LogDC: Problem Diagnosis for Declartively-deployed Cloud Applications with Log

Jingmin Xu *†, Pengfei Chen †, Lin Yang †, Fanjing Meng †, Ping Wang *
* School of Software and Microelectronics, Peking University,
† IBM Research China
{xujingm, cpfchen, ylyang, mengfj}@cn.ibm.com, pwang@pku.edu.cn

***Abstract*—Recently, as the evolution of application's development and management paradigms, the deployment declaration becomes a standard interface connecting application developers and Cloud platforms. Kuberenetes is such a system for automating deployment, scaling, and management of micro-service based applications. However, managing and operating such a cloud benefit with additional complexities from the declarative deployment. This paper proposes a log model based problem diagnosis tool for declaratively-deployed cloud applications with the full lifecycle Kubernetes logs. With the runtime logs and deployment declarations, we can pinpoint the root causes in terms of abnormal declarative items and log entries. The advantage of this approach is that we provide a precise log model of a normal deployment to help diagnose problems. The experimental results show that our approach can find out the anomalies of some real-world Kubernetes problems, some of which have been confirmed as bugs. Within the given fault types, our approach can pinpoint the root causes at 91% in Precision and at 92% in Recall.**

***Index Terms*—Cloud; Log analysis; Kubernetes; Declarative deployment;**

I. Introduction

With the fast adoption of Cloud-native applications in e-Business, the application's development and management paradigms have evolved in many aspects, such as breaking down monolithic blocks into micro-services, automated deployment of distributed components, rapid horizontal scaling for resilience, to name a few. The growth of Cloud platform's capability reflects this shift in the application domain. In this mutual evolution of application and Cloud, deployment declaration becomes a standard interface connecting application developers and Cloud platforms [1,2]. Mainstream Cloud platforms claim to provide automation, scalability, agility, and easy usage to achieve application's business goals. To leverage these Cloud functionalities, developers need to describe the desired application structure and state. We observe that composing declaration becomes the de-facto manner in Cloud-native application development such as OpenStack [3], and Kubernetes [4].

However, managing and operating such a cloud benefit with additional complexities from the declarative deployment. Less coding in development phase usually means more troubles during in running phase. It is a double-edged sword for both Cloud operator and application developer. In our previous work [5], we pointed out that being more distributed and complex makes it more difficult to find the root cause of a failure in the cloud. Because more dynamics (e.g., autoscaling, automatic restart, rollout) are introduced by the declarative deployment mechanism may cover the truth and interfere our judgements of system problems. To tackle this problem, we proposed a log based approach as log plays an irreplaceable role in problem diagnosis as it can reflect the execution states of the system with intelligible texts [6]. More precisely, we use historical logs generated for certain type of operations to build a "reference model" that represents the normal behavior of the system. When the system fails to process the same type of operation, the operator can compare the logs collected for the failed operation with the reference model, and deduce what the problem might be, based on their deviation of the current logs from the reference behavior. Actually, a large number of studies have been done on log-based diagnosis.

Although the previous log-based diagnosis approach can help operators perform problem diagnosis quickly and effectively. However, it also needs to be updated to adapt to the cloud platform. **First**, as Cloud is shifting from resource oriented to application oriented. Cloud operations' varieties have increased to a higher level. We are no longer able to precisely build log-based reference model for a specific application, like we did for resource operations. Instead, we ought to generalize the approach to build a reference model for a kind of applications. **Second**, as declaratively-deployed application's desired state is maintained by the Cloud platform throughout its lifecycle, application's behavior has more diversities and dynamics, like immediate restart and auto-scaling. To effectively diagnose problems, these behaviors should be captured by the log reference model. Also, the log model out to be extended to the full lifecycle of an application.

To address the above issues, the methodology needs to be redesigned and the system needs to be reinvented. In this paper, we propose LogDC, a log model based problem diagnosis tool for declaratively-deployed Cloud applications. The motivation of LogDC is to offer intuitive "compare and find" experience in the previously tedious and difficult troubleshooting process of distributed applications. The construction of reference models is purposely designed as low-touch as possible and machine learning techniques are leveraged in multiple aspects. More specific, LogDC addresses following key challenges, namely i). Build the reference model for one class of applications; ii). Identify normal behaviors through an application's lifecycle.

In LogDC, we devise solutions to each of the problems described above. For classifying applications, we design novel method to extract features from application's deployment declaration and train classifier based on them. For model building, we leverage the fact that in a stable Cloud platform, the majority of deployed workloads are in their desired state, and collect normal samples based on clustering algorithms. By reducing the dependencies on the log entry's chronologic order during reference model's constructing and matching phases, LogDC is capable to diagnose problems in an application's full lifecycle.

We verify our approach on Kubernetes. Public available declarations collected from GitHub are carefully labeled and trained, to demonstrate the feasibility of proposed declaration-based application classification. Declarations from several typical application classes are deployed in a Kubernetes cluster and generated logs are collected for model construction. Logs from failure cases are compared to the trained models. Via

978-1-5386-1412-9/17 $31.00 © 2017 IEEE
DOI 10.1109/ICEBE.2017.52

experimental studies, our approach demonstrates a strong ability in diagnosing multiple kinds of problems ranging from Cloud platform functional failures to application's erroneous configurations in both deploying and running phases. To the best of our knowledge, this is the first work bridging the log analysis and the declarative deployments in declaratively-deployed cloud. Moreover, the core approach and design principle are generally applicable to other Cloud platforms as well.

The remainder of this paper is structured as follows: Section II shows some related work. Section III states the background and overview of our system. The classification of deployment is demonstrated in Section IV. And we show the normal pattern modeling procedure in Section V. The anomaly detection with log is stated in Section VI. The proposed approach is validated in Section VII. Section VIII concludes this paper.

II. Related work

Log is widely used for diagnosing anomalies of software systems because of its simplicity and effectiveness. Analyzing logs for problem detection and identification has been an active research area. These work first parse logs into log templates based on code analysis or clustering mechanism, and build problem detection and identification model. The models can be classified into three categories, namely i) log quantity based model, ii) template quantity based model, iii) state transition based model. Log quantity based model is very simple, it counts the number of logs in a time window and if the log number increases sharply, it is considered as a problem [7]. This model can be applied in online scenario. However, it is too straight forward that can not meet the requirement of today's complex software systems. Template quantity based model usually count the number of different templates in a time window, and set up a vector for each time window. Then it utilizes methods such as machine learning algorithms to distinguish outliers [8,9]. This model is simple for implementation, however, it can not provide help for problem identification and diagnosis. If a problem is detected, engineers still need to manually search logs to find the cause. State transition model is the current research hotspots. It extracts correct template sequence indicating execution paths at first, and then generating a state transition model to compare with log sequences in production environment to detect conflicts [5]. This model has three advantages: i) it can diagnose problems that deeply buried in log sequences such as performance degradation, ii) it can provide engineers with the context log messages of problems, iii) it can provide engineers with the correct log sequence and tell engineers what should have happened. Lin, et al. [10] proposed a clustering approach to ease log-based problem identification; LogMine [11] leverages a fast hierarchical clustering method to generate log patterns; our previous work [6,12] proposed a log dependency based anomaly detection approach. But all of these approaches do not adopt the deployment information to assist problem diagnosis.

III. Background and Overview

In this section, we briefly describe the concept of declarative deployment, the basic architecture of Kubernetes, and the framework of LogDC.

A. Declarative Deployment

Declarative deployment leverages a structural model to describe the desired application structure and state which are enforced by a deployment engine [1]. In Kubernetes, a deployment declaration is structured using YAML format. Figure 1 shows the declaration of a simplified example application. It defines the initial Cloud deployment's configuration, like what

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
```

Fig. 1. A sample Kubebernetes deployment yaml file.

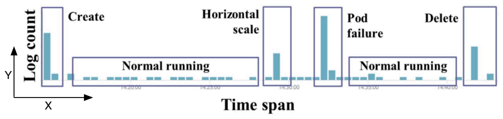


Fig. 2. A bird view of application lifecycle from log's perspective.

is the name of this application, which container image is used, and how the network is setup. Furthermore, the application's behavior is tuned by environment variables and additional policies, like resource limits and security setting. Generally speaking, the deployment file shows the static characterization of one application. Always, one application contains multiple software components. Therefore, an application deployment is a composition of multiple deployments. In Kubernetes, these deployments will be deployed, scheduled, and managed independently. Without losing generality, we only study the log characteristics of one single deployment throughout this paper.

B. Application Lifecycle on Kubernetes

As a typical declarative cloud platform, Kubernetes is designed for automating deployment, scaling, and management of containerized applications [4]. Kubernetes components continuously generate logs since an application is created, till it is deleted. Figure 2 gives a bird view of an exampled application lifecycle from log's perspective. The X-axis represents the time span and Y-axis represents the number of logs generated. The lifecycle consists of several application operation phases, like create, delete, scale, and recover, as well as several normal running phases. During operation phases, the number of generated log increases as multiple Kubernetes components collaborate to fulfill the operation. On the contrary, the number keeps low and steady during running phases. The reference model aims to capture logs from both kind of phases.

C. LogDC Components and Processes

Figure 3 shows the basic framework of LogDC. Generally speaking, LogDC comprises five modules, namely data crawler, deployment classification, model builder, log processing, and anomaly localization. Data crawler is in charge of collecting historical deployment yaml files and runtime logs. With the collected deployment yaml files, the deployment classification module extracts the text features of these deployments and builds a classification model for each type of deployment. Simultaneously, the log processing module weaves the log entries generated by different kubernetes components, builds log templates, and extracts log features. Given the log features and declaration classes, the model builder trains a reference model for each class of deployment. Then the anomaly localization module localizes the anomalies for the new collected log data with the corresponding reference model. The final output is suspicious declarative primitives or log entries. The details will be disclosed in the following sections.

IV. Deployment classification

As introduced in Section I , to pinpoint the root causes of an abnormal deployment, our approach first retrieves a log

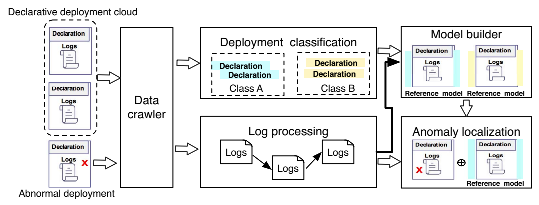


Fig. 3. The basic framework of LogDC.

重要图

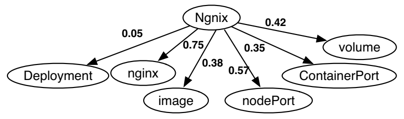


Fig. 5. The classification model of Nginx.

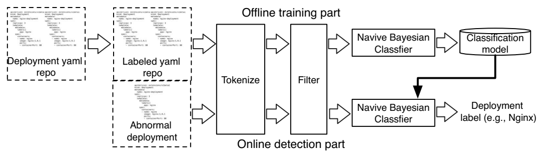


Fig. 4. Deployment yaml classification

reference model from the reference model repository. To identify such a reference model, we classify different deployments into different categories and label the reference model with a specific category. Therefore, we demonstrate the deployment classification procedure in this section.

A. Deployment Collection and Clean

To obtain a large sample of deployments, we crawl the deployments from our internal system and Github. On Github, there are a large number of deployment files uploaded by individual users and organizations. We collect thousands of deployment files with keywords “replicas or deployment ” and with an extension “yaml”. For example, from abduegal’s kubernetes-workshop code repository, we find a jenkins-deployment.yaml. However, not all of the deployment files can be deployed successfully. Thus, we manually filter out the deployments with explicit errors such as lack of critical configurations and syntax errors. After that, we label the deployment with its deployment name which is a configuration item in the yaml file. It is worth noting that different versions of the same application may be labeled with different names.

B. Deployment Classification

Figure 4 shows the framework of the deployment classification. Generally speaking, it comprises an offline training part and an online detection part. In the offline training part, we extract the text features from the corresponding deployments and classify these deployments with a Naive Bayesian approach [13]. The output of the offline training part is a group of classes, each of which is a Naive Bayesian network. In the online detection part, the abnormal deployment is classified with the pre-trained Naive Bayesian model. The output of the online detection part is a deployment label (e.g., Nginx).

In this paper, we treat each deployment file as a text document. To extract its features, we first tokenize the text document with the tokenizer APIs provided by NLTK (i.e., Natural Language Tookit) [14]. NLTK is a natural language library with a suite of text processing functions such as tokenization, stemming, tagging, parsing, etc. And each token is transformed into lower case. Then we filter out the meaningless tokens such as stop words and punctuation marks. Moreover, some predefined tokens (e.g., apiVersion) are also filtered out. After that, each token is taken as a text feature. The text features are then fed into the Naive Bayesian classifier to train a classification model. Here, we choose Naive Bayesian classifier as the text features are independent of each other and it only requires a small number of training data to estimate the parameters for necessary classification.

Given an abnormal deployment, the text features are extracted and fed into the Naive Bayesian classifier. The classifier returns the deployment class with the largest probability. To avoid such a situation that the value of the largest probability is too small, we pre-set a threshold ϵ (e.g., $\epsilon = 0.05$). If the largest probability is lower than ϵ, we say the abnormal deployment is a new deployment. The anomaly localization procedure will not be triggered. Once the problem is resolved by the system operators, this new deployment will be added into the deployment repository.

V. Normal Pattern Modeling

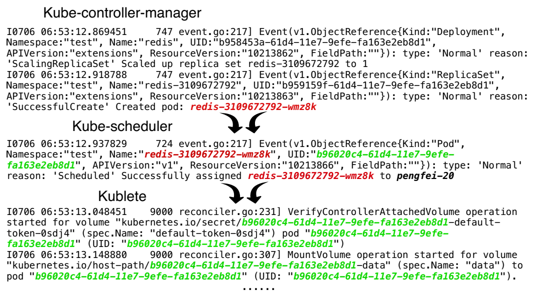
To pinpoint the root causes of problems from the perspective of log, we need to build up a log reference model for a normal deployment and compare it with the logs generated by the abnormal deployment. This section demonstrates the reference model building procedure.

A. Log Weaving

Recall that Kubernetes comprises multiple components just as introduced in Section III (B). The logs correlated with one deployment are distributed on different nodes. Therefore, to build a log reference model for the full lifecycle of one deployment, we need to weave the logs generated by different Kubenetes components. Kubenetes is a highly managed cloud platform. To manage the resources (e.g., Deployment, Replica set, Pod), Kubernetes assigns a UID and a name to each of them. The sample logs generated by *kube-controller-manager*, *kube-scheduler*, and *kubelet* are shown in Figure 6. From this figure, we observe that a UID named “*b958453a-61d4-11e7-9efe-fa163e2eb8d1*” is assigned to the “redis” deployment. While the replica “redis-3109672792” has a UID “*b959159f-61d4-11e7-9efe-fa163e2eb8d1*”. With these UIDs and names, we can weave the logs distributed in different components together. As stated in Section III (B), *kube-controller-manager* first creates pods for one deployment and assigns pod names to these pods. Then, *kube-scheduler* assigns UIDs to the created pods. Meanwhile, the created pods will be scheduled to some available nodes. After that, *kubelet* on the assigned worker node is charge of conducting a series of actions (e.g., mount volume) to establish the pods. Therefore, based on the deployment workflow, we fist find the pod created by one deployment in the logs generated by *kube-controller-manager* . Then other logs relevant to this pod can be weaved by the UID and pod name. Figure 6 shows an example of the weaving process with the logs coming from a “redis” deployment. It is worth noting that the weaved logs include not only the operation logs (e.g., create pod) but also the runtime logs (e.g., health-check log).

B. Log Feature Extraction

From Figure 6, we can observe that one log entry contains the log level (i.e., I, W, E), the timestamp, the thread ID,



```
I0706 06:53:12.869451     747 event.go:217] Event(v1.ObjectReference{Kind:"Deployment",
Namespace:"test", Name:"redis", UID:"b958453a-61d4-11e7-9efe-fa163e2eb8d1",
APIVersion:"extensions", ResourceVersion:"10213862", FieldPath:""}): type: 'Normal' reason:
'ScalingReplicaSet' Scaled up replica set redis-3109672792 to 1
I0706 06:53:12.918788     747 event.go:217] Event(v1.ObjectReference{Kind:"ReplicaSet",
Namespace:"test", Name:"redis-3109672792", UID:"b959159f-61d4-11e7-9efe-fa163e2eb8d1",
APIVersion:"extensions", ResourceVersion:"10213863", FieldPath:""}): type: 'Normal' reason:
'SuccessfulCreate' Created pod: redis-3109672792-wmz8k
```

```
I0706 06:53:12.937829     724 event.go:217] Event(v1.ObjectReference{Kind:"Pod",
Namespace:"test", Name:"redis-3109672792-wmz8k", UID:"b96020c4-61d4-11e7-9efe-
fa163e2eb8d1", APIVersion:"v1", ResourceVersion:"10213866", FieldPath:""}): type: 'Normal'
reason: 'Scheduled' Successfully assigned redis-3109672792-wmz8k to pengfei-20
```

```
I0706 06:53:13.048451    9000 reconciler.go:231] VerifyControllerAttachedVolume operation
started for volume "kubernetes.io/secret/b96020c4-61d4-11e7-9efe-fa163e2eb8d1-default-
token-0sdj4" (spec.Name: "default-token-0sdj4") pod "b96020c4-61d4-11e7-9efe-
fa163e2eb8d1" (UID: "b96020c4-61d4-11e7-9efe-fa163e2eb8d1")
I0706 06:53:13.148880    9000 reconciler.go:307] MountVolume operation started for volume
"kubernetes.io/host-path/b96020c4-61d4-11e7-9efe-fa163e2eb8d1-data" (spec.Name: "data") to
pod "b96020c4-61d4-11e7-9efe-fa163e2eb8d1" (UID: "b96020c4-61d4-11e7-9efe-fa163e2eb8d1").
                                          ......
```

Fig. 6. The log weaving process on Kubernetes platform.

the source file, and the message body. In this paper, we only leverage the message body to build up normal models and diagnose root causes. Other information will be screened out. There are many multiple variations for one kind of operation due to the variables contained in log entries. While the variable information is useless for us. Thus, we need to extract the invariants i.e., *template* from logs.

Template mining technique is mature and widely utilized in most research work. We have implemented an advanced template mining approach in our previous work [6, 12]. Each raw log is transformed to a certain template. In other words, we replace each raw log with its matched template and generate a sequence of templates. Note that in online phase, templates have already been mined, and online raw logs are mapped to the mined templates immediately. After transformation, the variables are replaced by a wide card "*" and the punctuation marks, and stop words are screened out. Particularly, we consider the numbers, the UIDs, and names of deployments, replica sets, pods, and nodes as the variables. For example, the message body of *kube-scheduler* in Figure 6 is transformed to "*Event v1 ObjectReference Kind pod Namespace test Name (*) UID (*) APIVersion v1 ResourceVersion FieldPath type Normal reason Scheduled Successfully assigned (*) to (*)*".

To determine the pattern of a normal deployment, we need to extract some key features. Here we recognize the composition of the name of one Kubenetes component and the template as a feature. For example, (*kube-scheduler*, *template-1*) is taken as a feature of "redis" deployment. Our approach maintains a feature repository. The repository contains all features extracted from the historical deployments. Each deployment is identified as a feature vector. The feature vector comprises "0" and "1" which denote the absence and the existence of one feature respectively. With the feature vector, we can train a normal model and diagnose the root causes for one deployment, which will be stated in the following sections.

C. Model Building

Due to the rapid evolution of micro services, a large number of deployments coming from the same application exist in a Kubernetes cluster. We collect these deployments and the corresponding logs. With the historical logs, we train log models for these deployments respectively. However, the first problem is that we cannot determine whether the deployment is normal trivially as some deployments do not report abnormal signals (e.g., error logs, service anomalies) explicitly. Therefore, our model building approach can leverage the logs generated by both of normal and abnormal deployments simultaneously. There is no need to label the data manually.

First, we use K-means clustering algorithm to separate normal and abnormal deployments (i.e., $K = 2$). Then we adopt logistic regression [15] to learn the patterns of normal and abnormal deployments. K-means is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining. It aims to partition n observations into K clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. Given a set of observations $(x_1, x_2, \cdots, x_n), x_i \in \mathbf{R}^d$ which denotes the feature vector of one deployment, K-means clustering aims to partition the n observations into $k(\leq n)$ clusters, namely $\mathbf{C} = \{C_1, C_2, \cdots, C_k\}$, so as to minimize the within-cluster sum of squares. Formally, the objective is to find:

$$\underset{\mathbf{C}}{argmin} \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2 = \underset{\mathbf{C}}{argmin} \sum_{i=1}^{k} |C_i| Var(C_i)$$

, where μ_i denotes the mean of cluster C_i, $Var(C_i)$ denotes the variance of cluster C_i. To find the best cluster, an "EM" (Expectation Maximization) [16] algorithm is commonly used. As K-means is always the default approach in many machine learning libraries such as *scikit-learn* [17], we just leverage the existing libraries rather than propose new algorithms. We try to leverage K-means approach to separate the normal and abnormal deployments. Therefore, K is set as 2. On the other word, there are two cluster centers, namely a normal cluster center C_n and an abnormal cluster center C_a. A basic assumption is that most of logs are collected from normal deployments. Therefore, the cluster with a larger number of deployments is classified as normal deployment and the other cluster is classified as abnormal deployment.

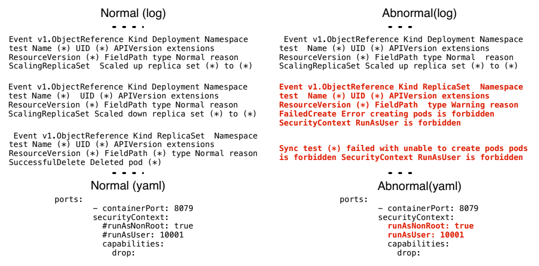
When a new log feature vector generated by a deployment is given, we need to determine whether it is normal or abnormal in real time. An institutive way is to classify the feature vector based on the distances to the central points C_n and C_a respectively. But it is still difficult to find such a distance threshold to instruct classification as we do not have a golden rule to follow. Considering this problem and the characteristics of the input data, we leverage logistic regression to conduct classification. Logistic regression is a kind of regression model where the dependent variables are categorical. In this paper, the dependent variables are binary that is it can only take two values, namely "0" (normal) and "1" (abnormal). Formally, the logistic regression is written as:

$$F(\mathbf{x}) = \frac{1}{1 + e^{(\beta_0 + \beta \mathbf{x})}},$$

where $\mathbf{x}$ denotes the input feature vector, β_0 and β denote the coefficients respectively. Pay attention to that β is a vector as $\mathbf{x}$ is a vector. With the historical logs generated from the same deployment, we estimate the model coefficients with a maximum likelihood estimation [16]. Each kind of deployment is corresponding to one log model. We select a log model based on the result of the deployment classification which is stated in Section IV (B).

VI. Anomaly localization

The anomaly localization procedure is triggered by an explicit anomaly or by the system administrator. If the procedure is triggered by the system administrator, we need to determine whether the deployment is healthy using the corresponding log model. Then we localize the root causes of anomalies if it is not healthy. If the anomaly localization request is submitted, we first determine which kind of deployment the current application belongs to. Then we select the log model


Normal (log)

```
Event v1.ObjectReference Kind Deployment Namespace
test Name (*) UID (*) APIVersion extensions
ResourceVersion (*) FieldPath type Normal reason
ScalingReplicaSet  Scaled up replica set (*) to (*)

Event v1.ObjectReference Kind Deployment Namespace
test Name (*)  UID (*) APIVersion extensions
ResourceVersion (*) FieldPath type Normal reason
ScalingReplicaSet Scaled down replica set (*) to (*)

 Event v1.ObjectReference Kind ReplicaSet  Namespace
test Name (*) UID (*) APIVersion extensions
ResourceVersion (*) FieldPath (*) type Normal reason
SuccessfulDelete Deleted pod (*)
```

Normal (yaml)

```
ports:
        - containerPort: 8079
        securityContext:
          #runAsNonRoot: true
          #runAsUser: 10001
          capabilities:
            drop:
```

Abnormal(log)

```
 Event v1.ObjectReference Kind Deployment Namespace
test  Name (*) UID (*) APIVersion extensions
ResourceVersion (*) FieldPath type Normal  reason
ScalingReplicaSet Scaled up replica set (*) to (*)

Event v1.ObjectReference Kind ReplicaSet  Namespace
test  Name (*) UID (*) APIVersion extensions
ResourceVersion (*) FieldPath  type Warning reason
FailedCreate Error creating pods is forbidden
SecurityContext RunAsUser is forbidden

Sync test (*) failed with unable to create pods pods
is forbidden SecurityContext RunAsUser is forbidden
```

Abnormal(yaml)

```
ports:
        - containerPort: 8079
        securityContext:
          runAsNonRoot: true
          runAsUser: 10001
          capabilities:
            drop:
```

Fig. 7. The differences between two deployments in logs and yaml file

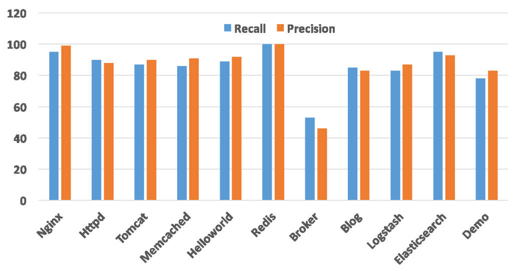


Fig. 8. The validation result of deployment classification.

corresponding to the deployment to localize anomalies. In this paper, we consider the anomalies hidden in the declarative yaml files and logs. To find the anomalies in logs, we compare the log feature vector of the current deployment and the features contained in the log model (i.e., the features with non-zero coefficients). Missing and emerging features are both recognized as anomalies. To find the anomalies in yaml files, we select the different deployment items between the current deployment file and the reference deployment by a "*diff*" operation. Figure 7 shows the differences between one normal deployment and one abnormal deployment in logs and yaml files and the differences are highlighted in red color.

VII. Experimental Validation

To validate the effectiveness of our proposed approach, we design and and develop a proof of concept named LogDC. LogDC adopts Github APIs to crawl more than two thousands of yaml files which contain "deployment" and "replicaset" in their file names. After the clean step introduced in Section IV (A), we obtain 1020 yaml files and label them with 80 kinds of deployments. To classify the deployments, LogDC leverages the the text processing methods in NLTK to parse the deployment files. At runtime, an agent named "filebeat" [18] continuously collects the deployment yaml files stored in a repository and logs generated by different components and forwards them to a central storage. We implement LogDC with Python as some open-source packages (e.g., scikit-learn) could be leveraged rather than reinvent wheels. Moreover, we run LogDC in a docker container. Thus, it can be seamless integrated into a kubernetes cluster.

A. Experiment setup

We set up a controlled Kubernetes cluster to quantitatively validate the effectiveness of LogDC. The cluster comprises four nodes, one master and three workers. Each node is configure with 2 CPU cores, 8 GB memory, 100 GB disk space, and one Gigabit NIC. The version of Kubernetes is 1.6.2.

To mimic the real problems happened in Kubernetes cluster, we investigated the issues in Kubernetes' issue repository [19]. From these issues, we randomly select five typical issues of them, namely security issue, node port conflict issue, inappropriate configuration issue, non-existing docker image issue, and mount volume issue. To reproduce these issues, we select 5 different kinds of deployments. For the security issue, we adopt the "front-end" deployment which is one component of "Sock-shop" [20]. In "front-end" deployment yaml file, the "securityContext" is configured as "runAsNonRoot: true runAsUser: 10001". However, when it is deployed in our cluster, the pod cannot be created due to "RunAsUser is forbidden." ; For the node port conflict issue, we select the Nginx deployment. In Nginx deployment yaml file, the node port is turned on and configured as 30001. When two pods are assigned to the same node, the port conflict issue is reported. To increase the possibility that more than one pods co-locate in one node, we configure the number of replica sets as 5 in Nginx deployment file; For the inappropriate configuration issue, we select the Prometheus deployment. To reproduce this issue, we limit the memory of prometheus as 600MB. As time goes on, the metric points collected by Prometheus are accumulated in memory. Therefore, once the memory is exhausted, the Prometheus does not work any more. For the non-existing docker image issue, we choose postgres SQL deployment and configure the "image" item with a image name that does not exist in the docker repository; For the mount volume issue, we select redis deployment and configure the "volumes" item with a path that does not exist. For each issue, we also prepare the normal deployment which runs well in our cluster. To get a log model, each normal deployment is repeated for 30 times. The duration of each experiment is 30 minutes including the deployment time and the running time. After that, each abnormal deployment is deployed for 10 times. If our approach can find out the abnormal configuration items or the true log entries, we say the anomaly localization is correct. To be more quantitative, we adopt two metrics, namely Precision and Recall [21], to validate the effectiveness of the deployment classification procedure and the anomaly localization procedure. The results will be shown in next section.

B. Results

We first show the effectiveness of the deployment classification procedure. As we have more than 80 classes of deployments, it is difficult to show all the validation results. Therefore, Figure 8 only shows the validation results of several kinds of deployments. From this figure, we observe that Precision and Recall of most kinds of deployments stay above 80%. Some of them even approach 100%. The results show that our approach can identify the class of a certain deployment precisely. The exceptional cases such as "broker" achieve a low Precision and Recall due to insufficient training samples or very similar yaml files with other deployments.

Then we validate the effectiveness of the anomaly localization procedure. Figure 9 shows the validation result of anomaly localization for the five issues. Generally speaking, Precision and Recall stay at a high level i.e., over 90%, as the logs generated by these issues can be distinguished from the logs generated by the normal deployments. Moreover, the characteristics of the abnormal logs are typical and stable. But for the configuration issue, we cannot always pinpoint the real abnormal log entries as this issue does not always trigger anomalies. It depends on the running time and memory resources left on one node.

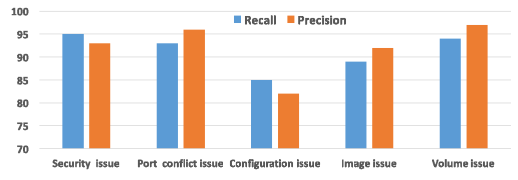


Fig. 9. The validation result of anomaly localization.

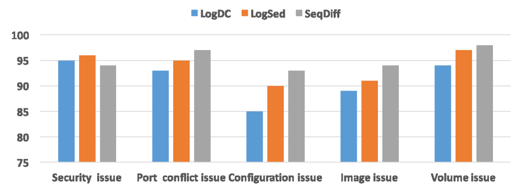


Fig. 10. The comparison of Recall for different approaches.

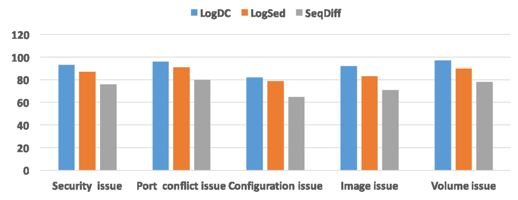


Fig. 11. The comparison of Precision for different approaches.

C. Comparisons

To thoroughly validate the effectiveness of LogDC, we compare it with several state-of-the-art approaches. In our previous work[8], we proposed an approach named "LogSed" to discover the event sequences for interleaved logs. By detecting the violation of event sequence, we can detect anomalies and find the abnormal log entries. To compare with it, we leverage the previous work to analyze the the logs generated by each deployments. Another intuitive and commonly used approach named "SeqDiff" is to compare the log sequences between normal and abnormal executions. This approach finds out all the differences that cannot be matched between each other. Figure 10 and Figure 11 show the comparisons of Precision and Recall for different approaches. From Figure 10, we observe that the Recalls of these approaches are compatible in general. But Recall of "SeqDiff" is a little higher than the one of the other two approaches. Because "SeqDiff" is sensitive to differences and can find out the anomalies even when the log entries are the same but the sequences are different. While LogDC is not sensitive to the order information. However, in terms of Precision, "SeqDiff" achieves the lowest value as it is blind to a specific type of issue. While Precision of LogDC is about 18% higher than the one of "SeqDiff" as LogDC sets up a log model to describe the normal and abnormal deployments with log features rather than log sequences.

VIII. Conclusion and Future Work

This paper proposes a log model based problem diagnosis tool for declaratively-deployed cloud applications. For deployment classification, this paper designs a novel method to extract features from application's deployment declaration and trains classifiers based on them. For model building, this paper leverages the fact that in a stable Cloud platform, the majority of deployed workloads are in their desired state, and collect normal samples based on clustering algorithms. By reducing the dependencies on the log entry's chronologic order during reference model's constructing and matching phases, LogDC is capable to diagnosing problems in an application's full lifecycle. The experimental results show that our approach can achieve high precisions and recalls of deployment classification and anomaly detection, which outperforms some relevant state-of-the-art. In the future work, we will discuss the effectiveness when multiple faults occur simultaneously and the optimization of deployment classification and anomaly localization.

Acknowledgment

We thanks all the anonymous reviewers. Ping Wang is the corresponding author.

References

[1] D. Fahland, L. Daniel, J. Mendling, H.A. Reijers, B. Weber, M. Weidlich, and S. Zugal, "Declarative versus Imperative Process Modeling Languages: The Issue of Understandability", *BMMDS/EMMSAD*, vol. 1, no. 29 : 353-365, 2009.

[2] C.B. Liu, B. T. Loo, and Y. Mao, "Declarative automated cloud resource orchestration", *in Acm Symposium on Cloud Computing*, pp. 1-8, 2011.

[3] Heat, https://wiki.openstack.org/wiki/Heat, [Accessed on July 17, 2017].

[4] Kubernetes, https://kubernetes.io/, [Accessed on July 17, 2017].

[5] B. C. Tak, S. Tao, L. Yang, C. Zhu, and Y. Ruan, "LOGAN: Problem Diagnosis in the Cloud Using Log-Based Reference Models", *in Proceedings of the 2016 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 62-67, 2016.

[6] T. Jia, L. Yang, P.F. Chen, Y. Li, F.J. Meng, and J.M. Xu, "LogSeD: Anomaly diagnosis through mining time-weighted control flow graph in logs", in *in Proceedings of 10th IEEE International Conference on Cloud Computing*, to appear, 2017.

[7] C. Lim, N. Singh, S. Yajnik, "A log mining approach to failure analysis of enterprise telephony systems", *in Proceedings of IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pp. 398-403, 2008.

[8] K. Kc, X. Gu, "ELT: Efficient Log-based Troubleshooting System for Cloud Computing Infrastructures", *in Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 11(1):11-20, 2011.

[9] W. Xu, L. Huang, A. Fox, et al, "Online System Problem Detection by Mining Patterns of Console Log", *in Proceedings of IEEE International Conference on Data Mining*, pp. 588-597, 2009.

[10] Q. Lin, H. Zhang, J.G. Lou, et al, "Log Clustering Based Problem Identification for Online Service Systems", *in Proceedings of International Conference on Software Engineering Companion*, pp. 102-111, 2016.

[11] H. Hamooni, B. Debnath, J. Xu, H Zhang, G.F. Jiang, et al., "LogMine: Fast Pattern Recognition for Log Analytics", *in Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pp. 1573-1582, 2016.

[12] T. Jia, L. Yang, P.F. Chen, Y. Li, F.J. Meng, and J.M. Xu, "An Approach For Anomal Diagnosis Based On Hybrid Grap Model With Log For Distributed Services", *in Proceedings of the 24th IEEE International Conference on Web Services*, to appear, 2017.

[13] E. Frank, and R. Bouckaert, "Naive bayes for text classification with unbalanced classes", *in Proceedings of the Knowledge Discovery in Databases: PKDD*, pp.503-510, 2006.

[14] NLTK, http://www.nltk.org/, [Accessed on July 17, 2017].

[15] Logistic regression, https://en.wikipedia.org/wiki/Logistic_regression, [Accessed on July 17, 2017].

[16] Jr. Hosmer, W. David, L. Stanley, and X. Sturdivant Rodney, "Applied logistic regression", *John Wiley & Sons*, Vol. 398, 2013.

[17] scikit-learn, http://scikit-learn.org/stable/index.html, [Accessed on July 17, 2017].

[18] Filebeat, https://www.elastic.co/products/beats/filebeat, [Accessed on July 17, 2017].

[19] Kubenetes issues, https://github.com/kubernetes/kubernetes/issues, [Accessed on July 17, 2017].

[20] Sock-shop, https://github.com/microservices-demo/microservices-demo, [Accessed on July 17, 2017].

[21] P.F. Chen, Y. Qi, D. Hou, P.F. Zheng, "CauseInfer:Automatic and Distributed Performance Diagnosis with Hierarchical Causality Graph in Large Distributed Systems", *in Proceedings of the IEEE Conference on Computer Communications*, pp. 1887-1895, 2014.