Università degli Studi di Napoli Federico II

Ph.D. Thesis

in

Information Technology and Electrical Engineering

Fault Injection for Cloud Computing Systems

From Failure Mode Analysis To Runtime Failure Detection

Pietro Liguori

Tutor: Prof. Domenico Cotroneo
Co-Tutor: Prof. Roberto Natella

Coordinator: Prof. Daniele Riccio

XXXIV Ciclo

Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Abstract

Nowadays, *cloud computing* systems are considered an attractive solution for running services with high-reliability requirements, such as in the telecom and healthcare domains, and have gained huge attention over the past decades because of continuously increasing demands. These systems consist of processes distributed across a data center, which cooperate by message passing and remote procedure calls. They are very complex, as they typically consist of software components of millions of lines of code, which run across dozens of computing nodes.

It is very difficult to avoid software bugs when implementing the rich set of services of cloud computing systems. As a result, many high-severity failures have been occurring in the cloud infrastructures of popular providers, causing outages of several hours and the unrecoverable loss of user data. Therefore, the high-reliability requirements of such systems are still too far to reach.

Fault-injection techniques, i.e., the deliberate insertion of faults into an operational system to determine its response, offer an effective solution to improve the reliability of the systems. These techniques are also important to identify *failure modes* of the infrastructure, in order to improve the detection and the recovery capabilities of the entire system. Although fault injection has reached a level of maturity that it is routinely used in many real-world systems, its adoption in cloud computing infrastructures raises several issues that have to be addressed.

First, the user needs to inject realistic faults to be emulated in the experiments when targeting complex and distributed systems. The problem of *defining a fault model* becomes more difficult when injecting *software faults* (i.e., design and/or programming defects), since they depend on a variety of technical and organizational factors, including the programming language, the software development process, the maturity of the system,

the expertise of developers, and the application domain.

Second, the *execution of the fault injection experiments* in cloud systems is not trivial. Given the complexity of such systems (millions of LoCs), the fault injection campaigns can easily reach thousands of experiments due to the combination of the number of realistic fault types to inject and the space of the fault points where to inject. To assess the effects of the injection, failure data should be collected during every experiment by guaranteeing independence among the executions (e.g., by performing the system clean-up, the restart of the services, the revert of the database, etc.). In the light of these considerations, the execution of the fault-injection experiments should ideally be fully automated and supported by a complete fault injection workflow.

Finally, the *identification of the failure symptoms*, a key step towards improving the reliability of cloud systems, often relies on the knowledge, the experience, and the intuition of human analysts since existing fault injection solutions provide limited support to the analyst for understanding what happened during an experiment. Unfortunately, manual analysis is too difficult and time-consuming, because of i) the *high volume of messages* generated by large distributed systems that the human analyst needs to scrutinize; ii) the *non-determinism* in distributed systems, in which the timing and the order of messages can unpredictably change even if there is no failure, which introduces noise in the analysis, and increases the effort of the human analyst to pinpoint the failure (i.e., to discriminate the anomalies caused by a fault from genuine variations of the system); iii) the use of *"off-the-shelf" software components*, either proprietary or open-source (such as application frameworks, middleware, data stores, etc.), whose events and protocols can be difficult to understand and to manually analyze.

The first contribution of this thesis is a fault-injection tool-suite for cloud systems [54]. The tool-suite is designed to be programmable and highly usable, by performing fault injection campaigns with customized fault types. The tool has been used to empirically analyze the impact of high-severity failures in the context of a large-scale, industry-applied case study [58] and for subsequent analysis that aims to better understand the failure nature of these systems and to design run time monitoring strategy, which is capable of improving the failure detection capabilities. As for the failure nature, we know that these systems fail in complex and unexpected ways. For instance, recent outages reports showed that failures escape fault-tolerance mechanisms, due to unexpected combinations of events and of interactions among hardware and software components, which were not anticipated by the system designers. These failures are especially problematic when they are *silent*, i.e., not accompanied by any explicit failure notification, such as API error codes, or error entries in the logs. This behavior hinders the timely detection and recovery, lets the failures silently propagate through the system, and makes the traceback of the root cause more difficult, and recovery actions more costly (e.g., reverting a database state). Therefore, understanding how the system can fail (i.e., the *failure mode analysis*) and promptly identifying the failure at runtime (i.e., *runtime failure detection*) are crucial activities to improve the fault-tolerance mechanisms and define proper recovery strategies of cloud systems.

As for the failure mode analysis, ***the thesis proposes a novel algorithm to identify failure symptoms and error propagation analysis*** [56]. The algorithm adopts a probabilistic model and revealed to be very accurate in identifying the anomalies, i.e., failure symptoms, in noisy execution traces of the system, by significantly reducing the false alarms (i.e., genuine variations are not mistaken for failure symptoms) without discarding true anomalies (i.e., actual anomalies caused by a fault are not missed). In order to analyze failures from the set of anomalies and find recurring failure patterns, ***this thesis adopts two machine learning***

approaches: one based on unsupervised learning algorithms [53] and, the other, based on deep learning ones [55]. The former approach combines *clustering* with the proposed anomaly detection algorithm in order to automatically identify the failure classes among large sets of fault injection experiments. The approach achieved high accuracy ($\sim 90\%$ purity) under different conditions, but at the cost of manually setting the weights of the features, which requires a deep knowledge of the system internals, and efforts to best tune them concerning the specific workload. The latter approach, instead, overcomes the challenges of noise and complexity of the feature space by leveraging deep learning for unsupervised machine learning. The approach saves the manual efforts spent on feature engineering, by using an autoencoder to automatically transform the raw failure data into a compact set of features. The results demonstrate that the proposed approach can identify clusters with accuracy similar, or in some cases, even superior, to the fine-tuned clustering, with a low computational cost.

The empirical analysis pointed out that cloud systems often exhibit a *non-fail-stop* behavior, in which it continues to execute despite inconsistencies in the state of the virtual resources due to missing or incorrect error handlers. From these results, the ***thesis proposes a lightweight approach to runtime verification tailored for the monitoring and analysis of cloud computing systems*** [59]. The approach defines a set of monitoring rules from correct executions of the system in order to specify the desired system behavior. The rules are then synthesized in a runtime monitor that verifies whether the system's behavior follows the desired one. Any runtime violation of the monitoring rules gives a timely notification to avoid undesired consequences, e.g., non-logged failures, non-fail-stop behavior, failure propagation across sub-systems, etc. The approach reveals to be very effective, achieving a failure detection rate of over 90% and improving the fault-tolerance mechanisms of the system.

Acknowledgments

I wish to thank my advisor, Professor Domenico Cotroneo, for his constant support in my life. The numerous and lengthy conversations I had with him were a constant source of inspiration and broad perspective. My debt to him goes beyond this dissertation. I wish to acknowledge my co-advisor, Professor Roberto Natella, for his available assistance during my studies. His influence is visible throughout this whole dissertation. My sincere thanks also go to Ing. Luigi De Simone, whose constant support and motivation were essential to overcome the difficulties faced during my Ph.D. program.
I am deeply indebted to the external evaluators, Professor Bojan Cukic and Professor Leonardo Mariani, for their constructive criticisms and substantial suggestions in the writing of this dissertation. Professor Bojan Cukic deserves special thanks for his valuable support during my abroad period at the University of North Carolina at Charlotte.
I would like to express my sincere gratitude to Professor Stefano Russo for always being valuable guidance throughout my studies. I would also like to offer my special thanks to Dr. Samira Shaikh and my colleague Erfan Al-Hossami for their invaluable advice and continuous support during my whole abroad period. I wish to extend my sincere thanks to all members of the DESSERT research group at the University of Naples Federico II for - but not limited to - our precious collaboration over several years.
Last but not least, I wish to thank my family and all people close to me for the support, help, and motivation in all difficult circumstances of life. Thank you for always being there for me.
This dissertation is dedicated to my mother Lina.

Naples, Italy, March 14, 2022 Pietro Liguori

This page intentionally left blank.

Contents

This page intentionally left blank.

List of Acronyms

The following acronyms are used throughout this text.

AMPQ Advanced Message Queuing Protocol

API Application Programming Interface

AST Abstract Syntax Tree

CLI Command Line Interface

COUNT Counted-Events Rules

CPU Central Processing Unit

CSV Comma Separated Value

DBMS Database Management System

DEC Deep Embedded Clustering

DEPL New Deployment Workload

DNN Deep Neural Network

DSL Domain Specific Language

EPL	Event Processing Language
FMEA	Failure Mode and Effects Analysis
HMM	Hidden Markov Model
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LCS	Longest Common Subsequence
LOC	Lines of Code
ODC	Orthogonal Defect Classification
MIFS	Missing IF Statement
MFC	Missing Function Call
MP3	Moving Picture Expert Group-1/2 Audio Layer 3
MR	Monitoring Rules
NET	Network Management Workload
OCC	Occurred-Events Rules
OCM	OpenStack Coverage Mechanisms
ORD	Ordered-Events Rules (ORD)
OS	Operating System
REST	Representational State Transfer

ROC	Receiver Operating Characteristic
RPC	Remote Procedure Calls
SEQ	Sequence-based Approach
SMP	Symmetric Multiprocessor System
SQL	Structured Query Language
SSH	Secure Shell Protocol
STO	Storage Management Workload
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
VM	Virtual Machine
VMM	Variable-order Markov Model
WPF	Wrong Parameter Function

This page intentionally left blank.

List of Figures

List of Tables

Chapter 1

Fault Injection in Cloud Computing Systems

1.1 Reliability Issues in Cloud Computing Systems

As computer systems grow increasingly complex, they also become increasingly likely to have faults, stemming from their requirements specification, their design, their implementation, or their operating environment [6]. This is the case of the *cloud computing systems*.

Cloud computing is formally defined as a "*model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*" [156].

Cloud systems are considered an attractive solution for running services with high-reliability requirements, such as in the telecom and healthcare domains [236, 129, 72, 264], and have gained huge attention over the past decades because of continuously increasing demands [238]. These systems consist of processes distributed across a data center, which cooperate by message passing and remote procedure calls (e.g., through message queues and REST API calls). They are quite complex, as they typically consist of software components of millions of lines of code (LoC), which run across dozens of computing nodes, such as in the case of OpenStack (see Appendix A), the most widely deployed open-source cloud software in the world [176].

The OpenStack cloud computing platform is developed in Python language and is mostly deployed as infrastructure-as-a-service (IaaS) in both public and private clouds where virtual servers and other resources are made available to users. It provides abstractions and APIs for programmatically creating, destroying, and snapshotting virtual machine instances; attaching and detaching volumes and IP addresses; configuring security, network, topology, and load balancing settings; and many other services to cloud infrastructure consumers. The system consists of several independent parts, named *services* (also referred to as *components*, *subsystems*, or *projects*). The three most important services of OpenStack [68, 227] are: (i) the Nova subsystem, which provides services for provisioning instances (VMs) and handling their life cycle; (ii) the Cinder subsystem, which provides services for managing block storage for virtual instances; and (iii) the Neutron subsystem, which provides services for provisioning virtual networks, including resources such as *floating IPs*, *ports* and *subnets* for instances. In turn, these services include several components (e.g., the Nova service includes *nova-api*, *nova-compute*, etc.), which interact through message queues internally to OpenStack. Figure 1.1 shows the logical architecture of OpenStack [175].

The communications intra and inter-services are based on Remote Procedure Calls (RPC). OpenStack projects use an open standard for messaging middleware known as Advanced Message Queuing Protocol (AMQP).

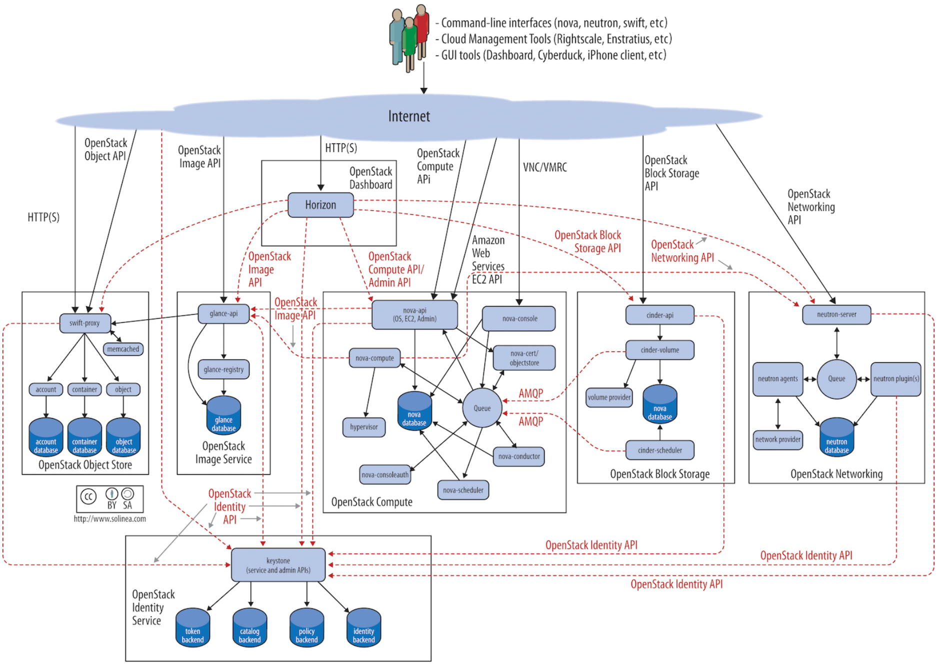


Figure 1.1. OpenStack Logical Architecture.

This messaging middleware enables the OpenStack services that run on multiple servers to talk to each other. Moreover, each OpenStack project has a related client project that includes Python API bindings and a command-line interface (CLI). The OpenStack clients enable the user to access the project API through easy-to-use commands. OpenStack APIs are RESTful APIs and use the HTTP protocol. They include methods, URIs, media types, and response codes. Users can run the commands from the command line or include the commands within scripts to automate tasks.

It is very difficult to avoid software bugs when implementing the rich set of services of cloud computing systems: let's think that, at the time of writing, the OpenStack project codebase consists of million lines of code

(LoC) [30, 184], which implies thousands of residual software bugs even under the most optimistic assumptions on the bugs-per-LoC density [155, 239]. As a result of these bugs, many high-severity failures have been occurring in cloud infrastructures of popular providers, causing outages of several hours and the unrecoverable loss of user data [140, 164, 99, 100].

To prevent severe failures, software developers invest efforts in mitigating the consequences of residual bugs. Examples are defensive programming practices, such as assertion checking and logging, to timely detect an incorrect state of the system [146, 84] and for providing to system operators useful information for quick troubleshooting [268, 267, 83]. Another important approach to mitigate failures is to implement fault containment strategies. Examples are *i)* interrupting a service as soon as a failure occurs (i.e., a *fail-stop* behavior), by turning high-severity failures, such as data losses, into lower-severity API exceptions that can be gracefully be handled [35, 237, 188]; *ii)* notifying the cloud management system and operators about the failures through error logs, so that they can diagnose issues and undertake recovery actions, such as restoring a previous state checkpoint or backup [249, 85]; *iii)* separating system components across different domains to prevent cascading failures across components [134, 10, 105].

1.2 Addressing Reliability Issues with Fault Injection

To get data about software failures and improve fault-tolerant mechanisms, the *fault-injection* technique is a valuable solution. Fault injection is formally defined as "*the process of introducing faults in a system in order to assess its behavior and to measure the efficiency (coverage, latency, etc.) of fault tolerance mechanisms*" [8, 247, 48], and is a fundamental technique to ascertain the fault-tolerance properties of the systems. This technique

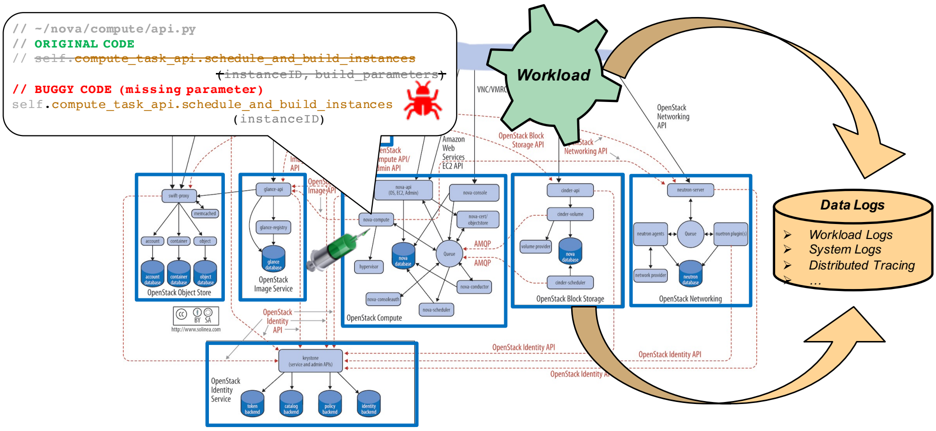


Figure 1.2. Overview of a fault-injection experiment in OpenStack.

consists of the deliberate insertion of *faults* (such as resource exhaustion, software bugs, connection loss, etc.) into a software system in a controlled experiment in order to trigger failures.

Figure 1.2 shows an overview of a fault-injection experiment in OpenStack. The injection consists of the mutation of the original code of the Nova service with a buggy code by removing the input parameter `build_parameters` from the target function. This fault, typically named *missing parameter*, is one of the most common bugs in OpenStack [58]. The target system is then exercised with a *workload*, i.e., a set of directives used to stress the system by simulating a user (or a group of users) that performs service requests and triggers the injected fault during the experiments. Finally, *data logs* such as the logs produced by the workload and the system, the messages exchanged among services, etc., are collected during the experiments to scrutinize the effects of the injection and assess the fault-tolerance mechanisms of the system.

This technique has reached a level of maturity that it is routinely used to reveal failures in real-world systems, including cloud computing software such as key-value data stores and distributed computing frameworks

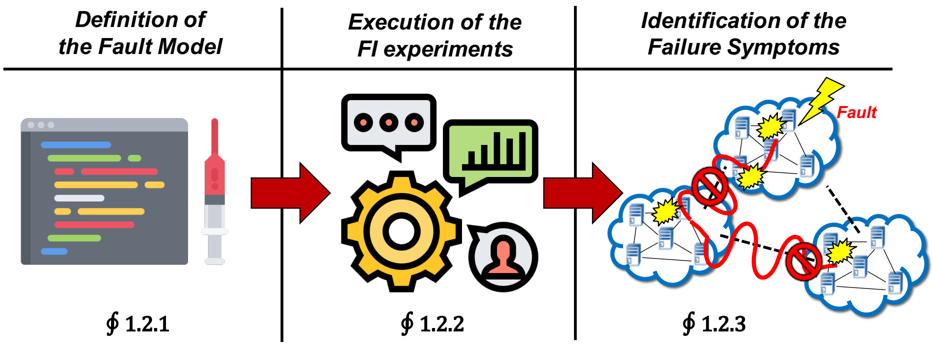


Figure 1.3. Issues of Fault Injection in Cloud Computing Systems.

(e.g., Cassandra, ZooKeeper) [98], entire cloud computing services (e.g., streaming services deployed by Netflix) [168] and infrastructures (e.g., IaaS providers such as Amazon) [213]. Nevertheless, its adoption in cloud systems still raises several issues that have to be addressed. Figure 1.3 summarizes these issues.

First, the user needs to inject realistic faults to be emulated in the experiments when targeting complex and distributed systems. The problem of defining a fault model becomes more difficult when injecting *software faults* (i.e., design and/or programming defects [14]), since they depend on a variety of technical and organizational factors, including the programming language, the software development process, the maturity of the system, the expertise of developers, and the application domain [112, 111]. Second, the *execution of the fault injection experiments* in cloud systems is not trivial. Given the complexity of such systems (millions of LoCs), the fault injection campaigns can easily reach thousands of experiments due to the combination of the number of realistic fault types to inject and the space of the fault points where to inject. To assess the effects of the injection, failure data should be collected during every experiment by guaranteeing independence among the executions (e.g., by performing the system clean-up, the restart of the services, the revert of the database, etc.). Finally, cloud computing systems are often exposed to unpredictable failure conditions due to failures that can propagate across several components or layers of the system (e.g., storage, virtual network, compute instances, etc.) in complex ways, leading to cascading effects (*failure propagation*). Hence, the *identification of the failure symptoms* becomes a key step towards improving the reliability of the systems. However, this analysis often relies on the knowledge, experience, and intuition of human analysts since existing fault injection solutions provide limited support to the analyst for understanding what happened during an experiment.

1.2.1 Definition of the Fault Model

The ***fault model*** entails the definition of three main aspects, namely *what* to inject (i.e., which kind of fault), *when* to inject (i.e., the timing of the injection), and *where* to inject (i.e., the part of the system targeted by the injection) [48, 147, 122, 130, 60]. The *what* can be represented by bit-flips [110]; program exceptions for amplifying unit- and integration-tests [1, 121]; node crashes, network partitions and latency for networked and distributed systems [123, 98]. The *when* and *where* to inject are sampled from a (large) space of possibilities across time and program locations.

Although the *hardware fault-injection* has been proved to provide an effective means to assess the fault-tolerance mechanisms of safety-critical software [108, 223], the focus of this thesis is on injecting software faults since we are interested in assessing the severity of failures caused by software bugs in cloud computing infrastructures. Indeed, mitigating the severity of software failures caused by residual bugs is a relevant issue for high-reliability systems [62], yet it still represents an open research challenge. Since software bugs are *human* mistakes in the source code, the traditional fault-tolerance strategies for hardware and network faults often do not apply. For example, if a service is broken because of a regression bug, then retrying to execute the service API with the same inputs would result again in a failure; a retrial would only succeed in the case that the software bug is triggered by a transient condition, such as a race condition [93, 94, 37]. If recovery is not possible, the failed operation must be necessarily aborted and the user should be notified [169, 159] so that the failure can be handled at a higher level of the business logic. For example, the end-user can skip the failed operation, or put on hold the workflow until the bug is fixed.

Despite the variability of software faults across systems, the existing software fault injection tools are based on a predefined, fixed software fault model, that cannot be easily customized by users. Most of the existing tools adopt the *Orthogonal Defect Classification* (ODC), proposed in the '90s (e.g., bugs in initialization, algorithm, interfaces, etc.), or derived the fault model from bug samples of third-party open-source and commercial projects [48, 75].

A modern software fault injection tool should be able to modify the fault model for the following reasons. First, a typical necessity in industry, which arises when a critical failure occurs, is to introduce regression tests against the fault that caused the failure, to assure that the same failure cannot occur again [266]. Second, to preserve the efficiency of the fault injection campaign, it is important to avoid injecting bugs that are unlikely to affect a system; e.g., some classes of faults may be prevented by testing and static analysis policies adopted by the company [28]. Third, as the scale and the complexity of systems increase, the need for a more sophisticated fault model grows. For instance, modern distributed systems, such as cloud applications, have to integrate a variety of components, including third-party and open-source ones, and they have to deal with high volumes of traffic. For these systems, the user needs to inject more variants of design/programming defects than those reported in the literature, including performance bottlenecks, resource management issues, lack of interoperability between components, security issues, failed updates, etc., and adapt these faults to their projects. In general, the potential users of software fault injection want to tune the fault model so that it reflects their experience and expectations about failures. All these use cases require a greater degree of control over the fault model than what is provided by existing fault injection tools.

1.2.2 Execution of the FI Experiments

The combination of the number of realistic fault types to inject and the space of the fault points where to inject make the ***execution of the fault injection experiments*** in cloud computing systems a difficult and time-consuming task. Moreover, during the execution of the experiments, the human analyst collects data (e.g., system logs, workload logs, events, etc.) from the target system to analyze the effects of the injection. This analysis requires independence among the executions to relate failure to the specific bug that caused it.

Therefore, a fault injection tool should ideally provide a complete fault injection workflow, which assists test engineers at applying software fault injection in these systems. Hence, a fault-injection tool should provide full automation in the execution of the experiments, by collecting failure data and guaranteeing independence among the executions (e.g., by performing the system clean-up, the restart of the services, the revert of the database, etc.). Moreover, to further increase the usability of the fault injection technique, the tool should locate the fault injection points in the system, i.e., a statement (or group of statements) in the source code where to inject the faults configured in the fault model, and allow the users to select such statements, according to their needs, and provide the configuration for the workload used to exercise the system.

A Fault Injection Tool-suite

To address the previous limitations, this dissertation presents a new fault injection tool designed to be *programmable*, enabling users to add and customize a software fault model. By using the tool, users can specify new software fault models using a *domain-specific language* (DSL) for fault injection. The tool compiles the specification into an automatically-generated fault injector. Finally, the generated fault injector is applied to the software-under-test to generate fault-injected versions and to execute experiments. To achieve better usability, the tool presented in this thesis is provided as *software-as-a-service*, and includes a workflow for configuring the fault load and the workload to i) fully automate the execution of experiments using container-based virtualization and parallelization, and to ii) perform failure data analysis. The tool also provides the automatic analysis of the fault-injection experiments in terms of service failures, logging, and recovery, and includes advanced features, such as the graphical representation of the fault-injection experiments.
The tool has been used to *empirically analyze the impact of high-severity failures* in the context of OpenStack cloud computing platform, and for subsequent analysis that aims to better understand the failure nature of these systems and to design run time monitoring strategy, which is capable of improving the failure detection capabilities.

1.2.3 Identification of the Failure Symptoms

Interpreting the outcome of fault injection experiments, i.e., the ***failure symptoms***, is a key step towards improving reliability. In particular, the analyst needs to assess the effects of the fault on the target system, and how they lead to a service failure, as they provide indications on where to improve fault tolerance mechanisms.

However, cloud computing systems are often exposed to unpredictable failure conditions [87] due to failures that can propagate across several components or layers of the system (e.g., storage, virtual network, compute instances, etc.) in complex ways, leading to cascading effects (***failure***

propagation) that make recovery actions more problematic.

In the case of *temporal propagation*, the analysis identifies *latent* failures in the system, which manifest as a failure only after a while. Temporal propagation represents an opportunity for improving error handling: for example, by detecting the data affected by these failures with more thorough consistency checks, and by preventing that they turning into failures through software rejuvenation; or, if the failure could not be recovered, by enforcing a *fail-stop* behavior, i.e., a service is stopped and a failure is notified to error handlers and/or to the users as soon as it occurs, in order to reduce its severity. In the case of *spatial propagation*, a failure propagates across several components or layers of the cloud system, which increases the risk of cascading failures, and makes recovery more problematic (e.g., only recovering the last component in the propagation chain does not correct errors in the previous components). Spatial propagation can be prevented by blocking failures at components' interfaces, by looking at execution traces from fault injection experiments.

Therefore, identifying and analyzing the propagation of the failures is an important activity to design more effective recovery actions. The current state of practice is to detect the failure symptoms (e.g., service unavailability, performance degradation) by monitoring the quality of service during the fault injection test; more sophisticated solutions detect failures by monitoring properties expressed with formal specifications, such as finite state machines [67], relational logic [98], and special-purpose languages [210].

This analysis too often relies on the knowledge, experience, and intuition of human analysts since existing fault injection solutions provide limited support to the analyst for understanding what happened during an experiment [166]. Indeed, once a service failure has been triggered by fault injection and detected by monitoring mechanisms, a human analyst still needs to analyze the chain of events (e.g., messages) that occurred among the location where the fault/error is injected and the component that experiences the service failure.

Unfortunately, manual analysis is too difficult and time-consuming, because of i) the *high volume of messages* generated by large distributed systems that the human analyst needs to scrutinize; ii) the *non-determinism* in distributed systems, in which the timing and the order of messages can unpredictably change even if there is no failure, which introduces noise in the analysis, and increases the effort of the human analyst to pinpoint the failure (i.e., to discriminate the anomalies caused by a fault from genuine variations of the system); iii) the use of *"off-the-shelf" software components*, either proprietary or open-source (such as application frameworks, middleware, data stores, etc.), whose events and protocols can be difficult to understand and to manually analyze.

Motivating Example

To better understand the research problem, we discuss an example of a fault-injection experiment on the OpenStack cloud computing platform, shown in a simple graphical representation in Figure 1.4.

This representation shows remote procedure calls that are made for communication in the distributed system. These calls are displayed as intervals over the timeline of the experiment. We consider both API calls between the client and the OpenStack REST APIs (the topmost sequence of calls), and internal API calls within OpenStack, which are performed by Nova, Neutron, and Cinder using message queues (the other three sequences of calls). To see the effects of the injected fault, we show two subplots: the former shows a normal execution of the system (*fault-free execution*), in which no fault is injected; the latter shows the execution of the system when a fault is injected in the Nova subsystem (*faulty execution*). Since both executions are performed under the same conditions (i.e., same software and hardware configuration, same workload, etc.), any

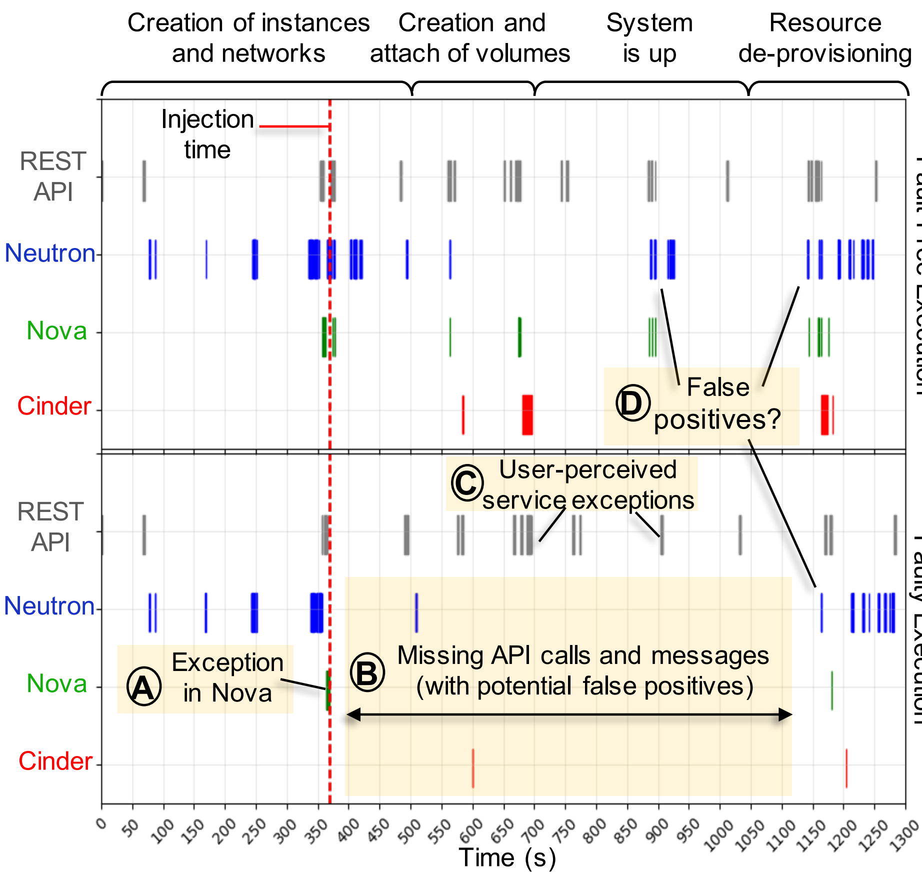


Figure 1.4. A graphical representation of a fault-injection experiment.

deviation between the faulty and the fault-free execution is considered an anomaly due to the injected fault.

The workload used in this example first creates several resources (i.e, networks, instances, volumes, etc.), then it performs basic operations to stimulate the different components of the system (e.g., attaching a volume to an instance, checking the connectivity, reboot an instance, etc.) before cleaning up the created resources. All these operations are performed by invoking the OpenStack APIs.

One of these API calls is an asynchronous request for creating a new VM instance. After the API call ends, OpenStack Nova takes a few minutes for creating and initializing the instance. During these operations, we inject a Python exception to force a failure ((A)). Figure 1.4 points out that there are several API calls in the fault-free execution that are missing in the faulty execution ((B)) since the injected fault causes a failure that affects several OpenStack subsystems over a relatively long period. Indeed, Nova does not complete the initialization of the VM instance due to the fault, leaving the VM in an inactive state. Moreover, the OpenStack Neutron subsystem was also unable to attach the virtual network to the VM instance. Later on (i.e., after about five minutes) the workload client experienced a service exception when calling the API of the Cinder subsystem, which manages storage volumes in OpenStack ((C)). Consequently, the workload could not attach the volume to the VM instance. Both Nova and Neutron do not raise any API exception, but the failure only became apparent to the client when invoking the API of the Cinder subsystem.

The analysis of a fault-injection experiment can be inaccurate due to the non-determinism of the API calls in distributed systems. For example, the Neutron subsystem uses asynchronous messages and polling for distributing state updates across its components, thus such messages could be easily misclassified as anomalies. Moreover, due to the asynchronous nature of several APIs, it is difficult to properly identify whether API calls order does not matter (i.e., is due to non-determinism) or should be carefully taken into account because of the failure. In this point, Figure 1.4 also highlights events that could be false positives ((D)), both among the fault-free and the faulty execution. Thus, we need to understand if the differences among such two executions are due to the non-determinism in the system (i.e., they are not related to the failure) or not (i.e., they are actually anomalies). Considering the false positives makes the debugging more difficult and cumbersome for the human analyst, as each execution may include hundreds of API calls to analyze with only a few ones relevant for understanding the failure.

The experiment in Figure 1.4 is an example of both temporal and spatial propagation: the issue propagates both across subsystems (from Nova to Neutron and Cinder) and across time since the client perceives the failure only after a relatively long time. This behavior is problematic from the point of view of high availability, and thus of defining proper recovery actions, as the propagation delay also increases the time-to-detect and the time-to-recover the failure. Furthermore, the longer the propagation chain the more difficult will be for a developer to reason about how to best tolerate the fault, e.g., whether to manage the fault in Nova, Neutron, and/or Cinder and at which time to manage the fault during the workload. For example, the API could return a more timely notification of the failure to the client, either by introducing a callback mechanism in the Nova API that creates the instance or by returning an error from other API calls to Nova or Neutron.

Anomaly Detection Algorithm to Failure Symptoms Identification

To provide automated support for analyzing failures triggered by fault injection, this thesis dissertation introduces an approach that extends fault injection, by combining it with black-box tracing and anomaly detection algorithm for failure analysis. The driving idea is to train a *probabilistic model* of the events in the distributed system under test under *fault-free* conditions, by using variable-order Markov Models for analyzing event sequences. Afterward, the system is tested with fault injection, and event traces are collected under *faulty* conditions. The faulty event traces are analyzed with anomaly detection by using the probabilistic model, and the anomalous events are reported to the human analyst for understanding how to avoid failures. The approach avoids the human analyst manually inspecting thousands of events by automatically identifying the few relevant events that are related to the injected fault while discarding noisy, uninteresting events.

1.3 From Failure Mode Analysis to Runtime Failure Detection

It is well known that failures in cloud computing systems might have huge financial implications for the companies involved and their customers. Unfortunately, cloud-computing systems fail in complex and unexpected ways. For instance, recent outages reports showed that failures escape fault-tolerance mechanisms, due to unexpected combinations of events and of interactions among hardware and software components, which were not anticipated by the system designers [87, 107]. These failures are especially problematic when they are *silent*, i.e., not accompanied by any explicit failure notification, such as API error codes, or error entries in the logs. This behavior hinders the timely detection and recovery, lets the failures silently propagate through the system, and makes the traceback of the root cause more difficult and recovery actions more costly (e.g., reverting a database state) [56, 58].

Therefore, understanding how the system can fail (i.e., the ***failure mode analysis***) and promptly identifying the failure at runtime (i.e., ***runtime failure detection***) are crucial activities to improve the fault-tolerance mechanisms and define proper recovery strategies of the cloud computing systems.

1.3.1 Failure Mode Analysis

The analysis of the failure modes in complex systems such as cloud computing is a difficult and time-consuming task. In the current fault-injection approaches, analysts write failure specifications before the experiments. Then, they look for occurrences of these failures within the experimental data [253]. For example, the most sophisticated approaches check formal specifications over events and outputs, by using finite state machines [67], temporal logic predicates [10], relational logic [98], and special-purpose languages [210]. Since these specifications are mostly based on prior knowledge and experience of system designers about failures, they are not meant for discovering new, unknown failure modes of a distributed system, which are missed by the failure specifications. Moreover, writing failure specifications is a time-consuming and cumbersome task, which makes fault injection less applicable in practice.

Moreover, when considering complex cloud systems, it is typical to perform a large number of experiments (e.g., several thousand), since these systems include tens of processes and nodes and millions of lines of source code in which faults can be injected. For each experiment, the system generates high volumes of log files (up to hundreds of MBs) and long execution traces (e.g., thousands of events per trace). Thus, it is not feasible in practice for the analyst to analyze all of these data in a reasonable amount of time.

Machine Learning Approaches to Failure Mode Analysis

In order to analyze failures from the set of anomalies and find recurring failure patterns, this thesis adopts two machine learning approaches: one based on unsupervised learning algorithms and, the other, based on deep learning ones.

The former approach combines *clustering* with the proposed anomaly detection algorithm in order to automatically identify the failure classes among large sets of fault injection experiments. The approach achieved high accuracy under different conditions, but at the cost of manually setting the weights of the features, which requires a deep knowledge of the system internals, and efforts to best tune them concerning the specific workload.

The latter approach, instead, overcomes the challenges of noise and complexity of the feature space by leveraging deep learning for unsupervised machine learning. This approach saves the manual efforts spent on feature engineering, by using an autoencoder to automatically transform the raw failure data into a compact set of features.

1.3.2 Runtime Failure Detection

The empirical analysis performed with the proposed fault injection tool-suite pointed out that cloud systems often exhibit a *non-fail-stop* behavior, in which it continues to execute despite inconsistencies in the state of the virtual resources due to missing or incorrect error handlers. This analysis suggests the need for strategies of runtime failure detection in order to promptly test the availability of virtual resources.

To perform the runtime monitoring of the cloud application operations, an operation's log is the main source of information for monitoring the operation behavior [82]. Yet, there are several severe limitations in log analysis [172] since logs are usually low-level, noisy, and they lack information of changes to resource states. A further key technique to identify the failure at runtime is represented by *runtime verification strategies*, which perform redundant, end-to-end checks (e.g., after service API calls) to assert whether the virtual resources are in a valid state [18]. For example, these checks can be specified using temporal logic and synthesized in a runtime monitor [66, 41, 272, 206], e.g., a logical predicate for a traditional OS can assert that a thread suspended on a semaphore leads to the activation of another thread [10]. Runtime verification is now a widely employed method, both in academia and industry, to achieve reliability and security properties in software systems [17].

However, the application of these strategies to perform the runtime detection of failures in cloud computing systems is very challenging [272, 82]. First, the public service of a cloud system usually receives thousands of user requests in a very short time, which may be handled in a complex process. For example, Figure 1.5 summarizes the complex request flow, in terms of different requests among the services, for provisioning an instance in OpenStack [215, 192]. The user's request for the instance creation is handled by the Compute (Nova) component and involves the interaction between multiple components inside the system, such as Keystone for the

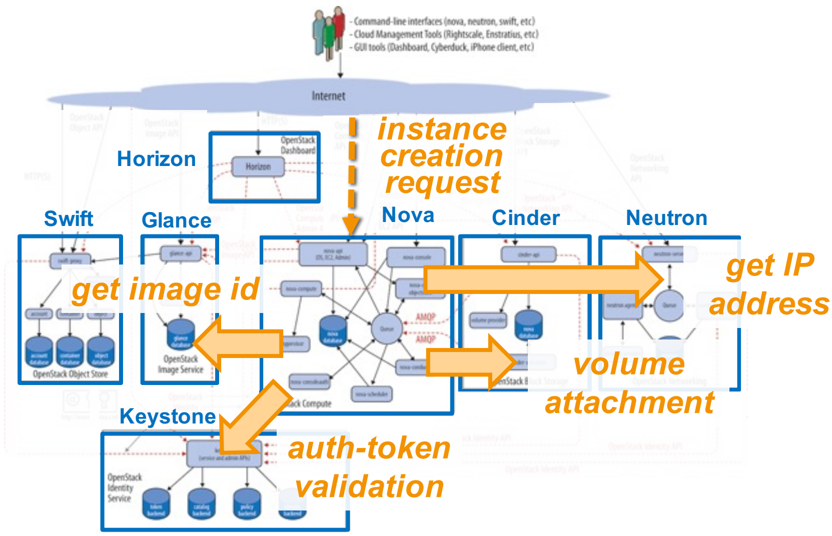


Figure 1.5. Request flow for provisioning instance in OpenStack.

client authentication, Neutron (Quantum) for networking, Cinder for block storage, and Glance for images. Therefore, massive trace data would be produced in cloud systems, which is a real problem for real-time monitoring. Moreover, the specification methods are usually not sufficient to accurately and flexibly express the monitoring requirements due to the non-deterministic behavior of these systems.

Event Stream Processing Approach to Runtime Verification

To address these difficulties and perform runtime detection of the failures, this thesis dissertation proposes a lightweight approach to runtime verification tailored for the monitoring and analysis of cloud computing systems. The approach uses a non-intrusive form of tracing of *events* (e.g., messages) in the system under test and builds a set of lightweight monitoring rules from correct executions of the system in order to specify the desired system behavior.
The approach analyzes the executions of the system in *fault-free* conditions to define a set of *failure monitoring rules*. These rules encode

the expected, correct behavior of the system, and detect a failure if a violation occurs. We synthesize the rules in a runtime monitor that verifies whether the system's behavior follows the desired one. Any runtime violation of the monitoring rules gives a timely notification to avoid undesired consequences, e.g., non-logged failures, non-fail-stop behavior, failure propagation across subsystems, etc.
The approach does not require any knowledge about the internals of the system under test and it is especially suitable in multi-tenant environments or when testers may not have a full and detailed understanding of the system.

1.4 Thesis Structure

The thesis is structured as follows.

■ Chapter 2 provides a systematic review of the literature to show an overview of previous and related works.

■ Chapter 3 introduces a new fault injection tool, *ProFIPy*, for Python software. The tool is designed to be *programmable*, to enable users to specify their software fault model, using a *domain-specific language* (DSL) for fault injection. Moreover, to achieve better usability, *ProFIPy* is provided as *software-as-a-service* and supports the user through the configuration of the fault load and workload, failure data analysis, and full automation of the experiments using container-based virtualization and parallelization. The tool also provides the automatic analysis of the fault-injection experiments in terms of service failures, logging, and recovery, and includes advanced features, such as the graphical representation of the fault-injection experiments to help the user to understand what happened during a failure.

■ Chapter 4 investigates the impact of failures in the context of widespread

OpenStack cloud management system, by performing fault injection and by analyzing the impact of the resulting failures in terms of fail-stop behavior, failure detection through logging, and failure propagation across components. The analysis points out that most of the failures are not timely detected and notified; moreover, many of these failures can silently propagate over time and through components of the cloud management system, which call for more thorough run-time checks and fault containment.

■ Chapter 5 proposes a novel approach that joins fault injection with anomaly detection to identify the symptoms of failures and analyze the propagation of the errors. We evaluated the proposed approach in the context of the OpenStack cloud computing platform and show that the approach can significantly improve the accuracy of failure analysis in terms of false positives and negatives, with a low computational cost.

■ Chapter 6 introduces a new paradigm (*fault injection analytics*) that applies unsupervised machine learning on execution traces of the injected system, to ease the discovery and interpretation of failure modes. We evaluated the proposed approach in the context of fault injection experiments on the OpenStack cloud computing platform, where we show that the approach can accurately identify failure modes with a low computational cost.

■ Chapter 7 presents a novel approach for analyzing failure data from cloud systems, to relieve human analysts from manually fine-tuning the data for feature engineering. The approach leverages Deep Embedded Clustering (DEC), a family of unsupervised clustering algorithms based on deep learning, which uses an autoencoder to optimize data dimensionality and inter-cluster variance. We applied the approach in the context of the OpenStack cloud computing platform, both on the raw failure data and in combination with an anomaly detection pre-processing algorithm. The results show that the performance of the proposed approach, in terms of purity of clusters, is comparable to, or in some cases even better than manually fine-tuned clustering described in Chapter 6, thus avoiding the need for deep domain knowledge and reducing the effort to perform the analysis.

■ Chapter 8 proposes an approach to runtime verification, for monitoring and failure detection of cloud computing systems. The approach uses a non-intrusive form of tracing of events in the system under test and derives a set of lightweight monitoring rules from correct executions of the system in order to specify the desired system behavior. We evaluated the approach in the OpenStack cloud management platform showing that the approach can be applied with high failure detection coverage.

List of Publications

The following previously published material has been, in parts verbatim, included in this thesis.

- D. Cotroneo, L. De Simone, A. Di Martino, P. Liguori, and R. Natella, "Enhancing the Analysis of Error Propagation and Failure Modes in Cloud Systems", *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2018, pp. 140-141. DOI: 10.1109/ISSREW.2018.00-13

- D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, "How bad can a bug get? an empirical analysis of software failures in the OpenStack cloud computing platform", In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*, 2019, Association for Computing Machinery, New York, NY, USA, pp. 200–211. DOI: 10.1145/3338906.3338916

- D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, "FailViz: A Tool for Visualizing Fault Injection Experiments in Distributed Systems", *2019 15th European Dependable Computing Conference (EDCC)*, 2019, pp. 145-148. DOI: 10.1109/EDCC.2019.00036

- D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, "Enhancing Failure Propagation Analysis in Cloud Computing Systems," *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 139-150. DOI: 10.1109/ISSRE.2019.00023

- D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, "ProFIPy: Programmable Software Fault Injection as-a-Service", *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 364-372. DOI: 10.1109/DSN48063.2020.00052

- D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, "Fault Injection Analytics: A Novel Approach to Discover Failure Modes in Cloud-Computing Systems", in *IEEE Transactions on Dependable and Secure Computing*, September 2020. DOI: 10.1109/TDSC.2020.3025289

- D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and A. Scibelli, "Towards Runtime Verification via Event Stream Processing in Cloud Computing Infrastructures", In *Service-Oriented Computing – ICSOC 2020 Workshops (ICSOC 2020).* Lecture Notes in Computer Science, vol 12632. Springer, Cham. DOI: 10.1007/978-3-030-76352-7_19

- D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, "Enhancing the analysis of software failures in cloud computing systems with deep learning", in *Journal of Systems and Software*, Volume 181, 2021, 111043, ISSN 0164-1212. DOI: 10.1016/j.jss.2021.111043

The following publications are related to the different aspects covered in this thesis but have not been included.

- P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh, "Shellcode_IA32: A Dataset for Automatic Shellcode Generation", in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, 2021, pp. 58-54. DOI: 10.18653/v1/2021.nlp4prog-1.7

- P. Liguori, E. Al-Hossami, V. Orbinato, R. Natella, S. Shaikh, D. Cotroneo, and B. Cukic, "EVIL: Exploiting Software via Natural Language", *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, 2021. DOI: 10.1109/ISSRE52982.2021.00042

- P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh, "Can We Generate Shellcodes via Natural Language? An Empirical Study", in *Automated Software Engineering*, 2022. DOI: 10.1007/s10515-022-00331-3

- P. Liguori, C. Improta, S. De Vivo, R. Natella, B. Cukic, and D. Cotroneo, "Can NMT Understand Me? Towards Perturbation-based Evaluation of NMT Models for Code Generation", *The 1st Intl. Workshop on Natural Language-based Software Engineering (NLBSE 2022)*, 2022. Accepted for Publication

Chapter 2

State-of-the-Art

This chapter provides a systematic review of the literature to show an overview of previous and related works.

2.1 Fault Injection Adoption in Cloud Systems

The fault injection is widely used for evaluating fault-tolerant cloud computing systems. Well-known solutions in this field include *Fate* [98] and its successor *PreFail* [123] for testing cloud software (such as Cassandra, ZooKeeper, and HDFS) against faults from the environment, by emulating at API level the unavailability of network and storage resources, and crashes of remote processes. Similarly, Ju *et al.* [124] and *ChaosMonkey* [168] test the resilience of cloud infrastructures by injecting crashes (e.g., by killing VMs or service processes), network partitions (by disabling communication between two subnets), and network traffic latency and losses. Other fault models for fault injection include hardware-induced CPU and memory corruptions, and resource leaks (e.g., induced by misbehaving guests). *CloudVal* [197] and Cerveira *et al.* [38] applied these fault models to test the isolation among hypervisors and VMs. Pham *et al.* [198]

applied fault injection on OpenStack to create signatures of the failures, in order to support problem diagnosis when the same failures happen in production. The fault model is the main difference that distinguishes our work from previous studies. Most of them assess software robustness with respect to *external* events (e.g., a faulty CPU, disk, or network). In other studies, fault injection has been simulating software failures through process crashes and API errors, but this is a simplistic form of software bugs, which can cause generate more subtle effects (such as incorrect logic and data corruptions, as pointed out by bug studies). In this dissertation, we injected *software bugs* inside components by mutating their source code, to deliberately force their failure, and to assess what happens in the worst case that a bug eludes the QA process and gets into the deployed software.

We remark that previous work on mutation testing [120] also adopted code mutation, but with a different perspective than ours, since we leverage mutations for evaluating software fault tolerance. This dissertation contributes to this research field by showing new forms of analysis based on the injection of software faults (fail-stop behavior, logging, failure-propagation). The same approach is also suitable to other systems of similar size and complexity of OpenStack (e.g., where the need for coordination among large subsystems raises the risk for non-fail-stop behavior and failure propagation).

2.1.1 Analysis of Bugs and Failures of Cloud Systems

Previous studies on the nature of outages in cloud systems analyzed the failure symptoms reported by users and developers, and the bugs in the source code that caused these failures.

Among these studies, Li *et al.* [140] analyzed failures of Amazon Elastic Compute Cloud APIs and other cloud platforms, by looking at failure reports on discussion forums of these platforms. They proposed a new taxonomy to categorize both failures (content, late timing, halt, and erratic failures) and bugs (development, interaction, and resource faults). One of the major findings is that the majority of the failures exhibit misleading content and erratic behavior. Moreover, the work emphasizes the need for counteracting "development faults" (i.e., bugs) through "semantic checks of reasonableness" of the data returned by the cloud system. Musavi *et al.* [164] focused on API issues in the OpenStack project, by looking at the history of source-code revisions and bug fixes of the project. They found that most of the API changes are meant to fix API issues and that most of the issues are due to "programming faults". Gunawi *et al.* analyzed outage failures of cloud services [100], by inspecting headline news and public post-mortem reports, pointing out that software bugs are one of the major causes of the failures. In a subsequent study, Gunawi *et al.* analyzed software bugs of popular open-source cloud systems [99], by inspecting their bug repositories. The bug study pointed out the existence of many "killer bugs" that are able to cause cascades of failures in subtle ways across multiple nodes or entire clusters; and that software bugs exhibit a large variety, where "logic-specific" bugs represent the most frequent class. Most importantly, the study remarks that cloud systems tend to favor availability over correctness: that is, the systems attempt to continue running despite the bugs cause data inconsistencies, corruptions, or low-level failures are detected, in order to avoid that users could perceive outages, but putting at risk the correctness of the service.

These studies give insights into the nature of failures in cloud systems and point out that software bugs are a predominant cause of failures. While these studies rely on evidence that was collected "after the fact" (e.g., the failure symptoms reported by the users), we analyze failures in a controlled environment through fault injection, to get more detailed information on the impact on the integrity of virtual resources, error logs, failure propagation, and API errors.

2.1.2 Uncertainty in Fault Injection Experiments

Uncertainty is a key aspect in fault injection experimentation since the behavior of a complex system depends on many factors that are difficult or impossible to control. This problem is exacerbated when fault-injection is used in cloud computing, where the human analyst has to deal with the non-deterministic nature of such systems. State-of-the-art provides several works that addressed this problem by applying solutions based on statistical techniques. Several studies leveraged the statistical models to model the probability of failures during hardware fault-injection experiments [11, 226, 190]. Arlat *et al.* [9] proposed a solution that brings together the coverage evaluation of the fault coverage and the occurrence of the faults to estimate the dependability of the complex fault-tolerant systems. By estimating the probabilities of the failure modes of the system, Voas *et al.* [248] presented a solution to reduce the uncertainty of whether different software faults impact the behavior of the system. To assess the quality of the measurements in terms of uncertainty, repeatability, resolution, and intrusiveness, Bondavalli *et al.* [33, 34] applied the principles of *measurement theory*. In AMBER project [253], the authors used data mining to identify the factors (i.e., workloads, the fault types, etc.) with the highest impact on the performance and availability of the target system. Gulenko *et el.* [97] introduced an anomaly detection approach that leverages an online clustering method to define the normal behavior of monitored components. Wu *et al.* [255] proposed a method that applies a dependency graph and an autoencoder to identify the causes of the performance degradation in the microservices of the cloud. Both previous works evaluated the proposed solutions by injecting performance anomalies in the cloud computing system. The Loki tool [39] addressed the problem of injecting faults in controlled global states of distributed systems since it is difficult due to the lack of a global clock and communication delays (e.g., between a central controller and a local injector). The tool performs a post-experiment analysis of event traces collected from nodes, using an off-line clock synchronization algorithm, to identify whether injections hit the desired state, and repeats the experiments only when needed.

All these studies are based on the assumption that failures can be accurately and automatically identified. We consider this dissertation complementary to them since it provides novel techniques for identifying the failure modes of the target system.

2.2 Fault Modeling

The idea of software fault modeling for fault injection purposes was initially investigated by Chillarege et al. [45], who analyzed a dataset of failures of IBM OS and DBMS products at users' sites [234, 235], to identify recurring patterns in the faults that caused them, and to inject the same patterns by corrupting program data and code, e.g., as in the *FINE* tool [126]. In the same period, they also introduced the *Orthogonal Defect Classification* (ODC) [46, 44], where one the goals was to classify software fault data into orthogonal categories, including *Initialization*, *Algorithm*, *Interface*, *Checking*, and *Synchronization* defects. Christmansson and Chillarege [48] proposed to inject software faults by following the statistical distribution of OS faults across these categories, such that the injected faults are representative of faults experienced by the users of the OS in the field. Similarly, Chen and colleagues [170, 171] defined a software fault model for OSes based on data for the IBM MVS and Tandem GUARDIAN90 OS products [234, 134], and used this fault model to emulate realistic OS and DBMS crashes, to assess crash recovery mechanisms. This fault model was later merged in the well-known fault injection tool of the *Nooks* project [237].

The work on the *G-SWFIT* fault injection technique by Madeira and colleagues [74, 75] aimed to define a *generic* software fault model (i.e., not tailored for a specific system) that could go beyond specific OS and DBMS products, and that could be used for injecting faults even without any field failure data for the specific system under testing. To define such a generic fault model, they analyzed a sample of bugs in several open-source projects in C [74, 75] and Java [21, 220], and looked for bug-fixes (e.g., program elements that were changed to fix the bug, such as new assignments, control flow constructs, function calls, etc.) which were recurring more than the norm, and which occurred consistently across all of the projects. Based on this analysis, they defined a software fault model with 13 fault types, covering 60% of the sample of bugs in the open-source projects [75]. This fault model was used in several other tools, including *SAFE* [61], *HSFI* [243], and *FastFI* [224]. However, these tools focus on a fixed software fault model, with no ability to customize the injected faults according to the specific needs of a project or company.

Winter *et al.* [250] and Giuffrida *et al.* [91] showed that implementing a new fault model in a tool takes both significant programming effort, e.g., in terms of SLOC and other metrics, and considerable expertise in program analysis and transformation, e.g., to implement a software fault injection tool using the *LLVM* compiler suite, which is not affordable for the average user of a fault injection tool.

2.3 Fault Injection Tools

Some tools provide a limited ability to customize the fault model with a lower effort: among them, the *FIDLFI* tool [4] provides the user with a configuration language to control the *trigger* of fault injection (i.e., instructions and paths that trigger the injection), *target* (i.e., instruction source and destination registers to inject), and *action* (e.g., corruption, freeze, delay, etc.). The *FAIL-FCI* tool [106] provides a fault injection language tailored for grid systems, which specifies protocol states and nodes to inject (e.g., node crashes). *PreFail* [123] and *FATE* [98], which inject crashes and I/O API errors, allow the user to write *policies* in Python to select the location and timing of potential injections by considering the allocation of processes across nodes and racks (e.g., network partitions between different racks), and the coverage of injectable points in the software-under-test. *LFI* [151], which injects errors at C library calls, allows the user to configure what functions and error codes should be injected, and when to trigger the injection (e.g., when a specific function appears in the stack frame) using an XML configuration file. The commercial tools *QA Systems Cantata* [219] and *Razorcat TESSY* [104] provide user-friendly GUIs to select a source-code statement to inject, similarly to breakpoints in a GUI debugger.

It is important to note that these tools do not support rich software fault models as in *G-SWFIT* and derivatives, as they only provide limited control on *what* to inject, e.g., they focus on API and library calls, register accesses, nodes, etc., but do not allow to create new fault types for injecting arbitrary changes to the software. The proposed *ProFIPy* tool provides a new language to gain a higher degree of control, where the user can specify transformation rules about which parts of the program to inject, in terms of program elements (e.g., assignments, expressions, control flow directives, and combinations of thereof), and how to transform these program elements into faulty ones.

2.4 Event Trace Analysis in Distributed Systems

Research studies on debugging distributed systems lead to a variety of *profiling* techniques to pinpoint bugs and performance bottlenecks. Aguilera *et al.* [2] collect black-box network traces of communications between hosts, in order to analyze requests as they move through the system (e.g., web requests across the tiers of a web application). Their approach infers causal paths of the requests, by tracing call pairs (i.e., request messages, and their corresponding responses), and by analyzing statistical correlations. However, this approach focuses on synchronous (RPC-style) interactions between components, and it is not meant to analyze asynchronous interactions (i.e., the server immediately replies to a request, before issuing causally-related requests and performing more work) and rare events (as the approach focus on the most frequent interactions).

Magpie [16] and Pinpoint [43] reconstruct causal paths by using more sophisticated tracing infrastructures, by tracing detailed events at the OS-level and the application server level. The tracing tags incoming requests with a unique *path identifier*, and associates resource usage throughout the system with that identifier. This fine-grain tracing approach does not rely on statistical inference and can provide high accuracy, but it also brings considerable complexity, which makes it difficult to deploy it in practice, especially when considering cloud computing infrastructures with many heterogeneous components (e.g., OSes, middleware, interpreters, etc.).

Gu *et al.* [95] proposes a methodology to extract knowledge on distributed system behavior of request processing without source code or prior knowledge. The authors construct the distributed system's component architecture in request processing and discover the heartbeat mechanisms of target distributed systems.

Pip [210] is a system for automatically checking the behavior of a distributed system against programmer-written expectations about the system. Pip provides a domain-specific expectations language for writing declarative descriptions of the expected behavior of large distributed systems and relies on user-written annotations of the source code of the system to gather events and to propagate path identifiers across chains of requests. This approach provides flexibility for the analysis but requires access to the source code, and non-negligible efforts to annotate it.

More recent studies contributed to tools resembling debuggers, but for distributed systems. Pensieve [270] is an approach for producing the path to failure, in a similar way to delta debugging: it combines static analysis, and re-execution of the system with iteratively-refined logging, in order to reconstruct the intermediate path backward from the failure to the user inputs and events that cause the failure. Friday [88] is a distributed debugger that allows developers to replay a failed execution of a distributed system, and to inspect the execution through breakpoints, watchpoints, single-stepping, etc., at the global-state level. ShizViz [27] is an interactive tool for visualizing execution traces of distributed systems, which allows developers to intuitively explore the traces and to perform searches; moreover, the tool provides support for comparing distributed executions with a pairwise comparison, even if without probabilistic techniques to filter-out benign variations due to non-determinism.

Recent fault injection solutions addressed cloud computing systems. The *Fate* [98] tool, and its successor *PreFail* [123], simulate disk failures, network partitions, and crashes of nodes, by exploring multiple occurrences of faults during the same experiment, to test recovery procedures more thoroughly (e.g., at tolerating further network/disk faults occurring during recovery). To address the combinatorial explosion of experiments, these tools adopt user-programmable policies to prune redundant experiments (e.g., injections in symmetric states or in paths that were already covered). Ju *et al.* [124], *ChaosMonkey* [168], and *Jepsen* [127] test the resilience of cloud infrastructures by injecting crashes (e.g., by killing VMs or service processes), network partitions (by disabling communication between two subnets), and network traffic latency and losses. *CloudVal* [197] and Cerveira *et al.* [38] use fault injection (CPU and memory corruptions, resource leaks) to test the isolation among hypervisors and VMs. Pham *et al.* [198] applied fault injection on OpenStack to create signatures of the failures, in order to support problem diagnosis when the same failures happen in production. Once fault injection reveals a failure, in most cases it is the tester's responsibility to look at what happened during the test, and come up with an interpretation of the issue and a potential solution to make the system more fault-tolerant.

There are several approaches to identify anomalies in the cloud based on models derived from fault-free executions, also in combination with fault injection. Qiang *et al.* [96] presented an unsupervised failure detection method using an ensemble of Bayesian models that characterizes normal execution states of the system and detects anomalous behaviors. The method estimates the probability distribution of runtime performance data collected by health monitoring tools when cloud servers perform normally. Sauvanaud *et al.* [222] described a new approach to detect Service Level Agreements (SLAs) violations and preliminary symptoms of SLAs violations by means of machine learning models and based on monitoring data. Mariani *et al.* [150] presented a lightweight and precise approach to predict failures and locate the corresponding faults in multi-tier distributed systems. The approach blends anomaly-based and signature-based techniques to identify multi-tier failures that impact on performance indicators, with high precision and low false positive rate. Islam *et al.* [117] described a machine-learning-based anomaly detector used for proactive detection of problems in the IBM Cloud Platform's components and showed that the detector can capture anomalies up to 20 minutes earlier than the previously existing one.

This dissertation proposes an approach that differs from anomaly detection solutions using ML models or employing self-adapted monitoring [5, 222, 77], and it is unique in the design space of distributed debugging tools. To the best of our knowledge, this is the first approach that applies distributed debugging techniques for interpreting fault injection experiments. In the context of fault injection, the fault-free executions are used as a reference for identifying anomalies in fault-injected executions performed under the same conditions (same workload, same node deployment, etc.): therefore, the approach does not rely on programmer-written specifications to identify failures (even if such specifications could cooperate with our approach to gain further insights); moreover, our approach does not rely on inferring causal relationships (which requires more intrusive instrumentation and may be inaccurate for asynchronous and rare interactions). Since the approach only relies on modeling the observed sequences of events, it can be easily deployed and integrated into interactive tools for debugging and visualization, to provide more robust trace comparison and analysis abilities.

2.5 Failure Mode Analysis

The existing fault injection tools detect the occurrence of failures by looking for specific events, such as service errors returned by the distributed system to its clients (e.g., API errors); performance degradation and bottlenecks; high-severity error messages in the logs of the system; and assertion failures introduced by developers inside the software. *Destini* [98] uses a declarative relational logic language (Datalog) to allow developers to customize *test specifications* (i.e., fault-tolerance properties that need to be fulfilled in the presence of faults), and for checking that the system complies with them. These specifications are expressed in terms of events (e.g., failures and protocol events), and relations over them representing expectations and facts (e.g., data blocks or packets that are expected in a given state, which are compared with the ones that are actually observed during the test). Similarly, *P#* [67] identifies failures using liveness specifications (e.g., lack of progress, such as the inability to restore a failed node) and safety specifications (validity assertions on the local and global states of the system), written with a domain-specific language in terms of communicating state machines with asynchronous events. Mariani *et al* [149] proposed a lightweight fault localization approach that trains machine learning models with correct executions only, and compensates the inaccuracy that derives from training with positive samples, by elaborating the outcome of machine learning techniques with graph theory algorithms.

The previous solutions require domain expertise and human effort to be applicable. This dissertation investigates techniques to automate the identification of failure modes without supervision, to ease the adoption of fault injection by practitioners.

The use of clustering to automatically discover and analyze failure modes is a topic widely addressed by previous research. Arunajadai *et al.* [13] described a clustering-based method for grouping failure modes in electromechanical consumer products. The approach groups failure modes based on their occurrence, to determine whether a failure should be considered by itself or whether it tends to accompany other kinds of failures. Then, the analyst can prioritize critical failure modes. The approach uses a hierarchical clustering algorithm with the complete linkage method. Chang *et al.* [40] combines clustering with risk management, by grouping failure modes that have similar risk levels concerning three factors (severity, occurrence, detection), and visualizes them to ease multi-criteria decision making. Their approach clusters and visualizes failures as a tree structure that is easy to understand. It is evaluated in the context of farming applications. Duan *et al.* [73] analyze evaluations of failure modes in natural language by FMEA experts, using fuzzy sets to extract features, and the *k-means* algorithm to cluster the failure modes. Xu *et al.* [259] proposed a method to construct the component-failure mode (CF) matrix automatically, by mining unstructured texts using the Apriori algorithm and the semantic dictionary WordNet to build a standard set of failure modes. As in the work by Arunajadai *et al.* [13], the matrix is used for grouping the failure modes using clustering algorithms, such as the *K-means*. Rahimi *et al.* [207] analyzed a large dataset of truck crash data, based on police reports about the driver, vehicle, crash, and citation information.

They address the problem of high-dimensionality spaces, by adopting block clustering to investigate heterogeneity in the crash dataset. This approach considers two sets (observations and variables) simultaneously and organizes the data into homogeneous blocks. Liu *et al.* contributed with several studies on the failure mode and effects analysis [113]. They improved failure mode analysis using two-dimensional uncertain linguistic variables and alternative queuing [144] and proposed a novel approach combining HULZNs and DBSCAN algorithms to assess and cluster the risk of failure modes [143]. They evaluated the feasibility of the proposed approaches in real use-case scenarios, showing the ability to classify failure modes in complex and uncertain conditions.

Different from these solutions, this dissertation introduces an approach tailored for the domain of cloud system failures, where the data consist of symbolic sequences, which are obtained from events recorded through distributed tracing technology. Our approach leverages deep neural networks, to automatically cluster the failure modes without manual effort for feature engineering. Moreover, we also investigate clustering in combination with anomaly detection for cloud systems.

2.6 Runtime Failure Detection

Promptly detecting failures at runtime is fundamental to stop failure propagation and mitigate its effects on the system. In this work, we exploit runtime verification to state the correctness of a system execution according to specific properties. In literature, some studies refer to runtime verification as runtime monitoring or dynamic analysis. Runtime monitoring consists of the observation of behaviors of the target system during its operation instead of verifying the system according to a specific model.

Over the last decades, several efforts have been spent on methodologies and tools for debugging and monitoring distributed systems. *Aguilera et al.* [2] proposed an approach to collect black-box network traces of communications between nodes. The objective was to infer causal paths of the requests by tracing call pairs and by analyzing correlations. Magpie [16] and Pinpoint [43] reconstruct causal paths by using a tracing mechanism to record events at the OS-level and the application server level. The tracing system tags the incoming requests with a unique *path identifier* and links resource usage throughout the system with that identifier. *Gu at al.* [95] proposes a methodology to extract knowledge on distributed system behavior of request processing without source code or prior knowledge. The authors construct the distributed system's component architecture in request processing and discover the heartbeat mechanisms of target distributed systems. Pip [210] is a system for automatically checking the behavior of a distributed system against programmer-written expectations about the system. Pip provides a domain-specific expectations language for writing declarative descriptions of the expected behavior of large distributed systems and relies on user-written annotations of the source code of the system to gather events and to propagate path identifiers across chains of requests. OSProfiler [182] provides a lightweight but powerful library used by fundamental components in OpenStack cloud computing platform [176]. OSProfiler provides an annotation system that can be able to generate traces for requests flow (RPC and HTTP messages) between OpenStack subsystems. These traces can be extracted and used to build a tree of calls which can be valuable for debugging purposes. To use OSProfiler, it is required deep knowledge about OpenStack internals, making it hard to use in practice.

Research studies on runtime verification focused on formalisms for describing properties to be verified. Typically, a runtime verification system provides a Domain Specification Language (DSL) for the description of properties to be verified. The DSL can be a stand-alone language or embedded in an existing language. Specification languages for runtime verification can be regular, which includes temporal logic, regular expressions, and state machines, but also non-regular, which includes rule systems, stream languages.

In the runtime verification literature, there is an established set of approaches for the specification of temporal properties, which include Linear Temporal Logic (LTL) [199], Property Specification Patterns (PSP) [76], and Event Processing Language (EPL) [80]. Linear Temporal Logic is the most common family of specification languages. This approach supports logical and temporal operators. LTL is extensively used as specification language in many model checkers [49, 31, 109]. The Property Specification Patterns consist of a set of recurring temporal patterns. Several approaches use PSP and/or extend original patterns used in [29]. Event Processing Language is used to translate event patterns in queries that trigger event listeners whether the pattern is observed in the event stream of a Complex Event Processing (CEP) environment [254]. In general, CEP is a technology for the collection, aggregation, and analysis of sequences of events that originated from various sources, occurring at different moments in time. The most interesting characteristic of CEP systems is that can be used in *Stream-based Runtime Verification* or *Stream Runtime Verification* (SRV) tools. SRV is a declarative formalism to express monitors using streams; the specifications are used to delineate the dependencies between streams of observations of the target systems and the output of the monitoring process.

Lola [64] is an SRV tool and implements a runtime verification as a stream computation, where output streams are defined in terms of input streams and/or other output streams. In particular, Lola defines a specification language and algorithms for both online and offline monitoring of synchronous systems and can be used to describe correctness/failure assertions but also statistical measures.

Esper [80] (see Appendix C) is an open-source software product for CEP and streaming analytics supporting Java and .NET languages. Esper provides an EPL language, a compiler, and a runtime environment. The language is declarative and data-oriented and extends the SQL standard for analyzing streams of events with respect to time. The Esper compiler compiles EPL source code into Java bytecode and the resulting executable code runs on a JVM within the Esper runtime environment. The Esper runtime provides an engine for online and real-time analysis. Finally, Esper is designed to provide low latency and high throughput and to be lightweight in terms of memory, CPU, and IO usage.

In [272], *Zhou et al.* propose a runtime verification based trace-oriented monitoring framework for cloud computing systems. The requirements of the monitoring can be specified by formal specification language, i.e. LTL, Finite State Machine (FSM). The tracing adopted in this approach is fine-grained, in which traces are a collection of events and relationships: every event records the details of one execution step in handling the user request (function name, duration), every relationship records the causal relation between two events. Using both the events and the relationships, it is possible to represent a trace into a so-called *trace tree*. In a trace tree, a node represents an event and an edge represents a relationship between events. This approach is generalizable at the cost of accessing the target source code to get the knowledge needed for instrumenting the code and gaining information about events relationships. However, this is not always the case, leading this approach difficult to exploit in practice. In [200], *Power and Kotonya* propose Complex Patterns of Failure (CPoF), an approach that provides reactive and proactive Fault-Tolerance (FT) via Complex Event Processing and Machine Learning for IoT (Internet of Things). Reactive-FT support is used to train Machine Learning models that proactively handle imminent future occurrences of known errors. Even if CPoF is intended for IoT systems, it inspired us in the use of Complex

Event Processing to build the monitor.

This dissertation proposes an approach presenting several points of novelty compared to state-of-the-art studies and tools in runtime verification literature. In particular, the proposed methodology relies on *black-box tracing*, instead of regular tracing, avoiding knowing about system internals and the collection of information about the relationships between events (i.e., uncorrelated events). Further, we provide a new set of monitoring rules that well fit distributed systems and cloud computing infrastructure requirements, in which we need to face peculiar challenges like multi-tenancy, complex communication between subsystems, lack of knowledge of system internals. Based on the analysis of the events collected during system operation, we can specify the normal behavior of the target system and perform *online anomaly detection*.

This page intentionally left blank.

Chapter 3

Fault Injection Tool-suite

This chapter presents a new fault injection tool, *ProFIPy* [54], designed to be *programmable*, enabling users to add and to customize a software fault model. By using this tool, users can specify new software fault models using a *domain-specific language* (DSL) for fault injection. A domain-specific language is a small, usually declarative, language that offers expressive power focused on a particular problem domain. In many cases, DSL programs are translated to calls to a common subroutine library and the DSL can be viewed as a means to hide the details of that library [244].

The tool compiles the specification into an automatically-generated fault injector. Finally, the generated fault injector is applied to the software-under-test to generate fault-injected versions and to execute experiments. To achieve better usability, *ProFIPy* is provided as *software-as-a-service*, and includes a workflow for configuring the fault load and the workload to i) fully automate the execution of experiments using container-based virtualization and parallelization, and to ii) perform failure data analysis. The tool has been designed for the popular Python language, which has recently arisen as one of the most widespread languages (e.g., among the GitHub and StackOverflow communities [90, 228]), and has found applications in several areas such as systems software (e.g., the OpenStack cloud platform is one of the largest projects in Python [187, 180]), enterprise and web applications and data science [202]. We present *ProFIPy* in the context of a Python project, by performing three fault injection campaigns in which we define three different fault loads.

3.1 Fault Injection Domain-Specific Language

ProFIPy allows the user to enter a *bug specification* using a high-level and easy-to-use DSL language, which is close to the Python language. The bug specification describes how the source code of the program should be transformed to introduce a software bug. It consists of two parts:

- **Code pattern**: a description of which parts of the program should be fault-injected. The fault injection tool parses the source code of the software and will generate a fault for every match of the code pattern.
- **Code replacement**: a description of the code that should be injected, which will replace the original source code that matched the code pattern.

The code pattern describes a combination of program entities (variables, expressions, blocks, control flow constructs, etc.) that will be searched for in the software-under-injection. The code pattern can either consist of a Python snippet of code; or, it can be a mix of Python code and DSL directives. In the former case, *ProFIPy* will look for *exact* matches between the Python snippet in the code pattern and the Python code in the software-under-injection. In the latter case, the DSL directives will make the pattern match several different variants of the Python snippet of code. Similarly, the code replacement can either be Python-only code, i.e., the injector will insert a fixed snippet of buggy code; or, it can contain a mix of Python and DSL directives, i.e., the injected buggy code can vary depending on what matched the code pattern.

Figure 3.1 shows three examples of bug specifications. These specifications inject three fault types from G-SWFIT [75]: the omission of a function call (MFC); the omission of a small block of statements surrounded by an IF construct (MIFS); and a wrong parameter in input to a function call (WPF). Differing from the G-SWFIT technique, we modified the definition of the fault types, to point out the features of the DSL language, and to emulate more accurately some of the bugs that we found in the OpenStack project [58, 52].

```
change {
    $BLOCK{tag=b1; stmts=1,*}
    $CALL{name=delete_*}(...)
    $BLOCK{tag=b2; stmts=1,*}
} into {
    $BLOCK{tag=b1}
    $BLOCK{tag=b2}
}
```

(a) Missing function call fault (MFC).

```
change {
    if $EXPR{var=node} :
        $BLOCK{stmts=1,4}
        continue
} into {
}
```

(b) Missing IF construct with statements (MIFS) fault.

```
change {
    $CALL#c{name=utils.execute}(..., $STRING#s{val=*-*}, ...)
} into {
    $CALL#c(..., $CORRUPT($STRING#s), ...)
}
```

(c) Wrong parameter in function call (WPF) fault.

Figure 3.1. Examples of fault specifications.

The MFC fault type from G-SWFIT looks for function calls in the software-under-injection, where there is no return value from the function call, or where the return value is ignored by the caller [75]. By targeting this kind of function call, the injector can emulate a function call omission by removing these function call statements, and yet to obtain a syntactically-correct program, as the removal does not break any dependency with the rest of the program. Moreover, the G-SWFIT study [75] recommended that the function call should only be removed when the function call is not the only statement in its block, to better reflect the real bugs from open-source projects that were analyzed in that study.

In Figure 3.1a, the code pattern (i.e., the *change { ... }* part of the specification) looks for any function or method call, by using the `$CALL` directive of the DSL. The `{name=delete_*}` syntax after `$CALL` means that we are targeting calls where the function name starts with "*delete_*" string, in order to inject faults in calls to the OpenStack Neutron APIs `delete_port`, `delete_subnet`, `delete_network`, etc. This is an example of how a user may want to customize fault injection according to domain knowledge: these APIs are prone to omissions (e.g., the Neutron bug #1028174 [132]), and users may want to simulate these faults to assess solutions for resource leak detection. The rest of the specification implements the rules of the MFC fault type. `$CALL` only matches statements where the function or method call is the outermost part of the statement: thus, a statement like `x = mycall()`, where the assignment is the outermost expression, would not match the code pattern of Figure 3.1a. The *(...)* syntax means that we are targeting function calls with any number of input parameters (zero, one, or more). The directives `$BLOCK` directives require that the function call must be both preceded and followed by one or more statements. Finally, the code replacement (i.e., the *into { ... }* part of the specification) means that we want to transform the matched code by replacing it only with the blocks that precede and follow the function call. The `{tag=...}` syntax after `$BLOCK` allows the user to give a label (e.g., `b1`, `b2`) to the parts of the code pattern that matched the software-under-injection, and to reuse these parts in the code replacement.

In the second example (Figure 3.1b), the MIFS fault type matches an IF construct with its statements (up to 4), and removes them, i.e., the code replacement part of the specification is empty. The specification mixes fragments of Python code (i.e., the `if` construct and `continue` keywords) and DSL directives (`$EXPR`, `$BLOCK`). Again, we refined the original fault type from G-SWFIT by leveraging domain knowledge, to inject into more specific targets. We emulate another recurring issue in OpenStack, in which metadata of resources (e.g., the `UUID` of instances) must have been initialized to allow operating on the resource, but a check on the validity of the metadata has been omitted (e.g., the Nova bug #1096722 [133]). To emulate this real bug, we target `if` constructs that check specific variables (e.g., variables called `node`, which are used throughout the OpenStack Nova codebase) and that skip an operation if the check fails (e.g., by issuing a `continue`).

In the third example (Figure 3.1c), the WPF fault type injects an invalid parameter to a function call. The bug specification replaces a `$CALL` statement with the same `$CALL` statement, but modifying one of the input parameters. We use again a `tag` to reuse code from the code pattern in the code replacement, by means of the `#c` syntax after `$CALL`, i.e., the matched function call is labeled as "*c*". We tailored the bug specification to match another recurring issue in OpenStack, in which an external utility (e.g., `iptables`, `dnsmasq`, `e2fsck`) is invoked at the host OS level, but with incorrect or missing parameters (e.g., the Nova bug #732549 [131]). Thus, we target the `utils.execute()` library function (the `name` attribute in `$CALL`), and look for a string literal (`$STRING`) among the input parameters of the function, where the string contains the character used by UNIX utilities to denote parameters. In the code replacement, we inject the same function call, but the string literal (labeled *s*) is wrapped by a function call that corrupts the string with random contents, using the `$CORRUPT`

DSL directive.

In addition to these examples, we have been using the DSL to define several fault models in an industrial context, in cooperation with Huawei Technologies Co. Ltd. The DSL provided us a fine-grain control over the injections, by combining DSL directives with Python code fragments. Other fault types include the injection of exceptions within `try` blocks, in order to increase the test coverage of error handlers [123, 151]; the injection of `None` values from library function calls, in order to test error handlers in which the returned value is checked by an `IF` construct after the call; the omission of optional input parameters to function calls; the omission of AND/OR clauses in `IF` conditions; wrong or missing initialization of data, such as key-value pair literals in Python dictionaries, using the `$CORRUPT` directive; high resource consumption (CPU, memory, storage), using the `$HOG` directive. The DSL can be used to inject more complex fault types, by using regular expressions for specifying search patterns; using the tagging syntax in the *change* block, to change the order of statements in the *into* block; mutating any arithmetic, boolean, and control flow expression of the Python grammar; injecting algorithmic bugs by removing entire portions of code (e.g., patterns with multiple nested loops and control flow constructs), and by injecting artificial time delays using a `$TIMEOUT` directive. More examples are presented in § 3.3.

3.2 The *ProFIPy* workflow

ProFIPy provides a complete fault injection workflow, which assists test engineers at applying software fault injection in Python systems. The *ProFIPy* workflow generates a set of mutated versions of the target software, according to user-defined bug specifications. These mutated versions are executed in a controlled environment, and further analyzed for drawing insights about the system behavior under failure. Figure 3.2 summarizes

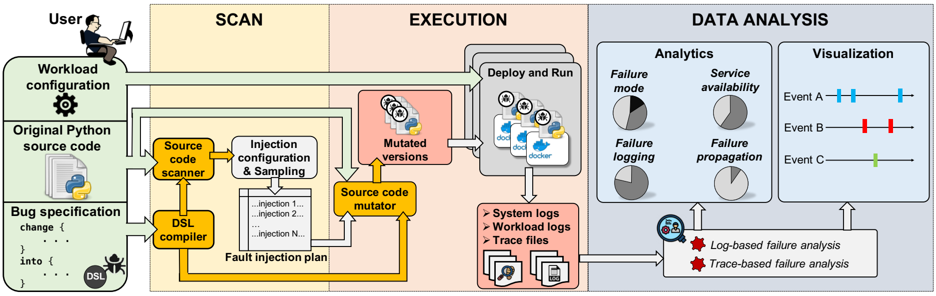


Figure 3.2. Workflow of the *ProFIPy* tool.

the workflow, which consists in a sequence of three main phases, that is, *Scan* (see § 3.2.1), *Execution* (see § 3.2.2), and *Data Analysis* (see § 3.2.3 and § 3.4). The following sub-sections provide details for each phase.

3.2.1 Scan

In the *Scan* phase, the user interacts with the *ProFIPy* tool to define the *fault injection plan*, which is the set of fault injection experiments to be run. Each experiment specifies a fault to be injected. *ProFIPy* takes in input the source code of the target software, and the bug specification described by using our DSL (section 3.1). The fault model is stored in a JSON file, and users can save and import fault models of previous fault injection campaigns. *ProFIPy* provides pre-defined fault models based on previous fault injection studies (section 2).

The *Scan* phase identifies *fault injection points* in the software, i.e., a statement (or group of statements) in the source code where *ProFIPy* can inject the software bug according to the user-defined specification. *ProFIPy* looks for arithmetic/boolean expressions, method and function calls, variable initializations, and other kinds of statements.

ProFIPy processes the target code using its Abstract Syntax Tree (AST) representation, which is commonly by program analyzers to represent the structure of a piece of code. The *DSL compiler* component takes the bug specification written using the DSL and generates a meta-model, which consists of a small AST that reflects the structure of the code in the code pattern. The meta-model will be used by the *source code scanner*, which visits the program's AST to find matches against the code pattern (i.e., portions of the program's AST that match the AST of the meta-model). The meta-model is also used by the *source code mutator* to generate fault-injected versions of the program (see § 3.2.2).

After obtaining a set of fault injection points, the user can select a subset of such locations according to their needs. For example, the user may want to perform experiments only for a specific component (e.g., class or file); the user may want to inject a sample of randomly-chosen faults (e.g., to enforce a limit on the number of experiments); or, the user can inject faults in all of the injection points. The set of injections defines the fault injection plan, which is used in the *Execution* phase.

In this chapter, the proposed DSL is tailored for the Python language. It is possible to define a similar DSL to support other languages, such as C/C++ and Java. Several of the bug patterns for Python could be re-used (i.e., patterns not involving special Python syntax). The porting would mostly affect the DSL compiler and the source code scanner and mutator.

3.2.2 Execution

In this phase, *ProFIPy* iterates over the fault injection plan. In each experiment, the *original* Python source code is transformed into a *mutated* version, which is identical to the original except for a few mutated statements. The mutation emulates a residual bug in the software. For example, to inject a wrong parameter bug in a method call, *ProFIPy* modifies the method call statement by replacing it with a call to the same method but with different or corrupted input parameters; to emulate an omission by the developer, *ProFIPy* deletes the method call in the mutated version. The set of mutated versions are the *faultload* that will be executed in the experiments. At the end of every experiment, *ProFIPy* collects logs from the target system for data analysis (§ 3.2.3).

The user also configures a *workload*, i.e., a set of directives to exercise the target software during the experiments. The workload emulates the operating conditions of the system and triggers the injected fault. Moreover, the workload serves to detect service failures and recovery abilities, e.g., by looking for crashes and timeouts of the workload (e.g., due to stalled service calls), or by performing consistency checks with test assertions on the outputs of the workload (e.g., after a resource has been modified by the workload, the behavior of the system should reflect the new state of the resource).

The user defines the workload by providing command-line directives. For example, the user can use UNIX shell commands to start the target software, e.g., to launch a UNIX daemon such as a network server. Command-line directives can be used both to invoke the command-line interface of the target Python program or to indirectly launch the software by running automated test scripts. These scripts can be uploaded by the user along with the target Python source code (Figure 3.2). Additionally, the user can specify command-line directives to launch workload generator tools, such as HTTP and RPC traffic generators, which in turn exercise the target software.

ProFIPy runs the fault injection experiments within a *container-based* experimental environment, by using the Docker virtualization system [70]. The tool first creates a container image, in which it copies the Python source code uploaded by the user. The user can customize the container image by adding configuration directives in *Dockerfile* format [69], such as, to install within the container external dependencies to run the Python software under test (e.g., using the *pip* command), and to install external tools (e.g., HTTP and RPC traffic generators). Then, for each fault to be injected, *ProFIPy* deploys a new container, by copying into it the mutated source code with the fault, and runs the workload directives defined by the user. The experiment ends when the workload completes, or when a user-defined timeout expires. Finally, *ProFIPy* cleans-up the experimental environment by deallocating the container. In this way, the tool can also clean-up any resource leaked or corrupted because of the injected fault (e.g., stale processes or files). Using containers also allows the tool to run several parallel experiments on independent sandboxes, to take advantage of multi-core CPUs. *ProFIPy* tunes the number of parallel experiments according to run at most $N - 1$ parallel containers at the same time, where N is the number CPU cores in the host system [251]. To avoid interferences in memory and I/O bandwidth, the tool further reduces the number of parallel containers if it hits a threshold for memory and I/O utilization.

ProFIPy can enable and disable the injected faulty code at any time during the execution of the target software. The mutated source code retains a copy of the original statements of the fault injection point, similarly to the EDFI fault injection tool [91]: *ProFIPy* mutates the source code by inserting an `IF ... ELSE ...` construct, where the two branches include respectively the original statements and the faulty ones. Then, the tool can control which of the two branches to execute, by writing a control variable (a "*trigger*") allocated in a shared memory area between the tool and the target software. This ability enables additional analyses of the effects of failures and recovery. The tool executes the workload for two times ("*rounds*"), without restarting the target program between the two executions. In the first round, the injected fault is enabled, so that it infects the target software with error states, possibly causing service failures. The workload is executed again in the second round, but the injected fault is disabled. Of course, if the target program fails and is unable to recover, the second workload execution will fail. The second round allows us to analyze the *scope* of the error states [265, 233]. In the best case, the error state is confined to service requests that were issued during the first round, and the requests during the second round are not affected by any error (e.g., the target software recovers a correct state with a restart). In the worst case, the error states are persistent even after that the faulty code is disabled, causing further failures during the second round. This analysis provides additional feedback to the user about the failure behavior of the target software.

During the experiments, *ProFIPy* saves the output of the target program (*stdout*, *stderr*) and the output of the workload directives (e.g., the commands for launching a workload generator, which reports service failures). Moreover, the tool can be configured to save log files that may be generated by the target software or by the workload. These outputs and logs are analyzed in the last phase of the *ProFIPy* workflow (*data analysis*), as discussed in the following.

3.2.3 Data Analysis

The *data analysis* evaluates the target software in terms of service failures, logging, and recovery. *ProFIPy* classifies the experiments into a set of **failure modes**, which include the crash and the timeout of the target software, and user-defined failure modes. The user can specify patterns (e.g., using keywords and regex) that the tool will look for among the outputs and the logs produced by the experiments. For example, failure modes can include failures of the workload (e.g., the workload stops due to a service API exception) and of the target software (e.g., the software detects an error state with an internal assertion, and reports it with a high-severity log message). The tool reports the statistical distribution of failure modes. The user can drill-down the individual classes of failures, to further inspect logs of experiments in that class. The user can also drill-down with respect to fault types and injected components, to identify the critical areas (e.g., components that are most prone to failures) where

Table 3.1. Injected fault types.

Fault Category	**Injection Target**	**Examples of Injections**
Failures when calling external library APIs	API calls to the `urllib` and `os` Python modules	Exceptions, `None` objects, omitted call, wrong call
Wrong inputs in Python-etcd API	`set(key, val)`, `get(key)`, `test_and_set(key, val, old)`, ...	String corruptions, `None` values, negative integers
Resource management bugs	`set(key, val)`, `get(key)`, `test_and_set(key, val, old)`, ...	Hog threads inside methods of Python-etcd

failure mitigations are most needed.

ProFIPy can analyze failures with respect to workload rounds. It computes a *service availability* metric, i.e., the percentage of experiments in which the software was (un)available in the second round of execution (injected fault disabled), because of error states generated during the first round (injection fault enabled) that persisted and were not recovered. These cases deserve a deeper analysis, e.g., to identify resource leaks that may occur in error handling paths, and that may cause more failures over time [114, 94].

3.3 Case Study

We present an application of *ProFIPy* in the context of *Python-etcd* [201], which is a library that provides Python bindings for the *etcd* distributed key-value store [81]. Huawei uses *Python-etcd* in their systems and asked for three fault classes to be evaluated using our fault injection tool (Table 3.1): (i) call failures when invoking APIs from external libraries (wrong response, timeouts, etc.), (ii) wrong inputs to the *Python-etcd* APIs, and (iii) resource management faults. We implemented these fault types using the *ProFIPy* DSL language.

We performed three fault injection campaigns on *Python-etcd* version 0.4.5. The workload used deploys the *etcd* server, and it uploads and queries several key-value pairs of a different kind (e.g., with directories, sub-keys, TTL, etc.) that we derived from *Python-etcd*'s integration tests. In the following subsections, we present the injected fault types and analyze failure modes using *ProFIPy*.

3.3.1 Errors from external APIs

In the first campaign of experiments, we injected faults at method calls in Python-etcd external modules, targeting the methods of `urllib` (a Python package for working with URLs) and from `os` (e.g., Python methods for file I/O). The injected fault types include:

- **Throw Exception**: The `raise` of the exception on a method call, according to pre-defined, per-API list of exceptions (e.g., `ConnectTimeoutError`);
- **Missing Function Call**: A method call is entirely omitted (e.g., replaced with the python statement `pass`);
- **Missing Parameters**: A method call is invoked with omitted parameters (e.g., the method uses a default parameter instead of the

correct one).

For this faultload, *ProFIPy* identified 26 points where to inject faults. In 13 cases, the workload covered the injected faulty code. We found failures in 12 experiments.

▷ **Reconnection failure**. In half of the cases, we found failures in both rounds of execution, as denoted by the *service availability* metric. The experiments did not complete within the timeout, and `etcd` was unable to reconnect even after the fault removal. We found that the `etcd` server was unable to bind to a TCP/IP port. Thus, restarting `etcd` does not suffice to recover from the fault, but the port needs to be explicitly freed. We need additional exception handlers to catch exceptions caused by network connections, such as time-outs.

▷ **Critical errors about `'member has already been bootstrapped'`**. In a few experiments, Python-etcd was unable to perform operations on `etcd` in the first round, due to an inconsistent state of the server caused by the fault. To recover from this failure, the system needs a more elaborated exception handling: it should explicitly remove the affected member by using the dynamic configuration API of `etcd`, and it should restart `etcd` by reverting to a previous consistent state.

▷ **Client process crash due to an exception**. In the remaining cases, the client process crashed during the first round due to an unhandled exception. Moreover, the system was not available after disabling the fault. In these cases, Python-etcd should provide exception handlers to catch these exceptions or to raise another kind of exception (such as *EtcdException*) to be managed by Python-etcd client process.

3.3.2 Wrong Inputs

In the second campaign of fault injection experiments, we injected faults in input parameters of Python-etcd API methods. We configured *ProFIPy* with fault types for injecting corrupted inputs, such as strings with random characters, None object references, negative integers, etc. For example, let us consider the `method test_and_set(key, value, old_value)` taking in input three parameters: A fault consists in injecting a corrupted input in the first parameter (string type) by randomly replacing the characters of the string.

The *ProFIPy* tool identified 66 locations where to inject these faults. In all of the cases, the injected faulty code was covered by the workload, and in 29 experiments we found the following failures in the first round of execution:

▷ **AttributeError: 'NoneType' object has no attribute 'startswith'**. This failure is due to an issue with Python-etcd. It happens when the tool injects a `None` value instead of a string (e.g., a *key* string). Python-etcd does not check whether the input strings are valid. Therefore, when a `None` value is passed in input, Python-etcd uses the `startswith` attribute on a `None` reference. To avoid this failure, Python-etcd should sanitize null strings in inputs.

▷ **EtcdKeyNotFound exception**. This failure happens when a wrong key or value is injected. In this case, the workload failed because it is not able to find the expected key or value in the `etcd` datastore. The caller (in this case, the workload) needs to get/set the correct keys and values. Thus, the Python-etcd client should handle these exceptions.

▷ **EtcdException: Bad response: 400 Bad Request**. This failure happens when *ProFIPy* injects a wrong key or value that is not valid (e.g., a non-ASCII string). When this value is passed to `etcd`, the server rejects the request with the *HTTP Error 400 Bad Request.* Python-etcd should be fixed to check and sanitize non-ASCII strings.

3.3.3 Resource Management Bugs

In the last campaign of experiments, we injected CPU hogs to overload Python-etcd. We used *ProFIPy* for injecting stale threads that generate a high CPU load. We targeted the same methods of the second campaign of experiments, by injecting a resource hog after the method call. The tool found 37 injectable locations, and the faulty code was always covered during the workload execution. In 14 experiments, the system experienced a service failure in the first round of execution. Most of these failures forced a process termination with the exception *"UnboundLocalError: local variable ... referenced before assignment"*. In other cases, the workload also failed because of inconsistent values read from the `etcd` datastore. The high CPU usage triggered race conditions in Python-etcd, and in the Python interpreter itself. Since it is hard to find and fix these issues, the failure should be mitigated, by cleaning-up stale threads that may cause high CPU consumption. This should be pursued by monitoring at run-time the CPU utilization of Python processes, and by killing or restarting stale threads if CPU utilization is too high.

3.3.4 Performance evaluation

ProFIPy can quickly inject faults even for large projects since the scan and mutation can be parallelized across several CPUs (it is an "embarrassingly parallel" task). It took less than one minute to scan and mutate *Python-etcd* on an 8-core Intel Xeon with 16 GB RAM. We also evaluated performance on the OpenStack project, by targeting the three most important modules (Nova, Neutron, and Cinder) accounting for about 400K lines of Python code. Using the same hardware, *ProFIPy* takes about 20 min to identify 17488 injectable locations using 120 different DSL patterns, which is reasonable for practical purposes given the large size of this project. The duration of the *execution* phase is beyond the control of our tool since it depends on the time to deploy the target system and run the workload. It took between 10s and 120s (worst case of a "hang" failure) to run a single experiment on *Python-etcd*, and about 30 min to run all of the tests of this section. For OpenStack, an experiment takes several tens of minutes, since it is a complex system that deploys VMs, loads large storage volumes, initializes databases, etc. We were able to execute experiments on OpenStack through nightly parallelized runs.

3.4 Advanced Features

ProFIPy includes more, optional features for deeper analysis of the large amounts of data produced by fault injection experiments. We briefly report here on these features.

3.4.1 Coverage Analysis

To reduce the time needed to run the fault injection experiments, *ProFIPy* performs a preliminary analysis to avoid injecting faults in *program paths* that are not *covered* by the workload. Most likely, the workload will not cover all of the paths in the program, and injecting into non-covered paths causes a waste of time since the fault would not cause any effect. Before executing the experiments, *ProFIPy* conducts a *coverage analysis*, by running a "fault-free" execution (i.e., no-fault injected) using the same workload that will be used for the experiments. It generates coverage information by adding logging statements at every fault injection point in the target program discovered by the *scan* phase (see § 3.2.1). After the fault-free run, *ProFIPy* generates a *reduced* fault injection plan, by only including the covered fault locations.

3.4.2 Failure Logging

ProFIPy checks whether the target system can detect error states and report diagnostic information on *log files*. The tool computes a *failure logging* metric, i.e., the percentage of experiments in which the target software both experienced a workload failure and logged at least one error message. Failures and error logs are identified with user-provided keywords and regex. This metric gives feedback about the logging abilities, and non-logged failures are opportunities for improving telemetry. An example of this analysis can be found in a previous study [58].

3.4.3 Service Recovery

The ability to enable/disable the injected faulty code provides additional analyses on the effects of failures and recovery. The *service availability* metric evaluates the percentage of experiments in which the software was (un)available when the injected fault has been disabled, i.e., whether the error states generated by the injection persist and were not recovered. These cases are also worth a deeper analysis by the user, e.g., the developers need to avoid resource leaks when the software executes error handling paths, since these leaked resources may cause more failures as the software continues to execute. To perform such an analysis, the tool executes the workload two times ("rounds"), without restarting the target program between the two executions. In the first round, the injected fault is enabled, and it can infect the target software with error states, and cause potential service failures. The workload is executed again in the second round, but the injected fault is disabled (of course, if the target program fails and is unable to recover, the second workload execution may fail). This can be leveraged to analyze the *scope* of the error states [265, 233]. In the best case, the error state is confined to service requests that were issued during the first round, and the requests during the second round are not affected by any error (e.g., the target software can recover a correct state with a restart). In the worst case, the error states are persistent even after the faulty code is disabled, causing further failures during the second round. This analysis provides additional feedback to the user about the failure behavior of the target software. *ProFIPy* also allows the user to perform the log analysis by distinguishing between workload rounds.

3.4.4 Failure Propagation

ProFIPy checks if the fault in the injected component propagated across other components. The tool computes a *failure propagation* metric, i.e., the percentage of injected faults that impacted on more than one component. This metric is applicable for larger software with a component-based architecture, where each sub-system generates a distinct log file, or where logs of the sub-systems can be separated with keywords and regex. The user configures a list of sub-systems, their source code files (e.g., a sub-folder of the source code), and their log files or patterns. The experiments that exhibit propagation are worth further investigation, e.g., to develop more robust interfaces between sub-systems to prevent the propagation and make recovery easier. The failure propagation analysis will be addressed in detail in the Chapter 5.

3.4.5 Failure Visualization

ProFIPy provides a graphical representation of an experiment to help human analysts to get a simplified overview of the fault-injection experiments and to better understand the results [57].

The tool instruments selected APIs in the target software and records their invocations during the experiment using the *Zipkin* distributed tracing framework [274] (see Appendix B).

In particular, the tool instruments the following communication points:

- The *RESTful API libraries* of the OpenStack subsystems (e.g., Nova, Neutron, Cinder) used for communication between OpenStack and its clients. Each OpenStack subsystem includes a *client* component, which includes API bindings for communication.

- The *OSLO Messaging library*, which uses a message queue library, by exchanging messages with an intermediary queuing server (RabbitMQ) through RPC messages. These messages are used for communication among OpenStack subsystems.

Only 5 selected functions of these components are instrumented, by adding a total of 20 lines of Python code.

The tool visualizes the API calls as *events* on timelines as interactive plots. Figure 3.3 shows the output provided by *ProFIPy* for a fault injection experiment on the OpenStack cloud computing platform. The graphical representation is oriented to a human analyst that needs to understand what happened during the experiment. This representation shows the events between the OpenStack clients and the OpenStack subsystems (labeled as *REST API*), and the inter and intra- subsystems API calls events (labeled using the name of the subsystem).

This experiment injected a fault in the Nova subsystem, which manages VM instances in OpenStack. During the experiment, OpenStack was exercised by a workload, which emulated a system administrator or customer that deploys a new virtual infrastructure, by calling the OpenStack REST APIs. One of these API calls is an asynchronous request to create a new VM instance. After the API call ends, Nova takes a few minutes to create and initialize the instance. During these operations, we injected a Python exception to force a failure.

In order to point out how the fault impacted on the system, this representation divides the events among *common*, *missing*, and *spurious* ones. The groups are obtained by applying an anomaly detection algorithm (dis-

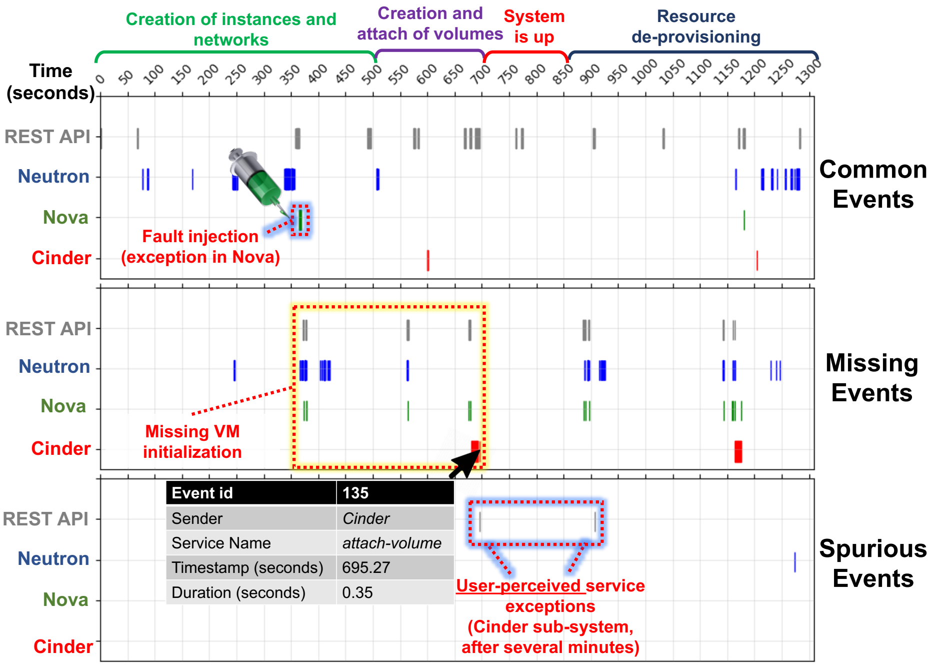


Figure 3.3. Graphical visualization of a fault injection experiment in Open-Stack.

cussed in § 5).

ProFIPy provides an interactive visualization of the experiment. A user can investigate a specific event by pointing the mouse at it: the tool displays a table with information about the event, which is important to facilitate the analysis of the failure. Our implementation uses *mpld3* [163], a library that brings together *Matplotlib*, the popular Python-based graphing library, and *D3js*, the popular JavaScript library for creating interactive data visualizations for the web. In the figure, we notice a large number of missing events. The failure affected several OpenStack subsystems over a relatively long time period. These events include several internal calls to initialize the instance and to attach it to its virtual resources (the "propagation chain" of the failure). The spurious events, instead, include the exceptions of two REST API calls to the client.

In our example, due to the injected fault, Nova did not complete the initialization of the VM instance, leaving it in an inactive state. Later on, after 5 minutes, the workload client experienced a service exception when calling the API of the Cinder subsystem, which manages storage volumes in OpenStack. By investigating the event pointed by the mouse, we notice that the event *<cinder-volume, attach-volume>* did not occur in the faulty execution (i.e., a missing event). Thus, *ProFIPy* helps the analyst in understanding that the workload did not attach a volume to the VM instance during the faulty execution.

Moreover, the OpenStack Neutron subsystem was also unable to attach the VM instance to the virtual network. Both Nova and Neutron did not raise any API exception, but the failure only became apparent to the client when invoking the API of the Cinder subsystem. Therefore, the problem propagated both across subsystems (from Nova to Neutron and Cinder) and across time, since the client perceived the failure only after a relatively long time. This behavior is problematic from the point of view of high availability, as the propagation delay also increases the time-to-detect and the time-to-recover the failure. Moreover, the longer the propagation chain, the more difficult will be for a developer to reason about how to best tolerate the fault, e.g., whether to manage the fault in Nova, Neutron, and/or Cinder and at which time to manage the fault during the workflow. For example, the API could return a more timely notification of the failure to the client, either by introducing a callback mechanism in the Nova API that creates the instance or by returning an error from other API calls to Nova or Neutron.

Chapter 4

Empirical Analysis of Software Failures in Cloud Systems

In this chapter, we empirically analyze the impact of high-severity failures in the context of a large-scale, industry-applied case study, to pave the way for failure mitigation strategies in cloud management systems. In particular, we analyze the OpenStack project, which is the basis for many commercial cloud management products [180] and is widespread among public cloud providers and private users [187]. Moreover, OpenStack is a representative real-world large software system, which includes several sub-systems for managing instances (Nova), volumes (Cinder), virtual networks (Neutron), etc., and orchestrates them to deliver rich cloud computing services.

We adopt software fault injection to accelerate the occurrence of failures caused by software bugs [48, 247, 166]: our approach deliberately injects bugs in one of the system components and analyzes the reaction of the cloud system in terms of fail-stop behavior, failure reporting through error logs, and failure propagation across components. We based fault injection on information on software bugs reported by OpenStack developers and users [179], in order to characterize frequent bug patterns occurring in this project. Then, we performed a large fault injection campaign on the three major subsystems of OpenStack (i.e., Nova, Cinder, and Neutron), for a total of 911 experiments, by using the fault-injection tool presented in Chapter 3.

4.1 Overview on the research problem

Mitigating the severity of software failures caused by residual bugs is a relevant issue for high-reliability systems [62], yet it still represents an open research challenge. Ideally, in the case that a fault occurs, a service should be able to mask the fault or recover from it in a transparent way to the user, such as, by leveraging redundancy. However, this is often not possible in the case of software bugs. Since software bugs are *human* mistakes in the source code, the traditional fault-tolerance strategies for hardware and network faults often do not apply. For example, if a service is broken because of a regression bug, then retrying to execute the service API with the same inputs would result again in a failure; a retrial would only succeed in the case that the software bug is triggered by a transient condition, such as a race condition [93, 94, 37]. If recovery is not possible, the failed operation must be necessarily aborted and the user should be notified [169, 159] so that the failure can be handled at a higher level of the business logic. For example, the end-user can skip the failed operation, or put on hold the workflow until the bug is fixed. If the failure does not immediately generate an exception from the OS or the programming language run-time, the service may continue its faulty execution until it corrupts in subtle ways the results or the state of resources. Such cases need to be mitigated by architecting the software into small, decoupled components for fault containment, in order to limit the scope of failure (e.g., the *bulkhead* pattern [169, 158]); and by applying defensive programming practices to perform redundant checks on the correctness of a service (e.g., pre and post-conditions to check that a resource has indeed been allocated or updated). In this way, the system can enforce a *fail-stop* behavior of the service (e.g., interrupting an API call that experiences a failure, and generating an exception), so that it can avoid data corruption and limit the outage to a small part of the system (e.g., an individual service call).

In this chapter, we study the extent of this problem in the context of a cloud management system. Applying software fault tolerance principles in such a large distributed system is difficult since its design and implementation is a trade-off between several objectives, including performance, backward compatibility, programming convenience, etc., which opens to the possibility of failure propagation beyond fault containment limits. We investigate this problem from three perspectives, by addressing the following three perspectives.

▷ **In the case that service experiences a failure, is it able to exhibit a fail-stop behavior?** If a service request could not be completed because of a failure, the service API should return an exception to inform about the issue. Therefore, we experimentally evaluate whether the service indeed halts on failure and whether the failure is explicitly notified to the user. In the worst case, the service API neither halts nor raises an exception, and the state of resources is inconsistent with respect to what the user is expecting (e.g., a VM instance was not actually created, or is indefinitely in the "*building*" state).

▷ **Are error reporting mechanisms able to point out the occurrence of a failure?** Error logs are a valuable source of information for automated recovery mechanisms and system operators to detect failures and restore service availability; and for developers to investigate the root cause of the failure. However, there can be gaps between failures and log messages. We analyze the cases in which the logs do not record any anomalous event related to a failure, since the software may lack checks to detect the anomalous events.

▷ **Are failures propagated across the services of the cloud management system?** To mitigate the severity of failures, failure should be limited to the specific service API that is affected by a software bug. If the failure impacts other services beyond the buggy one (e.g., the incorrect initialization of a VM instance also causes the failure of subsequent operations on the instance), it is more difficult to identify the root cause of the problem and to recover from the failure. Similarly, the failure may cause lasting effects on the cloud infrastructures (e.g., the virtual resources allocated for a failed instance cannot be reclaimed, or interfere with other resource allocations) that are difficult to debug and recover from. Therefore, we analyze whether failures can spread across different components of the system, and several service calls.

4.2 Methodology

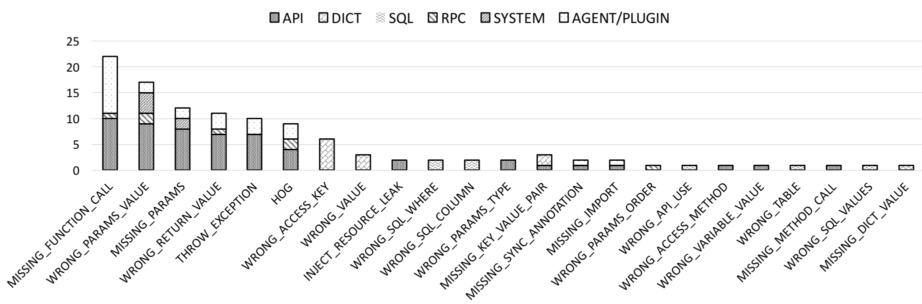


Figure 4.1. Distribution of bug types.

Our approach is to inject software bugs (§ 4.2.1, § 4.2.2) in order to obtain failure data from OpenStack (§ 4.2.3). Then, we analyze whether the system could gracefully mitigate the impact of the failures (§ 4.2.4).

4.2.1 Bug Analysis

A key aspect to performing software fault injection experiments is to inject representative software bugs [48, 75]. Since the body of knowledge on bugs in Python software [214, 189], the programming language of OpenStack, is relatively smaller compared to other languages, we seek more insights about bugs in the OpenStack project. Therefore, we analyzed the OpenStack issue tracker on the *Launchpad* portal [179], by looking for bug-fixes at the source code level, in order to identify *bug patterns* [75, 191, 153, 271, 240] for this project. From these patterns, we defined a set of bug types to be injected.

We went through the problem reports and inspected the related source code. We looked for reports where: (i) the root cause of the problem was a software bug, excluding build, packaging, and installation issues; (ii) the problem had been marked with the highest severity level (i.e., the problem has a strong impact on OpenStack services); (iii) the problem was fixed, and the bug-fix was linked to the discussion. We manually analyzed a sample of 179 problem reports from the Launchpad, focusing on entries with importance set to "*Critical*", and with status set to "*Fix Committed*" or "*Fix Released*" (such that the problem report also includes a final solution shipped in OpenStack). Of these problem reports, we identified 113 reports that met all of the three criteria. We shared the full set of bug reports (see Section 4.6).

The bugs encompass several areas of OpenStack, including bugs that affected the service APIs exposed to users (e.g., *nova-api*); bugs that affected dictionaries and arrays, such as a wrong key used in `image['imageId']`; bugs that affected SQL queries (e.g., database queries for information about instances in Nova); bugs that affected RPC calls between OpenStack subsystems (e.g., *rpc.cast* was omitted, or had a wrong topic or contents); bugs that affected calls to external system software, such as *iptables* and *dsnmasq*; bugs that affected pluggable modules in OpenStack, such as network protocol plugins and agents in Neutron. Figure 4.1 shows statistics about the bug types that we identified from the problem reports and their bug fixes. The five most frequent bug types include the following ones.

■ **Wrong parameters value**: The bug was an incorrect method call inside OpenStack, where a wrong variable was passed to the method call. For example, this was the case of the Nova bug #1130718 (https://bugs.launchpad.net/nova/+bug/1130718, which was fixed in https://review.openstack.org/#/c/22431/ by changing the exit codes passed through the parameter `check_exit_code`).

■ **Missing parameters**: A method call was invoked with omitted parameters (e.g., the method used a default parameter instead of the correct one). For example, this was the case of the Nova bug #1061166 (https://bugs.launchpad.net/nova/+bug/1061166, which was fixed in https://review.openstack.org/#/c/14240/ by adding the parameter `read_deleted='yes'` when calling the SQL Alchemy APIs).

■ **Missing function call**: A method call was entirely omitted. For example, this was the case of the Nova bug #1039400 (https://bugs.launchpad.net/nova/+bug/1039400, which was fixed in https://review.openstack.org/#/c/12173/ by adding and calling the new method

`trigger_security_group_members_refresh`).

■ **Wrong return value**: A method returned an incorrect value (e.g., `None` instead of a Python object). For example, this was the case of the Nova bug #855030 (https://bugs.launchpad.net/nova/+bug/855030, which was fixed in https://review.openstack.org/#/c/1930/ by returning an object allocated through `allocate_fixed_ip`).

■ **Missing exception handlers**: A method call lacks exception handling. For example, this was the case of the Nova bug #1096722 (https:

//bugs.launchpad.net/nova/+bug/1096722, which was fixed in https://review.openstack.org/#/c/19069/ by adding an exception handler for `exception.InstanceNotFound`).

4.2.2 Fault Injection

In this study, we perform *software fault injection* to analyze the impact of software bugs [247, 48, 166]. This approach deliberately introduces programming mistakes in the source code, by replacing parts of the original source code with faulty code. For example, in Figure 4.2, the injected bug emulates a missing optional parameter (a port number) to a function call, which may cause failure under certain conditions (e.g., a VM instance may not be reachable through an intended port). This approach is based on previous empirical studies, which observed that the injection of code changes can realistically emulate software faults [65, 48, 7], in the sense that *code changes produce run-time errors that are similar to the ones produced by real software faults*. This approach is motivated by the high efforts that would be needed for experimenting with hand-crafted bugs or with real past bugs: in these cases, every bug would require to carefully craft the specific conditions that trigger it (i.e., the topology of the infrastructure, the software configuration, and the hardware devices under which the bug surfaces). To achieve a match between injected and real bugs, we focus the injection on the most frequent five types found by the bug analysis. These bug types allow us to cover all of the main areas of OpenStack (API, SQL, etc.), and suffice to generate a large and diverse set of faults over the codebase of OpenStack.

We emulate the bug types by mutating the existing code of OpenStack. The Figure 4.2 shows the steps of a fault injection experiment. We used the *ProFIPy* tool presented in Chapter 3 to automate the bug injection process in Python code. The tool uses the *ast* Python module to generate an *abstract syntax tree* (AST) representation of the source code; then, it

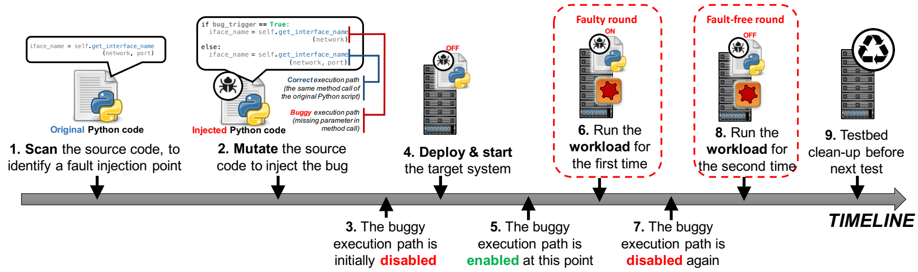


Figure 4.2. Overview of a fault injection experiment.

scans the AST by looking for relevant elements (function calls, expressions, etc.) where the bug types could be injected; it modifies the AST, by removing or replacing the nodes to introduce the bug; finally, it rewrites the modified AST into Python code, using the *astunparse* Python module. To inject the bug types of Section 4.2.2, we modify or remove method calls and their parameters. We targeted method calls related to the bugs that we analyzed, by targeting calls to internal APIs for managing instances, volumes, and networks (e.g., which are denoted by specific keywords, such as *instance* and *nova* for the methods of the Nova subsystem). Wrong input and parameters are injected by wrapping the target expression into a function call, which returns at run-time a corrupted version of the expression based on its data type (e.g., a null reference in place of an object reference, or a negative value in place of an integer). Exceptions are raised on method calls according to a pre-defined list of exception types.

The tool inserts fault-injected statements into an *if* block, together with the original version of the same statements but in a different branch (as in step 2 in Figure 4.2). The execution of the fault-injected code is controlled by a *trigger* variable, which is stored in a shared memory area that is writable from an external program. This approach has been adopted for controlling the occurrence of failures during the tests. In the first phase (**round 1**), we enable the fault-injected code, and we run a workload that exercises the service APIs of the cloud management system. During this phase, the fault-injected code could generate run-time errors inside the system, which will potentially lead to user-perceived failures. Afterward, in a second phase (**round 2**), we disable the injected bug, and we execute the workload for a second time. This fault-free execution points out whether the scope of run-time errors (generated by the first phase) is limited to the service API invocations that triggered the buggy code (e.g., the bug only impacts local session data). If failures still occur during the second phase, then the system has not able to handle the run-time errors of the first phase. Such failures point out the propagation of effects across the cloud management system (see § 4.1).

We implemented a workload generator to automatically exercise the service APIs of the main OpenStack sub-systems. The workload has been designed to cover several sub-systems of OpenStack and several types of virtual resources, in a similar way to integration test cases from the OpenStack project [185]. The workload creates VM instances, along with key pairs and a security group; attaches the instances to volumes; creates a virtual network, with virtual routers; and assigns floating IPs to connect the instances to the virtual network. Having a comprehensive workload allows us to point out propagation effects across sub-systems caused by bugs.

The experimental workflow is repeated several times. Every experiment injects a different fault, and only one fault is injected per experiment. Before a new experiment, we clean up any potential residual effect from the previous experiment, in order to be able to relate failure to the specific bug that caused it. The clean-up re-deploys OpenStack removes all temporary files and processes and restores the database to its initial state. However, we do not perform these clean-up operations between the two workload rounds (i.e., no clean-up between the steps 6 and 8 of Figure 4.2), since we want to assess the impact of residual side effects caused by the bug.

Table 4.1. Assertion check failures.

Name	**Description**
FAILURE IMAGE ACTIVE	The created *image* does not transit into the *ACTIVE* state
FAILURE INSTANCE ACTIVE	The created *instance* does not transit into the *ACTIVE* state
FAILURE SSH	It is impossible to establish a *ssh* session to the created instance
FAILURE KEYPAIR	The creation of a *keypair* fails
FAILURE SECURITY GROUP	The creation of a *security group* and *rules* fails
FAILURE VOLUME CREATED	The creation of a *volume* fails
FAILURE VOLUME ATTACHED	Attaching a *volume* to an instance fails
FAILURE FLOATING IP CREATED	The creation of a *floating IP* fails
FAILURE FLOATING IP ADDED	Adding a *floating IP* to an instance fails
FAILURE PRIVATE NETWORK ACTIVE	The created *network* resource does not transit into the *ACTIVE* state
FAILURE PRIVATE SUBNET CREATED	The creation of a *subnet* fails
FAILURE ROUTER ACTIVE	The created *router* resource does not transit into the *ACTIVE* state
FAILURE ROUTER INTERFACE CREATED	The creation of a router interface fails

4.2.3 Failure Data Collection

During the execution of the workload, we record inputs and outputs of service API calls of OpenStack. Any exception generated from the call (*API Errors*) is also recorded. In-between calls to service APIs, the workload also performs *assertion checks* on the status of the virtual resources, in order to point out failures of the cloud management system. In the context of our methodology, assertion checks serve as *ground truth* about the occurrence of failures during the experiments. These checks are valuable to point out the cases in which a fault causes an error, but the system does not generate an API error (i.e., the system is unaware of the failure state). Our assertion checks are similar to the ones performed by the integration tests as test oracles [124, 186]: they assess the connectivity of the instances through SSH and query the OpenStack API to check that the status of the instances, volumes and network is consistent with the expectation of the test cases. The assertion checks are performed by our workload generator. For example, after invoking the API for creating a volume, the workload queries the volume status to check if it is available (*VOLUME CREATED assertion*). These checks are useful to find failures not notified through the API errors. Table 4.1 describes the assertion checks.

If an API call generates an error, the workload is aborted, as no further operation is possible on the resources affected by the failure (e.g., no volume could be attached if the instance could not be created). In the case that the system fails without raising an exception (i.e., an assertion check highlights a failure, but the system does not generate an API error), the workload continues the execution (as a hypothetical end-user, being unaware of the failure, would do), regardless of failed assertion check(s). The workload generator records the outcomes of both the API calls and of the assertion checks. Moreover, we collect all the log files generated by the cloud management system. This data is later analyzed for understanding the behavior of the system under failure.

4.2.4 Failure Analysis

We analyze fault injection experiments according to three perspectives discussed in Section 4.1. The first perspective classifies the experiments *with respect to the type of failure that the system experiences.* The possible cases are the following ones.

■ **API Error**: In these cases, the workload was not able to correctly execute, due to an exception raised by a service API call. In these cases, the cloud management system has been able to handle the failure in a fail-stop way, since the user is informed by the exception that the virtual resources could not be used, and it can perform recovery actions to address the failure. In our experiments, the workload stops on the occurrence of an exception, as discussed before.

■ **Assertion failure**: In these cases, the failure was not pointed out by an exception raised by a service API. The failure was detected by the assertion checks made by the workload in-between API calls, which found an incorrect state of virtual resources. In these cases, the execution of the workload was not interrupted, as no exception was raised by the service APIs during the whole experiment, and the service API did (apparently) work from the perspective of the user. These cases point out non-fail-stop behavior.

■ **Assertion failure(s), followed by an API Error**: In these cases, the failure was initially detected by assertion checks, which found an incorrect state of virtual resources in-between API calls. After the assertion check detected the failure, the workload continued the execution, by performing further service API calls, until an API error occurred in a later API call. These cases also point out issues at handling the failure, since the user is unaware of the failure state and cannot perform recovery actions.

■ **No failure**: The injected bug did not cause a failure that could be perceived by the user (neither by API exceptions nor by assertion checks). The effects of the bug may be tolerated by the system (e.g., the system switched to an alternative execution path to provide the service); or, the injected source code was harmless (e.g., an uninitialized variable is later assigned before use). Since no failure occurred, these experiments are not further analyzed, as they do not allow to draw conclusions on the failure behavior of the system.

Failed executions are further classified according to a second perspective, *with respect to the execution round in which the system experienced a failure*. The possible cases are the following ones.

▷ **Failure in the faulty round only**: In these cases, a failure occurred in the first (faulty) execution round (Figure 4.2), in which a bug has been injected; and no failure is observed during the second (fault-free) execution round, in which the injected bug is disabled, and in which the workload operates on a new set of resources. This behavior is the likely outcome of an experiment since we are deliberately forcing a service failure only in the first round through the injected bug.

▷ **Failure in the fault-free round (despite the faulty round)**: These cases are concerns for fault containment since the system is still experiencing failures despite the bug being disabled after the first round and the workload operates on a new set of resources. This behavior is due to residual effects of the bug that propagated through session state, persistent data, or other shared resources.

Finally, the experiments with failures are classified from the perspective of *whether they generated logs able to indicate the failure*. In order to make more resilient a system, we are interested in whether it produces information for detecting failures and for triggering recovery actions. In practice, developers are conservative at logging information for post-mortem analysis, by recording high volumes of low-quality log messages that bury the truly important information among many trivial logs of similar severity and contents, making it difficult to locate issues [273, 139, 268]. Therefore, we cannot simply rely on the presence of logs to conclude that a failure was detected.

To clarify the issue, Figure 4.3 shows the distribution of the number of log messages in OpenStack across severity levels, *TRACE* to *CRITICAL*, during the execution of our workload generator, and *without* any failure. We can notice that all OpenStack components generate a large number of messages with severity *WARNING*, *INFO*, and *DEBUG* even when there is no failure. Instead, there are no messages of severity *ERROR* or *CRITICAL*. Therefore, even if a failure is logged with severity *WARNING* or lower, such log messages cannot be adopted for automated detection and recovery of the failure, as it is difficult to distinguish between "informative" messages and actual issues. Therefore, to evaluate the ability of the system to support recovery and troubleshooting through logs, we classify

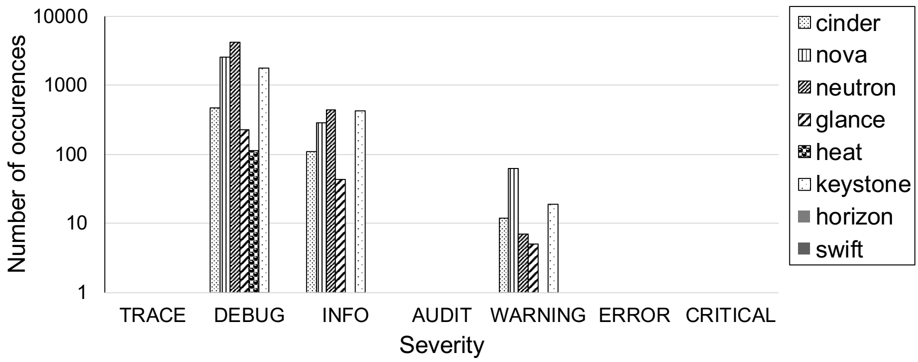


Figure 4.3. Distribution of log messages severity during a fault-free execution of the workload.

failures according to the presence of one or more *high-severity message* (i.e., *CRITICAL* or *ERROR*) recorded in the log files (**logged failures**), or no such message (**non-logged failures**).

4.3 Experimental Evaluation

In this work, we present the analysis of OpenStack version 3.12.1 (release *Pike*), which was the latest version of OpenStack when we started this work. We injected bugs into the most fundamental services of OpenStack [68, 227]: (i) the **Nova** subsystem, which provides services for provisioning instances (VMs) and handling their life cycle; (ii) the **Cinder** subsystem, which provides services for managing block storage for instances; and (iii) the **Neutron** subsystem, which provides services for provisioning virtual networks for instances, including resources such as *floating IPs*, *ports* and *subnets*. Each subsystem includes several components (e.g., the Nova sub-system includes *nova-api*, *nova-compute*, etc.), which interact through message queues internally to OpenStack. The Nova, Cinder, and Neutron sub-systems provide external REST API interfaces to cloud users.

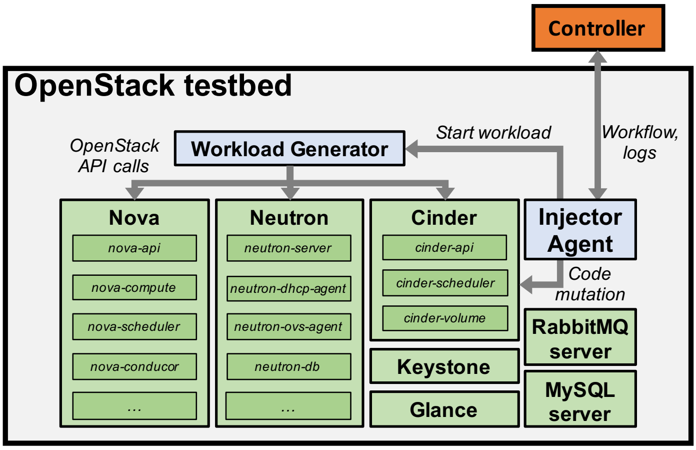


Figure 4.4. OpenStack testbed architecture.

Figure 4.4 shows the testbed used for the experimental analysis of OpenStack. We adopted an all-in-one virtualized deployment of OpenStack, in which the OpenStack services run on the same VM, for the following reasons: (1) to prevent interferences on the tests from transient issues in the physical network (e.g., sporadic network faults, network delays caused by other user traffic in our local data center, etc.); (2) to parallelize a high number of tests on several physical machines, by using the *Packstack* installation utility [208] to have a reproducible installation of OpenStack across the VMs; (3) to efficiently revert any persistent effect of a fault injection test on the OpenStack deployment (e.g., file system issues), in order to assure independence among the tests. Moreover, the all-in-one virtualized deployment is a common solution for performing tests on OpenStack [209, 152]. The hardware and VM configuration for the testbed includes: 8 virtual Intel Xeon CPUs (E5-2630L v3 @ 1.80GHz); 16GB RAM; 150 GB storage; Linux CentOS v7.0.

In addition to the core services of OpenStack (e.g., Nova, Neutron, Cinder, etc.), the testbed also includes our components to automate fault injection tests. The *Injector Agent* is the component that analyzes and instruments the source code of OpenStack. The *Injector Agent* can: (i) scan the source code to identify injectable locations (i.e., source-code statements where the bug types discussed in § 4.2.2 can be applied); (ii) instrument the source code by introducing logging statements in every injectable location, in order to get a profile of which locations are covered during the execution of the workload (***coverage analysis***); (iii) instrument the source code to introduce a bug into an individual injectable location.

The *Controller* orchestrates the experimental workflow. It first commands the *Injector Agent* to perform preliminary coverage analysis, by instrumenting the source code with logging statements, restarting the OpenStack services, and launching the *Workload Generator*, but without injecting any fault. The *Workload Generator* issues a sequence of API calls in order to stimulate OpenStack services. The *Controller* retrieves the list of injectable locations and their coverage from the *Injector Agent*. Then, it iterates over the list of injectable locations that are covered, and issues command for the *Injector Agent* to perform fault injection tests. For each test, the *Injector Agent* introduces an individual bug by mutating the source code, restarts the OpenStack services, starting the workload, and triggers the injected bug as discussed in § 4.2.2. The *Injector Agent* collects the logs files from all OpenStack subsystems and from the *Workload Generator*, which are sent to the *Controller* for later analysis (§ 4.2.4).

We performed a full scan of injectable locations in the source code of Nova, Cinder, and Neutron, for a total of 2 016 analyzed source code files. We identified 911 injectable faults that were covered by the workload. Figure 4.5 shows the number of faults per sub-system and per type of fault. The number of faults for each type and sub-system depends on the number of calls to the target functions, and on their input and output parameters, as discussed in § 4.2.2. We executed one of the tests per injectable location, by injecting one fault at a time.

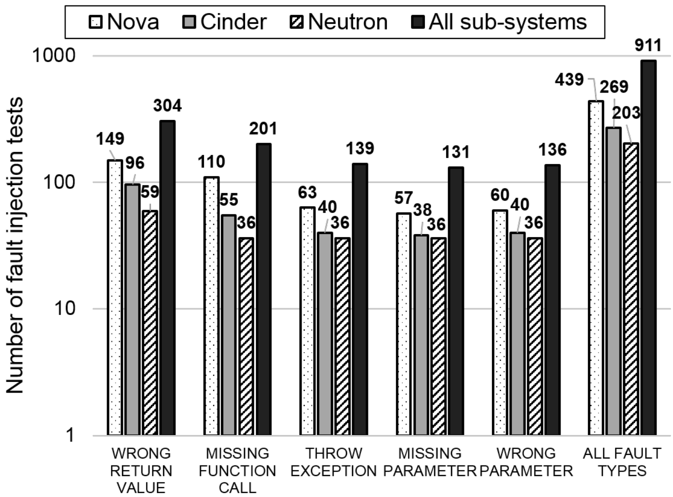


Figure 4.5. Number of fault injection tests.

After executing the tests, we found failures respectively in 52.6% (231 out of 439 tests), 46.4% (125 out of 269 tests), and 61% (124 out of 203 tests) of tests in Nova, Cinder, and Neutron, for a total of 480. In the remaining 47.3% of the tests (431 out of 911 tests), instead, there were neither an API error nor assertion failures: in these cases, the fault was not activated (even if the faulty code was covered by the workload), or there was no error propagation to the component interface. The occurrence of tests not causing failures is a typical phenomenon that occurs with code mutations, which may not infect the state even when the faulty code is executed [48, 130]. Yet, the injections provided us with a large and diverse set of failures for our analysis.

4.3.1 Does OpenStack Show a Fail-Stop Behavior?

We first analyze the impact of failures on the service interface APIs provided by OpenStack. The *Workload Generator* (which impersonates a user of the cloud management system) invokes these APIs, looks for

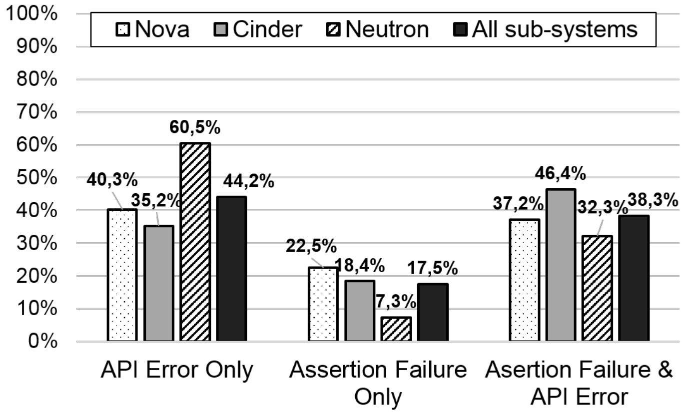


Figure 4.6. Distribution of OpenStack failures.

errors returned by the APIs, and performs assertion checks between API calls. A fail-stop behavior occurs when an API returns an error before any failed assertion check. In such cases, the *Workload Generator* stops the occurrence of the API error. Instead, it is possible that an API invocation terminates without returning any error, but leaving the internal resources of the infrastructure (instances, volumes, etc.) in a failed state, which is reported by assertion checks. These cases represent violations of the fail-stop hypothesis, and represent a risk for the users as they are unaware of the failure. To investigate this aspect, we initially focus on the faulty round of each test, in which fault injection is enabled (Figure 4.2).

Figure 4.6 shows the number of tests that experienced failures, divided into *API Error only*, *Assertion Failure only*, and *Assertion Failure(s), followed by an API Error*. The figure shows the data divided with respect to the subsystem where the bug was injected (respectively in Nova, Cinder, and Neutron); moreover, Figure 4.6 shows the distribution across all fault injection tests. We can see the cases in which the system does not exhibit a fail-stop behavior (i.e., the categories *Assertion Failure only* and *Assertion Failure followed by an API Error*) represent the majority of the failures.

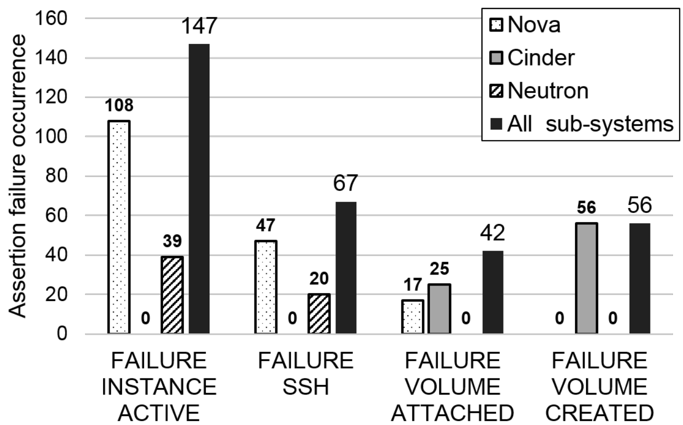


Figure 4.7. Distribution of assertion check failures.

Figure 4.7 shows a detailed perspective on the failures of assertion checks. Notice that the number of assertions is greater than the number of tests classified in the Assertion failure category (i.e., *Assertion Failure only* and *Assertion Failure followed by an API Error*) since a test can generate multiple assertion failures. The most common case has been one of the instances not active because the instance creation failed (i.e., it did not move into the *ACTIVE* state [186]). In other cases, the instance could not be reached through the network or could not be attached to a volume, even if in the *ACTIVE* state. A further common case is the failure of the volume creation, but only the faults injected in the Cinder sub-system caused this assertion failure.

These cases point out that OpenStack lacks redundant checks to assure that the state of the virtual resources after a service call is in the expected state (e.g., newly-created instances are active). Such redundant checks would assess the state of the virtual resources before and after a service invocation and would raise an error if the state does not comply with the expectation (such as a new instance could not be activated). However, these redundant checks are seldom adopted, most likely due to

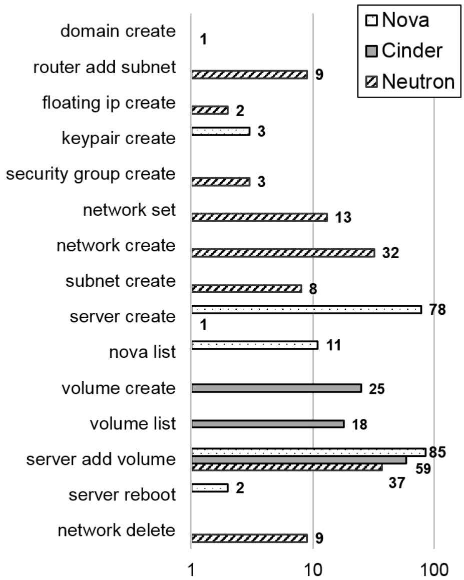


Figure 4.8. Distribution of API Errors.

the performance penalty they would incur, and because of the additional engineering efforts to design and implement them. Nevertheless, the cloud management system is exposed to the risk that residual bugs can lead to non-fail-stop behaviors, where failures are notified with a delay or not notified at all. This makes it not trivial to prevent data losses and to automate recovery actions.

Figure 4.8 provides another perspective on API errors. It shows the number of tests in which each API returned an error, focusing on 15 out of 40 APIs that failed at least one time. The API with the highest number of API errors is the one for adding a volume to an instance (*openstack server add volume*), provided by the Cinder sub-system. This API generated errors even when faults were injected in Nova (instance management) and Neutron (virtual networking). This behavior means that the effects of fault injection propagated from other sub-systems to Cinder (e.g., if an instance

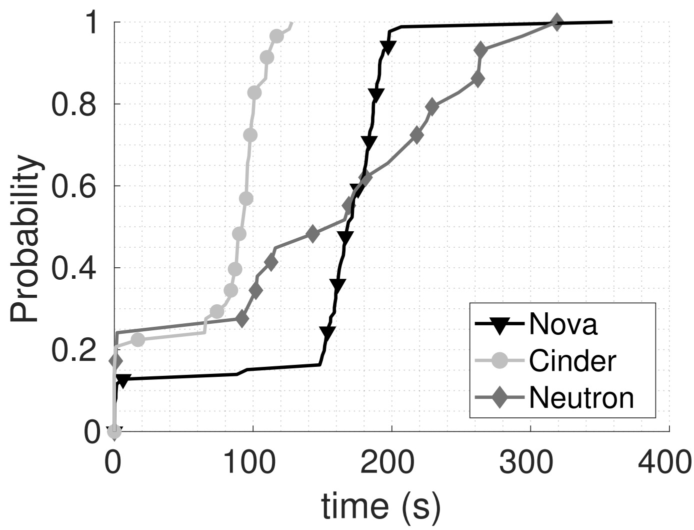


Figure 4.9. Cumulative distribution of API Error latency.

is in an incorrect state, other APIs on that resource are also exposed to failures). On the one hand, this behavior is an opportunity for detecting failures, even if in a later stage. On the other hand, it also represents the possibility of a failure to spread across sub-systems, thus defeating fault containment and exacerbating the severity of the failure. We will analyze fault propagation in more detail in Section 4.3.3.

To understand the extent of non-fail-stop behaviors, we also analyze the period of time (***latency***) between the execution of the injected bug and the resulting API error. This latency should be as low as possible. Otherwise, the longer the latency, the more difficult is to relate an API error with its root cause (i.e., an API call invoked much earlier, on a different sub-system or virtual resource); and the more difficult it is to perform troubleshooting and recovery actions. To track the execution of the injected bug, we instrumented the injected code with logging statements to record the timestamp of its execution. If the injected code is executed several times before a failure (e.g., in the body of a loop), we conservatively consider the last timestamp. We consider separately the cases where the API error is preceded by assertion check failures (i.e., the injected bug was triggered by an API different from the one affected by the bug) from the cases without any assertion check failure (e.g., the API error arises from the same API affected by the injected bug).

Figure 4.9 shows the distributions of latency for API errors that occurred after assertion check failures, respectively for the injections in Nova, Cinder, and Neutron. Table 4.2 summarizes the average, the 50^{th}, and the 90^{th} percentiles of the latency distributions. We note that in the first category (API errors after assertion checks), all sub-systems exhibit a median API error latency longer than 100 seconds, with cases longer than several minutes. This latency should be considered too long for cloud services with high-availability SLAs (e.g., four *nines* or more [23]), which can only afford a few minutes of monthly outage. This behavior points out that the API errors are due to a "reactive" behavior of OpenStack, which does not actively perform any redundant check on the integrity of virtual resources, but only reacts to the inconsistent state of the resources once they are requested in a later service invocation. Thus, OpenStack experiences a long API error latency when a bug leaves a virtual resource in an inconsistent state. This result suggests the need for improved error checking mechanisms inside OpenStack to prevent these failures.

In the case of failures that are notified by API errors without any preceding assertion check failure (the second category in Table 4.2), the latency of the API errors was relatively small, less than one second in the majority of cases. Nevertheless, there were few cases with an API error latency higher than one minute. In particular, these cases happened when bugs were injected in Nova, but the API error was raised by a different sub-system (Cinder). In these cases, the high latency was caused by the propagation of the bug's effects across different API calls. These cases are further discussed in § 4.3.3.

Table 4.2. Statistics on API Error latency.

	Subsys.	**Avg [s]**	**50^{th} %ile [s]**	**90^{th} %ile [s]**
API Errors after an Assertion failure	Nova	152.25	168.34	191.60
	Cinder	74.52	93.00	110.00
	Neutron	144.72	166.00	263.60
API Errors only	Nova	3.73	0.21	0.55
	Cinder	0.30	0.01	1.00
	Neutron	0.30	0.01	1.00

4.3.2 Is OpenStack Able to Log Failures?

Since failures can be notified to the end-user with a long delay, or even not at all, it becomes important for system operators to get additional information to troubleshoot these failures. In particular, we here consider log messages produced by OpenStack sub-systems.

We computed the percentage (***logging coverage***) of failed tests which produced at least one high-severity log message (see also § 4.2.4). Table 4.3 provides the logging coverage for different subsets of failures, by dividing them with respect to the injected subsystem and to the type of failure. From these results, we can see that OpenStack logged at least one high-severity message (i.e., with severity level *ERROR* or *CRITICAL*) in most of the cases. The Cinder subsystem shows the best results since logging covered almost all of the failures caused by fault injection. However, in the case of Nova and Neutron, logs missed some of the failures. In particular, the failures without API errors (i.e., *Assertion Failure only*) exhibited the lowest logging coverage. This behavior can be problematic for recovery and troubleshooting since the failures without API errors lack an explicit error notification. These failures are also the ones in need of complementary sources of information, such as logs.

To identify opportunities to improve logging in OpenStack, we analyzed the failures without any high-severity log across, with respect to the bug types injected in these tests. We found that *MISSING FUNCTION CALL* and *WRONG RETURN VALUE* represent the 70.7% of the bug types that lead to non-logged failures (43.9% and 26.8 %, respectively). The *WRONG RETURN VALUE* faults are the easiest opportunity for improving logging and failure detection since the callers of a function could perform additional checks on the returned value and record anomalies in the logs. For example, one of the injected bugs introduced a *WRONG RETURN VALUE* in calls to a database API called by the Nova sub-system to update the information linked to a new instance. The bug forced the function to return a *None* instance object. The bug caused an assertion check failure, but OpenStack did not log any high-severity message. By manually analyzing the logs, we could only find one suspicious message with the only *WARNING* severity and with little information about the problem, as this message was not related to database management.

The non-logged failures caused by a *MISSING FUNCTION CALL* emphasize the need for redundant end-to-end checks to identify inconsistencies in the state of the virtual resources. For example, in one of these experiments, we injected a *MISSING FUNCTION CALL* in the *LibvirtDriver* class in the Nova subsystem, which allows OpenStack to interact with the *libvirt* virtualization APIs [141]. Because of the injected bug, the Nova driver omits to attach a volume to an instance, but the Nova subsystem does not perform checks that the volume is indeed attached to the instance. This kind of end-to-end checks could be introduced at the service API interface of OpenStack (e.g., in *nova-api*) to test the availability of the virtual resources at the end of API service invocations (e.g., by pinging them).

4.3.3 Do Failures Propagate Across OpenStack?

We analyze failure propagation across sub-systems, to identify more opportunities to reduce their severity. We consider failures of both the

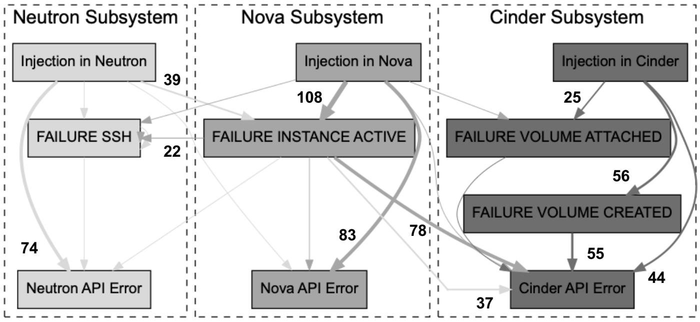
Table 4.3. Logging coverage of high-severity log messages.

	Logging coverage		
Subsystem	**API Errors only**	**Assertion failure only**	**Assertion failure and API Errors**
Nova	90.32%	80.77%	82,56%
Cinder	100%	95,65%	100%
Neutron	98.67%	66.67%	95%

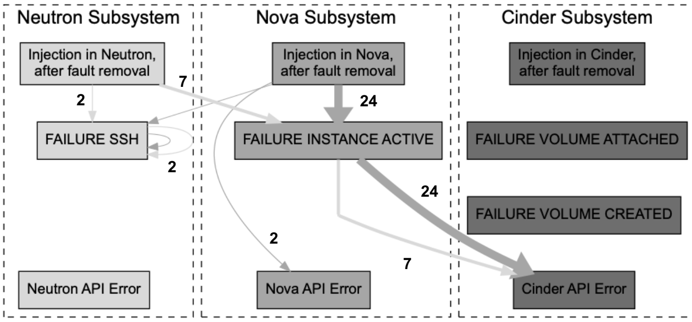
"faulty" and the "fault-free" rounds, respectively (Figure 4.2).

In the faulty round, we are interested in whether the injected bug impacted sub-systems beyond the injected one. To this aim, we divide the API errors with respect to the API that raised the error (e.g., an API exposed by Nova, Neutron, or Cinder). Similarly, we divide the assertion check failures with respect to the sub-system that manages the virtual resource checked by the assertion. There is a **spatial** fault propagation across the components if an injection on a sub-system (say, Nova) causes an assertion check failure or an API error on a different sub-system (say, Cinder or Neutron).

Figure 4.10a shows a graph of events that occurred during the faulty round of the tests with a failure. The nodes on the top of the graph represent the sub-systems where bugs were injected; the nodes on the middle represent assertion check failures; the nodes on the bottom represent API errors. The edges that originate from the nodes on the top represent the number of injections that were followed by an assertion check failure or an API error. Moreover, the edges between the middle and the bottom nodes represent the number of tests where an assertion check failure was followed by an API error. The most numerous cases are emphasized with proportionally thicker edges and annotated with the number of occurrences. We used different shades to differentiate the cases with respect to the injected sub-system.



(a) During faulty round.



(b) After removing the injected fault (fault-free round).

Figure 4.10. Fault propagation during fault injection tests.

The failures exhibited a propagation across OpenStack services in a significant amount of cases (37.5% of the failures). In many cases, the propagation initiated from an injection in Nova, which caused a failure at activating a new instance; as discussed in the previous subsections, the unavailability of the instance was detected in a later stage, such as when the user attaches a volume to the instance using the Cinder API. Even worse, there are some cases of propagation from Neutron across Nova and Cinder. These failures represent a severe issue for fault containment since an injection in Neutron not only caused a failure of their APIs but also impacted virtual resources that were not managed by these sub-systems. Therefore, the failures are not necessarily limited to the virtual resources managed by the sub-system invoked at the time of the failure, but also to other related virtual resources. Therefore, end-to-end checks on API invocations should also include resources that are indirectly related to the API (such as checking the availability of an instance after attaching a volume). For as concerns Cinder, instead, there are no cases of error propagation from this sub-system across Nova and Neutron.

We further analyze the propagation of failures by considering what happens during the fault-free round of execution. The fault-free round invokes the service APIs after the buggy execution path is disabled as dead code. Moreover, the fault-free round executes on new virtual resources (i.e., instances, networks, routers, etc., are created from scratch). Therefore, it is reasonable to expect (and it is indeed the case) that the fault-free round executes without experiencing any failure. However, we still observe a subset of failures (7.5%) that propagate their effects to the fault-free round. These failures must be considered critical since they are affecting service requests that are supposed to be independent but are still exposed to **temporal** failure propagation through shared state and resources. We remark that the failures in the fault-free round are caused by the injection in the faulty round. Indeed, we assured that previous injections do not impact the subsequent experiments by restoring all the persistent state of OpenStack before every experiment.

Figure 4.10b shows the propagation graph for the fault-free round. The most cases, the Nova sub-system was unable to create new instances, even after the injected bug is removed from Nova. A similar persistent issue happens for a subset of failures caused by injections in Neutron. These sub-systems both manage a relational database that holds information on the virtual instances and networks, and we found that the persistent issues are solved only after the databases are reverted to the state before fault injection. This recovery action can be very costly since it can take a significant amount of time, during which the cloud infrastructure may become unavailable. For this reason, we remark the need for detecting failures as soon as they occur, such as using end-to-end checks at the end of service API calls. Such detection would support quicker recovery actions, such as reverting the database changes performed by an individual transaction.

4.4 Threats to Validity

The injection of software bugs is still a challenging and open research problem. We addressed this issue by using code mutations to generate realistic run-time errors. This technique is widespread in the field of mutation testing [120, 125, 194, 193] to devise test cases; moreover, it is also commonly adopted by studies on software dependability [48, 247, 171, 75, 91] and on assessing bug-finding tools [71, 217]. In our context, bug injection is meant to anticipate the potential consequences of bugs on service availability and resource integrity. To strengthen the connection between the real and the experimental failures, we based our selection of code mutations on past software bugs in OpenStack. The injected bug types were consistent with code mutations typically adopted for mutation testing and fault injection (e.g., the omission of statements). Moreover, the analysis of OpenStack bugs gave us insights on where to apply the injections (e.g., on method calls for controlling Nova, for performing SQL queries, etc.). Even if some categories of failures may have been over-or under-represented (e.g., the percentages for failures that were not detected or that propagated), our goal is to point out the existence of potential, critical classes of failures, despite possible errors in the estimates of the percentages. In our experiments, these classes were large enough to be considered a threat to cloud management platforms.

4.5 Discussion and Lessons Learned

The analysis of fault injections pointed out the impact of the injected bugs on the end-users (e.g., service unavailability and resource inconsistencies) and on the ability of the system to recover and to report about the failure (e.g., the contents of log files, and the error notifications raised by the OpenStack service API). In particular, the results of the experimental campaign revealed the following findings:

- In the majority of the experiments (55.8%), OpenStack failures were not mitigated by a fail-stop behavior, leaving resources in an inconsistent state (e.g., instances were not active, volumes were not attached) unbeknownst to the user; In the 31.3% of these failures, the problem was never notified to the user through exceptions; the others were only notified after a long delay (longer than 2 minutes on average). This behavior threatens data integrity during the period between the occurrence of the failure and its notification (if any) and hinders failure recovery actions.

- In a small fraction of the experiments (8.5%), there was no indication of the failure in the logs. These cases represent a high risk for system operators since they lack clues for understanding the failure and restoring the availability of services and resources;

- In most of the failures (37.5%), the injected bugs propagated across several OpenStack components. Indeed, 68.3% of these failures were notified by a different component from the injected one. Moreover, there were relevant cases of failures that caused subtle residual effects on OpenStack (7.5%): even after removing the injected bug from OpenStack, cleaning up all virtual resources, and restarting the workload on a set of new resources, the OpenStack services were still experiencing a failure, that could only be recovered by fully restarting the OpenStack platform and restoring its internal database from a backup.

These results point out the risk that failures are not timely detected and notified, and that they can silently propagate through the system. Based on this analysis, we identify a set of directions towards a more reliable cloud management system.

■ **Need for deeper run-time verification of virtual resources.** Fault injections pointed out OpenStack APIs that leaked resources on failures, or left them in an inconsistent state, due to missing or incorrect error handlers. For example, the *server-create* API failed without creating a new VM, but it did not deallocate virtual resources (e.g., instances in "dead" state, unused virtual NICs) created before the failure. These failures can be prevented through fault injection. Moreover, residual faults should be detected and handled using run-time verification strategies, which perform redundant, end-to-end checks after a service API call, to assert whether the virtual resources are in the expected state. For example, these checks can be specified using temporal logic and synthesized in a run-time monitor [66, 41, 272, 206], e.g., a logical predicate for a traditional OS can assert that a thread suspended on a semaphore leads to the activation of another thread [10]. In the context of cloud management, the predicates should test at run-time the availability of virtual resources (e.g., volumes and connectivity), similarly to our assertion checks (Table 4.1). In Chapter 8, we propose a novel runtime failure detection approach tailored for cloud computing systems.

■ **Increasing the logging coverage.** The logging mechanisms in OpenStack reported high-severity error messages for many of the failures. However, there were failures with late or no API errors that would benefit from logs to diagnose the failure, but such logs were missing. In particular, fault injection identified function call sites in OpenStack where the injected wrong return values were ignored by the caller. These cases are opportunities for developers to add logging statements and to improve the coverage of logs (e.g., by checking the outputs produced by the faulty function calls). Moreover, the logs can be complemented with the run-time verification checks.

■ **Preventing corruptions of persistent data and shared state.** The experiments showed that undetected failures can propagate across several virtual resources and sub-systems. Moreover, we found that these propagated failures can impact shared state and persistent data (such as databases), causing permanent issues. Fault injection identified failures that were detected much later after their initial occurrence (i.e., with high API error latency, or no API errors at all). In these cases, it is very difficult for operators to diagnose which parts of the system have been corrupted, thus increasing the cost of recovery. Therefore, in addition to timely failure detection (using deeper run-time verification techniques, as discussed above), it becomes important to address the corruptions as soon as the failure is detected since the scope of recovery actions can be smaller (i.e., the impact of the failure is limited specific resources involved by the failed service API call). One potential direction of research is on selectively undoing recent changes to the shared state and persistent data of the cloud management system [249, 221].

4.6 Experimental artifacts

We release the following artifacts to support future research on mitigating the impact of software bugs: *(i)* the analysis of OpenStack bug reports (`https://doi.org/10.6084/m9.figshare.7731629`), *(ii)* raw logs produced by the experiments (`https://doi.org/10.6084/m9.figshare.7732268`), and *(iii)* tools for reproducing our experimental environment in a virtual machine (`https://doi.org/10.6084/m9.figshare.8242877`).

This page intentionally left blank.

Chapter 5

Identification of the Failure Symptoms in Cloud Systems

This chapter introduces a novel anomaly detection algorithm to find unusual events and interactions (i.e., symptoms of failures) that occurred in fault injection experiments. We designed the algorithm to be robust to noise in cloud systems, caused by *non-determinism* of timing and order of events, and to be quickly trained only a small set of *fault-free* executions of the distributed system, by using a *variable-order Markov Model*. Anomaly detection can aid human analysts in scrutinizing more efficiently the events that occurred during an experiment, by discarding uninteresting events, e.g., unusual, yet benign orderings of events caused by non-determinism. We evaluated the algorithm on the widespread *OpenStack* cloud management platform [180, 187]. We targeted the three main sub-systems of OpenStack (Nova, Neutron, Cinder) with fault injection under several scenarios. We show that anomaly detection can pinpoint anomalous events with a high hit rate, and can halve the number of false alarms due to non-determinism.

5.1 Methodology

The approach analyzes the cloud-computing system as a set of *black-box* communicating components, without leveraging any a priori information about their internals (e.g., the approach is unaware of invariants and pre-/post-conditions in the system). Thus, we apply unsupervised machine learning on execution traces to identify failure patterns.

The approach focuses on *messages* exchanged in the distributed system during the fault injection experiments. In general, messages are the key observation point for debugging and verification of distributed systems, since they reflect well the activity of the system [136]. For example, nodes perform work when they receive messages to provide a service to another node (e.g., through remote procedure calls), and reply with messages to provide the response and results; moreover, nodes use messages to asynchronously notify a new state to other nodes in the system. The approach is plugged into the public communication interfaces, such as REST APIs and message queues, based on off-the-shelf protocols and libraries, and it collects raw traces of messages exchanged among the components.

An important design objective is to make the approach *robust to non-determinism in distributed systems*, where the timing and the order of messages can unpredictably change (e.g., due to sporadic delays) regardless of the occurrence of failures. Thus, there is a need to discriminate between variations in the message order due to failures and by "benign" variations caused by non-determinism. To mitigate this uncertainty, we adopt a probabilistic model for anomaly detection that screens out the benign variations.

Another design objective is to use as few training samples as possible. The approach trains a model by executing the system several times. However, since the execution time to run a cloud workload can be significantly long (e.g., in our experiments, a single run takes tens of minutes),

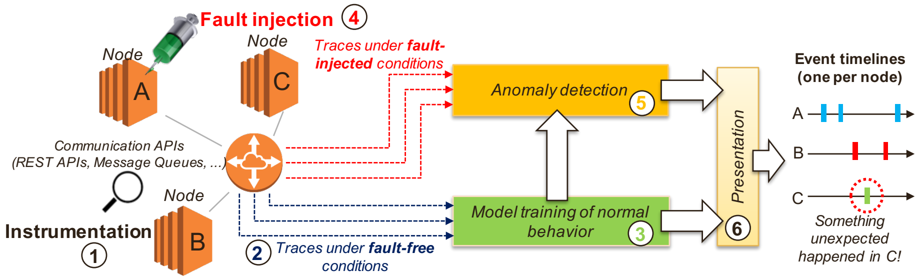


Figure 5.1. Overview of the proposed approach.

it is mandatory to keep these runs at a minimum to make the approach affordable in practice.

Figure 5.1 shows an overview of the proposed approach. We first instrument communication APIs (step 1). Then, we exercise the system with a workload, and with no-fault injected (step 2). We record a trace of all messages exchanged among the components, and between the components and the clients. Since no fault is injected, such trace is denoted as *fault-free trace*. We generate several fault-free traces, by running the workload several times. The fault-free traces are used as a training set to create a *model of normal behavior* (step 3). We adopt a probabilistic model to account for the natural variability of the interactions (e.g., different ordering, type, etc.) in the training traces.

We remark that having a representative experimental environment (i.e., matching the real-world operational environment, in terms of user workload, hardware, etc.) is a problem not limited to this approach, but it is a more general problem for fault injection [20]. Our goal is to facilitate the analysis of fault injection data, regardless of how well the data matches the operational environment (e.g., by architecting a realistic workload, by using a realistic configuration, etc.).

Once the model has been trained, the approach performs the fault injection experiments (step 4). We focus on injecting one fault per experiment, as injecting multiple faults concurrently is still an open research problem and not yet adopted in real projects, due to the high number of combinations among multiple faults. This step produces *fault-injected traces* (also *faulty traces*), one per experiment. The fault-injected traces are then analyzed using the previously defined normal behavior model to identify anomalies (step ⑤). Since all the executions (i.e., fault-free and fault-injected ones) are performed under the same conditions (i.e., same software and hardware configuration, same workload, etc.), any deviation between a fault-injected trace and the probabilistic model is attributed to the injected fault and it is considered as an anomaly.

In order to emphasize messages that were omitted because of the injected fault (i.e. only occurring in fault-free conditions), and new messages that were caused by the injected fault (i.e., only occurring under faulty conditions), the results of anomaly detection are visualized by presenting to the human analyst the messages of both the fault-injected and of a fault-free execution (step ⑥).

The anomaly detection algorithm constitutes the core of the proposed approach. Figure 5.2 shows a detailed flowchart of this algorithm. In the rest of this section, we discuss the phases of the workflow and present an example of fault injection analytics of a real system.

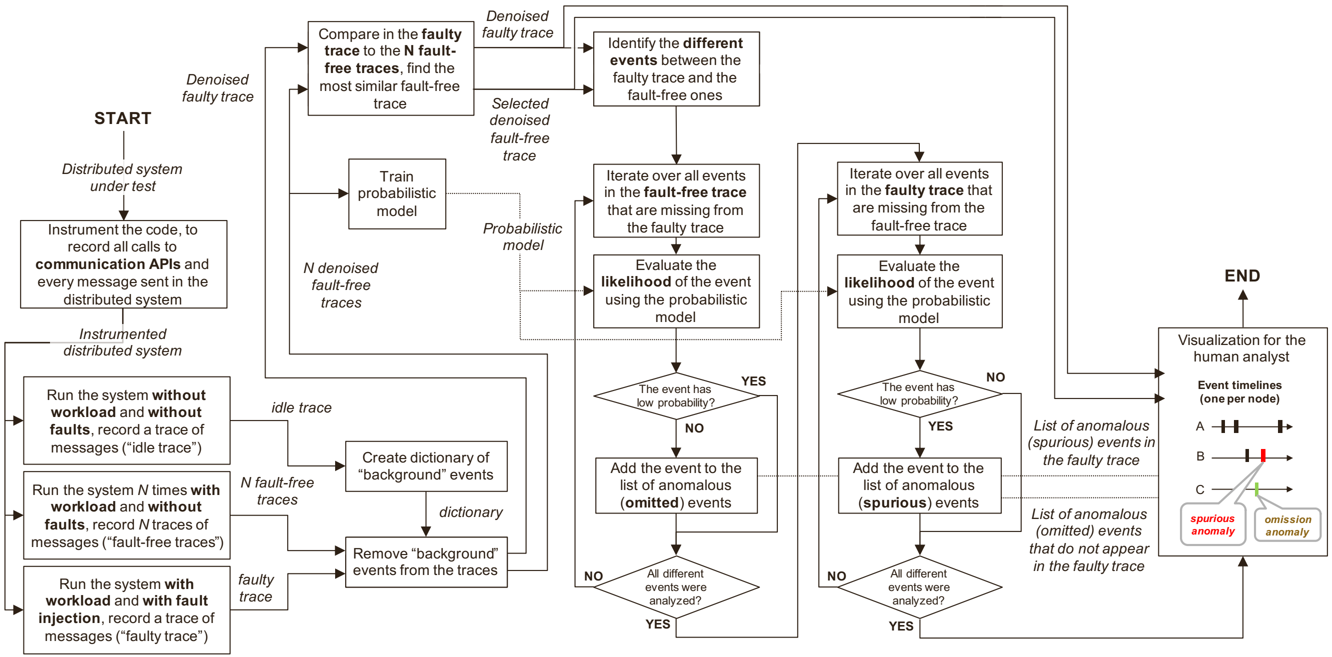


Figure 5.2. Detailed workflow of the anomaly detection.

5.1.1 Instrumentation

The first step of the approach consists of instrumenting the system under test, to collect the messages exchanged by nodes during the experiments [225]. To this purpose, the approach wraps the *communication APIs* that are invoked by every component in the system.

This instrumentation is a form of "black-box tracing", since it does not require any knowledge about the internals of the system under test, but it requires only basic information about the communication APIs being used. This approach is especially suitable when testers may not have a full and detailed understanding of the entire cloud platform. Moreover, this kind of distributed tracing is already familiar to developers for debugging, performance monitoring and optimization, root cause analysis, and service dependency analysis [47, 42].

The information recorded by the instrumented APIs includes the time at which a communication API has been called and its duration; the component that invoked the API (*message sender*) and the remote service that has been requested through the API call (*service API*). Moreover, we record information about the response message (e.g., the status line and the message body in an HTTP response, the body of the message, etc.). We refer to the calls to communication APIs (i.e., the messages collected during the experiments) as **events**. Thus, the execution of the system generates a **trace** of events that are ordered with respect to the timestamp given by the event collector.

This anomaly detection technique is designed to be tolerant to the non-determinism in the ordering of the events (e.g., due to random messaging delays) by using a probabilistic technique, which is discussed in Section 5.1.4.

5.1.2 Data Collection

Once the system has been instrumented, it is executed with the workload, collecting traces without injecting any fault (***fault-free traces***). Such fault-free traces (also known as *golden runs* or *reference runs*) have been adopted for fault injection experiments in small systems (e.g., embedded ones), by using the traces as a reference to understand how the fault-injected system derailed from a proper execution [135, 137, 166]. We generalize this approach to support more complex systems, such as cloud computing ones, and use unsupervised machine learning to discover unknown failure modes. In the next steps, we will use fault-free traces to train a model of the "normal" behavior of the distributed system, which we will use as a reference for analyzing failures. The model takes into account the variability of events across executions of the system (e.g., differences in the relative ordering of messages). Then, the system is executed again under fault injection, using the same workload of fault-free runs. For each experiment, we inject a different fault, and we collect a trace (**faulty trace**) of the events that are generated during the execution. Thus, we obtain several traces, one per experiment.

To recognize events that are generated by background and asynchronous activities, which are independent of the workload, we collect a third type of trace, namely (***idle trace***), which contains events occurring in the distributed system not caused by the workload or by the injected faults. Indeed, if these events are not removed from our analysis, they might be erroneously identified as (false) anomalies. Examples of such events are garbage collection, resource monitoring, updating database indexes, etc., and they can be triggered at arbitrary times. Another example in the OpenStack cloud computing platform is the events generated by the invocation of the method *sync_instance_info* of the *Nova Scheduler* component: this method is periodically called by compute nodes to notify the UUIDs of instances on the hosts, and it is not related to the workload.

To identify these events, we perform a separate execution of the cloud system, by leaving it in an idle state (i.e., no workload is applied) for several minutes before and after a fault-free execution. We record into the idle trace any background message collected during these periods. Then, we remove such background events from both the fault-free and faulty traces.

5.1.3 Trace Pre-processing

Each event in the system is described by the couple of <*message sender*, *service API*>. In our context, the *service API* represents the name of the invoked method (e.g., *create volume*), whereas the *message sender* is the name of the sub-system invoking the method (e.g., *Cinder*). The proposed approach represents the events within a trace with unique identifiers (i.e., *symbols*) so that two events of the same type are identified by the same symbol. Besides the specific event, we also consider the response status in the assignment of the symbols. For example, if the event is an HTTP message, we differentiate among invocations of the same GET method with different status codes (e.g., 200 for success, and 404 for failure). Events in a trace are ordered by their time of collection, and then converted into *sequences of symbols*: each symbol represents a specified couple <*message sender*, *service API*>, and the response status.

Once all execution traces have been converted into sequences, before resorting to anomaly detection, we perform preliminary filtering of events that do not represent anomalies. We identify events that do not exhibit any difference between the fault-injected and the fault-free executions, i.e., events that occur regardless of the injected fault. Since these events are not related to the failure modes, they can be discarded from the analysis. To identify these events, we look for overlapping symbols (i.e., same type, same order) between the faulty sequences and the fault-free ones.

The approach identifies overlapping symbols between the sequences, by computing the *longest common sub-sequence* (LCS) of the sequences [25]:

the LCS is a subset of symbols that are present in both sequences in the same order, and that can be obtained by removing (a minimal number of) symbols from the original sequences. This kind of problem is recurrent in computer science, such as in bioinformatics and source code versioning (e.g., in the *diff* Unix tool), and can be solved with efficient algorithms [115, 165]. The approach identifies a *selected fault-free trace* that is *most similar* to the fault-injected trace, i.e., the one with most overlapping symbols, by computing the normalized length of the LCS ($nLCS$) between the faulty trace and the fault-free ones, where $nLCS(x, y) = \frac{|LCS(x,y)|}{\sqrt{l_x \cdot l_y}}$, and where l_x and l_y are the lengths of the individual strings x and y. Then, it generates a list of differences (i.e., non-common events) between the selected fault-free trace and the faulty trace. These non-common events are further analyzed with a probabilistic model, to tell which ones are indeed anomalies.

5.1.4 Probabilistic Modeling

The analysis performed with LCS still does not suffice to identify failure-related events, since the differences in the faulty trace can be either actual symptoms of a failure (i.e., real anomalies, caused by the injected fault); or non-anomalous events (i.e., events that may, or may not occur in fault-free conditions, or may occur in a different order, due to non-deterministic behavior). The latter type of events may lead to *false alarms*, which may divert the attention of the human analyst. To overcome inaccuracies, we use a probabilistic model, in cascade after the trace analysis with LCS, to evaluate whether a non-common event is indeed an anomaly.

In particular, the approach uses a *Markov model* to estimate the probability of an event. Markov modeling is a popular approach for the probabilistic analysis of sequences of symbols (e.g., to predict the probability of a future symbol), such as in bioinformatics [231], data compression [212], and text and speech recognition [205]. Markov models do not require massive datasets to be trained, which is instead the case for other anomaly detection techniques like neural networks. The size of the training set is an important concern in our context, as developers have a limited time budget to spend for fault injection testing [10]. Since executions can take several hours in commercial-grade systems, we need to train the model with a minimal number of fault-free executions.

Among Markov models, *Hidden Markov Models* (HMMs) are a powerful and very popular technique among researchers in dependable computing, such as for anomaly detection and fault diagnosis purposes in critical infrastructures [19, 275, 63, 36]. HMMs separate *observations* (e.g., events) from the (hidden) *states* of the underlying stochastic process that generates the observations, since in many systems the current state is unknown for an external observer, and must be indirectly inferred from events [205]. However, we found that HMMs are not suitable for our anomaly detection problem. The main issue with HMMs is the high flexibility of the model, in terms of the high number of parameters that need to be tuned in the training phase (e.g., the number and probabilities of the hidden states). During the training phase, we cannot rely on a human analyst to annotate the events with the corresponding hidden state of the system, as it would be exceedingly time-consuming and error-prone for complex distributed systems with many unknown states. Instead, training HMMs with unannotated traces significantly increases the required size of the training set (e.g., up to thousands of traces using the EM algorithm) [36]. Another issue is the *zero frequency problem*, that is, modeling the probability of events with no occurrences in the training set, which is often the case in anomaly detection [252].

Therefore, we opt for a non-hidden Markov model where the states are a direct representation of the observed events. However, a simple Markov chain still does not suffice for our purposes, since the probability of the next state (i.e., the next event of the sequence) would only depend on the current state (i.e., the *memoryless* property). In general, this is not the case for event sequences that can be generated by a distributed system; in practice, the probability of an event is highly correlated with the history of the previous events. For example, in the OpenStack platform, the occurrence of an event representing a "volume attach" operation must be preceded by a sequence of several preliminary operations on the volume and on the instance to be attached (e.g., an instance must have been created and initialized).

Ultimately, we opted for *higher-order* Markov models, where the probability of events takes into account the history of the previous states of a sequence. In particular, since we do not have a fixed cardinality for the conditioning set of events in history, we adopt *Variable-order Markov Models* (VMMs). VMMs estimate the probability that a symbol σ can appear after a sequence s (named *context*), by counting the joint occurrences of σ and s in the training sequence to build the predictor $\hat{P}$, for variable cardinalities of s [24].

In this work, we use the notation defined by Begleiter et al. [24]. Let Σ be a finite alphabet. A learner is given a training sequence $x_1^n = x_1 x_2 ... x_n$, where $x_i \in \Sigma$ and $x_i x_{i+1}$ is the concatenation of x_i and x_{i+1}. Based on x_1^n, the goal is to learn a model $\hat{P}$ that provides a probability assignment for any future outcome given some past. Specifically, for any context $s \in \Sigma$ and symbol $\sigma \in \Sigma$, the learner should generate a conditional probability $\hat{P}(\sigma|s)$. The accuracy of the predictor $\hat{P}(\cdot|\cdot)$ is typically measured by its average log-loss $l(\hat{P}, x_1^T)$ with respect to a test sequence $x_1^T = x_1 ... x_T$:

$$\ell(\hat{P}, x_1^T) = -\frac{1}{T} \sum_{i=1}^{T} \log \hat{P}(x_i | x_1 ... x_{i-1}) \tag{5.1}$$

There exist many algorithms in the scientific literature for training and applying VMMs [24]. In particular, one important aspect that characterizes VMM algorithms is how they handle the zero-frequency problem (i.e., sequences with zero occurrences in the training set). If the probability is estimated by simply counting the number of occurrences, the unobserved events would get a zero probability, with an infinite log-loss. This problem is especially relevant in the case of long sequences with a rich alphabet, where the training set is "sparse" and only covers a tiny part of the multi-dimensional space of the sequences. The sequence of events generated by a distributed system also falls in this condition.

The approach uses the *Prediction by Partial Matching, Method C* (PPM-C) lossless compression algorithm [51], which is a variant of the original PPM algorithm published in 1984 by Cleary and Witten [50] that includes a set of improvements proposed by Moffat [161]. PPM is a statistical modeling technique that builds a predictor by combining several fixed-order context models [51], with different values of the order k, ranging from zero to an upper bound D (i.e., the maximal order of the Markov model) [154].

All PPM variants manage the zero-frequency problem by using two mechanisms, called *escape* and *exclusion*. For each context s of length $k \leq D$, the algorithm allocates a uniform probability mass $\hat{P}_k(escape|s)$ (which varies across PPM variants) for all symbols that did not appear after the context s in the training sequence. The remaining mass $1 - \hat{P}_k(escape|s)$ is distributed among all other symbols that have non-zero counts for this context. Using the escape mechanism, the conditional probability is given by [24]:

$$\hat{P}(\sigma|s_{n-D+1}^{n}) = \begin{cases} \hat{P}_D(\sigma|s_{n-D+1}^{n}), & \dagger \\ \hat{P}_D(escape|s_{n-D+1}^{n}) \cdot \hat{P}(\sigma|s_{n-D+2}^{n}), & \ddagger \end{cases} \tag{5.2}$$

$\dagger$ if $s_{n-D+1}^{n}\sigma$ occurred in the training sequence $\qquad$ $\ddagger$ otherwise where $\hat{P}_D(\cdot|\cdot)$ is a conditional probability with fixed-order D, which can be calibrated according to frequency counts from the observed sequences in the training set.

The exclusion mechanism is used to tune the probability estimates. This probability is inversely proportional to the size of the alphabet (for example, the probability of the escape is $1/|\Sigma|$ in the case of an empty context $s = \epsilon$), but the PPM-C introduces a correction. If a symbol σ appears after the context s of length $k \leq D$, it is redundant to consider σ as part of the alphabet when computing $\hat{P}_k(\cdot|s')$, for all s' suffix of s. Therefore, the estimates $\hat{P}_k(\cdot|s')$ are corrected by considering a smaller alphabet of observations [24]. For more information on PPM and the *Method C* variant, we refer the reader to the work by Begleiter et al. [24].

We set the maximum order D of the VMMs to 5. Indeed, it has been found that PPM achieves the best compression for this choice and that its accuracy saturates when the context is increased beyond this value [51].

5.1.5 Classification of Anomalies

The ultimate result of anomaly detection is to classify the events into:

- ***Common events***: Events that occurred both in the faulty trace and in at least one of the fault-free traces, with the same type and order.

- ***Anomalous events***: Differences between the faulty trace and the fault-free traces. They are further classified into:

  - ***Spurious events***: Events that would normally not occur under fault-free conditions.
  - ***Missing events***: Events that occur in fault-free conditions, but do not occur under fault injection.

As discussed in § 5.1.3, we first use the LCS algorithm to identify common events of a faulty trace, by comparing it to a *selected fault-free trace* (i.e., one of the fault-free traces in the training set, with the highest similarity to the faulty trace). Then, we further analyze the *LCS differences* (i.e., non-common events according to the LCS) using the VMM model (§ 5.1.4). We train the VMM with a set of $n-1$ fault-free traces, by using all the fault-free traces except the *selected fault-free trace*. Then, we apply the VMM to compute the probabilities of LCS differences, to determine whether they are indeed anomalous, as follows:

▷ **Analysis of LCS differences that only appear in the fault-injected trace.** The fault-injected trace takes the role of the *test sequence* for the VMM. We focus on symbols of the test sequence that were highlighted as differences in the previous LCS analysis. The goal is to confirm whether these symbols are actually unlikely events, not only with respect to the selected fault-free trace (i.e., the one used for determining the LCS) but also according to the whole set of fault-free traces in the training set. For each event not included in the LCS, we compute the probability of the event according to the VMM. If the probability is lower than a threshold $\epsilon_{\text{SPURIOUS}}$, then the symbol has a low likelihood to appear in that position of the sequence; thus, the VMM confirms that the symbol represents a **spurious** anomalous event. Otherwise, the event is considered non-anomalous.

▷ **Analysis of LCS differences that only appear in the selected fault-free trace.** The *selected fault-free trace* takes the role of the *test sequence* for the VMM. As for the previous step, we focus on symbols of the test sequence that were highlighted as differences in the previous LCS analysis. In this case, we consider the events that only appear in the selected fault-free trace: therefore, from the point of view of the fault-injected trace, these events represent *omissions*. This step confirms whether these events are likely, and thus their omission should be considered an anomaly. The approach applies the VMM to the events that only appear in the fault-free trace, by computing the probabilities of such events according to the remaining fault-free traces in the dataset. If the probability of the event is higher than a threshold $\epsilon_{\text{MISSING}}$, then there is a high likelihood for the symbol to be in that position of the sequence. Therefore, the fact that the event is missing in the fault-injected trace should be considered an anomaly, and thus it is marked as a **missing** anomalous event. Otherwise, if the probability of the event is less than the threshold, then the lack of such an event from the fault-injected trace is considered non-anomalous.

We remark that even if the two steps perform similar comparisons, the results obtained by them are different and complementary. If the fault-injected trace contains an anomalous event with a *low probability value* according to the VMM, then it is confirmed as spurious. Similarly, if the fault-injected trace does not contain an event with a *high probability value* in the selected fault-free trace, then the event is confirmed to be an omission. A practical approach is to select conservative thresholds (e.g., $\epsilon_{\text{SPURIOUS}} = 20\%$ and $\epsilon_{\text{MISSING}} = 80\%$), so that the VMM can filter out most of the LCS differences that are not actually spurious/missing events; and to leave to the human analyst the decision about the uncertain events. Therefore, the sensitivity of the probabilistic model is an important factor that makes it applicable in practice. We further analyze it in our experiments.

5.2 Experimental Evaluation

We evaluate our anomaly detection algorithm with experiments on the OpenStack cloud management platform, which is a relevant case of a large and complex distributed system.

5.2.1 Experimental Setup

We injected faults into the three most important sub-systems of OpenStack [68, 227]: (i) *Nova*, which provides services for provisioning instances (i.e., VMs) and handling their life cycle; (ii) *Neutron*, which provides services for provisioning virtual networks, including resources such as floating IPs, network interfaces, subnets, and routers; and (iii) *Cinder*, which provides services for managing block storage resources. Each of these three sub-systems represents by itself a complex system, and they are developed as independent projects by distinct and dedicated teams [230, 229]. We targeted OpenStack version 3.12.1 (release *Pike*), deployed on Intel Xeon servers (E5-2630L v3 @ 1.80GHz) with 16 GB RAM, 150 GB of disk storage, and Linux CentOS v7.0, connected through a Gigabit Ethernet LAN.

In our tests, we injected faults during the interactions among OpenStack components. We targeted the internal APIs used by OpenStack components for managing instances, volumes, networks, and other resources. The injected faults represent exceptional cases, such as a resource that is not found or unavailable, a processing delay when retrieving a resource, or an incorrect value caused by the user, the configuration, or a bug inside OpenStack. In particular, we focus on the following types of faults:

- **Throw exception**: An exception is raised on a method call, according to a pre-defined, per-API list of exceptions.
- **Wrong return value**: A method returns an incorrect value. The wrong return value is obtained by corrupting the targeted object, depending on the data type (e.g., by replacing an object reference with a null reference, or by replacing an integer value with a negative one).
- **Wrong parameter value**: A method is called with an incorrect input parameter. Input parameters are corrupted according to the

data type, as for the previous point.

- **Delay**: A method is blocked for a long time before returning a result to the caller. This fault can trigger timeout mechanisms inside OpenStack, and cause stalls.

We performed three distinct fault injection campaigns, in which we applied three different workloads:

- **New deployment** (DEPL): This workload configures a new virtual infrastructure from scratch, by stimulating all of the target subsystems (i.e., Nova, Cinder, and Neutron) in a balanced way. This workload creates VM instances, along with key pairs and a security group; creates and attaches volumes to an existing instance; creates a virtual network and a subnet, with a virtual router; assigns a floating IP to connect the instances to the virtual network; reboots the instances, and then deletes them.

- **Network management** (NET): This workload includes network management operations, to stress more the Neutron sub-system and virtual networking. The workload initially creates a network and a VM and generates network traffic via the public network. After that, it creates a new network with no gateway, brings up a new network interface within the instance, and generates traffic to check whether the interface is reachable. Finally, it performs a router rescheduling, by removing and adding a router resource.

- **Storage management** (STO): This workload performs storage management operations on instances and volumes, to stress more the Nova and Cinder sub-systems. In particular, the workload creates a new volume from an image, boots an instance, then rebuilds the instance with a new image (e.g., as it would happen for an update of the image). Finally, it performs a cleanup of the resources.

All the workloads invoke the OpenStack APIs provided by the Nova, Cinder, and Neutron sub-systems. We designed the workloads to cover several sub-systems of OpenStack and several types of virtual resources, similar to integration test cases from the OpenStack project [185], to point out potential failure propagation effects across sub-systems.

During the execution of the workload, any exception generated by API calls (*API Errors*) is recorded. In-between calls to service APIs, the workload also performs *assertion checks* on the status of the virtual resources, to point out failures of the cloud management system. These checks assess the connectivity of the instances through SSH and query the OpenStack API to ensure that the status of the instances, volumes, and the network is consistent with the expectation of the tests. In our context, assertion checks serve as *ground truth* about the occurrence of failures during the experiments. These checks are valuable to point out the cases in which a fault causes an error, but the system does not generate an API error (i.e., the system is unaware of the failure state) [58].

We consider an experiment as failed if at least one API call returns an API error or if there is at least one assertion check failure. Before every experiment, we clean up any potential residual effect from the previous experiment, to be able to relate failure to the specific fault that caused it. We re-deploy the cloud management system, remove all temporary files and processes, and restore the OpenStack database to its initial state.

5.2.2 Failure Dataset

We used the *ProFIPy* tool (see § 3) to scan the source code of Nova, Cinder, and Neutron to find all the injectable API calls, and to introduce faults by mutating the calls. For each workload, we identified the injectable locations that were covered by the workload itself, and we performed one fault injection test per covered location. In total, we performed 2 538 fault injection experiments, and we observed failures in 1 314 experiments (52%).

In the remaining tests, there were neither API errors nor assertion failures since the fault did not affect the behavior of the system (e.g., the corrupted state is not used in the rest of the experiment, or the error was tolerated). This is a typical phenomenon that occurs in fault injection experiments [48, 130]; yet, the experiments provided us with a large and diverse set of failures for our analysis. We focus on non-tolerated faults since they are the ones of interest for the analysts. Failures point out scenarios that are not yet handled by the cloud system, and that require additional fault tolerance mechanisms. The purpose of the proposed approach is to ease the identification of these failure modes.

Table 5.1 shows, for each workload, the number of event types d observed in the distributed system during the execution of the workloads, the average length of the fault-free sequences (in terms of the number of events in the trace), the total number of fault injection experiments for the workload, and the number of experiments that experienced at least one failure.

The number of unique events (i.e., different types of operations performed by the system) and the number of events (i.e., total operations of the system) per trace reflect the extent and diversity of the workloads. DEPL is the most stressful one in both regards, followed by NET and by STO. Moreover, the DEPL and the NET workloads are more non-deterministic than STO because the former perform a massive use of the network-related operations. Indeed, network operations are performed by the Neutron sub-system in an asynchronous way, such as by exchanging periodic and concurrent status polls among agents deployed in the datacenter and the Neutron server. This behavior leads to more non-deterministic variations in the traces. These differences among the workloads are useful to evaluate our approach under different degrees of complexity and non-determinism.

In our implementation, we adopt the *Zipkin* distributed tracing sys-

Table 5.1. Workload characteristics.

Workload	Num. unique events	Avg. num. of events per fault-free trace	Num. of total exps.	Num. of failed exps.
DEPL	64	285	1076	537
NET	40	252	561	262
STO	41	109	901	515

tem [274], due to its maturity, high performance, and support for several programming languages. The instrumented APIs send data via HTTP to a collector, which stores trace data. The collected events are ordered with respect to the timestamp given by the Zipkin collector.

As explained in § 3.4.5, we instrumented the following communication points to collect the events:

- The *OSLO Messaging library*, which uses a message queue library to exchange messages with an intermediary queuing server (RabbitMQ) through RPCs. These messages are used for communication among OpenStack sub-systems. In particular, we instrumented the *cast* and *call* methods, which are used when the RPC methods do not return or return a value to the caller, respectively [183].

- The *RESTful API libraries* of each OpenStack sub-system, i.e., *novaclient* for Nova (implements the OpenStack Compute API [174]), *neutronclient* for Neutron (implements the OpenStack Network API [178]), and *cinderclient* for Cinder (implements the OpenStack Block Storage API [173]). These interfaces are used for communication between OpenStack and its clients (e.g., IaaS customers).

Zipkin puts a negligible overhead in terms of run-time execution, as it adopts an asynchronous collection mechanism to avoid impacting critical execution paths. Indeed, we only needed to instrument 5 selected lines of code (i.e., the *cast* and *call* methods of OSLO to broadcast messages, and the clients), by adding simple annotations (the Zipkin context manager/decorator) only at the beginning of these methods (a total of 20 lines of Python code). Our instrumentation neither modified the internals of OpenStack sub-systems nor used any domain knowledge about them.

5.2.3 Evaluation Metrics

We evaluate anomaly detection with respect to the ability to properly classify the events within a trace. In particular, we evaluate the *false alarm rate* and the *hit rate* [263]. In our context, a false alarm occurs when a non-anomalous event is classified as an anomalous one, and a hit occurs when an anomalous event is correctly classified as such. The false-alarm rate is given by the total number of false alarms over the total number of non-anomalous events. The false-alarm rate should be as small as possible. The hit rate is given by the total number of hits over the total number of anomalous events. The hit rate should be as large as possible. Both metrics range between 0 and 1.

Our fault-injection experiments generated over 450 thousands events over $2,538$ execution traces, with 109 distinct event types (i.e., unique events). A key concern for evaluating anomaly detection is the need for a reliable ground truth about the actual label of the events (anomalous or non-anomalous). Unfortunately, manually assigning labels to such a large set of data is prone to errors and unfeasible in practice. Thus, we adopted an automated evaluation method and opted for conservative estimates where needed (i.e., by underestimating the accuracy of the proposed approach). Firstly, we build for each workload an anomaly detection model based on the LCS algorithm with 50 fault-free traces. Then, in order to define the ground truth of the anomalies, we run the distributed system under fault-free conditions a large number of times, generating an additional set of 500 fault-free traces, which is an order of magnitude larger than the training set of the model. Finally, we apply the LCS algorithm to these traces. Since these traces are fault-free, the differences pointed out by the LCS can be considered as false alarms. We record a *list of false-alarm event types* by adding an event type if it caused a false alarm. In total, the list includes respectively 38, 30, 18 event types for the three workloads. Instead, common events (i.e., non-anomalous) are considered as true negatives.

In our experimental evaluation, we consider an anomaly raised by a detector as a false alarm if its type belongs to the list of false-alarm event types. This method is very conservative since we are labeling all events of these types as false positives, even if these events could represent true anomalies for some experiments. This approach under-estimates the ability of the VMM at identifying true anomalies since we only take into account anomalies for events that were never affected by false alarms in our initial extensive analysis. Furthermore, our classification assumes that the LCS is not affected by false negatives, thus overestimating the accuracy of the LCS approach.

5.2.4 Experimental Results

We aim to evaluate how the probabilistic model can prevent false alarms and, at the same time, not discard hits. We analyzed the fault-injection experiments that experienced a failure (i.e., an API error to the clients, or a failure identified by our assertion checks). To provide context for the evaluation, we compare three approaches:

- ***LCS***, the baseline approach, which just aligns and compares traces (as in existing techniques based on *reference runs* [110, 135, 166]), without using a probabilistic model to account for non-determinism;
- ***LCS with VMM***, the proposed approach, which applies a Variable-

order Markov Model after LCS, as discussed in § 5.1.5;

- ***LCS with HMM***, a different probabilistic approach, which applies a Hidden Markov Model (instead of VMM) after LCS.

These approaches allow us to separately evaluate the relative influence of LCS and the probabilistic models on the accuracy of anomaly detection, pointing out any improvements due to the adoption of the probabilistic model. Moreover, we compare the accuracy of the proposed approach (VMM) with respect to a traditional probabilistic model (HMM).

We are interested in evaluating the accuracy of anomaly detection under different sizes of the training set (i.e., the number of fault-free traces). We expect that, while increasing the number of training traces, the accuracy of the approaches improves. However, since false alarm and hit rates are related and often conflicting metrics, we look for trade-offs between these metrics [116]. Thus, we use ROC curves in Figure 5.3 to represent both the metrics, computed over all experiments, and for different sizes of the training set between 5 and 50. Our evaluation deliberately targets the case of a limited training dataset, since it is typical for developers to have only a limited time budget to conduct test activities. In our case, an experiment takes on average 40 minutes (including the time to re-deploy OpenStack components, to revert the state of its databases and volumes, etc.), thus, 50 executions take about 33 compute hours, which we ran in parallel across several machines. If we used more training traces in our evaluation, the accuracy figures would not have been representative of what developers would achieve within a realistic amount of time.

The results show that *LCS with VMM* achieves a hit rate higher than 90%. The hit rate saturates around 98% when the probabilistic model is trained with 20 fault-free traces, for all workloads. This size for the training set can be considered small enough for practitioners to apply the proposed approach. The proposed approach comes with a false alarm rate

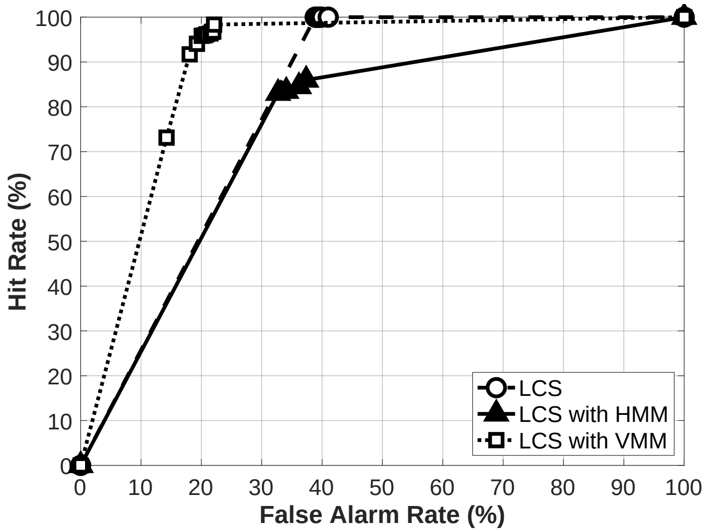


Figure 5.3. Approaches comparison.

of around 22%. This result means that the probabilistic model can discard many of the differences that are caused by non-deterministic behavior, even if a moderate amount of false alarms still needs to be tolerated by practitioners.

To put these results in context, we can compare them with the results for the *LCS* approach. The *LCS* achieves a perfect hit rate (100%) since, with our conservative evaluation, we consider this baseline approach not affected by false negatives. The false alarm rate for *LCS* is between 39-41%. The false alarm rate does not improve much by increasing the size of the training set since the LCS only identifies differences between the fault-injected trace and one *selected* fault-free trace from the training set (thus, the remaining training traces do not contribute to identifying anomalies).

The VMM is applied in pipeline after the LCS, by analyzing non-common events identified by the LCS (§ 5.1.5). Thus, the VMM can reduce the false alarm rate compared to the LCS, by classifying a "benign" non-common event as non-anomalous. However, the VMM can also reduce the hit rate, since it can classify a real anomaly as non-anomalous. Overall, the *LCS with VMM* approach achieves a better trade-off than *LCS* between a false alarm and hit rates. The loss in hit rate with respect to LCS is about 2% since a very small number of real anomalies are discarded by the VMM. At the same time, the gain in terms of false alarm rate is quite significant, since about half of the false alarms are discarded by the VMM.

The results in Figure 5.3 also point out that the *LCS with HMM* achieves worse performance than *LCS with VMM* at identifying anomalies. In our analysis, we carefully configured the HMM approach in order to perform a fair comparison against VMM (i.e., the one that gives the best results for HMM, in order to prevent any bias in favor of our proposed solution). To integrate HMM into our analysis, we configured the classification thresholds ($\epsilon_{SPURIOUS}$ and $\epsilon_{MISSING}$) by performing a preliminary calibration, and we selected the thresholds that achieve the lowest number of false positives without reducing the hit rate. Moreover, we varied the number of hidden states, ranging between 2 and 100. As in previous research that adopted HMMs, we initialized the transition and the symbol probabilities with random values [204, 232], then we used the Baum-Welch algorithm to re-estimate the parameters using the forward-backward procedure, as in the work of Batista et al. [22]. The ROC curve reports the results for the best configuration of the HMM approach.

Even if the HMM reduces the false alarms compared to the plain *LCS* approach, the false alarm rate (about 35%) is still significantly higher than the *LCS with VMM*. The hit rate for *LCS with HMM* (about 85%) is also worse than the *LCS with VMM*. We attribute this behavior to the excessive flexibility of HMMs, as they require to train a high number of parameters, which are not tuned well when using only a few tens of training fault-free traces (e.g., 50 traces are still not enough to get a good accuracy). A similar problem would occur or be even exacerbated when using other high-dimensionality models such as neural networks [167]. Instead, even with

Table 5.2. Evaluation of anomaly detection, with $n = 20$.

Workload	Approach	Avg. Hits per exp.	Avg. False Alarms per exp.
DEPL	LCS	14	92
	LCS with HMM	8	82
	LCS with VMM	13	50
NET	LCS	5	120
	LCS with HMM	5	106
	LCS with VMM	5	58
STO	LCS	22	51
	LCS with HMM	21	50
	LCS with VMM	21	25

a lower number of training traces, VMMs can achieve a better accuracy, where 20 traces suffice to reach a good trade-off between the false alarm and hit rates.

Finally, Table 5.2 shows, for each workload, the average absolute numbers of hits and false alarms per experiment, when using 20 training traces. It is interesting to notice that, for each workload, the number of false alarms is significantly higher than the number of hits. This difference points out that the injected faults lead to only a small number of anomalies, while the number of false alarms can be very high due to the non-determinism of distributed systems. These differences are higher for the DEPL and NET workloads that have a higher degree of non-determinism. Moreover, the table highlights that the VMM always provides the lowest number of false alarms regardless of the workload, with a limited loss in terms of hits.

5.2.5 Sensitivity Analysis

In the previous analysis, we adopted conservative values for the VMM thresholds ($\epsilon_{SPURIOUS} = 20\%$ and $\epsilon_{MISSING} = 80\%$), so that the approach can filter out most of the anomalies discovered by the *LCS* technique. Naturally, the choice of the thresholds can influence the number of false alarms and hits of the approach. Thus, we performed a sensitivity analysis to estimate the influence of the thresholds ($\epsilon_{\text{SPURIOUS}}$ and $\epsilon_{\text{MISSING}}$) on the hit and false alarm rates. We fixed the number of training traces to 20. We remark that, when the probability of a spurious event is higher than the $\epsilon_{\text{SPURIOUS}}$, the event is marked as non-anomalous. Similarly, a missing event is marked as non-anomalous when its probability is lower than the $\epsilon_{\text{MISSING}}$. Therefore, when $\epsilon_{\text{SPURIOUS}}$ is set to 0%, the VMM discards all anomalies, while a $\epsilon_{\text{SPURIOUS}}$ set to 100% results in not discarding any anomaly. Finally, setting the $\epsilon_{\text{MISSING}}$ to 0% implies not to discard any anomaly, and setting the threshold to 100% discards all anomalies.

▷ $\boldsymbol{\epsilon_{\text{MISSING}}}$. We first analyze the accuracy of the VMM with respect to omission anomalies. Figure 5.4 shows the rate of hits and of *true positives* (i.e., the complement of false alarms, defined as 1 − false alarm rate, for readability), by varying the $\epsilon_{\text{MISSING}}$ from 0% to 100%. We can observe that the hit rate is higher than 0,99 until a value of $\epsilon_{\text{MISSING}}$ equal to 50%. Then, the hit rate decreases slightly, until $\epsilon_{\text{MISSING}}$ reaches 90%. Finally, the hit rate decreases rapidly to 0 at 99%, since even the probability of events with high likelihood falls below the threshold. Instead, the true positive rate increases linearly after 1%, with significant improvement at 80%. Thus, $\epsilon_{MISSING} = 80\%$ is a good trade-off between hits and false alarms. The designer can fine-tune this threshold to prioritize hits over false alarms or vice versa if errors with respect to one of these metrics are not tolerated.

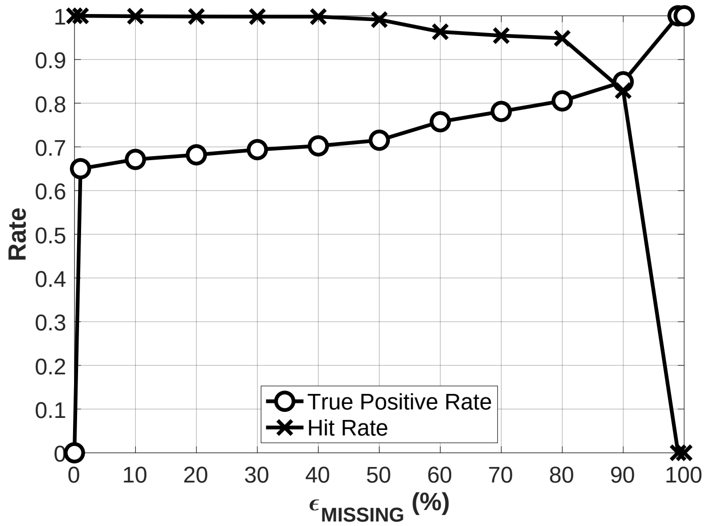
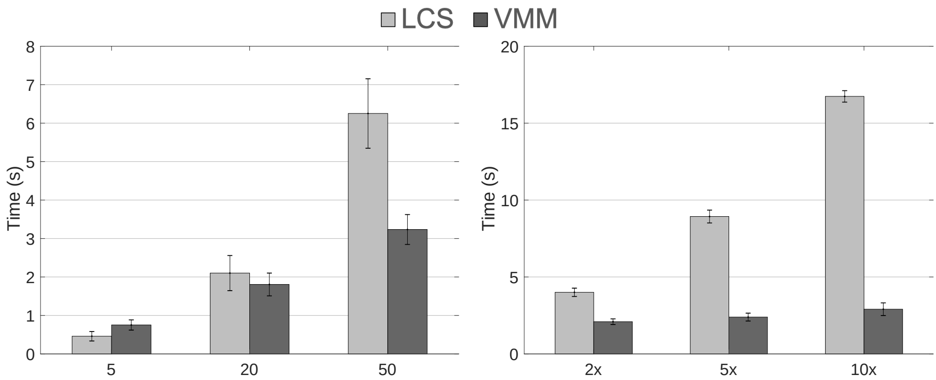


Figure 5.4. Sensitivity analysis for omission anomalies ($\epsilon_{\text{MISSING}}$).

▷ $\epsilon_{\textbf{SPURIOUS}}$. We performed the same analysis on $\epsilon_{\text{SPURIOUS}}$, not plotted for brevity. The analysis points out that the hit rate is even less sensitive rather than $\epsilon_{\text{MISSING}}$. Indeed, the hit rate only drops at 0.0 with $\epsilon_{\text{SPURIOUS}}$ equal to 0%, for which all anomalies are discarded. Given that a spurious anomaly is an event that does not normally happen under fault-free conditions, the associated symbol is never encountered in the training set. The probabilistic model assigns to it a low probability since it is inversely proportional to the size of the dictionary [51] and since we collect dozens of different symbols in our experiments. Thus, a conservative $\epsilon_{\text{SPURIOUS}}$ (e.g., 20%) is a good choice since it does not impact the hits and, at the same time, discards many false alarms.

5.2.6 Computational Cost

In this section, we evaluate the computational cost and scalability of the anomaly detection algorithm. Figure 5.5 shows the time taken to analyze event traces, for increasing volumes of data, i.e., by varying the number of traces to analyze, and the number of the events per trace.



(a) Number of training traces. **(b)** Trace size.

Figure 5.5. Execution time for *LCS with VMM*.

In Figure 5.5a, we consider the average time to apply the approach on a single test trace with a fixed number of events. The figure points out that the number of training has a higher impact on the computational time of the *LCS* technique rather than the computational time of the *VMM* technique. Indeed, the most of the time for analysis is incurred because of the search for the *selected fault-free trace*, i.e., the training trace most similar to the one under analysis (see also § 5.1.3 and Figure 5.2). Once the *selected fault-free trace* has been found, the VMM algorithm can be executed very quickly, taking about $3s$ with 50 training traces. Therefore, the analysis of even thousands of fault injection experiments can be performed in a reasonable amount of time. Since the traces can be analyzed independently from each other, they can be partitioned across several CPUs (e.g., using SMP machines): for example, in our workstation with 8 SMP cores, it takes about 40 minutes to analyze the two thousand traces that were produced by our fault injection experiments.

Finally, we analyze the impact on the execution time for applying the approach by varying the number of events per trace (see Figure 5.5b). We consider test traces of increasing size, by replicating the same sequence of events several times ($2x$, $5x$, $10x$). The execution time grows linearly, as in the previous analyses. We also found that the size of the traces has a limited impact on the computational time of the *VMM* technique.

Chapter 6

Failure Mode Analysis in Cloud Computing Systems

In this chapter, we introduce a new paradigm to data analysis for fault injection experiments, which we call *fault injection analytics*. Our approach combines *distributed tracing* to gather raw failure data, and *unsupervised machine learning* to discover the failure modes of the injected system.

The approach aims to make easier, for human analysts, the identification of the failure modes among large amounts of data produced by fault-injection experiments. When considering complex cloud systems, it is typical to perform a large number of experiments (e.g., several thousand), since these systems include tens of processes and nodes and millions of lines of source code in which faults can be injected. Moreover, for each experiment, the system generates high volumes of log files (up to hundreds of MBs) and long execution traces (e.g., thousands of events per trace). Thus, it is not feasible in practice for the analyst to analyze all of these data in a reasonable amount of time.

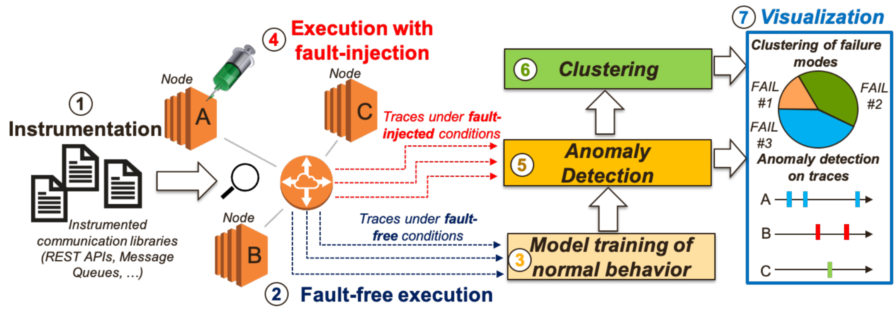


Figure 6.1. Overview of the proposed approach.

The approach combines clustering with the anomaly detection algorithm proposed in Chapter 5 in order to automatically identify the failure classes among large sets of fault injection experiments. This approach allows human analysts to find recurring failure patterns and to add new fault-tolerance mechanisms for them. It is sufficient for the analyst to only analyze one or a few experiments from the same class, thus making the analysis more efficient.

6.1 Methodology

The approach proposed in this chapter extends the anomaly detection algorithm presented in Chapter 5 by including an additional step. Indeed, the results of anomaly detection (i.e., the deviations between a fault-injected trace and the model) are the input of the clustering phase (step 6). This step aims to partition fault injection experiments in a number of groups such that experiments belonging to the same group exhibit the same anomalies (i.e., *failure mode*). Finally, the failure modes are visualized to the human analyst (step 7), by displaying the distribution of failure modes across all the experiments. Moreover, the user can focus on a specific experiment, by visualizing the anomalies of the execution over timelines. Figure 6.1 summarizes the proposed solution.

6.1.1 Failure Clustering

To identify failure modes, we perform *clustering* to group the experiments into classes (clusters), where each class represents a distinct failure mode of the system under test. In general, clustering algorithms reveal hidden structures in a given data set, by grouping "similar" data objects together while keeping "dissimilar" data objects in separated classes [258]. Formally speaking, consider a set of n distinct data objects $\{x_1, \ldots, x_n\}$ and a number of k clusters. A (hard) clustering technique assigns to each data object a label l_i representing its class, with $i \in [1, k]$ [119]. In the context of failure data, a data object represents an execution of the system while it was experiencing a fault. The i-th execution is represented by a vector of features $x_i = [f_1, \ldots, f_d]$. Each feature is a number that represents how many events of a given type occurred during the execution, with d unique types of events. The number of features easily bumps up, due to a large amount of failure data (e.g., hundreds of message types, GBs of log files, thousand of traces, and experiments).

In our context, the clustering of the experiments helps the human analyst in the identification of the failure modes and in analyzing a large amount of data of the fault-injection campaigns (hundred of MB of logs, thousand of traces and experiments, etc.).

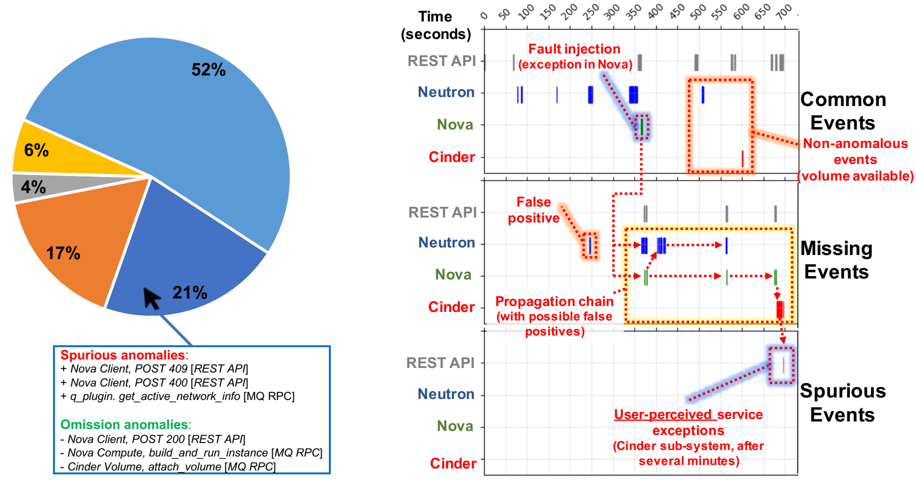
To apply the clustering, the approach represents each fault-injection experiment with a vector of features. The number of features is twice the number d of unique events (i.e., the symbols in the dictionary of events) that were traced during the experiments. Given that anomalies can be classified as spurious or missing, we include in the vector two features for each symbol: the number of times that the symbol occurred as a spurious anomaly (the first d features), and the number of times that the symbol occurred as a missing anomaly (the last d features). For example, let us suppose that the dictionary consists of three different symbols, A, B, C (i.e., a dictionary with three unique events). Let be $x_i = [1, 1, 0, 0, 2, 3]$ the vector associated to the faulty trace collected during the i^{th} experiment. These features can be interpreted as follows:

- Anomaly detection identified two spurious events, one for the symbol A and one for the symbol B.

- Anomaly detection identified five missing events, two for the symbols B and three for the symbol C.

We pre-process the vectors before clustering, by scaling down the features for the missing events, in order to give higher importance to the features that represent spurious anomalies. The preliminary selection and transformation features (*feature engineering*) is used to make the failure data more amenable for analysis [162, 269, 260]. This policy is motivated by the empirical observation that omission anomalies tend to be much more frequent than spurious anomalies since fail-stop behaviors (i.e., failure modes in which the system stops its execution) are more frequent than other failure modes. Since spurious anomalies are rarer, we want to give them more emphasis since they provide valuable information on unusual failure modes that deviate from fail-stop behaviors (e.g., when the system reacts by performing wrong operations).

This representation holds concise information about the anomalies of the experiments. Spurious events are indicators of wrong interactions that happened in the distributed system during the experiment while missing events point out actions that were not performed. We apply a clustering algorithm on these vectors, to group the experiments that exhibit similar anomalies. Thus, clusters describe distinct failure modes exhibited by the system. Our approach is not bound to a specific clustering algorithm; we rely on the anomaly detection algorithm to detect the symptoms of the failures with high accuracy, in order to favor the quality of the failure clusters.



(a) Distribution of failure modes.

(b) Anomalous events in a specific fault injection experiment.

Figure 6.2. Example of fault injection data analysis.

6.1.2 Visualization

Visualizing the execution of distributed systems is a key step to enable designers to debug failures, yet effectively summarizing information is an open research problem [26, 27, 16, 210]. Therefore, we designed a dashboard to leverage unsupervised machine learning to obtain summarized information about failure modes, in order to present them in a simplified way. The dashboard does not require the user to manually configure the failure modes, thus supporting the analysis and discovery of unknown failure modes.

Besides providing basic statistics about the experiments (e.g., number, duration), the first feedback for the user is the *distribution of failure modes* *across the fault injection experiments* (Figure 6.2a). Both the categories (i.e., the failure modes) and their sizes (i.e., the number of experiments) are automatically generated through unsupervised machine learning. In the example of Figure 6.2a, based on fault injections on the OpenStack platform, every failure mode is labeled with a summary of the spurious and omission anomalies that occurred in that failure mode. The dashboard groups the experiments into a few classes (one per failure mode) to simplify the analysis of failure modes. The user can quickly get a better understanding of each failure mode, by only looking at one or a few experiments for that class.

The dashboard also supports the user at inspecting *anomalous events that occurred within individual experiments*. When the user selects an experiment, the dashboard displays the timespans of RPCs (e.g., message queues) and REST API calls. Timespans are divided with respect to the origin of the messages, such as the Nova, Neutron, and Cinder sub-systems and external clients in the case of OpenStack. The dashboard divides interactions among three groups, as defined in § 5.1.5: *common*, *missing*, and *spurious* events. In the example shown in Figure 6.2b, the spurious events are exceptions raised by two REST API calls. The missing events are internal calls to initialize a new VM instance and to attach virtual resources to it. Due to the injected fault in the Nova sub-system, it did not complete the initialization of the instance, leaving it in an inactive state, and propagated the problem to Neutron and Cinder. The visualization supports analysts at reasoning about how to best handle faults, e.g., when in the flow of interactions, and whether to manage it in Nova, Neutron and/or Cinder.

6.2 Experimental Evaluation

In this section, we evaluate the accuracy of the proposed approach at identifying failure modes in fault injection tests. For the evaluation, we used the failure dataset described in § 5.2.2. The approach pursues this goal by *clustering* the execution traces so that the human analysts can analyze the data more easily. For example, the analyst only focuses on a sample of the experiments for each cluster instead of inspecting the whole set of experiments, which would be unfeasible for large fault injection campaigns.

We evaluate both the ability to identify the number of classes in the data (i.e., how many distinct failure modes occurred in the experiments), and to assign the fault injection experiments to the classes (i.e., the failure mode to which an experiment belongs). First, we evaluate clustering according to an *internal* criteria (§ 6.2.1), in which we assess the quality of clustering in terms of quantities that only involve the data samples. Then, we assess the quality of clustering according to an *external* criteria (§ 6.2.2), in which we compare the results of clustering against a reference classification of the data (i.e., an external ground truth). The internal evaluation assesses how well the clustering algorithm can identify the number of classes, as internal criteria are also adopted by clustering algorithms to estimate the number of classes. The external evaluation assesses how well the clustering algorithm assigns the data samples to the classes, assuming that the number of classes has been given in input to the algorithm.

We perform clustering using the vector representation of executions traces based on the VMM, as in § 6.1.1. We adopt an unsupervised clustering algorithm, the *K-Medoids* with the *squared euclidean* distance measure. The algorithm forms clusters by minimizing the sum of the dissimilarities between objects and a reference point for their cluster. Differently from the classical *K-Means*, which takes the mean value of the objects in a cluster as a reference point, the *K-Medoids* algorithm uses a *medoid*, i.e., the most centrally located object in a cluster. Thus, *K-Medoids* is less sensitive to outliers than *K-Means* [12, 245].

As a reference for the evaluation, we also analyze two alternative, simpler approaches to clustering, which we refer to as *LCS* and *SEQ*.

- *SEQ* is a baseline approach based on plain sequences of events from fault-injection experiments (i.e., it does not use anomaly detection): this approach represents each experiment with a vector of d features, where d is the number of symbols in the dictionary. Each feature represents the number of times that a specific symbol occurred during the execution. For example, let us suppose that we collected three different message types, A, B, C. Let be $x_i = [4, 2, 1]$ the vector associated to a trace collected during the i^{th} fault injection experiment. This implies that the events A, B, C were observed 4, 2 and 1 times, respectively, during the i^{th} experiment.

- *LCS* performs clustering on vector representations that are similar to the approach proposed in § 5.1, but without applying the probabilistic model. Thus, evaluating *LCS* gives information on the influence of the probabilistic model on clustering (e.g., due to fewer false anomalies, which can distort the similarity measure).

We built a ground truth for the evaluation, by performing preliminary labeling of failures. The problem of having a ground truth is a quite common open problem in all the research work dealing with log analysis. Data labeled by real system administrators represent the ideal case with the actual ground truth, but this option requires a significant resource commitment from a company. Therefore, we mitigated this problem by using the same data source that would be used by a system administrator for analyzing failures, e.g., by OpenStack logs, API Errors experienced by clients, assertion checks from OpenStack developers, anomalies in the traces, etc., to classify the experiments with respect to their failure modes, based on our previous experience with OpenStack [58]. System logs are usually good indicators of system state as they contain reports of events that occur on the several interrelated components of complex systems [142]. Previous works leveraged the collection of system logs as sources of data, which could be analyzed by a system to make it aware of its internal state [242, 3, 86, 148]. Also, to reduce the possibility of errors in manual labeling, multiple authors discussed cases of discrepancy, obtaining a consensus for the failure modes.

We found the following types of failure modes:

- **Instance Failure**: The creation of the instance fails, or the instance is created but it is in an error state.
- **Volume Failure**: The creation of the volume and/or the attach of the volume to the instance fails, or the volume is created but is in an error state.
- **Network Failure**: The creation of network resources (e.g., networks, subnets, etc.) fails.
- **SSH Failure**: The instance is correctly created and up, but it is not reachable.
- **Cleanup Failure**: The deletion of resources (previously created by the workload) fails.
- **No Failure**: There was no failure during the experiment.

Table 6.1 shows the failure modes found for each workload (i.e., 6, 4, and 4 failures mode respectively for DEPL, NET, and STO workloads) and represents our ground truth for clustering. Even if we use the same labels for the failure modes across the three workloads, each failure mode should be considered different for each workload since they involve different resources and APIs during execution (e.g., DEPL and STO have

Table 6.1. Failure Mode Classes per Workload.

Failure Mode	DEPL	NET	STO
Instance Failure	224	56	320
Volume Failure	151	-	38
Network Failure	52	30	-
SSH Failure	41	176	-
Cleanup Failure	69	-	157
No Failure	539	299	386

both cleanup failures, but with different behaviors). This classification represents our ground truth for evaluating the results of clustering.

We shared the failure dataset on GitHub[1] to help the research community in the application and evaluation of new solutions for clustering the failure modes of the systems. For every experiment of the three fault-injection campaigns, the dataset contains the events exchanged in the system and the corresponding failure label. We shared the representations of experiments with and without the anomaly detection phase.

6.2.1 Internal Evaluation

After performing fault-injection experiments, the human analyst first needs to get a qualitative understanding of how the system can fail under faults, i.e., to discover how many distinct failure modes the system exhibits. Since the analyst does not know a priori the number K of failure modes, it is part of the task of our unsupervised analysis to determine this number. A common heuristic is: (i) to configure the clustering algorithm to run with a tentative value of K; (ii) to evaluate the "validity" of the clusters, in terms of low distance between samples assigned to the same cluster, and high distance between samples assigned to different clusters; and, (iii)

[1] https://github.com/dessertlab/Failure-Dataset-OpenStack

Table 6.2. Number of clusters using the *Silhouette* index, with different clustering approaches.

Workload	Actual clusters	SEQ	LCS	LCS with VMM
DEPL	**6**	2	6	6
NET	**4**	5	3	5
STO	**4**	4	3	4

to repeat these steps for increasing values of K until the validity index reaches a "knee" point (i.e., the value of K after which the validity index significantly drops) [103].

In this evaluation, we apply the procedure described before in the same way an analyst would do (i.e., without prior knowledge of the number of clusters). We compare the number of clusters obtained with respect to our ground truth knowledge of the failure modes (i.e., 6 failure modes for DEPL, and 4 failure modes for NET and STO). We adopt the *Silhouette* index as a cluster validity technique [216], which computes the average dissimilarity between points to evaluate the cohesion of data within clusters and the separation between clusters. For a given cluster $\{\tau_k\}_{k=1}^{K}$, this method assigns to each sample $i \in \tau_k$ a measure $s_i = {}^{(b_i - a_i)}/_{max(a_i, b_i)}$ (*Silhouette width*), where a_i is the average distance between the i^{th} sample and all of the samples included in τ_k, and b_i is the minimum average distance of i to all points in any other cluster. By averaging the Silhouette width of samples in the same cluster, and then averaging these values across clusters, we obtain a *Global Silhouette value* that can be used as clustering validity index [32].

We configure the clustering algorithm with tentative values for the number of K clusters, with values between $K = 2$ and $K = 20$. Table 6.2 shows the number of clusters suggested by the *Silhouette* index, for the three vector representations and the three workloads. In the case of clustering based on *VMM*, the "knee" point matches, or is very close, to the number of clusters in our ground truth, for all of the three workloads. The other two clustering approaches (i.e., *LCS* and *SEQ*) are only accurate for some workloads but do not perform well for other ones. For example, in the case of the DEPL workload, the knee point at $K = 2$ for *SEQ* is much lower than the actual number of clusters $K = 6$ in our ground truth. For the NET and STO workloads, the validity index for *LCS* drops at $K = 3$ clusters, but clustering should find at least $K = 4$ clusters according to the ground truth. Overall, the vector representation with VMM leads to a more reliable indication of the number of clusters.

6.2.2 External Evaluation

The external evaluation assesses clustering algorithms as in a classification problem, by comparing the clusters with respect to the failure modes in our ground truth (Table 6.1). We compare, for each element in the dataset, the cluster assigned to the element with the actual class of the element, according to the ground truth. We adopt the following rule for the comparison [160]: for every cluster generated by the algorithm, we identify the ground-truth class with the largest overlap and assign every element in the cluster to the ground-truth class. In the case of a poor clustering algorithm, multiple clusters may be assigned to the same ground-truth class, but it never assigns the same cluster to multiple ground-truth classes.

In quantitative terms, let C be the number of ground-truth classes $\{\omega_c\}_{c=1}^{C}$. The *purity* of a cluster is defined as the fraction of elements in the cluster that matches the ground-truth class [257]. Assuming K clusters, for each cluster $\{\tau_k\}_{k=1}^{K}$ we define $P_k = {}^{1}/_{n_k} \cdot max(n_k^c)$, where n_k is the size of the cluster τ_k, and n_k^c is the number of elements in the cluster τ_k that belong to the class with label w_c. The overall *purity* achieved by a clustering algorithm is the weighted sum of purities across classes, given

Table 6.3. Purity of clusters, with different techniques.

Workload	SEQ	LCS	LCS with VMM
DEPL	0.74	0.91	0.94
NET	0.85	0.81	0.86
STO	0.82	0.86	0.90

by $P = \sum_{k=1}^{K} {}^{n_k}/_{n} \cdot P_k$. The larger the value of purity, the better the clustering quality.

We compute for each workload the purity obtained by the three clustering techniques. Table 6.3 shows the results. We perform 50 repetitions and compute the average value of purity across repetitions. We omit the standard deviation since it is negligible (lower than 1e−03). The results suggest that, for all workloads, the *LCS with VMM* always provides the highest purity value. Moreover, we can notice that the VMM leads to an increase in the value of purity ranging between 3% and 5% when compared to the basic *LCS* approach. The *SEQ* technique leads to worse results, especially in the case of a very stressful workload such as DEPL, where the sequence of events is longer and with more types of events. We performed the statistical hypothesis test (*Student's t-test*) to verify that differences are statistically significant: this is indeed the case, as the test rejects the null hypothesis at the 1% significance level. Thus, the proposed probabilistic model can enhance the accuracy of failure mode clustering.

6.3 Critical Consideration

The approach presented in this chapter leverages machine learning to support human analysts in identifying failure modes. From thousands of fault-injection experiments and events, the techniques identify the recurring failure modes (e.g., a dozen of clusters in our previous experience), on which the analyst can focus failure mitigation strategies.

Unfortunately, it requires careful tuning by the human analyst to achieve high accuracy. Indeed, we found that accuracy improves when weights are fine-tuned for the most important features. For example, features representing asynchronous (i.e., non-blocking) messages are more prone to be false positives and less representative of the failure modes; thus, giving a higher weight to features representing synchronous messages (i.e., blocking the caller) increase the accuracy of clustering. Similarly, spurious anomalies on REST API calls often denote exceptions raised by the system, and are more representative of the failure modes.

Human analysts must deal with hundreds of events. Some of the events are relevant symptoms of the failure mode, such as exceptions received by the client from REST API calls. Other events are not a symptom of the failure but are benign variations caused by asynchronous updates from Neutron. In order to accurately cluster this failure mode, the features representing REST API calls should be assigned a larger weight than some of the Neutron events, which are non-deterministic and are prone to noise.

The fine-tuning of weights requires considerable effort by the human analyst, which represents a significant cost and limits the usefulness of the failure mode analysis. Moreover, the tuning requires detailed knowledge of the internals of the system under test, which may be not available for large projects based on software components from different teams and third parties (e.g., commercial vendors). Thus, manual-fine tuning of feature weights is a difficult and time-consuming task, and the human analyst needs a different approach for failure mode analysis.

Chapter 7

Improving Failure Mode Analysis with Deep Learning

Failure mode analysis techniques must be robust to noise in the failure data. As shown in Chapter 6, the adoption of unsupervised machine learning techniques, such as clustering and anomaly detection, comes to the rescue but still faces some limitations. These techniques require the preliminary selection and transformation features (*feature engineering*) [162, 269, 260], to make the failure data more amenable for analysis. This effort requires deep domain knowledge and represents a significant up-front cost.

In this chapter, we propose a novel approach for efficiently identifying recurrent failure modes from failure data. The approach leverages deep learning for unsupervised machine learning, to overcome the challenges of noise and complexity of the feature space. Our approach saves the manual efforts spent on feature engineering, by using an autoencoder to automatically transform the raw failure data into a compact set of features. The approach transforms the data by jointly optimizing for the reconstruction

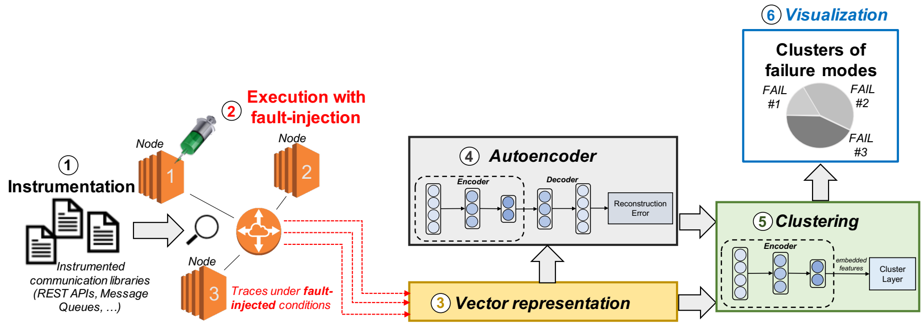


Figure 7.1. Overview of the proposed solution.

error (i.e., the transformed features are still representative of the sample) and inter-cluster variance (i.e., to make it easier to identify groups of similar failures).

7.1 Methodology

To overcome the open issues of existing techniques, we provide a novel solution to perform failure mode analysis, which does not require a manual effort by the human analyst for feature engineering. To this purpose, we use *deep learning* techniques for generating the features.

Our solution leverages *Deep Embedded Clustering* (DEC), a family of algorithms that performs clustering on the embedded features of an autoencoder [256, 89, 102, 138, 262, 101]. The application of unsupervised learning algorithms is taking place in the context of cloud computing systems since they do not require a large amount of data for training or labeled data. For example, Riganelli *et al.* [211] applied the Hierarchical Temporary Memory (HTM) - an unsupervised learning algorithm - to support online failure prediction in cloud systems.

The solution proposed in this chapter (Figure 7.1) uses DEC on the raw vector representations of the fault-injected traces, which are the same ones of the *SEQ* approach discussed in § 6.2. This proposed approach relieves the human analyst from fine-tuning the feature weights in the clustering stage, thus saving manual efforts.

An alternative version of the proposed solution is in combination with anomaly detection, by applying it on anomaly vectors, as in the *LCS with VMM* (by replacing the step 6 of Figure 6.1). In this case, the human analyst invests effort to train an anomaly detection model using fault-free traces, but without manual feature engineering. This combined approach can further improve the accuracy of failure mode analysis. We also analyze this approach in the experimental part of this work.

More in detail, DEC transforms the data with a non-linear mapping $f_\theta : X \rightarrow Z$, where θ are the learnable parameters, X is the input data (i.e., features about failure), and Z is the embedded feature space (i.e., a new, smaller set of transformed features). We apply a deep neural network (DNN) to parametrize the f_θ mapping for DEC clusters data by simultaneously learning (i) a set of k clusters centers in the embedded feature space Z, and (ii) the parameters θ of the DNN that performs the mapping between data points (i.e., the input data) and Z. DEC consists of two phases: the initialization of the parameters with a deep autoencoder and the optimization of the parameters.

7.1.1 Parameter Initialization

To initialize the parameters, we use a multi-layer deep autoencoder. An autoencoder is a neural network composed of two parts, an encoder, and a decoder. The goal of the encoder is to compress the input features to lower-dimensional features. The decoder part, on the other hand, takes the compressed features as input and reconstructs them as close to the original data as possible. Autoencoder is an unsupervised learning algorithm in nature since during training it only uses unlabelled data. Our approach applies a fully connected symmetric autoencoder since our vectors are compressed and decompressed in a specular way.

We initialize the autoencoder network layer by layer so that the layers work as a *denoising autoencoder* [246, 145] trained to reconstruct the previous layer's output after random corruption of the data. We set the network input dimension equal to d, where d is the number of the vector features (which depends on the number of unique events).

After the training, we concatenate all the layers of the encoder followed by the layers of the decoder, to form a multi-layer deep autoencoder with a bottleneck coding layer in the middle. All layers of the neural network are densely (fully) connected. Our solution is intentionally meant to adopt a typical and regular DNN architecture, to avoid hand-tuning by the human analyst as much as possible. Thus, the value d is the only parameter that depends on specific the failure dataset under analysis.

The autoencoder is trained to minimize the reconstruction loss. Then, we discard the decoder layers, and we apply the encoder layers as our initial mapping between the data space and the feature space.

To start the clustering phase, we need to initialize the cluster centers. Therefore, we firstly input the initialized DNN with the data points to get embedded data points, and then apply a clustering algorithm in the feature space Z to obtain k initial centroids. Our solution adopts the *K-Medoids*, a clustering method that performs the clustering phase by minimizing the sum of the dissimilarities between objects and a reference point for their cluster. As a reference point, this method uses the *medoid*, i.e., the most centrally located object in a cluster. Therefore, this method is considered less sensitive to outliers than the classical *K-Means*, which takes the mean value of the objects in a cluster as a reference point [12, 245].

7.1.2 Parameter Optimization

The approach trains the non-linear mapping f_θ with two joint objectives: the DNN minimizes the reconstruction error; and, it maximizes inter-cluster variance in the embedded feature space. Towards these goals, the approach alternates between (i) computing a "soft" assignment between the current cluster centroids and the embedded data samples (i.e., a vector of probabilities that the sample is a member of each cluster); and (ii) updating the mapping f_θ and the cluster centroids to maximize inter-cluster variance. We repeat the process until meeting a convergence criterion.

To measure the similarity between the embedded data points and the k centroids, we build a custom layer, named *cluster layer*, to convert the input features to cluster label probability. To quantify the similarity between every embedded point and a centroid (i.e., to assign the probability in the soft assignment), we computed the *Student's t-distribution*.

Then, we recompute the clusters iteratively by learning from the current soft assignment. In particular, the clustering model is trained to minimize the distance between the soft assignments and an artificial "target" distribution, which is a transformed version of the probabilities in the soft assignment that widens the gap between the probabilities [195]. In our case, we compute the target distribution by raising the soft assignments to the second power and normalizing the values. The approach gives more emphasis on data points assigned with high probability, and at the same time, it also optimizes for the ones with low probability. By optimizing for the low distance between the actual soft assignments and the target distribution, we obtain clusters with larger intra-cluster variance, thus improving the cluster quality.

For the optimization, we minimize the *Kullback–Leibler divergence* (KL) between the soft assignments and the target [118]. The KL divergence is a loss function that measures the difference between two distributions. We update the target distributions after a specific number of clustering iterations. The clustering model is then trained to minimize the KL divergence loss between the output of the clustering and the target distribution. We leveraged the Stochastic Gradient Descent (SGD) with momentum [203] to optimize simultaneously both the cluster centers and the DNN parameters. The parameter optimization process stops when a percentage of points below a *convergence threshold* changes the assigned cluster between two iterations in a row. We set the convergence threshold equal to 0.1%.

7.2 Experimental Evaluation

In this section, we evaluate the proposed approach in the context of failure data from the OpenStack cloud computing platform. For the evaluation, we used the failure dataset presented in § 5.2.

We evaluated our solution in two scenarios:

- The deep neural network technique is applied on the raw failure data, without performing any anomaly detection. This is the same data as in the *SEQ* approach (see § 6.2).
- The deep neural network technique is applied on top of anomaly detection, i.e., on the anomaly vectors. This is the same data generated by the *LCS with VMM* approach (see Section 6.1).

For each of these cases, we compare the proposed approach (*DEC*) against baselines, in which we apply traditional clustering. For the baselines, we consider both the case of plain features (*k-medoids w/o fine-tuning*), and a manual fine-tuning of the weights of the features (*k-medoids with fine-tuning*). We remark that the fine-tuning of the features is a difficult and time-consuming task, due to the exploration of a large number of features (hundreds of event types) and the deeper study of event types in OpenStack (e.g., synchronous and asynchronous events, missing and spurious events, RPC messages and REST APIs, etc.). This exploratory data analysis was performed with Matlab code and took around two weeks of manual effort.

To evaluate different use-cases and conditions, we applied our solution to perform clustering on the data from the three fault-injection campaigns, one for each workload. The input data X is a matrix with the number of rows equal to the number of fault-injection experiments. The columns are dependent on the number of different event types d observed during the execution of the workload. In particular, the number of columns is d when the clustering is applied without the help of the anomaly detection, and $2d$ when the clustering is applied with the anomaly detection (since the algorithm discerns the spurious events from the omitted ones, as explained in § 5.1.

We set the hyper-parameters to minimize the reconstruction loss. During the phase of pre-training, we performed a basic tuning of the parameters following the common practices of previous studies [157, 128]. We randomly initialized the weights of the layers. The layers were pre-trained for $100,000$ iterations and a drop-out rate set to 20%. We trained DEC with additional $100,000$ iterations but without a drop-out rate. We set the size of the mini-batch to 256, the starting learning rate to 10%, which is divided by 10 every $20,000$ iterations, and the weight decay to 0 [256]. For each dataset, we tuned the autoencoder by configuring the number and the dimension of the inner layers (between 2 and 4 layers, of decreasing dimension from d to K), and the distance metric for clustering ($L1$, city block, and $L2$, euclidean). Moreover, to initialize the centroids of the clusters, we selected the best solution after running the k-medoids with 30 repetitions.

To evaluate the quality of the clustering, we compare the cluster assigned to the experiment with the failure class of the experiment defined in our ground truth (Table 6.1). To associate the clusters to the failure classes, we identify, for every cluster, the failure label with the largest overlap and assign every element in the cluster to the ground-truth class [160], as also described in §6.2. We remark that this evaluation is conservative

Table 7.1. Purity values of clustering without performing anomaly detection (*SEQ* data). Bolded values are the best performance.

Clustering Approach	DEPL	NET	STO
k-medoids w/o fine-tuning	0.70	0.80	0.80
k-medoids with fine-tuning	0.74	0.85	0.82
DEC	**0.86**	**0.86**	**0.92**

Table 7.2. Purity values of clustering on top of anomaly detection (*LCS with VMM)*. Bolded values are the best performance.

Clustering Approach	DEPL	NET	STO
k-medoids w/o fine-tuning	0.80	0.78	0.87
k-medoids with fine-tuning	**0.94**	**0.86**	**0.90**
DEC	0.84	0.83	0.89

since it can assign multiple clusters to the same ground truth, but it can not associate the same cluster to different classes of failure.

Table 7.1 and Table 7.2 show the clustering results, in terms of purity, without and with anomaly detection, respectively. The results without anomaly detection (Table 7.1, *SEQ* data) show that the use of the DEC achieves a higher purity compared to traditional clustering, both without and with fine-tuning of feature weights. This behavior applies to each of the three workloads. The scenario without anomaly detection is the most important one since it is the case of the busy system designer that needs quick feedback from fault injection tests, to quickly perform the next iteration of development. For example, the designer may add or revise fault-tolerance mechanisms, and test them again on a new round of fault injection experiments. In these cases, avoiding training an anomaly detection model is useful to speed up data analysis.

In the case of clustering in combination with anomaly detection (Table 7.2, data from *LCS with VMM*), the data have already been processed and reduced before clustering. Therefore, clustering achieves better results than using data without anomaly detection. In particular, clustering benefits most in the case of manual fine-tuning of the feature weights, as *k-medoids with fine-tuning* always achieves better results than both the basic *k-medoids w/o fine-tuning* and *DEC*. However, these better results come at the cost of manually setting the weights of the features, which requires a deep knowledge of the system internals, and efforts to best tune them concerning the specific workload. Instead, the *DEC* approach achieves performance that is close to the case of fine-tuning, with significantly less effort from the human analyst. Moreover, *DEC* always returns better results than the basic *k-medoids*, consistently over all the workloads, and both with and without anomaly detection. Our experiments also pointed out that the standard deviation is below 5%, and data are normally distributed around the mean.

To better understand the impact of the clustering on the analysis of failure modes, we inspected the distribution of the failure data samples across the clusters and compared it to the distribution of the actual failure modes (Table 6.1). Ideally, the distribution across clusters matches the actual failure modes, so that the human designer can prioritize the development of fault tolerance mechanisms according to the distribution. Moreover, it is sufficient for the human designer to only analyze one or a few experiments from the same class, thus making the analysis more efficient. To map the clusters to the failure modes of Table 6.1, we followed the approach described in § 6.2. We remark that this analysis does not focus on the quality of clusters (i.e., samples misclassified in the wrong cluster), as the previous analysis already provided figures about the purity of the clusters. Here, we focus on the distribution of the clusters that would be presented to the human designer, as the shape of the distribution influences the interpretation of the failure data.

Figure 7.2 shows the distributions of the clusters for the proposed approach (*DEC*), for the baselines (*k-medoids* with and without fine-tuning), and the actual distribution of the failure modes according to the ground truth. The size of the clusters for *Instance Failure*, *Network Failure*, and *Cleanup Failure* from the clustering techniques are close to the actual frequency of these failure modes. Instead, there are noticeable differences for the remaining failure modes. In the case of *Volume Failure*, the *k-medoids w/o fine-tuning* misses this failure mode, while the cluster from *k-medoids with fine-tuning* is only half of the actual frequency of this failure mode. In the case of *SSH Failure*, which accounts for a minor part of the failures, all of the clustering approaches do not report any failure. We do not attribute this result to the clustering techniques, but to the similarity of events occurring in this failure mode to the ones occurring for *Instance Failure*, which misleads clustering. Instead, we believe that this failure mode could be better analyzed by looking not only at the execution traces but also at additional information sources, such as system logs. Finally, both *k-medoids* with and without fine-tuning over-estimate the cases of *No Failure*, as they report several hundreds of no-failures more than the actual size of this class. This error is the most severe one since it misleads the human designer at believing that the system fails less frequently than the actual truth (e.g., about -20% of neglected failures). Thus, with the simple *k-medoids*, the analyst would unjustly trust the reliability of the system. Instead, in the proposed approach, the share of cases of *No Failure* is close to the ground truth.

We evaluated the computational cost of the proposed approach to estimate the overhead introduced by the use of deep learning to cluster the failure data. We performed several evaluations, by varying the workloads, the vector representation of the experiments (i.e., with and without the anomaly detection), and the layers of the neural network. We found that

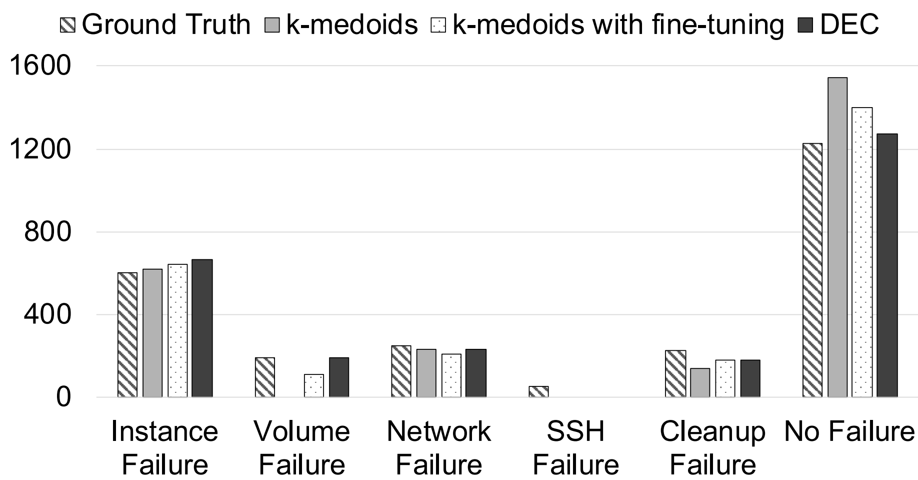


Figure 7.2. Distribution of failure modes from different clustering techniques (*SEQ* data).

the use of DEC for clustering introduces an average overhead of ~ 23 seconds compared to the basic use of the k-medoids. This time includes the initialization of the cluster centers with k-medoids (i.e., the parameter initialization) and the training of the DNN (i.e., the parameter optimization). The standard deviation is high ($\sim 75\%$ of the average value) since the configuration of the DNNs impacts the computational cost. Nevertheless, the overhead introduced by DEC can be considered acceptable, given that the proposed solution avoids the manual fine-tuning of features, which represents a difficult and time-consuming task.

This page intentionally left blank.

Chapter 8

Runtime Failure Detection in Cloud Computing Systems

Runtime verification strategies, a key technique to identify the failures at runtime, perform redundant, end-to-end checks (e.g., after service API calls) to assert whether the virtual resources are in a valid state. For example, these checks can be specified using temporal logic and synthesized in a runtime monitor [66, 41, 272, 206], e.g., a logical predicate for a traditional OS can assert that a thread suspended on a semaphore leads to the activation of another thread [10]. Runtime verification is now a widely employed method, both in academia and industry, to achieve reliability and security properties in software systems [17]. This method complements classical exhaustive verification techniques (e.g., model checking, theorem proving, etc.) and testing.

In this chapter, we propose a lightweight approach to runtime verification tailored for the monitoring and analysis of cloud computing systems. We used a non-intrusive form of tracing of events in the system under test, and we build a set of lightweight monitoring rules from correct executions of the system in order to specify the desired system behavior. We synthesize the rules in a runtime monitor that verifies whether the system's behavior follows the desired one. Any runtime violation of the monitoring rules gives a timely notification to avoid undesired consequences, e.g., non-logged failures, non-fail-stop behavior, failure propagation across subsystems, etc. The approach does not require any knowledge about the internals of the system under test and it is especially suitable in multi-tenant environments or when testers may not have a full and detailed understanding of the system. We investigated the feasibility of our approach in the OpenStack cloud management platform, showing that the approach can be easily applied in the context of an "off-the-shelf" distributed system. In order to preliminary evaluate the approach, we executed a campaign of fault-injection experiments in OpenStack. Our experiments show that the approach can be applied in a cloud computing platform with high failure detection coverage.

8.1 Methodology

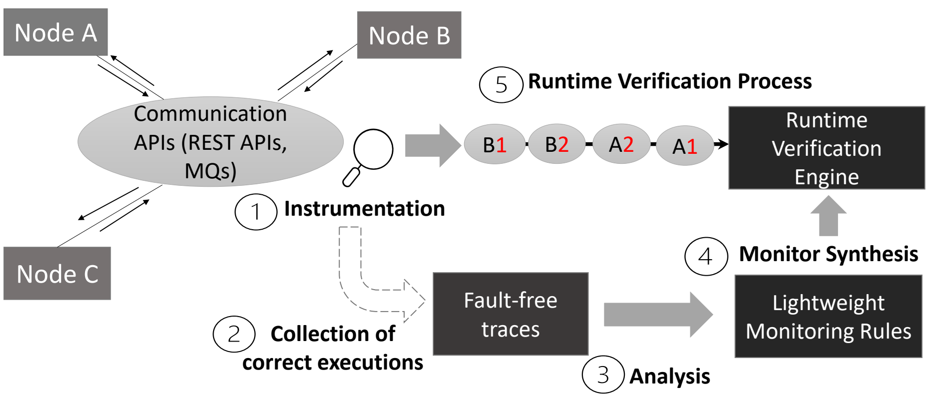


Figure 8.1. Overview of the proposed approach.

Figure 8.1 shows an overview of the proposed approach. Firstly, we instrument the system under test to collect the events exchanged in the system during the experiments (step ①). This instrumentation is a form of *black-box tracing* since we consider the distributed system as a set of black-box components interacting via public service interfaces. To instrument the system, we do not require any knowledge about the internals of the system under test, but only basic information about the communication APIs being used. This approach is especially suitable when testers may not have a full and detailed understanding of the entire cloud platform. Differently from traditional distributed system tracing [182], this lightweight form of tracing does not leverage any propagation of the event *IDs* to discriminate the events generated by different users or sessions.

In the step ②, we collect the *correct executions* of the system. To define its normal (i.e., correct) behavior, we exercise the system in "fault-free" conditions, i.e., without injecting any faults. Moreover, to take into account the variability of the system, we repeat several times the execution of the system, collecting different "*fault-free traces*", one per each execution. We consider every fault-free trace a *past correct execution* of the system.

Step ③ analyzes the collected fault-free traces to define a set of *failure monitoring rules*. These rules encode the expected, correct behavior of the system, and detect a failure if a violation occurs. This step consists of two main operations. Firstly, the approach extracts only the attributes useful for expressing the monitoring rules (e.g., the name of the method, the name of the target system, the timestamp of the event, etc.). Then, we define the failure monitoring rules by extracting "*patterns*" in the event traces. We define a "*pattern*" as a recurring sequence of (not necessarily consecutive) events, repeated in every fault-free trace, and associated with an operation triggered by a workload. In this work, we identify patterns by manually inspecting the collected traces. In future work, we aim to develop algorithms to identify patterns using statistical analysis techniques, such as invariant analysis [78, 261, 92]. We assume that the format of these rules can detect many of the failures that appear in cloud computing systems: if at least one of the rules is violated, then a failure occurred.

Finally, we synthesize a monitor from failure monitoring rules, expressed according to a specification language (step 4). The monitor takes as inputs the events related to the system under execution, and it checks, at runtime, whether the system's behavior follows the desired behavior specified in the monitoring rules (step 5). Any (runtime) violation of the defined rules alerts the system operator of the detection of a failure.

8.2 Monitoring Rules

In general, we can express a monitoring rule by observing the events in the traces. For example, suppose there is an event of a specific type, say A, that is eventually followed by an event of a different type, say B, in the same user session (i.e., same ID). The term *event type* refers to all the events related to a specific API call provided by a specific service. This rule can be translated into the following pseudo-formalism:

$$a \rightarrow b \ and \ id(a) = id(b), \quad with \ a \in A, \ b \in B \tag{8.1}$$

The rules can be applied in the multi-user scenario and concurrent systems as long as the information on the user IDs is available. However, introducing an ID in distributed tracing requires both an in-depth knowledge about the internals and intrusive instrumentation of the system. Therefore, to make our runtime verification approach easier to apply, we propose a set of coarse-grained monitoring rules (also known as *lightweight monitoring rules*) that do not require the use of any propagation ID. To apply the rules in a multi-user scenario, we define two different sets of events, A and B, as in the following.

$$
\begin{aligned}
A &= \{all\ distinct\ events\ of\ type\ \text{“}A\text{”}\ happened\ in\ [t,\ t+\Delta]\} \\
B &= \{all\ distinct\ events\ of\ type\ \text{“}B\text{”}\ happened\ in\ [t,\ t+\Delta]\}
\end{aligned}
\tag{8.2}
$$

with $|A| = |B| = n$. Our monitoring rule for the multi-user case then asserts that there should exist a binary relation R over A and B such that:

$$
\begin{aligned}
R = \{(a,b) \in A \times B \mid a \to b,& \\
&\nexists\ a_i, a_j \in A,\ b_k \in B \mid (a_i, b_k), (a_j, b_k), \\
&\nexists\ b_i, b_j \in B,\ a_k \in A \mid (a_k, b_i), (a_k, b_j)\ \}
\end{aligned}
\tag{8.3}
$$

with $i, j, k \in [1, n]$. That is, every event in A has an event in B that follows it, and every event a is paired with exactly one event b, and viceversa. These rules are based on the observation that, if a group of users performs concurrent operations on shared cloud infrastructure, then a specific number of events of type A is eventually followed by the same number of events of type B. The idea is inspired by the concept of flow conservation in network flow problems. Without using a propagation ID, it is not possible to define the couple of events a_i and b_i referred to the same session or the same user i, but it is possible to verify that the total number of events of type A is equal to the total number of events of type B in a pre-defined time window.

In the context of OpenStack, users perform their requests via Dashboard or command line by using the API provided from the client-side of the project (e.g., novaclient). The OpenStack API is implemented as a set of web services in the Representational State Transfer (REST) architectural style. An interaction with one of the services takes the form of sending a request (including formatted data) to a particular address on the server and then parsing the response. The REST calls are based on the

HTTP protocol. Hence, we can discern information such as the method invoked (i.e., GET, DELETE, POST, PUSH, etc.), the client performing the request (i.e., cinderclient, neutronclient, and novaclient), and the response status code (i.e., 2xx for successful requests, 4xx for client errors, 5xx for server-side errors, etc.). In the case of the REST API, an event type is identified as the pair client performing the request and the method invoked (e.g., `<novaclient, GET>`).

On the other hand, OpenStack's internal services communicate via RPC messages to serve users' requests. These messages contain fields such as the method invoked, the caller (the system's service), and the body of the message, which further contains useful information to serve the request. For example, if a user aims to launch an instance from an image by using the command line, he runs the command `openstack server create`, which sends the request to Nova subsystem [177]. Once the request is received, Nova starts communicating with other subsystems to serve the request, i.e., the subsystem exchanges messages with Keystone to verify user's authentication, Glance to get the image, Neutron to create networks, and sub-networks, and Cinder for the volume attachment. In the case of the RPC calls, an event type is a pair consisting of the service providing the API and method invoked (e.g., `<cinder-volume, create_volume>`)

The identification of the event (or more events) characterizing the action taken by the user is needed to specify a monitoring rule. The previous example helps to understand that, the event flow used to serve users' requests starts from a REST API call. However, there is not a one-to-one relationship between the first RPC message of the event flows and the REST API starting the request. Indeed, we observed that the first RPC message exchanged among the subsystems is the method `schedule_and_build_instances` of the `nova-conductor` component. This events arises from the POST method called by `novaclient`, but not every `novaclient POST` generates the `schedule_and_build_instances` event since the same POST method can be used to create or add different resources. Therefore, we can not identify a pattern starting from a REST API call, but we can instead leverage an RPC message for this purpose. For example, if we observe the event `schedule_and_build_instances` from `nova-conductor`, we can realize that the user's request is the creation of an instance. Similarly, if we observe the event `create_volume` of the `cinder-scheduler` component, we derive that the user requested the creation of a volume, and so on. In the following, we name the events heading the flow of RPC messages as *head event*.

To derive the monitoring rules, we based the analysis on finding the patterns of events starting with a head event. The key idea is that, if we can link a pattern of recurring events following a specific head event, then we can derive rules to monitor user activities and identify anomalies (e.g., out-of-order events, missing events, etc.). Unfortunately, due to the non-determinism of the cloud systems, we can not manually infer rules by simply observing fault-free executions. Indeed, the head event starting from a user request is not necessarily followed by the same number and the same order of events. Actually, this is seldom the case. Moreover, the high volume of messages in the system makes the manual inspection very difficult and prone to error.

Therefore, we investigate the additional information contained in the body of the RPC messages since it contains a rich set of information. To automatically derive the fields of interest from the body, we analyzed the trace executions of the system and applied a set of filters on the fields of the body message, as shown in the Algorithm 1. This analysis allowed us to pinpoint the Oslo Context variables, i.e., the variables used to provide context-aware log records when specifying a request and, thus, to maintain useful information about a request context. Among these variables, the analysis highlighted the `_context_request_id`, i.e., the identifier of a request and the `_context_global_request_id`, i.e., a request-id which

Algorithm 1 Pseudocode for the identification of the body fields

```
J: number of messages in the trace
for j in 1..J do
    if trace(j).is_rpc then
        for field in trace(j).body.keys do
            fields_list = fields_list ∪ field
for field in fields_list do
    for j in 1..J do
        if field ∈ trace(j).body.keys then
            not_null_count[field]++
            value = trace(j).body[field]
            if value == None then
                none_count[field]++
            else
                not_null_none_count[field]++
                values = values ∪ value
                counter[value]++
    values_count[field] = values.length
    maximum_value_repetition[field] = counter.maximum
    average_value_repetition[field] = counter.average
for field in fields_list do                                   ▷ Filtering fields
    if (not_null_none_count[field] >= not_null_none_threshold) AND (values_count[field] >= values_count_threshold) AND (maximum_value_repetition[field] >= maximum_value_repetition_threshold) AND (average_value_repetition[field] >= average_vale_repetition_threshold) then
        filtered_fields_list = filtered_fields_list ∪ field
```

Algorithm 2 Pseudocode of the Pattern Finder

```
I: number of trace executions
J: number of messages in the trace
for i in 1..I do
    for j in 1..J do
        p = trace(i, q) | ∃ trace(i, q') ∈ p | trace(i, q).timestamp <= trace(i, q').timestamp + window_size AND trace(i, q).REQUEST_ID = trace(i, j).REQUEST_ID
        gp = trace(i, q) | ∃ trace(i, q') ∈ p | trace(i, q).timestamp <= trace(i, q').timestamp + window_size AND (trace(i, q).REQUEST_ID = trace(i, j).REQUEST_ID OR trace(i, q).GLOBAL_REQUEST_ID = trace(i, j).REQUEST_ID)
        patterns = patterns ∪ p
        global_patterns = global_patterns ∪ gp
        frequencies[p]++
        frequencies[gp]++
```

may have been sent in from another service to indicate this is part of a chain of requests [181]. These variables allow us to identify the flow of RPC messages related to the same user's request and, consequently, to recognize the head event of the flow.

To automatically infer the monitoring rules, we use an algorithm that analyzes the logs collected during the fault-free executions of the system. As described in Chapter 3 and Chapter 5, we use Zipkin distributed tracing system to collect the logs of the system's execution. These logs are aggregate information in JSON (JavaScript Object Notation) format. We use a parser that creates a file in CSV (Comma Separated Value) format containing the timestamp of the events, the name of the method called, and the component invoking the method, the status code of the REST API, and the context variables contained in the body of the RPC messages. The context variables are used to discover a flow of related events (i.e., a pattern). To properly identify the patterns, we use a specified temporal window that allows us to find related events whose difference of timestamp is not greater than the duration of the temporal window. Indeed, since the executions of the system under test contained in the logs can last tens of minutes, sometimes hours, it is meaningless to link different messages that are too temporally distant. When the pattern in the temporal window is found as common among all fault-free executions, then we derive a monitoring rule. The Algorithm 2 shows the pseudo-code used to automatically infer the patterns from the fault-free executions.

The analysis of fault-free executions pointed out different types of monitoring rules. Suppose to observe three different RPC events in a specified temporal window, say a, b, c belonging to three different event types, say A, B, C, respectively. Suppose also that the event a is the head event, i.e., the occurrence of this event identifies a pattern of events that follow the heading one. We categorize the monitoring rules as follows:

■ **Ordered-Events Rules** (ORD): Rules based on a flow of events that follows always the same order and occurrence. For example, the event b and c follow a always with the same pattern (e.g., $a \rightarrow b \rightarrow c$). These rules interest the services less affected from the non-determinism and where it is possible to find a fixed pattern for the same operation. The ORD rules can identify failures as out-of-order or missing events in the pattern. The Algorithm 3 shows the pseudocode for the ORD rules.

■ **Occurred-Events Rules** (OCC): Rules based on a flow of events that occur after the head event, but without following any specific order and/or occurrence. For example, the event type b happens before or after the event type c in different executions (e.g., $a \rightarrow b \rightarrow c$, or $a \rightarrow c \rightarrow b$), or with different occurrences (e.g., $a \rightarrow b \rightarrow b \rightarrow c$, or $a \rightarrow b \rightarrow c \rightarrow b$, etc.). These rules are frequent when non-determinism affects the flow of events and it is not possible to identify a fixed pattern among all the executions. A failure can cause a missing or an out-of-order event in the pattern. The OCC rules are robust to the identification of failure due to missing events in

Algorithm 3 Pseudocode for the ORD Rules

```
for p in patterns do
    frequence = 0
    for i in 1..I do                                  ▷ i: trace execution index
        for j in 1..J do                              ▷ j: row of the trace execution
            if trace(i, j).rpc_type == p.first_line.rpc_type then
                frequence++
    if frequences[p] / frequence >= threshold then
        ord_rules = ord_rules ∪ p
for gp in patterns do
    frequence = 0
    for i in 1..I do
        for j in 1..J do
            if trace(i, j).rpc_type == gp.first_line.rpc_type then
                frequence++
    if frequences[gp] / frequence >= threshold then
        ord_rules = ord_rules ∪ gp
```

the pattern, but not to the out-of-order events since there is not a specific ordering involved in the rule. The Algorithm 4 shows the pseudocode for the OCC rules.

■ **Counted-Events Rules** (COUNT): Rules based on the assumption that an event is repeated several times varying in a range of value (e.g., $min < a < max$, where min and max represent the minimum and the maximum number of times the event is repeated in fault-free conditions, respectively). The COUNT rules can be applied when the system is trying to serve a request involving multiple-repeated operations (e.g., polling requests) but it is not able to complete the action. This leads to an anomalous repetition of events (e.g., $a > max$). The Algorithm 5 shows the pseudocode for the COUNT rules.

While the ORD and OCC rules are based on a flow of events following the head event a, the COUNT rules are based on the repetition of the

Algorithm 4 Pseudocode for the OCC Rules

```
for p in patterns do
    total_pattern_frequence = ∑ frequencies of patterns which are permutations of p
    for line in p do
        frequence = 0
        for i in 1..I do
            for j in 1..J do
                if trace(i, j).rpc_type == line.rpc_type then
                    frequence++
        if total_pattern_frequence / frequence >= threshold then
            occ_rules = occ_rules ∪ p
for gp in global_patterns do
    total_pattern_frequence = ∑ frequencies of patterns which are permutations of p
    for line in gp do
        frequence = 0
        for i in 1..I do
            for j in 1..J do
                if trace(i, j).rpc_type == line.rpc_type then
                    frequence++
        if total_pattern_frequence / frequence >= threshold then
            occ_rules = occ_rules ∪ gp
```

Algorithm 5 Pseudocode for the COUNT Rules

```
for p in patterns do
    if p.has_all_lines_with_same_rpc_type then
        minimum = p.line
        maximum = p.len
        for q in patterns do
            if q.has_all_lines_with_same_rpc_type then
                if q.rpc_types = p.rpc_types then
                    if if q.len > maximum then
                        maximum = q.len
                    if q.len < minimum then
                        minimum = q.len
        rep_rules = rep_rules ∪ (p.first_line, minimum, maximum)
for gp in global_patterns do
    if gp.has_all_lines_with_same_rpc_type then
        minimum = gp.len
        maximum = gp.len
        for q in global_patterns do
            if q.has_all_lines_with_same_rpc_type then
                if q.rpc_types = gp.rpc_types then
                    if q.len > maximum then
                        maximum = q.len
                    if q.len < minimum then
                        minimum = q.len
        rep_rules = rep_rules ∪ (gp.first_line, minimum, maximum)
for p in patterns do                                    ▷ Pattern Reduction
    if ∃ q ∈ patterns t.c. q ⊂ p then
        frequencies[q] += frequencies[p]
        patterns = patterns \ p
for gp in global_patterns do
    if ∃ q ∈ patterns t.c. q ⊂ gp then
        frequencies[q] += frequencies[gp]
        global_patterns = global_patterns \ gp
```

head event in a specific range of occurrences. This difference makes the runtime verification of the COUNT rules difficult in practice. Let's make a simple example. Suppose to have n different users, with $n > 1$, performing the same action concurrently. In the case of ORD and OCC rules, if we observe n times the same head event, then the same monitoring rule is activated the same number of times and controls if the patterns are verified n times. In the case of COUNT rules activated concurrently, since the occurrences of the head event will presumably be higher than the threshold (the max value) defined in the rule, then the approach will raise an exception, resulting in a false alarm case. It is clear that, in order to make the COUNT rules effective in practice, we need to discern the concurrent requests activating these rules. We address this issue by looking at the resource involved in the user's requests (e.g., the id of the network, the id of a device, etc.) and contained in the body of the RPC messages. Therefore, if we observe head events activating the same COUNT rule by targeting n different resources, then we derive that the number of head events should range in $(n * min < |a| < n * max)$.

In addition to the monitoring rules based on RPC messages, we infer a further type of monitoring rule based on the status code of the REST API. Indeed, a POST or a PUT method with a status code 4xx (client error), and 5xx (server error) is an indication of the incapability of the client in performing the request and the incapability of the server to fulfill the request, respectively. These events can not be observed during fault-free executions since they are failure symptoms, but they are common in faulty conditions, i.e., when the fault is injected into the system.

8.3 Experimental Evaluation

In this section, we infer the monitoring rules in the context of the OpenStack cloud computing platform. We targeted OpenStack version 3.12.1

(release *Pike*), deployed on Intel Xeon servers (E5-2630L v3 @ 1.80GHz) with 16 GB RAM, 150 GB of disk storage, and Linux CentOS v7.0, connected through a Gigabit Ethernet LAN.

8.3.1 Multi-tenant Workload

In the previous chapters, we used single-user workloads to produce failure data and evaluate the anomaly detection approach (Chapter 5), and the failure mode analysis approaches (Chapter 6, and Chapter 7). Although these workloads consist of a large set of operations stressing the OpenStack subsystems, they do not include concurrent operations performed by different users. We believe that to properly evaluate the monitoring rules in the context of cloud infrastructures, we need to take into account these situations. Hence, we used a multi-tenant workload including different users performing concurrent operations on the infrastructure. The users have 6 different behaviors, which are specified by 6 different sub-workloads, described in the following:

- ***Volume***: The user performs operations strictly related to the block storage (Cinder service).
- ***Instance***: The user's requests are related to the Nova service for the creation of the instance.
- ***Network***: The user simply creates network resources (networks, sub-networks, IP addresses, routers, etc.), stressing the Neutron service.
- ***Instance Volume***: The user creates an instance from an image, then a volume.
- ***Volume Instance***: The user creates an instance from a volume using a different API from the previous sub-workload.

- ***DEPL***: The single-user workload used in the previous chapters (Chapter 5, 6, 7) that stresses Nova, Cinder and Neutron services.

The six behaviors are run concurrently to obtain the multi-tenant workload. The *Volume*, *Network*, *Instance Volume*, and *Volume Instance* workloads are run twice by different users. Therefore, the multi-tenant workload includes 10 different users running concurrently.

8.3.2 Fault-free Analysis

We applied the algorithm described in § 8.2 on 100 fault-free traces, collected by executing the system with the multi-tenant workload and without injecting any fault. To set the length of the time window, we made a conservative choice by setting the time window equal to the longest duration of the actions observed during the execution of the workload in fault-free conditions (~ 35 seconds).

We derived 7 types of monitoring rules based on RPC messages, related to different actions, regardless of the user starting the request. Table 8.1 summarizes the rules. Rules # 1, 2, and 3 refer to the creation of the instance, volume, and network, respectively, and thus cover the most frequent actions on an Infrastructure-as-a-Service platform, i.e., the creation of the resources. We notice that the rules related to the creation of the instance and volume are of type ORD, while the one related to the creation of the network is OCC due to the asynchronous nature of the Neutron subsystem. We found also the rule for the attachment of the volume to an instance (Rule # 4), which is another relevant and frequent operation performed by users to use the created instance, and for the deletion of the instance (Rule #5). Moreover, we derived two further rules related to the network operations (Rules # 6 and 7), i.e., the update of the security groups, the sets of IP filter rules that are applied to all project instances, and define networking access to the instance, and the operation of pinging the instance via SSH, which is the only rule of type COUNT.

Table 8.1. Monitoring Rules.

Rule #	**Rule Description**	**Rule Type**	**# of Events**	**Subsystems**
1	*Instance Creation*	ORD	4	Nova
2	*Volume Creation*	ORD	2	Cinder
3	*Network Creation*	OCC	3	Neutron
4	*Volume Attachment*	ORD	4	Nova, Cinder
5	*Instance Deletion*	ORD	3	Nova
6	*Security Group Update*	ORD	2	Neutron
7	*Ping Instance via SSH*	COUNT	6-26	Neutron

We notice that the monitoring rules based on RPC messages do not cover all the possible operations of the workload. The volume deletion and the instance reboot are some examples of operations not covered by the monitoring rules. By investigating the fault-free traces, we observed that these operations do not involve a sequence of events, but only a single head event. However, to verify a monitoring rule of type ORD or OCC, we need a sequence of at least two events, i.e., at least one event has to follow the head event activating the rule since the users' behavior is not deterministic and, thus, can not be predicted (e.g., we can not assume that all the users will delete a volume). In addition to the rules shown in Table 8.1, we also included rules based on REST API returning status code $4xx$ (client error) and $5xx$ (server error) in our set of monitoring rules.

8.3.3 Evaluation

To evaluate the monitoring rules, we performed a fault-injection campaign in OpenStack. We exercised the system with the multi-tenant workload and injected faults in Nova, Cinder, and Neutron subsystems. In particular, we used the the *ProFIPy* tool (see § 3) to inject the following fault-types:

- **Throw exception**: An exception is raised on a method call, according to a pre-defined, per-API list of exceptions.
- **Wrong return value**: A method returns an incorrect value. The wrong return value is obtained by corrupting the targeted object, depending on the data type (e.g., by replacing an object reference with a null reference, or by replacing an integer value with a negative one).
- **Wrong parameter value**: A method is called with an incorrect input parameter. Input parameters are corrupted according to the data type, as for the previous point.

Before every experiment, we clean up any potential residual effect from the previous experiment, to be able to relate failure to the specific fault that caused it. We re-deploy the cloud management system, remove all temporary files and processes, and restore the OpenStack database to its initial state. In total, we performed 637 fault injection experiments, and we observed failures in 496 experiments ($\sim 78\%$). We consider an experiment as failed if at least one API call returns an API error or if there is at least one assertion check failure. We remark that assertion checks serve as *ground truth* about the occurrence of failures during the experiments since they are valuable to point out the cases in which a fault causes an error, but the system does not generate an API error (i.e., the system is unaware of the failure state) [58].

We evaluated the monitoring rules in terms of *precision* and *recall*. The former is defined as the ratio between the true positives and the sum of true positives and false positives. The latter is computed as the number of true positives divided by the sum of true positives and false negatives. In our context, we define a true positive case when the monitoring rules identify the failure caused by the fault injected into the system. A false negative happens when the monitoring rules do not identify the failure experienced by the system due to the injected fault. The false positives refer to the cases in which the monitoring rules identify a failure but the system is not actually failed. To provide a more complete evaluation, we aggregate precision and recall, using the F_1 score, defined as the harmonic mean of the two metrics. All the metrics range between 0 (total misclassification) and 1 (perfect classification).

To provide context for the evaluation, we compare three approaches:

- **OpenStack Coverage Mechanisms** (OCM): The failure coverage mechanisms of the OpenStack cloud infrastructure used to notify the users when the system is not able to serve the requests;
- **Monitoring Rules** (MR): The proposed approach, which applies the monitoring rules to identify the failures;
- **OpenStack with Monitoring Rules** (OCM with MR): The combination of the OpenStack coverage mechanisms with the monitoring rules is useful to estimate the improvement obtained by implementing an external monitoring solution to support OpenStack.

For each OpenStack service targeted by the fault-injection experiments, Table 8.2 shows the results, in terms of precision, recall, and F_1 score obtained by the three approaches.

The table highlights that the OCM approach provides perfect precision over all the services, i.e., the system notification of a failure is always a

Table 8.2. Approaches comparison.

OpenStack Service	**Approach**	**Precision**	**Recall**	**F_1 score**
Nova	*OCM*	1.00	0.30	0.46
	MR	0.89	1.00	0.94
	OCM with MR	0.89	1.00	0.94
Cinder	*OCM*	1.00	0.28	0.44
	MR	0.85	0.84	0.85
	OCM with MR	0.85	0.85	0.85
Neutron	*OCM*	1.00	0.71	0.83
	MR	0.87	0.31	0.46
	OCM with MR	0.95	0.92	0.93
All subsystems	*OCM*	1.00	0.36	0.53
	MR	0.87	0.82	0.85
	OCM with MR	0.88	0.93	0.91

true positive case. However, the accuracy dramatically decreases when we evaluate the false negatives since the OCM approach systematically provides the worst performances over all the services, as a consequence of the inability of the system in promptly identifying the failures, as already discussed in Chapter 4. Different from the system's coverage mechanisms, the MR approach provides some false positive cases. Nevertheless, the precision achieved by the approach is still very close to the one provided by the OCM approach. The considerations on the false-negative cases, instead, are way different. We can notice, indeed, how the MR rules can effectively bring a substantive improvement on the recall values for Nova and Cinder subsystems, while it provides worse performance for Neutron. Overall the subsystems, the MR approach increases the recall compared to the system's fault-tolerance mechanisms.

The F_1 score allows us to compare the approaches both in terms of false positives and false negatives, and thus provides a comprehensive evaluation of the approaches. The metric suggests that overall the fault-injection experiments, the MR approach massively improves the performances obtained with the plain OpenStack coverage mechanisms (84% vs 53%). In particular, the proposed approach achieves a F_1 score higher for Nova and Cinder services, while the performances are worse for the Neutron service. We attribute this to the non-determinism affecting the network service that causes either a missing coverage or a missing activation of the monitoring rules.

When the monitoring rules are used in combination with the OpenStack coverage mechanisms (OCM with MR approach), we can notice that, although the rules slightly impact the precision of the system, they massively help in the reduction of the false-negative cases, overall the services. Even for the Neutron service, when the recall for the MR approach is lower than the one provided by OpenStack coverage mechanisms, the OCM with MR approach takes benefits from the monitoring rules since they help identify failures not detected by the plain mechanisms of the system.

To provide a more comprehensive evaluation, we also analyzed the promptness of the monitoring rules in the identification of failures. Ideally, a failure should be identified as soon as the system experiences it to quickly restore the services and thus increase the reliability of the system. For every fault-injection experiment, we computed t_{fail} as the time of identification of the failure, which is the timestamp of the first API error raised by the system to notify the failure for the OCM approach, and the timestamp of the activation of a monitoring rule that identified a failure for the MR approach. Moreover, we defined t_{start} and t_{end} as the timestamp of the start and the end of the workload execution, respectively. To perform the comparison between OpenStack coverage mechanisms and the monitoring rules in terms of promptness, we defined a *failure notification interval*, i.e., the difference between the time of identification of the failure and a common initial time. We used the start of workload execution as the initial time, i.e., the failure notification interval is equal to $t_{fail} - t_{start}$.

Since the approaches are compared on the same experiments and in the same conditions, a lower failure notification time indicates the ability to promptly identify failures.

To perform a fair and robust evaluation, we did not consider the notification of the false-positive cases in this analysis since they would unfairly help the monitoring rules. Moreover, we considered the failure notification interval of undetected failures as the whole duration of the workload (i.e., $t_{end} - t_{start}$) otherwise they would not be considered at all in this analysis. Table 8.3 shows the average failure notification interval (in seconds) provided by the approaches.

Table 8.3. Average Failure notification interval (seconds).

Approach	**Nova**	**Cinder**	**Neutron**	**All subsystems**
OCM	711.02	451.58	401.37	553.86
MR	507.88	371.25	404.32	439.80
OCM with MR	507.26	368.68	328.85	424.06

The table shows that the MR approach provides a notably lower failure notification time when compared to the OCM approach for Nova and Cinder services, and a comparable notification time for the Neutron service. Overall the fault-injection experiments, the average notification time of the monitoring rules is $\sim$ 114 seconds lower than the average notification time of the OpenStack coverage mechanisms. The failure notification time of the OCM with the MR approach is very close to the MR approach and thus proves that the contribution of the monitoring rules is crucial for the prompt detection of failures at runtime. Also for the Neutron service, where the MR approach showed the worst performance due to the asynchronous nature of the network operations, the OCM with MR approach notably decreases the average failure notification time with respect to the OCM approach ($\sim$ 77 seconds).

8.3.4 Sensitivity Analysis

We performed a sensitivity analysis of the monitoring rules by varying the length of the time window used by the algorithm to identify the patterns.

Table 8.4 shows the results of the analysis by setting the time window equal to 5, 20, 35, and 50 seconds. Unsurprisingly, we found that the performance of the approach improves by increasing the length of the time window. Indeed, a short time window increases the number of false positives and limits the true negatives. As matter of fact, we found that a time window equal to 5 seconds provides a false positive rate equal to 1. On the other hand, a larger time window allows the algorithm to find more patterns more robust to the non-deterministic variations of the events, increasing both the precision and the recall.

The analysis shows also that there is a saturation of the performance of the monitoring rules. Indeed, from 35 to 50 seconds, we have an increase of the F_1 score of 1%. Since the goal is to detect failures as soon as possible, the choice of the time window should be a valid compromise between performance and detection time and, therefore, a time window equal to 35 seconds (or also 20 seconds) can be considered a more valuable choice.

Table 8.4. Sensitivity Analysis of the time window.

Time Window (seconds)	**Precision**	**Recall**	F_1 **score**
5	0.75	0.87	0.81
20	0.85	0.92	0.88
35	0.88	0.92	0.90
50	0.90	0.92	0.91

8.3.5 Computational Cost

We performed the analysis of the computational cost needed to derive the monitoring rules from fault-free traces. The computational times include the time needed to parse the logs, filter events, and adopt the algorithm to find the patterns. We found that the overall time needed to simultaneously analyze 50 different fault-free execution traces (which contain $\sim$ 140000 rows) is lower than 70 seconds (i.e., less than 1.5 seconds per trace, on average). The computational cost increases linearly with the number of traces.

8.4 Runtime Monitor

After identifying the monitoring rules, we synthesize the rules in a runtime monitor that verifies whether the system's behavior follows the desired one. Any runtime violation of the monitoring rules gives a timely notification to avoid undesired consequences, e.g., non-logged failures, non-fail-stop behavior, failure propagation across sub-systems, etc.

We translate the rules in the *Event Processing Language* (EPL), a particular specification language provided by the Esper software (see Appendix C), and allow the expression of different types of rules (i.e., temporal, statistical, etc.). The language is a SQL-standard language with extensions, offering both typical SQL clauses (e.g., `select`, `from`, `where`, `insert into`) and additional clauses for event processing (e.g, `pattern`, `output`).

We applied the EPL statements, derived from the monitoring rules, to detect failures in OpenStack when multiple users perform requests concurrently (multi-tenant workload). Since we do not collect a user ID, we use a *counter* to take into account multi-tenancy operations. We associate a different counter to each head event: when a head event occurs, we increment its counter. This allows us to keep track of the same actions performed simultaneously by different users and, consequently, to activate multiple monitoring rules, one per each different action. For example, suppose that, in the same time window, we observe twice the event type `<conductor, schedule_and_build_instances>`, which is the head event of the request flow related to the instance creation. We derive that two users are concurrently requesting (in the same time window) the creation of an instance. Therefore, we activate two monitoring rules to fully control the users' actions. The value of the counter is sent, along with the event name, to the Esper Runtime.

To express the monitoring rules, we use the clause of `pattern`, useful for finding time relationships between events. Pattern expressions usually consist of filter expressions combined with pattern operators. We use the pattern operators `every`, `followed-by` ($\rightarrow$), and `timer:interval`. The operator `every` defines that every time a pattern subexpression connected to this operator turns true, the Esper Runtime starts a new active subexpression. Without this operator, the subexpression stops after the first time it becomes true. The operator $\rightarrow$ operates on events order, establishing that the right-hand expression is evaluated only after that the left-hand expression turns true. The operator `timer:interval` establishes the duration of the time window during which to observe the arriving events (it starts after that the left-hand expression turns true).

Listing 8.1 shows the EPL translation of the rule *Volume Creation* in the multi-user case.

Listing 8.1. EPL rule in the multi-user scenario

```
@name('Rule#1') select * from pattern [every a = Event(name="
     cinder-scheduler_create_volume") -> (timer:interval(
     secondsToWait seconds) and not b=Event(name="cinder-
     volume_create_volume", countEvent = a.countEvent))];
```

Every time the Esper Runtime observes an event `<cinder, scheduler_create_ volume>` with its counter value, it waits for the observation of the event `<cinder, volume_create_volume>` with the same counter value within a time window of `secondsToWait` seconds. If this condition is not verified, the approach generates a failure detection message.

The monitor synthesis is automatically performed once EPL rules are compiled. The Esper Runtime acts like a container for EPL statements which continuously executes the queries (expressed by the statements) against the data arriving as inputs. For more detailed information on Esper, we refer the reader to the official documentation [79].

The apply the runtime monitor in the context of OpenStack, we trace all messages exchanged in the system by using Zipkin, which stores all information in an online collector (see Appendix B). We then extract periodically the events stored in the Zipkin collector and extract information such as the invoked method, the service providing the API, the timestamp, and the status code of the REST API. The processed information is then pushed into a queue, named *Esper Inputs Waiting Queue*, which stores the flow of events. The events in the queue are sent as inputs to the Esper Runtime, which compares the flow of events to every statement compiled by the Esper Compiler (i.e., the monitoring rules): if that event satisfies the condition of a rule, then the rule moves to the next condition, otherwise, it raises an exception.

Conclusion

Nowadays, cloud computing systems are extensively used to run services in different domains around the world. However, it is very difficult to avoid software bugs when implementing the rich set of services of cloud computing systems. As a result, many high-severity failures have been occurring in the cloud infrastructures of popular providers, causing outages of several hours and the unrecoverable loss of user data. Therefore, the high-reliability requirements of such systems are still too far to reach. Fault injection represents a valid solution to assess the fault-tolerance mechanisms and improve the overall reliability, but its adoption in cloud systems still faces important issues.

This thesis dissertation addressed these open issues by proposing effective solutions to apply fault-injection in cloud systems and to better understand the failure nature of these systems and design time monitoring strategy, which is capable of improving the failure detection capabilities.

In Chapter 3, we introduced *ProFIPy*, a tool designed to be programmable and highly usable, by performing fault injection campaigns with customized fault loads in Python software. The programmability of the tool through a DSL was useful to easily and quickly customize fault injections to comply with the fault classes requested by the company, based on their internal software requirements.

In Chapter 4, we used the tool proposed in Chapter 3 to empirically assess the severity of failures caused by software bugs, through the deliberate injection of software bugs. We applied this methodology in the context of the OpenStack cloud management system. The experiments pointed out that the behavior of OpenStack under failure is not amenable to automated detection and recovery. In particular, the system often exhibits a *non-fail-stop* behavior, in which it continues to execute despite inconsistencies in the state of the virtual resources, without notifying the user about the failure, and without producing logs for aiding system operators. Moreover, we found that the failures can spread across several sub-systems before being notified and that they can cause persistent effects that are difficult to recover. Finally, we point out areas for future research to mitigate these issues, including run-time verification techniques to detect subtle failures in a more timely fashion and to prevent persistent corruption.

In Chapter 5, we proposed a novel anomaly detection approach to identify the failure symptoms and enhance the error propagation analysis. The approach analyzes the execution traces of distributed systems under fault injection, by comparing the executions to fault-free ones to point out anomalies. To address the problem of non-determinism (which may lead to "benign" anomalies not actually related to failures), we develop a sequence comparison approach supported by a probabilistic model. The probabilistic model is built from a group of several fault-free execution traces, in order to reflect "benign" variations that normally occur in the distributed system. Moreover, to make the approach applicable to black-box systems and not reliant on intrusive instrumentation, we base our probabilistic model only on externally observable traces of messages, which are analyzed as sequences of symbols using Variable-order Markov Models. We evaluated the approach within the OpenStack cloud computing platform: we found that the VMM limits the false positives compared to a nonprobabilistic comparison of execution sequences, without significant loss in terms of false negatives. Moreover, the VMM is lightweight enough to be applicable with a low computational cost.

In Chapter 6, we presented a novel approach for discovering failure modes in distributed systems, by combining fault injection, distributed tracing, and unsupervised learning algorithms. By adopting a probabilistic model (VMM), our approach can identify anomalies in noisy execution traces by significantly reducing the false alarms without discarding true anomalies. To further help the human analyst in analyzing failures, we presented a novel technique that clusters fault injection experiments according to classes of failure modes. The results showed that clustering can achieve high accuracy under different conditions.

In Chapter 7, we presented a novel approach for analyzing failure data from cloud systems, by using unsupervised learning algorithms and deep learning to cluster the failure data into failure classes. The proposed approach relieves the human analyst from manually tuning the features to achieve a good performance at clustering failure data. The approach leverages an autoencoder for dimensionality reduction and parameter initialization, in combination with a clustering layer to optimize both the reconstruction error and inter-cluster distance. The results show that the proposed approach can achieve performance comparable to, or in some cases even better than, the performance of manually-tuned clustering, which entails a deep knowledge of the domain and a significant human effort. In all cases, the proposed approach performs better than unsupervised clustering when no feature engineering is made to the dataset. The approach has been designed to be applied without any a priori information about the types of features in the failure data, in order to minimize the manual effort. This is especially important when the cloud system is still under active development when multiple versions are updated, tested, and released at a quick pace. However, our approach cannot exceed the accuracy that can be achieved by leveraging the knowledge of the human analyst about the system. Furthermore, since the approach uses deep neural networks, it requires high hardware requirements to keep computational times acceptable, in particular when the amount of data to analyze is very large.

In Chapter 8, we proposed an approach to runtime monitoring in cloud computing infrastructures that automatically infers a set of monitoring rules from the fault-free executions of the system. We used the approach in the OpenStack cloud computing platform, showing the feasibility of the approach in a large and complex "off-the-shelf" distributed system. The approach derived different rule types, which have been evaluated in the context of fault-injection experiments by using a multi-tenant workload. The analysis proved that the monitoring rules improve the fault tolerance mechanisms of the system, providing high accuracy in the identification of failures at runtime and decreasing the time to notify the failure. We also implemented a prototype of a monitoring solution by using a specification language to analyze the overhead introduced by the approach in a real user scenario and showed that the approach can be applied in practice with low effort and computational costs.

The limitations of this dissertation are represented by the threats to external validity, i.e., the possibility to generalize the application of the solutions and the results. Indeed, although all the solutions have been extensively validated in OpenStack, which is not a trivial cloud platform, it still represents a single usage scenario. The development of the tools and the execution of the experiments also on other cloud computing platforms is nearly prohibitive but, at the same time, we have to consider that cloud systems are quickly moving towards container-based platforms (e.g., Kubernetes, Openshift, etc.), and the results obtained with heavy-weight virtual machine-based platforms might not hold for container-based platforms, which sometimes include self-healing mechanisms.

To mitigate these limitations, the dissertation focused on the three major OpenStack projects (i.e., Nova, Neutron, and Cinder), which are large and diverse enough to get interesting insights on the application of the proposed solutions across different projects and different languages (e.g., Python versus C and Java). The diversity of the projects was reflected by differences in terms of project-specific patterns, due to the programming idioms, API conventions, and process of the projects, and in terms of the different messages exchanged in the services, i.e., the number, type, and sources of non-determinism.

A further valuable aspect to take into consideration is that, although cloud systems are very heterogeneous, the communication protocols targeted in this dissertation and used to collect the events are independent of OpenStack. Indeed, RPC is widely used in client-server computing and is readily used to take advantage of cloud resources [15], and many cloud providers use the REST architectural style for offering such resources [196]. The tracing system used in this thesis is widely adopted in different real-world usage scenarios. Just to pinpoint some relevant examples, Zipkin is used at Salesforce to perform distributed tracing for microservices [218] or to gather timing data for all the disparate services involved in managing a request with the Twitter API [241].

Since the anomaly detection approach, the failure mode analysis, and the runtime monitoring solution depend on the observation of the events exchanged in the system, we expect that the application of all these solutions on a different cloud platform is feasible because of the maturity of the distributed tracing system and the same communication protocols of the cloud systems, although we acknowledge it may lead to different results.

At the end of the day, the solutions proposed, discussed, and validated in this thesis contribute extensively and significantly to the topic of reliability in cloud computing and can be considered of interest for both the academic and industrial communities.

This page intentionally left blank.

Appendix A

Introduction to OpenStack

OpenStack is a popular cloud computing platform widespread among public cloud providers and private users [187], and the basis of over many commercial products [180, 187] OpenStack contains a large set of components, each providing APIs to manage virtual resources, and consists of $\sim$ 20 million LoC [184]. OpenStack embraces a modular architecture to provide a set of core services that facilitates scalability and elasticity as core design tenets, as shown in Figure A.1. This chapter briefly reviews OpenStack components, their use cases, and security considerations.

A.1 Compute

OpenStack Compute service (nova) provides services to support the management of virtual machine instances at scale, instances that host multi-tiered applications, dev or test environments, “Big Data” crunching Hadoop clusters, or high-performance computing. The Compute service facilitates this management through an abstraction layer that interfaces with supported hypervisors (we address this later on in more detail). Compute security is critical for an OpenStack deployment. Hardening techniques

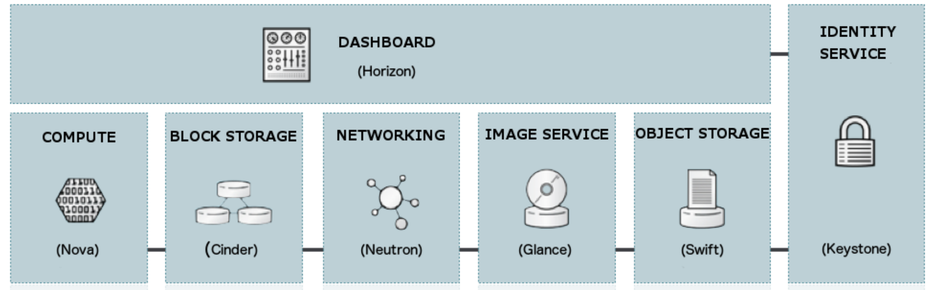


Figure A.1. OpenStack service overview.

should include support for strong instance isolation, secure communication between Compute sub-components, and resiliency of public-facing API endpoints.

A.2 Object Storage

The OpenStack Object Storage service (swift) provides support for storing and retrieving arbitrary data in the cloud. The Object Storage service provides both a native API and an Amazon Web Services S3-compatible API. The service provides a high degree of resiliency through data replication and can handle petabytes of data. It is important to understand that object storage differs from traditional file system storage. Object storage is best used for static data such as media files (MP3s, images, or videos), virtual machine images, and backup files. Object security should focus on access control and encryption of data in transit and at rest. Other concerns might relate to system abuse, illegal or malicious content storage, and cross-authentication attack vectors.

A.3 Block Storage

The OpenStack Block Storage service (cinder) provides persistent block storage for compute instances. The Block Storage service is responsible for managing the life-cycle of block devices, from the creation and attachment of volumes to instances, to their release. Security considerations for block storage are similar to that of object storage.

A.4 Shared File Systems

The Shared File Systems service (manila) provides a set of services for managing shared file systems in a multi-tenant cloud environment, similar to how OpenStack provides for block-based storage management through the OpenStack Block Storage service project. With the Shared File Systems service, you can create a remote file system, mount the file system on your instances, and then read and write data from your instances to and from your file system.

A.5 Networking

The OpenStack Networking service (neutron, previously called quantum) provides various networking services to cloud users (tenants) such as IP address management, DNS, DHCP, load balancing, and security groups (network access rules, like firewall policies). This service provides a framework for software-defined networking (SDN) that allows for pluggable integration with various networking solutions. OpenStack Networking allows cloud tenants to manage their guest network configurations. Security concerns with the networking service include network traffic isolation, availability, integrity, and confidentiality.

A.6 Dashboard

The OpenStack Dashboard (horizon) provides a web-based interface for both cloud administrators and cloud tenants. Using this interface, administrators and tenants can provision, manage, and monitor cloud resources. The dashboard is commonly deployed in a public-facing manner with all the usual security concerns of public web portals.

A.7 Identity Service

The OpenStack Identity service (keystone) is a shared service that provides authentication and authorization services throughout the entire cloud infrastructure. The Identity service has pluggable support for multiple forms of authentication. Security concerns with the Identity service include trust in authentication, the management of authorization tokens, and secure communication.

A.8 Image Service

The OpenStack Image service (glance) provides disk-image management services, including image discovery, registration, and delivery services to the Compute service, as needed. Trusted processes for managing the life cycle of disk images are required, as are all the previously mentioned issues with respect to data security.

A.9 Data Processing Service

The Data Processing service (sahara) provides a platform for the provisioning, management, and usage of clusters running popular processing frameworks. Security considerations for data processing should focus on data privacy and secure communications to provisioned clusters.

A.10 Other Supporting Technology

Messaging is used for internal communication between several OpenStack services. By default, OpenStack uses message queues based on the AMQP. Like most OpenStack services, AMQP supports pluggable components. Today the implementation back end could be RabbitMQ, Qpid, or ZeroMQ. Because most management commands flow through the message queuing system, message-queue security is a primary security concern for any OpenStack deployment. Several of the components use databases though it is not explicitly called out. Securing database access is yet another security concern.

This page intentionally left blank.

Appendix B

Zipkin

Zipkin is a distributed tracing system [274]. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data.

Applications need to be "instrumented" to report trace data to Zipkin. This usually means the configuration of a tracer or instrumentation library. The most popular ways to report data to Zipkin are via HTTP or Kafka, though many other options exist, such as Apache ActiveMQ, gRPC, and RabbitMQ. The data served to the UI are stored in memory, or persistently with a supported backend such as Apache Cassandra or Elasticsearch.

Tracers live in the applications and record timing and metadata about operations that took place. They often instrument libraries, so that their use is transparent to users. For example, an instrumented web server records when it received a request and when it sent a response. The trace data collected is called a *Span*. Tracing information is collected on each host using the instrumented libraries and sent to Zipkin. When the host requests another application, it passes a few tracing identifiers along with the request to Zipkin in order to tie the data together into spans.

Instrumentation is written to be safe in production and has little overhead. For this reason, they only propagate IDs in-band, to tell the receiver there's a trace in progress. Completed spans are reported to Zipkin out-of-band, similar to how applications report metrics asynchronously. For example, when an operation is being traced and it needs to make an outgoing HTTP request, a few headers are added to propagate IDs. Headers are not used to send details such as the operation name.

The component in an instrumented app that sends data to Zipkin is called a Reporter. Reporters send trace data via one of several transports to Zipkin collectors, which persist trace data to storage. Later, storage is queried by the API to provide data to the UI. Figure B.1 shows a diagram describing this flow.

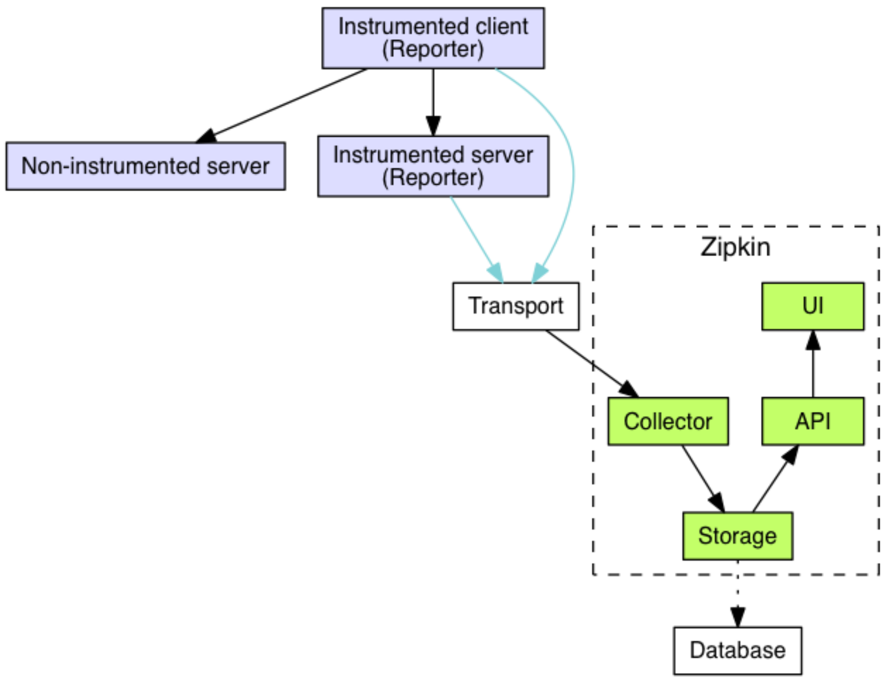


Figure B.1. Zipkin architecture.

Identifiers are sent in-band and details are sent out-of-band to Zipkin. In both cases, trace instrumentation is responsible for creating valid traces and rendering them properly. For example, a tracer ensures parity between the data it sends in-band (downstream) and out-of-band (async to Zipkin).

Trace instrumentation report spans asynchronously to prevent delays or failures relating to the tracing system from delaying or breaking user code. Spans sent by the instrumented library must be transported from the services being traced to Zipkin collectors. There are three primary transports: HTTP, Kafka, and Scribe.

There are 4 components that makeup Zipkin, briefly described in the following.

■ **Zipkin Collector**. Once the trace data arrives at the Zipkin collector daemon, it is validated, stored, and indexed for lookups by the Zipkin collector.

■ **Storage.** Zipkin was initially built to store data on Cassandra since Cassandra is scalable, has a flexible schema, and is heavily used within Twitter. However, we made this component pluggable. In addition to Cassandra, we natively support ElasticSearch and MySQL. Other backends might be offered as third-party extensions.

■ **Zipkin Query Service**. Once the data is stored and indexed, we need a way to extract it. The query daemon provides a simple JSON API for finding and retrieving traces. The primary consumer of this API is the Web UI.

■ **Web UI.** We created a GUI that presents a nice interface for viewing traces. The web UI provides a method for viewing traces based on service, time, and annotations. Note: there is no built-in authentication in the UI!

This page intentionally left blank.

Appendix C

Esper

Esper is a language, compiler and runtime for complex event processing (CEP) and streaming analytics [80]. It enables rapid development of applications that process large volumes of incoming messages or events, regardless of whether incoming messages are historical or real-time in nature. Esper filters and analyzes events in various ways, and respond to conditions of interest.

Esper offers a language by name Event Processing Language (EPL) that implements and extends the SQL-standard and enables rich expressions over events and time. The Esper compiler compiles EPL into byte code that can be saved in jar package file format for distribution and execution. The Esper runtime loads and executes byte code produced by the Esper compiler. The runtime provides a highly scalable, memory-efficient, in-memory computing, minimal latency, real-time streaming-capable processing engine for online and real-time arriving data and high-variety data, as well as for historical event analysis.

The compiler and runtime are not limited to running on a single machine and run well inside a distributed stream processing framework. The compiler and runtime make sense and can run in any architecture and any container, as they have no dependencies on external services and do not require any particular threading model or model of how time advances and do not require any external storage. EPL works well with event-time and watermark-based time management.

The Esper runtime has an horizontal scale-out architecture for linear horizontal scalability, elastic scaling, load distribution, balancing and re-balancing, fault tolerance, dynamic discovery of nodes through seed nodes, replication and multi-datacenter support. The design priorities for Esper are: *i)* low latency and high throughput; *ii)* expressiveness, conciseness, extensibility of the EPL language; *iii)* compliance to standards and best practices; *iv)* light-weight in terms of memory, CPU and IO usage.

C.1 Event Processing Language

Event Processing Language is designed for Complex Event Processing and Streaming Analytics. It is organized in modules that are compiled into bytecode by the compiler. A module is an EPL source code unit and it is composed of a set of statements. Optionally, a module can have a name that is used in a similar way to a package name in a programming language. The statements are continuous queries that analyze events and time: they can be used to detect situations. Moreover, they can have listeners attached to them so that predefined actions can be triggered every time an event that matches the condition of the statement is met. A statement has always a name that is used to identify it within a deployment.

A statement can declare different EPL-objects (Event types, Variables, Named windows, Tables, Contexts, Expressions, and Scripts, Indexes) and use different access modifiers (private, public, and protected) to control the access to them. The Esper runtime can be seen as a statement container. An actor (i.e., a user) can interact with Esper by compiling and deploying modules containing statements, as shown in Figure C.1.

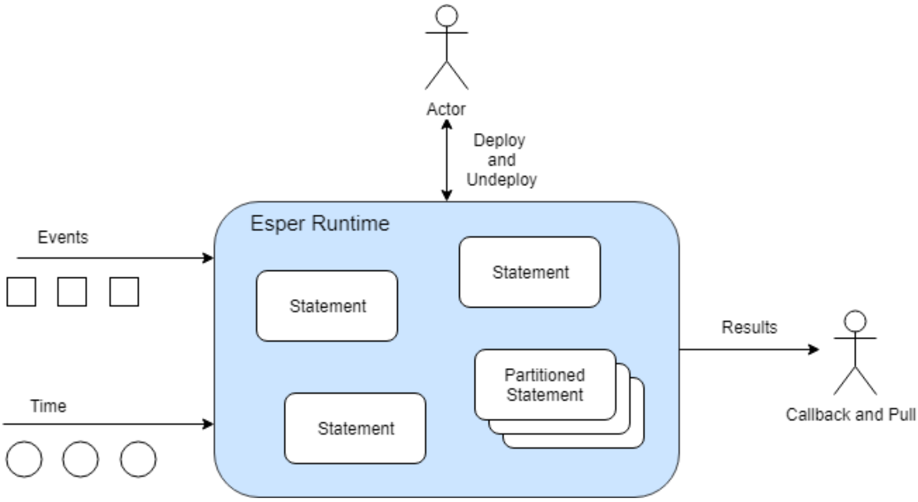


Figure C.1. Esper Runtime

A basic select allows selecting all the arriving events of interest, as follows.

Listing C.1. Basic select

```
select * from MyEvent
```

Upon the event of a new MyEvent event arriving, the runtime passes the arriving event as it is to callbacks. After that, the runtime effectively forgets the current event.

An aggregation function groups multiple events together to form a single value. The following example counts the number of MyEvent events arriving and passes the new count to callbacks. After that, the runtime forgets the current event but remembers the current count.

Listing C.2. Basic Aggregation

```
select count(*) from MyEvent
```

A filter can be used to consider only a subset of the events MyEvent that arrives (for example, only those events with the attribute temperature

> 35).

Listing C.3. Event Filter

```
select * from MyEvent(temperature > 35)
```

A data window retains events for aggregation, match-recognize patterns, subqueries, etc. It can be defined as a length window (keeps the last N events) or a time window (keeps the last N seconds of events). Upon the arrival of a new event, the runtime adds that event to the window but also passes the same event to callbacks.

Listing C.4. Basic Data Window

```
select * from MyEvent#length(10)
```

This concept can be combined to obtain more expressive statements. A basic EPL pattern matches when an event or multiple events occur that match the definition of the pattern. A pattern can have five types of operators: every, logical operators (and, or, not), the followed-by operator, guards that cause termination of pattern subexpression (as timer:within), observers that observe time events (timer:interval, timer:at).

Listing C.5. EPL Pattern

```
every a = A -> b=B(attribute1 = a.attribute1)
```

The operator followed-by ($\rightarrow$) is used to express a temporal relationship between events. The operator every is used to clarify that not only the first event of a certain type has to be considered, but every one of them. A more complete description of the EPL language can be found by consulting the documentation [79].

Bibliography

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 83–93, 2015.

[2] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review*, 37(5):74–89, 2003.

[3] Michal Aharon, Gilad Barash, Ira Cohen, and Eli Mordechai. One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 227–243. Springer, 2009.

[4] Maryam Raiyat Aliabadi and Karthik Pattabiraman. Fidl: A fault injection description language for compiler-based sfi tools. In *International Conference on Computer Safety, Reliability, and Security*, pages 12–23. Springer, 2016.

[5] Javier Alonso, Lluis Belanche, and Dimiter R Avresky. Predicting software anomalies using machine learning techniques. In *2011 IEEE 10th international symposium on network computing and applications*, pages 163–170. IEEE, 2011.

[6] Hany H Ammar, Bojan Cukic, Ali Mili, and Cris Fuhrman. A comparative analysis of hardware and software fault tolerance: Impact on software reliability engineering. *Annals of Software Engineering*, 10(1):103–150, 2000.

[7] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411, 2005.

[8] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, J-C Fabre, J-C Laprie, Eliane Martins, and David Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on software engineering*, 16(2):166–182, 1990.

[9] Jean Arlat, Alain Costes, Yves Crouzet, Jean-Claude Laprie, and David Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on computers*, 42(8):913–923, 1993.

[10] Jean Arlat, J-C Fabre, and Manuel Rodríguez. Dependability of cots microkernel-based systems. *IEEE Transactions on computers*, 51(2):138–163, 2002.

[11] Jean Arlat and Regina Moraes. Collecting, analyzing and archiving results from fault injection experiments. In *2011 5th Latin-American Symposium on Dependable Computing*, pages 100–105. IEEE, 2011.

[12] Preeti Arora, Shipra Varshney, et al. Analysis of k-means and k-medoids algorithm for big data. *Procedia Computer Science*, 78:507–512, 2016.

[13] Srikesh G Arunajadai, Scott J Uder, Robert B Stone, and Irem Y Tumer. Failure mode identification through clustering analysis. *Quality and Reliability Engineering International*, 20(5):511–526, 2004.

[14] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[15] Paramvir Bahl, Richard Y Han, Li Erran Li, and Mahadev Satyanarayanan. Advancing the state of mobile cloud computing. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*, pages 21–28, 2012.

[16] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *HotOS*, pages 85–90, 2003.

[17] Ezio Bartocci and Yliès Falcone. *Lectures on Runtime Verification: Introductory and Advanced Topics*, volume 10457. Springer, 2018.

[18] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In *Lectures on Runtime Verification*, pages 1–33. Springer, 2018.

[19] Claudio Basile, Meeta Gupta, Zbigniew Kalbarczyk, and Ravi K Iyer. An approach for detecting and distinguishing errors versus attacks in sensor networks. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 473–484. IEEE, 2006.

[20] Ali Basiri, Niosha Behnam, Ruud De Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[21] Tania Basso, Regina Moraes, Bruno P Sanches, and Mario Jino. An investigation of java faults operators derived from a field data study on java software faults. In *Workshop de Testes e Tolerância a Falhas*, pages 1–13, 2009.

[22] Luana Batista, Eric Granger, and Robert Sabourin. Improving performance of hmm-based off-line signature verification systems through a multi-hypothesis approach. *International Journal on Document Analysis and Recognition (IJDAR)*, 13(1):33–47, 2010.

[23] Eric Bauer and Randee Adams. *Reliability and availability of cloud computing.* John Wiley & Sons, 2012.

[24] Ron Begleiter, Ran El-Yaniv, and Golan Yona. On prediction using variable order markov models. *Journal of Artificial Intelligence Research*, 22:385–421, 2004.

[25] Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In *Proceedings Seventh International Sympo-*

sium on String Processing and Information Retrieval. SPIRE 2000, pages 39–48. IEEE, 2000.

[26] Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D Ernst. Visualizing distributed system executions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(2):1–38, 2020.

[27] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D Ernst. Debugging distributed systems. *Communications of the ACM*, 59(8):32–37, 2016.

[28] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[29] Domenico Bianculli, Carlo Ghezzi, Cesare Pautasso, and Patrick Senti. Specification patterns from research to industry: a case study in service-based applications. In *Proc. ICSE*, pages 968–976. IEEE, 2012.

[30] Black Duck Software, Inc. The OpenStack open source project on Open Hub. https://www.openhub.net/p/openstack.

[31] Stefan Blom, Jaco van de Pol, and Michael Weber. Ltsmin: Distributed and symbolic reachability. In *Proc. CAV*, pages 354–359. Springer, 2010.

[32] Nadia Bolshakova and Francisco Azuaje. Cluster validation techniques for genome expression data. *Signal processing*, 83(4):825–833, 2003.

[33] Andrea Bondavalli, Andrea Ceccarelli, Lorenzo Falai, and Michele Vadursi. Foundations of measurement theory applied to the evaluation of dependability attributes. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 522–533. IEEE, 2007.

[34] Andrea Bondavalli, Andrea Ceccarelli, Lorenzo Falai, and Michele Vadursi. A new approach and a related tool for dependability measurements on distributed systems. *IEEE Transactions on Instrumentation and Measurement*, 59(4):820–831, 2010.

[35] George Candea and Armando Fox. Crash-only software. In *HotOS*, volume 3, pages 67–72, 2003.

[36] Laura Carnevali, Francesco Santoni, and Enrico Vicario. Learning marked markov modulated poisson processes for online predictive analysis of attack scenarios. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 195–205. IEEE, 2019.

[37] Gabriella Carrozza, Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. Analysis and prediction of mandelbugs in an industrial software system. In *2013 IEEE Sixth international conference on software testing, verification and validation*, pages 262–271. IEEE, 2013.

[38] Frederico Cerveira, Raul Barbosa, Henrique Madeira, and Filipe Araujo. Recovery for Virtualized Environments. In *Proc. EDCC*, pages 25–36, 2015.

[39] Ramesh Chandra, Ryan M Lefever, Kaustubh R Joshi, Michel Cukier, and William H Sanders. A global-state-triggered fault injector for distributed system evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):593–605, 2004.

[40] Wui Lee Chang, Kai Meng Tay, and Chee Peng Lim. Clustering and visualization of failure modes using an evolving tree. *Expert Systems with Applications*, 42(20):7235–7244, 2015.

[41] Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 569–588, 2007.

[42] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*, pages 595–604. IEEE, 2002.

[43] Yen-Yang Michael Chen. *Path-based failure and evolution management.* University of California, Berkeley, 2004.

[44] Ram Chillarege, Inderpal S Bhandari, Jarir K Chaar, Michael J Halliday, Diane S Moebus, Bonnie K Ray, and Man-Yuen Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on software Engineering*, 18(11):943–956, 1992.

[45] Ram Chillarege and Nicholas S Bowen. Understanding large system failures-a fault injection experiment. In *1989 The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 356–357. IEEE Computer Society, 1989.

[46] Ram Chillarege, Wei-Lun Kao, and Richard G Condit. Defect type and its impact on the growth curve. In *ICSE*, volume 91, pages 246–255, 1991.

[47] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 217–231, 2014.

[48] Jörgen Christmansson and Ram Chillarege. Generation of an error set that emulates software faults based on field data. In *Proceedings of Annual Symposium on Fault Tolerant Computing*, pages 304–313. IEEE, 1996.

[49] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proc. CAV*, pages 359–364. Springer, 2002.

[50] John Cleary and Ian Witten. Data compression using adaptive coding and partial string matching. *IEEE transactions on Communications*, 32(4):396–402, 1984.

[51] John G Cleary and William J Teahan. Unbounded length contexts for ppm. *The Computer Journal*, 40(2_and_3):67–75, 1997.

[52] Domenico Cotroneo, Luigi De Simone, Antonio Ken Iannillo, Roberto Natella, Stefano Rosiello, and Nematollah Bidokhti. Analyzing the context of bug-fixing changes in the openstack cloud computing platform. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 334–345. IEEE, 2019.

[53] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. Fault injection analytics: A novel approach to discover failure modes in cloud-computing systems. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[54] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. Profipy: Programmable software fault injection as-a-service. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 364–372. IEEE, 2020.

[55] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. Enhancing the analysis of software failures in cloud computing systems with deep learning. *Journal of Systems and Software*, 181:111043, 2021.

[56] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. Enhancing failure propagation analysis in cloud computing systems. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 139–150. IEEE, 2019.

[57] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. Failviz: A tool for visualizing fault injection experiments in distributed systems. In *2019 15th European Dependable Computing Conference (EDCC)*, pages 145–148. IEEE, 2019.

[58] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 200–211, 2019.

[59] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Angela Scibelli. Towards runtime verification via event stream processing in cloud computing infrastructures. In *International Conference on Service-Oriented Computing*, pages 162–175. Springer, 2020.

[60] Domenico Cotroneo, Anna Lanzaro, Roberto Natella, and Ricardo Barbosa. Experimental analysis of binary-level software fault injection in com-

plex software. In *2012 Ninth European Dependable Computing Conference*, pages 162–172. IEEE, 2012.

[61] Domenico Cotroneo and Roberto Natella. Fault injection for software certification. *IEEE Security & Privacy*, 11(4):38–45, 2013.

[62] Domenico Cotroneo, Roberto Pietrantuono, and Stefano Russo. Combining operational and debug testing for improving reliability. *IEEE Transactions on Reliability*, 62(2):408–423, 2013.

[63] Alessandro Daidone, Felicita Di Giandomenico, Andrea Bondavalli, and Silvano Chiaradonna. Hidden markov models as a support for diagnosis: Formalization of the problem and synthesis of the solution. In *2006 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, pages 245–256. IEEE, 2006.

[64] Ben d'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. Lola: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174. IEEE, 2005.

[65] Murial Daran and Pascale Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. *ACM SIGSOFT Software Engineering Notes*, 21(3):158–171, 1996.

[66] Nelly Delgado, Ann Q Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on software Engineering*, 30(12):859–872, 2004.

[67] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, et al. Uncovering bugs in distributed storage systems during testing (not in production!). In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 249–262, 2016.

[68] James Denton. *Learning OpenStack Networking (Neutron)*. Packt Publishing Ltd, 2015.

[69] Docker, Inc. Dockerfile reference. https://docs.docker.com/engine/reference/builder.

[70] Docker, Inc. Home page of Docker. https://www.docker.com.

[71] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[72] Charalampos Doukas and Ilias Maglogiannis. Bringing iot and cloud computing towards pervasive healthcare. In *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 922–926. IEEE, 2012.

[73] Chun-Yan Duan, Xu-Qi Chen, Hua Shi, and Hu-Chen Liu. A new model for failure mode and effects analysis based on k-means clustering within hesitant linguistic environment. *IEEE Transactions on Engineering Management*, 2019.

[74] Joao Duraes and Henrique Madeira. Emulation of software faults by educated mutations at machine-code level. In *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.*, pages 329–340. IEEE, 2002.

[75] Joao A Duraes and Henrique S Madeira. Emulation of software faults: A field data study and a practical approach. *Ieee transactions on software engineering*, 32(11):849–867, 2006.

[76] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Proc. ICSE*, pages 411–420, 1999.

[77] Jens Ehlers, André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Self-adaptive software system monitoring for performance anomaly localization. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 197–200, 2011.

[78] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for

dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[79] Espertech. Esper reference. http://esper.espertech.com/release-8.7.0/reference-esper/html_single/index.html.

[80] EsperTech. Home page of Esper. http://www.espertech.com/esper.

[81] etcd. Home page of etcd. https://etcd.io.

[82] Mostafa Farshchi, Jean-Guy Schneider, Ingo Weber, and John Grundy. Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In *2015 IEEE 26th international symposium on software reliability engineering (ISSRE)*, pages 24–34. IEEE, 2015.

[83] Mostafa Farshchi, Jean-Guy Schneider, Ingo Weber, and John Grundy. Metric selection and anomaly detection for cloud operations using log and metric correlation analysis. *Journal of Systems and Software*, 137:531–549, 2018.

[84] Vincenzo De Florio and Chris Blondia. A survey of linguistic structures for application-level fault tolerance. *ACM Computing Surveys (CSUR)*, 40(2):1–37, 2008.

[85] Min Fu, Liming Zhu, Ingo Weber, Len Bass, Anna Liu, and Xiwei Xu. Process-oriented non-intrusive recovery for sporadic operations on cloud. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 85–96. IEEE, 2016.

[86] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*, pages 149–158. IEEE, 2009.

[87] Peter Garraghan, Renyu Yang, Zhenyu Wen, Alexander Romanovsky, Jie Xu, Rajkumar Buyya, and Rajiv Ranjan. Emergent failures: Rethinking cloud reliability at scale. *IEEE Cloud Computing*, 5(5):12–21, 2018.

[88] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *NSDI*, volume 7, pages 285–298, 2007.

[89] Kamran Ghasedi Dizaji, Amirhossein Herandi, Cheng Deng, Weidong Cai, and Heng Huang. Deep clustering via joint convolutional autoencoder embedding and relative entropy minimization. In *Proceedings of the IEEE international conference on computer vision*, pages 5736–5745, 2017.

[90] Inc. GitHub. The state of the octoverse. https://octoverse.github.com.

[91] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Edfi: A dependable fault injection tool for dependability benchmarking experiments. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, pages 31–40. IEEE, 2013.

[92] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. Inferring and asserting distributed system invariants. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1149–1159, 2018.

[93] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on reliability in distributed software and database systems*, pages 3–12. Los Angeles, CA, USA, 1986.

[94] Michael Grottke and Kishor S Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer*, 40(2):107–109, 2007.

[95] Jing Gu, Long Wang, Yong Yang, and Ying Li. Kerep: Experience in extracting knowledge on distributed system behavior through request execution path. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 30–35. IEEE, 2018.

[96] Qiang Guan, Ziming Zhang, and Song Fu. Ensemble of bayesian predictors for autonomic failure management in cloud computing. In *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–6. IEEE, 2011.

[97] Anton Gulenko, Florian Schmidt, Alexander Acker, Marcel Wallschläger, Odej Kao, and Feng Liu. Detecting anomalous behavior of black-box services modeled with distance-based online clustering. In *2018 IEEE 11th*

International Conference on Cloud Computing (CLOUD), pages 912–915. IEEE, 2018.

[98] Haryadi S Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M Hellerstein, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. Fate and destini: A framework for cloud recovery testing. In *Proceedings of NSDI'11: 8th USENIX Symposium on Networked Systems Design and Implementation*, page 239, 2011.

[99] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM symposium on cloud computing*, pages 1–14, 2014.

[100] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 1–16, 2016.

[101] Liang Guo, Yaguo Lei, Saibo Xing, Tao Yan, and Naipeng Li. Deep convolutional transfer learning network: A new method for intelligent fault diagnosis of machines with unlabeled data. *IEEE Transactions on Industrial Electronics*, 66(9):7316–7325, 2018.

[102] Xifeng Guo, Long Gao, Xinwang Liu, and Jianping Yin. Improved deep embedded clustering with local structure preservation. In *Ijcai*, pages 1753–1759, 2017.

[103] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. Clustering validity checking methods: Part ii. *ACM Sigmod Record*, 31(3):19–27, 2002.

[104] Julien Happich. Automated fault injection without source code change. http://www.eenewseurope.com/news/automated-fault-injection-without-source-code-change.

[105] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Fault isolation for device drivers. In *2009 IEEE/IFIP In-*

ternational Conference on Dependable Systems & Networks, pages 33–42. IEEE, 2009.

[106] William Hoarau, Sébastien Tixeuil, and Fabien Vauchelles. Fail-fci: Versatile fault injection. *Future Generation Computer Systems*, 23(7):913–919, 2007.

[107] Kjell J Hole and Christian Otterstad. Software systems with antifragility to downtime. *Computer*, 52(2):23–31, 2019.

[108] Andrea Höller, Gerhard Schönfelder, Nermin Kajtazovic, Tobias Rauter, and Christian Kreiner. Fies: a fault injection framework for the evaluation of self-tests for cots-based safety-critical systems. In *2014 15th International Microprocessor Test and Verification Workshop*, pages 105–110. IEEE, 2014.

[109] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[110] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[111] Fuqun Huang. Human error analysis in software engineering. In Fabio De Felice and Antonella Petrillo, editors, *Theory and Application on Cognitive Factors and Risk Management*, chapter 2. IntechOpen, Rijeka, 2017.

[112] Fuqun Huang, Bin Liu, and Bing Huang. A taxonomy system to identify human error causes for software defects. In *The 18th international conference on reliability and quality in design*, pages 44–49, 2012.

[113] Jia Huang, Jian-Xin You, Hu-Chen Liu, and Ming-Shun Song. Failure mode and effect analysis improvement: A systematic literature review and future research agenda. *Reliability Engineering & System Safety*, 199:106885, 2020.

[114] Yennun Huang, Chandra Kintala, Nick Kolettis, and N Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Twenty-fifth international symposium on fault-tolerant computing. Digest of papers*, pages 381–390. IEEE, 1995.

[115] James Wayne Hunt and M Douglas MacIlroy. *An algorithm for differential file comparison.* Bell Laboratories Murray Hill, 1976.

[116] Kai Hwang, Ying Chen, and Hua Liu. Defending distributed systems against malicious intrusions and network anomalies. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 8–pp. IEEE, 2005.

[117] Mohammad S Islam, William Pourmajidi, Lei Zhang, John Steinbacher, Tony Erwin, and Andriy Miranskyy. Anomaly detection in a large-scale cloud platform. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 150–159. IEEE, 2021.

[118] Mohammed Jabi, Marco Pedersoli, Amar Mitiche, and Ismail Ben Ayed. Deep clustering: On the link between discriminative models and k-means. *IEEE transactions on pattern analysis and machine intelligence*, 43(6):1887–1896, 2019.

[119] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

[120] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[121] Zu-Ming Jiang, Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. Fuzzing error handling code in device drivers based on software fault injection. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 128–138. IEEE, 2019.

[122] Andréas Johansson and Neeraj Suri. Error propagation profiling of operating systems. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 86–95. IEEE, 2005.

[123] Pallavi Joshi, Haryadi S Gunawi, and Koushik Sen. Prefail: A programmable tool for multiple-failure injection. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 171–188, 2011.

[124] Xiaoen Ju, Livio Soares, Kang G Shin, Kyung Dong Ryu, and Dilma Da Silva. On fault resilience of openstack. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–16, 2013.

[125] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665, 2014.

[126] W-I Kao, Ravishankar K. Iyer, and Dong Tang. Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults. *IEEE Transactions on Software Engineering*, 19(11):1105–1118, 1993.

[127] Kyle Kingsbury. Jepsen: A framework for distributed systems verification, with fault injection. https://github.com/jepsen-io/jepsen, 2018.

[128] Maryam Koohzadi, Nasrollah Moghadam Charkari, and Foad Ghaderi. Unsupervised representation learning based on the deep multi-view ensemble learning. *Applied Intelligence*, 50(2):562–581, 2020.

[129] Mu-Hsing Kuo. Opportunities and challenges of cloud computing to improve health care services. *Journal of medical Internet research*, 13(3):e67, 2011.

[130] Anna Lanzaro, Roberto Natella, Stefan Winter, Domenico Cotroneo, and Neeraj Suri. An empirical study of injected versus actual interface errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 397–408, 2014.

[131] Launchpad. Bug #732549 "execvp fallout". https://bugs.launchpad.net/nova/+bug/732549, Mar. 2011.

[132] Launchpad. Bug #1028174 "tenant cannot delete network when dhcp-agent is running". https://bugs.launchpad.net/neutron/+bug/1028174, Jul. 2013.

[133] Launchpad. Bug #1096722 "inconsistent nova-bm state will prevent launching new instances". https://bugs.launchpad.net/nova/+bug/1096722, Jan. 2013.

[134] Inhwan Lee and Ravishankar K Iyer. Faults, symptoms, and software fault tolerance in the tandem guardian90 operating system. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 20–29. IEEE, 1993.

[135] Matthew Leeke and Arshad Jhumka. Evaluating the use of reference run models in fault injection analysis. In *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 121–124. IEEE, 2009.

[136] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 517–530, 2016.

[137] Gizelle Sandrini Lemos and Eliane Martins. Specification-guided golden run for analysis of robustness testing results. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 157–166. IEEE, 2012.

[138] Fengfu Li, Hong Qiao, and Bo Zhang. Discriminatively boosted image clustering with fully convolutional auto-encoders. *Pattern Recognition*, 83:161–173, 2018.

[139] Heng Li, Weiyi Shang, and Ahmed E Hassan. Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, 22(4):1684–1716, 2017.

[140] Zhongwei Li, Qinghua Lu, Liming Zhu, Xiwei Xu, Yue Liu, and Weishan Zhang. An empirical study of cloud api issues. *IEEE Cloud Computing*, 5(2):58–72, 2018.

[141] libvirt. Home page of libvirt. https://www.libvirt.org.

[142] Chinghway Lim, Navjot Singh, and Shalini Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 398–403. IEEE, 2008.

[143] Hu-Chen Liu, Xu-Qi Chen, Jian-Xin You, and Zhiwu Li. A new integrated approach for risk evaluation and classification with dynamic expert weights. *IEEE Transactions on Reliability*, 2020.

[144] Hu-Chen Liu, Yu-Ping Hu, Jing-Jing Wang, and Minghe Sun. Failure mode and effects analysis using two-dimensional uncertain linguistic variables and alternative queuing method. *IEEE Transactions on Reliability*, 68(2):554–565, 2018.

[145] Xugang Lu, Yu Tsao, Shigeki Matsuda, and Chiori Hori. Speech enhancement based on deep denoising autoencoder. In *Interspeech*, volume 2013, pages 436–440, 2013.

[146] Michael R Lyu. Software reliability engineering: A roadmap. In *Future of Software Engineering (FOSE'07)*, pages 153–170. IEEE, 2007.

[147] Henrique Madeira, Diamantino Costa, and Marco Vieira. On the emulation of software faults by software fault injection. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 417–426. IEEE, 2000.

[148] Adetokunbo Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. System state discovery via information content clustering of system logs. In *2011 Sixth International Conference on Availability, Reliability and Security*, pages 301–306. IEEE, 2011.

[149] Leonardo Mariani, Cristina Monni, Mauro Pezzé, Oliviero Riganelli, and Rui Xin. Localizing faults in cloud systems. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 262–273. IEEE, 2018.

[150] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Rui Xin. Predicting failures in multi-tier distributed systems. *Journal of Systems and Software*, 161:110464, 2020.

[151] Paul D Marinescu and George Candea. Lfi: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 379–388. IEEE, 2009.

[152] Andrey Markelov. How to build your own virtual test environment. In *Certified OpenStack Administrator Study Guide*, pages 7–18. Springer, 2016.

[153] Matias Martinez, Laurence Duchien, and Martin Monperrus. Automatically extracting instances of code change patterns with ast analysis. In *2013 IEEE international conference on software maintenance*, pages 388–391. IEEE, 2013.

[154] S Mohammad Mavadati, Huanghao Feng, Anibal Gutierrez, and Mohammad H Mahoor. Comparing the gaze responses of children with autism and typically developed individuals in human-robot interaction. In *2014 IEEE-RAS International Conference on Humanoid Robots*, pages 1128–1133. IEEE, 2014.

[155] Steve McConnell. *Code complete*. Pearson Education, 2004.

[156] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[157] Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, Matthias Urban, Michael Burkart, Maximilian Dippel, Marius Lindauer, and Frank Hutter. Towards automatically-tuned deep neural networks. In *Automated Machine Learning*, pages 135–149. Springer, Cham, 2019.

[158] Microsoft Corp. Bulkhead pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead.

[159] Microsoft Corp. Circuit breaker pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker.

[160] Dharmendra S Modha and W Scott Spangler. Feature weighting in k-means clustering. *Machine learning*, 52(3):217–237, 2003.

[161] Alistair Moffat. Implementing the ppm data compression scheme. *IEEE Transactions on communications*, 38(11):1917–1921, 1990.

[162] S Mostafa Mousavi, Weiqiang Zhu, William Ellsworth, and Gregory Beroza. Unsupervised clustering of seismic signals using deep convolutional autoencoders. *IEEE Geoscience and Remote Sensing Letters*, 16(11):1693–1697, 2019.

[163] mpld3. Home page of mpld3. http://mpld3.github.io.

[164] Pooya Musavi, Bram Adams, and Foutse Khomh. Experience report: An empirical study of api failures in openstack cloud environments. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 424–434. IEEE, 2016.

[165] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[166] Roberto Natella, Domenico Cotroneo, and Henrique S Madeira. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)*, 48(3):1–55, 2016.

[167] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. Anomaly detection from system tracing data using multimodal deep learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 179–186. IEEE, 2019.

[168] Netflix. Chaos Monkey. https://netflix.github.io/chaosmonkey.

[169] Netflix, Inc. Hystrix: Latency and fault tolerance for distributed systems. https://github.com/Netflix/Hystrix/wiki/How-it-Works.

[170] Wee Teck Ng, Christopher M Aycock, Gurushankar Rajamani, and Peter M Chen. Comparing disk and memory's resistance to operating system crashes. In *Proceedings of ISSRE'96: 7th International Symposium on Software Reliability Engineering*, pages 185–194. IEEE, 1996.

[171] Wee Teck Ng and Peter M Chen. The design and verification of the rio file cache. *IEEE Transactions on Computers*, 50(4):322–337, 2001.

[172] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Communications of the ACM*, 55(2):55–61, 2012.

[173] OpenStack. Block Storage API. https://developer.openstack.org/api-ref/block-storage.

[174] OpenStack. Compute API. https://developer.openstack.org/api-ref/compute.

[175] OpenStack. Design. https://docs.openstack.org/arch-design/design.html.

[176] OpenStack. Home page of OpenStack. https://www.openstack.org.

[177] OpenStack. Launch an instance from an image. https://docs.openstack.org/ocata/user-guide/cli-nova-launch-instance-from-image.html.

[178] OpenStack. Networking API v2.0. https://developer.openstack.org/api-ref/network/v2.

[179] OpenStack. OpenStack in Launchpad. https://launchpad.net/openstack.

[180] OpenStack. The OpenStack marketplace. https://www.openstack.org/marketplace.

[181] OpenStack. Oslo Context Usage. https://docs.openstack.org/oslo.context/latest/user/usage.html.

[182] OpenStack. osprofiler. https://docs.openstack.org/osprofiler/latest.

[183] OpenStack. RPC Client. https://docs.openstack.org/oslo.messaging/latest/reference/rpcclient.html.

[184] OpenStack. Stackalytics. https://www.stackalytics.com.

[185] OpenStack. Tempest Testing Project. https://docs.openstack.org/tempest.

[186] OpenStack. Virtual Machine States and Transitions. https://docs.openstack.org/nova/latest/reference/vm-states.html.

[187] OpenStack. The world #RunsOnOpenStack. https://www.openstack.org/user-stories.

[188] David Oppenheimer, Archana Ganapathi, and David A Patterson. Why do internet services fail, and what can be done about it? In *USENIX symposium on internet technologies and systems*, volume 67. Seattle, WA, 2003.

[189] Matteo Orrú, Ewan Tempero, Michele Marchesi, Roberto Tonelli, and Giuseppe Destefanis. A curated benchmark collection of python systems for empirical studies on software engineering. In *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 1–4, 2015.

[190] Lucas Palazzi, Guanpeng Li, Bo Fang, and Karthik Pattabiraman. A tale of two injectors: End-to-end comparison of ir-level and assembly-level fault injection. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 151–162. IEEE, 2019.

[191] Kai Pan, Sunghun Kim, and E James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[192] Pandiyan M. Openstack instance creation – request work flow. https://maestropandy.wordpress.com/2016/05/26/openstack-instance-creation-request-work-flow/.

[193] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.

[194] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 537–548. IEEE, 2018.

[195] Xi Peng, Hongyuan Zhu, Jiashi Feng, Chunhua Shen, Haixian Zhang, and Joey Tianyi Zhou. Deep clustering with sample-assignment invariance prior. *IEEE transactions on neural networks and learning systems*, 31(11):4857–4868, 2019.

[196] Fabio Petrillo, Philippe Merle, Naouel Moha, and Yann-Gaël Guéhéneuc. Are rest apis for cloud computing well-designed? an exploratory study. In *International Conference on Service-Oriented Computing*, pages 157–170. Springer, 2016.

[197] Cuong Pham, Daniel Chen, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Cloudval: A framework for validation of virtualization environment in cloud infrastructure. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 189–196. IEEE, 2011.

[198] Cuong Pham, Long Wang, Byung Chul Tak, Salman Baset, Chunqiang Tang, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Failure diagnosis for distributed systems using targeted fault injection. *IEEE Transactions on Parallel and Distributed Systems*, 28(2):503–516, 2016.

[199] Amir Pnueli. The temporal logic of programs. In *Proc. SFCS*, pages 46–57. IEEE, 1977.

[200] Alexander Power and Gerald Kotonya. Providing fault tolerance via complex event processing and machine learning for iot systems. In *Proc. IoT*, pages 1–7, 2019.

[201] pypi. python-etcd. https://pypi.org/project/python-etcd.

[202] Python. Applications for Python. https://www.python.org/about/apps.

[203] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[204] Yan Qiao, XW Xin, Yang Bin, and S Ge. Anomaly intrusion detection method based on hmm. *Electronics letters*, 38(13):663–664, 2002.

[205] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[206] Rick Rabiser, Sam Guinea, Michael Vierhauser, Luciano Baresi, and Paul Grünbacher. A comparison framework for runtime monitoring approaches. *Journal of Systems and Software*, 125:309–321, 2017.

[207] Alireza Rahimi, Ghazaleh Azimi, Hamidreza Asgari, and Xia Jin. Clustering approach toward large truck crash analysis. *Transportation research record*, 2673(8):73–85, 2019.

[208] RDO. Packstack: Create a proof of concept cloud. https://www.rdoproject.org/install/packstack.

[209] Red Hat, Inc. Evaluating OpenStack: Single-Node Deployment. https://access.redhat.com/articles/1127153.

[210] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, volume 6, pages 9–9, 2006.

[211] Oliviero Riganelli, Paolo Saltarel, Alessandro Tundo, Marco Mobilio, and Leonardo Mariani. Cloud failure prediction with hierarchical temporary memory: An empirical assessment. *arXiv preprint arXiv:2110.03431*, 2021.

[212] Jorma Rissanen. A universal data compression system. *IEEE Transactions on information theory*, 29(5):656–664, 1983.

[213] Jesse Robbins, Kripa Krishnan, John Allspaw, and Thomas A Limoncelli. Resilience engineering: Learning to embrace failure: A discussion with jesse robbins, kripa krishnan, john allspaw, and tom limoncelli. *Queue*, 10(9):20–28, 2012.

[214] Gema Rodríguez-Pérez, Andy Zaidman, Alexander Serebrenik, Gregorio Robles, and Jesús M González-Barahona. What if a bug has a different origin? making sense of bugs without an explicit bug introducing change. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–4, 2018.

[215] Romil Gupta. Request Flow for Provisioning Instance in Openstack. https://ilearnstack.wordpress.com/2013/04/26/request-flow-for-provisioning-instance-in-openstack/.

[216] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[217] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 224–234, 2018.

[218] Salesforce Engineering. Anomaly Detection in Zipkin Trace Data. https://engineering.salesforce.com/anomaly-detection-in-zipkin-trace-data-87c8a2ded8a1.

[219] Ben Sampson. How to automate software fault injection testing, without changing source code. https://www.aerospacetestinginternational.com/opinion/how-to-automate-software-fault-injection-testing-without-changing-source-html.

[220] Bruno Pacheco Sanches, Tânia Basso, and Regina Moraes. J-swfit: A java software fault injection tool. In *2011 5th Latin-American Symposium on Dependable Computing*, pages 106–115. IEEE, 2011.

[221] Suhrid Satyal, Ingo Weber, Len Bass, and Min Fu. Rollback mechanisms for cloud management apis using ai planning. *IEEE Transactions on Dependable and Secure Computing*, 17(1):148–161, 2017.

[222] Carla Sauvanaud, Kahina Lazri, Mohamed Kaâniche, and Karama Kanoun. Anomaly detection and root cause localization in virtual network functions. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 196–206. IEEE, 2016.

[223] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. Fail*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *2015 11th european dependable computing conference (edcc)*, pages 245–255. IEEE, 2015.

[224] Oliver Schwahn, Nicolas Coppik, Stefan Winter, and Neeraj Suri. Fastfi: Accelerating software fault injections. In *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 193–202. IEEE, 2018.

[225] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[226] Daniel Skarin, Raul Barbosa, and Johan Karlsson. Goofi-2: A tool for experimental dependability assessment. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 557–562. IEEE, 2010.

[227] Michael Solberg and Ben Silverman. *OpenStack for architects*. Packt Publishing Ltd, 2017.

[228] Stack Overflow. Developer survey results. https://insights.stackoverflow.com/survey/2019.

[229] Stackalytics. OpenStack Pike Commits. https://www.stackalytics.com/?release=pike&metric=commits.

[230] Stackalytics. OpenStack Pike Lines of Code. https://www.stackalytics.com/?release=pike&metric=loc.

[231] Mario Stanke and Stephan Waack. Gene prediction with a hidden markov model and a new intron submodel. *Bioinformatics*, 19(suppl_2):ii215–ii225, 2003.

[232] Kyriakos Stefanidis and Artemios G Voyiatzis. An hmm-based anomaly detection approach for scada systems. In *IFIP International Conference on Information Security Theory and Practice*, pages 85–99. Springer, 2016.

[233] Manabu Sugimoto, Takafumi Kubota, and Kenji Kono. Short-liveness of error propagation in kernel can improve operating systems availability. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks–Supplemental Volume (DSN-S)*, pages 23–24. IEEE, 2019.

[234] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability: A study of field failures in operating systems. In *FTCS*, volume 21, pages 2–9, 1991.

[235] Mark Sullivan and Ram Chillarege. A comparison of software defects in database management systems and operating systems. In *FTCS*, pages 475–484, 1992.

[236] Nabil Sultan. Making use of cloud computing for healthcare provision: Opportunities and challenges. *International Journal of Information Management*, 34(2):177–184, 2014.

[237] Michael M Swift, Muthukaruppan Annamalai, Brian N Bershad, and Henry M Levy. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS)*, 24(4):333–360, 2006.

[238] Hamed Tabrizchi and Marjan Kuchaki Rafsanjani. A survey on security challenges in cloud computing: issues, threats, and solutions. *The journal of supercomputing*, 76(12):9493–9532, 2020.

[239] Andrew S Tanenbaum, Jorrit N Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

[240] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–312. IEEE, 2019.

[241] Twitter Engineering. Distributed Systems Tracing with Zipkin. https://blog.twitter.com/engineering/en_us/a/2012/distributed-systems-tracing-with-zipkin.

[242] Risto Vaarandi. A breadth-first algorithm for mining frequent patterns from event logs. In *International Conference on Intelligence in Communication Systems*, pages 293–308. Springer, 2004.

[243] Erik Van Der Kouwe and Andrew S Tanenbaum. Hsfi: Accurate fault injection scalable to large code bases. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 144–155. IEEE, 2016.

[244] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[245] T Velmurugan and T Santhanam. Computational complexity between k-means and k-medoids clustering algorithms for normal and uniform distributions of data points. *Journal of computer science*, 6(3):363, 2010.

[246] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

[247] E Voas, Frank Charron, Gary McGraw, Keith Miller, and Michael Friedman. Predicting how badly" good" software can behave. *IEEE Software*, 14(4):73–83, 1997.

[248] Jeffrey Voas, Anup Ghosh, Frank Charron, and Lora Kassab. Reducing uncertainty about common-mode failures. In *Proceedings The Eighth International Symposium on Software Reliability Engineering*, pages 308–319. IEEE, 1997.

[249] Ingo Weber, Hiroshi Wada, Alan Fekete, Anna Liu, and Len Bass. Automatic undo for cloud management via {AI} planning. In *Eighth Workshop on Hot Topics in System Dependability (HotDep 12)*, 2012.

[250] Stefan Winter, Constantin Sârbu, Neeraj Suri, and Brendan Murphy. The impact of fault models on software robustness evaluations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 51–60, 2011.

[251] Stefan Winter, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. No pain, no gain? the utility of parallel fault injections. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 494–505. IEEE, 2015.

[252] Ian H Witten and Timothy C Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *Ieee transactions on information theory*, 37(4):1085–1094, 1991.

[253] Katinka Wolter, Alberto Avritzer, Marco Vieira, and Aad Van Moorsel. *Resilience assessment and evaluation of computing systems*. Springer, 2012.

[254] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proc. SIGMOD/PODS*, pages 407–418, 2006.

[255] Li Wu, Jasmin Bogatinovski, Sasho Nedelkoski, Johan Tordsson, and Odej Kao. Performance diagnosis in cloud microservices using deep learning. In *International Conference on Service-Oriented Computing*, pages 85–96. Springer, 2020.

[256] Junyuan Xie, Ross Girshick, and Ali Farhadi. Unsupervised deep embedding for clustering analysis. In *International conference on machine learning*, pages 478–487. PMLR, 2016.

[257] Hui Xiong, Junjie Wu, and Jian Chen. K-means clustering versus validation measures: a data-distribution perspective. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 39(2):318–331, 2008.

[258] Rui Xu and Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on neural networks*, 16(3):645–678, 2005.

[259] Zhaoguang Xu, Yanzhong Dang, Peter Munro, and Yuhang Wang. A data-driven approach for constructing the component-failure mode matrix for fmea. *Journal of Intelligent Manufacturing*, 31(1):249–265, 2020.

[260] Zhou Xu, Tao Zhang, Jacky Keung, Meng Yan, Xiapu Luo, Xiaohong Zhang, Ling Xu, and Yutian Tang. Feature selection and embedding based cross project framework for identifying crashing fault residence. *Information and Software Technology*, 131:106452, 2021.

[261] Maysam Yabandeh, Abhishek Anand, Marco Canini, and Dejan Kostic. Finding almost-invariants in distributed systems. In *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, pages 177–182. IEEE, 2011.

[262] Bo Yang, Xiao Fu, Nicholas D Sidiropoulos, and Mingyi Hong. Towards k-means-friendly spaces: Simultaneous deep learning and clustering. In *international conference on machine learning*, pages 3861–3870. PMLR, 2017.

[263] Nong Ye, Yebin Zhang, and Connie M Borror. Robustness of the markov-chain model for cyber-attack detection. *IEEE transactions on reliability*, 53(1):116–123, 2004.

[264] Zhiyuan Yin, F Richard Yu, Shengrong Bu, and Zhu Han. Joint cloud and wireless networks operations in mobile cloud computing environments with telecom operator cloud. *IEEE Transactions on Wireless Communications*, 14(7):4020–4033, 2015.

[265] Takeshi Yoshimura, Hiroshi Yamada, and Kenji Kono. Using fault injection to analyze the scope of error propagation in linux. *Information and Media Technologies*, 8(3):655–664, 2013.

[266] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 249–265, 2014.

[267] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 293–306, 2012.

[268] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, 30(1):1–28, 2012.

[269] Wei Zhang, Xiaowei Dong, Huaibao Li, Jin Xu, and Dan Wang. Unsupervised detection of abnormal electricity consumption behavior based on feature engineering. *IEEE Access*, 8:55483–55500, 2020.

[270] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 19–33, 2017.

[271] Hao Zhong and Na Meng. Towards reusing hints from past fixes. *Empirical Software Engineering*, 23(5):2521–2549, 2018.

[272] Jingwen Zhou, Zhenbang Chen, Ji Wang, Zibin Zheng, and Wei Dong. A runtime verification based trace-oriented monitoring framework for cloud

systems. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 152–155. IEEE, 2014.

[273] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th International Conference on Software Engineering*, pages 415–425, 2015.

[274] Zipkin. Home page of Zipkin. https://zipkin.io.

[275] Saman Zonouz, Katherine M Rogers, Robin Berthier, Rakesh B Bobba, William H Sanders, and Thomas J Overbye. Scpse: Security-oriented cyber-physical state estimation for power grid critical infrastructures. *IEEE Transactions on Smart Grid*, 3(4):1790–1799, 2012.