2019 IEEE 12th International Conference on Cloud Computing (CLOUD)

An Approach to Cloud Execution Failure Diagnosis Based on Exception Logs in OpenStack

Yue Yuan, Wenchang Shi, Bin Liang and Bo Qin
School of Information, Renmin University of China, Beijing, China
{yuanyue,wenchang,liangb,bo.qin}@ruc.edu.cn

***Abstract*—Cloud is getting ubiquitous and scales up rapidly. It is critical to effectively detect and efficiently repair system anomalies for a robust cloud. Many efforts have been made to facilitate analysis of system problems with the readily-available and massive cloud logs. However, most tools can still not automatically recognize failures related to a specific cloud operating system task. To diagnose execution failures of a cloud, it is inevitable to monitor corresponding system tasks. In this paper, we propose a lightweight approach to identify cloud behaviors related to failed executions of the cloud operating system for failure diagnosis, by exploiting logs of ERROR logging level in a cloud. Instead of working on execution sequences extracted from logs for all system tasks, we focus on automated recognition of exception logs generated by a system task. These logs are critical snippets of execution traces for failure diagnosis of a cloud. In our work, exception logs are extracted and associated with the respective system task. Efforts can be reduced by comparing patterns of new error cloud behaviors with cloud behaviors met before. With experiments on OpenStack, a popular open source cloud operating system, we demonstrate that our work is effective and efficient for execution failure diagnosis of a cloud. Our approach can also be used as a complementary method for log-based troubleshooting tools concentrating on execution sequences.**

***Index Terms*—IaaS cloud, failure diagnosis, log analysis**

I. Introduction

A cloud operating system serves as the middle layer for users to interact with the underlying infrastructures. It directly communicates with cloud users, and plays an essential role in a cloud system. Open source cloud operating systems such as OpenStack [1] provide convenient interfaces to access infrastructure resources such as to create a VM. Like any distributed system, clouds are susceptible to failures. In addition, external cloud service requests are often accomplished by cloud operating system tasks with coordination of different cloud service components. Various drivers and plugins of each component are provided for backend implementations. Compared with traditional computer systems, clouds are getting increasingly complex. Moreover, flexible system configurations are allowed for various requirements of cloud service providers, and numerous optional input parameters can be used by cloud users, which in turn increases the possibility of cloud failures. Failure diagnosis is both complicated and time consuming, even for skilled engineers. Failures in clouds deployed on OpenStack can take hours and days to fix [19]. Execution failures of a cloud operating system can directly impact users (including applications developed on the cloud) accessing relevant resources, and thus may cause significant damages to cloud users both in monetary and non-monetary terms [4]. Therefore, failure diagnosis is both urgent and challenging towards building a robust cloud.

Before a cloud platform is officially used in a production environment, plenty of tests with numerous test cases are conducted in a simulated environment to ensure the normal operation of the cloud. As a large scale cloud operates in the production environment, once failures are reported, they are often diagnosed with execution logs of the cloud system. System logs are widely utilized by engineers to ensure system dependability [5]. Nevertheless, in a cloud, different distributed service daemons of multiple components generate relevant logs recording their execution traces in the respective log file. A large volume of interleaved logs distributed on numerous server nodes are produced. It is hard to spot the critical clues for execution failure diagnosis of the cloud operating system. Faced with the big data of informative logs, many researchers have concentrated on automatic analysis of system logs to gain operational insights of the system.

As mentioned in a recent study [8], log analysis for anomaly detection and diagnosis in computer systems has been widely studied and it is still an active research field, in particular when considering the ever more complex computer systems. Recently, many tools like [14] have focused on analysis of execution sequences extracted from system logs. On the other hand, a majority of the production failed executions in today's distributed systems output error log messages that can be used to characterize the failure [6]. A recent tool CAM [12] can find test alarm causes based on test exception logs in system and integration testing. Moreover, Leaps [15] analyzes the API sequences extracted from logs for cloud security auditing, where cloud APIs are mapped to events extracted from logs, such as using a log event to identify the VM creation task in a cloud. In this paper, we propose a lightweight approach to identify cloud behaviors related to failed executions of the cloud operating system for failure diagnosis with exception logs of a cloud. We focus on logs of ERROR logging level generated during each system task execution, which are critical snippets of the execution traces for failure diagnosis of a cloud operating system. Relevant exception logs are extracted and associated with the log event identifying a system task.

Instead of roughly analyzing all exception logs, we identify exception logs as a part of workflow trace related to a system task in question. An example scenario is to seek the answer to a question of what are the causes of each failure in the

2159-6190/19/$31.00 ©2019 IEEE
DOI 10.1109/CLOUD.2019.00031

numerous VM creation requests of users, in a cloud serving varieties of services like VM deletion and VM live migration in parallel. We would automatically extract exception logs related to the respective system task execution handling each VM creation request, and automate the diagnosing process for each failure. However, *in a cloud, exception logs from a server node may be generated by a system task initiated on another server node, a system task can generate exception logs spanning multiple server nodes and logs related to different tasks are interleaved.* It is not easy to associate and analyze exception logs for failure diagnosis in a cloud.

We facilitate the diagnosing process by taking full advantage of historical failures, such as failures occurring in the test stage before a cloud is officially used. Patterns of exception logs related to different failures of a system task to describe error cloud behaviors are figured out from limited samples extracted from past runs of the cloud system. These patterns generally characterize relevant cloud behaviors resulted from failed executions of the cloud operating system. In this way, new failures can be automatically recognized for diagnosis. Furthermore, relevant typical information is presented as clues for further detailed diagnosis, including information to identify involved server nodes and key identifiers to associate relevant logs of other logging levels.

Above all, we make the following contributions:

- We associate exception logs with the respective system task in a cloud, where relevant logs related to a system task in question are extracted from various log files spanning all server nodes in the cloud for root cause analysis of corresponding system execution failures.
- We facilitate the root cause analysis for each system task in question by comparing relevant logs with the historical exception logs, where we model the exception logs using a recent natural language processing tool word2vec [7] for distributed representation in a metric space.
- We validate our approach on realistic log data in OpenStack. The experiment results show the capability our approach for cloud failure diagnosis.

This paper is organized as follows. Background is presented in Section II. Our approach is detailed in Section III and followed by experiment evaluations in Section IV. Related work is discussed in Section V. The paper is finally concluded in Section VI.

II. Background

A. OpenStack

OpenStack is an open source distributed cloud operating system implemented in Python. Currently hundreds of organizations develop clouds based on OpenStack. It consists of multiple service components, including four core components of Keystone for authentication, Nova for computing, Glance for image, and Neutron for networking. Each service component comprises multiple running service daemons which may distributed on different server nodes. Service daemons communicate with each other via remote procedure calls (RPCs) using Advanced Message Queuing Protocol (AMQP). Typically, four types of server nodes are involved, namely controller nodes, compute nodes, networking nodes and storage nodes. These types can be cooperated into one server node as needed. OpenStack manages VMs by invoking interfaces offered by different infrastructures, like hyperviors of Libvirt, Xen, etc, on compute nodes through different drivers. Key data related to cloud resources is stored in a central database on the controller node. Users can access cloud infrastructure resources through APIs provided by each service component. As the OpenStack community continues to grow, progressively increasing service components have been developed.

To accomplish an external request in each service component, multiple distributed service daemons are usually involved in the task execution of the cloud operating system. Because of concurrent mechanisms and fault tolerance, a failed system execution may be associated with numerous exception logs across several server nodes where the real root cause information may be buried.

Moreover, in OpenStack, there are fault notifications indicative of some failures, but they can often be misleading [18]. Furthermore, in some scenarios, these fault notifications may even be absent altogether, forcing developers/operators to mine the system logs to determine the actual source of the problem [19].

B. Logging in OpenStack

As a Python-based system, OpenStack implements its shared logging function based on standard logging packages in Python. Developers place logging statements at various critical points in the source code. System logs, especially for service systems, record important information including context information identified by request identifiers about user requests. In OpenStack, logs related to a common session can be associated by request ID recorded in each log entry. Typically, as shown in Fig. 1, OpenStack logs of ERROR logging level, with no context information part, record tracing back information of latest function calls about a raised exception.

The timestamp, logging level, request identifier, and message in a log entry as shown in Fig.1 are exploited in our work. Note that, in this paper, error logs refer to logs with logging level of ERROR, exception logs refer to error logs resulted from a raised exception, tracing logs refer to error logs with no context information part, logs related to an execution of a system task refer to logs with the same request identifier, a log entry refers to a log line and log content refers to the message part of a log entry.

```
logging_context_format_string = %(asctime)s.%(msecs)03d %(process)d %(levelname)s %(name)s
                                [%(request_id)s %(user_identity)s] %(instance)s%(message)s

logging_exception_prefix = %(asctime)s.%(msecs)03d %(process)d ERROR %(name)s %(instance)s
```

Fig. 1: Default configurations of logging format in OpenStack.

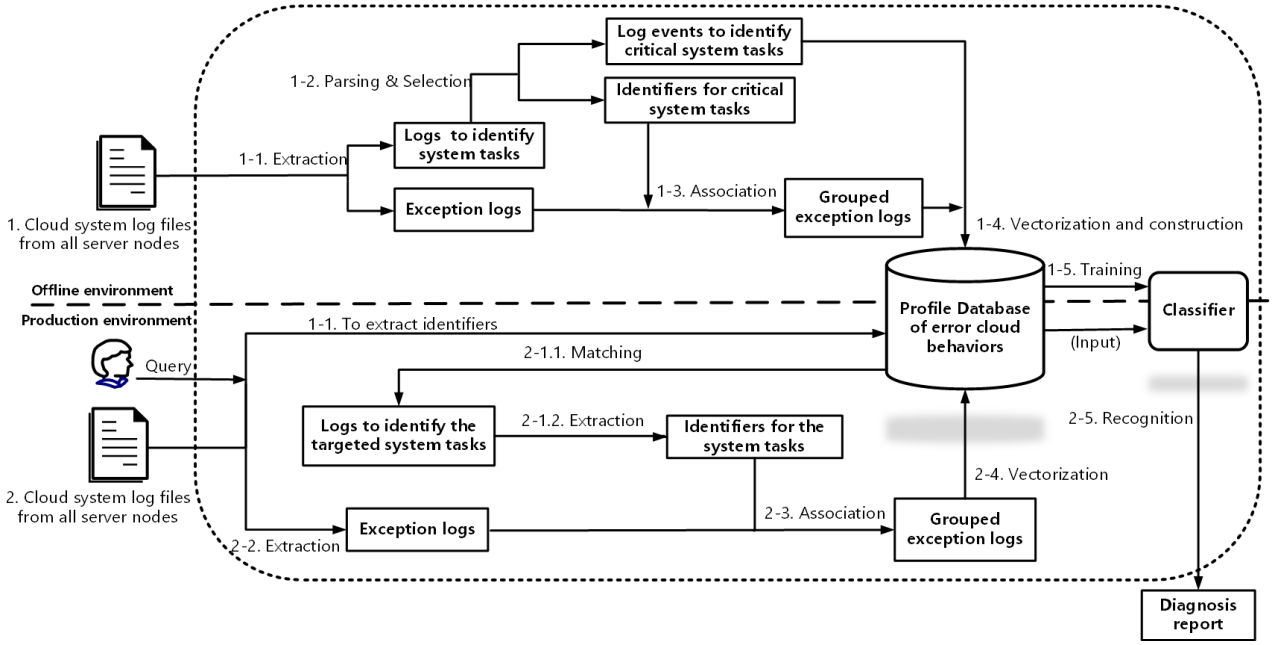


Fig. 2: Overview of the approach.

III. Approach

A. Overview

As shown in Fig. 2, the overall working process is divided into two phases: the offline phase in offline environments and the production phase in production environments. In the offline phase, we extract and group exception logs related to predefined critical system tasks. Each group is relevant to a specific execution of a system task. Then logs in each group are vectorized and stored in the Profile Database. Finally, we train a classifier for each system task based on the Profile Database. In the production phase, query information of a certain type of system task such as VM creation is provided. Exception logs related to the system task in question are extracted and grouped. After vectorization with the help of Profile Database, each group of exception logs related to an execution of the system task would be labeled with corresponding failure types by the related classifier for the system task, and relevant solutions would also be recommended.

B. Preprocessing of logs for system task identification

Firstly, critical system tasks are defined in the offline phase. Similar to log parsing [5], which abstracts structured events from unstructured raw logs, we extract regular expressions as events from logs to describe system tasks. Specifically, each service component in OpenStack provides independent API to handle external requests. Thus, we use all the first log entries related to each request identifier to identify each system task, each of which represents the entry point of the service program to handle the request. We abstract regular expressions from these log entries. Each regular expression corresponds to an event identifying a system task. Then, one can decide the critical system tasks to monitor, and select the corresponding events.

To be more specific, given n log entries $L =< l_1, l_2, ..., l_n >$, we get the first log entry, which has the minimal timestamp, related to each request identifier and parse these log entries into events:

$$T_{log} = \{(r_i, e(l_i))|1 < i < m, l_i \in L\}$$

where r_i is the request identifier extracted from logs, l_i is the first related log entry, $e(l_i)$ is the event parsed from l_i, and m is the total of request identifiers appearing in L. All request identifiers are $R = \{r_i|1 < i < m\}$. All parsed events are $E = \{e_i|1 < i < p\}$, where e_i is each event representing each system task and p is the total of events. One can then select the events, related to critical system tasks to monitor, from E.

Afterwards, we can collect all request identifiers relevant to the same event e_i to form a data set:

$$R_i = \{r_{i_j}|1 < j < p_i, r_{i_j} \in R\}$$

where r_{i_j} is each request identifier related to event e_i, and p_i is the total of request identifiers related to e_i. In this way, after extracting the logs matching e_i in the production phase, we can get all request identifiers R_i related to any predefined system task described by e_i .

C. Association and vectorization of exception logs

As shown in Fig. 3, logs related to an exception identified by a request identifier mainly consist of tracing information and error messages. The tracing information represents the latest function calls related to the exception, and the error messages directly describes the exception. For a specific execution of

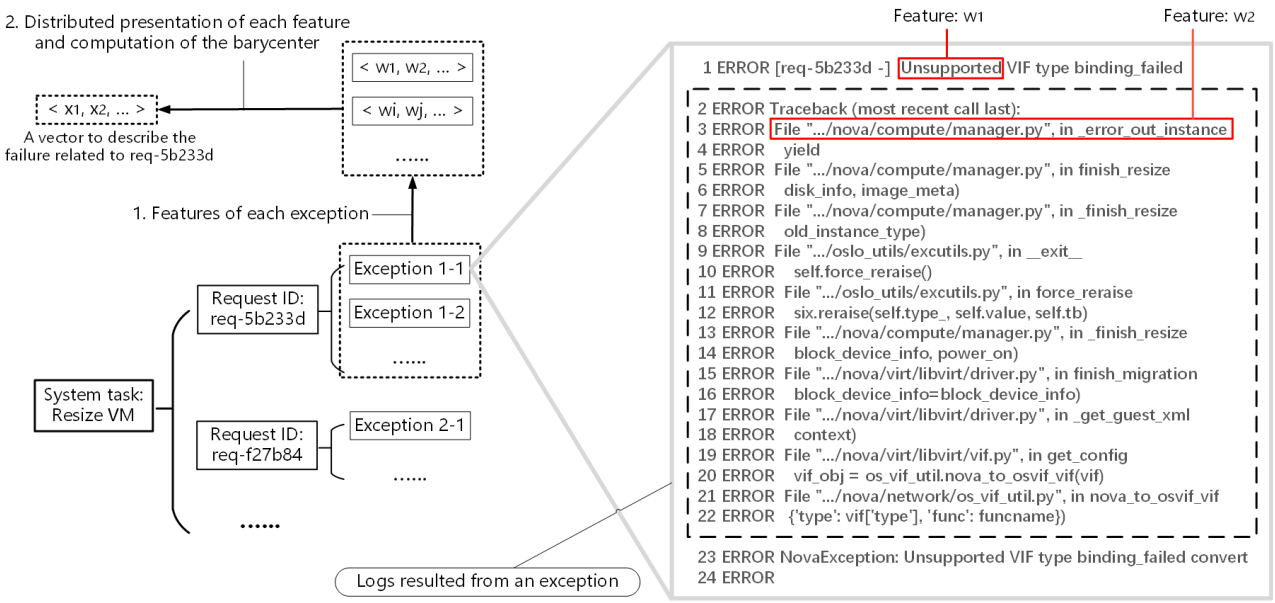


Fig. 3: An simple example of the vectorization of grouped exception logs. In the logs resulted from an exception, logs with the dashed box are tracing information and the rest are error messages.

a system task identified by a request identifier r_i, we get all relevant exception logs. We also record the server nodes where these logs originate. Then we form a data set EX_i to describe the exceptions related to the request identifier r_i:

$$EX_i = \{(v_{i_j}, h_{i_j}) | 1 < j < q_i, v_{i_j} =< l_{i_{j1}}, l_{i_{j2}}, ..., l_{i_{jend}} >\}$$

where v_{i_j} represent logs resulted from each exception identified by r_i, h_{i_j} is the server node where the exception logs are extracted, q_i is the total of exceptions raised in the execution of the system task, and $l_{i_{jk}} \in L$ is each log entry resulted from the exception described by v_{i_j}. Note that h_{i_j} is extracted just in the production phase and this information about server nodes would be provided as the final additional diagnosis information. Moreover, $l_{i_{jk}}$ can be either tracing information representing a function call or an error message describing the exception.

After association of the logs, as shown in Fig. 3, we then treat each line of the tracing information and each word in the error messages as features for any exception described by v_{i_j}:

$$w_{i_{j_k}} = \begin{cases} l_{i_{j_k}} & \text{if } l_{i_{j_k}} \text{ is tracing information} \\ \text{each word in } l_{i_{j_k}} & \text{if } l_{i_{j_k}} \text{ is an error message} \end{cases}$$

where $w_{i_{j_k}}$ is a feature of the exception described by v_{i_j}. In this way, we convert EX_i into a exception feature data set:

$$C_i = \{< w_{i_{j_1}}, w_{i_{j_2}}, ..., w_{i_{j_{end}}} > | 1 < j < q_i\}$$

where each element in C_i represents an exception v_{i_j} in EX_i identified by request identifier r_i, and there are q_i elements in C_i.

Afterwards, we get a corpus related to a system task instantiated by the request identifiers R_i extracted in SectionIII-B:

$$COR(R_i) = \{< w_{k_1}, w_{k_2}, ..., w_{k_{end}} > | 1 < k < \sum_{j=1}^{p_i} q_{i_j}\}.$$

where each element in $COR(R_i)$ represents the features of an exception, p_i is the total of request identifiers related to the system task, and q_{i_j} is the total of exceptions raised during the system execution of each request $r_{i_j} \in R_i$. Based on $COR(R_i)$, we use the word2vec [7] algorithm to train and get the distributed presentation of any feature w_{k_j} of an exception, where each feature w_{k_j} is treated as a word and each element in $COR(R_i)$ describing an exception is treated as a natural language sentence.

Then for each request identifier r_i related to the system task, we get a vector $\boldsymbol{x}_i$ by calculating the barycenter (similar to [8]) of the corresponding exception data set C_i. To be more specific, the coordinate of each element representing an exception in C_i is first computed as the average position of all $w_{k_{j_k}}$ the element contains, and the coordinate of $\boldsymbol{x}_i$ is then computed as the average position of all elements in C_i.

Finally, we get a data set of vectors for a system task instantiated by the request identifiers R_i :

$$D(R_i) = \{(r_{i_j}, \boldsymbol{x}_{i_j}) | 1 < j < p_i\}$$

where r_{i_j} is each request identifier describing an execution of the system task, $\boldsymbol{x}_i$ is the distributed representation of exceptions raised during the system execution, and p_i is the total of executions of the system task.

TABLE I: Description of the Profile Database of error cloud behaviors.

Constituent	Description
Event	A regular expression to identify a critical system task.
COR	A set of sequences related to a critical system task where each sequence consists of features describing an exception and is transformed from logs related to the exception.
Word2vec model	A model related to a critical system task trained from COR.
D	A set of vectors related to a critical system task where each vector describes an failed system execution of the task and is mapped from relevant exception logs by the word2vec model.
Labels	Failure types and relevant solutions which are attached to each element of D.

D. Construction of the Profile Database

The Profile Database of error cloud behaviors as shown in Fig. 2 is constructed in the offline phase. After monitoring error cloud behaviors of each critical system task, we collect the corresponding log files and extract essential information from the logs to construct a Profile Database. As shown in Table I, for each critical system task, all relevant features $COR(R_i)$ describing the respective exception, the trained word2vec model, all vectors $D(R_i)$ describing each execution of the system task as mentioned in SectionIII-C and the event to describe the system task are stored to fill the Profile Database. Moreover, labels are attached to each request object in $D(R_i)$ according to the corresponding scenario (e.g., labels can be attached according to the targeted scenario after failure injection similar to [8], [11], [17]). These labels are also stored in the database. Note that error cloud behaviors can recur by failure injections, where failures are injected into the system and cloud user operations related to the system task are replayed.

Furthermore, once new failures are addressed, relevant logs can be integrated into the Profile Database for further failure analysis. Features of the new grouped exception logs are added. The word2vec model is updated. New vectors with the labels to describe the cloud failures are inserted.

Finally, based on the Profile Database, we train a classifier for each critical system task, using $D(R_i)$ and the corresponding labels as the training set. In this way, once the training is finished, the resulting classifier can be used to provide the diagnose result for new system task executions.

E. Report generation

In the production phase, for any targeted system task in question, relevant request identifiers are extracted with the help of the predefined event in the Profile Database. To be more specific, critical logs matching the regular expression of the event to identify the task are kept. Identifiers are then extracted from the critical logs. After that, relevant exception logs related to each identifier are associated, and then vectorized by the word2vector model. In this way, a data set $D(R_i)$ as mentioned in SectionIII-C is constructed, where each object in $D(R_i)$ is a vector to describe a system task execution identified by a request identifier. For each request associated with the targeted system task, we use the classifier related to this system task to recognize the each vector. A label is then attached to each vector, indicating the failure type. Additionally, relevant server nodes involved in the system execution to handle each request are also provided to ease the manual inspections. Finally, information as shown in Table II are filled as the final report.

TABLE II: Diagnosis Report

Field	Description
System Task	a predefined task name
Request Identifier	extracted from logs to identify a task execution
Failure Type	diagnosis result to identify the cloud behavior
Addition Information	recommended treatment
Server Nodes	involved server nodes

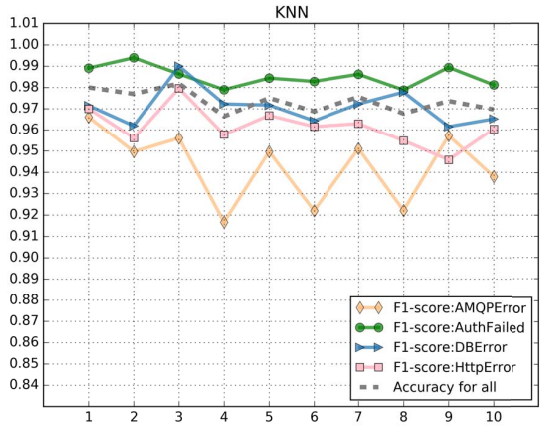
IV. Experiment and Evaluation

In this section, experiment settings are first described, and experimental results are then presented. Finally, comparisons with existing work are discussed.

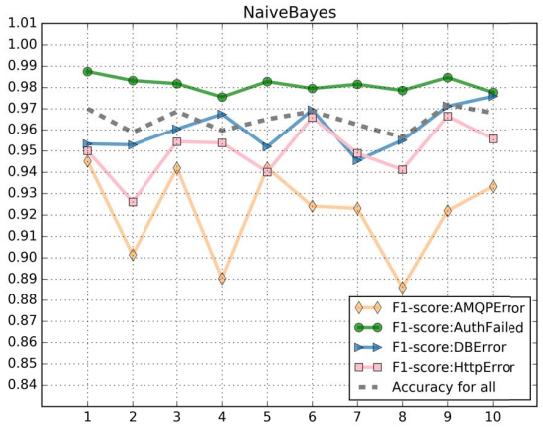
Experimental settings. The test cloud is based on OpenStack version Ocata, consisting of one controller node and 50 compute nodes. The logging level of OpenStack is set to DEBUG. To emulate the load, a total of 4500 cloud user operation requests of VM creation and numbers of VM deletion and VM migration requests are submitted concurrently through command line interfaces. In the experiment, we targeted on the system task related to VM-create requests. Four types of failures are injected at the respective execution point: 1) Neutron timeout during network resource provision; 2) authentication failure while fetching an image; 3) DB connection exception when saving VM states; 4) AMQP error during communications between service daemons. We collect the OpenStack logs from all server nodes as the data source for later analysis. In addition, we implement our approach in Python. We employ Gensim [3] for implementation of the word2vec algorithm, and Scikit-learn [2] for implementation of the classification algorithms.

Results. The main objective of the experiments is to demonstrate the capability of our approach for cloud failure recognition. F1-score, which is the harmonic mean of the precision and recall, of each failure type and the overall accuracy are used to evaluate the model. All the 4500 requests with system execution failures in the experiment are identified from logs. Relevant exception logs are extracted and associated with the respective request. All non alphanumeric characters are removed from the exception logs.

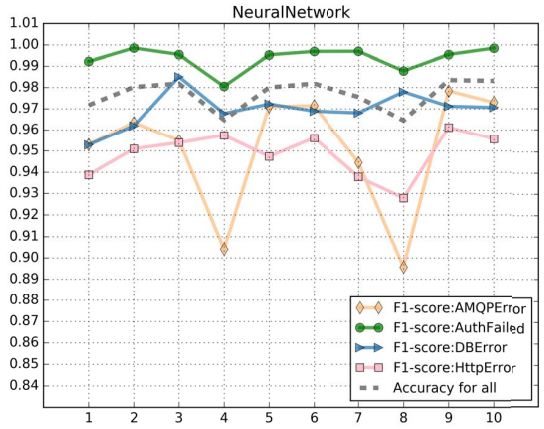
We first check the result after elimination of the requests associated with identical exception logs for better understandings of the dataset. 1046 requests, around 23% of the total requests, are retained after the elimination. Then we use the hold-out validation approach with random sampling to split the 1046 requests into two data sets. In the experiment, 40% of the data is used as a test set and the rest is a training



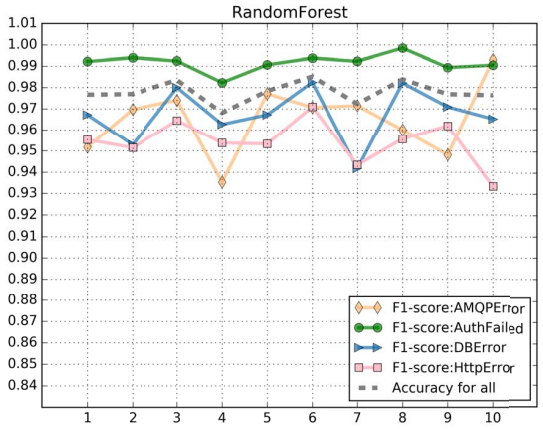
(a) Evaluation result of k-NearestNeighbor.



(b) Evaluation result of Naive Bayes.



(c) Evaluation result of Neural Network.



(d) Evaluation result of Random Forest.

Fig. 4: Results of F1-score and accuracy for different classifiers where the hold-validation with random sampling is repeated for 10 times.

set. Moreover, for each hold-out validation, we select four simple and state of art supervised machine learning algorithms to implement the classifiers: k-NearestNeighbor, Naive Bayes, Neural Networks, and Random Forests, with Scikit-learn library implementations: KNeighbors Classifier, GaussianNB , MLP Classifier and RandomForest Classifier. We compare the evaluation results of the four kind of classifiers, and assess the global strategy as a hole.

Fig. 4 shows the evaluation results of the four classifiers, and each sub figure shows the F1-scores of each failure type and the overall accuracy. The model performs good enough, with Random forest exhibiting a strong recognition ability (the lowest overall accuracy of 96.45% to identify each failure type in the experiment). While the F1-score result for authentication failure achieves good performance for all classifiers (with Random Forest achieving the highest 99.84%), the result for AMQP error is less precise (with Naive Bayes reaching the lowest point of 88.61%). All in all, these results confirm the soundness of our approach.

Comparisons with the existing work. We compare our approach against four most related machine leaning based log analysis approaches. Table III presents the comparison results. Three aspects are considered: the target, the main preprocessing tool to handle the logs, and whether a system task can be identified. In what follows, we discuss more the three aspects in detail.

Firstly, Deeplog and [8] do not distinguish between system task executions for anomaly detection. CAM focus on test alarms. Since logs related to each test alarm are recoded respectively, there is no need to use other techniques to group logs. Moreover, LogCluster and our work both target on each system task execution, and do a fine-gained analysis. Logs related to each system execution need to be extracted and grouped together. We adopt the similar principle to LogCluster

TABLE III: Comparisons with the related work.

Proposal	Objective	Target	Preprocessing	System task identification
LogCluster [14]	problem diagnosis	each system task execution	log parser	No
CAM [12]	problem diagnosis	each test alarm	TF/IDF	No
Deeplog [11]	anomaly detection	overall system executions	log parser	No
Bertero et al. [8]	anomaly detection	overall system executions	word2vec	No
Our work	problem diagnosis	each system task execution	word2vec	Yes

for log extraction. LogCluster use taskID for Hadoop clouds, and we use the request identifier to extract and group the logs for OpenStack.

Secondly, LogCluster and Deeplog exploit log parsing tools to convert each log entry into a log event. After parsing, the logs are converted into a log event sequence. Each log event is an element of the log sequence, like a word is an element of a sentence. However, when faced with a huge amount of logs, to parse each log entry can be extremely time consuming. CAM, [8] and our work employ natural language processing tools to convert logs into structured data for training set construction. In the experiment, with our approach, it takes about 37 minutes to extract 4500 VM creation requests and their associated exception logs from a total of approximately 93 gigabytes log data, and convert them into vectors. Nevertheless, using the other four tools, it take hours to preprocessing the logs. LogCluster as well as Deeplog need to parse each log entry, and CAM as well as [8] need to handle each word, whereas we just need to handle each word of the extracted exception logs related to the targeted system task. In addition, for a smaller log data of 2.1 gigabytes, with ou approach, it just takes around 3.4 seconds to extract 600 failed VM live migration requests and their associated exception logs, and convert them into vectors.

Finally, we can identify system tasks. In cloud environments like OpenStack, questions like why VM creation failed can sometimes be raised, especially in a test environment for the test of a specific system task (like VM creation task) involved with numerous user cases. Failures related to the target system task need to be identified and relevant logs should be extracted. The problem diagnosis tools LogCluster and CAM treat all failure types as a whole and do not distinguish between logs of different tasks. With our approach, we can identify logs related to the target task, by exploiting the first log entry related to each request identifier. Moreover, we train a classifier for failures of each system task, instead of constructing a learning model for all failures like LogCluster and CAM. In this way, we do not depend on the classifier to distinguish failures of different tasks.

V. Related Work

Although many studies have proposed new methods to parse unstructured logs into structured events for further automatic log analysis, log files contain a large variety of log events as different complex system tasks are handled by the system and log parsing is still intricate as well as time consuming. Recent work [5] provides several useful log parsing tools. In our work, instead of parsing all the massive logs, we focus on extract events from several logs recording critical execution points which can identify different system tasks. Existing log parsing tools can all be used in our framework.

Automated log analysis for anomaly detection and diagnosis in computer systems has been widely studied. For example, [13] presents a lightweight approach for uncovering differences between execution logs of pseudo and large-scale cloud deployments to assist developers of debugging system problems. Recently, many new log mining tools are proposed, such as Log-cluster [14] introduces a clustering-based approach to ease problem identification in large-scale online service systems. Some tools like [4], [8], [10]–[12], [15], [20] make full use of supervised machine learning techniques. [4] presents a regression-based approach to detect errors in the execution of cloud system operations, in particular focusing on rolling upgrade operations. [8] targets on minimal human intervention for log file processing. DeepLog [11] develops a deep neural network model for system anomaly detection. LeaPS [15] aims to facilitate the proactive security auditing of cloud operations by extracting dependency models from system runtime events making use of Bayesian network. CAM [12] exploits information retrieval techniques to efficiently infer test alarm causes based on test logs, which relieves the burden to manually analyze the causes of test alarms in software testing. [20] detects attacks at an early stage using a belief propagation framework.

In addition, many have concentrated on developing workflow models to monitor system behaviors. For instance, CSight [9] mines logs of system executions to infer a model of that system's behavior and helps engineers monitor concurrent systems. CloudSeer [17] enables effective workflow monitoring

by building an automation based on normal system executions of a cloud. Moreover, [16] forms a control flow graph based on system execution logs spanning distributed components and detect anomalous run-time behavior of distributed applications. [21] develops Finite State Automation models mined from logs to monitor the system.

Compared with above work, we focus on exception logs related to each system task execution rather than taking the overall logs produced by all system tasks as a whole. Moreover, we focus on digging the causes of cloud failures related to each system task in question instead of roughly anomaly detection or anomaly analysis of the overall system.

VI. Conclusion

In this paper, we propose an automatic log-based method to identify cloud behaviors resulted from failed executions of the cloud operating system for failure diagnosis by comparison of relevant exception logs with the historical logs. Failures are recognized and related solutions are recommended for system recovery. Different from previous studies parsing all logs into structured events, we just parse a few critical logs each of which records an entry execution point of a cloud service program. Moreover, we uniquely focus on exception logs associated with system tasks in question. We model these exception logs as natural languages using a recent natural language processing tool for feature representation, so that exception log pattern related to the same failure type can be recognized. We conduct experiments on OpenStack, and apply our approach to realistic log data of OpenStack. Experimental results demonstrate the capability of our approach to identify error cloud behaviors for execution failure diagnosis of a cloud. In addition, once the profile database to describe error cloud behaviors in our approach is constructed, it can also be used for failure diagnosis in tests before the cloud is officially used. Furthermore, our method can also be used for failure diagnosis of internal tasks of OpenStack, such as periodic task which achieves fault resilience in the cloud.

Acknowledgment

This work was supported in part by the National Nature Science Foundation of China under grant NO. (61472429, 61070192, 91018008, 61303074, 61170240).

References

[1] OpenStack. Openstack: Open source software for creating private and public clouds. http://www.openstack.org.

[2] Scikit-learn. Scikit-learn: Machine learning in python. http://scikit-learn.org/

[3] Gensim. Gensim: topic modeling for humans. https://radimrehurek.com/gensim/

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

[5] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R. Lyu. Towards Automated Log Parsing for Large-Scale Log Data Analysis. IEEE Transactions on Dependable and Secure Computing. 15(6): 931-944, 2018.

[6] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17. 2017.

[7] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. Advances in Neural Information Processing Systems, 26:3111-3119, 2013.

[8] Christophe Bertero, Matthieu Roy, Carla Sauvanaud and Gilles Tredan. Experience report: Log mining using natural language processing and application to anomaly detection. In Proceedings of the 2017 IEEE 28th International Symposium on Software Reliability Engineering, ISSRE'17, 2017.

[9] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In Proceedings of the 36th International Conference on Software Engineering, ICSE'14. 2014.

[10] Rui Ding, Qiang Fu, Jian Guang Lou, Qingwei Lin, Dongmei Zhang, and Tao Xie. Mining historical issue repositories to heal large-scale online service systems. In Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN'14. 2014.

[11] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17. 2017.

[12] He Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. What causes my test alarm?: Automatic cause analysis for test alarms in system and integration testing. In Proceedings of the 39th International Conference on Software Engineering, ICSE '17. 2017.

[13] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Bram Adams, Ahmed E. Hassan, and Patrick Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13. 2013.

[14] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. In Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16. 2016.

[15] Suryadipta Majumdar, Yosr Jarraya, Momen Oqaily, Amir Alimohammadifar, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. Leaps: Learning-based proactive security auditing for clouds. In Proceedings of the European Symposium on Research in Computer Security, Cham, ESORCS'17, 2017.

[16] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B. Dasgupta, and Subhrajit Bhattacharya. Anomaly detection using program control flow graph mining from execution logs. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16. 2016.

[17] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. In Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16. 2016.

[18] Dhruv Sharma, Rishabh Poddar, Kshiteej Mahajan, Mohan Dhawan and Vijay Mann. Hansel: diagnosing faults in OpenStack. In Proceedings of the 11th International on Conference on Emerging Networking EXperiments and Technologies, CoNEXT'15, 2015.

[19] Ayush Goel, Sukrit Kalra, and Mohan Dhawan. Gretel: Lightweight fault localization for openstack. In Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies, CoNEXT '16. 2016.

[20] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H. Chin, and Sumayah Alrwais. Detection of early-stage enterprise infection by mining large-scale log data. In Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN'15. 2015.

[21] Maayan Goldstein, Danny Raz, and Itai Segall. Experience report: Log-based behavioral differencing. In Proceedings of the 2017 IEEE 28th International Symposium on Software Reliability Engineering, ISSRE'17, 2017