albert G. Greenberg, and Alex C. Snoeren. 2005. IP Fault Localization Via Risk Modeling. In *NSDI*.

[45] Hiromitsu Kumamoto, Kazuo Tanaka, Koichi Inoue, and Ernest J Henley. 1980. Dagger-sampling Monte Carlo for system unavailability evaluation. *IEEE Transactions on Reliability* 29, 2 (1980), 122–125.

[46] Sangmin Lee, Rina Panigrahy, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. 2011. Validating heuristics for virtual machines consolidation. *Microsoft Research, MSR-TR-2011-9* (2011), 1–14.

[47] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos Kawazoe Aguilera, and Michael Walfish. 2011. Detecting failures in distributed systems with the Falcon spy network. In *SOSP*. 279–294.

[48] James Lewis and Martin Fowler. 2014. Microservices. (2014). https://martinfowler.com/articles/microservices.html

[49] Dan Li, Chuanxiong Guo, Haitao Wu, Kun Tan, and Songwu Lu. 2009. FiConn: Using Backup Port for Server Interconnection in Data Centers. In *INFOCOM*. 2276–2285.

[50] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas E. Anderson. 2013. F10: A Fault-Tolerant Engineered Network. In *NSDI*. 399–412.

[51] Microsoft. 2017. Azure status history. (2017). https://azure.microsoft.com/en-us/status/history/

[52] Rich Miller. 2008. Failure Rates in Google Data Centers. (2008). http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers/

[53] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. 2009. PortLand: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*. 39–50.

[54] Arun Natarajan, Peng Ning, Yao Liu, Sushil Jajodia, and Steve E. Hutchinson. 2012. NSDMiner: Automated discovery of Network Service Dependencies. In *INFOCOM*. 2507–2515.

[55] Jordan Novet. 2017. Microsoft confirms Azure storage issues around the world. (2017). https://venturebeat.com/2017/03/15/microsoft-confirms-azure-storage-issues-around-the-world/

[56] NSDMiner. 2017. NSDMiner. (2017). https://sourceforge.net/projects/nsdminer/

[57] OpenNebula. 2017. OpenNebula. (2017). http://opennebula.org/

[58] Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. 2006. A scalable approach to attack graph generation. In *CCS*. 336–345.

[59] Barry W. Peddycord III, Peng Ning, and Sushil Jajodia. 2012. On the Accurate Identification of Network Service Dependencies in Distributed Systems. In *LISA*. 181–194.

[60] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. 2016. Scaling network verification using symmetry and surgery. In *POPL*. 69–83.

[61] Rahul Potharaju and Navendu Jain. 2013. When the network crumbles: an empirical study of cloud network failures and their impact on services. In *SoCC*.

[62] C. V. Ramamoorthy, Gary S. Ho, and Yih-Wu Han. 1977. Fault tree analysis of computer systems. In *AFIPS National Computer Conference*. 13–17.

[63] Mario Rios, Keith Bell, Daniel Kirschen, and Ron Allan. 1999. Computation of the value of security. *Manchester Centre for Electrical Energy* (1999).

[64] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *SIGCOMM*. 123–137.

[65] Thomas L Saaty. 2000. *Fundamentals of decision making and priority theory with the analytic hierarchy process*. Vol. 6. Rws Publications.

[66] Bianca Schroeder and Garth A. Gibson. 2007. Disk Failures in the Real World: What Does an MTTF of 1, 000, 000 Hours Mean to You?. In *FAST*. 1–16.

[67] Bianca Schroeder and Garth A. Gibson. 2010. A Large-Scale Study of Failures in High-Performance Computing Systems. *IEEE Trans. Dependable Sec. Comput.* 7, 4 (2010), 337–351.

[68] Mina Sedaghat, Eddie Wadbro, John Wilkes, Sara de Luna, Oleg Seleznjev, and Erik Elmroth. 2016. DieHard: Reliable Scheduling to Survive Correlated Failures in Cloud Data Centers. In *CCGrid*. 52–59.

[69] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*. 183–197.

[70] Ankit Singla, Chi-Yao Hong, Lucian Popa, and Philip Brighten Godfrey. 2012. Jellyfish: Networking Data Centers Randomly. In *NSDI*. 225–238.

[71] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. 2017. Sieve: Actionable Insights from Monitored Metrics in Distributed Systems. In *Middleware*.

[72] Daniel Turner, Kirill Levchenko, Alex C. Snoeren, and Stefan Savage. 2010. California fault lines: understanding the causes and impact of network failures. In *SIGCOMM*. 315–326.

[73] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. 2010. Characterizing cloud computing hardware reliability. In *SoCC*. 193–204.

[74] VMware. 2017. VMware Cloud Management Platform. (2017). https://www.vmware.com/virtualization/cloud-management

[75] Wikipedia. 2017. 68-95-99.7 Rule. (2017). https://en.wikipedia.org/wiki/68-95-99.7_rule

[76] Wikipedia. 2017. Central Limit Theorem. (2017). https://en.wikipedia.org/wiki/Central_limit_theorem

[77] Haitao Wu, Guohan Lu, Dan Li, Chuanxiong Guo, and Yongguang Zhang. 2009. MDCube: a high performance network structure for modular data center interconnection. In *CoNEXT*. 25–36.

[78] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. 2012. NetPilot: automating datacenter network failure mitigation. In *SIGCOMM*. 419–430.

[79] Jimmy Yang and Feng-Bin Sun. 1999. A comprehensive review of hard-disk drive reliability. In *Reliability and Maintainability Symposium*. 403–409.

[80] Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. 2014. Heading Off Correlated Failures through Independence-as-a-Service. In *OSDI*. 317–334.

[81] Ennan Zhai, Liang Gu, and Yumei Hai. 2015. A Risk-Evaluation Assisted System for Service Selection. In *ICWS*. 671–678.

[82] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*. 765–784.

[83] Quanlu Zhang, Shenglong Li, Zhenhua Li, Yuanjian Xing, Zhi Yang, and Yafei Dai. 2015. CHARM: A Cost-Efficient Multi-Cloud Data Hosting Scheme with High Availability. *IEEE Trans. Cloud Computing* 3, 3 (2015), 372–386.

[84] Ao Zhou, Shangguang Wang, Bo Cheng, Zibin Zheng, Fangchun Yang, Rong Chang, Michael Lyu, and Rajkumar Buyya. 2016. Cloud Service Reliability Enhancement via Virtual Machine Placement Optimization. *IEEE Transactions on Services Computing* (2016).