On Fault Resilience of OpenStack

Xiaoen Ju[†] Livio Soares[‡] Kang G. Shin[†] Kyung Dong Ryu[‡] Dilma Da Silva[§]
[†]University of Michigan [‡]IBM T.J. Watson Research Center [§]Qualcomm Research Silicon Valley

Abstract

Cloud-management stacks have become an increasingly important element in cloud computing, serving as the resource manager of cloud platforms. While the functionality of this emerging layer has been constantly expanding, its fault resilience remains under-studied. This paper presents a systematic study of the fault resilience of OpenStack—a popular open source cloud-management stack. We have built a prototype fault-injection framework targeting service communications during the processing of external requests, both among OpenStack services and between OpenStack and external services, and have thus far uncovered 23 bugs in two versions of OpenStack. Our findings shed light on defects in the design and implementation of state-of-the-art cloud-management stacks from a fault-resilience perspective.

Categories and Subject Descriptors

D.4.2 **[Operating Systems]:** Reliability

Keywords

Fault injection, cloud-management stack, OpenStack

1 Introduction

With the maturation of virtual machine (VM) technology in both software design and hardware support, cloud computing has become a major computing platform. In addition to public cloud services (e.g., Amazon EC2 [2]) that have been available since the early stage of cloud platform deployment, there is an emerging demand for other types of cloud platforms, notably, private and hybrid clouds. This demand leads to the role transition from cloud users to a combination of cloud users and providers, broadening the scope of cloud providers from major IT companies to organizations of any size. It has also prompted the research and development of cloud-management stack—a new software stack that functions as a high-level cloud operating system and is key to resource management in cloud platforms.

The significant attention to cloud-management stacks from both academia and industry has led to a rapid increase in the number of features in recent years. Fault resilience of this layer, however, is still regarded as an optional feature and remains under-studied, despite its importance demonstrated by real-world failures and its significant impact on its managed cloud platforms [3, 4, 29]. Fault-resilience-related issues constantly perplex the users of cloud-management stacks. For example, when faults occur, VM creation may fail or take extremely long time, and VMs may be marked as successfully created but lack critical resources (e.g., IP addresses), thus remaining unusable. A thorough investigation of the fault resilience of cloud-management stacks that demystifies the above issues is long overdue.

In this paper, we present the first systematic study on the fault resilience of OpenStack, a popular open source cloud-management stack. Leveraging conventional wisdom in the fault-injection literature as well as its application in fault-injection studies targeting large-scale distributed systems [12, 19, 38], we study the execution of OpenStack in the presence of two common faults in the cloud environment: server crashes and network partitions. OpenStack is considered *fault-resilient* during the processing of an external request, if it maintains correct and consistent states and behaviors, even in case of occurrence of faults. As external requests are an important source of inputs to OpenStack and usually trigger state transitions, we focus on OpenStack's fault-resilience during external request processing. We inject faults into inter-service communications during request

SOCC'13, October 01–03 2013, Santa Clara, CA, USA.
Copyright 2013 ACM 978-1-4503-2428-1/13/10$15.00
http://dx.doi.org/10.1145/2523616.2523622

processing, as they characterize service collaboration of which design and implementation is difficult to be fault-resilient. Specifically, we target communications among OpenStack's compute, image, and identity services, as well as external services such as databases, hypervisors, and messaging services.

We take a white-box approach in this fault-injection study, exposing high-level semantics of OpenStack (e.g., service *A* sends a request *R* to service *B* via communication channel *C*) by augmenting its wrapper layer of communication libraries with our logging and coordination module. 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, because it closely mirrors OpenStack's existing logging infrastructure. In a broader sense, this white-box approach is of value to the current trend of DevOps integration [33], enabling developers and operators to better understand the functioning of software in realistic deployment environments. It facilitates the design of a spectrum of approaches to hardening cloud-management stacks, such as fault-injection studies, as exemplified in this paper, and online fault detection and analysis, which we plan to explore in future.

We study 11 external APIs[1] of OpenStack. For each API, we execute the related request and identify all fault-injection cases, each corresponding to the combination of a fault type and a location in the execution path of the request. We then conduct single-fault injections 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, we check the results against predefined specifications regarding the expected states and behaviors of OpenStack. When specifications are violated, we manually investigate the execution of OpenStack and identify bugs.

We study two OpenStack versions, namely, essex (2012.1) and grizzly (2013.1), the latter being the first version of the most recent release series,[2] and identify a total of 23 bugs. As in the preliminary version of this work [22], we categorize those bugs into seven categories and perform an in-depth study for each category. We then identify several common fault-resilience issues in OpenStack, such as permanent service blocking due to the lack of timeout protection, irrecoverable inconsistent system states due to the lack of periodic checking and state stabilization, and misleading behaviors due to incautious return code checking.

The major contributions of this paper are three-fold:

1. We apply fault-injection techniques to cloud-management stacks and present the design and implementation of an operational prototype fault-injection framework for this emerging software layer, using OpenStack as the target of our study.
2. We conduct the first systematic fault-resilience study on OpenStack, identifying 23 bugs.
3. We categorize the bugs, present an in-depth analysis for each bug category, and discuss related fault-resilience issues.

The remainder of the paper is organized as follows. Section 2 provides the background of cloud-management stacks and OpenStack in particular. Section 3 presents an overview of the our fault-injection framework, followed by in-depth discussions of its major components in Sections 4–6. Section 7 presents those bugs identified by our framework in the two OpenStack versions and their related fault-resilience issues. Section 8 discusses several interesting aspects of the study, followed by a discussion of the related work in Section 9. Finally, Section 10 concludes the paper.

2 Background

In this section, we briefly discuss cloud-management stacks and then provide background information about OpenStack, focusing on its major components, supporting services, communication mechanisms, as well as threading model.

2.1 Cloud-Management Stacks

Cloud-management stacks are an emerging software layer in the cloud ecosystem, responsible for the formation and management of cloud platforms. A cloud-management stack manages cloud platforms via the cooperation of distributed services. Typical services include an external API service for communicating with external users, an image service for managing VM images (e.g., registration and deployment), a compute service for managing VMs (e.g., creating and deleting VMs) on supporting hosts, a volume service for managing persistent storage used by VMs (e.g., providing block devices and object stores) and a network service for managing networks used by VMs (e.g., creating and deleting networks, manipulating firewalls on supporting hosts).

Besides its own services, a cloud-management stack requires external services to fulfill its functionality. In particular, it often relies on a hypervisor, such as Xen [6], KVM [23] or Hyper-V [28], to manage VMs.

[1] We use *external APIs* and *external requests* interchangeably.

[2] Since 2012, OpenStack has released three series: essex (first version released on Apr. 5, 2012), folsom (first version released on Sep. 27, 2012) and grizzly (first version released on Apr. 4, 2013).

2.2 OpenStack

OpenStack [31] is a state-of-the-art open source cloud management stack, implemented in Python. It consists of several common services, such as a compute service group, an image service group, a network service, and several persistent storage services, as described in Section 2.1. Other OpenStack services in a typical cloud setting include an identity service for authenticating services and users and a dashboard service for providing a graphical interface to users and administrators. In addition, OpenStack relies on hypervisors installed on compute nodes (i.e., nodes where VMs run) for VM management.[3] OpenStack also uses a database service to store persistent states related to its managed cloud.

OpenStack employs two major communication mechanisms. Compute services use remote procedure calls (RPCs) conforming to the Advanced Message Queuing Protocol (AMQP) for internal communications within the service group. Other OpenStack services conform to the REpresentational State Transfer (REST) architecture and communicate with each other via the Web Server Gateway Interface (WSGI).

OpenStack uses the SQLAlchemy library to communicate with database backends, such as MySQL and SQLite. Interaction with hypervisors is abstracted into virtualization drivers. Specifically, OpenStack designs a common hypervisor-driver interface and implements drivers using common hypervisor APIs, such as libvirt and Xen API.

OpenStack services are implemented as green threads via the eventlet and greenlet libraries, which employ a user-level cooperative multithreading model: a thread runs non-preemptively until it relinquishes control. Upon thread yielding, a hub thread becomes 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 the same process.

3 Overview

This section presents the scope of our project, followed by a discussion of our design principles. We then present an overview of the components and the workflow of our fault-injection framework.

[3]Recent versions of OpenStack support experimental baremetal VM provisioning, which does not require hypervisor support.

3.1 Scope of the Project

We target fault-resilience-related programming bugs in OpenStack, because they affect OpenStack's intrinsic fault-resilience from the perspective of its design and implementation. Configuration bugs, in contrast, are considered faults in this paper. For example, an erroneous configuration may lead to network partitions, which are in turn used for fault injection in our framework. In addition, bugs that can only be manifested by a sequence of faults are not in the scope of this paper, due to the use of single-fault injections.

3.2 Design Principles

Our design builds on prior research in distributed systems tracing, fault injection, and specification checking. Instead of proposing a new fault-injection methodology, we discuss our experience in building an operational fault-injection prototype for OpenStack, following the design principles listed below.

Inject faults in service communications. Cloud-management stacks rely on the cooperation of services distributed to a cloud environment to fulfill their functionality. This cooperation requires fault-resilient communication mechanisms. Given the importance of service communications and the fast advances of sophisticated single-process debugging techniques, our fault-injection prototype targets service communications in OpenStack.

Expose domain-specific information. Domain knowledge has proven valuable for debugging, monitoring, and analyzing distributed systems. Sigelman *et al.* [36] 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, we expose OpenStack's high-level semantics to the fault-injection module and achieve high fault-injection efficiency by injecting faults to high-level communication flows instead of generic low-level events in runtime systems or operating systems.

Common cases first. It is extremely difficult and costly to thoroughly investigate every aspect of the fault resilience of cloud-management stacks. We thus focus on common cases, injecting common faults during the processing of OpenStack's most commonly used external APIs. The selection of faults is based on existing knowledge of the related work [19, 38], as well as our experience with large-scale production-level cloud systems. The selection of APIs is based on our experience with several experimental OpenStack deployments.

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

3.3 Components

Our fault-injection framework consists of three components: 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 synthesizes 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 behaviors (e.g., an HTTP status code returned to an external user) of OpenStack at the end of each fault-injection experiment comply to predefined specifications. Figure 1 presents an overview of the system and a high-level workflow, the latter discussed in the next section.

3.4 Workflow

The workflow consists of three stages: fault-free execution, fault injection, and specification checking. For a given external request, we start with fault-free execution of OpenStack, resulting in successful processing of the request. The logs produced during the fault-free execution are then fed to a parser to generate an execution graph (detailed in Section 5), characterizing communications among services. Combining the execution graph and a predefined fault specification, our framework generates a collection of test plans, 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 the collaboration of the logging and coordination module and the fault-injection module, with each experiment corresponding to a test plan. Experiment results are checked against predefined state and behavior specifications. We manually identify bugs from experiments causing specification violations.

Table 1: Communication log format

Attribute	Value/Explanation
Identifier	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

4 Logging and Coordination

After the overview of our framework, we start an in-depth discussion of its major components with the logging and coordination module.

4.1 Logging High-Level Semantics

Following the design principle of exposing domain-specific information to the fault-injection controller, our logging and coordination module explicitly maintains high-level semantics of several types of communications in its logs, including RPC, REST, database, hypervisor and shell operations.[4] Table 1 enumerates the key attributes in a communication log.

4.2 Unique Tag

A unique tag is created when OpenStack receives an external request. This tag is then propagated through OpenStack services along the request processing path. Recent versions of OpenStack employ similar techniques for tracing the request processing within service groups. In contrast, our framework assigns a system-wide unique tag to each external request and traces its processing within the scope of the entire stack. Unique tags facilitate the extraction of log entries related to a given external request. Otherwise, concurrent request processing would cause OpenStack to generate intertwined log entries and increase the complexity of log analysis. Although our study currently targets fault injection during the processing of a single external request, 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 updates of service liveness.

[4] Shell operations, which OpenStack performs on local nodes via the execution of local executables, are considered a special type of communication between a parent process (an OpenStack service) and a child process (which runs a target executable).

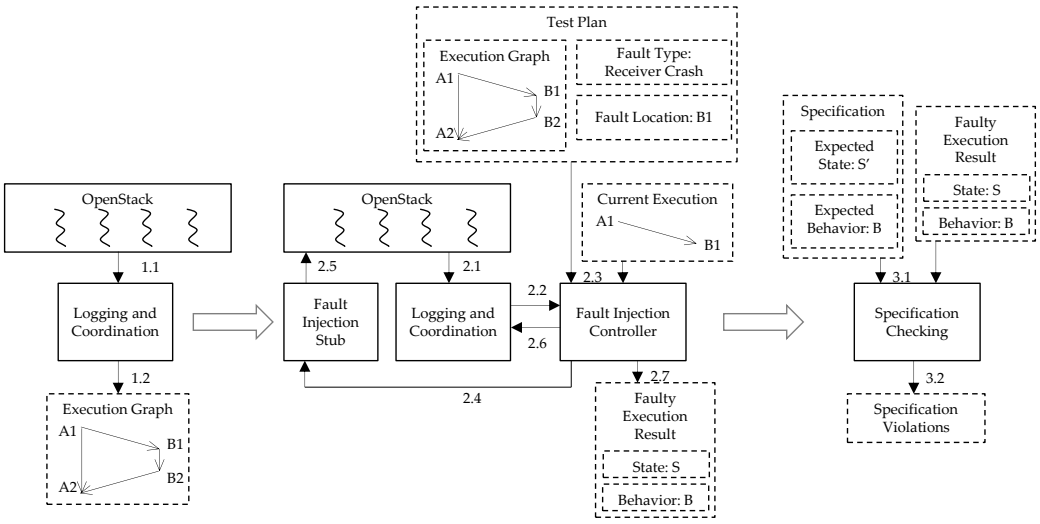

Figure 1: System overview and workflow. Boxes with a solid border represent OpenStack and major components of the framework. Boxes with a dashed border represent key non-executable objects in the framework. Three stages—fault-free execution, fault injection and specification checking—are separated by arrows with a gray border. Step 1.1: log OpenStack communications in a fault-free execution. Step 1.2: convert logs to an execution graph. Step 2.1: log communications in a fault-injection experiment and pause communicating entities during logging. Step 2.2: send logs to fault-injection controller. Step 2.3: make fault-injection decisions according to a test plan. Step 2.4: inform fault-injection stub of the fault-injection decisions. Step 2.5: inject faults. Step 2.6: resume execution. Step 2.7: collect results from fault-injection experiments. Step 3.1: check results against specifications. Step 3.2: report specification violations.

System-wide tag propagation requires modifications to the communication mechanisms in OpenStack. Specifically, we insert a new field representing unique tags in both request contexts used by OpenStack services and thread-local storage of those services. When a green thread of a service 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 freshly initialized one. The thread then associates this tag to all inter-service communications during its current activation.

Our framework cannot trace a unique tag once it propagates 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 as independent from the first one. So far, we have not encountered such cases in our study, and the logging mechanism suffices for our use.

4.3 Common Implementation Pattern

We implement the logging module by augmenting the communication layers between OpenStack and external services and libraries. In general, this module can be implemented at several layers along communication paths: inside OpenStack's core application logic where

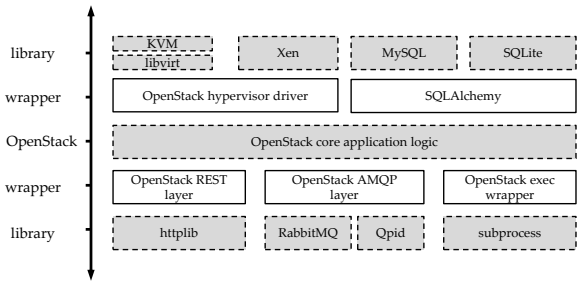

Figure 2: Illustration of logging instrumentation. White boxes with a solid border represent the instrumented layers for exposing high-level semantics in communications between OpenStack core logic and supporting services and libraries.

high-level communications are initiated, at OpenStack's wrapper layer of communication libraries, in communication libraries themselves, as well as in system libraries and interfaces. Our logging instrumentation resides at OpenStack's wrapper layer of communication libraries, as illustrated in Figure 2. The advantages of logging at this layer are two-fold. First, high-level semantics can be accurately exposed instead of being inferred at this layer. Second, logging at this layer incurs minimum implementation effort, because it consolidates communications originated from and destined for OpenStack ser-

vices. On the one hand, this layer is shared across OpenStack services and can thus use the same implementation to log communications from different services. On the other hand, this layer is implemented at the granularity of communication category (e.g., one implementation for all AMQP client libraries) and can thus use the same implementation to log communications from supporting services within each category, abstracting away details related to individual services.

Logging snippets are placed in the WSGI implementation of OpenStack and the eventlet 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 compliant 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 SQLAlchemy library for logging database operations. The compute service implements a helper function to perform shell operations on local hosts. We also augment 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 (e.g., Java SDK) or an operating system interface, our approach is incomplete in that it can only cover major communication mechanisms and is oblivious to other channels (e.g., a customized socket communication). Note, however, that in a well-designed cloud-management stack, the majority of inter-service communications are conducted via several well-defined interfaces, which have been instrumented in our study on OpenStack. In addition, system-level approaches usually lead to a significantly larger number of logs, degrading system performance and necessitating the use of advanced log parsing and inference logic in the fault-injection module. In our framework, we trade logging completeness for simplicity in exposing high-level semantics and potentially high logging performance.

4.4 RPC Trampolines

Logging RPC communications within the compute service group is implemented in part by modifying OpenStack's AMQP implementation. Such modifications, however, cannot form a complete picture of RPC communications, because the RPC caller and callee (or *producer* and *consumer* in AMQP terminology) is decoupled by an AMQP broker. An RPC cast from an OpenStack compute service is sent to an 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 completing the RPC cast. RPC calls are similar except that the return value goes through the AMQP broker as well.

For fine-grained control of fault injection, we intend to differentiate the 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 callee. A straightforward solution would be to extend our logging module to the implementation of the AMQP broker (either RabbitMQ or Qpid). This solution, however, 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 elevation of abstraction layers from the message transferring protocol (detailing packet formats) to a high-level RPC message with application semantics, incurring significant implementation overhead.

Our implementation leaves the AMQP broker intact and instead logs its activity via RPC trampolines—compute service proxies responsible for RPC forwarding. We create a trampoline for each compute service and modify 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 perspective of execution flow logging, RPC trampolines faithfully represent the AMQP broker, thus completing the picture of RPC communications.

4.5 Coordination

Besides generating detailed logs with high-level semantics, the logging snippets also serve as coordination points, synchronizing the execution of OpenStack 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.[5] The server makes fault-injection decisions, injects faults when necessary, and resumes the execution of the logged service by replying a "continue execution" message to the logging module. The use of logging module for coordination is also one major difference between our implementation and the existing notification mechanisms in OpenStack.

[5] Strictly speaking, the logging module blocks the execution of a green thread of the logged OpenStack service.

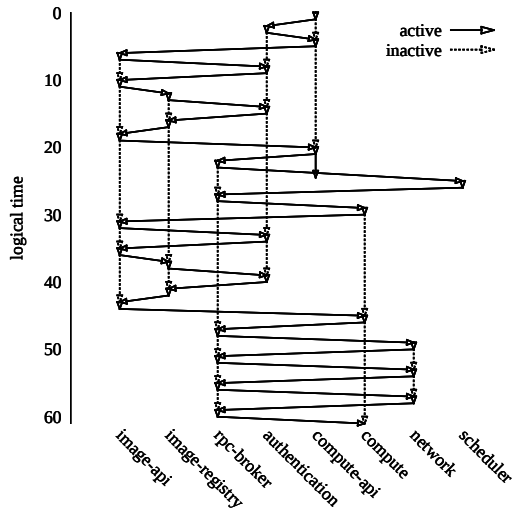

Figure 3: Illustration of the execution graph for VM creation, consisting of REST and RPC communication paths. Solid arrows represent active executions. Dashed arrows represent either idleness or waiting for the return of a function call.

5 Fault Injection

The fault-injection module is responsible for extracting execution graphs from logs, generating test plans, and injecting faults to OpenStack.

5.1 Execution Graphs

An execution graph depicts the execution of OpenStack during the processing of an external request. It is a directed acyclic graph extracted from logs of a fault-free request processing procedure, with each vertex representing a communication event in OpenStack. Each event is characterized by the communicating entity (e.g., an `image-api` service) and the type of communication (e.g., a REST request send operation). Edges represent causality among events. An edge connects two vertices (*i*) if they form a sender-receiver pair or (*ii*) if they belong to the same service and one precedes the other. Figure 3 shows a simplified execution graph related to a VM-creation request.

5.2 Test Plans

A test plan consists of three elements: an execution graph, a fault-injection location, and a fault type. We study two types of faults, namely, *server crash*[6] and *network partition*. These fault types are common causes of failures in cloud environments and are well-studied in the literature. Other types of faults, such as invalid inputs [30] and performance degradation [13], are not considered. Correlated faults are also common in real-world deployments but are not within the scope of this paper, due to the limitation imposed by our current single-fault-injection implementation.

[6]We use the terms *server* and *service* interchangeably, assuming that each service conceptually resides in a dedicated server host.

Procedure 1 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←TestPlan(exe_graph, node, fault)
      test_plans.append(new_plan)
return test_plans
```

Procedure 1 demonstrates the generation of test plans. Iterating over an execution graph, the algorithm exhaustively accounts for all fault types applicable to each vertex (e.g., a sender server crash targeting REST communications can only be injected to the vertices performing REST request/response send operations) and generates test plans accordingly. This procedure provides an opportunity for global testing optimization: *global* in that 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 within each cluster can then be selectively tested to reduce overall testing cost. Given that a fault-injection experiment in our 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 types of faults to be injected and the types of communications in which faults are injected. The fault specification functions as a test-case filter, enabling the design of a set of experiments focusing only on a specific fault type (e.g., sender server crashes) and/or a specific communication type (e.g., REST communications). The format of the specification can be extended to support other filters, such as confining fault injection to a subset of OpenStack services.

A test plan is fulfilled via the cooperation of a test server 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 corresponds. Then, the test server employs the same log parsing logic for execution graph generation to analyze 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 specified in the plan. And OpenStack runs until the request processing is completed.

5.3 Injection Operations

Server-crash faults are injected by killing relevant service processes via systemd. We modify configurations of systemd such that when it stops the relevant services, a `SIGKILL` signal is sent, instead of the default `SIGTERM` signal. Network-partition faults are injected by inserting iptables rules to service hosts that should be network-partitioned, forcing them to drop packets from each other.

6 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 behaviors of OpenStack.

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. Yet it is a key task for developing an effective specification-checking module. In effect, the coverage and the granularity of states and behaviors in the specifications determine the ability of the checking module to detect erroneous behaviors and states of the target system. Several approaches have been reported in the literature, including relying on developers to generate specifications [34], reusing system design specifications [19], and employing statistical methods [10].

To the best of our knowledge, OpenStack does not provide detailed and comprehensive specifications on system behaviors or state transitions during the processing of external requests.[7] The specifications we use in this study are generated based on our understanding of OpenStack, existing knowledge in fault-resilient system design, and first principles, which mirrors the developer-specification-generation approach.

Specifically, we manually generate specifications by inferring OpenStack developers' assumptions and 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 Pip [34]). In addition, such specifications are best-efforts, with a coverage constrained by our understanding of OpenStack. Nevertheless, the usefulness of such specifications is demonstrated by the identification of bugs reported in this paper.

[7] OpenStack does publish certain state-transition diagrams, such as transitions of VM states [32]. But they do not cover all the aspects we consider in our study.

Specification 1 VM State Stabilization Specification

```
query = select VM from compute_database
    where VM.state in collection(VM unstable states)
if query.count() = 0 then
    return Pass
return Fail
```

6.1 Specification-Generation Guidelines

Listed below are our specification-generation guidelines.

Do not block external users. OpenStack should not block external users due to faults during request processing.

Present clear error messages via well-defined interfaces. OpenStack should expose clear error states to external users via well-defined interfaces and avoid confusing information.

Stabilize system states eventually. 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.

6.2 Spec Checking Implementation

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 principle of using developer-familiar building blocks (cf. Section 3), we adopt a hybrid approach, applying declarative checking on persistent states stored in OpenStack's databases and imperative checking on the other states and behaviors of OpenStack. Specifically, database-related checks are implemented via the SQLAlchemy library. Others are implemented as generic Python scripts. This hybrid approach largely conforms to OpenStack's existing implementation: OpenStack adopts the same approaches to controlling its states and behaviors.

We implement specifications for OpenStack's states stored in databases and local filesystems, as well as OpenStack's behaviors, such as the HTTP status code returned to an external user after processing a request. We also specify the expected states of cloud platforms managed by OpenStack, such as the states of local hypervisors on compute hosts and the Ethernet bridge configurations.

Below we present three specification examples.

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
```

Specification 1 indicates the expectation of VM state stabilization. The state of a VM, after the processing of an external request (e.g., a VM creation) and a sufficient period of quiescence, should enter a stable state (e.g., the `ACTIVE` state, indicating that the VM is actively running) instead of remaining in a transient state (e.g., the `BUILD` state, indicating that the VM is in the process of creation). This is an example of using declarative checking on database states.

Specification 2 requires that, when a VM is 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. Specifically, we check 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 states maintained in the database of OpenStack image service regarding the image store in the local filesystem are in accordance with the view of 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, then its location attribute in the image database should be `local`, and vice versa. This specification shows a combined check on the views of a database and the filesystem of a service host.

7 Results

In this section, we discuss the bugs uncovered by our framework. We apply it 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.

7.1 Experiment Setting

Our study covers three OpenStack service groups: the identity service (keystone), the image service (glance), and the compute service (nova). For external services, our 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).

We configure the identity service to use UUID tokens for authentication. Regarding the image service, we configure it to use either a local filesystem or an HTTP server as its backend store. As for the compute service, we use QEMU as the backend hypervisor, controlled by the libvirt interface. In essex, we limit the reconnection from the Qpid client library to the backend broker to 1.

We run OpenStack services in VMs with 1 virtual CPU and 2GB memory. All OpenStack VMs run on an HP BladeSystem 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 instance. There are two exceptions to the above deployment 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.

7.2 Bugs in OpenStack

We test 11 external OpenStack APIs, inject 3848 faults, implement 26 specifications, detect 1520 violations and identify 23 bugs. Our results are summarized in Table 2. This table shows the significance of the bugs 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 sum of bugs in the table exceeds by far the number of distinct bugs, and the latter from the fact that the ratio of specification violations to bugs is greater than 1 for each API, which is also shown in Figure 4.

We classify the bugs into seven categories (cf. Table 3) and present an in-depth discussion of each category. Compared to our preliminary work [22], this paper contains newly-identified bugs in OpenStack and also discusses the evolution of bugs and fixes across the two OpenStack versions that we study, demonstrating both the improvement of OpenStack's fault resilience and the remaining issues.

Table 2: Summary of fault-injection results. 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 abbreviated to "Part."

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

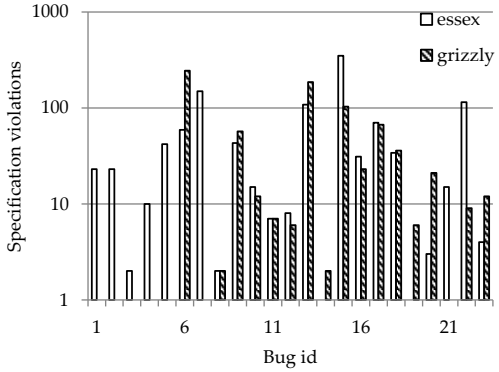

Figure 4: Bug distribution. The Y-axis shows the number of fault-injection experiments in which at least one specification violation leading to the identification of a given bug is detected.

Table 3: 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

7.2.1 Timeout

Timeout is a common mechanism in distributed systems to prevent 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 both OpenStack services and external supporting services. Setting correct timeout values, however, is known to be difficult. Given the numerous interactions and inter-dependencies among service groups and the variety of deployment environments, it is very difficult, if not impossible, for OpenStack to provide a comprehensive set of timeout values covering all execution paths that can potentially block the system.

For example, REST communications, one of the two major communication mechanisms used in OpenStack (cf. Section 2), fall out of the safety net of the timeout mechanism in the OpenStack essex version. Consequently, a service 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 our execution-graph-based fault injection over coarser-grained approaches [8]: such a bug can hardly be efficiently revealed by the latter due to the requirement on the synchronization between fault injection and send/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 timeout of 600 seconds. 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 resilience of OpenStack has been clearly enhanced due to the support of this critical timeout setting.

7.2.2 Periodic Checking

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

Our 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 state of the VM transits from `None` (i.e., non-existence) to `BUILD` (i.e., under construction) to `ACTIVE` (i.e., 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 a VM creation should cause the VM to enter a stable state (`ACTIVE` if the processing is successful and `ERROR` otherwise) in a timely manner, despite faults.

This bug can be fixed by a periodic checking logic that converts a VM's state from `BUILD` to `ERROR` if the related VM creation request is detected to have failed. A bug fix has been integrated into grizzly, conducting such a transition based on a configurable VM creation timeout value. This fix, albeit effective in preventing the prolonged `BUILD` state, leads to a state transition bug (cf. Section 7.2.3).

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 detection of VM related faults. This fault-handling logic fails when the VM related fault that it intends to log is itself a database fault (e.g., a database service crash fault during a VM creation). In such situation, the fault record is discarded, making it difficult for postmortem 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 state of the database service and merging those temporary messages into the database when possible.

7.2.3 State Transition

OpenStack maintains a large number of states in its databases, with complicated state-transition diagrams among them. Our 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. Thus, 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 disappearance of the fault.

The state transition from `ERROR` to `ACTIVE` without external event triggering can be confusing and problematic. According to our experience, upon receipt of a VM creation error, an external user is likely to issue another VM creation request. The bug fix for the previous periodic-checking-related problem thus induces a new state-transition bug, potentially creating more VMs than needed for an external user. We 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.

7.2.4 Return Code Checking

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 source of bugs even in well-organized projects [20, 27]. Our 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, however, the compute API service attributes such an error to the invalidity of the token, 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 digress from expected execution flows without being detected. For example, 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 successfully executed and, without checking its return code, proceeds to start the VM. As a result, the VM may lose network connectivity if a fault occurs during the execution of that command.

7.2.5 Cross-Layer Coordination

OpenStack relies on various supporting services to maintain its functionality and supports interaction with multiple services in each external service category via a set of layers of abstraction. Take the RPC messaging services as an example. OpenStack implements a unified layer for the AMQP protocol used for RPC communications, delegating operations to a lower layer implemented for a specific AMQP broker. The lower layer is a wrapper of the client-side library 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 multi-layer abstraction stack, albeit valuable and well-designed, imposes stringent requirements on cross-layer coordination. Incorrect interpretations of behaviors of one layer may lead to subtle bugs in another layer. For instance, the Qpid client library is configured to automatically reconnect to the AMQP broker after a connection disruption in essex. A threshold value is designed to control the maximum number of reconnections. A connection maintained by the client library resides in a temporary erroneous state until the threshold is reached, at which time the connection enters a permanent erroneous state. The OpenStack wrapper of the client library does not coordinate properly with the client library regarding the temporary-permanent error state transition, causing the wrapper to vainly retry a connection that has been marked as irrecoverable by the client library.

7.2.6 Library Interference

The extensive use of external libraries commonly found in large-scale software systems may lead to unexpected library interference. For example, OpenStack uses a patched version of Python standard library functions to support cooperative thread scheduling (cf. Section 2). Subtle incompatibility in the patched functions, however, can engender bugs that are hard to detect.

Take the communication between OpenStack and a 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 read-end of a pipe that is not yet ready for reading, thus permanently blocking an entire compute service.

7.2.7 Miscellaneous Bugs

Our 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 stale states in the connection object resulting from the previously failed `open`, causing all following retries to fail with an "already open" error message.

7.3 Resilience Evolution in OpenStack

Comparing the results of the two versions, we identify several interesting aspects in the evolution of OpenStack's fault resilience.

Timeout is necessary. Using timeouts to bound the execution of distributed operations is a well-known and important approach to fault-resilience improvement (e.g., Falcon [24]). As discussed in Section 7.2.1, the use of timeout for REST communications effectively solves the indefinite sender blocking issue in essex. Systematically configuring timeouts, however, remains an open question. For instance, different components in a REST communication flow (e.g., a WSGI pipeline) have different default timeout values. Moreover, 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 to further improve the fault resilience of OpenStack.

Return code matters. Carefully checking return codes enables prompt error detection. For example, in grizzly, during the processing of a VM deletion request, 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 (e.g., transiting the VM to the `ERROR` state), reducing the possibility of network resource leakage under faults.[8]

[8] We classify this bug in the periodic checking category, because it belongs to a general resource leakage issue that can be systematically addressed by the periodic checking mechanism. The fix in grizzly, in contrast, presents a simpler approach to solve this specific problem.

Keep cross-layer coordination simple. By simplifying the cross-layer coordination, OpenStack reduces the chances of bugs hiding between abstraction layers. For example, by disabling automatic reconnection in the Qpid client library and reserving full control only in its wrapper layer, OpenStack avoids the bug discussed in Section 7.2.5. In general, confining the decision-making logic regarding a specific aspect of a cloud-management stack to a single layer, instead of coordinating the decision making in different layers, is considered a good design practice.

8 Discussions

We now discuss our additional thoughts on this fault-resilience study.

8.1 Applicability to Real-World Issues

Our framework can be applied to solve fault-resilience issues related to cloud-management stacks in real-world deployments. For example, one of Amazon's cascading failures [4] was caused by a memory leakage bug on storage servers, which was triggered by the crash of a data-collection server, 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 our framework via the combination of a crash fault injected to the data-collection server and a specification on the memory usage of the data-reporting logic on storage servers. However, our framework needs to be extended for supporting injection of multiple faults and scaling specification-checking logic from an individual request to multiple requests. Such improvements will enable our framework to handle complicated issues with multiple root causes [3].

8.2 Generality of Our Study

One may consider the generality of our study in two aspects: the reusability of implementation and applicability of our findings to other cloud-management stacks.

As to implementation reusability, the fault-injection module and the specification-checking module of our 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 for our study across the two OpenStack versions. In general, such cross-version porting is straightforward. The logging and coordination module integrated in OpenStack contains about 800 lines of code, most of which are reusable across the two versions. Specifications need to be moderately adjusted to cope with minor semantic evolution (e.g., database schema changes).

Regarding our findings, the specific bugs and the related analysis presented in this paper are OpenStack-specific and cannot be generalized to other cloud-management stacks. The bug categories and the related fault-resilience issues, however, are general. Despite the numerous differences in the detailed design and implementation of cloud-management stacks, many of them [11, 15] share a common high-level scheme: they have similar service groups, rely on similar external supporting services, and employ similar communication mechanisms. Thus, our findings in this paper have the potential to shed light on fault resilience in other cloud-management stacks with a similar design.

8.3 Use of Execution Graphs

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

Our choice of the use of execution graph is mainly for future extensibility of our framework. On the one hand, an execution graph depicts the entire execution flow related to the processing of an external request and thus allows an intelligent test planner to conduct fault-injection experiments with better test coverage and less resource consumption (cf. Section 5.2). 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 [10]. We plan to investigate such uses in future.

8.4 Nondeterminism

In our framework, we require that the external request processing during a fault-injection experiment match its execution graph up to the fault-injection location. In other words, communication events observable by our framework in fault-free execution generally need to be deterministic. This requirement is satisfied in most of our experiments.

There are, however, 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 our experiments. Another source is environmental inputs. For example, the operations taken by a compute service for network setup on its local host depend on whether the compute service and a network service are co-located on the same host.

We handle such cases by annotating related execution graphs so that all the paths observed in fault-free execution are marked as valid. For other rare nondeterministic events, we effectively bypass them by re-executing failed fault-injection experiments.

9 Related Work

Cloud-management stacks are a type of distributed system. Our study of the fault resilience of this layer benefits from prior research on fault resilience of distributed systems in general and cloud systems in particular. In this section, we compare our work with existing fault-resilience studies. We also compare our execution path extraction and specification checking with similar techniques employed in distributed systems debugging and failure detection.

Fault-resilience studies. Fault injection is commonly used to study the fault resilience of cloud systems. FATE is a fault-injection framework targeting the recovery logic in cloud applications, systematically exploring failure scenarios with multiple-failure injection [19]. Dinu and Ng injected crash faults to components on Hadoop's compute nodes and studied their effects on application performance [12]. In contrast, our framework targets cloud-management stacks and examines the recovery logic by conducting single-fault injection during the processing of external requests. Similar to the target of FATE, we study the functionality and correctness of recovery logic, which is difficult to be made correct [21].

Failure as a Service (FaaS) is proposed as a new generic service to improve the fault resilience of cloud applications in real deployments [18]. We suggest an alternative approach for cloud-management stacks, presenting an integrated fault-injection framework with domain knowledge. By combining the two, cloud-management stacks may enhance fault resilience by better balancing the cost and coverage of fault injection.

Model checking (e.g., Modist [38]) is another common approach to examining the fault resilience of distributed systems. 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, however, necessitates the extensive use of domain knowledge in order to make it practical to check highly complicated operations (e.g., VM creation) in cloud-management stacks.

Execution path extraction. Execution paths have been extensively used for workload modeling [5, 37], performance [1, 35] and correctness debugging [10, 16, 17], and evolution analysis [7] in distributed systems. Such information is either exposed via special logging modules (at either system or user level) or inferred in sophisticated post-processing. Applying existing knowledge to our framework, we extract execution paths related to external request processing via user-level logging, which explicitly exposes high-level semantics.

Specification checking. Prior research has explored various approaches to specification checking in distributed systems. Regarding specification expression, both imperative approaches [25, 26] and declarative approaches [19, 34] have been studied. In our framework, we employ a hybrid approach in expressing specifications on the states and behaviors of cloud-management stacks, combining imperative checking and declarative checking. Similar combinations have been used to query and analyze distributed trace events [14]. Regarding specification generation, most existing approaches, including ours, require developers to implement specifications. Recent advances in filesystem-checker testing leverage the characteristics in the checkers to automatically generate implicit specifications [9]. The applicability of similar approaches to cloud-management stacks remains an open question.

10 Conclusions

In this paper, we conducted a systematic study on the fault resilience of OpenStack. We designed and implemented a prototype fault-injection framework that injects faults during the processing of external requests. Using this framework, we uncovered 23 bugs in two OpenStack versions, classified them into seven categories, and presented an in-depth discussion of the fault-resilience issues, which must be addressed in order to build fault-resilient cloud-management stacks. Our future work consists of refining and automating specification generation logic and exploring potential use of execution graphs in fault-resilience-related areas.

Acknowledgments

We thank Benjamin Reed (our shepherd) and the anonymous reviewers for their comments and suggestions. The work reported in this paper was supported in part by the US Air Force Office of Scientific Research under Grant No. FA9550-10-1-0393.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the nineteenth ACM symposium*

on Operating systems principles, SOSP '03, pages 74–89, New York, NY, USA, 2003. ACM.

[2] Amazon. Amazon elastic compute cloud (Amazon EC2). http://aws.amazon.com/ec2/. Retrieved in September 2013.

[3] 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.

[4] 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.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[6] 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.

[7] S. A. Baset, C. Tang, B. C. Tak, and L. Wang. Dissecting open source cloud evolution: An openstack case study. In *USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud'13. USENIX Association, 2013.

[8] C. Bennett and A. Tseitlin. Chaos monkey released into the wild. http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html. Retrieved in September 2013.

[9] 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.

[10] 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 - Volume 1*, NSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[11] CloudStack. Apache CloudStack: Open source cloud computing. http://cloudstack.apache.org/. Retrieved in September 2013.

[12] 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.

[13] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *2013 ACM Symposium on Cloud Computing*, SOCC'13, New York, NY, USA, 2013. ACM.

[14] U. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Trans. Comput. Syst.*, 30(4):13:1–13:35, Nov. 2012.

[15] Eucalyptus. The Eucalyptus cloud. http://www.eucalyptus.com/eucalyptus-cloud/iaas. Retrieved in September 2013.

[16] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: a pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design and implementation*, NSDI'07, pages 271–284, Berkeley, CA, USA, 2007. USENIX Association.

[17] 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.

[18] 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-2011-87*.

[19] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, 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.

[20] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, 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.

[21] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, P. Bodik, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Workshop on Hot Topics in Operating Systems*, HotOS XIV. USENIX Association, 2013.

[22] 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.

[23] 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.

[24] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and 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.

[25] 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.

[26] 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, pages 19–19, Berkeley, CA, USA, 2007. USENIX Association.

[27] P. D. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Trans. Comput. Syst.*, 29(4):11:1–11:38, Dec. 2011.

[28] Microsoft. Microsoft Hyper-V server 2012. `http://www.microsoft.com/en-us/server-cloud/hyper-v-server/default.aspx`. Retrieved in September 2013.

[29] Microsoft. Summary of Windows Azure service disruption on Feb 29th, 2012. `http://blogs.msdn.com/b/windowsazure/archive/2012/03/09/summary-of-windows-azure-service-disruption-on-feb-29th-2012.aspx`. Retrieved in September 2013.

[30] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[31] OpenStack. OpenStack open source cloud computing software. `http://www.openstack.org/`. Retrieved in September 2013.

[32] OpenStack. Virtual machine states and transitions. `http://docs.openstack.org/developer/nova/devref/vmstates.html`. Retrieved in September 2013.

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

[34] 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.

[35] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: black-box performance debugging for wide-area systems. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 347–356, New York, NY, USA, 2006. ACM.

[36] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[37] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and 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 Annual technical conference*, USENIX'09, pages 19–19, Berkeley, CA, USA, 2009. USENIX Association.

[38] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and 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.