The Journal of Systems & Software 181 (2021) 111043

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Enhancing the analysis of software failures in cloud computing systems with deep learning

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

Università degli Studi di Napoli Federico II, Naples, Italy

ARTICLE INFO

Article history:
Received 18 March 2021
Received in revised form 22 May 2021
Accepted 28 June 2021
Available online 12 July 2021

Keywords:
Failure mode analysis
Software failures
Fault injection
Cloud computing
Deep learning
OpenStack

ABSTRACT

Identifying the failure modes of cloud computing systems is a difficult and time-consuming task, due to the growing complexity of such systems, and the large volume and noisiness of failure data. This paper presents a novel approach for analyzing failure data from cloud systems, in order to relieve human analysts from manually fine-tuning the data for feature engineering. The approach leverages Deep Embedded Clustering (DEC), a family of unsupervised clustering algorithms based on deep learning, which uses an autoencoder to optimize data dimensionality and inter-cluster variance. We applied the approach in the context of the OpenStack cloud computing platform, both on the raw failure data and in combination with an anomaly detection pre-processing algorithm. The results show that the performance of the proposed approach, in terms of purity of clusters, is comparable to, or in some cases even better than manually fine-tuned clustering, thus avoiding the need for deep domain knowledge and reducing the effort to perform the analysis. In all cases, the proposed approach provides better performance than unsupervised clustering when no feature engineering is applied to the data. Moreover, the distribution of failure modes from the proposed approach is closer to the actual frequency of the failure modes.

1. Introduction

Nowadays, cloud computing fuels critical services for our life, such as telecom, healthcare, transportation, and other domains where high reliability is mandatory. However, cloud software systems can fail in unpredictable ways, due to cascading propagation of faults across their components (Garraghan et al., 2018; Hole and Otterstad, 2019). To prevent service outages, cloud system designers need to know in advance how their software system behaves under failure (*failure modes*) before deploying it in operation. Knowledge of failure and repair characteristics is valuable for designers in order to plan failure management solutions (Vishwanath and Nagappan, 2010; Li et al., 2006; Garraghan et al., 2014).

To get data about software failures, *fault injection* is typically adopted (Hsueh et al., 1997; Arlat et al., 1990), i.e., the deliberate insertion of *faults* (such as resource exhaustion, software bugs, connection loss, etc.) into a software system in a controlled experiment, in order to trigger failures. These experiments produce a large amount of failure data, in terms of hundreds of thousands of events and execution traces. From this large amount of data, *failure modes analysis* aims to identify which are the recurring failure modes and their relative frequency. Such analysis guides the human designer towards prioritizing the development of failure management mechanisms for the most frequent and severe failure modes.

Unfortunately, failure mode analysis is a difficult and time-consuming task, due to the size and complexity of failure data. Moreover, failure mode analysis is hindered by the non-deterministic behavior of cloud systems, which causes random variations in the timing and the ordering of events in the system, thus introducing noise in the failure data (Zhao et al., 2010). Therefore, failure mode analysis techniques must be robust to noise in the failure data. The adoption of unsupervised machine learning techniques, such as clustering and anomaly detection, comes to the rescue but still faces some limitations. These techniques require the preliminary selection and transformation features (*feature engineering*) (Mousavi et al., 2019; Zhang et al., 2020; Xu et al., 2021), to make the failure data more amenable for analysis. This effort requires deep domain knowledge and represents a significant up-front cost.

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

* 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.2021.111043
0164-1212/

the raw failure data into a compact set of features. The approach transforms the data by jointly optimizing for the reconstruction error (i.e., the transformed features are still representative of the sample) and inter-cluster variance (i.e., to make it easier to identify groups of similar failures).

We evaluated the proposed approach on a dataset of thousands of failures from OpenStack (2018a), a popular platform used in several private and public cloud computing systems, and the basis of over 30 commercial products (OpenStack project, 2018a,c). As an additional contribution to this work, we publicly released this dataset for the research community. We compare the proposed approach to a manually fine-tuned clustering technique. The results demonstrate that the proposed approach can identify clusters with accuracy similar, or in some cases, even superior, to the fine-tuned clustering, with a low computational cost.

The paper is structured as follows: Section 2 discuss the related work; Section 3 provides details on the background; Section 4 presents the proposed approach; Section 5 evaluates the approach; Section 6 concludes the paper.

2. Related work

Uncertainty in fault injection experiments. Uncertainty is a key aspect in fault injection experimentation since the behavior of a complex system depends on many factors that are difficult or impossible to control. This problem is exacerbated when the fault-injection is used in cloud computing, where the human analyst has to deal with the non-deterministic nature of such systems. State-of-the-art provides several works that addressed this problem by applying solutions based on statistical techniques. Several studies leveraged the statistical models to model the probability of failures during hardware fault-injection experiments (Arlat and Moraes, 2011; Skarin et al., 2010; Palazzi et al., 2019). Arlat et al. (1993) proposed a solution that brings together the coverage evaluation of the fault coverage and the occurrence of the faults to estimate the dependability of the complex fault-tolerant systems. By estimating the probabilities of the failure modes of the system, Voas et al. (1997) presented a solution to reduce the uncertainty of whether different software faults impact the behavior of the system. To assess the quality of the measurements in terms of uncertainty, repeatability, resolution, and intrusiveness, Bondavalli et al. (2007, 2010) applied the principles of *measurement theory*. In AMBER project (Wolter et al., 2012), the authors used data mining to identify the factors (i.e., workloads, the fault types, etc.) with the highest impact on the performance and availability of the target system. Loki tool (Chandra et al., 2004) uses an off-line clock synchronization algorithm to collect traces of events exchanged among the nodes, and performs a post-experiment analysis to identify if the injections hit the desired state, and repeats the experiments only when needed. Gulenko et al. (2018) introduced an anomaly detection approach that leverages an online clustering method to define the normal behavior of monitored components. Wu et al. (2020) proposed a method that applies a dependency graph and an autoencoder to identify the causes of the performance degradation in the microservices of the cloud. Both previous works evaluated the proposed solutions by injecting performance anomalies in the cloud computing system.

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

Failure mode clustering. The use of clustering to automatically discover and analyze failure modes is a topic widely addressed by previous research. Arunajadai et al. (2004) described a clustering-based method for grouping failure modes in electromechanical consumer products. The approach groups failure modes based on their occurrence, to determine whether a failure should be considered by itself or whether it tends to accompany other kinds of failures. Then, the analyst can prioritize critical failure modes. The approach uses a hierarchical clustering algorithm with the complete linkage method. Chang et al. (2015) combines clustering with risk management, by grouping failure modes that have similar risk levels concerning three factors (severity, occurrence, detection), and visualizes them to ease multi-criteria decision making. Their approach clusters and visualizes failures as a tree structure that is easy to understand. It is evaluated in the context of farming applications. Duan et al. (2019) analyze evaluations of failure modes in natural language by FMEA experts, using fuzzy sets to extract features, and the *k-means* algorithm to cluster the failure modes. Xu et al. (2020) proposed a method to construct the component-failure mode (CF) matrix automatically, by mining unstructured texts using the Apriori algorithm and the semantic dictionary WordNet to build a standard set of failure modes. As in the work by Arunajadai et al. (2004), the matrix is used for grouping the failure modes using clustering algorithms, such as the *K-means*. Rahimi et al. (2019) analyzed a large dataset of truck crash data, based on police reports about the driver, vehicle, crash, and citation information. They address the problem of high-dimensionality spaces, by adopting block clustering to investigate heterogeneity in the crash dataset. This approach considers two sets (observations and variables) simultaneously and organizes the data into homogeneous blocks. Liu et al. contributed with several studies on the failure mode and effects analysis (Huang et al., 2020). They improved failure mode analysis using two-dimensional uncertain linguistic variables and alternative queuing (Liu et al., 2018) and proposed a novel approach combining HULZNs and DBSCAN algorithms to assess and cluster the risk of failure modes (Liu et al., 2020). They evaluated the feasibility of the proposed approaches in real use-case scenarios, showing the ability to classify failure modes in complex and uncertain conditions.

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

3. Background

This section provides information on cloud computing systems, with emphasis on OpenStack, on failure mode analysis, and on the open issues that are addressed in this work.

3.1. Overview of cloud computing systems

A cloud computing system consists of processes distributed across a data center, which cooperate by message passing and remote procedure calls (e.g., through message queues and REST API calls). These systems are quite complex, as they typically consist of software components of millions of lines of code (LoC), which run across dozens of computing nodes.

In this work, we consider OpenStack as our case study. OpenStack contains a large set of components, each providing APIs to manage virtual resources, and consists of ∼20 million LoC (OpenStack project, 2018b). The three most important components of

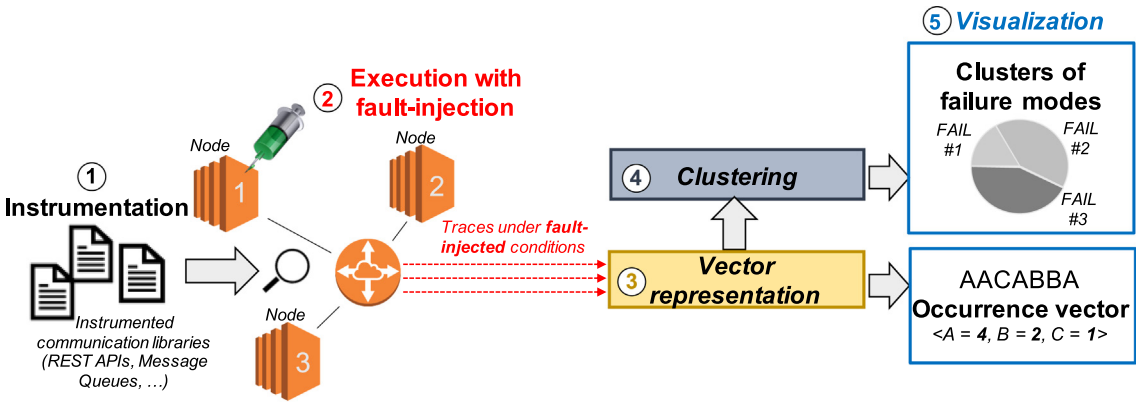

Fig. 1. Failure mode analysis based on plain sequences of messages (*SEQ*).

OpenStack (Denton, 2015; Solberg, 2017) are: (i) the **Nova** subsystem, which provides services for provisioning instances (VMs) and handling their life cycle; (ii) the **Cinder** subsystem, which provides services for managing block storage for virtual instances; and (iii) the **Neutron** subsystem, which provides services for provisioning virtual networks, including resources such as *floating IPs*, *ports* and *subnets* for instances. In turn, these subsystems include several components (e.g., the Nova sub-system includes *nova-api*, *nova-compute*, etc.), which interact through message queues internally within OpenStack. The Nova, Cinder, and Neutron subsystems provide external REST API interfaces to cloud users.

3.2. The problem of failure mode analysis

Cloud systems can fail in many different ways, and the effects of failures (*failure modes*) are often not known in advance by the developers. In the most subtle cases, the system may be still available to users, but return wrong data, exhibit poor performance, or corrupt the state of resources, leading to poor quality of service. To identify failure modes of cloud systems, we frame the problem as a *data analysis* task. The input of the analysis is the (failed) executions of the system. Data about failed executions are obtained from fault injection experiments (i.e., faults are introduced to assess fault-tolerance), and from a deployed system in operation. Every execution is represented as a sequence of events (*trace*) that occurred during the execution. This data amounts to thousands of executions and thousands of events in each execution. Therefore, we analyze the traces using *unsupervised machine learning* techniques, to automatically discover recurring patterns among the failures. The effect of failures is valuable for developers, as they can introduce failure management strategies against them.

In our context, an event in a trace represents a *message* exchanged between nodes in the system. In cloud computing systems, the nodes perform or serve a request after receiving messages to provide a service to another node (e.g., through remote procedure calls), and reply with messages to provide the response and results; moreover, nodes use messages to asynchronously notify a new state to other nodes in the system. Therefore, the messages are considered the main observation point for debugging and verification of distributed systems since they reflect the state and the activity of the system (Leesatapornwongsa et al., 2016). These messages can be recorded in execution traces for later analysis, using distributed tracing technologies (Sigelman et al., 2010; Nedelkoski et al., 2019), which wrap *communication APIs* invoked by the processes.

To identify failure modes, we perform *clustering* to group the experiments into classes (clusters), where each class represents a distinct failure mode of the system under test. In general, clustering algorithms reveal hidden structures in a given data set, by grouping "similar" data objects together while keeping "dissimilar" data objects in separated classes (Xu and Wunsch, 2005). Formally speaking, consider a set of n distinct data objects $\{x_1, \ldots, x_n\}$ and a number of k clusters. A (hard) clustering technique assigns to each data object a label l_i representing its class, with $i \in [1, k]$ (Jain et al., 1999). In the context of failure data, a data object represents an execution of the system while it was experiencing a fault. The ith execution is represented by a vector of features $x_i = [f_1, \ldots, f_d]$. Each feature is a number that represents how many events of a given type occurred during the execution, with d unique types of events. The number of features easily bump up, due to a large amount of failure data (e.g., hundreds of message types, GBs of log files, thousand of traces, and experiments).

For example, let us suppose that we collected three different message types, A, B, C. Let be $x_i = [4, 2, 1]$ the vector associated to a trace collected during the ith fault injection experiment. This implies that the events A, B, C were observed 4, 2 and 1 times, respectively, during the ith experiment.

The steps for failure mode analysis are summarized in Fig. 1. We label this basic approach as *SEQ* since it is based on plain sequences of events from fault-injection experiments.

3.3. Machine learning techniques for failure mode analysis

Clustering execution traces from cloud systems is a challenging issue, due to the *non-deterministic* behavior of these systems. Even if the system is executed several times in the same way, under the same workload, the timing and the order of messages can unpredictably change, due to delays in communication and computation. Thus, there is a need to discriminate between variations in the traces due to different failure modes (which should be divided into different clusters), and "benign" variations caused by non-determinism (which should still be grouped in the same cluster).

To improve failure mode analysis, in previous work we proposed an approach based on *anomaly detection* (Fig. 2), which screens out benign variations from the traces (Cotroneo et al., 2019a, 2020). The anomaly detection identifies specific events of interest (i.e., failure symptoms) from the large set of events that are generated from fault injection experiments. As a matter of fact, only a few events are actually failure symptoms, such as messages out-of-order or missing, and messages that deliver exceptions. Therefore, it is useful to identify these symptoms and focus clustering on them.

The anomaly detection approach first executes the system several times, using a fault-free workload in which no fault is

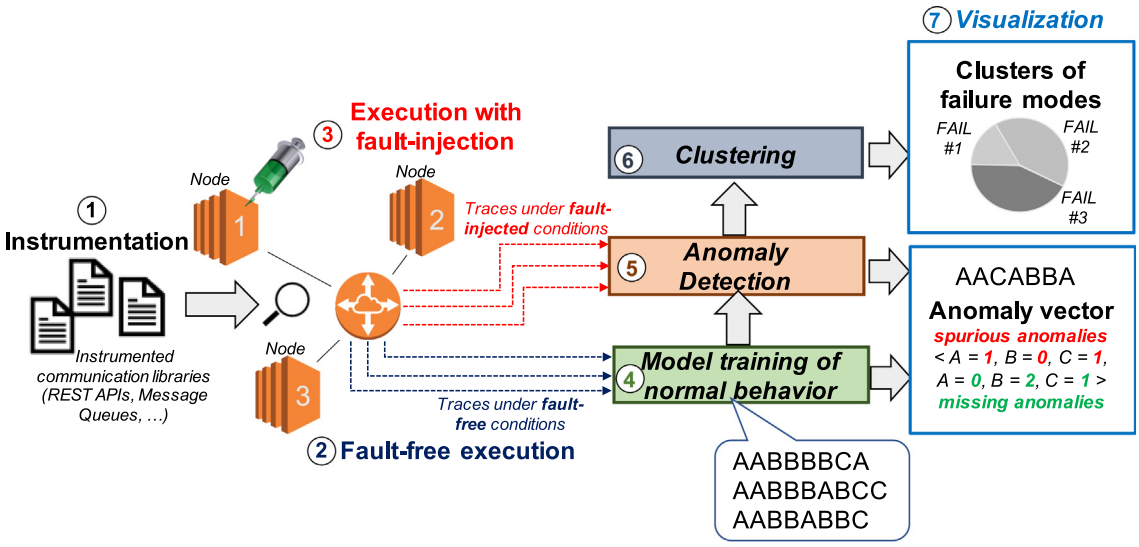

Fig. 2. Failure mode analysis based on anomaly detection, using sequence matching and variable-order Markov models (*LCS with VMM*).

injected. Thus, any variation in these traces is a benign one, and they are denoted as ***fault-free traces***. The fault-free traces will be used as a reference for "normal" (i.e., non-failure) behavior. The approach then performs fault injection experiments and collects ***fault-injected traces*** from these executions. The approach uses string analysis techniques (*longest common subsequence*, LCS (Bergroth et al., 2000)) to identify common events among fault-free traces and to mark them as non-anomalous. Then, it applies a *Variable-order Markov Model* (VMM) to analyze non-deterministic variations and to generate a new vector of features (*anomaly vector*). For each message type t, the vector includes two features, respectively:

- **Spurious anomalies**: the number of times that an anomalous message of type t appears in the fault-injected trace;
- **Omission anomalies**: the number of times that the message type t *does not* appear in the fault-injected trace, but the message should have been occurred according to the probabilistic model.

In total, the number of features is twice the number d of unique event types. For example, let us suppose that we observe three different types of messages, A, B, C. Let be $x_i = [1, 0, 1, 0, 2, 1]$ the vector that represents the ith fault-injected trace. These features can be interpreted as follows:

- The first three features (valued 1, 0, 1) are spurious anomalies. Anomaly detection identified two spurious events, one for the event type A and one for the event type C.
- The last three features (valued 0, 2, 1) are omitted anomalies. Anomaly detection identified three omitted events, two for the event type B, and one for the event type C.

3.4. Open issues

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

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

To better understand this problem, Fig. 3 shows a graphical representation of the activation of components in the OpenStack cloud computing system during a fault-injection experiment (Cotroneo et al., 2019b). The figure divides events into *common*, *spurious*, and *omitted* events, as described in Section 3.3. For simplicity, the figure does not include events for resource monitoring, garbage collection, etc. Nevertheless, the figure shows that human analysts must deal with hundreds of events. Some of the events are relevant symptoms of the failure mode, such as exceptions received by the client from REST API calls. Other events are not a symptom of the failure but are benign variations caused by asynchronous updates from Neutron. In order to accurately cluster this failure mode, the features representing REST API calls should be assigned a larger weight than some of the Neutron events, which are non-deterministic and are prone to noise.

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

4. Proposed approach

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

Our solution leverages *Deep Embedded Clustering* (DEC), a family of algorithms that performs clustering on the embedded features of an autoencoder (Xie et al., 2016; Ghasedi Dizaji et al., 2017; Guo et al., 2017; Li et al., 2018; Yang et al., 2017; Guo et al., 2018). The proposed solution (Fig. 4) uses DEC on the raw vector representations of the fault-injected traces, which are the same

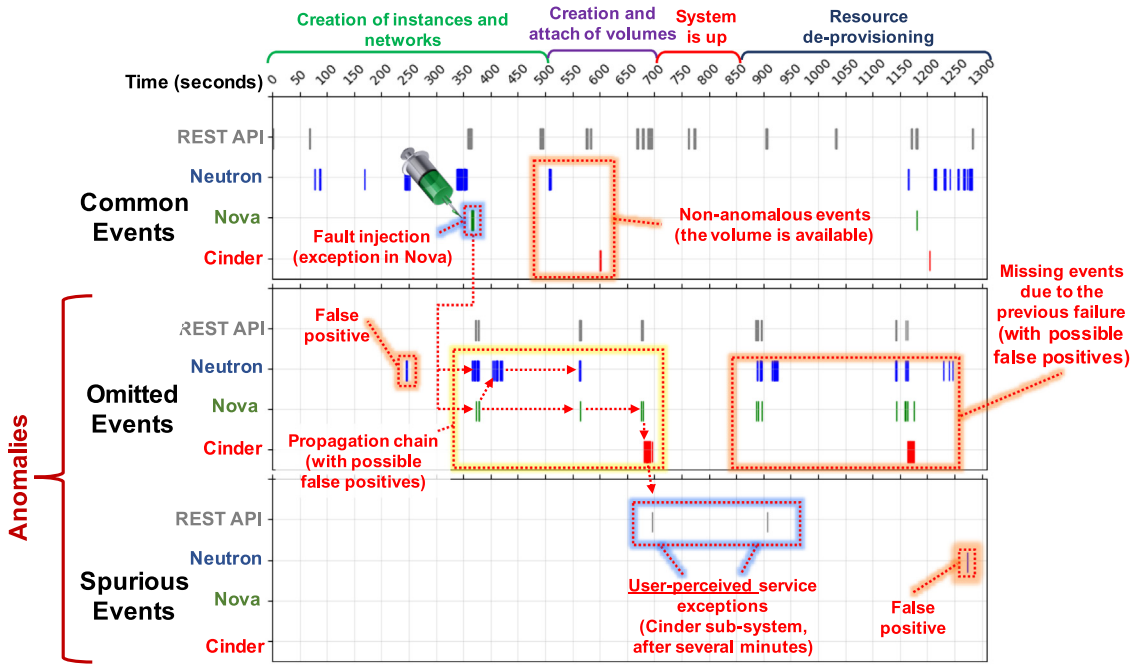

Fig. 3. Graphical representation of a fault-injection experiment in the OpenStack cloud computing system.

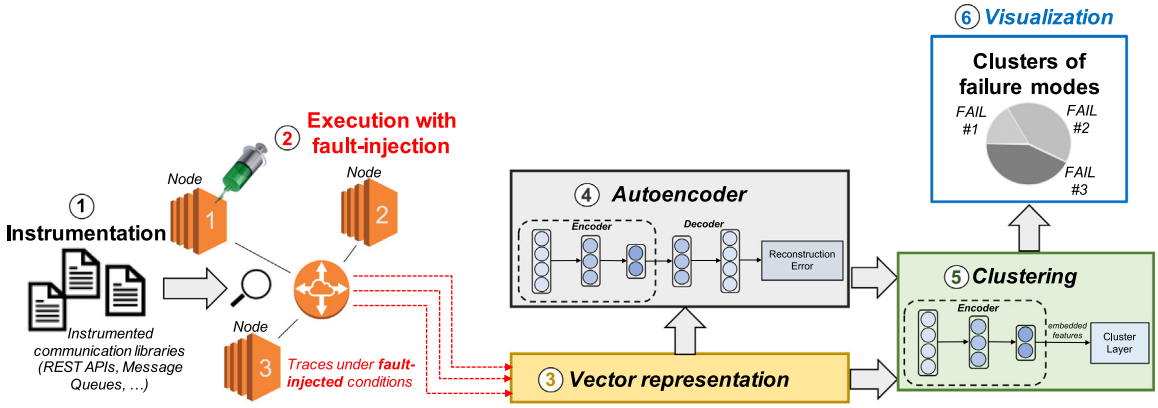

Fig. 4. Overview of the proposed solution.

ones of the *SEQ* approach discussed before (by replacing the step ④ of Fig. 1). This proposed approach relieves the human analyst from fine-tuning the feature weights in the clustering stage, thus saving manual efforts.

An alternative version of the proposed solution is in combination with anomaly detection, by applying it on anomaly vectors, as in the *LCS with VMM* (by replacing the step ⑥ of Fig. 2). In this case, the human analyst invests effort to train an anomaly detection model using fault-free traces, but without manual feature engineering. This combined approach can further improve the accuracy of failure mode analysis. We also analyze this approach in the experimental part of this work.

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

4.1. Parameter initialization

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

We initialize the autoencoder network layer by layer so that the layers work as a *denoising autoencoder* (Vincent et al., 2008;

Lu et al., 2013) trained to reconstruct the previous layer's output after random corruption of the data. We set the network input dimension equal to d, where d is the number of the vector features (which depends on the number of unique events).

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

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

To start the clustering phase, we need to initialize the cluster centers. Therefore, we firstly input the initialized DNN with the data points to get embedded data points, and then apply a clustering algorithm in the feature space Z to obtain k initial centroids. Our solution adopts the *K-Medoids*, a clustering method that performs the clustering phase by minimizing the sum of the dissimilarities between objects and a reference point for their cluster. As a reference point, this method uses the *medoid*, i.e., the most centrally located object in a cluster. Therefore, this method is considered less sensitive to outliers than the classical *K-Means*, which takes the mean value of the objects in a cluster as a reference point (Arora et al., 2016; Velmurugan and Santhanam, 2010).

4.2. Parameter optimization

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

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

Then, we recompute the clusters iteratively by learning from the current soft assignment. In particular, the clustering model is trained to minimize the distance between the soft assignments and an artificial "target" distribution, which is a transformed version of the probabilities in the soft assignment that widens the gap between the probabilities (Peng et al., 2019). In our case, we compute the target distribution by raising the soft assignments to the second power and normalizing the values. The approach gives more emphasis on data points assigned with high probability, and at the same time, it also optimizes for the ones with low probability. By optimizing for the low distance between the actual soft assignments and the target distribution, we obtain clusters with larger intra-cluster variance, thus improving the cluster quality.

For the optimization, we minimize the *Kullback–Leibler divergence* (KL) between the soft assignments and the target (Jabi et al., 2019). The KL divergence is a loss function that measures the difference between two distributions. We update the target distributions after a specific number of clustering iterations. The clustering model is then trained to minimize the KL divergence loss between the output of the clustering and the target distribution. We leveraged the Stochastic Gradient Descent (SGD) with momentum (Qian, 1999) to optimize simultaneously both the cluster centers and the DNN parameters. The parameter optimization process stops when a percentage of points below a *convergence threshold* changes the assigned cluster between two iterations in a row. We set the convergence threshold equal to 0.1%.

Table 1
Workload characteristics.

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

5. Experiments

In this section, we evaluate the proposed approach in the context of failure data from the OpenStack cloud computing platform. We obtain failure data from fault-injection experiments, which were performed on 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. In particular, in our experiments, we targeted Nova, Cinder, and Neutron subsystems, which are considered the three most important services of OpenStack (Denton, 2015; Solberg, 2017).

We injected faults during the execution of OpenStack, by simulating exceptional conditions during the interactions among its components. We targeted the internal APIs used by OpenStack components for managing instances, volumes, networks, and other resources. For example, we injected faults during calls to the *cinder-volume* component within the Cinder subsystem to perform operations on the volumes.

To define the faults to inject into the target system, we analyzed over 200 problem reports on the OpenStack bug repository. This analysis allowed us to identify the most recurrent bugs in OpenStack over the last years. In particular, we choose the following faults, which are among the most frequent in OpenStack (Cotroneo et al., 2019c):

- ***Throw exception***: The target method raises an exception, according to the per-API list of exceptions;
- ***Wrong return value***: The target method returns an incorrect value. In particular, the returned value is corrupted according to its data type (e.g., we replace an object reference with a null reference, or replace an integer value with a negative one);
- ***Wrong parameter value***: The target method is called with an incorrect input parameter. Input parameters are corrupted according to the data type, as for the previous fault type;
- ***Delay***: The target method returns the result after a long delay. This fault can trigger timeout mechanisms inside the system or can cause a stall.

5.1. Workloads

We performed three distinct sets of fault injection experiments (*campaigns*), in which we applied three different workloads.

▷ ***New deployment workload*** (DEPL): This workload configures a new virtual infrastructure from scratch. It creates VM instances,

volumes, key pairs, and a security group; attaches the instances to the volume; creates a virtual network consisting of a subnet and a virtual router; assigns a floating IP to connect the instances to the virtual network; reboots the instances, and then deletes all the created resources. This workload is meant to stress in a balanced way Nova, Cinder, and Neutron subsystems.

▷ ***Network management workload*** (NET): This workload includes network management operations, to focus interest on the operations related to the virtual networks and, therefore, on the Neutron subsystem. The workload initially creates a network and a VM, then generates network traffic via the public network. After that, it creates a new network with no gateway, brings up a new network interface within the instance, and generates traffic to check whether the interface is reachable. Finally, it performs a router rescheduling, by removing and adding a virtual router resource.

▷ ***Storage management workload*** (STO): This workload mainly performs the operations related to the storage management of instances and volumes to stress the Nova and Cinder subsystems. It creates a new volume from an image, boots an instance, then rebuilds the instance with a new image before cleaning up the resources.

All of these workloads invoke the OpenStack APIs provided by the Nova, Cinder, and Neutron subsystems. We implemented the workloads by reusing integration test cases from the *OpenStack Tempest* project (OpenStack, 2018b) since these tests are already designed to trigger several subsystems and components of OpenStack and their virtual resources. We selected these workloads to evaluate our approach in different conditions (i.e., networks operations, storage operations, etc.) and to emphasize the propagation of the failure across different subsystems that can be caused by the injected faults.

To understand when the system experiences a failure, our workload generator performs *assertion checks* on the status of the virtual resources. For example, the workload assesses the connectivity of the VM instances via SSH, or query the OpenStack API to check the status of the instances, volumes, and networks. The checks helped us at manually diagnosing the outcome of every experiment, in addition to logs produced by the system. We used this information to build a *ground truth* of the failures during the experiments, i.e., a reference for evaluating the accuracy of the proposed approach (see the next subsection). We consider an experiment as failed if at least one API call returns an error (**API error**) or if there is at least one assertion check failure (**assertion check failure**). Before every experiment, we re-deploy the cloud management system, remove all temporary files and processes, and restore the OpenStack database to its initial state. These actions are needed to avoid any residual effect of the previous experiments that can impact the current one.

5.2. Failure dataset

To inject faults in Nova, Neutron, and Cinder, we performed a full scan of their source code, using an automated fault injection tool (Cotroneo et al., 2020), to identify all injectable API calls. We then checked whether the injectable API calls are indeed executed by the workloads. In the experimental campaign, we performed one fault injection test for each injectable location that is covered by the workloads. In total, we performed 2538 fault injection tests, and we observed failures in 1314 tests (52%). In the remaining tests (33%), 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). This is a typical phenomenon that often occurs in fault injection experiments (Christmansson and Chillarege, 1996; Lanzaro et al., 2014); yet, the experiments provided us a large and diverse set of failures for our analysis.

Table 2
Failure mode classes per workload.

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

Table 1 shows, for each workload, the number of event types d observed in the distributed system during the execution of the workloads, the average length of the fault-free sequences (in term of the number of events in the trace), the total number of fault injection experiments for the workload, and the number of experiments that experienced at least one failure. The number of event types and the total number of events reflects the extent and diversity of the work put on the system. We notice that DEPL is the most extensive workload in terms of both distinct operations and the total number of operations, followed by NET and by STO.

To every experiment of the fault-injection campaigns, we assigned a label expressing the *failure class*, or *failure mode*, i.e., the type of failure that the system experienced during the experiment. The classes of failure serve as *ground-truth* for evaluating the results of the clustering. A good clustering solution, indeed, should be close to the classification of the ground truth. Having a reliable ground truth is a common problem in the research problems involving the analysis of the logs. System logs are usually good indicators of system state as they contain reports of events that occur on the several interrelated components of complex systems (Lim et al., 2008). Previous works leveraged the collection of system logs as sources of data, which could be analyzed by a system to make it aware of its internal state (Vaarandi, 2004; Aharon et al., 2009; Fu et al., 2009; Makanju et al., 2011). Therefore, to assign a failure-class to every experiment, we leveraged the assertion checks and the API errors raised by OpenStack. Furthermore, we investigated the logs of the systems and the anomalies in the traces. To reduce the possibility of errors in manual labeling, all the authors discussed cases of discrepancy, obtaining a consensus on the failure modes.

In our experiments, we found the following types of failure modes:

- ***Instance Failure***: Failure of the operations related to the VM instance. For example, the creation of the virtual machine fails, or the virtual machine is created but it is in not a valid state.
- ***Volume Failure***: Failure of the operations related to the volume, such as the creation and/or the attach of the volume to the virtual machine, or also the volume is created but it is in an error state.
- ***Network Failure***: Failure of the operations related to the networks, such as the creation of networks and sub-networks, the association of the IP address to the virtual machine, etc.
- ***SSH Failure***: Failure to reach the virtual machine via SSH. For example, even if the virtual machine is correctly created and up, it is not reachable for the connection.
- ***Cleanup Failure***: Failure related to the operations performed in the last phase of the workload, when the system is not able to serve the requests of deletion of the resources previously created.
- ***No Failure***: The system can perform all requests without raising any failure during the experiment.

Table 3
Purity values of clustering without performing anomaly detection (*SEQ* data). Bold values are the best performance.

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

Table 4
Purity values of clustering on top of anomaly detection (*LCS with VMM*). Bold values are the best performance.

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

Even if we use the same labels for the failure modes across the three workloads, each failure mode should be considered different for each workload since they involve different resources and APIs during execution (e.g., DEPL and STO have both cleanup failures, but with different behaviors). We found 6 different failure modes for DEPL and 4 failure modes for both NET and STO. Since DEPL is our most stressful workload, it is unsurprising to identify a higher number of failure classes among the experiments of this fault-injection campaign.

We shared the failure dataset on GitHub[1] to help the research community in the application and evaluation of new solutions for clustering the failure modes of the systems. For every experiment of the three fault-injection campaigns, the dataset contains the events exchanged in the system and the corresponding failure label. We shared the representations of experiments with and without the anomaly detection phase (as shown in Figs. 2 and 1, respectively).

5.3. Evaluation metrics

To evaluate the quality of the clustering, we compare the cluster assigned to the experiment with the failure class of the experiment defined in our ground truth (Table 2). To associate the clusters to the failure classes, we identify, for every cluster, the failure label with the largest overlap and assign every element in the cluster to the ground-truth class (Modha and Spangler, 2003). This evaluation is conservative since it can assign multiple clusters to the same ground truth, but it cannot associate the same cluster to different classes of failure.

In quantitative terms, let C be the number of ground-truth classes $\{\omega_c\}_{c=1}^{C}$. The *purity* of a cluster is defined as the fraction of elements in the cluster that matches the ground-truth class (Xiong et al., 2009). Assuming K clusters, for each cluster $\{\tau_k\}_{k=1}^{K}$ we define $P_k = {}^1/_{n_k} \cdot max(n_k^c)$, where n_k is the size of the cluster τ_k, and n_k^c is the number of elements in the cluster τ_k that belong to the class with label w_c. The overall *purity* achieved by a clustering algorithm is the weighted sum of the purities across classes, given by $P = \sum_{k=1}^{K} {}^{n_k}/_{n} \cdot P_k$. Purity ranges between 0 (total misclassification) and 1 (perfect clustering).

5.4. Experimental results

We evaluated our solution in two scenarios:

- The deep neural network technique is applied on the raw failure data, without performing any anomaly detection. This is the same data as in the *SEQ* approach (see Section 3).
- The deep neural network technique is applied on top of anomaly detection, i.e., on the anomaly vectors. This is the same data generated by the *LCS with VMM* approach (see Section 3).

For each of these cases, we compare the proposed approach (*DEC*) against baselines, in which we apply traditional clustering. For the baselines, we consider both the case of plain features (*k-medoids w/o fine-tuning*), and a manual fine-tuning of the weights of the features (*k-medoids with fine-tuning*), as discussed in Section 3. We remark that the fine-tuning of the features is a difficult and time-consuming task, due to the exploration of a large number of features (hundreds of event types) and the deeper study of event types in OpenStack (e.g., synchronous and asynchronous events, missing and spurious events, RPC messages and REST APIs, etc.). This exploratory data analysis was performed with Matlab code and took around two weeks of manual effort.

To evaluate different use-cases and conditions, we applied our solution to perform clustering on the data from the three fault-injection campaigns, one for each workload. The input data X is a matrix with the number of rows equal to the number of fault-injection experiments. The columns are dependent on the number of different event types d observed during the execution of the workload. In particular, the number of columns is d when the clustering is applied without the help of the anomaly detection, and $2d$ when the clustering is applied with the anomaly detection (since the algorithm discerns the spurious events from the omitted ones, as explained in Section 3.3.

We set the hyper-parameters to minimize the reconstruction loss. During the phase of pre-training, we performed a basic tuning of the parameters following the common practices of previous studies (Mendoza et al., 2019; Koohzadi et al., 2020). We randomly initialized the weights of the layers. The layers were pre-trained for 100,000 iterations and a drop-out rate set to 20%. We trained DEC with additional 100,000 iterations but without a drop-out rate. We set the size of the mini-batch to 256, the starting learning rate to 10%, which is divided by 10 every 20,000 iterations, and the weight decay to 0 (Xie et al., 2016). For each dataset, we tuned the autoencoder by configuring the number and the dimension of the inner layers (between 2 and 4 layers, of decreasing dimension from d to K), and the distance metric for clustering ($L1$, city block, and $L2$, euclidean). Moreover, to initialize the centroids of the clusters, we selected the best solution after running the k-medoids with 30 repetitions.

Tables 3 and 4 show the clustering results, in terms of purity, without and with anomaly detection, respectively. The results without anomaly detection (Table 3, *SEQ* data) show that the use of the DEC achieves a higher purity compared to traditional clustering, both without and with fine-tuning of feature weights. This behavior applies to each of the three workloads. The scenario without anomaly detection is the most important one since it is the case of the busy system designer that needs quick feedback from fault injection tests, to quickly perform the next iteration of development. For example, the designer may add or revise fault-tolerance mechanisms, and test them again on a new round of fault injection experiments. In these cases, avoiding training an anomaly detection model is useful to speed up data analysis.

In the case of clustering in combination with anomaly detection (Table 4, data from *LCS with VMM*), the data have already been processed and reduced before clustering. Therefore, clustering achieves better results than using data without anomaly detection. In particular, clustering benefits most in the case of manual fine-tuning of the feature weights, as *k-medoids with fine-tuning* always achieves better results than both the basic *k-medoids w/o fine-tuning* and *DEC*. However, these better results come at the cost of manually setting the weights of the features, which requires a deep knowledge of the system internals, and

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

efforts to best tune them concerning the specific workload. Instead, the *DEC* approach achieves performance that is close to the case of fine-tuning, with significantly less effort from the human analyst. Moreover, *DEC* always returns better results than the basic *k-medoids*, consistently over all the workloads, and both with and without anomaly detection. Our experiments also pointed out that the standard deviation is below 5%, and data are normally distributed around the mean.

To better understand the impact of the clustering on the analysis of failure modes, we inspected the distribution of the failure data samples across the clusters and compared it to the distribution of the actual failure modes (Table 2). Ideally, the distribution across clusters matches the actual failure modes, so that the human designer can prioritize the development of fault tolerance mechanisms according to the distribution. Moreover, it is sufficient for the human designer to only analyze one or a few experiments from the same class, thus making the analysis more efficient. To map the clusters to the failure modes of Table 2, we followed the approach described in Section 5.3. We remark that this analysis does not focus on the quality of clusters (i.e., samples misclassified in the wrong cluster), as the previous analysis already provided figures about the purity of the clusters. Here, we focus on the distribution of the clusters that would be presented to the human designer, as the shape of the distribution influences the interpretation of the failure data.

Fig. 5 shows the distributions of the clusters for the proposed approach (*DEC*), for the baselines (*k-medoids* with and without fine-tuning), and the actual distribution of the failure modes according to the ground truth. The size of the clusters for *Instance Failure*, *Network Failure*, and *Cleanup Failure* from the clustering techniques are close to the actual frequency of these failure modes. Instead, there are noticeable differences for the remaining failure modes. In the case of *Volume Failure*, the *k-medoids w/o fine-tuning* misses this failure mode, while the cluster from *k-medoids with fine-tuning* is only half of the actual frequency of this failure mode. In the case of *SSH Failure*, which accounts for a minor part of the failures, all of the clustering approaches do not report any failure. We do not attribute this result to the clustering techniques, but to the similarity of events occurring in this failure mode to the ones occurring for *Instance Failure*, which misleads clustering. Instead, we believe that this failure mode could be better analyzed by looking not only at the execution traces but also at additional information sources, such as system logs. Finally, both *k-medoids* with and without fine-tuning overestimate the cases of *No Failure*, as they report several hundreds of no-failures more than the actual size of this class. This error is the most severe one since it misleads the human designer at believing that the system fails less frequently than the actual truth (e.g., about −20% of neglected failures). Thus, with the simple *k-medoids*, the analyst would unjustly trust the reliability of the system. Instead, in the proposed approach, the share of cases of *No Failure* is close to the ground truth.

5.5. Computational cost

We evaluated the computational cost of the proposed approach to estimate the overhead introduced by the use of deep learning to cluster the failure data. We performed several evaluations, by varying the workloads, the vector representation of the experiments (i.e., with and without the anomaly detection), and the layers of the neural network. We found that the use of DEC for clustering introduces an average overhead of ∼23 seconds compared to the basic use of the k-medoids. This time includes the initialization of the cluster centers with k-medoids (i.e., the parameter initialization) and the training of the DNN (i.e., the parameter optimization). The standard deviation is high (∼75% of the average value) since the configuration of the DNNs impacts the computational cost. Nevertheless, the overhead introduced by DEC can be considered acceptable, given that the proposed solution avoids the manual fine-tuning of features, which represents a difficult and time-consuming task.

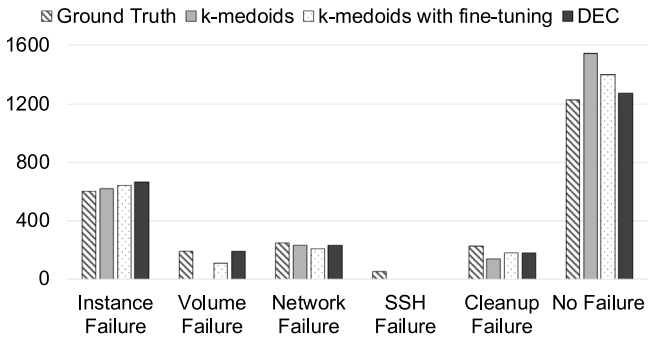

Fig. 5. Distribution of failure modes from different clustering techniques (*SEQ* data).

6. Conclusion

In this paper, we presented a novel approach for analyzing failure data from cloud systems, by using unsupervised learning algorithms and deep learning to cluster the failure data into failure classes. The proposed approach relieves the human analyst from manually tuning the features to achieve a good performance at clustering failure data. The approach leverages an autoencoder for dimensionality reduction and parameter initialization, in combination with a clustering layer to optimize both the reconstruction error and inter-cluster distance.

We presented results on failure data from the popular OpenStack cloud computing platform. The results show that the proposed approach can achieve performance comparable to, or in some cases even better than, the performance of manually-tuned clustering, which entails a deep knowledge of the domain and a significant human effort. In all cases, the proposed approach performs better than unsupervised clustering when no feature engineering is made on the dataset.

The approach has been designed to be applied without any a-priori information about the types of features in the failure data, in order to minimize the manual effort. This is especially important when the cloud system is still under active development when multiple versions are updated, tested, and released at a quick pace. However, our approach cannot exceed the accuracy that can be achieved by leveraging the knowledge of the human analyst about the system. Furthermore, since the approach uses deep neural networks, it requires high hardware requirements to keep computational times acceptable, in particular when the amount of data to analyze is very large.

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.

Acknowledgments

This work has been partially supported by the University of Naples Federico II in the frame of the Programme F.R.A., project id OSTAGE. We are grateful to Gabriella Karamanolis for her help in the early stage of this work.

References

Aharon, M., Barash, G., Cohen, I., Mordechai, E., 2009. One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer, pp. 227–243.

Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.-C., Laprie, J.-C., Martins, E., Powell, D., 1990. Fault injection for dependability validation: A methodology and some applications. IEEE Trans. Softw. Eng. 16 (2), 166–182.

Arlat, J., Costes, A., Crouzet, Y., Laprie, J.-C., Powell, D., 1993. Fault injection and dependability evaluation of fault-tolerant systems. IEEE Trans. Comput. 42 (8), 913–923.

Arlat, J., Moraes, R., 2011. Collecting, analyzing and archiving results from fault injection experiments. In: 2011 5th Latin-American Symposium on Dependable Computing. IEEE, pp. 100–105.

Arora, P., Varshney, S., et al., 2016. Analysis of k-means and k-medoids algorithm for big data. Procedia Comput. Sci. 78, 507–512.

Arunajadai, S.G., Uder, S.J., Stone, R.B., Tumer, I.Y., 2004. Failure mode identification through clustering analysis. Qual. Reliab. Eng. Int. 20 (5), 511–526.

Bergroth, L., Hakonen, H., Raita, T., 2000. A survey of longest common subsequence algorithms. In: Proc. SPIRE. IEEE, pp. 39–48.

Bondavalli, A., Ceccarelli, A., Falai, L., Vadursi, M., 2007. Foundations of measurement theory applied to the evaluation of dependability attributes. In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07). IEEE, pp. 522–533.

Bondavalli, A., Ceccarelli, A., Falai, L., Vadursi, M., 2010. A new approach and a related tool for dependability measurements on distributed systems. IEEE Trans. Instrum. Meas. 59 (4), 820–831.

Chandra, R., Lefever, R.M., Joshi, K.R., Cukier, M., Sanders, W.H., 2004. A global-state-triggered fault injector for distributed system evaluation. IEEE Trans. Parallel Distributed Syst. 15 (7), 593–605. http://dx.doi.org/10.1109/TPDS.2004.14.

Chang, W.L., Tay, K.M., Lim, C.P., 2015. Clustering and visualization of failure modes using an evolving tree. Expert Syst. Appl. 42 (20), 7235–7244.

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

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

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: Proc. ESEC/FSE. ACM, pp. 200–211.

Denton, J., 2015. Learning OpenStack Networking. Packt Publishing Ltd.

Duan, C.-Y., Chen, X.-Q., Shi, H., Liu, H.-C., 2019. A new model for failure mode and effects analysis based on k-means clustering within hesitant linguistic environment. IEEE Trans. Eng. Manage..

Fu, Q., Lou, J.-G., Wang, Y., Li, J., 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In: 2009 Ninth IEEE International Conference on Data Mining. IEEE, pp. 149–158.

Garraghan, P., Townend, P., Xu, J., 2014. An empirical failure-analysis of a large-scale cloud computing environment. In: 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering. pp. 113–120. http://dx.doi.org/10.1109/HASE.2014.24.

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.

Ghasedi Dizaji, K., Herandi, A., Deng, C., Cai, W., Huang, H., 2017. Deep clustering via joint convolutional autoencoder embedding and relative entropy minimization. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 5736–5745.

Gulenko, A., Schmidt, F., Acker, A., Wallschläger, M., Kao, O., Liu, F., 2018. Detecting anomalous behavior of black-box services modeled with distance-based online clustering. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). IEEE, pp. 912–915.

Guo, X., Gao, L., Liu, X., Yin, J., 2017. Improved deep embedded clustering with local structure preservation.. In: IJCAI. pp. 1753–1759.

Guo, X., Zhu, E., Liu, X., Yin, J., 2018. Deep embedded clustering with data augmentation. In: Asian Conference on Machine Learning. PMLR, pp. 550–565.

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

Hsueh, M.-C., Tsai, T.K., Iyer, R.K., 1997. Fault injection techniques and tools. Computer 30 (4), 75–82.

Huang, J., You, J.-X., Liu, H.-C., Song, M.-S., 2020. Failure mode and effect analysis improvement: A systematic literature review and future research agenda. Reliab. Eng. Syst. Saf. 199, 106885.

Jabi, M., Pedersoli, M., Mitiche, A., Ayed, I.B., 2019. Deep clustering: On the link between discriminative models and k-means. IEEE Trans. Pattern Anal. Mach. Intell..

Jain, A.K., Murty, M.N., Flynn, P.J., 1999. Data clustering: a review. ACM Comput. Surv. 31 (3), 264–323.

Koohzadi, M., Charkari, N.M., Ghaderi, F., 2020. Unsupervised representation learning based on the deep multi-view ensemble learning. Appl. Intell. 50 (2), 562–581.

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. ACM, pp. 397–408.

Leesatapornwongsa, T., Lukman, J.F., Lu, S., Gunawi, H.S., 2016. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. ACM SIGPLAN Not. 51 (4), 517–530.

Li, H., Groep, D., Wolters, L., Templon, J., 2006. Job failure analysis and its implications in a large-scale production grid. In: 2006 Second IEEE International Conference on E-Science and Grid Computing (E-Science'06). IEEE, p. 27.

Li, F., Qiao, H., Zhang, B., 2018. Discriminatively boosted image clustering with fully convolutional auto-encoders. Pattern Recognit. 83, 161–173.

Lim, C., Singh, N., Yajnik, S., 2008. A log mining approach to failure analysis of enterprise telephony systems. In: 2008 IEEE International Conference on Dependable Systems and Networks with FTCS and DCC (DSN). IEEE, pp. 398–403.

Liu, H.-C., Chen, X.-Q., You, J.-X., Li, Z., 2020. A new integrated approach for risk evaluation and classification with dynamic expert weights. IEEE Trans. Reliab..

Liu, H.-C., Hu, Y.-P., Wang, J.-J., Sun, M., 2018. Failure mode and effects analysis using two-dimensional uncertain linguistic variables and alternative queuing method. IEEE Trans. Reliab. 68 (2), 554–565.

Lu, X., Tsao, Y., Matsuda, S., Hori, C., 2013. Speech enhancement based on deep denoising autoencoder.. In: Interspeech. 2013, pp. 436–440.

Makanju, A., Zincir-Heywood, A.N., Milios, E.E., 2011. System state discovery via information content clustering of system logs. In: 2011 Sixth International Conference on Availability, Reliability and Security. IEEE, pp. 301–306.

Mendoza, H., Klein, A., Feurer, M., Springenberg, J.T., Urban, M., Burkart, M., Dippel, M., Lindauer, M., Hutter, F., 2019. Towards automatically-tuned deep neural networks. In: Automated Machine Learning. Springer, Cham, pp. 135–149.

Modha, D.S., Spangler, W.S., 2003. Feature weighting in k-means clustering. Mach. Learn. 52 (3), 217–237.

Mousavi, S.M., Zhu, W., Ellsworth, W., Beroza, G., 2019. Unsupervised clustering of seismic signals using deep convolutional autoencoders. IEEE Geosci. Remote Sens. Lett. 16 (11), 1693–1697.

Nedelkoski, S., Cardoso, J.S., Kao, O., 2019. Anomaly Detection and Classification using Distributed Tracing and Deep Learning. In: Proc. CCGRID, pp. 41–250.

OpenStack, 2018a. Openstack. URL http://www.openstack.org/.

OpenStack, 2018b. Tempest testing project. URL https://docs.openstack.org/tempest.

OpenStack project, 2018a. The openstack marketplace. URL https://www.openstack.org/marketplace/.

OpenStack project, 2018b. Stackalytics. URL https://www.stackalytics.com.

OpenStack project, 2018c. User stories showing how the world #runsonopenstack. URL https://www.openstack.org/user-stories/.

Palazzi, L., Li, G., Fang, B., Pattabiraman, K., 2019. A tale of two injectors: End-to-end comparison of IR-level and assembly-level fault injection. In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 151–162.
Peng, X., Zhu, H., Feng, J., Shen, C., Zhang, H., Zhou, J.T., 2019. Deep clustering with sample-assignment invariance prior. IEEE Trans. Neural Netw. Learn. Syst. 31 (11), 4857–4868.
Qian, N., 1999. On the momentum term in gradient descent learning algorithms. Neural Netw. 12 (1), 145–151.
Rahimi, A., Azimi, G., Asgari, H., Jin, X., 2019. Clustering approach toward large truck crash analysis. Transp. Res. Rec. 2673 (8), 73–85.
Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C., 2010. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Tech. rep., Google, Inc., URL https://research.google.com/archive/papers/dapper-2010-1.pdf.
Skarin, D., Barbosa, R., Karlsson, J., 2010. GOOFI-2: A tool for experimental dependability assessment. In: 2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN). IEEE, pp. 557–562.
Solberg, M., 2017. OpenStack for Architects. Packt Publishing.
Vaarandi, R., 2004. A breadth-first algorithm for mining frequent patterns from event logs. In: International Conference on Intelligence in Communication Systems. Springer, pp. 293–308.
Velmurugan, T., Santhanam, T., 2010. Computational complexity between K-means and K-medoids clustering algorithms for normal and uniform distributions of data points. J. Comput. Sci. 6 (3), 363.
Vincent, P., Larochelle, H., Bengio, Y., Manzagol, P.-A., 2008. Extracting and composing robust features with denoising autoencoders. In: Proceedings of the 25th International Conference on Machine Learning, pp. 1096–1103.
Vishwanath, K.V., Nagappan, N., 2010. Characterizing cloud computing hardware reliability. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 193–204.
Voas, J., Ghosh, A., Charron, F., Kassab, L., 1997. Reducing uncertainty about common-mode failures. In: Proceedings the Eighth International Symposium on Software Reliability Engineering. IEEE, pp. 308–319.
Wolter, K., Avritzer, A., Vieira, M., Van Moorsel, A., 2012. Resilience assessment and evaluation of computing systems. Springer.
Wu, L., Bogatinovski, J., Nedelkoski, S., Tordsson, J., Kao, O., 2020. Performance diagnosis in cloud microservices using deep learning. In: AIOPS 2020-International Workshop on Artificial Intelligence for IT Operations.
Xie, J., Girshick, R., Farhadi, A., 2016. Unsupervised deep embedding for clustering analysis. In: International Conference on Machine Learning. pp. 478–487.
Xiong, H., Wu, J., Chen, J., 2009. K-means clustering versus validation measures: a data-distribution perspective. IEEE Trans. Syst. Man Cybern. B 39 (2), 318–331.
Xu, Z., Dang, Y., Munro, P., Wang, Y., 2020. A data-driven approach for constructing the component-failure mode matrix for FMEA. J. Intell. Manuf. 31 (1), 249–265.
Xu, R., Wunsch, D., 2005. Survey of clustering algorithms. IEEE Trans. Neural Netw. 16 (3), 645–678.
Xu, Z., Zhang, T., Keung, J.W., Yan, M., Luo, X., Zhang, X., Xu, L., Tang, Y., 2021. Feature selection and embedding based cross project framework for identifying crashing fault residence. Inf. Softw. Technol. 131, 106452.
Yang, B., Fu, X., Sidiropoulos, N.D., Hong, M., 2017. Towards k-means-friendly spaces: Simultaneous deep learning and clustering. In: International Conference on Machine Learning. PMLR, pp. 3861–3870.
Zhang, W., Dong, X., Li, H., Xu, J., Wang, D., 2020. Unsupervised detection of abnormal electricity consumption behavior based on feature engineering. IEEE Access 8, 55483–55500.
Zhao, W., Melliar-Smith, P., Moser, L.E., 2010. Fault tolerance middleware for cloud computing. In: 2010 IEEE 3rd International Conference on Cloud Computing. IEEE, pp. 67–74.

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 is a Ph.D. student 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.