UDK 004.254 (045)
Maksymovych O.V., Melnyk V.M., Bortnyk K.Ya., Lavrenchuk S.V., Samarchuk V.F.
Lutsk national technical university

PLIABILITY OF FAULTS IN OPEN STACK VERSIONS

Maksymovych O.V., Melnyk V.M., Bortnyk K.YaLavrenchuk S.V., Samarchuk V.F. Pliability of faults in open stack versions. *Stacks of cloud-management have become an important elements in cloud computing. They are serving as the resource managers for cloud platforms. While their functionality has been constantly expanding, their fault resilience remains under-studied. This article presents a fault resilience systematic study of OpenStack – a popular cloud-management stack for the open source. We have built an example fault-injection framework directing service communications during the external requests processing, both among OpenStack services and between OpenStack and external services, and have thus far uncovered 23 bugs in two OpenStack versions. Our findings hut light on defects in the design and implementation of advanced cloud management stacks from a fault-resilience perspective.*
Keywords: *reliability, fault-injection framework, fault-resilience perspective, cloud-management stack, open stack.*

Максимович О.В., Мельник В.М., Бортник К.Я., Лавренчук С.В., Самарчук В.Ф. Стійкість до збоїв версій відкритих стеків. *Стеки управління cloud'ами стали важливим елементом cloud обчислень. Вони слугують менеджерами ресурсів для cloud-платформ. У той час як їхня функціональність постійно розширюється, їхня стійкість до збоїв не до кінця вивчена. Ця стаття презентує системне дослідження стійкості до збоїв Open Stack – популярного стеку керування cloud'ами на базі відкритого коду. Створено приклад структури направляючого комунікаційного сервісу для генерування помилок під час обробки зовнішніх запитів, для роботи як серед сервісів OpenStack, так і для роботи між OpenStack та зовнішніми сервісами, і, таким чином, виявлено на даний момент 23 необроблені помилки у двох версіях OpenStack. Проробленні пошуки просвітлили погляд недоліки в проектуванні і реалізації поширених стеків управління cloud'ами з точки зору перспективи стійкості до збоїв.*
Ключові слова: надійність, структура генерування помилок, перспектива стійкості до збоїв, стек управління cloud'ом, відкритий стек.

Максимович О.В., Мельник В.М., Бортник К.Я., Лавренчук С.В., Самарчук В.Ф. Устойчивость к сбоям версий открытых стеков. Стеки управления cloud'ами стали важным элементом cloud-вычислений. Они служат менеджерами ресурсов для cloud-платформ. В то время как их функциональность постоянно расширяется, их устойчивость к сбоям не до конца изучена. Эта статья представляет системное исследование устойчивости к сбоям Open Stack – популярного стека управления cloud'ом на базе открытого кода. Создан пример структуры направляющего коммуникационного сервиса для генерации ошибок при обработке внешних запросов, для работы как среди сервисов OpenStack, так и для работы между OpenStack и внешними сервисами, и, таким образом, обнаружено на данный момент 23 необработанные ошибки в двух версиях OpenStack. Проделанные поиски сделали просвет на недостатки в проектировании и реализации распространенных стеков управления cloud'ами с точки зрения перспективы устойчивости к сбоям.
Ключевые слова: надежность, структура генерации ошибок, перспектива устойчивости к сбоям, стек управления облаком, открытый стек.

Introduction

With the virtual machine (VM) technology development in both: software design and hardware support, cloud computing has become a major computing platform. In addition to public cloud services that have been available since the early stage of cloud platform deployment [1], there is an evolving demand for other types of cloud platforms, particularly, private and hybrid clouds. The demand leads to the role transition from cloud users to a cloud users and providers combination, broadening the scope of cloud providers from major IT companies to any size organizations. It has also prompted the cloud management stack research and development – a new software stack functioning as a high-level cloud operating system and remains key to resource management in cloud platforms.

In recent years an attention to cloud-management stacks from academy and industry has led to a rapid increase in the number of features. However, fault resilience of this layer, as an optional feature, is still regarded and remains under-studied, despite its importance demonstrated by real-world failures and its significant impact on managed cloud platforms [2, 3, 4]. Fault-resilience-related issues constantly stun the users of cloud-management stacks. For example, when faults occur, VM creation may fail or take some long time, and VMs may be marked as successfully created but lack critical resources (IP addresses), thus remaining unusable. A fault resilience investigation of cloud-management stacks is long overdue that demystifies the above issues.

In this paper, the first systematic study on the fault resilience of OpenStack is present, as a popular open source cloud-management stack. The conventional wisdom in the fault-injection literature as well as its application in fault-injection studies directing large-scale distributed systems [5, 6, 7], that studied the

OpenStack execution in the two common faults presence with the cloud environment: server crashes and network partitions. OpenStack is considered fault-resilient during the external request processing, if it maintains correct and steady states and behaviours, even in case of faults occurrence. As external requests are an important source of inputs to OpenStack and usually trigger state transitions, focusing on OpenStack's fault resilience during external request processing. We insert faults into inter-service communications during request processing, as they characterize service collaboration of which design and implementation is difficult to be fault resilient. Specifically, there are targeted communications among OpenStack's computing, image, identity services, as well as external services such as databases, hypervisors, and messaging.

An approach is taken in this fault-injection study, to expose high-level OpenStack semantics (service A sends a request R to service B via communication channel C) by supplementing its wrapper layer of communication libraries with our sorting and module of coordination. Exposing, instead of inferring, high-level semantics reduces the amount of logs, simplifies the extraction of communication patterns and facilitates efficient fault injection. This approach can also be easily integrated into OpenStack's notification mechanism. It closely mirrors OpenStack's existing logging infrastructure. To mean broader, this white-box approach is valuable to the DevOps integration current trend [8], allowing developers and operators to better understand the software functioning in realistic deployment environments. It facilitates the spectrum design of approaches to hardening cloud-management stacks, such as fault-injection studies, and online fault detection and analysis, which are planned to explore in future.

11 external APIs1 of OpenStack were been studied and for each API were been executed late request with identifying all fault injection cases, each corresponding to the combination of a fault type and location in the request execution path. A single-fault injections were conducted by re-executing the same request and iterating through the fault-injection cases, each time injecting a distinct fault into the execution flow. Upon completion of fault injection experiments, the results were checked against predefined specifications regarding the expected states and OpenStack behaviours. When specifications are violated, the execution of OpenStack and bugs identifying were manually investigated.

Two OpenStack versions were studied, namely, "essex" and "grizzly", the latter being the first version of the most recent release series, and identify in total 23 bugs. As in the preliminary work version [9], those bugs were been categorized into seven groups and performed an in-depth study for each category. several common fault-resilience issues were been identified then in OpenStack, such as permanent service blocking due to the timeout protection lack, irrecoverable inconsistent system states due to the lack of periodic checking and state stabilization, and misleading behaviours due to code checking of the incautious return. The major contributions of this paper are three-fold:

1. Applied fault-injection techniques to cloud management stacks and presented design and implementation for an operational prototype fault injection framework for this emerging software layer, using OpenStack as the study target.
2. Conducted the first systematic fault-resilience OpenStack study, identifying 23 bugs.
3. Categorized bugs, presenting deep analysis for each bug category, and discussing related fault resilience issues.

Cloud-Management Stack and OpenStack Background

It is briefly discussed cloud-management stacks and then provided background information about OpenStack, concentrating on its components, supporting services, communication mechanisms, and threading model. Cloud-management stacks are an emerging software layer in the cloud ecosystem. They are responsible for the cloud platforms formation and management. A cloud management stack manages cloud platforms via distributed services cooperation, which including an external API service for communicating with external users, an image service for managing VM images (registration and deployment), a computer service for managing VMs (creating and deleting VMs) on supporting hosts, a volume service for managing persistent storage used by VMs (providing block devices and object stores) and network services for managing networks used by VMs (creating and deleting networks, manipulating firewalls on supporting hosts). Its own service as a cloud-management stack requires external services to fulfil its functionality. In particular, it often relies on a hypervisor, described in [10] (Xen), [11] (KVM) or [12] (Hyper-V) for managing VMs.

OpenStack is a state-of-the-art open source cloud management stack [13], implemented in Python. It contains several common services, such as a computation service group, an image service group, network service, and several persistent storage services. Other OpenStack services in a typical cloud setting include an identity service for validating services and users and a dashboard service for graphical interface providing to users and administrators. OpenStack relies on hypervisors installed on computer nodes (where VMs run) for VM management and uses database service to store persistent states related to its managed cloud.

OpenStack employs two major communication mechanisms. Compute services use remote procedure calls (RPCs) compatible to the Advanced Message Queuing Protocol (AMQP) for internal communications within the service group. Other OpenStack services conform to the RE presentational State Transfer (REST) architecture and communicate with each other via the Web Server Gateway Interface (WSGI). OpenStack uses the SQL Alchemy library to communicate with database back ends, such as MySQL and SQLite. Interaction with hypervisors is inattentive to virtualization drivers. Specifically, OpenStack designs a common hypervisor-driver interface and implements drivers using common hypervisor APIs (libvirt and Xen). Its services are implemented as green threads via event let and green let libraries, which employ a user-level cooperative multithreading model: a thread runs non-pre-emptively until it surrenders control. Upon thread yielding, a hub thread turn out to be active, makes a scheduling decision and then transfers control to the scheduled thread. This model requires several standard Python libraries to be patched with green thread-compatible implementations in order to prevent I/O functions issued by one green thread from blocking the other in same process.

Project Scope, Design Principles, Components and Workflow Overview

This section presents the project scope, followed by our design principles discussion. It is present an overview of the components and the workflow of our fault-injection framework. The fault-resilience-related programming bugs are target in OpenStack. They affect OpenStack's inherent fault-resilience from its design and implementation perspective. Configuration bugs, in contrast, are considered faults in this article. For example, a mistaken configuration may lead to network partitions, which are used for fault injection in our framework. Bugs that can only be manifested by a sequence of faults are not in this paper scope, due to single-fault injections use.

Design builds on prior research in distributed systems tracing, fault injection, and specification checking. Instead of proposing a new fault-injection methodology, it is discussed experience in building an operational fault-injection prototype for OpenStack, following below design principles. Cloud management stacks rely on the services cooperation distributed to a cloud environment to fulfil their functionality. This cooperation requires fault-resilient communication mechanisms. Given the service communications importance and the fast advances of sophisticated single-process debugging techniques, fault injection prototype targets service communications in OpenStack.

Domain knowledge has proven valuable for debugging, monitoring, and analysing distributed systems. In [14] showed that developers of applications running in a distributed environment were willing to expose and exploit domain knowledge in a production-level tracing infrastructure designed for application transparency, despite the infrastructure's decent performance without such knowledge. In our prototype OpenStack's high-level semantics expose the fault-injection module and achieve high fault-injection efficiency by injecting faults to high-level communication flows but generic low-level events in runtime systems or operating systems. It is extremely difficult and costly to thoroughly investigate every aspect of the cloud-management stacks fault resilience. It is focused on common cases, injecting common faults during the processing of OpenStack's most commonly used external APIs. These faults are based on existing knowledge related in works [6, 7], and experience with large-scale production-level cloud systems. The APIs selection is based on experience with OpenStack several experimental deployments.

To simplify adoption of our framework, it is used building blocks that cloud-management stack developers are aware with. The choice between the high-level semantics exposure and inference is an exemplification of this principle, because developers have built logging and notification mechanisms exposing such information. Another example is that it is used hybrid approach to implement OpenStack specifications, combining imperative checking via generic Python scripts with declarative checking via the SQL Alchemy library, both of which are widely employed by developers of OpenStack.

Our fault-injection framework consists of a logging and coordination module, a fault injection module, and a specification-checking module. The logging and coordination module is responsible for logging communications among services during external request processing and coordinating the execution of OpenStack and a fault-injection controller. The fault injection module is conceptually composed of a fault injection controller running at a test server node and fault-injection stubs running with OpenStack. The fault injection controller manufactures information collected by the logging and coordination module, makes fault injection decisions, and demands fault-injection stubs to inject faults into OpenStack. The specification-checking module verifies whether the internal states and the externally visible behaviours (HTTP status code returned to external user) of OpenStack at the end of each fault-injection experiment comply with predefined specifications. Fig. 1 presents a system overview and a high-level workflow, latter discussed in next.

The workflow consists fault-free execution, fault injection, and specification checking. For given external request, it starts with fault-free OpenStack execution resulting in successful request processing. The logs produced during the fault-free execution are fed to a parser to generate an execution graph (next detailed), describing communications among services. Combining the execution graph and a predefined fault specification, the framework generates a test plans collection, each consisting of a fault type from the fault specification and a fault-injection location in the execution graph. Fault-injection experiments are then conducted via logging and coordination module and the fault-injection module collaboration, with each experiment corresponding to a test plan. The results of experiment are checked against predefined state and behaviour specifications. It is manually identified experiment bugs causing specification violations.

Table1 – Communication log format

Attribute	Value/Explanation
Identi?er	Unique for each communication
Tag	Unique for each external request
Timestamp	Physical time
Entity	OpenStack/external service name
Type	RPC cast/call/reply, REST request/response, database/hypervisor/ shell operation
Direction	Send/receive, call/return

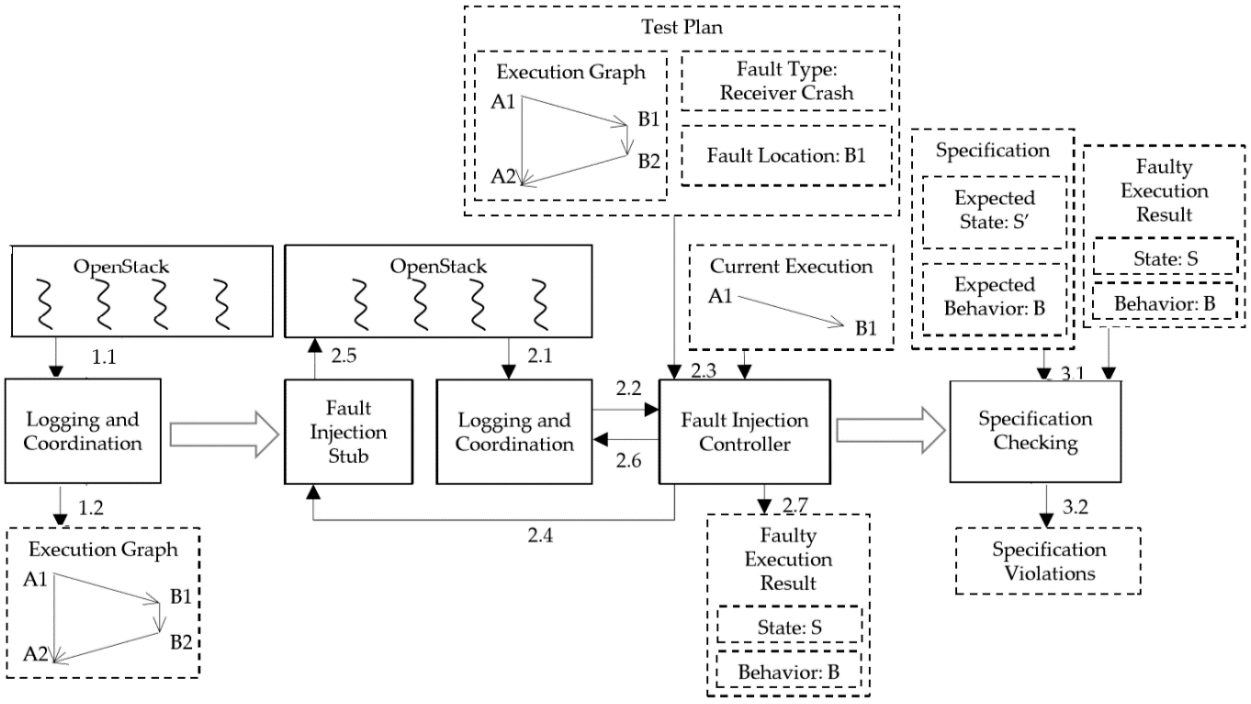

Fig. 1 – System overview and workflow

Logging and Coordination

After the framework overview we start an in depth discussion of its major components with the logging and coordination module. Following the domain specific information design principle exposing to the fault-injection controller, logging and coordination module openly maintains high-level several communications types' semantics in its logs, including RPC, REST, database, hypervisor and shell operations. Key attributes in communication log are enumerated in Table 1. A unique tag is created when

OpenStack receives an external request. The tag is then broadcast through OpenStack services along the request processing path. Recent OpenStack versions employ similar techniques for tracing the request processing within service groups. In contrast, the framework assigns a system-wide unique tag to each external request and traces its processing within the entire stack scope. Unique tags facilitate the log entries extraction related to a given external request. Otherwise, concurrent request processing would cause OpenStack to generate intertwined log entries and increase complexity of the log analysis. Although study currently boards fault injection during the single external request processing, the unique tag is still useful in that. It distinguishes the logs related to request processing from those generated by background tasks, such as periodic service liveness updates.

Boxes in fig. 1 with a solid border represent framework OpenStack and major components. Boxes with a dashed border represent key non-executable objects in the framework. Three stages: fault-free execution, fault injection and specification checking are separated by arrows: step «log OpenStack communications» in a fault-free execution, step «convert logs» to an execution graph, step «log communications» in a fault-injection experiment and pause communicating entities during logging, step «send logs» to fault-injection controller, step «make fault-injection decisions» according to a test plan, step «inform fault-injection stub» of the fault-injection decisions, step «inject faults», step «resume execution», step «collect results» from fault-injection experiments, step «check results» against specifications, step «report specification violations».

System-wide tag broadcast requires modifications to the communication mechanisms in OpenStack. Specifically, it is inserted a new field representing unique tags in both request contexts used by OpenStack services and thread-local storage for those services. When a green service thread is activated during the request processing, it updates the tag value in its thread-local storage with either the tag in the activating request if such a tag exists, or a newly initialized one. The thread associates this tag to all inter-service communications during its current activation. Framework cannot trace a unique tag once it broadcasts across the OpenStack boundary to external services. Consequently, if an OpenStack service communicates with an external service, which in turn communicates with another OpenStack service, then our framework will treat the second communication independent from the first one. So far, it is not encountered such cases in this study, and the logging mechanism suffices for the use.

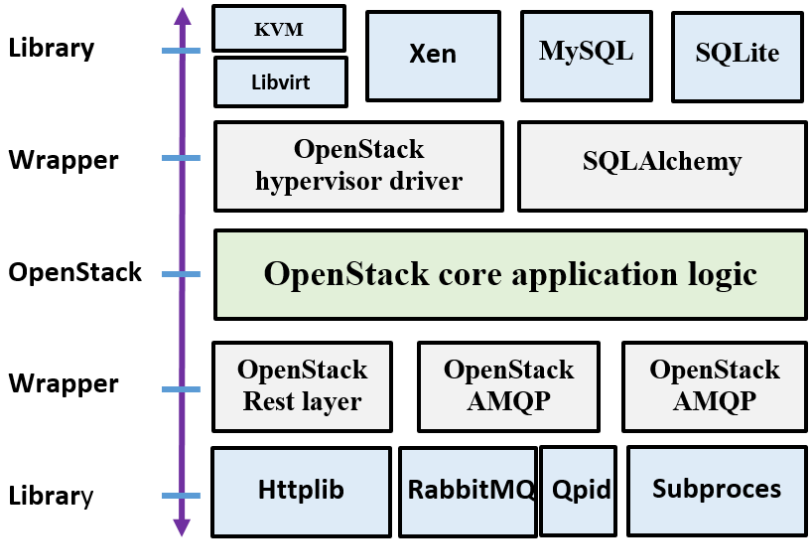

Fig. 2 – Illustration of logging instrumentation

Implementation Pattern

There is the logging module implement by supplementing the communication layers between OpenStack and external services and libraries. On the whole, this module can be implemented at several layers along with communication paths: inside the logic of the OpenStack's core application where high-level communications are initiated, at OpenStack's wrapper layer of communication libraries, in communication libraries themselves, and in system libraries and interfaces. The logging arrangement resides at OpenStack's communication libraries wrapper layer as shown in Fig. 2. The advantages of logging at this layer are two-fold. First, semantics of the high-level can be precisely exposed instead of being incidental at this layer. Second, logging at this layer suffers minimum implementation effort, because it combines communications invented from and destined for OpenStack services. From one side, this layer is shared across OpenStack services and can use the same implementation to log communications from different services. On the other side, this layer is implemented at the communication category granularity (one implementation for all AMQP client libraries) and can use the

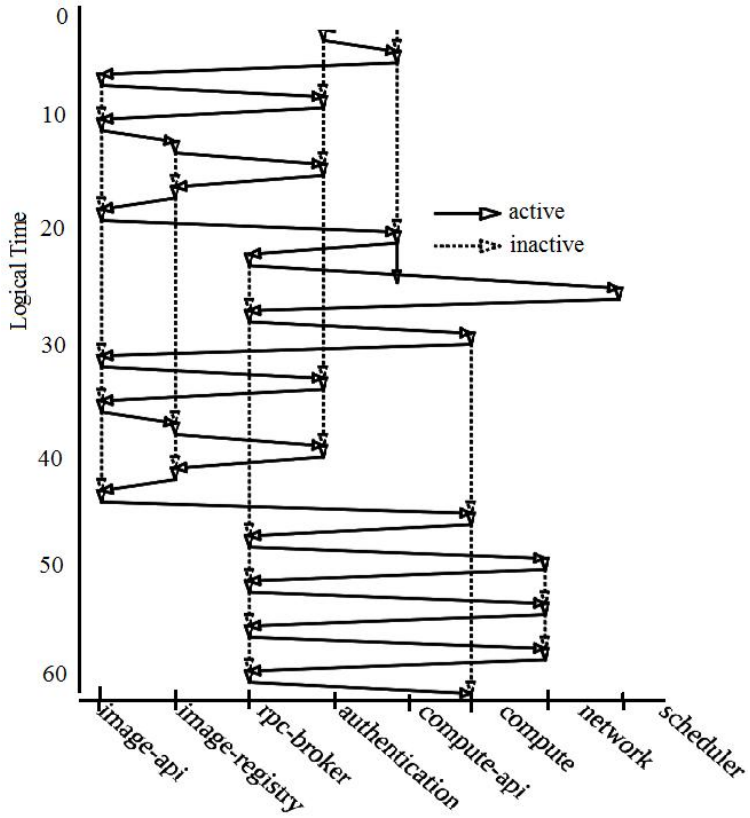

Fig. 3 - Execution graph illustration for VM creation, consisting of REST and RPC communication paths. Solid arrows represent active executions and dashed one represent idleness or waiting for the function call return

same implementation to log communications from supporting services in each category, conceptualizing away details related to individual services.

Whiteboxes with a solid border represent the instrumented layers for exposing high-level communications semantics between OpenStack core logic and supporting services and libraries. Logging snippets are placed in the WSGI implementation of OpenStack and the event let library, as well as several OpenStack client-side libraries for logging REST communications, and in the AMQP interface of OpenStack for logging RPC communications. For logging communications between OpenStack and hypervisors, we implement a logging driver acquiescent with OpenStack's hypervisor interface and use it to wrap the real drivers OpenStack selects to communicate with hypervisors. Communications between OpenStack and hypervisors are thus intercepted and recorded by the logging driver. We insert logging snippets into the SQL Alchemy library for logging database operations. The compute service gears a helper function to perform shell operations on local hosts. We also supplement that function to log such operations.

One drawback of this integrated user-level logging implementation is logging incompleteness. Compared to a system-level logging approach targeting a language level interface (Java SDK) or an operating system interface, the approach is incomplete in that it can only cover major communication mechanisms and is unaware to other channels (a customized socket communication). It need to submit that in a well-designed cloud management stack, the majority of inter-service communications are conducted via several well-defined interfaces, which were instrumented in the study on OpenStack. Additionally, system-level approaches usually lead to a significantly larger number of logs, humiliating system performance and necessitating the use of advanced log parsing and inference logic in the fault injection module. In the framework, there are traded logging completeness for simplicity in exposing high-level semantics and potentially high logging performance.

RPC communications within the compute service group is realised in part by modifying OpenStack's AMQP implementation. Such modifications cannot form a complete RPC communications picture, because the RPC caller and called (or producer and consumer in AMQP terminology) is decoupled by an AMQP broker. RPC cast from an OpenStack compute service is sent to AMQP exchange at the broker and then routed to a message queue. Another compute service subscribing to the message queue then receives the RPC, thus effecting the RPC cast. RPC calls are similar except that the return value goes through the AMQP broker as well.

For fine-grained fault injection control, it is intended to differentiate two stages of RPC message propagation – the first from the RPC caller to the AMQP broker and the second from the AMQP broker to the RPC collie. A direct solution would be to extend our logging module to the AMQP broker implementation (either Rabbit MQ or Qpid). This solution requires a general-purpose AMQP broker to include OpenStack-specific programming logic. Moreover, retrieving unique tags from RPC messages at an AMQP broker implies the abstraction layers elevation from the message transferring protocol

(detailing packet formats) to a high-level RPC message with application semantics incurring significant implementation overhead.

The implementation leaves the AMQP broker intact and instead logs its activity via RPC trampolines – compute service proxies responsible for RPC forwarding. The created trampoline for each compute service is modifying OpenStack's client-side AMQP implementation so that RPCs addressed to a service are delivered instead to its trampoline. The trampoline records those RPCs and forwards them to the original destination. From the execution flow logging perspective, RPC trampolines represent the AMQP broker, thus completing the RPC communications picture.

Generating detailed logs with high-level semantics, the logging scraps also serve as coordination points, synchronizing the OpenStack execution and fault-injection servers. During fault-injection experiments, the logging module sends log messages to a fault-injection server and then blocks the logged OpenStack service. The server makes decisions of fault-injection, injects faults once necessary, and resumes the logged service execution by replying a «continue execution» message to the logging module. The logging module use for coordination is also one major difference between our implementation and the existing notification mechanisms in OpenStack.

Fault Injection

The fault-injection module is responsible for extracting execution graphs from logs, generating test plans, and injecting faults to OpenStack. An execution graph depicts the OpenStack execution during the external request processing. It is directed acyclic graph extracted from logs of fault free request processing procedure with each vertex representing in OpenStack a communication event. Each event is characterized by the communicating entity (an image-API service) and the communication type (REST request send operation). Edges represent causality among events. An edge connects two vertices: 1) if they form a sender-receiver pair or 2) if they belong to the same service and one precedes other. Fig. 3 shows a simplified execution graph related to a VM-creation request.

A test plan consists of three elements: an execution graph, a fault-injection location, and a fault type. Two types of faults are studied: servercrash6 and network partition. These fault types are common failure causes in cloud environments and are well-studied in the literature. Other fault types, such as invalid inputs [15] and performance degradation [16] are not considered here. Correlated faults are also common in real-world deployments but are not within the scope of this work, due to the limitation imposed by our current single-fault injection implementation.

```
Procedure1  Test Plan Generation
test_plans ← an empty list
for all node in exe graph do
   for all fault in fault specs do
      if fault can be injected to node then
         new_plan ← Test Plan (exe graph, node, fault)
         test_plans.append(new plan)
return test_plans
```

Procedure1 demonstrates the test plans generation. Iterating over an execution graph, the algorithm accounts for all fault types applicable to each vertex (a sender server crash targeting REST communications can only be inserted to the vertices performing REST request or response send operations) and makes accordingly test plans. This procedure provides an opportunity for global testing optimization: global because the fault-injection module has a view of the entire execution flow. For example, execution-graph vertices can be clustered by customized criteria, each cluster assigned with a testing priority. Vertices can then be selectively tested within each cluster to reduce overall testing cost. Given that a fault-injection experiment in the framework takes several minutes to complete and that an exhaustive set of test plans for one external request usually leads to hundreds of experiments, such a global optimization opportunity provided by an execution graph is valuable and worth further exploration. For test plan generation, a fault specification is used to define the faults types to be injected and the types of communications in which faults are injected. AS a test-case filter, the fault specification functions enabling the design of experiments set focusing only on a specific fault type (sender-server-crashes) and/or a specific communication type (REST communications). The specification format can be extended

to support other filters, such as confining fault injection to OpenStack services subset. A test plan is fulfilled via the test server cooperation and the logging and coordination module. The test server initializes the execution environment and then re-executes the external request to which the test plan resembles. Then, the test server employs the same log parsing logic for execution graph generation to analyse each log sent by the logging and coordination module. It tracks OpenStack's execution by using the execution graph in the test plan until the fault-injection location has been reached. A fault is then injected as plan specified. And OpenStack runs till the request processing is completed.

Faults of the server-crash are injected by killing relevant service processes via system. Configurations of system are modified such that when it stops the relevant services, a signal SIGKILL is sent, instead of the default signal SIGTERM. Forcing them to drop packets from each other, network-partition faults are injected by inserting IP-tables rules to service hosts that should be network-partitioned.

Specification Checking

The specification-checking module is responsible for verifying whether the results collected from OpenStack executions with injected faults comply with expectations on the states and OpenStack behaviors. Writing specifications for a large-scale complex distributed system is notoriously difficult, due to the numerous interactions and implicit inter-dependencies among various services and their execution environments. It is a key task for developing an effective specification checking module. In effect, the coverage and the states and behaviors granularity in the specifications determine the checking module ability to detect erroneous behaviors and target system states. A few approaches have been reported in the literature, including relying on developers to generate specifications [17], reusing system design specifications [6], and employing statistical methods [18]. To the best of our knowledge, OpenStack does not provide detailed and comprehensive specifications on system behaviors or state transitions during the external requests processing. The specifications that are used in this study are generated based on OpenStack understanding, existing knowledge in fault-resilient system design, and first principles, which mirrors the developer specification-generation approach. Specifically, specifications generated manually by inferring OpenStack developers' expectations on system states and behaviors. This process requires extensive reverse-engineering efforts, such as source-code reading and log analysis. Specifications generated in such a manner may require further debugging and refinements (similar to fixing incorrect expectations in [17]). Such specifications are best-efforts, with a coverage constrained by OpenStack understanding. Such specifications usefulness also is demonstrated by the bugs identification reported in this paper.

Specification 1 VM State Stabilization Specification
query = *selectVM* from *compute database*
where *VM.state in collection*(*VM unstable states*)
if *query.count*() = 0 then
return *Pass*
return *Fail*

Specification-Generation Guidelines listed below are usable specification-generation guidelines. "*Do not block external users*" announce that OpenStack shouldn't block external users due to faults during request processing. "*Present clear error messages via well-defined interfaces*" says that OpenStack should expose clear error states to external users via well-defined interfaces and avoid confusing information. "*Stabilize system states eventually*" informs that upon restoration of faulty services and with the quiescence of externally triggered activities, OpenStack should eventually stabilize inconsistent states caused by faults during request processing time.

Specification checking can be implemented via a general-purpose programming language or a specially designed specification language, using an imperative approach or a declarative approach. Following the using developer-familiar building blocks principle, a hybrid approach is adopted, applying declarative checking on persistent states stored in OpenStack's databases and imperative checking on the other states and OpenStack behaviors. Database-related checks are implemented via the SQL Alchemy library and others are implemented as generic scripts. This hybrid approach largely imitates OpenStack's existing implementation: Open Stack adopts the same approaches to controlling its states and behaviors.

Specifications implemented for OpenStack's states stored in databases and local filesystems and OpenStack's behaviors, such as the Http status code returned to an external user after processing a request. The specified expected states of cloud platforms are managed by OpenStack, such as the local hypervisors states on compute hosts and the Ethernet bridge configurations.

Specification 2 Ethernet Configuration Specification
if(*VM.state = ACTIVE*) and
  ((*VM.host.Ethernet not setup*)
  or (*network controller.Ethernet not setup*)) then
  return *Fail*
return *Pass*

Specification 3 Image Local Store Specification
query = select image from image database
  where image.location is local
if *local image store.images = query.all()* then
  return *Pass*
return *Fail*

Below was present three specification examples. Specification 1 indicates the VM state stabilization expectation. The VM state after the external request processing (VM creation) and a sufficient quiescence period should enter as table state (the ACTIVE state, indicating the VM running activity) instead of remaining in a transient state (the BUILD state, indicating the VM creation process). This is a using example declarative checking on database states.

Specification 2 requires VM actively running, the Ethernet bridges on the compute host where that VM resides and the host running the network controller service have been correctly set up. It is checked whether the bridges have been associated with the correct Ethernet interfaces dedicated to the subnet to which the VM belongs. It exemplifies the imperative checking on OpenStack-managed cloud platforms.

Specification 3 checks whether the database maintained states of OpenStack image service regarding the image store in the local filesystem are in accordance with the filesystem. It requires that if an image is uploaded to the local image store and thus exists in the filesystem of the image service host, the nits location attribute in the image database should be local, and vice versa. This specification shows a combined check on the database views and the service host filesystem.

Results

Here is discussing the bugs uncovered by our framework applied to OpenStack essex and grizzly versions, and find 23 bugs in total: 13 common to both versions, 9 unique to essex, and 1 unique to grizzly. The study covers three OpenStack service groups: the identity service (keystone), the image service (glance), and the compute service (nova). For external services, the framework supports the Qpid messaging service, the MySQL database service, the libvirt service for hypervisor interaction and the Apache Http server used as an image store for OpenStack.

The identity service is configured to use UUID tokens for authentication. Regarding the image service, it is configured to use a local filesystem or Http server as its backend store. As for the compute service, QEMU is used as the backend hypervisor, controlled by the libvirt interface. In essex, itis limited the reconnection from the Qpid client library to the backend broker equal 1.

OpenStack services run in VMs with 1 virtual CPU and 2GB memory. All OpenStack VMs run on an HP Blade System c7000 enclosure, each blade equipped with 2 AMD Opteron 2214HE (2.2GHz) processors and 8GB memory. For each fault-injection experiment, all services are deployed in one VM by default, each started as a single example. There are two exceptions to the above placement guideline. If a shell-operation fault should be injected to a compute service, then that service is placed in one VM and the other services are placed in another VM in order to prevent interference among similar shell operations of different services. If a network partition fault should be injected, then the pair of to be partitioned services are placed in two VMs and the other services are launched in a third VM.

There are tested 11 external OpenStack APIs, inject 3848 faults, implement 26 specifications, detect 1520 violations and identify 23 bugs. The results are summarized in Table 2. The table shows the bugs significance and issues discovered in our study, because they are manifested (I) in several frequently

used APIs and (II) in numerous locations along the execution paths of those APIs. The former observation is drawn from the fact that the bugs sum in the table exceeds by far the number of distinct bugs. The fact is that the ratio of specification violations to bugs is greater than 1 for each API, which is also shown in Fig. 4.

The bugs are classified into seven categories (Table 3) and present deep discussion of each category. Compared to our preliminary work [9], this article contains newly-identified bugs in OpenStack and also discusses the bugs evolution and fixes across the two OpenStack versions that is studying, demonstrating the OpenStack's fault resilience improvement and the outstanding issues.

Table 2: Summary of fault-injection results.

API	Faults				Specification Violations				Bugs			
	essex		grizzly		essex		grizzly		essex		grizzly	
	crash	part	crash	part	crash	part	crash	part	crash	part	crash	part
VM create	217	133	311	229	93	43	150	49	8	6	3	2
VM delete	79	61	102	82	51	15	45	23	9	5	5	2
VM pause	24	17	35	29	16	13	6	4	5	6	2	1
VM reboot	64	36	139	104	9	11	0	5	3	4	0	1
VM rebuild	159	106	242	183	103	67	0	13	5	5	0	1
VM image create (local)	142	119	171	150	59	106	90	79	4	2	3	3
VM image create (http)	107	92	171	150	24	84	79	71	3	2	3	3
VM image delete (local)	59	44	22	15	23	37	12	9	3	2	2	2
VM image delete (http)	59	44	22	15	23	37	10	8	2	2	1	1
Tenant create	7	6	7	6	0	6	0	6	0	1	0	1
User create	7	6	7	6	0	6	0	6	0	1	0	1
Role create	7	6	7	6	0	6	0	6	0	1	0	1
User-role create	9	8	10	9	0	8	0	9	0	1	0	1
Sum	940	678	1246	984	401	439	392	288	42	38	19	20

The results of image service APIs are broken down according to whether a local file system or an Http service is used as the image store. Specification violations related to an API are counted as the number of fault-injection experiments in which at least one violation is detected. The network-partition fault is shortened to "Part."

Table3: Bug Categories

Category	Count		
	common	essex only	grizzly only
Timeout	1	1	0
Periodic checking	6	4	0
State transition	1	0	1
Return code checking	4	1	0
Cross-layer coordination	0	1	0
Library interference	0	1	0
Miscellaneous	1	1	0
Total	13	9	1

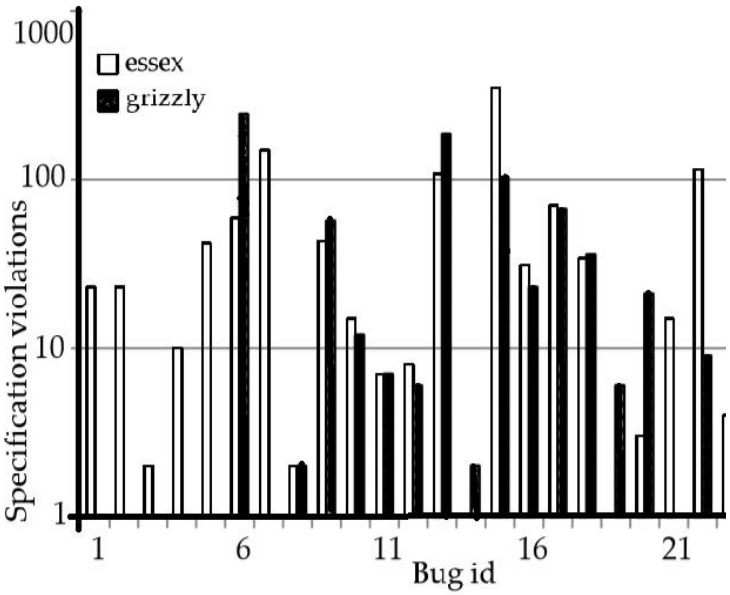

Fig. 4: Bug distribution. The Y-axis is the number of fault-injection experiments in which at least one specification violation leading to th given bug identification is detected

Timeout is a common mechanism in distributed systems to avoid one faulty service from indefinitely blocking other services and affecting system-wide functionality. OpenStack extensively uses the timeout mechanism with settings scattered across multiple service configurations to control various timeout behaviors of OpenStack services and external supporting services. Setting correct timeout values is known to be difficult. Given the numerous interactions and inter-dependencies among service groups and the deployment environments variety, it is very difficult if not impossible for OpenStack to provide a comprehensive set of timeout values covering all execution paths that can block the system.

For example, REST communications, one of the two major communication mechanisms used in OpenStack, fall out of the safety net of the timeout mechanism in the OpenStack essex version. Consequently, service is waiting for a response from another service via the REST mechanism may be indefinitely blocked if the two services become network partitioned after the request is sent but before the response is received. This also exemplifies the advantage of an execution-graph-based fault injection over coarser-grained approaches [19], such a bug can hardly be efficiently revealed by the latter due to the requirement on the synchronization between fault injections and send or receive operations.

This bug is fixed in the OpenStack grizzly version by supporting timeout values for REST communications and exposing such settings in its configurations. For instance, the image client library provides a default 600 seconds timeout. The identity client library, on the other hand, uses a default value of None, effectively disabling the timeout mechanism. While a system-wide default timeout value for REST communications might be a better solution, the fault OpenStack resilience has been clearly enhanced due to this critical timeout setting support.

Periodic checking is another critical mechanism for achieving fault resilience in OpenStack. In a cloud environment where faults are in evitable, periodic checking can monitor service liveness, resume interrupted executions, clean up garbage, and prevent resource leakage.

The fault-injection framework identifies several bugs caused by the lack of proper periodic checking mechanisms. Take the processing of a VM creation request as an example. During fault-free request processing, the VM state transits from None (non-existence) to BUILD (under construction) to ACTIVE (actively running). If the execution is interrupted after the VM has transited to BUILD, it is possible that, in essex, the VM indefinitely remains in that transient state. This is a bug because VM creation should cause the VM to enter a stable state (ACTIVE if the processing is successful but otherwise ERROR) in a timely manner, despite faults. This bug can be fixed by a periodic checking logic that converts VM's state from BUILD to ERROR if the related VM creation request is detected as failed. A bug fix has been integrated into grizzly, conducting such a transition based on a configurable VM

creation timeout value. This fix although effective in preventing the prolonged BUILD state, leads to a state transition bug.

Another bug in both OpenStack versions that can be fixed by periodic checking is related to database access in a fault-handling logic. OpenStack implements a decorator function wrap instance fault, injecting a fault record into a database table upon VM related faults detection. This fault-handling logic fails when the VM related fault that it intends to log is itself a database fault (database service crash fault during a VM creation). In such situation the fault record is discarded, making it difficult for post mortem fault analysis. This bug can be fixed by keeping a temporary log of the fault messages that cannot be stored in the designated database table at the time when they are generated. Then periodically checking the database service state and merging those temporary messages into the database when possible.

OpenStack maintains a large number of states in its databases, with complicated state-transition diagrams among them. A tool indicates that in faulty situations users may experience problematic state transitions. For example, grizzly employs a periodic task to convert a VM from BUILD to ERROR if it exceeds the maximum time that a VM is allowed to remain in BUILD. However, OpenStack does not cancel the VM creation request related to that VM. If a VM creation request experiences a transient fault, then a VM may transit from BUILD to ERROR and then to ACTIVE, because it can be created after the execution of the periodic task and the fault disappearance.

The state transition from ERROR to ACTIVE without external event triggering can be confusing and problematic. According to experience, upon VM creation error receipt an external user is likely to issue another VM creation request. The bug fix for the previous periodic-checking-related problem induces a new state-transition bug, potentially creating more VMs than needed for an external user. It suggest canceling an external request and negating its effect once it is believed to be erroneous in addition to the state stabilization employed by OpenStack.

Return code is commonly used as an indicator of the execution state from a function callee to its caller. Although thorough checking on return codes has long been established as a good programming practice, prior work has identified return code related bugs to be a major bug's source even in well-organized projects [20, 21]. Study on OpenStack confirms this observation. For example, during the processing of a VM creation request, when the identity service cannot authenticate a user-provided token passed from a compute API service due to an internal fault, it returns an error code to the compute API service correctly indicating the service fault. Due to a flawed return code checking logic the compute API service attributes such an error to the token invalidity generates a misleading error message accordingly and returns it to the external user.

Another example in this category is related to the execution of shell commands. OpenStack compute services implement a common function for executing shell commands allowing the caller to specify a list of expected return codes. If the return value falls out of that list, an exception is raised. A common bug pattern related to improper use of this function results from disabling its return code checking logic causing OpenStack when executing under faults, to deviate from expected execution flows without being detected. During the network setup procedure related to a VM creation, the *brctl addif* command is used to associate an Ethernet interface with a bridge on the compute host where the VM is placed. OpenStack assumes that the command can be fruitfully executed without checking its return code proceeds to start the VM. So, the VM may lose network connectivity if a fault occurs during execution of that command.

OpenStack relies on various supporting services to maintain its functionality and supports interaction with multiple services in each external service category via set of abstraction layers. Take the RPC messaging services as an example. OpenStack implements a unified layer for the AMQP protocol used for RPC communications, giving operations to a lower layer implemented for a specific AMQP broker. The lower layer is a client-side library wrapper provided by its corresponding broker such as RabbitMQ and Qpid. This client-side library comprises messaging implementation details and is responsible for the actual communication with the AMQP broker.

This valuable and well-designed multi-layer abstraction stack imposes stringent requirements on cross layer coordination. Incorrect interpretations of one layer behaviors may lead to subtle bugs in another layer. The Qpid client library is configured to automatically reconnect the AMQP broker after a connection disruption in essex. A threshold value is designed to control the maximum reconnections number. A connection maintained by the client library resides in a temporary erroneous state until the

threshold is reached at the time the connection enters a permanent erroneous state. The OpenStack client library wrapper does not coordinate properly with the client library regarding the temporary-permanent error state transition and causing the wrapper to vainly retry a connection that has been marked as irrecoverable by the client library.

The extensive external libraries use commonly found in large-scale software systems may lead to unexpected library interference. OpenStack uses a patched version of Python standard library functions to support cooperative thread scheduling. Subtle incompatibility in the patched functions, however, can engender bugs that are hard detected. Take the communication between OpenStack and Qpid broker again as an example. During a reconnection from the Qpid client library to a Qpid broker service the client internally uses a conventional consumer/producer synchronization via a select/read call pattern on a pipe. Due to incompatibility between the patched version of select and its counterpart in Python standard library, this Qpid client library, when invoked in essex, may perform a read on the pipe read-end that is not yet ready for reading, thus permanently blocking an entire compute service.

The framework also detects a simple implementation bug: in essex, when a fault disrupts a connection opening procedure in the Qpid wrapper layer, a subsequent open call is issued without first invoking close to clean up sour states in the connection object resulting from the previously failed open, causing all following retries to fail with an «already open» error message.

Comparing the results of the two versions, there are identified several interesting aspects in the OpenStack's fault resilience evolution. Using timeouts to limit the distributed operations execution is a well-known and important approach to fault-resilience improvement [22]. Use of timeout for REST communications effectively solves the indefinite sender blocking issue in essex. Systematically configuring timeouts also remains an open question. Different components in a REST communication flow (WSGI pipeline) have different default timeout values. The timeout settings of some important supporting services cannot be controlled by OpenStack. For example, OpenStack does not specify the timeout for SQL statement execution, thus causing long blocking time if the SQL statement issuing service and the database backend is network partitioned. These issues need to be properly addressed further improving the OpenStack fault resilience.

Carefully checking *return codes* enables prompt error detection. In grizzly, during the VM deletion request processing the compute service issues a RPC call instead of a RPC cast as in essex to the network service, demanding the latter to reclaim relevant network resources. This modification allows the compute service to detect errors in the network service and reacts accordingly (transiting the VMtothe ERROR state), reducing the possibility of network resource leakage under faults.

By simplifying the *cross-layer coordination*, OpenStack reduces the bugs hiding chances between abstraction layers. By disabling automatic reconnection in the Qpid client library and reserving full control only in its wrapper layer OpenStack avoids the bug discussed before. Confining the decision-making logic regarding a specific aspect of cloud-management stack to a single layer, instead of coordinating the decision making in different layers, is considered a good design practice.

Discussions

The framework can be applied to solve fault-resilience issues related to cloud-management stacks in real-world deployments. One of Amazon's cascading failures [3] was caused by a memory leakage bug on storage servers, which was triggered by the data-collection server crash, the subsequent inability to connect to that failed server from storage servers, and the deficient memory reclamation for failed connections. This bug can be detected by the framework via the crash fault combination injected to the data-collection server and specification on the memory usage of the data reporting logic on storage servers. The framework needs to be extended for supporting multiple faults injection and scaling specification-checking logic from an individual request to multiple requests. Such improvements will enable the framework to handle complicated issues with multiple root causes [2].

One may consider the generality of this study in two aspects: the implementation reusability and findings applicability to other cloud-management stacks. As to implementation reusability, the fault-injection module and the specification-checking module of the framework are reusable in relevant studies for other cloud-management stacks. The logging and coordination module and the specifications used for checking the behaviors and states of OpenStack are domain-specific and require porting efforts. This also holds the study across two OpenStack versions. Such cross-version porting is straightforward. The logging and coordination module integrated in OpenStack contains some code lines, most of which are

reusable across the two versions. Specifications need to be moderately adjusted to handle with minor semantic (database schema changes) evolution.

Regarding the findings, the specific bugs and the related analysis presented in this work are OpenStack specific and cannot be sweeping to other cloud management stacks. The bug categories and the related fault-resilience issues are general. Despite the numerous differences in the detailed design and implementation of cloud-management stacks, many of them [23, 24] share common high-level scheme. They have similar service groups, rely on similar external supporting services, and employ similar communication mechanisms. The findings in this article have the potential to elucidate fault resilience in other cloud management stacks with a similar design.

The execution graphs and test plans use is optional. Instead of obtaining an execution graph related to request processing and generating test plans based on the graph before fault-injection experiments, an experiment could be started without prior knowledge and inject faults when a relevant communication event occurs.

Our choice of the execution graph use is mainly for future framework extensibility. On the one hand, an execution graph depicts the entire execution flow related to the external request processing and thus allows an intelligent test planner to conduct fault injection experiments with better test coverage and less resource consumption. On the other hand, execution graphs are useful for fault-resilience studies other than our current fault-injection framework, such as graph-comparison-based online fault detection [18].

In The framework is required that the external request processing during a fault-injection experiment match its execution graph up to the fault-injection location. The communication events observable by our framework in fault-free execution generally need to be deterministic. This requirement is satisfied in most of experiments. There are cases where additional care is required to accommodate nondeterminism. One source of nondeterminism is periodic tasks, which is resolved by deferring the tasks interfering with experiments. Another source is environmental inputs. The operations taken by a compute service for network setup on its local host depend on whether the compute service and network service are co-located on the same host. There are handled such cases by annotating related execution graphs so that all the paths observed in fault-free execution are marked as valid.

Related Work

Cloud-management stacks are type of distributed system. Our fault resilience study of this layer benefits from prior research on fault pliability of distributed systems in general and cloud systems. Here is compared the result of this work with existing fault resilience studies and execution path extraction and specification checking with similar techniques employed in distributed systems debugging and failure detection.

Fault injection is commonly used to study the cloud systems *fault resilience*. FATE is a fault-injection framework directing the recovery logic in cloud applications and exploring failure scenarios with multiple-failure injection [6]. In the work [5] crash faults were been injected to components on Hadoop's compute nodes and studied their effects on application performance. The framework targets cloud-management stacks and examines the recovery logic by conducting single-fault injection during the external requests processing. Similar to the target of FATE, it is studied the recovery logic functionality and correctness, which is difficult to be made correct.

Failure of Service (FaaS) is proposed as a new generic service to improve the cloud applications fault resilience in real deployments [25]. It is suggested that an alternative approach for cloud-management stacks, presenting an integrated fault-injection framework with domain knowledge. By combining them, cloud management stacks may enhance fault resilience by better balancing the cost and fault injection coverage.

Model checking in [7] is another common approach to examining the distributed systems fault resilience. Compared to our fault-injection-based approach, it checks the target system more thoroughly by exploring all possible execution paths instead of those observed by a fault-injection framework. This same thoroughness requires the extensive domain knowledge use in order to make it practical to check highly complicated operations in cloud-management stacks of the VM creation.

Execution path have been extensively used for workload modeling [26, 27], performance [28] and correctness debugging [18, 29], and evolution analysis [30] in distributed systems. Such information is exposed via special logging modules or inferred in sophisticated post-processing. Applying existing

knowledge to the framework, execution paths is extracted relating to external request processing via user-level logging, which explicitly exposes high-level semantics.

Prior research has explored various approaches to *specification checking* in distributed systems. Regarding specification expression, imperative approaches [31, 32] and declarative approaches [6, 17] have been studied. In our framework, a hybrid approach is employed in expressing specifications on the states and behaviors of cloud-management stacks, combining imperative and declarative checking. Similar combinations have been used to query and analyze distributed trace events [33]. Regarding specification generation, our and most existing approaches require developers to implement specifications. Recent advances in filesystem-checker testing leverage the characteristics in the checkers to automatically generate implicit specifications [34]. The applicability of similar approaches to cloud-management stacks remains an open question for today.

10 Conclusions

In this article, a systematic study is conducted on the fault resilience of OpenStack. It is designed and implemented a prototype fault-injection framework that injects faults during the external requests processing. Using this framework, it is uncovered 23 bugs in two OpenStack versions, classified them into seven categories, and presented deep discussion of the fault resilience issues, which must be addressed in order to build fault-resilient cloud-management stacks. In future for the research can be taken refining and automating specification generation logic in fault-resilience areas and exploring potential use of execution graphs.

1. Amazon. Amazon elastic compute cloud (Amazon EC2). http://aws.amazon.com/ec2/. Retrieved in September 2013.
2. Amazon. Summary of the Amazon EC2 and Amazon RDS service disruption in the US east region. http://aws.amazon.com/ message/65648/. Retrieved in September 2013.
3. Amazon. Summary of the October 22, 2012 AWS service event in the US-east region. http:// aws.amazon.com/message/680342/. Retrieved in September 2013.
4. Microsoft. Summary of Windows Azure service disruption on Feb 29th, 2012. http://blogs.msdn.com/b/windowsazure/archive/2012/03/09/ summary-ofwindows-azure-service-disruptionon-feb-29th-2012.aspx. Retr. in September 2013.
5. F. Dinu and T. E. Ng. Understanding the effects and implications of compute node related failures in Hadoop. In Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing, HPDC '12, pages 187–198, New York, NY, USA, 2012. ACM.
6. H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. ArpaciDusseau, K. Sen, and D. Borthakur. FATE and DESTINI: a framework for cloud recovery testing. In Proceedings of the 8th USENIX conference on Networked systems design and implementation, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.
7. J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M.Yang, F.Long, L.Zhang, L.Zhou. Modist: transparent model checking of unmodified distributed systems. In Proceedings of the 6th USENIX symposium on Networked systems design and implementation, NSDI'09, pages 213– 228, Berkeley, CA, USA, 2009. USENIX Association.

A. Reddy. DevOps: The IBM approach. Technical report, IBM, 2013.

8. X. Ju, L. Soares, K. G. Shin, and K. D. Ryu. Towards a fault-resilient cloud management stack. In USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'13. USENIX Association, 2013.
9. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In Proceedings of the nineteenth ACM symposium on Operating systems principles, SOSP '03, pages 164 – 177, New York, NY, USA, 2003. ACM.

A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In Ottawa Linux Symposium, pages 225 – 230, 2007.

10. Microsoft. Microsoft Hyper-V server 2012. http://www.microsoft.com/enus/server-cloud/hyper-v-server/ default.aspx. Retrieved in September 2013.
11. OpenStack. OpenStack open source cloud computing software. http://www.openstack.org/. Retrieved in September 2013.
12. B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C.Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.
13. B.P. Miller, L. Fredriksen, B. So. An empirical study of the reliability of UNIX utilities. Commun. ACM, 33(12):32 – 44, Dec. 1990.
14. T. Do, M. Hao, T. Leesatapornwongsa, T. Patanaanake, and H. S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In2013ACMSymposiumonCloudComputing, SOCC'13, New York, NY, USA, 2013. ACM.
15. P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: detecting the unexpected in distributed systems. In Proceedings of the 3rd conference on Networked Systems Design and Implementation - Volume 3, NSDI'06, Berkeley, CA, USA, 2006. USENIX Association.
16. M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation – Vol. 1, NSDI'04,Berkeley,CA,USA,2004.USENIXAssociation.

17. C. Bennett and A. Tseitlin. Chaos monkey released into the wild. http://techblog. netflix.com/2012/07/chaos-monkeyreleased-into-wild.html. Retrieved in September 2013.
18. H. S. Gunawi, C. Rubio-Gonz´alez, A. C. ArpaciDusseau, R. H. Arpaci-Dussea, and B. Liblit. Eio: error handling is occasionally correct. In Proceedings of the 6th USENIX Conference on File and Storage Technologies, FAST'08, pages 207–222, Berkeley, CA, USA, 2008. USENIX Association.
19. P. D. Marinescu, G. Candea. Efficient testing of recovery code using fault injection. ACM Trans. Comp. Syst., 29(4):11:1 – 11:38, Dec. 2011.
20. J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, M. Walfish. Detecting failures in distributed systems with the falcon spy network. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, pages 279 – 294, New York, NY, USA, 2011. ACM.
21. CloudStack. Apache CloudStack: Open source cloud computing. http://cloudstack. apache.org/. Retrieved in September 2013.
22. Eucalyptus. The Eucalyptus cloud. http://www.eucalyptus.com/eucalyptuscloud/iaas. Retrieved in September 2013.
23. H. S. Gunawi, T. Do, J. M. Hellerstein, I. Stoica, D. Borthakur, and J. Robbins. Failure as a service (Faas): A cloud service for large-scale, online failure drills. In Technical Report UCB/EECS-201187.
24. P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. InProceedingsofthe6thconferenceon Symposium on Operating Systems Design & Implementation - Volume 6, OSDI'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.
25. B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, R.N.Chang. vpath: precise discovery of request processing paths from black-box observations of thread and network activities. In Proceedings of the 2009 conference on USENIX Annualtechnicalconference,USENIX'09,pages19– 19, Berkeley, CA, USA, 2009. USENIX Association.
26. P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, A.Vahdat. WAP5: black-box performance debugging for wide-area systems. In Proceedings of the 15th international conference on WorldWideWeb'06, pp. 347 –356, New York, NY, USA, 2006. ACM.
27. D.Geels, G.Altekar, S.Shenker and I.Stoica. Replay debugging for distributed applications. In Proceedings of the annual conference on USENIX '06 Annual Technical Conference, ATEC '06, Berkeley, CA, USA, 2006. USENIX Association.
28. S.A. Baset, C. Tang, B.C. Tak, and L. Wang. Dissecting open source cloud evolution: An open stack case study. In USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'13.USENIXAssociation, 2013.
29. X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: debugging deployed distributed systems. In Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, NSDI'08, pages 423–437, Berkeley, CA, USA, 2008. USENIX Association.
30. X. Liu, W. Lin, A. Pan, and Z. Zhang. Wids checker: combating bugs in distributed systems. In Proceedings of the 4th USENIX conference on Networked systems design and implementation, NSDI'07, pp. 19 –19, Berkeley, CA, USA, 2007. USENIX Association.
31. U.Erlingsson, M.Peinado, S.Peter, M.Budiu, and G. Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. ACM Trans. Comp. Syst., 30(4):13:1–13:35, Nov. 2012.
32. J. Carreira, R. Rodrigues, G. Candea, and R. Majumdar. Scalable testing of file system checkers. In Proceedings of the 7th ACM European conference on Computer Systems, EuroSys '12, pages 239– 252, New York, NY, USA, 2012. ACM.