Dependability Analysis on OpenStack IaaS Cloud: 
Bug Analysis and Fault Injection

YUAN Xiaoyong, LI Ying*, WU Zhonghai
School of Software and Microelectronics
Peking University, Beijing, China

LIU Tiancheng
IBM Research - China
Beijing, China

Abstract—This paper proposes a comparative study of cloud
dependability between two methods – bug analysis and fault
injection for assessing the impact of component failure on cloud
service availability. We focus on the IaaS cloud with open source
platform OpenStack. The actual bug data are analyzed to show
numerical examples of dependability assessment. A fault
injection tool has also been developed to create failures of
components and then observe their effects on services. The
comparison analysis between two methods shows that bug
analysis method has richer features for analyzing but not as
precise as fault injection.

Keywords:—dependability, bug analysis, fault injection, cloud
computing, OpenStack

I. INTRODUCTION

Infrastructure-as-a-Service (IaaS) cloud service is a novel
paradigm for providing computation resources to users and
growing up to take the place of traditional data centers on an
ever-increasing scale for its pay-as-you-go manner [1].
Organizations and companies choose to run their businesses on
IaaS cloud platforms for cost saving and elastic deployment.
Dependability guarantee is an essential capability of cloud
platform because cloud provider must offer assurance
regarding availability, reliability and security at a reasonable
cost.

A dependable cloud platform becomes more difficult to
achieve because it provides large scale delivery of services.
Availability and reliability studies may be performed through
fault injection, which is creating failure situations and then
observing their effects on services. This method helps to
evaluate the dependability of a cloud platform and to provide
guidance for designing deployment architecture or improving
maintenance policies. The fault injection method depends on
how failure situations are designed, which may not reflect the
real cases. In this paper, we provide a comparative study of
dependability of cloud platform with two methods: bug
analysis and fault injection. Besides developing a fault
injection tool to support availability and reliability studies in
the open source cloud platform OpenStack, we analyze the bug
data reported at OpenStack.org [2] with the aim to
understanding different types of failures and their impact on
cloud services with real data. OpenStack is a well-known open
source platform to build IaaS clouds. OpenStack software
components may be combined in a distinct architecture, whose
dependability may be impacted by single or multiple failures.

This paper studies nearly 8300 bug reports during 4 years and
explores the details of 928 relevant bug reports implying the
impact of component failure on service availability.

Our analysis offers interesting insights into OpenStack
dependability. We find that Nova is the most influential one
among observed components of OpenStack. Failures of Nova
cause most of function unavailability in bug reports. Keystone
is another essential component that none of functions can pass
tests when Keystone is injected with faults. Though impacts of
bugs are not severe on average, the severe level upgrades with
increasing of the number of components involved in the
function.

In this paper, our contributions include: 1) analyzing bug
reports of cloud platform and focusing on the dimension of
function and component; 2) providing the tool to inject faults
into OpenStack subcomponents and identifying unavailable
service from 106 basic functions; 3) comparing two methods in
different aspects and analyzing their advantages and
disadvantages in usage.

The rest of this paper is organized as follows. Section 2
discusses related work. Section 3 presents a dependability
model. Section 4 and 5 describe our work on bug analysis and
fault injection on OpenStack. In Section 6, we compare these
two methods. Finally, Section 7 concludes our work in this
paper.

II. RELATED WORK

A. Bug Analysis

Many efforts have been worked to study bug characteristics
of open-source softwares.

Tan et al. [11] and [14] analyze bug characteristics of open-
source software Linux, Apache Web Server and Mozilla. They
focus on three dimensions: root cause, impacts and
components, as well as the statistical correlation between these
dimensions. In their summary, the semantic bugs are the
dominant root cause in the current trend. Moreover, most
semantic bugs that result in incorrect functionality are also one
of important factors of causing unavailability. To reduce
manual effort, they use machine learning techniques to classify
bugs automatically. Compared with their work, we focus on
bug characteristics of incorrect functionalities, and study their
causes and influences on dependability.

*Corresponding author
The project is supported by National Natural Science Foundation of China
(No. 61232005). The work is also supported by IBM UR Program.
Rahmani et al. [19] analyze bug reports in 5 open-source projects for software reliability. They count the number of bugs in each project and build a software reliability model with Weibull distribution based on bug quantity. They find future releases of the projects follow the similar reliability patterns. We suggest that bug impact and its relationship with component can also help analyze software reliability.

Bug quality is essential for our bug analysis. Bettenburg et al. [10] summarize bug characteristics for a good report. They have surveyed developers and users of Apache, Eclipse, and Mozilla. Steps to re-produce, stack traces, and test cases are considered important for developers. We leverage their findings to assess quality of bug reports for OpenStack.

B. Fault Injection

Fault injection is commonly used to evaluate and characterize system dependability features. Many researchers have presented their fault injection methods for cloud systems.

Le et al. [20] present software implemented fault injection (SWIFI) in virtualized systems. They inject faults into virtual machine (VM) and virtual machine monitor (VMM), and compare results among non-virtualized, paravirtualized and fully-virtualized systems. Challenges for fault injection in virtualized systems are concluded.

Ju et al. [21] build a prototype fault-injection framework targeting service communications during the processing of external requests, both among OpenStack components and between OpenStack and third-party software. Comparing with their work, we take a black-box approach in fault injection study. Though a black-box approach lacks an in-depth analysis for semantics of OpenStack, it is easy to extend test coverage for hundreds of functions in OpenStack. This helps us know a whole picture of cloud service dependability.

Benz and Bohnert [17] present a dependability modeling framework for test high availability capability of cloud. They use fault injection to crash cloud component, count its downtime and model the expected impact. Different functions are included into a certain use case for the purpose of application for complex scenarios. But relationship between function and component is viewed as priori knowledge, not acquired by fault injection method.

Souza et al. [22] implement a fault injection tool for Eucalyptus cloud platform. They inject faults in Eucalyptus hardware and software components at runtime and uphold reparation in the same tool. The verification of this tool is limited to software process and hardware resources, instead of functionality of cloud service.

III. DEPENDABILITY MODEL

OpenStack is designed to be loosely coupled with components including Nova, Keystone, Cinder, etc. In this case, OpenStack provides flexibility when designing and deploying a dependable cloud. However, because OpenStack is a complex and distributed system, it is hard to have a whole picture of component dependency for dependability. Following the dependability modeling framework approach [23], we define a dependability model of OpenStack. It is a hierarchical modeling framework to assess the impact of component failures on the dependability of services delivered to users, which have three levels: Function Level, Component Level and Subcomponent level, as showed in Fig. 1.

Function Level contains functions run by cloud operators and cloud users, which directly reflects the dependability for a cloud. If any function is unavailable during the operation time, the cloud is assumed to be unavailable. OpenStack provides various public interfaces in Function Level: RESTful API, public CLI and dashboard. Referring to the list of actions by function in Amazon’s EC2 cloud [4], we choose 106 basic functions of OpenStack. As EC2 is a public cloud, most functions published by Amazon are user oriented, meanwhile, OpenStack is a platform built for public, private and hybrid cloud, which means that functions should be used by both cloud operators and cloud users. Thus functions provided by OpenStack should be a superset of those by EC2. Considering OpenStack began in 2010, and EC2 is more mature and has richer features than OpenStack, we eliminate functions that haven’t been implemented by OpenStack. Eventually, we have 106 functions to test on OpenStack.

Component Level contains various components to fulfill different functions. In this paper, our work focuses on 5 core components in OpenStack: Nova (compute service), Neutron (network service), Cinder (block storage service), Keystone (identity service), and Glance (image service). Equipped with database software (e.g. MySQL), messaging server (e.g. RabbitMQ or ZeroMQ), and virtual machine monitor (e.g. KVM or Xen), these core components can be deployed into a basic IaaS cloud.

Subcomponent Level contains detailed subcomponents that run in different components. In the context of OpenStack cloud, different components run various numbers of subcomponents. For example, Nova runs 7 subcomponents, otherwise, Keystone runs only 1 subcomponent.

The relationship between Component Level and Subcomponent Level is clear. For OpenStack Component Level, components are logically different from others. Different components are developed by different groups of developers, loading different configuration files, and deployed in different physical servers. However, Subcomponent Level actually is a bunch of Python files, which completes various missions for its component. The fault in a subcomponent, logically equivalent to the fault in a component, would affect functions and probably cause unavailability of some functions, in other words, downtime of cloud services. However, OpenStack doesn’t provide clear knowledge of the dependency graph between components (or subcomponents) and functions. In the following sections, we will use two approaches: bug
analysis and fault injection to build this dependency graph and help analyze OpenStack dependability.

IV. BUG ANALYSIS

There have been some research efforts on bug characteristics in open-source software wherein component is one of major analysis dimensions [14]. Failed functionality is often considered as one of consequences of component bugs. We aim to explore the relationship between component failure and service availability. For this purpose, we analyze reported bugs of OpenStack and try to answer three questions:

1) Which function is the most vulnerable one?

Functions are the cloud services offered by cloud operators. End Users access OpenStack services and deploy their applications on OpenStack platform through its command line, python-client API, and dashboard. Failures in a function may affect overall availability of the cloud service.

2) Which component failure has the most impact on the availability of cloud service?

OpenStack is composed by various components. Relationship between components and functions is not clear. These components are developed by different people for different purposes. As a result, their complexity, scale, and quality are not same. This heterogeneity affects the dependability of OpenStack cloud that different deployment and high availability configuration of components may lead to different availability.

3) What is the dependency between component failure and function availability?

Dependency between components and functions can help to evaluate OpenStack dependability as discussed in Section 2.

A. Bug retrieval

OpenStack has a complete and formal bug report process. OpenStack Community uses Launchpad.net to track their known issues and defects in the software [7][8]. The bug information contains items including: ID, bug title, detailed description, affect components, status (e.g., New, Incomplete, Confirmed, Triaged, In Progress, Fix Committed, Fix Released, Invalid, Opinion, Won’t Fix), importance (setting after Confirm Stage), milestone (i.e., OpenStack Release) and assignment (the person who is assigned to fix the bug). Table 1 shows examples of bug reports from Launchpad.

According to our analysis purpose, irrelevant bugs are crawled and filtered from Launchpad into database. The bug filtering rules are defined as follows:

1. A bug whose affected components are not included in the set of Nova, Neutron, Cinder, Keystone and Glance;
2. A bug whose stage is before Confirm Stage, which means the bug is unconfirmed;
3. A bug that doesn’t belong to dependability issues, for example, to improve performance, or for enriching functionality, migrating database, and updating changes in the document;
4. A bug that doesn’t present functionality issues;
5. A bug that is related with upgrade problem (i.e. upgrade from Havana release to Icehouse release). Upgrade is a threat to cloud dependability, but it is not involved in this paper.

The bug satisfying any one of above rules will be filtered. First two rules are performed automatically, while the last three rules must be analyzed manually. An open source web crawling framework Scrapy [9] has been used to obtain OpenStack bug data that about 8300 bug reports of 4 years (since the initial stage of OpenStack) have been crawled. After filtering, 928 relevant bug reports have been selected for analysis. They are caused by failures of five components (Nova, Neutron, Cider, Keystone, and Glance) and affect 106 functions. Thus, we classify bugs into two dimensions: component and function.
is divided into components. Different components are marked for example in nova!

The total report's importance to the system availability. Fig. 2(a) shows the number of bug reports of five components. Nova is the most reported component and Keystone is the least one. We consider the scale and complexity for different components, this finding doesn’t change much. We choose thousand lines of code (KLOC) as a direct metric of software scale and complexity, and get components’ KLOC data from [13]. Fig. 2(b) shows that Nova is the most reported component, about 2 bugs are reported in one thousand lines, while other components have round 0.5 bugs per KLOC.

To analyze the impact of components’ failure, we use importance of bugs assigned by reviewers in bug reports. Impact level is divided into critical, high, medium, low and undecided/wishlist. Definition of impact level is described in [8]. Because some bugs affect more than one component, they have more than one impact levels (for example in Table 1, 3rd bug’s id is 1095271). Thus we define impact level as its most severe impact. We sum number of bugs reported in same component with same impact. Fig. 3 shows the proportion of impact for bugs caused by different components’ failures. For all components, impacts of most bugs are not significant. For Fig. 3, the proportion of bugs of low and undecided/wishlist impact is between 39% and 62% among components. Bugs with critical and importance impact are no more than 21%. Neglecting undecided/wishlist, reports of medium impact are the majority of the total reports. Among different components, Nova has the biggest proportion of critical and high importance (21%).

Secondly, we study bug reports from functions’ view. Table 2 illustrates the top 10 functions in terms of the number of reported bugs. Boot instance is listed as the top function according to the number of bug reports, which is coincident to the fact that it is the dominant function in OpenStack currently. So many bugs in this function are inconceivable. Most functions in Table 2 are frequently used in cloud. Considering bug lies in functions used frequently, a judicious guessing for now is that OpenStack still lacks testing, and many latent bugs in infrequent functions haven’t been found yet. Fig. 4 shows the distribution of number of detected bugs for different functions and components. Each point illustrates a combination of function and component. Different components are marked in different colors. Some functions in Nova are more frequently reported than other components.

It can be found in Table 2 that all of these functions except resize instance have bugs in at least two components. The reasons may lie in logical complexity of the function, or OpenStack developing policy. Different components in OpenStack are developed by different groups, led by different groups.
Project Technical Lead (PTL) and discussed in different IRC channels, which may cause some inconsistencies during software development.

To study the relationship between functions and components, we build a table of 107*5 items that describes various functions affected by failures of different components, part of which is showed in Table 5. Meanwhile, we define the function that has bugs in different components as cross-component while others as non-cross-component. For example, boot instance is a cross-component function, while resize instance is a non-cross-component function. Fig. 5(a) shows the proportion of bug reports in different function categories. One component means the function is non-cross-component. Two and more components means the function has bugs in as many components. Fig. 5(b) shows the proportion of functions in different categories. From Fig. 5(a) and Fig. 5(b), 71% bugs belong to cross-component functions, however, 67% functions are non-cross-component. It indicates that more bugs are detected in cross-component functions than non-cross-component functions.

When we bring impact metrics into cross-component functions, surprisingly, the impact level of bugs becomes severe with the increasing number of the involved components. Fig. 6 shows the proportion of functions involved in different components with different impact levels.

V. FAULT INJECTION

In this section, we use another method, fault injection, to evaluate OpenStack cloud dependability as it is a powerful method to analyze the dependability of computer systems. Fault injection tests fault detection, fault isolation, and reconfiguration and recovery capabilities [3]. The fault-error-failure model is fundamental to understand fault injection method. In [16], a fault is active when it causes an error, otherwise it is dormant. A service failure means that at least one (or more) external state of the system deviates from the correct service state. The deviation is an error. For fault injection, we want to identify which service, or function in this context, will deviate from the correct state.

Because our target system is OpenStack cloud, we use SWIFI [18] to evaluate cloud dependability. SWIFI is a powerful method to reproduce fault in software level. It is more flexible to inject and detect fault than other injection methods like hardware implemented fault injection, but limited to its fault coverage.

OpenStack is a complex and loosely coupled software system. It is composed of several components, some of which have several subcomponents. Compared with bug analysis, fault injection can be applied into Subcomponent Level. This is a finer level of granularity, compared with bug analysis. To build a dependability model on OpenStack, we inject faults into Subcomponent Level to simulate cloud failure.

A. Experimental setup

In the experiment, we setup three virtual machines, one for controller and network node, two for compute nodes, as a

<table>
<thead>
<tr>
<th>component function</th>
<th>Nova</th>
<th>Neutron</th>
<th>Cinder</th>
<th>Keystone</th>
<th>Glance</th>
<th>SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>boot instance</td>
<td>142</td>
<td>15</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>156</td>
</tr>
<tr>
<td>delete instance</td>
<td>60</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>66</td>
</tr>
<tr>
<td>attach volume</td>
<td>37</td>
<td>0</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>resize instance</td>
<td>47</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>47</td>
</tr>
<tr>
<td>live migrate instance</td>
<td>44</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>46</td>
</tr>
<tr>
<td>create volume</td>
<td>5</td>
<td>0</td>
<td>37</td>
<td>0</td>
<td>0</td>
<td>41</td>
</tr>
<tr>
<td>migrate instance</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>create image</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>22</td>
<td>41</td>
</tr>
<tr>
<td>detach volume</td>
<td>16</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>create port</td>
<td>3</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>18</td>
</tr>
</tbody>
</table>

reconfiguration and recovery capabilities [3]. The fault-error-failure model is fundamental to understand fault injection method. In [16], a fault is active when it causes an error, otherwise it is dormant. A service failure means that at least one (or more) external state of the system deviates from the correct service state. The deviation is an error. For fault injection, we want to identify which service, or function in this context, will deviate from the correct state.

Because our target system is OpenStack cloud, we use SWIFI [18] to evaluate cloud dependability. SWIFI is a powerful method to reproduce fault in software level. It is more flexible to inject and detect fault than other injection methods like hardware implemented fault injection, but limited to its fault coverage.

OpenStack is a complex and loosely coupled software system. It is composed of several components, some of which have several subcomponents. Compared with bug analysis, fault injection can be applied into Subcomponent Level. This is a finer level of granularity, compared with bug analysis. To build a dependability model on OpenStack, we inject faults into Subcomponent Level to simulate cloud failure.

A. Experimental setup

In the experiment, we setup three virtual machines, one for controller and network node, two for compute nodes, as a
typical OpenStack deployment. The virtual machines are provided by Oracle VirtualBox. Our target system is OpenStack Icehouse release. Nova, Neutron, Cinder, Keystone and Glance are installed into virtual machines. Convenience to restore to the original state from snapshotting is one of the advantages of virtual machines. Fault injection into cloud may bring unperceived harm to the cloud system, it is important to recover the original state quickly and correctly.

B. Fault injection tool

We develop a fault injection tool to evaluate dependability of cloud service. Fig. 7 illustrates an overview on this fault injection tool. **Controller** is designed to plan and control all work flows. Firstly, it loads the test case and relevant information from a XML file through **Loader** and prepares for test. Secondly, **Controller** starts detecting failure through **Detector**. Thirdly, **Controller** injects faults into the target system through **Fault Injector** and generates requested functions through **Request Generator**. After a given duration **Collector** stops testing and convert result into a CSV file through **Reporter**. Finally, **Controller** analyses collected data through **Analyzer**. After a test, **Cleaner** cleans up cloud and rolls back to original environment for the next round of test.

Considering components installed in OpenStack platform, we inject faults into following OpenStack subcomponents to simulate failure: nova-api, nova-conductor, nova-scheduler, neutron-server, nova-cert, nova-consoleauth, nova-novncproxy, neutron-server, neutron-metadata-agent, neutron-dhcp-agent, neutron-l3-agent, neutron-plugin-openswitch-agent, cinder-api, cinder-volume, cinder-scheduler, keystone, glance-api, and glance-registry. To apply to all the subcomponents, we only consider process crashes as injected faults in this paper. After faults are injected into Subcomponent Level, 106 OpenStack functions are executed one by one to verify the availability of cloud services.

C. Experimental result

After overall tests, the relationship between functions and subcomponents is built up similar to that in Section 3. Table 5 presents part of our result: “1” denotes the failure of function occurs after injecting a certain fault. Failure is measured as no response or incorrectness for the execution of function. “0” denotes the function is available during the test.

Fig. 8 shows the number of unavailable functions when injecting faults into different subcomponents. Failure of Keystone, Neutron-server and Nova-api are the top 3 subcomponents that cause unavailability of many functions.

Keystone is the most vulnerable subcomponent in this test. Actually, none of functions has passed the test after Keystone crashes. Keystone is an identity and authentication service, every function should be identified before they get executed. It is reasonable that failure of Keystone would cause unavailability of all functions.

In the experiment, failure of nova-cert, nova-novncproxy, neutron-metadata-agent, neutron-dhcp-agent, and neutron-l3-agent won’t cause service unavailability.

![Figure 9 Function Failure Statistics in Component Level](image)

VI. DISCUSSION

We use bug analysis and fault injection to analyze OpenStack dependability. Both of them have their merits and defects (Table 4).

A. Merit and defect of fault injection method

Compared with bug analysis, fault injection can be used to analyze dependability in a finer granularity. In our experiment, we inject faults into Subcomponent Level and analyze impact of failure of subcomponents on availability of functions. If necessary, we can inject fault into code level, because Python is a dynamic language, so that we can execute many common programming functions at runtime rather than during compilation time.

However, fault injection is limited to fault coverage inherently. It is hard to implement all types of fault and inject them into target system. For example SWIFI can’t inject fault into locations that are inaccessible to software [3]. Identification and characterization of software faults, and techniques to inject faults similar to the real ones are also key challenges for fault injection [18]. For example, when analyze reproducing process in bug reports, we find many bugs occur after a few steps of operations. It seems to be more than one faults and that occurs more than one times. In our fault injection experiment, it is hard to implement faults with such diversity and intricacy.

B. Merit and defect of bug analysis

For bug analysis, correctness is the most concern. To verify the correctness of bug analysis, we compare the result of bug analysis with that of fault injection method. Because data of two methods are not consistent in description, we apply two-step conversion before verification.

First, bug analysis only has data in Component Level, we have to summarize the result of fault injection method and convert it from Subcomponent Level to Component Level. Because the subcomponent is part of its component, failure caused by a subcomponent can be also viewed as caused by the component it belongs to. For example, in Table 5, *boot instance* is assigned as: Nova 1, Neutron 1, Cinder 0, Keystone 1, and Glance 1. Fig. 9 shows the quantity of function failures due to different components after conversion.

Second, we convert the number of bugs to binary representation like fault injection method. Once a bug is reported for a certain function caused by a component, we consider it as unavailability of this function.
The relationship between function and service availability. We conduct an experiment to study the relationship between two methods and find that the fault injection method is more effective in identifying component failures.

We use four metrics: Precision rate ($P = \frac{T_p}{T_p + F_p}$), Recall rate ($R = \frac{T_r}{T_r + F_r}$), F measure ($F = \frac{2PR}{P+R}$), and Accuracy ($A = \frac{T_p + T_r}{T_p + T_r + F_p + F_r}$) to evaluate the correctness.

To compare the specialty of Keystone as we discussed above, we adopt two measurements. If we include Keystone into components, then we get $P = 80.7\%$, $R = 38.4\%$, $F = 52.1\%$, and $A = 71.5\%$. If we count only 4 components except Keystone, then $P = 78.4\%$, $R = 61.8\%$, $F = 69.1\%$, and $A = 85.0\%$. From the correctness measurement, bug analysis is basically correct for all components except Keystone.

What’s more, bug analysis method largely depends on bug quality. A high-quality bug database is critical for bug analysis. Experience and skills owned by bug reporters and reviewers significantly affect bug quality as well.

From our observation, so far the bug reports of OpenStack have quality issues to some extent, including: a) duplicated bugs usually don’t harmful to developers but will have effect on the statistical result; b) most bug reports have steps to reproduce in bug reports, which is the most useful to developers [11], but a small portion of reports are casual and brief; c) some bug reports are lack of milestone information, which cause it is hard to analyze and trace bugs between certain OpenStack releases.

Bug analysis has richer features to retrieve and leverage, compared with fault injection method. In this paper, we use impact of bugs assigned by reviewers to help analyze component failure and service availability. The number of bugs is also a good metric for evaluate the relationship of two dimensions.

### VII. CONCLUSION AND FUTURE WORK

This paper uses bug analysis and fault injection to provide comparative study of dependability of OpenStack cloud service. We firstly select 928 relevant reports from 8300 bug reports in openstack.org by automatically filtering with the predefined filtering rules and manually analyzing the details of these bug reports, classifying them into different affected functions and caused components in order to study the relationship between function availability and component failures. Furthermore, for the same purpose, we apply fault injection method into OpenStack. Eventually, we compare these two methods and find out their merits and defects for analyze cloud dependability. The comparison analysis between two methods shows that bug analysis method has richer features for analyzing but not as precise as fault injection. In our future work, we will include more components of OpenStack and analyze more dimensions of bugs, besides function and component. Automatic methods (e.g., machine learning and information retrieval techniques) will be applied to reduce manual effort and human error.

### REFERENCES


---

Table 5 Result of Fault Injection

<table>
<thead>
<tr>
<th>Component</th>
<th>boot</th>
<th>instance</th>
<th>unshelve</th>
<th>instance</th>
<th>rebuild</th>
<th>instance</th>
<th>rescue</th>
<th>instance</th>
<th>attach</th>
<th>volume</th>
<th>detach</th>
<th>volume</th>
<th>e soft</th>
<th>reboot</th>
<th>instance</th>
<th>rescue</th>
</tr>
</thead>
<tbody>
<tr>
<td>nova-api</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>nova-compute</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>nova-conductor</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>nova-cert</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>nova-novncproxy</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>neutron-server</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>neutron-metadata</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>neutron-dhcp-agent</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>neutron-l3-agent</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>neutron-plugin-openvswitch-agent</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>cinder-api</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>cinder-volume</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>keystone-glance-api</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>glance-registry</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

---
