The Journal of Systems & Software 198 (2023) 111611

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Run-time failure detection via non-intrusive event analysis in a large-scale cloud computing platform☆

Domenico Cotroneo, Luigi De Simone, Pietro Liguori *, Roberto Natella

Università degli Studi di Napoli Federico II, Naples, Italy

ARTICLE INFO

Article history:
Received 1 September 2022
Received in revised form 19 November 2022
Accepted 3 January 2023
Available online 6 January 2023

Keywords:
Run-time verification
Failure detection
Cloud computing
OpenStack
Fault injection

ABSTRACT

Cloud computing systems fail in complex and unforeseen ways due to unexpected combinations of events and interactions among hardware and software components. These failures are especially problematic when they are silent, i.e., not accompanied by any explicit failure notification, hindering the timely detection and recovery.

In this work, we propose an approach to run-time failure detection tailored for monitoring multi-tenant and concurrent cloud computing systems. The approach uses a non-intrusive form of event tracing, without manual changes to the system's internals to propagate session identifiers (IDs), and builds a set of lightweight monitoring rules from fault-free executions. We evaluated the effectiveness of the approach in detecting failures in the context of the OpenStack cloud computing platform, a complex and "off-the-shelf" distributed system, by executing a campaign of fault injection experiments in a multi-tenant scenario. Our experiments show that the approach detects the failure with an F_1 score (0.85) and accuracy (0.77) higher than the ones provided by the OpenStack failure logging mechanisms (0.53 and 0.50) and two non-session-aware run-time verification approaches (both lower than 0.15). Moreover, the approach significantly decreases the average time to detect failures at run-time ($\sim$ 114 seconds) compared to the OpenStack logging mechanisms.

1. Introduction

As computer systems grow increasingly complex, they become more error-prone, stemming from their requirements specification, their design, their implementation, or their operating environment (Ammar et al., 2000). This is the case of *cloud computing systems*. Indeed, they include processes distributed across a data center, which often cooperate by message passing and remote procedure calls (e.g., message queues and REST API calls mechanisms). They are quite complex, as they typically consist of software components of millions of lines of code (LoC) that run across dozens of computing nodes.

It is well established that cloud computing systems failures have a huge economic impact on both providers and their customers (Li et al., 2018; Musavi et al., 2016; Gunawi et al., 2014, 2016). To make things worse, cloud computing systems also 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 contemplated ahead during the design phase (Garraghan et al., 2018; Hole and Otterstad, 2019). These failures are especially problematic when they are *silent*, that is, they are not accompanied by any explicit notification, such as API error codes or error entries in the logs. This behavior hinders timely detection and recovery, makes failures silently propagate through the system, and makes root cause analysis tricky and recovery actions more costly (e.g., reverting the database state) (Cotroneo et al., 2019a,c).

Therefore, the prompt detection of the failure at run-time (i.e., *run-time failure detection*) is a key step to improving the fault-tolerance and recovery mechanisms within cloud infrastructures. Generally, logging mechanisms are the main source of information to monitor operation behavior (Farshchi et al., 2015), but include several limitations since logs are noisy and lack information on changes in resource states (Oliner et al., 2012). An effective solution is represented by *run-time verification* strategies, which perform checks over events in the system (e.g., after-service API calls) to assert whether the resources are in a valid state (Bartocci et al., 2018). These checks can be specified as *monitoring rules* using temporal logic and synthesized in a run-time monitor (Delgado et al., 2004; Chen and Roşu, 2007; Zhou et al., 2014; Rabiser et al., 2017; Cotroneo et al., 2018).

Unfortunately, the application of these strategies in the context of cloud computing systems is very challenging (Zhou et al.,

☆ Editor: J.C. Duenas.

* Corresponding author.
E-mail addresses: cotroneo@unina.it (D. Cotroneo), luigi.desimone@unina.it (L. De Simone), pietro.liguori@unina.it (P. Liguori), roberto.natella@unina.it (R. Natella).

https://doi.org/10.1016/j.jss.2023.111611

2014; Farshchi et al., 2015). In practice, in multi-tenant and concurrent systems, the monitoring rules can be applied as long as the checked events are accompanied by *session identifiers* (IDs). Such IDs allow monitoring solutions to correlate events that belong to the same session (i.e., a set of operations performed on behalf of the same tenant, or by the same subsystem) and to perform checks, e.g., to detect omissions or out-of-order events (van Hoorn et al., 2009; Las-Casas et al., 2019; Horovitz et al., 2019; Krause, 2021; Li et al., 2022). Keeping track of IDs in distributed tracing systems requires intrusive modifications of systems' internals since IDs need to be propagated across API call chains over several components. However, the code instrumentation of the system requires an in-depth knowledge of its internals and is unfeasible for complex and "off-the-shelf" systems (Parker et al., 2020; Horovits, 2022). Moreover, this problem is exacerbated by the high number of requests and tenants, which trigger multiple sub-requests within the distributed system. For example, a simple Google search request triggers more than 200 subrequests and crosses hundreds of servers (Zhou et al., 2014). Since requests performed by concurrent tenants may overlap over time, the correlation of events to the same session is a cumbersome task without using any ID.

Last but not least, events in complex distributed systems are often asynchronous and non-deterministic, thus run-time verification approaches may heavily suffer from false positives/negatives (Satyanarayanan et al., 1992; Perrochon, 1998; Cotroneo et al., 2019a, 2020a). Sophisticated approaches check formal specifications over events and outputs, by using finite state machines (Deligiannis et al., 2016), temporal logic predicates (Arlat et al., 2002), relational logic (Gunawi et al., 2011), and special-purpose languages (Reynolds et al., 2006). 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 these approaches less applicable in practice.

To overcome these limitations, in this work we propose an approach (*Monitoring Rules*, MR) to run-time verification tailored for the monitoring and analysis of cloud computing systems. The approach uses a non-intrusive form of event tracing that does not require manual changes to the system's internals for propagating IDs. Instead, it automatically analyzes the raw (i.e., unmodified) events already produced by the system (e.g., raw RPC calls and messages over queues); then, it mines relationships among attributes within these events to correlate them; finally, the approach builds a set of lightweight monitoring rules on correlated events from a limited set of "normal" (i.e., *fault-free*) executions of the system. These rules encode the expected behavior of the system and detect a failure if a violation occurs. The proposed approach does not require any in-depth knowledge about the internals of the system, and it is designed to fit in concurrent and multi-tenant environments.

We evaluated the effectiveness of the approach in detecting failures in the context of the OpenStack cloud computing platform by executing a campaign of fault injection experiments in multi-tenant scenarios. OpenStack is a complex and "off-the-shelf" distributed system, and represents an important case study as it is one of the most widely deployed open-source cloud software in the world. The complexity of this system, which consists of over 1 million lines of Python code, makes this cloud platform widely targeted by research studies (Mariani et al., 2020; Ou et al., 2018; Wu et al., 2020; Cotroneo et al., 2021; Zheng et al., 2019). Our experiments show that the approach infers a set of monitoring rules by analyzing 50 fault-free executions of the system, and detects the failures with an F_1 score (0.85) and accuracy (0.77) higher than the ones provided by the OpenStack failure logging mechanisms (0.53 and 0.50) and two non-session-aware run-time verification approaches (both lower than 0.15). Moreover, the approach significantly decreases the average time to detect failures at run-time ($\sim$114 s) compared to the system's failure logging mechanisms.

In the following, Section 2 discusses related work; Section 3 presents the overview of the proposed approach; Section 4 introduces the OpenStack case study; Section 5 describes the process to infer the monitoring rules; Section 6 shows an implementation of the monitoring rules in a specification language; Section 7 experimentally evaluates the approach; Section 8 discusses the threats to validity; Section 9 concludes the paper.

2. Related work

In literature, some studies refer to run-time verification as *run-time monitoring* or *dynamic analysis*.

Monitoring and debugging distributed systems. Over the last decades, several efforts have been spent on methodologies and tools for monitoring and debugging distributed systems. For example, Magpie (Barham et al., 2003), Pinpoint (Chen, 2004), and Aguilera et al. (2003) identify causal paths in the distributed system, by tracing and correlating call requests and responses, and events at both the OS level and the application server level. These approaches were still too difficult to apply in practice, as they either focused only on synchronous (RPC-style) interactions between components and neglected asynchronous and concurrent ones; or, they required intrusive instrumentation of the entire software stack down to the OS. Pensieve (Zhang et al., 2017) is a tool for automatically reproducing failures from production distributed systems. Given log files output by the failure execution, the system's bytecode, a list of supported user commands, and a description of the symptoms associated with the failure, the tool outputs a sequence of user commands, packaged as a unit test, that can reliably reproduce the failure. Friday (Geels et al., 2007) 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 (Beschastnikh et al., 2016) is an interactive tool for visualizing execution traces of distributed systems, which allows developers to intuitively explore the traces and perform searches; moreover, the tool provides support for comparing distributed executions with a pairwise comparison. Gu et al. (2018) proposed a methodology to extract knowledge about the behavior of the distributed system 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. *LOUD* (Mariani et al., 2018) is an online metric-driven fault localization technique, which analyzes the dependencies among anomalous Key Performance Indicators (KPIs) commonly available in software systems at different abstraction levels to pinpoint the faulty resources that are likely responsible for future failures. Pip (Reynolds et al., 2006) 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 propagate path identifiers across chains of requests. Similar to Pip, Watchtower (Alpernas et al., 2021) is a run-time verification tool that analyzes serverless application logs to detect property violations. This tool accepts as input one or more safety properties, and then monitors and analyzes the application at run-time to detect violations. Cotroneo et al.

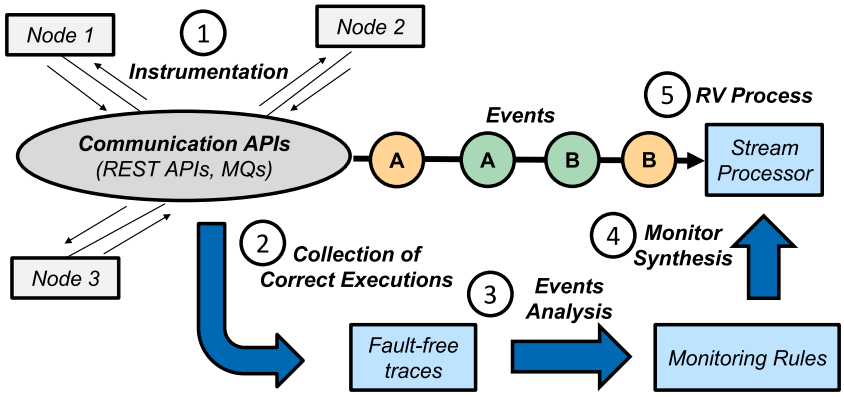

Fig. 1. Overview of the proposed approach.

(2019a) proposed a probabilistic approach, based on Variable-order Markov Model (Begleiter et al., 2004), to identify failures in cloud computing systems. The approach showed to be very effective in reducing false positives in a single-user workload. An et al. (2017) demonstrated that system call-based behavioral anomaly detection algorithms can effectively detect previously unknown malware on home routers, which is an essential component to the IoT, with high accuracy and low or no false alarms.

Property checking. Research studies on run-time verification focused on formalisms for describing properties to be verified. Typically, a run-time verification system provides a *Domain Specification Language* (DSL) for the description of properties to be verified. The DSL can be a standalone language or embedded in an existing language. Specification languages for run-time verification can be regular, which includes temporal logic, regular expressions, and state machines, but also nonregular, which includes rule systems, and stream languages. In the run-time verification literature, there is an established set of approaches for the specification of temporal properties, including *Event Processing Language* (EPL). This language is used to translate event patterns into queries that trigger event listeners and determine whether the pattern is observed in an event stream of a *Complex Event Processing* (CEP) environment (Wu et al., 2006), i.e., a technology for the collection, aggregation, and analysis of sequences of events originated from different sources and occurred at different time. Lola (d'Angelo et al., 2005) is a tool implementing run-time 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. Zhou et al. (2014) proposed a framework that brings run-time verification into the field of trace-oriented monitoring in cloud systems. The monitoring requirements of cloud systems can be specified by formal specification languages, such as finite-state machines, linear temporal logic, etc. Atlidakis et al. (2020) introduced security rules to capture desirable properties of REST APIs and services, and showed how a stateful REST API fuzzer can be extended with active property checkers that automatically test and detect violations of these rules in cloud systems. *Complex Patterns of Failure* (CPoF) (Power and Kotonya, 2019) is an approach that provides reactive and proactive fault tolerance through complex event processing and machine learning for IoT. The approach uses error events to train ML models to prevent and recover from errors in the future.

Our work exploits run-time verification to state the correctness of a system execution according to specific properties and proposes an approach presenting several points of novelty compared to state-of-the-art studies and tools in run-time verification literature. While previous work uses intrusive tracing of the system under test, or requires the check of formal specifications over events or system's outputs, the run-time verification approach proposed in this work i) simply relies on *black-box tracing* to collect the events exchanged in the system (i.e., it does not require knowledge about the system's internals) and ii) models the desired behavior of the system with a set of monitoring rules by analyzing the events collected during the system's operation. These monitoring rules fit cloud computing systems, where we need to face challenging aspects such as multi-tenancy and complex communication flow among the nodes of the system.

3. Approach

Fig. 1 shows an overview of the proposed approach. The approach is applied to distributed systems within several nodes, each of them providing services that can be requested by message -passing mechanisms.

First, the approach wraps the *communication APIs* of the system, which is instrumented accordingly to collect all messages exchanged by the nodes during operation (step ①). This instrumentation is a form of "black-box tracing" since we collect the messages exchanged among unmodified multi-module system (Koskinen and Jannotti, 2008) and is especially suitable for complex and distributed systems since it does not require any knowledge about the internals, but only basic information about the communication APIs being used. Moreover, this kind of tracing is already familiar to developers for debugging, performance monitoring and optimization, root cause analysis, and service dependency analysis (Chow et al., 2014; Chen et al., 2002). The information recorded by the instrumented APIs includes the time at which a communication API has been called and its duration, the node 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 code 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 system execution produces a ***trace*** of events that are ordered with respect to the timestamp given by an event collector. During the system execution, different events can be generated by different calls to the same API service invoked by the same message sender. In this case, we say that the events are of the same ***type***.

In step ②, we collect the *correct executions* of the system. To define its normal (i.e., correct) behavior, we exercise the system under "fault-free" conditions, that is, without injecting any faults. Moreover, to take into account the variability of the system,

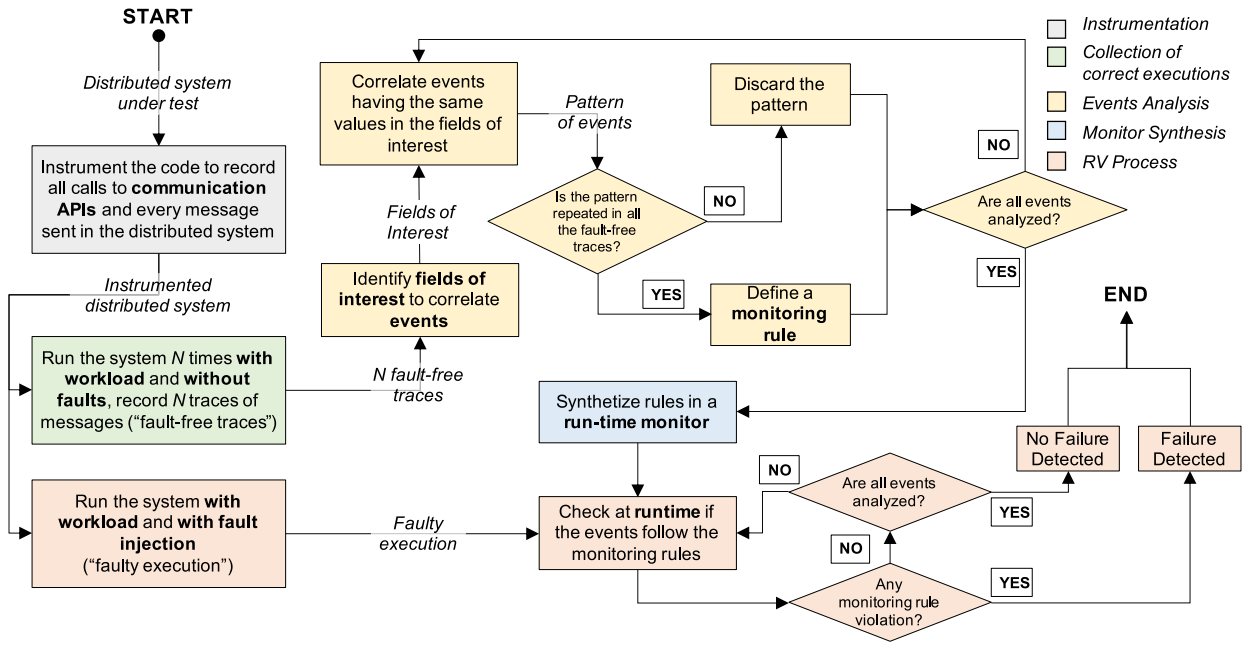

Fig. 2. Detailed workflow of the proposed approach.

we execute the system 100 times, collecting different "*fault-free traces*", one per execution.

Step ③ analyzes the collected fault-free traces to define a set of *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. The first is selecting key attributes of collected events (e.g., message sender, service API, event timestamp, etc.). Second, we define the failure monitoring rules by inferring *patterns* of events from the fault-free 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. Since we avoid using session identifiers to identify patterns, the approach uses an algorithm based on statistical analysis techniques (Ernst et al., 2007; Yabandeh et al., 2011; Grant et al., 2018) that analyzes the information contained in the body of the events and finds *fields of interest* to correlate different events.

Finally, in step ④ we synthesize an EPL-based *monitor* according to the obtained monitoring rules (Section 6). Because a system failure may cause missing or out-of-order events in the patterns, the monitor processes the stream of events during operation, and it checks, at run-time, whether the system behavior follows the desired behavior specified in the monitoring rules (step ⑤). To accelerate the occurrence of the failures, the approach performs the *fault-injection experiments*. We focus on injecting one fault per experiment, as injecting multiple faults concurrently is still an open research problem and has not yet been 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. Any (run-time) violation of the monitoring rules during the fault-injection experiments alerts the system operator that a failure occurred.

Fig. 2 shows a detailed flowchart of the proposed approach.

4. OpenStack case study

OpenStack is a cloud computing platform 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 tenants. It provides abstractions and APIs to programmatically create, destroy, and snapshot/revert virtual machine instances; attach and detach volumes and IP addresses; configure security, network topology, and load balancer settings; and many other services to cloud infrastructure consumers.

OpenStack consists of several independent parts, named *projects* (also referred to as *subsystems*). These projects are developed independently by dedicated teams (Stackalytics, 2022a,b), each representing a complex distributed system. The three most important subsystems of OpenStack (Denton, 2015; Solberg and Silverman, 2017) are: (i) Nova, which provides services for provisioning instances (VMs) and handling their life cycle; (ii) Cinder, which provides services for managing block storage for virtual instances; and (iii) Neutron, which provides services for provisioning virtual networks, including resources such as *floating IPs*, *ports* and *subnets* for instances. In turn, these subsystems include several distributed components (e.g., Nova includes *nova-api*, *nova-compute*, etc.), which interact through two communication protocols, i.e., HTTP-based RESTful APIs and remote procedure calls (RPC) (Bahl et al., 2012; Petrillo et al., 2016).

In OpenStack, tenants can send requests to a service via the dashboard or command line by using the API provided by a specific client developed within each project (e.g., *novaclient* is a client for the OpenStack Compute API). 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 involves sending an HTTP-based request to a particular node in the OpenStack cluster and then parsing the response. In the request, we can discern information such as the method invoked (e.g., GET, DELETE, POST, PUSH, etc.), the client performing the request (e.g., *cinderclient*, *neutronclient*, *novaclient*, etc.), and the status code (e.g., 2xx for successful requests, 4xx for client errors, 5xx for server-side errors, etc.). In the case of the REST API, we identify an event type with the pair client performing the request and the method invoked (e.g., `<novaclient, GET>`).

Furthermore, OpenStack internal subsystems (e.g., *nova-compute*, *cinder-volume*, etc.) use Advanced Message Queuing Protocol (AMQP), an open standard for messaging middleware. This messaging middleware enables the OpenStack services that run

on multiple nodes to talk to each other via RPC to serve tenants' requests. For example, when a tenant aims to create an instance, it invokes a REST API (i.e., the `/servers` POST method (OpenStack, 2022a)). The request is handled by the Nova subsystem, which starts communicating internally with other subsystems by using remote procedure calls. The first method invoked in the resulting flow of RPC messages is `schedule_and_build_instances`, then Nova exchanges messages with Keystone to verify the tenant's authentication, Glance to get the image, Neutron to create virtual networks, and Cinder for the block storage handling (OpenStack, 2022b)). The RPC messages contain information such as the method invoked, the caller (the system's service), and the body of the message. In the case of the RPC calls, we identify an event type with the pair subsystem providing the API and method invoked (e.g., `<cinder-volume, create_volume>`).

5. Events analysis

We can express a generic monitoring rule by observing the events in the traces. For example, suppose there is an event of a specific type, say A, that occurs before an event of a different type, say B, in the same tenant session (i.e., same ID). This monitoring rule can be translated into the following pseudo-formalism:

$$a \rightarrow b \ and \ id(a) = id(b), \quad with \ a \in A, \ b \in B \tag{1}$$

Generally, monitoring rules can be applied in multi-tenant cloud scenarios as long as the information on the tenant IDs is available. However, introducing IDs in distributed tracing systems requires both in-depth knowledge of the internals and intrusive instrumentation of the system. Therefore, to make our run-time 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 ID. To apply the rules in a multi-tenant scenario, we define two different sets of events, A and B, where A and B contain a set of n distinct events of type A and B, respectively, in a time window $[t_0, \ t_0 + \Delta t]$, assuming $|A| = |B| = n$.

Our monitoring rule for the multi-tenant case then asserts that there should exist a binary relation R over A and B such that:

$$\begin{aligned} R = \{(a,b) \in A \times B \mid a \rightarrow b, \\ \nexists \ a_i, a_j \in A, \ b_k \in B \mid (a_i, b_k), (a_j, b_k), \\ \nexists \ b_i, b_j \in B, \ a_k \in A \mid (a_k, b_i), (a_k, b_j) \ \} \end{aligned} \tag{2}$$

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 vice-versa. These rules are based on the observation that, if a group of tenants 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 impossible to verify the happened-before relation between the events a_i and b_i that refer to the same session or the same tenant 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.

5.1. Events correlation

To specify a monitoring rule, we need to identify one or more events characterizing the action taken by the tenant. In the example described in Section 4, the first RPC message exchanged among the subsystems includes the invocation of method `schedule_and_build_instances` by the `nova-conductor` component (it provides coordination and database query support for Nova). This RPC event follows the `/servers` POST method called by *novaclient*, but not every `<novaclient, POST>` event generates the `schedule_and_build_instances` call since the POST method can be used to create/add different resources, i.e., there is not a one-to-one relationship between the first RPC message of the event flows and the REST API starting the request.

Because we cannot discern patterns from the observation of the REST API calls, we need to look at the RPC messages. If we observe the method `schedule_and_build_instances` invoked by `nova-conductor` component, we infer that the tenant requested the creation of an instance. Similarly, if we observe the method `create_volume` invoked by the `cinder-scheduler` component (used to determine how to dispatch block storage requests), then we derive that the tenant aims to create a volume, and so on. By taking this into account, we refer to the first (with the lower timestamp) event that occurred in the pattern of RPC events as ***head event***.

To identify the monitoring rules, the approach focuses on finding patterns of events starting with a head event. The key idea is that, if we find a pattern of recurring events starting from a specific head event, then we can specify the rules to identify anomalies (e.g., out-of-order events, missing events, etc.). Unfortunately, due to the non-determinism of cloud systems, we cannot manually infer rules by simply observing fault-free executions. Indeed, the head event starting from a tenant request is not necessarily followed by the same number and/or the same order of events. Moreover, the high volume of messages in the system makes manual inspection very difficult and prone to errors.

To correlate the events in a pattern without using any session IDs, the approach analyzes the *fields* in the *body* of the RPC messages. The key idea is that, even if there is no common field to all events in a session or a trace, a subset of the sub-requests in the request flow may have some fields in common, such as the ID of a virtual resource (e.g., a volume, an instance), the tenant name, etc. Since the manual analysis of the body of the RPCs requires domain-specific knowledge of the system internals (Sharma et al., 2015), the approach keeps this operation lightweight by using an algorithm that analyzes all the events within a set of fault-free traces. More specifically, the algorithm analyzes all the fields f in the body of the RPC events collected in a set of fault-free traces and returns only the fields (*fields of interest*) that satisfy the following properties:

- P_1: *In every fault-free trace, the propagation of the values assumed by a field f should be higher than a threshold, say ϵ_1.* This property expresses that, in order to correlate events with a generic field f, then the values assumed by the field should ideally propagate across the fields of different events in the fault-free trace.
- P_2: *In every fault-free trace, the number of non-unique values assumed by a field f should be higher than a threshold, say ϵ_2.* Indeed, if a value assumed by a generic field f is repeated across all the events, then the property P_1 would correlate all the events of the trace in a single pattern. Therefore, this property ensures that the field f assumes different values across the uncorrelated events.

To assess how the choice of the thresholds ϵ_1 and ϵ_2 impacts the number of discarded fields in the body of the RPCs, we performed a sensitivity analysis by varying the thresholds between 0% and 100%, as shown in Fig. 3. The analysis highlights that the property P_1 is always satisfied also when the threshold ϵ_1 is high. In particular, almost the total number of the fields f propagate their values across the 90% of remaining events in the trace. However, most of the propagation of these values is due to fields that always assume a single value across almost all the events in a trace. This is the case for fields such as the OpenStack project ID (i.e., organizational units in the cloud to which administrators can assign tenants (OpenStack, 2022c)), which assume the same

Table 1
Correlation of the events in a pattern. By looking at the context request field, the approach finds two different patterns of 2 (**red color**) and 3 events (**blue color**), respectively. The two patterns are merged into a single one of 5 events since the values of the context request of the first pattern propagate across the context global request field of the second pattern.

	Timestamp	Message Sender	Service API	Status Code (REST API)	...	context global request	...	context request
	15...54199750	conductor	object action		...	None	...	req-961f14...
	...	...	...	...	...	...	...	...
	15...93483930	novaclient	DELETE	204	...		...	
ΔT	15...93548480	consoleauth	delete tokens for instance		...	None	...	req-e12682...
ΔT	15...93874240	compute	terminate instance		...	None	...	req-e12682...
	15...93898260	neutronclient	GET	200	...		...	
	15...94515930	neutronclient	DELETE	204	...		...	
ΔT	15...94538750	compute	external instance event		...	None	...	req-d59930...
ΔT	15...94899000	dhcp agent	port delete end		...	req-e12682...	...	req-0784d9...
ΔT	15...95111120	q-agent-notifier-security group-update	security groups member updated		...	req-e12682...	...	req-0784d9...
ΔT	15...95353280	q-agent-notifier-port-delete	port delete		...	req-e12682...	...	req-0784d9...
	15...99345540	cinder-volume	terminate connection		...	req-c0def5...	...	req-a0ffe...
	...	...	...	...	...	...	...	...
	15...78797886	q-reports-plugin	report state		...	None	...	req-7cb6a3...

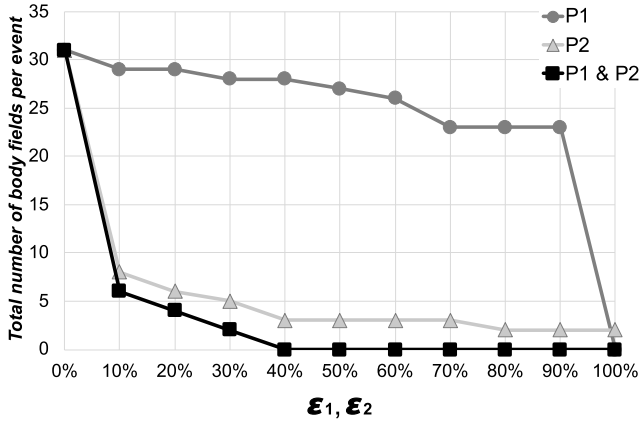

Fig. 3. Sensitivity analysis of the thresholds ϵ_1 and ϵ_2.

values across all the events in the system executions. Indeed, the figure shows that only less than one-third of the total number of the fields f assumes non-unique values for at least the 10% of times in the trace (P_2). Combining both properties, we obtain that all the fields are discarded when both thresholds are $\geq$40%. Therefore, to limit the number of fields of interest, we made a conservative choice of both thresholds, by setting them equal to 30%.

Once defined the properties, the algorithm filters the fields that do not satisfy both of them, working accordingly to the following steps:

1. Define the empty sets $F = \{\}$, $\Phi = \{\}$;
2. Extract all fields $[f_1, f_2, \ldots, f_N]$ from the body of all events in the traces and add them to the set F;
3. For every field f_i, with $i \in [1, N]$, if the field does not satisfy both properties P_1 and P_2, then $f_i \cup \Phi$;
4. Return $F \setminus \Phi$.

The application of the algorithm massively reduces the number of fields that can be analyzed manually. In our case study, from many fields in the body of the RPC messages (see Section 7.1), the algorithm returns the parameters used in the `oslo_context.context` of *Oslo Context* library, a base class for holding contextual information of a request (OpenStack, 2022d). More specifically, the algorithm returns the variable `_context_request`, i.e., the identifier of a request, and the variable `_context_global_request`, i.e., a request-id sent from another service to indicate that the event is part of a chain of requests (OpenStack, 2022d). The values of these variables are hexadecimal strings prefixed by "req-" (Sharma et al., 2015).

The approach correlates the events with the same values in fields `_context_request` and `_context_global_request` of the body of the RPC messages. However, since correlated events may occur too far in time (the workload execution may last tens of minutes or even hours), the approach defines a max time length of the pattern, that is, the temporal distance between the last event and the first event (i.e., the head event) in the chain of correlated events is lower than the length of a *time window* ΔT. Table 1 shows an example of the correlation process. Each event in the trace has several attributes, such as the timestamp, the sender, the service API, the fields of the body, etc. When, in a specified time window ΔT, the approach finds events having the same values in the field `_context_global_request` or in the field `_context_request` of an event that occurred before, then it defines a pattern of events in the fault-free trace. In the example of the table, the approach defines a pattern containing the events `<consoleauth, delete tokens for instance>` and `<compute, terminate instance>` because they have the same `_context_request`. Then, it extends the pattern with three further events having the `_context_global_request` equal to the `_context_request` of the two events in the pattern. The final pattern contains 5 events. Notice that no more events can be added to this pattern since the pattern length cannot be higher than the time window.

When a pattern of – not necessarily consecutive – events is repeated over all the fault-free executions, the approach defines a monitoring rule. For example, if the 5 events of the pattern shown in the example of Table 1 occur (regardless of the order) in all the fault-free executions of the system, then a new rule is added to the set of the monitoring rules. Therefore, suppose to

have m fault-free executions, and consider $p_{A,1}, p_{A,2}, \ldots, p_{A,m}$ as the patterns starting with the same head event type A found by the approach in the different executions. To define MR_A, i.e., the monitoring rule activated by the event type A, the approach finds all the common events in the different patterns:

$$MR_A = p_{A,1} \cap p_{A,2} \cap \ldots \cap p_{A,m} \quad (3)$$

MR_A will contain at least one event $a \in A$ because all the patterns start with the same head event. To check if the system's behavior is the desired one, we need at least one event following the head event used to activate the rule. Therefore, if at least one of the subsequent events is common to all the patterns, i.e., if $|MR_A| \geq 2$, then the approach defines a new monitoring rule.

5.2. Rules classification

Since the events in a pattern may occur in a different order, we classified the rules according to three different categories, explained in the following. Suppose to observe, in a specified time window, three different RPC events, say a, b, c belonging to three different event types, say A, B, C, respectively, and 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*** (ORD): Rules based on a flow of events that always follows the same order and occurrence. For example, the event b and c follow a always with the same pattern (e.g., $a \rightarrow b \rightarrow c$). Therefore, when the event a occurs, then the approach waits for the occurrence of the event b and the event c, with c following b, in a specified time window ΔT. These rules characterize the services less affected by the non-determinism and where it is possible to find a fixed pattern for the same operation. The ORD rules can detect failures causing out-of-order or missing events during the system execution.
- ***Occurred-Events*** (OCC): Rules based on a flow of events that occur after the head event without following any specific order and/or number. For example, this happens when the event type b occurs before or after event type c (e.g., $a \rightarrow b \rightarrow c$ or $a \rightarrow c \rightarrow b$). Therefore, when the event a occurs, then the approach waits for the occurrence of the event b and the event c, without a fixed ordering, in a specified time window ΔT. These rules take into account the non-determinism of the flow of events, i.e., we cannot identify a fixed pattern among all the system executions. The OCC rules can detect failures causing missing events in the pattern, but not out-of-order events.
- ***Counted-Events*** (COUNT): Rules based on the observation that an event (or more events) is repeated several times, varying in a range of value (e.g., $min_{count} < a < max_{count}$, where min_{count} and max_{count} represent the minimum and the maximum number of times the event is repeated under fault-free conditions, respectively). The COUNT rules can detect failures when the system is unable to serve a request involving multiple-repeated operations, such as polling requests on a resource. In this case, a failure leads to an anomalous repetition of events (i.e., $a > max_{count}$) since the requests are issued multiple times.

6. Monitor implementation

After identifying the monitoring rules, we synthesize the rules in a run-time monitor that verifies whether the system's behavior follows the desired one. Any run-time 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.

6.1. Implementation

We translate the rules in the *Event Processing Language* (EPL), a particular specification language provided by the *Esper* software (EsperTech, 2022), and allow the expression of different types of rules (i.e., temporal, statistical, etc.). The EPL extends the SQL standard language, offering both typical SQL clauses (e.g., `select`, `from`, `where`, `insert into`) and additional clauses for event processing (e.g., `pattern`, `output`). The *Esper compiler* compiles EPL source code into Java Virtual Machine (JVM) bytecode so that the resulting executable code runs on a JVM within the *Esper runtime* environment.

We applied the EPL statements derived from the monitoring rules to detect failures in OpenStack when multiple tenants perform requests concurrently. Since we do not collect a tenant ID, we use a *counter* to take into account multi-tenancy operations. To estimate the number of concurrent requests performed by different tenants, we associate a counter to every monitoring rule and increment its value every time we observe the head event. For example, if we observe twice the event type `<nova-conductor, schedule_and_build_instances>` (i.e., the head event of the request flow related to the instance creation) in the same time window, then we activate twice the monitoring rule since two different tenants are requesting the creation of an instance. The value of the counter is sent, along with the event name, to the *Esper runtime* component. Listing 1 shows the EPL translation of the rule *Volume Creation*.

Listing 1: OpenStack Volume Creation rule in EPL

```
@name('VolumeCreation') 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))];
```

When the *Esper runtime* observes the (head) event `<cinder-scheduler, create_volume>` with its counter value, it waits for the event `<cinder-volume, create_volume>` with the same counter value in a time window of `secondsToWait` seconds. If this condition is not verified, the approach notifies of a failure.

To express the monitoring rule, we used the clause `pattern` (to define a pattern of events), and the operators `every`, `followed-by` (→), and `timer:interval`. The operator `every` defines that every time we observe part of the pattern (e.g., the observation of the head event `cinder-scheduler, create_volume` in Listing 1), the *Esper runtime* actives a monitoring rule. Without this operator, the monitoring rule would be activated only once. The operator → defines the order of the events in the rule, while the operator `timer:interval` establishes the length of the time window.

The synthesis of the monitor 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 flow of events. We invite the reader to refer to the official documentation for more detailed information on Esper (Espertech, 2022).

6.2. Events collection

To collect the events exchanged in the system, we adopt the *Zipkin* distributed tracing system (Zipkin, 2022), due to its maturity, high performance, and support for several programming languages (Salesforce Engineering, 2022; Twitter Engineering, 2022). We instrument APIs to send data via HTTP to the *Zipkin collector*, which stores trace data. The collected events are ordered according to a timestamp given by the collector. In this work, we instrumented the following communication points in OpenStack:

- 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 subsystems.
- The *RESTful API libraries* of each OpenStack subsystem, i.e., *novaclient* for Nova (implements the OpenStack Compute API (OpenStack, 2022e)), *neutronclient* for Neutron (implements the OpenStack Network API (OpenStack, 2022f)), and *cinderclient* for Cinder (implements the OpenStack Block Storage API (OpenStack, 2022g)). 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 since it adopts an asynchronous collection mechanism to avoid critical execution paths. Moreover, we only instrument 5 selected lines of communication system code (e.g., the `cast` method of OSLO to broadcast messages), by adding simple annotations (the Zipkin context manager/decorator) only at the beginning of these methods (a total of 21 lines of Python code). Our instrumentation neither modified the internals of OpenStack subsystems nor used any domain knowledge.

We extract periodically the events stored in the *Zipkin collector* and information such as the invoked method, the service providing the API, the timestamp, the body of the RPC messages, 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 against every statement compiled by the *Esper compiler* (i.e., the monitoring rules): if that event satisfies the condition specified in a rule, then the rule moves to the next condition, otherwise, it raises an exception, notifying an unexpected behavior.

7. Experimental evaluation

We evaluate the proposed approach by performing fault injection experiments against the OpenStack cloud management platform, which is a relevant case study since it is a large and complex distributed system.

7.1. Setup

We targeted OpenStack version 3.12.1 (release *Pike*), deployed on Intel Xeon servers (E5-2630L v3 @ 1.80 GHz) with 16 GB RAM, 150 GB of disk storage, and Linux CentOS v7.0, connected through a Gigabit Ethernet LAN.

To evaluate the approach in realistic scenarios, we developed a multi-tenant workload generator, which simulates 10 different tenants performing concurrent operations on the cloud infrastructure. The tenants exhibit 6 different profiles, as described in the following:

- ***Volume Only***: The tenant performs operations strictly related to the block storage (Cinder subsystem);
- ***Instance Only***: The tenant stresses the Nova subsystem for the creation of VM instances;
- ***Network Only***: The tenant creates network resources (networks, sub-networks, IP addresses, routers, etc.), stressing the Neutron subsystem;
- ***Instance before Volume***: The tenant creates an instance from an image, then a storage volume;
- ***Volume before Instance***: The tenant creates a volume and then an instance starting from the volume;
- ***Instance, Volume, and Network***: The tenant stresses the Nova, Cinder, and Neutron subsystems in a balanced way.

Table 2
Monitoring Rules in the fault-free scenario.

Rule description	Rule type	# of events	Subsystems
Instance creation	ORD	4	Nova
Volume creation	ORD	2	Cinder
Network creation	OCC	3	Neutron
Volume attachment	ORD	4	Nova, Cinder
Instance deletion	ORD	3	Nova
Security group update	ORD	2	Neutron
Ping instance via SSH	COUNT	6–26	Neutron

These six profiles are run concurrently to generate a multi-tenant workload. The *Volume Only*, *Network Only*, *Instance before Volume*, and *Volume before Instance* profiles are run twice by different tenants.

The execution of the workload lasts ∼40 min and produces a large amount of data. On average, during every fault-free execution, we collected 89 different event types (75 RPC and 14 REST API), and ∼2400 different events (∼2050 RPC events, while the remaining are related to the REST API calls). For every execution trace, the bodies of all RPC events contain in total more than 60,000 fields (on average, ∼31 body fields per RPC event).

7.2. Fault-free analysis

Since the execution of the system execution is not trivial and time-consuming, we limited the number of fault-free traces. Indeed, we collected 50 fault-free traces, exercising the system with the multi-tenant workload. To set the time window ΔT and specify the length of the patterns, we made a conservative choice by setting it equal to the maximum time needed by OpenStack to serve any request performed by the multi-tenant workload in fault-free conditions (∼35 s, in our testbed). Although this duration depends on both the workload, i.e., the operations performed during the experiments, and the hardware where OpenStack is deployed (high hardware requirements imply a shorter time to serve the requests), it can be easily computed by running the workload in fault-free conditions (e.g., by using the logs of the workload).

We derived 7 types of monitoring rules based on RPC messages, as shown in Table 2. The rules include the creation of resources, such as instances, volumes, and networks, which are common operations on an IaaS cloud. 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. We attribute this to the asynchronous nature of the Neutron subsystem. The approach also identified the rule for the attachment of the volume to an instance and the deletion of the instance. Moreover, the approach derived two further rules related to the network operations: the update of the security groups (the sets of network filter rules that are applied to all instances, e.g., allowed/disallowed SSH traffic, etc.) that define networking access to the instance, and the connection to an instance via SSH, which is the only rule of type COUNT.

We notice that the monitoring rules inferred by our approach do not encompass all possible operations performed by the workload. Indeed, *volume deletion* and *instance reboot* are notable examples of operations not included in Table 2. We investigated the fault-free traces and observed that these operations do not involve a sequence of events, but only a single head event. However, to monitor the system, we need a pattern of at least two events, i.e., at least one event has to follow the head event activating the rule in a temporal window.

7.3. Fault injection experiments

We evaluated our approach by performing a fault-injection campaign in OpenStack. In total, we performed 637 experiments by injecting faults in Nova, Cinder, and Neutron subsystems (one fault per experiment). To perform the experiments, we developed a tool (Cotroneo et al., 2020b, 2019b) to automatically scan the source code of OpenStack, find all injectable API calls, and inject faults by mutating the calls. The tool identifies the injectable locations that are actually covered by the running workload and performs one fault injection test per covered location. To define the faults to inject into the target system, we analyzed over 179 problem reports on the OpenStack bug repository. This analysis allowed us to identify the most recurrent bugs in OpenStack over the last few years. In particular, we choose the following faults, which are among the most frequent in OpenStack (Cotroneo et al., 2019c):

- ***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 redeploy the cloud management system, remove all temporary files and processes, and restore the OpenStack database to its initial state.

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 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 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 in identifying the cases where a fault causes an error and the system does not generate an API error (i.e., the system is unaware of the failure state) (Cotroneo et al., 2019c).

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. In total, we observed failures in 496 experiments (∼78% of the total number of experiments). 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 (Christmansson and Chillarege, 1996; Lanzaro et al., 2014); yet, the experiments provided us with a large and diverse set of failures for our analysis.

In many failures, when the tenant performs a request by using the REST APIs of the system, the events related to these calls contain a status code 4*xx* or 5*xx*, indicating the incapability of the client/server to perform/serve the request. These events cannot be observed during fault-free executions since they reflect failure symptoms. In these cases, the flow of RPC events starting from the REST API call does not occur, making the RPC events-based rules not effective in detecting anomalies. Therefore, to support the monitoring rules, the approach notifies the failure when we observe a REST API call with status code 4*xx* or 5*xx*.

To help the research community in the application and evaluation of new solutions for detecting failures in the system, we shared on GitHub[1] the raw logs collected during the execution of the OpenStack cloud computing platform with the multi-tenant workload, by including both the fault-injection experiments and the fault-free executions of the system. The repository also contains the Python code we used to analyze the execution traces and infer the monitoring rules, and the scripts for the multi-tenant workload in the OpenStack cloud computing platform.

7.4. Evaluation metrics

We evaluated the approach in terms of *precision* and *recall*. The former is mathematically computed as the number of *true positives* identified by the approach over the total number of positives predicted (*true* and *false positives*). The latter, instead, is computed as the number of *true positives* identified by the approach over the total number of actual positives (*true positives* and *false negatives*). We consider failure detection as a true positive case only when the approach detects the "first" failure of the system. For example, if OpenStack fails to create an instance and the approach detects the failure only on the subsequent attachment of a volume to the failed instance, we consider the experiment as a *false negative* case since the first failure experienced by the system (i.e., the instance creation) was undetected. This conservative choice is due to the need to detect failures as soon as they occur in the system and avoid error propagation. The *false positives* cases, instead, refer to the experiments in which the approach identifies a failure before the actual failure of the system or when the system is not failed at all. To perform a comprehensive evaluation, we use the F_1 *score*, defined as the harmonic mean of the precision and recall. Moreover, to assess also the *false negatives*, we adopted the *accuracy* as a further metric, which is the ratio of correct detections (i.e., both *true positives* and *true negatives*, where the latter refers to the cases in which the approach does not notifies any failure and the system is actually not failed) over the total number of experiments. All metrics range from 0 (total misclassification) to 1 (perfect classification).

7.5. Experimental results

To provide context for the evaluation, we compared the proposed approach (**Monitoring Rules — MR**) against three baseline approaches:

- ***OpenStack Failure Logging Mechanisms*** (FL): The OpenStack built-in failure logging mechanisms, which notify the tenants via API errors if the system is not able to serve requests;
- ***Non-session-aware approach using unseen n-grams*** (UN): A non-session-aware approach based on unseen *n*-grams (similar to An et al. (2017)), where the *n*-gram represents a contiguous sequence of *n* events within a trace. The approach learns a *normal dictionary* consisting of all occurring *n*-grams from the fault-free traces. During the detection, the approach notifies a failure when a new *n*-gram that is not in the dictionary occurs;
- ***Non-session-aware approach using a probabilistic model*** (PM): A non-session-aware probabilistic approach using *Variable-order Markov Models* (Begleiter et al., 2004). The approach is trained on the sequence of the events that

[1] https://github.com/dessertlab/OpenStack-multi-tenant-workload.

Table 3
Approaches comparison. The best performance is **bold**. The worst performance is **red/bold**. Time window $\Delta T = 35$ s (MR approach).

Openstack subsystem	Approach	Precision	Recall	F_1 score	Accuracy
Nova	*FL*	**1.00**	0.30	0.46	0.47
	UN	0.26	**0.11**	0.16	0.09
	PM	**0.05**	0.30	**0.08**	**0.04**
	MR	0.89	**1.00**	**0.94**	**0.91**
	FL with MR	0.89	**1.00**	**0.94**	**0.91**
Cinder	*FL*	**1.00**	0.28	0.44	0.38
	UN	**0.11**	**0.02**	**0.03**	**0.02**
	PM	0.13	0.41	0.20	0.11
	MR	0.85	0.84	**0.85**	**0.74**
	FL with MR	0.85	**0.85**	**0.85**	**0.74**
Neutron	*FL*	**1.00**	0.71	0.83	0.80
	UN	**0.12**	**0.06**	**0.08**	**0.04**
	PM	0.13	0.32	0.18	0.10
	MR	0.87	0.31	0.46	0.50
	FL with MR	0.95	**0.92**	**0.93**	**0.91**
All subsystems	*FL*	**1.00**	0.36	0.53	0.50
	UN	0.19	**0.07**	**0.10**	**0.05**
	PM	**0.09**	0.35	0.14	0.08
	MR	0.87	0.82	0.85	0.77
	FL with MR	0.88	**0.93**	**0.91**	**0.85**

occurred in the system under fault-free conditions. The approach notifies a failure whenever the probability of an event occurring after a sequence of events is lower than a fixed threshold ϵ_{PM}.

Moreover, to estimate the improvement obtained by implementing an external monitoring solution to support OpenStack, we evaluated also the performance of the ***OpenStack failure logging mechanisms combined with the MR approach*** (FL with MR). All the baseline approaches can perform failure detection at run-time. To perform a fair comparison with the MR approach and highlight the challenges of detecting failures without using IDs, we limited the evaluation to non-session-aware approaches. Moreover, we did not include more complex approaches due to the limited number of fault-free traces (i.e., 50) used for training. In fact, run-time monitoring solutions based on deep learning neural networks require a huge amount of data to detect anomalies in large-scale cloud computing platforms (Islam et al., 2021).

For each OpenStack subsystem targeted during the fault injection campaign, Table 3 shows the results obtained in terms of precision, recall, F_1 score, and accuracy. To perform a fair evaluation, we conducted a sensitivity analysis for the UN and PM approaches by varying the number of n-grams between 1 and 5 (Whalen et al., 2014; Shin and Kim, 2020) and the threshold ϵ_{PM} between 0% and 100%. The table shows only the results obtained with the best configuration of the baseline approaches (for UN, set $n = 3$; for PM, set $\epsilon_{PM} = 1\%$).

The results highlight that the FL approach provides perfect precision over all the subsystems because the logging mechanisms of the system are not affected by false positive cases (i.e., an API error always implies a system failure). However, the recall provided by this approach is dramatically low (0.36 over all the subsystems) since OpenStack is not able to timely detect and notify the failures in many experiments due to missing logging mechanisms, as shown in previous studies (Cotroneo et al., 2019c; Marques et al., 2022).

Different from the system's logging mechanisms, the precision provided by the MR approach is not perfect because, in some cases, the approach wrongly identified a failure due to the non-determinism of the system. Nevertheless, the precision achieved is still very close to the one provided by the FL approach. The considerations on the false-negative cases, instead, are way different. Indeed, the results highlight how the MR approach can effectively bring a substantial improvement in the recall values over all the subsystems since the monitoring rules are able to catch the out-of-order or missing events in most of the cases (recall equal to 0.82). The approach provides worse performance only for the Neutron subsystem. We attribute this to the asynchronous nature of the network service, causing either a missing detection or a missing activation of the rules.

The results also show that the precision and the recall provided by both the other two non-session-aware baseline approaches, i.e., the approach based on the observation of the unseen n-grams (UN approach) and the probabilistic model approach (PM), are not comparable to one of the other approaches over all the metrics, regardless of the target subsystem. Indeed, both approaches are not able to fully model the behavior of a multi-tenant and concurrent system without discerning the calls executed by different tenants (using session IDs), given the massive number of events per trace.

The F_1 score and the accuracy allow us to compare the approaches both in terms of false positives and false negatives, and thus provides a comprehensive evaluation of the approaches. The metrics have similar values over all the subsystems and suggest that, for the fault injection experiments, the MR approach massively improves the performance obtained with the plain OpenStack logging mechanisms (85% vs. 53% for the F_1, 77% vs. 50% for the accuracy). In particular, the proposed approach achieves an F_1 score and accuracy higher for Nova and Cinder subsystems, while the performance is again worse for the Neutron subsystem. The UN and the PM approaches are very far from the performance of the MR approach as they provide a F_1 score equal to 0.10 and 0.14, respectively. The accuracy is even worse (0.05 for UN, 0.08 for PM) since both approaches are not able to provide true negative cases (i.e., they always identify a failure also when the system is not failed at all). This further emphasizes the results obtained by the MR approach without using any session IDs.

Finally, when the monitoring rules are used in combination with the OpenStack logging mechanisms (FL with MR approach), we can notice that, although the rules slightly impact the precision of the system by wrongly notifying a failure due to the non-determinism of the system, they massively reduced the false-negative cases, overall the subsystems. Even for the Neutron subsystem, when the recall for the proposed approach is lower than OpenStack logging mechanisms, the FL with MR approach takes advantage of the monitoring rules since they help OpenStack to notify failures not detected by the logs of the system.

7.6. Detection latency

To provide a more comprehensive evaluation, we also analyzed the promptness of the targeted approaches in the identification of failures. Ideally, a failure should be identified as soon as the system experiences it to quickly restore services and thus increase the reliability of the system. Therefore, we performed a comparison in terms of *failure detection latency* by computing the time difference between the failure detection time (t_{fail}) and a *common starting time*. Given the high non-determinism of the target system, it is not trivial to identify a reliable fault activation time (Avižienis et al., 2004; Avizienis et al., 2004), especially in a multi-tenant scenario. To overcome this issue, we used the starting time of the workload (t_{start}) as the *common starting time*, thus the failure detection latency is equal to $t_{fail} - t_{start}$. Because the approaches are compared over the same experiments and under the same conditions, a shorter failure detection latency indicates the ability to quickly detect failures. To perform a fair comparison, we did not include the false positive cases in this analysis.

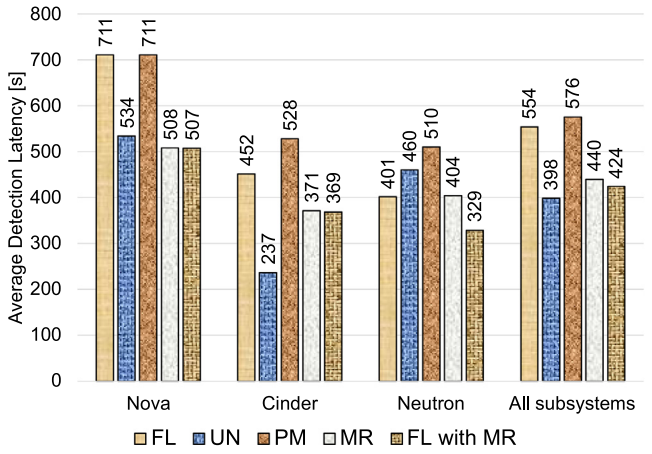

Fig. 4. Average detection latency of the different approaches. Time window $\Delta T = 35$ s (MR approach).

Table 4
Sensitivity analysis of the time window ΔT across all subsystems. The best performance is **bold**. Worst performance is **red/bold**.

	ΔT (s)			
Metric	5	20	35	50
Precision	**0.73**	0.83	0.87	**0.89**
Recall	**0.77**	**0.82**	**0.82**	0.81
F_1 score	**0.75**	0.83	**0.85**	**0.85**
Accuracy	**0.60**	0.74	**0.77**	**0.77**
Detection latency (s)	**366.26**	423.63	439.80	**457.81**

Fig. 4 shows the average failure detection latency (in s) provided by the approaches over all fault injection experiments. The figure shows that the MR approach provides a notably lower failure detection latency when compared to the FL approach for Nova and Cinder subsystems, and a comparable detection latency for the Neutron subsystem. Overall the fault injection experiments, the average detection latency of MR is ∼114 s lower than the average detection latency of FL. The failure detection latency of the FL with MR 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 run-time. Also for the Neutron subsystem, where the MR approach showed the worst performance due to the asynchronous nature of the network operations, the FL with MR approach notably decreases the average failure detection latency with respect to the FL approach (∼77 s). Finally, it is worth noticing that the UN approach, which provides the worst performance in terms of accuracy in detecting the failures, shows the lowest failure detection latency across all subsystems, as it raises an exception every time an unseen sequence of n-grams is observed. The probabilistic approach (PM), instead, is not able to provide timely detection of the failures due to the threshold value, which is set very low (i.e., 1%) to limit as many as possible false positive cases. Indeed, since the approach identifies a failure only when the occurrence of an event is very unlikely, then the average time to identify the failure increased.

7.7. Sensitivity analysis

In the previous analysis, we adopted a conservative value for the time window by setting it equal to the maximum time needed by OpenStack to serve any tenant's request in our setup. Since the choice of the time window influences the length of the patterns, we performed a sensitivity analysis. Table 4 shows the results of the MR approach by setting the time window ΔT equal to 5, 20, 35, and 50 s.

Unsurprisingly, we found that the performance of the approach improves by increasing the time window since a shorter ΔT increases the number of false positives and limits the true negatives. As matter of fact, we found that when ΔT is equal to 5 s, the approach provides a *false positive rate* equal to 1. The table also shows that, although the increment of the time window implies an improvement of the precision, the recall saturates when ΔT is equal to 20 s, and slightly decreases when the time window is set to 50 s since a pattern too temporally long is affected by false-negative cases. Therefore, both the F_1 score and the accuracy of the MR approach saturates when ΔT is higher than 35 s. Moreover, as expected, the table shows that a larger time window implies an increment in failure detection latency. Since the choice of the time window should be a valid trade-off between the ability to detect failures and the failure detection latency, a time window equal to 35 s (or also 20 s) is considered a more proper choice.

7.8. Computational cost

We performed the analysis (on the same system used for the experiments) of the computational cost required to derive the monitoring rules from fault-free traces. The computational cost includes the time needed to parse the logs, filter events, and run the algorithm to find the patterns. We found that the overall time needed to simultaneously analyze 50 different fault-free execution traces (which contain ∼120K rows) is lower than 70 s (i.e., less than 1.5 seconds per trace, on average). The computational cost increases linearly with the number of traces.

8. Threats to validity

Case Study. OpenStack is one of the most widely deployed open-source cloud software in the world and represents an important case study. The execution of fault-injection experiments on OpenStack is non-trivial because, to guarantee independence among the experiments, we restart all the services, restore the database, clean up the files, execute the workload, etc (every experiment lasts 1 h on average). Therefore, the application of the approach and the execution of the fault-injection experiments also on other cloud computing platforms is very time-consuming. To mitigate this threat to validity, we targeted three systems from the OpenStack umbrella project (i.e., Nova, Neutron, Cinder), which are large and diverse enough to get interesting insights into the application of the proposed approach across different independent systems. This diversity is reflected by differences both in terms of project-specific patterns (programming idioms, API conventions), and different events exchanged in the systems (number, type, and non-determinism).

Baseline approaches. To perform a fair evaluation, we compared the results of the MR approach with two non-session-aware approaches. The choice of the baseline approaches also depended on the public availability of open-source code to reproduce the experiments and the portability of the approaches on a different system such as OpenStack. We adopted an n-gram-based solution and a probabilistic model approach because sequence analysis approaches are widely used in practice. As matter of fact, there are several research studies adopting n-grams (or similar algorithms) and probabilistic models to perform run-time detection of anomalies up to now (Khreich et al., 2017; Ariff et al., 2021; Brown et al., 2022; Cailliau and Lamsweerde, 2019; Carreon et al., 2021; Bartolo Burlò et al., 2021). Moreover, we did not include more complex approaches, such as neural networks-based approaches, since they require massive data for training (Islam et al., 2021; Huch et al., 2018; Girish and Rao, 2021). Indeed, the

collection of the system executions is an essential concern in our context as developers have a limited time budget to spend for fault-injection testing. Since executions can take several hours in commercial-grade systems, we need to limit the number of fault-free executions (as matter of fact, we collected only 50 executions).

9. Conclusion

In this work, we proposed an approach to run-time verification in cloud computing systems that derives a set of monitoring rules from the fault-free executions of the system. The rules are then synthesized in a monitor solution by using a specification language. We applied the approach in the OpenStack cloud computing platform, where we evaluated the ability of the monitoring rules in detecting failures in a campaign of fault injection experiments with a multi-tenant workload. Our experiments showed that the approach achieves better performance, in terms of F_1 score, when compared to the OpenStack logging mechanisms and two non-session-aware run-time verification approaches, and significantly decreases the time to detect the failure at run-time. The approach, when used in combination with the failure logging mechanisms of the system, provides an F_1 score higher than 90% and accuracy of 85%, improving the fault tolerance mechanisms of the system.

The MR approach can be applied in different OpenStack configurations as it does not depend on any configuration parameter, except the size of the time window (i.e., the time duration of the patterns). In our experiments, we made a conservative choice by setting the size equal to the maximum time taken by OpenStack to serve a request in fault-free conditions (35 s, in our testbed). The sensitivity analysis of the time window showed that this conservative choice provides the best performance in terms of F_1 score and accuracy. Although the maximum time to serve the requests depends on both the workload, i.e., the operations performed during the experiments, and the hardware where OpenStack is deployed (high hardware requirements imply a shorter time to serve the requests), it can be easily computed by running the workload in fault-free conditions (e.g., by using the logs of the workload).

CRediT authorship contribution statement

Domenico Cotroneo: Resources, Writing – review & editing, Supervision, Project administration, Funding acquisition. **Luigi De Simone:** Software, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Pietro Liguori:** Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Visualization. **Roberto Natella:** Conceptualization, Methodology, Resources, Writing – original draft, Writing – review & editing, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We shared on https://github.com/dessertlab/OpenStack-multi-tenant-workload the code, the multi-tenant workload and raw logs collected during the execution of the workload on OpenStack.

Acknowledgments

This work has been partially supported by the University of Naples Federico II, Italy in the frame of the Programme F.R.A., project id OSTAGE. We are grateful to our former students Nicola Apa and Roberto Scarpati for their help in the early stage of this work.

References

Aguilera, M.K., Mogul, J.C., Wiener, J.L., Reynolds, P., Muthitacharoen, A., 2003. Performance debugging for distributed systems of black boxes. Oper. Syst. Rev. 37 (5), 74–89.

Alpernas, K., Panda, A., Ryzhyk, L., Sagiv, M., 2021. Cloud-scale runtime verification of serverless applications. In: Proceedings of the ACM Symposium on Cloud Computing. pp. 92–107.

Ammar, H.H., Cukic, B., Mili, A., Fuhrman, C., 2000. A comparative analysis of hardware and software fault tolerance: Impact on software reliability engineering. Ann. Softw. Eng. 10 (1), 103–150.

An, N., Duff, A., Naik, G., Faloutsos, M., Weber, S., Mancoridis, S., 2017. Behavioral anomaly detection of malware on home routers. In: 2017 12th International Conference on Malicious and Unwanted Software (MALWARE). IEEE, pp. 47–54.

Ariff, N.A.M., Mas' ud, M.Z., Bahaman, N., Hamid, E., Anuar, N.A., 2021. Ensemble method for mobile malware detection using N-gram sequences of system calls. Int. J. Commun. Netw. Inf. Secur. 13 (2), 236–241.

Arlat, J., Fabre, J.-C., Rodríguez, M., 2002. Dependability of COTS microkernel-based systems. IEEE Trans. Comput. 51 (2), 138–163.

Atlidakis, V., Godefroid, P., Polishchuk, M., 2020. Checking security properties of cloud service REST APIs. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). IEEE, pp. 387–397.

Avižienis, A., Laprie, J.-C., Randell, B., 2004. Dependability and its threats: a taxonomy. In: Building the Information Society. Springer, pp. 91–120.

Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. 1 (1), 11–33.

Bahl, P., Han, R.Y., Li, L.E., Satyanarayanan, M., 2012. Advancing the state of mobile cloud computing. In: Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services. pp. 21–28.

Barham, P., Isaacs, R., Mortier, R., Narayanan, D., 2003. Magpie: Online modelling and performance-aware systems. In: HotOS. pp. 85–90.

Bartocci, E., Falcone, Y., Francalanza, A., Reger, G., 2018. Introduction to runtime verification. In: Lectures on Runtime Verification. Springer, pp. 1–33.

Bartolo Burlò, C., Francalanza, A., Scalas, A., Trubiani, C., Tuosto, E., 2021. Towards probabilistic session-type monitoring. In: International Conference on Coordination Languages and Models. Springer, pp. 106–120.

Begleiter, R., El-Yaniv, R., Yona, G., 2004. On prediction using variable order Markov models. J. Artificial Intelligence Res. 22, 385–421.

Beschastnikh, I., Wang, P., Brun, Y., Ernst, M.D., 2016. Debugging distributed systems. Commun. ACM 59 (8), 32–37.

Brown, P., Brown, A., Gupta, M., Abdelsalam, M., 2022. Online malware classification with system-wide system calls in cloud IaaS. arXiv preprint arXiv:2208.04891.

Cailliau, A., Lamsweerde, A.V., 2019. Runtime monitoring and resolution of probabilistic obstacles to system goals. ACM Trans. Auton. Adapt. Syst. (TAAS) 14 (1), 1–40.

Carreon, N.A., Lu, S., Lysecky, R., 2021. Probabilistic estimation of threat intrusion in embedded systems for runtime detection. ACM Trans. Embedd. Comput. Syst. (TECS) 20 (2), 1–27.

Chen, Y.-Y.M., 2004. Path-Based Failure and Evolution Management. University of California, Berkeley.

Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E., 2002. Pinpoint: Problem determination in large, dynamic internet services. In: Proceedings International Conference on Dependable Systems and Networks. IEEE, pp. 595–604.

Chen, F., Roşu, G., 2007. 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. pp. 569–588.

Chow, M., Meisner, D., Flinn, J., Peek, D., Wenisch, T.F., 2014. 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). pp. 217–231.

Christmansson, J., Chillarege, R., 1996. Generation of an error set that emulates software faults based on field data. In: Proceedings of Annual Symposium on Fault Tolerant Computing. IEEE, pp. 304–313.

Cotroneo, D., De Simone, L., Liguori, P., Natella, R., 2020a. Fault injection analytics: A novel approach to discover failure modes in cloud-computing systems. IEEE Trans. Dependable Secure Comput..

Cotroneo, D., De Simone, L., Liguori, P., Natella, R., 2020b. ProFIPy: Programmable software fault injection as-a-service. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, pp. 364–372.

Cotroneo, D., De Simone, L., Liguori, P., Natella, R., 2021. Enhancing the analysis of software failures in cloud computing systems with deep learning. J. Syst. Softw. 181, 111043.

Cotroneo, D., De Simone, L., Liguori, P., Natella, R., Bidokhti, N., 2019a. Enhancing failure propagation analysis in cloud computing systems. In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 139–150.

Cotroneo, D., De Simone, L., Liguori, P., Natella, R., Bidokhti, N., 2019b. Failviz: A tool for visualizing fault injection experiments in distributed systems. In: 2019 15th European Dependable Computing Conference (EDCC). IEEE, pp. 145–148.

Cotroneo, D., De Simone, L., Liguori, P., Natella, R., Bidokhti, N., 2019c. 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. pp. 200–211.

Cotroneo, D., De Simone, L., Natella, R., 2018. Run-time detection of protocol bugs in storage i/o device drivers. IEEE Trans. Reliab. 67 (3), 847–869.

d'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z., 2005. LOLA: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME'05). IEEE, pp. 166–174.

Delgado, N., Gates, A.Q., Roach, S., 2004. A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. Softw. Eng. 30 (12), 859–872.

Deligiannis, P., McCutchen, M., Thomson, P., Chen, S., Donaldson, A.F., Erickson, J., Huang, C., Lal, A., Mudduluru, R., Qadeer, S., et al., 2016. Uncovering bugs in distributed storage systems during testing (not in production!). In: 14th {*USENIX*} Conference on File and Storage Technologies ({*FAST*} 16). pp. 249–262.

Denton, J., 2015. Learning OpenStack Networking (Neutron). Packt Publishing Ltd.

Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C., 2007. The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69 (1–3), 35–45.

EsperTech, Home page of Esper, http://www.espertech.com/esper.

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

Farshchi, M., Schneider, J.-G., Weber, I., Grundy, J., 2015. 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). IEEE, pp. 24–34.

Garraghan, P., Yang, R., Wen, Z., Romanovsky, A., Xu, J., Buyya, R., Ranjan, R., 2018. Emergent failures: Rethinking cloud reliability at scale. IEEE Cloud Comput. 5 (5), 12–21.

Geels, D., Altekar, G., Maniatis, P., Roscoe, T., Stoica, I., 2007. Friday: Global comprehension for distributed replay. In: NSDI, Vol. 7. pp. 285–298.

Girish, L., Rao, S.K., 2021. Anomaly detection in cloud environment using artificial intelligence techniques. Computing 1–14.

Grant, S., Cech, H., Beschastnikh, I., 2018. Inferring and asserting distributed system invariants. In: Proceedings of the 40th International Conference on Software Engineering. pp. 1149–1159.

Gu, J., Wang, L., Yang, Y., Li, Y., 2018. Kerep: Experience in extracting knowledge on distributed system behavior through request execution path. In: 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, pp. 30–35.

Gunawi, H.S., Do, T., Joshi, P., Alvaro, P., Hellerstein, J.M., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Sen, K., Borthakur, D., 2011. FATE and DESTINI: A framework for cloud recovery testing. In: Proceedings of NSDI'11: 8th USENIX Symposium on Networked Systems Design and Implementation. p. 239.

Gunawi, H.S., Hao, M., Leesatapornwongsa, T., Patana-anake, T., Do, T., Adityatama, J., Eliazar, K.J., Laksono, A., Lukman, J.F., Martin, V., et al., 2014. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In: Proceedings of the ACM Symposium on Cloud Computing. pp. 1–14.

Gunawi, H.S., Hao, M., Suminto, R.O., Laksono, A., Satria, A.D., Adityatama, J., Eliazar, K.J., 2016. Why does the cloud stop computing? lessons from hundreds of service outages. In: Proceedings of the Seventh ACM Symposium on Cloud Computing. pp. 1–16.

Hole, K.J., Otterstad, C., 2019. Software systems with antifragility to downtime. Computer 52 (2), 23–31.

Horovits, D., 2022. Introduction to instrumentation with OpenTracing and Jaeger. https://logz.io/learn/opentracing-jaeger-guide-to-instrumentation/.

Horovitz, S., Arian, Y., Vaisbrot, M., Peretz, N., 2019. Non-intrusive cloud application transaction pattern discovery. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). IEEE, pp. 311–320.

Huch, F., Golagha, M., Petrovska, A., Krauss, A., 2018. Machine learning-based run-time anomaly detection in software systems: An industrial evaluation. In: 2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE). IEEE, pp. 13–18.

Islam, M.S., Pourmajidi, W., Zhang, L., Steinbacher, J., Erwin, T., Miranskyy, A., 2021. Anomaly detection in a large-scale cloud platform. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, pp. 150–159.

Khreich, W., Khosravifar, B., Hamou-Lhadj, A., Talhi, C., 2017. An anomaly detection system based on variable N-gram features and one-class SVM. Inf. Softw. Technol. 91, 186–197.

Koskinen, E., Jannotti, J., 2008. Borderpatrol: isolating events for black-box tracing. Oper. Syst. Rev. 42 (4), 191–203.

Krause, B., 2021. Design and implementation of a non-intrusive distributed tracing system for wireless embedded networks.

Lanzaro, A., Natella, R., Winter, S., Cotroneo, D., Suri, N., 2014. An empirical study of injected versus actual interface errors. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 397–408.

Las-Casas, P., Papakerashvili, G., Anand, V., Mace, J., 2019. Sifter: Scalable sampling for distributed traces, without feature engineering. In: Proceedings of the ACM Symposium on Cloud Computing. pp. 312–324.

Li, Z., Lu, Q., Zhu, L., Xu, X., Liu, Y., Zhang, W., 2018. An empirical study of cloud API issues. IEEE Cloud Comput. 5 (2), 58–72.

Li, B., Peng, X., Xiang, Q., Wang, H., Xie, T., Sun, J., Liu, X., 2022. Enjoy your observability: an industrial survey of microservice tracing and analysis. Empir. Softw. Eng. 27 (1), 1–28.

Mariani, L., Monni, C., Pezzé, M., Riganelli, O., Xin, R., 2018. Localizing faults in cloud systems. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp. 262–273.

Mariani, L., Pezzè, M., Riganelli, O., Xin, R., 2020. Predicting failures in multi-tier distributed systems. J. Syst. Softw. 161, 110464.

Marques, H., Laranjeiro, N., Bernardino, J., 2022. Injecting software faults in Python applications. Empir. Softw. Eng. 27 (1), 1–33.

Musavi, P., Adams, B., Khomh, F., 2016. Experience report: An empirical study of API failures in OpenStack cloud environments. In: 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 424–434.

Oliner, A., Ganapathi, A., Xu, W., 2012. Advances and challenges in log analysis. Commun. ACM 55 (2), 55–61.

OpenStack, Compute API, https://docs.openstack.org/api-ref/compute/?expanded=create-server-detail.

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

OpenStack, Manage projects, users, and roles, https://docs.openstack.org/keystone/pike/admin/cli-manage-projects-users-and-roles.html.

OpenStack, Oslo context library, https://docs.openstack.org/oslo.context/latest/index.html.

OpenStack, Compute API, https://developer.openstack.org/api-ref/compute.

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

OpenStack, Block storage API, https://developer.openstack.org/api-ref/block-storage.

Ou, Z., Song, M., Hwang, Z.-H., Ylä-Jääski, A., Wang, R., Cui, Y., Hui, P., 2018. Is cloud storage ready? Performance comparison of representative IP-based storage systems. J. Syst. Softw. 138, 206–221.

Parker, A., Spoonhower, D., Mace, J., Sigelman, B., Isaacs, R., 2020. Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices. O'Reilly Media.

Perrochon, L., 1998. Real time event based analysis of complex systems. Perrochon.Com.

Petrillo, F., Merle, P., Moha, N., Guéhéneuc, Y.-G., 2016. Are REST APIs for cloud computing well-designed? An exploratory study. In: International Conference on Service-Oriented Computing. Springer, pp. 157–170.

Power, A., Kotonya, G., 2019. Providing fault tolerance via complex event processing and machine learning for IoT systems. In: Proc. IoT. pp. 1–7.

Rabiser, R., Guinea, S., Vierhauser, M., Baresi, L., Grünbacher, P., 2017. A comparison framework for runtime monitoring approaches. J. Syst. Softw. 125, 309–321.

Reynolds, P., Killian, C.E., Wiener, J.L., Mogul, J.C., Shah, M.A., Vahdat, A., 2006. Pip: Detecting the unexpected in distributed systems. In: NSDI, Vol. 6. p. 9.

Salesforce Engineering, Anomaly detection in zipkin trace data, https://engineering.salesforce.com/anomaly-detection-in-zipkin-trace-data-87c8a2ded8a1.

Satyanarayanan, M., Steere, D.C., Kudo, M., Mashburn, H., 1992. Transparent logging as a technique for debugging complex distributed systems. In: Proceedings of the 5th Workshop on ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring. pp. 1–3.

Sharma, D., Poddar, R., Mahajan, K., Dhawan, M., Mann, V., 2015. Hansel: Diagnosing faults in openstack. In: Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies. pp. 1–13.

Shin, Y., Kim, K., 2020. Comparison of anomaly detection accuracy of host-based intrusion detection systems based on different machine learning algorithms. Int. J. Adv. Comput. Sci. Appl. 11, 252–259.
Solberg, M., Silverman, B., 2017. OpenStack for Architects. Packt Publishing Ltd.
Stackalytics, OpenStack pike lines of code, https://www.stackalytics.com/?release=pike&metric=loc.
Stackalytics, OpenStack pike commits, https://www.stackalytics.com/?release=pike&metric=commits.
Twitter Engineering, Distributed systems tracing with Zipkin, https://blog.twitter.com/engineering/en_us/a/2012/distributed-systems-tracing-with-zipkin.
van Hoorn, A., Hasselbring, W., Waller, J., Ehlers, J., Frey, S., Kieselhorst, D., 2009. Continuous monitoring of software services: Design and application of the kieker framework.
Whalen, S., Boggs, N., Stolfo, S.J., 2014. Model aggregation for distributed content anomaly detection. In: Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop. pp. 61–71.
Wu, E., Diao, Y., Rizvi, S., 2006. High-performance complex event processing over streams. In: Proc. SIGMOD/PODS. pp. 407–418.
Wu, X., Zheng, W., Chen, X., Wang, F., Mu, D., 2020. CVE-assisted large-scale security bug report dataset construction method. J. Syst. Softw. 160, 110456.
Yabandeh, M., Anand, A., Canini, M., Kostic, D., 2011. Finding almost-invariants in distributed systems. In: 2011 IEEE 30th International Symposium on Reliable Distributed Systems. IEEE, pp. 177–182.
Zhang, Y., Makarov, S., Ren, X., Lion, D., Yuan, D., 2017. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In: Proceedings of the 26th Symposium on Operating Systems Principles. pp. 19–33.
Zheng, W., Feng, C., Yu, T., Yang, X., Wu, X., 2019. Towards understanding bugs in an open source cloud management stack: An empirical study of OpenStack software bugs. J. Syst. Softw. 151, 210–223.
Zhou, J., Chen, Z., Wang, J., Zheng, Z., Dong, W., 2014. A runtime verification based trace-oriented monitoring framework for cloud systems. In: Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on. IEEE, pp. 152–155.
Zipkin, Home page of Zipkin, https://zipkin.io.

Domenico Cotroneo (Ph.D.) is a full professor at the University of Naples Federico II, Italy. His research interests include software fault injection, dependability assessment, and field-based measurement techniques.

Luigi De Simone (Ph.D.) is a postdoctoral researcher at the University of Naples Federico II, Italy. His research interests include dependability benchmarking, fault injection testing, virtualization reliability and its application on safety-critical systems.

Pietro Liguori (Ph.D.) is a postdoctoral researcher at the University of Naples Federico II, Italy. His research activity includes anomaly detection, failure analysis, and software fault injection in cloud computing infrastructures. His research interests also focus on neural machine translation to automatically generate software exploits.

Roberto Natella (Ph.D.) is an assistant professor at the University of Naples Federico II, Italy. His research interests include dependability benchmarking, software fault injection, software aging and rejuvenation, and their application in OS and virtualization technologies.