The Journal of Systems and Software 163 (2020) 110524

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Cloud reliability and efficiency improvement via failure risk based proactive actions☆

Yuli Tian[a,b], Jeff Tian[c,d,*], Ning Li[a,b]

[a] School of Computer Science, Northwestern Polytechnical University, Xi'an, Shannxi, China
[b] Ministry of Industry and Information Technology Key Laboratory of Big Data Storage and Management, Northwestern Polytechnical University, Xi'an, Shannxi, China
[c] Department of Computer Science, Southern Methodist University, Dallas, Texas, USA
[d] School of Informatics, Northwest University, Xi'an, Shannxi, China

ARTICLE INFO

Article history:
Received 19 October 2019
Revised 16 December 2019
Accepted 18 January 2020
Available online 20 January 2020

Keywords:
Cloud computing system
Reliability
Efficiency
Risk identification
Failure mitigation and fault tolerance

ABSTRACT

Due to the huge magnitude and complexity of cloud computing systems (CCS), failures are inevitable, which lead to reliability and efficiency losses. Failure mitigation, fault tolerance, and recovery actions can be performed to improve CCS reliability and efficiency. Using data collected during CCS operation, failure prediction and risk identification techniques could anticipate such failure occurrences. In this paper, we develop a framework to combine risk identification with follow-up proactive actions for CCS reliability and efficiency improvement. We start by analyzing cloud failures and the related operational data. Then a tree based predictive model is trained to diagnose high risk cloud tasks. By proactively terminating these high risk tasks, both the number of CCS failures and the resource consumption could be significantly reduced. The impact of these proactive actions can be simulated to quantify the improvement to both system reliability and efficiency. The new approach has been applied on the Google cluster dataset, covering approximately 400GB of operational data over 29 consecutive days, to demonstrate its viability and effectiveness.

1. Introduction

According to the National Institute of Standards Technology (NIST), cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction (Mell and Grance, 2011). Cloud users could run heavy computing tasks remotely by accessing cloud resources through networks without significant consumption or over provisioning of local resources. At the other end, cloud computing systems (CCS) handle millions of user's requests concurrently on virtual machines created from their resource pool. Due to the enormous magnitude and complexity of CCS, failures are inevitable. Such failures can negatively affect CCS in terms of both reliability and efficiency, causing enormous losses (Sharma et al., 2016; Garraghan et al., 2014; Ponemon Institute, 2016).

Various techniques have been developed and applied to prevent or tolerate CCS failures and to recover from them. For example, replication techniques run redundant processes on backup resources to provide extra reliability (Yu et al., 2015; Bonvin et al., 2010). Recovery techniques ensure CCS could recover from failures (Elnozahy et al., 2002; Liu et al., 2009; Belalem and Limam, 2013). Proactive approaches anticipate possible failure occurrences then take measures to avoid them or to minimize consequent system losses, for example, by using VM migration techniques (Hines et al., 2009; Shribman and Hudzia, 2012). The effectiveness of such proactive methods depends on the accuracy of the risk identification or failure prediction involved (Islam et al., 2012; Tak et al., 2016; Chen et al., 2014). Another attempt is to utilize the emerging fog computing system to provide more reliability and efficiency for particular services (Numhauser, 2012; Pereira et al., 2019; Xu et al., 2016). In this paper, we present a new cloud failure analysis and risk identification method combined with follow-up actions for cloud reliability and efficiency improvement. The approach has been applied to the Google cluster dataset (Reiss et al., 2011) to demonstrate its viability and effectiveness.

The rest of the paper is organized as following: Section 2 introduces background and related work. Our research problem and

☆ Edited by: Prof J. C. Duenas.

* Corresponding author.
E-mail address: tian@smu.edu (J. Tian).

https://doi.org/10.1016/j.jss.2020.110524
0164-1212/

solution strategy are described in Section 3. Section 4 describes the data and some preliminary analyses, with the main results presented in Section 5. A variation of our method providing early risk identification is presented in Sections 6. Discussions and conclusions can be found in Section 7 and Section 8 respectively.

2. Background and related work

This section describes the background and related work about CCS reliability and efficiency, techniques for risk identification and failure prediction, and empirical study methods and tools.

2.1. CCS reliability and efficiency

Cloud computing has become an important way to meet people's computational and storage demands. Reliability and efficiency are essential to keep CCS delivering dependable and affordable services. CCS reliability measures system capability of satisfying users requests from an external user's perspective, which is defined as the probability of failure-free execution for a specified period of time or the number of execution instances under this specific environment (Musa et al., 1987). CCS efficiency or performance efficiency represents the performance relative to the amount of resources used, which is more important to CCS providers due of business concerns (ISO/IEC, 2011).

CCS reliability can be analyzed with input-domain models (Lyu, 1996), which measure operational reliability similar to the application environment of this study. For example, Nelson model (Nelson, 1978) gives reliability R, with the number of failures f and the number of total runs n, in Eq. (1).

$$R = 1 - \frac{f}{n} \tag{1}$$

CCS efficiency can be defined based on different resource measures, for example, energy, time and computational resources such as CPU, memory, and I/O utilization (ISO/IEC, 2011). In this paper, we measure CCS efficiency E with the number of runs n and their resource consumption U in Eq. (2).

$$E = \frac{U}{n} \tag{2}$$

To run a highly reliable and efficient CCS has always been a huge challenge to cloud providers and developers. With the emergence and proliferation of cloud computing, various CCS quality assurance and improvement methods have been proposed and applied through CCS lifecycle from design to operational stages (Sharma et al., 2016). Reactive methods such as replication and recovery ensure CCS function after failure occurrences. Proactive methods such VM migration anticipate possible failures and take measures to avoid them or to minimize consequent system losses. In addition, CCS often utilize enhanced services in closer proximity to end users, such as through fog computing or edge computing, to ensure system integrity and availability. Fog computing is an extended cloud computing paradigm to the edge of the network dealing with the growth of connected devices (Numhauser, 2012; Pereira et al., 2019; Xu et al., 2016). With its advantages in mobility, location-awareness, and closer to usage scenario, fog computing provides enhanced reliability and efficiency for particular services (Pereira et al., 2019; Xu et al., 2016).

Replication approaches provide fault tolerance by running replicas on backup resources (Yu et al., 2015; Bonvin et al., 2010). In addition to the difficulty to maintain the consistency between the replicas and the running process, preparing the backup resources can add significant cost. Checkpointing is one of the most popular failure recovery techniques, in which snapshots of ongoing processes are saved as checkpoints (Elnozahy et al., 2002; Liu et al., 2009; Belalem and Limam, 2013). When a failure occurs, the system will be rolled back to the latest saved checkpoint and restarted from there. Checkpointing approaches improve system reliability by allowing multiple execution attempts while incurring some overhead on time and resources (Fu, 2010).

To avoid the large overhead associated with such failure-recovery mechanisms, cloud provider can predict the occurrences of system failures and handle potential failures proactively. VM migration methods prevent failure occurrences based on failure prediction results by migrate running processes from suspected resource to healthy resources (Hines et al., 2009; Shribman and Hudzia, 2012). However, the effectiveness of such proactive failure management methods depends upon the accurate prediction of failure occurrences (Islam et al., 2012).

2.2. Risk identification and failure prediction

Prediction techniques are widely used in software engineering, from assessing future operational reliability of systems to forecasting whether a system would encounter a runtime failure (Lyu, 1996; Salfner et al., 2010). Defect prediction techniques uses software metrics, change history, runtime monitoring, and even developer characteristics to anticipate software failure occurrences. For example, which file or even which line of code contains software defects that may lead to runtime failures, whether a code change introduce software defects, or whether a running process may encounter a failure.

In recent years, along with the wide-spread application of machine leaning techniques, neural network based methods have been used to predict cloud failures (Chen et al., 2014; Islam and Manivannan, 2017; Liu et al., 2017). Such methods build black-box classification models to learn the temporal characteristics of resource usage in order to capture abnormal usage patterns and use the results to identify high risk cloud processes.

CCS abnormal behavior can also be tracked by comparing to a standard reference model (Tak et al., 2016). Such methods use successful execution logs to build a task model for each of the system functions and compare running tasks to the model to identify anomalies. To build and maintain many such task models can be a huge challenge to CCS providers as modern CCS system evolves constantly.

These techniques have produced cloud failure prediction results with increasing levels of accuracy. The next step is to integrate these predictions as feedback to cloud service providers to guide proactive actions for an effective and efficient implementation of failure-recovery mechanisms – a step we take in this research to extend and integrate existing work originated from two different communities, the technical community to produce highly dependable and affordable cloud services and the software engineering community that diagnose and solve software and system problems to ensure system reliability and efficiency.

Tree based modeling (TBM), a statistical analysis technique based on recursively partitioning, has been used in software defect diagnose, risk identification, and reliability assessment and improvement (Tian, 1995; Kiciman and Fox, 2005). Comparing to other black-box prediction models such as machine learning algorithms and neural networks mentioned above, TBM can provide constructive interpretations of failure patterns and highlight high-risk areas of the system, which can be helpful for follow-up fault removal and quality improvement. For example, in combination with cause-effect analysis, system developers and maintainers can trace back to failure root causes. Furthermore, TBM could handle both numerical data, such as priority, CPU rate, and memory capacity, and categorical data, such as machine features or platform types, at the same time, which is the case for the mixed types of data for CCS. In this paper, we use TBM to predict CCS

Table 1
Prediction result criteria.

		Prediction condition	
		Risky	Healthy
Real Condition	Fail	True Positive (TP)	False Negative (FN)
	Pass	False Positive (FP)	True Negative (TN)

failures, followed by proactive actions for both CCS reliability and efficiency improvement.

When a large number of data attributes or variables are available to build prediction models, as is the case for CCS data, we need to select a few variables as predictor variables for our TBM as the recursive partitioning algorithm employed in TBM is quite computationally intensive. We use mutual information (MI) (Cover and Thomas, 2012), a measure of mutual dependency between variables, to make this selection. MI between two variables X and Y can be calculated by Eq. (3), where p is the joint probability mass function.

$$\mathrm{MI}(X,Y) = \sum_{x\in X}\sum_{y\in Y} p(x,y)\log\frac{p(x,y)}{p(x)p(y)} \tag{3}$$

Comparing to other measures of dependency, MI can deal with both numerical and categorical data. MI is also robust against noise and skewed distribution.

2.3. Empirical studies, evaluation metrics, and tools

We develop a new cloud risk identification and improvement framework following the quality improvement paradigm (QIP) (Oivo and Basili, 1992). QIP can be summarized in three steps: 1) characterize the current project and set quantifiable goals, 2) execute appropriate process change towards the goal and collect feedback, 3) perform post analysis for evaluation and packaging of the experience for deploying the quality improvement actions. In order to ensure the practicality and generalizability of the new method to other CCS, we implement QIP on the generic and representative Google cluster dataset (https://github.com/google/cluster-data), which has been widely used for research (Chen et al., 2014; Zhu et al., 2014; Moreno et al., 2014). We set this generic dataset as our baseline, on which a goal of CCS reliability and efficiency improvement is proposed. Next, TBM based risk identification is performed and follow-up proactive actions are simulated on the original dataset, on which new CCS reliability and efficiency are measured and compared to the baseline for post analysis and method validation.

When making a prediction of failure risks, there are four combinations of actual and predicted results: true positive (TP), false positive (FP), false negative (FN), and true negative (TN), as shown in Table 1. The true positive condition refers to the situation that a predictor correctly identifies a risk, such as correctly classifying a failure as risky in this study. The false positive condition refers to the situation that a predictor incorrectly classifies a healthy run as risky. The false negative condition refers to the situation that a predictor fails to identify a risk and incorrectly classifies it as healthy. The true negative condition refers to the situation that a predictor correctly classifies a healthy run. We use the metric *Accuracy* defined in Eq. (4) to evaluate overall prediction results.

$$Accuracy = \frac{TP+TN}{TP+TN+FT+FN} \tag{4}$$

The metric *Accuracy* is the proportion of correct predictions among the whole population. We also use true positive rate, known as $Recall_{TP}$, and false positive rate, known as $Recall_{FP}$, to evaluate how many risky executions can be correctly identified and how many passes are misclassified as risky. $Recall_{TP}$ and $Recall_{FP}$ are defined as:

$$Recall_{TP} = \frac{TP}{TP+FN} \tag{5}$$

$$Recall_{FP} = \frac{FP}{FP+TN} \tag{6}$$

Higher values for *Accuracy* and $Recall_{TP}$, and lower values for $Recall_{FP}$ would be more desirable.

All the computing, statistical analysis, and modeling associated with this research are supported by a powerful statistical computing tool, R (https://www.r-project.org/). Due to the huge amount of data, we performed our experiments on a high-performance computing cluster, ManeFrame (https://www.smu.edu/academics/csc).

3. Problem and solution strategy

This study aims at CCS reliability and efficiency improvement by predicting and proactively avoiding potential failures. Due to the high complexity and concurrency of CCS, it is difficult to reproduce system failures, which makes it even harder to remove system defects. Failure mitigation and fault tolerance techniques are often used as alternative solutions to enhance CCS reliability (Belalem and Limam, 2013; Bonvin et al., 2010; Xu et al., 2016). Such techniques incur significant overhead in terms of system computational resources or service time. Failure prediction methods from software engineering research could identify high risk cloud tasks and prevent failures by proactive actions (Hines et al., 2009; Shribman and Hudzia, 2012). What is needed is an integration of these recent work from the CCS community and the software engineering community to apply the cloud failure predictions to proactively improve CCS reliability and efficiency, where reliability is one of the ultimate goals for cloud users, while efficiency is critical to ensure the competitiveness of cloud service providers.

We develop our risk identification and failure prevention strategy to address the above problems. Our method is illustrated in Fig. 1. Data collected throughout CCS task life-cycle is used to train a TBM, which is then applied for CCS risk identification. Diagnosed risky executions are terminated immediately. The impact of such proactive actions are two-fold: 1) fewer failures as a direct result of failure prevention, which can be quantified as higher reliability; 2) reduced resource consumption due to the fact that the tasks or processes heading towards failures are proactively terminated before they consume resources between the termination point and failure point. These actions and their impact can be simulated by processing the original execution data, taking into consideration of the termination decisions. The simulated results are compared to the baseline dataset to evaluate and validate our method. This new method consists of four steps:

1. Data preparation and preliminary analysis. Collect and preprocess CCS data, perform preliminary analysis, and determine the baseline CCS reliability and efficiency.
2. Model training. Train a TBM to identify high risk executions and periodically update it to keep up with the latest cloud operations.
3. Proactive actions and impact simulation. Apply the trained model to terminate diagnosed risky executions and build the simulation dataset to evaluate and quantify the impact of these proactive actions.
4. Evaluation. Compare the simulation result to the original dataset in terms of both system reliability and efficiency to validate the method.

We develop the new method at the task level, the basic execution unit of CCS. Therefore, we reorganize the dataset in a task-centric manner, with the CCS behavior described and indexed by

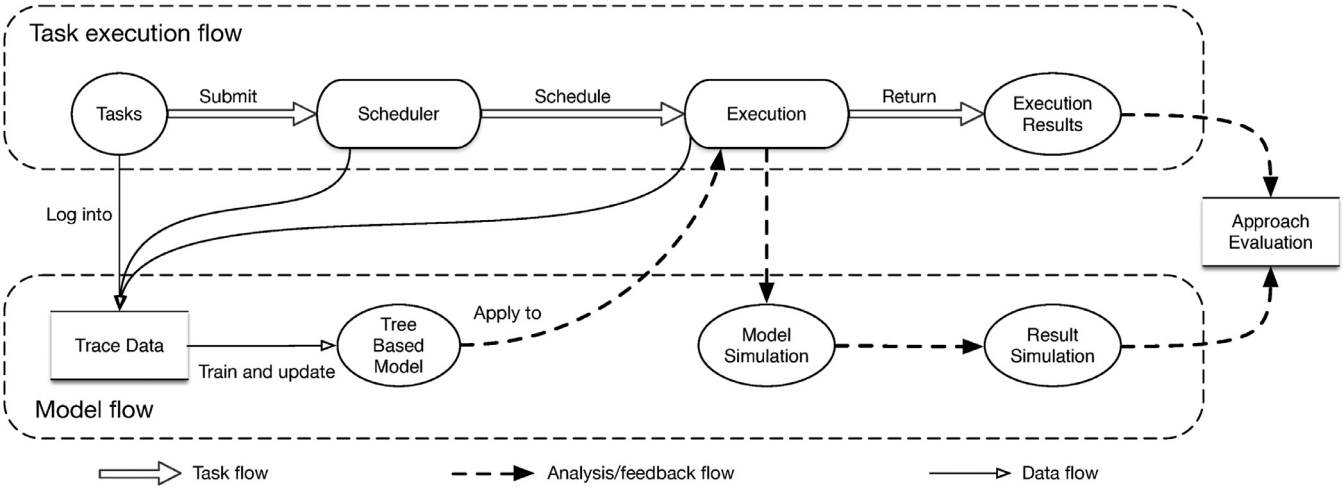


Fig. 1. Method overview.

tasks. The reorganized dataset consists of all the task execution instances and their attributes describing user specifications, CCS configurations, and runtime monitoring data. Then a failure analysis is performed in order to characterize the CCS to set a baseline reliability and efficiency level.

CCS reliability and efficiency are defined differently from different perspectives in this research. CCS task execution is the basic unit, to allocate computational resources and to execute. A task lifecycle may contain several executions due to failure-retry by CCS, but only the final returned execution results are visible and meaningful to end users. Therefore, we define CCS reliability and efficiency from the CCS user's perspective on the task basis, i.e., only the final results are counted, while all intermediated execution results are ignored. In contrast, we define CCS reliability and efficiency from the CCS provider's perspective on the execution basis, i.e., every execution is counted separately and marked as a successful execution or a failed execution individually.

Nelson model Eq. (1) is used to establish the reliability baseline at the task level from the user's perspective and at the execution level from the CCS provider's perspective. CCS efficiency baseline can be set by Eq. (2). Similar to CCS reliability, we calculate CCS efficiency differently based on different usage measurements for different interest parties. In this case, we use task completion time measured by natural time as usage measurement for users. From the CCS provider's perspective, we measure three main computational resources: CPU, memory, and I/O utilization.

To gain a general understanding of cloud failures, we use mutual information (MI, Eq. (3)) between task execution result and task attributes to measure the strength of their linkage. MI is further used for data dimensionality reduction in combination with the elbow method, which locates an obvious change point as a threshold to select a few data attributes as predictor variables for our TBM.

Next, we train tree base models (TBM) with historical execution trace data. The execution result indicator (success or failure) will be treated as the response variable, and a few selected data attributes as predictor variables. We train and update the TBM periodically to keep up with the latest cloud operations. CCS usually have very high throughput, and usage patterns may change significantly in a short period. Therefore, it is important to update the predictive model to the current usage pattern to ensure prediction accuracy. The trained TBM predicts whether a running task would encounter a failure. The accuracy of the prediction method can be measured by the overall accuracy Eq. (4), the correct termination recall Eq. (5), and the incorrect termination recall Eq. (6).

The trained TBM is used to examine runtime executions and to immediately terminate executions identified as being risky to avoid failure occurrences and additional CCS resource consumption. Consequently, the four prediction cases described in Table 1 lead to four action-impact pairs: 1) correct termination (CT), or the true positive (TP) case, denotes the case that a failure is correctly identified and terminated; 2) incorrect termination (IT) or the false positive (FP), denotes the case that a healthy execution is incorrectly diagnosed as risky and terminated; 3) correct pass (CP), or the true negative (TN), denotes a healthy execution has correctly passed; 4) incorrect pass (IP), or the false negative (FN), denotes a failure is incorrectly classified as healthy and has passed. Among them, a correct termination prevents a failure from occurring, which increases system reliability as well as efficiency. Correct and incorrect passes would not affect system operation because no action was taken based on modeling results. Incorrect terminations are harmful and should be avoided as much as possible because terminating healthy executions would reduce successful executions and CCS reliability. Unless the predictive model gives strong confidence to classify a task instance to the high risk group, we let it pass and proceed. On the other hand, setting the threshold too high may result in too few correct terminations and too many incorrect passes, which is not desirable either. Therefore, we experimentally select an appropriate classification threshold value for best overall result.

Once an execution has been diagnosed as risky, it would be terminated immediately. Table 2 summarizes failure counting and resource consumption criteria for both the original dataset and the simulation dataset. For the simulation dataset, we distinguish terminations and their outcomes at the execution level and task level, denoted as $Outcome_e$ and $Outcome_t$ respectively. Correct terminations at the execution level reflect appropriate system interference on risky executions resulting in fewer failure occurrences, which should be seen as successes from the CCS provider's perspective, and lower system resource consumption. However, incorrect terminations of healthy executions will lead to unnecessary re-executions and increase system resource consumption, which should be seen as failures from the CCS provider's perspective. For both the correct termination case and the incorrect termination case, only the consumption before termination point is counted. Total resource consumption of the simulation dataset is denoted as U_e at the execution level and as U_t at the task level.

We extend the Nelson model defined in Eq. (1) to measure CCS reliability as the probability that the system could successfully complete an execution. CCS reliability enhanced by the new

Table 2
Failure counting and resource consumption criteria.

	Original dataset		Simulation dataset		
	Outcome	Consumption	$Outcome_e$	$Outcome_t$	Consumption U_e, U_t
Correct Termination	Failure	Consumption	Success	Not Counted	Partial Consumption
Incorrect Termination	Success	Consumption	Failure	Not Counted	Partial Consumption
Correct Pass	Success	Consumption	Success	Success	Consumption
Incorrect Pass	Failure	Consumption	Failure	Failure	Consumption

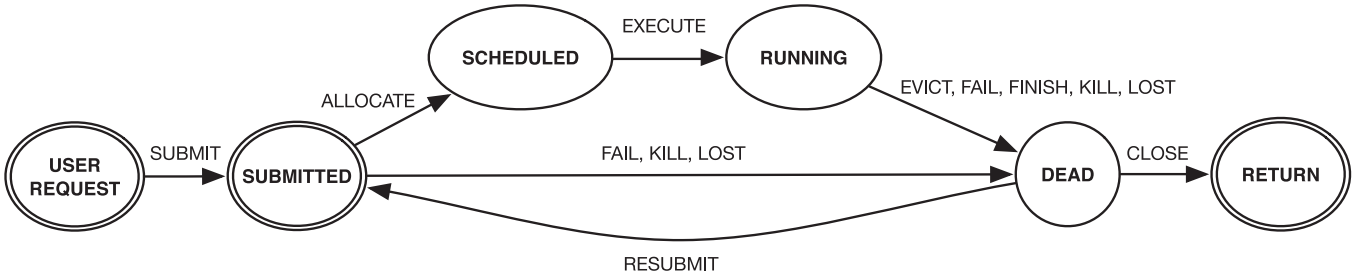


Fig. 2. Google cluster task life-cycle.

Table 3
Re-organized task-centric data format.

Category	Description	#indiv. attr.
External	Job/Task identifiers, specifications, and their change records	12
Internal	Job/Task resource specifications, and their change records	10
System	Task attempts, scheduling specifications, machine configurations, attributes and their change records	79
Runtime	Task runtime constraints, violations, and monitoring usages	87

method R_e at the execution level could be calculated as following:

$$R_e = \frac{CP_e + CT_e}{CP_e + IP_e + CT_e + IT_e} \tag{7}$$

Terminations conducted by CCS happen at the execution level, which is transparent to end users. Therefore, they should be counted as neither failures nor successes from user's perspective. CCS reliability from the user's perspective is defined as following:

$$R_t = \frac{CP_t}{CP_t + IP_t} \tag{8}$$

CCS efficiency can be measured by how much system computational resources or natural time are spent to complete an execution or a task. Computational resources consumption of the simulation dataset U_e is used to calculate enhanced CCS efficiency E_e at the execution level from CCS provider's perspective. Similarly, task completion time U_t is used to calculate CCS efficiency E_t at the task level from the user's perspective. Adapting Eq. (2), we calculate CCS efficiency after applying the new method at the execution level and the task level respectively as:

$$E_e = \frac{U_e}{CP_e + IP_e + CT_e + IT_e} \tag{9}$$

$$E_t = \frac{U_t}{CP_t + IP_t} \tag{10}$$

Following the same rule for CCS efficiency baseline, we use three instantiations of U_e to measure CPU, memory, I/O utilization separately at the execution level, and U_t to measure the task-level completion time for users.

4. Data and preliminary analysis

This section presents the first step of the new method, including data collection, pre-processing and preliminary failure analysis, which defines the baseline CCS reliability and efficiency.

4.1. Data collection and reorganization

We conducted our study on the Google cluster dataset collected in May 2011 (Reiss et al., 2011). This dataset of approximately 400GB records the usage traces of about 670,000 jobs, consisting of about 26 million tasks running on about 12,500 computing nodes in 29 consecutive days.

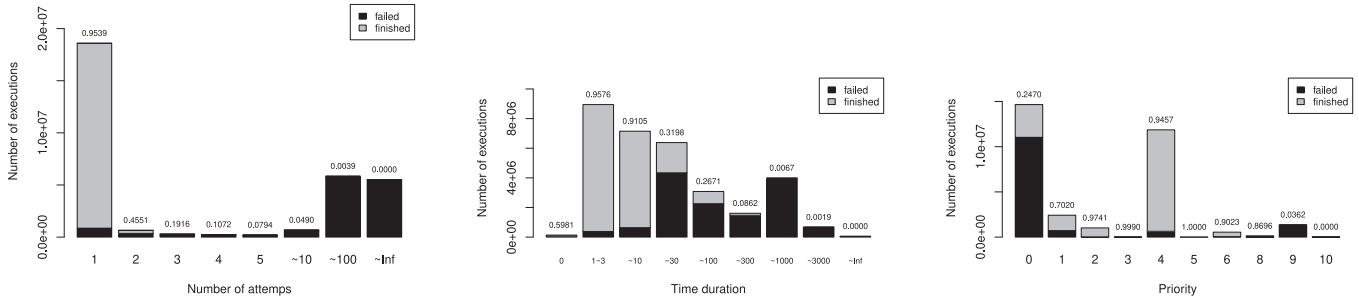
Google cluster task life-cycle is shown in Fig. 2. End users submit jobs consisting of one or more tasks to the cluster. Then a system scheduler allocates computing nodes for individual tasks to be executed on. The dataset logs the system behavior in different CSV files. Machine attributes and machine events files record CPU/memory capacity, platform type and kernel version operated on the computing nodes, etc. Job events and task events files record the events messages, such as `submitted` representing a job/task been submitted, `finish`/`fail`/`kill` referring to job/task completion status. A task may have some constraints to restrict them to be executed on certain machines. Such information is logged by task constraints files. The last part of the data is task usages, which monitors resource utilization in terms of CPU rate, memory utilization, I/O rate, etc.

In this study, we evaluate system reliability and efficiency using metrics for user-oriented tasks. Therefore, we reorganized the dataset into a task-centric format, with the system behavior described and indexed by tasks. Besides the original attributes provided by the datasets, we extracted additional historical attributes from the dataset recording task configuration modifications and execution attempts. For example, task attempts number is the number of attempted task executions. Machine event count refers to the number of machine configuration modifications. These historical attributes are shown to be highly correlated to failures.

Table 3 classifies the 188 re-organized task-centric data attributes extracted from the raw dataset into four categories, external, internal, system and runtime, as follows:

Table 4
Dataset summary.

	Total	Failed	Finished	Reliability	E-Time	E-CPU	E-Mem	E-IO
# of tasks	18,583,963	3,808,295	14,775,668	0.7951	2.85E+09	-	-	-
# of executions	32,016,553	13,828,583	18,187,970	0.5681	-	9.29E+10	1.99E+07	3.00E+07



(a) Over number of attempts (b) Over time duration (c) Over priority

Fig. 3. Failure distribution over different attributes.

- The external attributes are collected before task submission, such as user and task identifiers.
- The internal attributes are determined when a task enters the CCS based on user's own specifications, such as CPU, memory, and disk requests.
- The system attributes are associated with task allocation and scheduling, such as scheduling class and allocated CPU/memory.
- Runtime attributes record dynamic execution behavior of the task during the running stage, such as runtime constraints, CPU/memory utilization, and execution duration.

Once a task transits to `dead` status, its current execution ends at the system level with a completion event, including `finish`, `fail`, `kill`, `lost`, and `evict`. Then the task may be resubmitted and starts a new execution iteration, or enters `return` status indicating an end at task level with a completion event returned to its owner user.

The original dataset monitors execution runtime usage at 5 min intervals. Such data are organized into runtime crosscuts which combine task internal attributes, system attributes, and runtime aggregation attributes at each monitoring point, such as average CPU rate, peak memory usage and time duration. These crosscuts are used for diagnosing risky tasks.

Reorganizing Google cluster data into a task-centric format with the four categories of task attributes will allow us to identify high risk areas. We can also use different types of attribute combinations to simulate and evaluate failure prevention techniques on different scales. For example, we will use the internal and system attributes only for early failure prevention strategy in Section 6, which predict high risk tasks before execution.

4.2. Preliminary failure and reliability analyses

According to the task status transition diagram in Fig. 2, submitted task executions end their life-cycles with five different termination status, `finish` (37.74%), `fail` (28.65%), `kill` (21.44%), `evict` (12.15%) and `lost` (less than 0.1%). Among them, `finish` status indicates successful completion, and system failure leads to `fail` status. `kill` and `evict` status refer to legal abortion by the users or the system scheduler, while `lost` is associated with missing data. We can not determine whether a task is related to a failure from the last three task ending types, so this part of the data is excluded from our study. Furthermore, the dataset trace is extracted from real system operation within 29 consecutive days. The executions that started before tracing started and those that ended after tracing ended are also removed due to the incomplete task information, which accounts for about 0.1% of the dataset.

The trimmed dataset is summarized in Table 4, together with the preliminary analysis results that give us the baseline reliability and efficiency assessment for this CCS at both the task level from the CCS user's perspective and the execution level from the CCS provider's perspective. The dataset traced 32,016,553 task execution records, representing 18,583,963 tasks. 14,775,668 (79.51%) of the tasks ended successfully, indicating fair reliability from the user's perspective, taking the system as a black box, disregarding whether these tasks have experienced any intermediate failures before their final successes. However, from the CCS provider's perspective, we noticed that 43.19% of the executions ended with failures, indicating bad reliability when looking into the system at the execution level. This could be the result of system fault tolerance mechanisms, for example, resubmitting and rescheduling a failed task execution with different system configurations. Even such mechanisms help to deliver decent reliable services to end users, it is still a huge waste of system resources and user's time to execute tasks at such a low success rate. These observations demonstrate the need for a runtime risk identification framework to avoid excessive failure occurrences and to save system resource consumption. Next, we examine the failures and identify key attributes linked to them.

4.3. Key attribute analysis

Among the 188 individual data attributes in the dataset, some are closely correlated to the results at the task level or at the execution level. Next, we perform 2-way analysis, or joint distribution analysis, linking pairs of data attributes to examine failure distribution over other data attribute values. Fig. 3 shows task execution results distributions over three key attributes, the number of attempts, time duration, and priority. Each bar in these figures is divided into two parts: The dark portion represents the number of failures and the light portion represents the number of successes. The number on the top of each bar is the reliability according to the Nelson model Eq. (1) for the corresponding sub-domain.

The attribute associated with the most uneven failure-distribution is the number of attempts at the system level. High numbers of attempts may imply that such tasks have experienced numerous failures and a high likelihood to encounter another

failure. Fig. 3(a) shows the execution results distribution over the number of attempts. Over 95% of the executions succeeded on their first attempt, but only 45.54% of executions succeeded on their second attempt. The success rate keeps dropping along with the increase in the number of attempts. Less than 1% of the executions succeeded over their 6th to 10th attempts and no execution succeeded over its 11th to 100th attempts. By considering such low success expectation of tasks with high attempts, we would suggest discarding these tasks at scheduling and encourage users to re-check the submitted tasks.

The second most influential attribute is the number of runtime monitoring records. As execution in runtime would be monitored periodically, this attribute can be interpreted as an approximation of actual execution time. We noticed that long-duration tasks are more likely to fail than short-duration tasks, as Fig. 3(b) shows. Tasks executing less than 10 recording periods fail at 6.6%, while the rest of executions encounter failures with a probability of 80.8%. Long-duration tasks are linked to the majority share of system resource consumption. Therefore, to build a predictive model to diagnose such failure prone executions and terminate them during execution could significantly reduce system resource consumption.

The most distinguished attribute from the internal attribute group is priority. According to Google's specification, tasks with higher priority numbers generally get preference for resources over tasks with lower priority numbers (Reiss et al., 2011). Failure distribution over priority is shown in Fig. 3(c). We can see that executions with very low and very high priorities are more likely to fail than tasks with mid-range priorities. "Free" priority tasks ($priority \leq 1$) request little resources and their executions fail with the probability of 68.84%. While high priority tasks are "production" tasks ($priority \geq 9$), which are latency-sensitive and should not be evicted due to prioritized-allocation of machine resources. Although such executions only account for 9.62% of total usage, their failure rate reaches up to 96.44%, indicating serious problems that need to be investigated further.

4.4. Selecting data attributes for modeling

The three attributes above and a few other attributes selected from the 188 individual data attributes in our data set can potentially be used in TBM as predictor variables to identify high risk executions in our approach. On the other hand, not all 188 attributes can be used in TBM, which wouldn't be practical due to the excessive computational resources needed for modeling, as the recursive partitioning algorithm employed in TBM is quite computationally intensive. In our experience, reducing the number of attributes from 80 to 20 could shorten model training time from days to within one hour on the ManeFrame high performance computing node containing dual Intel Xeon E5-2695v4 2.1 GHz 18-core Broadwell processors with 45 MB of cache each and 256 GB of DDR4-2400 memory.

The attributes to be excluded first are external attributes because they are mostly un-configurable identifiers and specifications. Such attributes can hardly be used for defect interpretation and follow-up fault removal activities due to their low modifiability and controllability in the context of this research. However, they could be useful offline to guide focused inspection, causal analysis, and improvement activities performed manually by software developers.

To quantify the linkage between each individual data attribute A_i and the execution result indicator D, we use mutual information $\mathrm{MI}(A_i, D)$ between the pair. The larger value of $\mathrm{MI}(A_i, D)$ indicates more uneven failure distribution across attribute A_i, which may be used to identify high risk areas of software systems or high risk operations of end users related to the attribute. As described in Section 2, MI is more suitable for mixed numerical and categorical data with skewed or multi-polar distributions. Our dataset fits this profile.

Table 5
Selected attributes as predictor variables.

No.	Attribute A_i	Category	$\mathrm{MI}(A_i, D)$
1	Number of attempts	System	0.5096
2	Duration	Runtime	0.3638
3	Priority	Internal	0.2863
4	Memory per instruction	Runtime	0.1204
5	Machine attribute a	System	0.0950
6	Machine attribute b	System	0.0892
7	Job Scheduling class	Internal	0.0723
8	Job Scheduling class changes	System	0.0669
9	Cycles per instruction	Runtime	0.0592
10	Machine attribute c	System	0.0585
11	Machine attribute d	System	0.0582
12	Local disk space usage	Runtime	0.0568
13	Machine restriction	Internal	0.0338
14	Machine attribute changes	System	0.0305
15	CPU rate	Runtime	0.0231
16	Machine attribute e	System	0.0220
17	Maximum CPU rate	Runtime	0.0202
18	Machine Memory	System	0.0155
19	Machine changes	System	0.0124
20	Machine CPU	System	0.0114
21	Machine availableness	System	0.0112
22	Machine attribute f	System	0.0086
23	Scheduling class	System	0.0076
24	Machine attribute g	System	0.0074
25	Machine attribute h	System	0.0072
26	Machine attribute i	System	0.0072

Next, we use MI as the basis to exclude less-informative task attributes. The selected attributes are shown in Table 5. Here we only identify various machine attributes with different symbols because they are obscured in the original dataset. We use an elbow method to select task attributes with higher MI, above a threshold identified by the data. Arranging all the attributes in descending order according to their MI, the elbow method splits the entire group at the 26-th attribute followed by an MI drop from 0.0072 to 0.0050 and discards the tail. The reason for choosing such a splitting point instead of an earlier drop-off point is that we prefer to keep as many attributes as possible within our computational capacity for modeling.

5. Results

This section presents the results from applying our method on the Google cluster dataset for failure prediction and follow-up actions aimed at CCS reliability and efficiency improvement.

5.1. Failure prediction results and evaluation

Our method is applied on a daily basis. Each iteration contains a training phase, followed by a prediction phase. We use 7 days' data to train the model to keep model consistency because the system shows a 7-day periodical pattern (Garraghan et al., 2014). For example, on day i, we use the data collected from day $i-7$ to day $i-1$ to train a TBM and apply it on the data from day i. Therefore, we build 22 training-prediction pairs from the dataset, of which 9,885,649 failures and 14,041,552 successes collectively from the 22 single day prediction datasets are used to validate prediction results.

To make a failure prediction using TBM, an appropriate classification threshold needs to be selected first. This threshold H represents the confidence level to make a prediction that an execution is risky. For example, if a leaf node of the TBM contains X% failures, we can predict an execution fitting this profile to be risky if $X > H$.

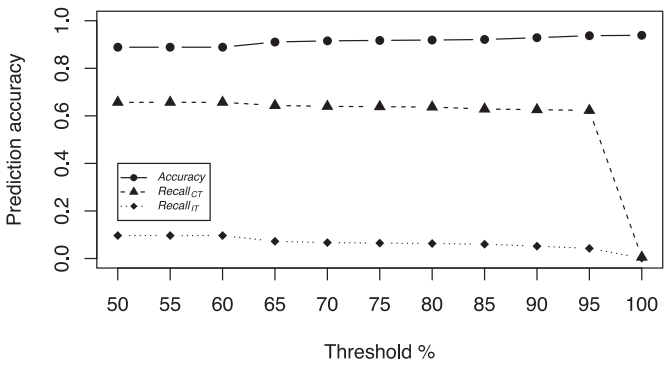


Fig. 4. Prediction accuracy over thresholds.

We examine 11 different threshold levels from 50% to 100%, with a uniform step interval of 5% to select an appropriate threshold H.

As a pilot, we trained a TBM with data from day 1 to day 7 then used the trained TBM to predict on day 8. At each threshold level, we calculated the prediction *Accuracy*, $Recall_{CT}$, and $Recall_{IT}$, with the results shown in Fig. 4. As we can see that *Accuracy* and $Recall_{IT}$ slightly increases and decreases respectively as the threshold increases, but do not show an elbow point. On the contrary, $Recall_{CT}$ keeps steady from 50% up to 95% threshold values, then significantly drops when the threshold exceeds 95%. Based on the observation of this elbow point after 95%, we set the failure identification confidence level at 95% to maximize overall model performance with the combination of the desirable high values for *Accuracy* and $Recall_{CT}$ and low value for $Recall_{IT}$.

Among the entire 9,885,649 failures, 91.21% of them would be correctly terminated ($Recall_{CT} = 91.21\%$), and 8.22% of the 14,041,552 successes would be incorrectly terminated ($Recall_{IT} = 8.22\%$) by our method. Overall prediction *Accuracy* reaches 91.55%. The prediction results show that the TBM could identify the majority of potential failures and would not negatively affect too many healthy executions. Next, we validate our method by examining its impact on CCS reliability and efficiency improvement.

5.2. Reliability and efficiency assessment and improvement

Table 6 shows the reliability and efficiency measurement results based on the simulated proactive actions of terminating high risk task executions, and compares them to that for the original dataset from both the CCS provider's perspective and the user's perspective. The table lists CCS reliability and efficiency measurements for both the simulated dataset and the original dataset. Besides that, the table also gives both the absolute and relative improvement for comparison. From the table, we can see that CCS reliability could be significantly improved from 0.5868 to 0.9155, or by 56.00%, from the CCS provider's perspective, and from 0.7867 to 0.9520, or by 21.00%, from the user's perspective. CCS efficiency also shows unanimous improvement from both the CCS provider's perspective and the user's perspective. From the provider's perspective, CCS efficiency is significantly improved by 52.17%, 35.65%, and 36.25% when measured with CPU, memory, and I/O utilization respectively. From the user's perspective, CCS efficiency improves 22.20% as measured by completion time.

Table 6
Reliability and efficiency comparison.

		Simulation	Original	Improv. (Relative)
System	R_e	0.9155	0.5868	+0.3287 (+56.00%)
	E-CPU	4.93E+10	1.03E+11	+5.39E+10 (+52.17%)
	E-Mem	1.42E+07	2.21E+07	+0.79E+07 (+35.65%)
	E-I/O	2.11E+07	3.31E+07	+1.20E+07 (+36.25%)
User	R_t	0.9520	0.7867	+0.1653 (+21.00%)
	E-Time	2.38E+09	3.06E+09	+0.68E+09 (+22.20%)

The results show that our method could significantly improve both CCS reliability and efficiency for users at the task level as well as for systems at the execution level. At the same time, it also comes with high overhead for data collection and modeling. If early risk identification method could be engaged to predict high risky executions before they start to execute, then no runtime data is required, which means our method could be implemented with lower overhead.

6. Pre-execution prediction method

This section presents a pre-execution prediction variation of our method that requires less data. For clarity, the base approach described in the earlier part of this paper is referred to as the *in-execution* case while the current variation is referred to as the *pre-execution* case.

6.1. Modifications to our method

CCS applications consistently produce a large amount of trace data during operation. Runtime data is closely linked to CCS failures as demonstrated by their high MI values in Table 5. However, the magnitude of the data can be a big challenge for data collection, processing, and modeling. For data collection, task usage data, which is the main part of runtime data, account for over 90% of the storage consumption. For dynamic in-execution risk identification, the runtime data is formulated into runtime crosscuts, accounting for 84.44% of all data instances. In order to make our method more practical, a possible solution is to diagnose high risk tasks before their actual execution and take the proactive action of not scheduling them for execution at all to avoid requiring the heavy runtime monitoring.

Fig. 5 shows a more practical, light-weight implementation of our method, the pre-execution case. Based on the basic framework of the in-execution case, this variation uses a subset of the previous data and applies the trained model in advance to the scheduling phase to avoid execution of the identified risky tasks. TBM is also used here to train predictive models following the same setup as for the in-execution case.

We exclude the runtime attributes from the dataset for the pre-execution case, because task execution is examined at scheduling phase, with runtime attributes still unavailable at that point. Therefore, only the 3 internal attributes and the 17 system attributes shown in Table 5 are used in this case, while excluding the 6 runtime attributes. Comparing to the in-execution case, the current one has less data requirements, which reduces the overhead on data collection and storage. Additionally, using fewer variables could also reduce modeling computation cost. On the mentioned ManeFrame super computing environment, TBM modeling time could be reduced from close to an hour for the in-execution case to around two minutes for the pre-execution case.

We apply the trained model on task scheduling phase to examine tasks to be scheduled before their execution. Failure counting and resource consumption criteria for the pre-execution case are shown in Table 7. Different from the in-execution case, the resource consumption by the terminated tasks should not be counted at all in simulation because these tasks will not be scheduled for execution. We use the same method as we did in the in-execution case to determine an appropriate classification threshold H. With similar results, we set failure identification confidence level at 95% for the pre-execution case.

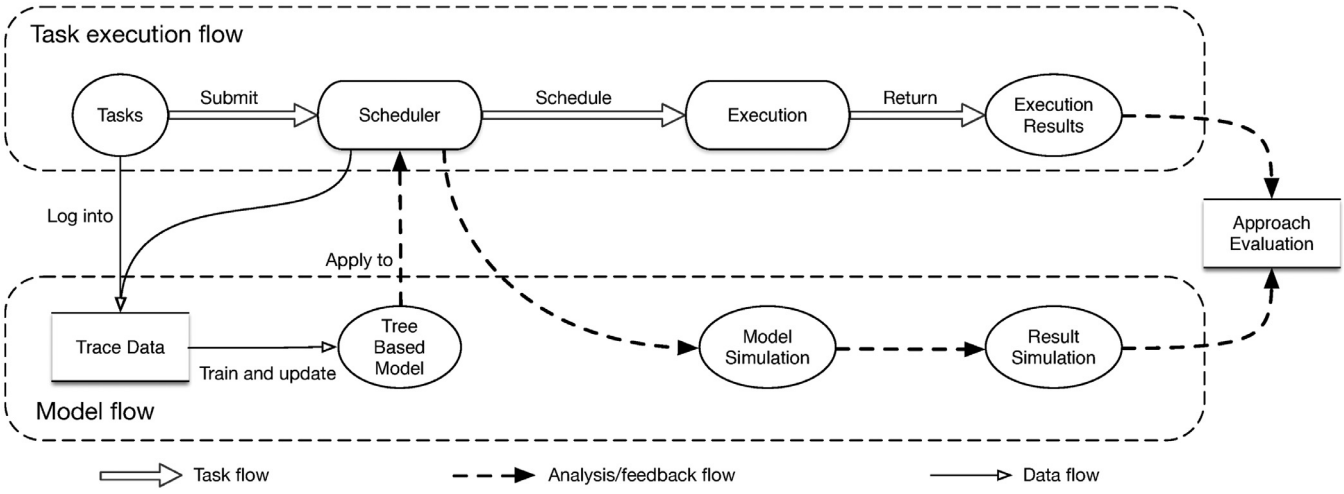


Fig. 5. Pre-execution prediction method overview.

Table 7
Failure counting and resource consumption criteria, pre-execution case.

	Original dataset		Simulation dataset		
	Outcome	Consumption U_o	$Outcome_e$	$Outcome_t$	Consumption U_m
Correct Termination	Failure	Consumption	Success	Not Counted	No Consumption
Incorrect Termination	Success	Consumption	Failure	Not Counted	No Consumption
Correct Pass	Success	Consumption	Success	Success	Consumption
Incorrect Pass	Failure	Consumption	Success	Failure	Consumption

Table 8
Reliability and efficiency comparison, pre-execution case.

		Simulation	Original	Improv. (Relative)
System	R_e	0.6950	0.5868	+0.1082 (+18.42%)
	E-CPU	6.42E+10	1.03E+11	+2.94E+10 (+37.71%)
	E-Mem	1.70E+07	2.21E+07	+0.51E+07 (+23.31%)
	E-IO	2.52E+07	3.31E+07	+0.79E+07 (+23.85%)
User	R_t	0.8467	0.7867	+0.0600 (+ 7.63%)
	E-Time	2.55E+09	3.06E+09	+0.51E+09 (+16.91%)

6.2. Results and evaluation

The pre-execution case has lower overall risk identification *Accuracy* of 69.50%, comparing to that of 91.55% for the in-execution case. The pre-execution case identifies less actual failures with $Recall_{CT}$ of 28.55% (91.21% for the in-execution case), implying lower method benefit. But on the other hand, only 1.68% of the healthy executions would be incorrectly terminated by the pre-execution case (8.22% for the in-execution case), implying lower incorrect termination overhead. The *Accuracy* decrease is likely caused by the fact that the pre-execution case utilizes fewer data attributes for modeling and prediction, especially excluding the runtime monitoring data which are more closely related to failures, both chronologically and symptomatically.

We use the same reliability and efficiency metrics for the in-execution case to evaluate the pre-execution case. Table 8 summarizes simulated system reliability and efficiency improvement for the pre-execution case. As we can see, the pre-execution case could consistently improve CCS reliability from 0.5868 to 0.6950, or by 18.42%, at the execution level, and from 0.7867 to 0.8467, or by 7.63%, at the task level. The pre-execution case also consistently improves system efficiency by saving 37.71% of CPU utilization, 23.31% of memory utilization, 23.85% IO bandwidth, and reducing completion time by 16.91% for cloud users. On the other hand, the pre-execution case is out-performed in both reliability and efficiency improvements by the in-execution case. The main reason for that is the decreased prediction accuracy due to the exclusion of the runtime group attributes for TBM modeling. In fact, as can be seen in Table 5 earlier, the runtime attributes make up 6 out of the top 17 attributes ranked by their MI values in relation to failures. Therefore, the pre-execution case shows lower risk identification accuracy, which negatively influences the follow-up actions aimed at CCS reliability and efficiency improvement.

To summarize, we developed a variation to our original method in this section, the pre-execution case, aimed at lowering data requirement for applying our method by identifying risky tasks ahead of their actual execution. The results on the Google cluster dataset shows both consistent CCS reliability and efficiency improvement at the execution level as well as at the task level. Even though the pre-execution case results in lower predictability and less improvement to reliability and efficiency compared to the in-execution case, it has achieved its primary goal of requiring significantly less data collection effort and significantly reducing modeling time.

7. Discussions

By integrating risk identification techniques commonly used in the software engineering community with followup proactive actions commonly used by the CCS providers, our method improves CCS reliability and efficiency by dynamically diagnosing high risk CCS tasks and then proactively terminating them to prevent CCS failures and reduce CCS computational resource consumption. For that purpose, we trained TBM for CCS failure prediction as the basis for follow-up proactive actions. The impact of these actions on reliability and efficiency can be analyzed, simulated, and compared to the baseline results. Results of applying our method on the Google cluster dataset showed our method to be viable and effective in improving CCS reliability and efficiency consistently, from both the CCS user's perspective and the provider's perspective.

These validation results on the generic and representative Google cluster dataset suggested our method to be generalizable to similar cloud computing systems.

We also developed a variation of our original method to produce early predictions and to take early termination actions at the task scheduling time to reduce the data requirement and the modeling overhead. The simulation results also showed the modified variation to be consistently effective in improving CCS reliability and efficiency from both the CCS user's perspective and the provider's perspective, although the improvement is not as significant as the original method as a consequence of the reduced prediction accuracy from the trained TBM using reduced data. Overall, this pre-execution variation of our method provides a more practical, light-weight implementation of our method with slightly reduced benefit on reliability and efficiency improvement.

The main external threat to the validity of this study comes from the Google cluster dataset and the analysis, risk identification, proactive termination, and impact simulation we performed on this dataset. Many attributes of the dataset are obscured, which makes it almost impossible for us to interpret discovered failure patterns or to conduct follow-up defect removal activities. However, this limitation does not appear to be a big obstacle for the practical application of our method in industry, because companies or CCS providers would have full access to their own CCS data and interpretations. In the cases where fog computing (Numhauser, 2012; Pereira et al., 2019; Xu et al., 2016) is integrated into the CCS, location and user oriented data can be integrated into the overall data set to provide the basis for failure prediction and corresponding proactive actions.

Another external threat is the representativeness of the Google cluster environment. Execution level reliability, at 58%, appears to be lower than the reliability of most industry-strength software systems (Lyu, 1996; Musa et al., 1987; Tian, 1995). We need to investigate this issue further, and need to apply our method to other CCS environments, including those involving fog computing or edge computing, to demonstrate its applicability, adaptability, and generalizability.

A related external threat is the lack of live, in-operation validation of our method. Because of the practical difficulties of applying new research results in operational CCS, the impact of our method is only evaluated based on simulated results on the original data in this study. Although we have carefully considered and justified the simulation decisions related to failure counting and resource consumption, there may still be unanticipated impact or unintended consequences on CCS operations. Therefore, we plan to apply our method online to operational CCS environments in the future in collaboration with selected CCS service providers, collect actual impact data after the proactive actions have been applied, further validate and fine tune our method for practical applications and deployment. The actual impact measurement can also be used to cross-validate our impact simulation or to fine tune it for impact assessment prior to future deployment of our method in other CCS.

Internal threats come from several aspects, including: 1) implementation overhead of our method, 2) the selection of TBM as our risk identification technique over others, 3) using MI to select data attributes for our TBM, and 4) the specific proactive actions to be performed.

First, our method is performed with overhead for data collection, processing, model training, and online diagnosis. All the data are extracted from existing CCS server logs, which would not incur extra overhead in data collection. As part of data processing, data screening could be skipped in practical applications by extracting only certain types of data deemed useful by research and prior applications. Online diagnosis only consumes limited system resources, which would not be a major problem. The main overhead comes from model training, which can also be partially solved by off-line training, consuming resource in a way that does not compete or interfere with normal CCS operations.

Second, we used tree based models (TBM) as our risk identification method instead of other methods. Researches using reference models or neural network models have shown their effectiveness (Tak et al., 2016; Chen et al., 2014; Islam and Manivannan, 2017; Liu et al., 2017). Whereas this study focuses more than merely identifying risky cloud tasks, but also cloud failure pattern detection and comprehension, and follow-up quality improvement activities. TBM has its own advantage in structural interpretation, which helps CCS provider to understand cloud failure patterns and triggers. Such an understanding forms the basis of follow-up proactive actions aimed at reliability and efficiency improvement, as well as additional offline actions for defect causal analysis, fault removal, process improvement, and preventive actions for future projects. Another reason to select TBM is its compatibility with mixed types of data. Comparing to other models, TBM could handle numerical and categorical data, both of which are commonly used in the cloud environment, at the same time. We will explore possibilities of combining the merits of other predictive models with our method to further improve prediction accuracy that may lead to additional improvement to CCS reliability and efficiency.

Third, we used MI combined with the elbow method to select data attributes for our TBM. Comparing to other methods, for example, principle component analysis (PCA) (Guan and Fu, 2013), our method uses the original data variables for better interpretation and comprehension. At the same time, our method could deal with both categorical data and numerical data simultaneously, where both types of data are captured for the CCS environment. PCA reduces the dimensionality of numerical data and uses around 5 transformed variables or principal components to produce highly stable and robust models. To be on the safe side, not to omit useful data attributes, we used the elbow method to select a significantly larger number of 26 variables in our study. In the future, we plan to convert categorical attributes to numerical ones so that we could explore the impact of using alternative data attribute selection and data dimensionality reduction approaches in our method.

Last but not least, we handle diagnosed CCS risks by simply terminating and resubmitting them, while there could be other failure mitigation and fault tolerance alternatives. To combine our new method with replication techniques (Yu et al., 2015; Bonvin et al., 2010), the negative impact of incorrect terminations could be diminished as the replicas could succeed. The new method can also be implemented on fog computing systems (Numhauser, 2012; Pereira et al., 2019; Xu et al., 2016) with off-line modeling to adapt to the light-weight computational environment. To combine with VM migration techniques (Hines et al., 2009; Shribman and Hudzia, 2012), we could transfer diagnosed risky executions to more reliable environments, which may prevent the executions from similar failures, thus provide incremental reliability improvement, and further reduce system resource consumption.

Our future work will focus on addressing the above limitations. We plan to refine our method for application to other representative CCS environments, including those involving fog computing or edge computing, and explore the use and integration with other post failure amendment and fault tolerance techniques into our method. To further ascertain the practical impact of our method, we plan to apply our method online to operational CCS environments, collecting actual impact data, instead of using simulated data as in the current study. We also plan to tune our method by choosing different risk identification techniques and different data attribute selection methods for a wide variety of application scenarios.

8. Conclusions

Cloud computing systems (CCS) integrate a huge amount of computational resources and provide configurable solutions to end users (Mell and Grance, 2011). Due to their complexity and magnitude, failures are inevitable, which waste system resources and reduce system reliability (Sharma et al., 2016; Garraghan et al., 2014; Ponemon Institute, 2016). Failure mitigation and fault tolerance mechanisms, such as migrating to other computation node or retrying failed execution, also consume system resources, resulting in system efficiency reduction (Bonvin et al., 2010; Belalem and Limam, 2013; Hines et al., 2009; Shribman and Hudzia, 2012). Therefore, to discover key failure patterns and to identify key attributes that can predict failures through appropriately selected and constructed predictive models are important (Islam et al., 2012; Tak et al., 2016; Chen et al., 2014). Diagnosing high-risk areas of the system can lead to focused follow-up actions that could improve reliability and efficiency.

To address the above problems, we developed a failure risk based CCS reliability and efficiency improvement method by combining risk identification with follow-up proactive actions. The development follows quality improvement paradigm (QIP) (Oivo and Basili, 1992) by setting the Google cluster dataset as the baseline to develop and evaluate our method. We trained tree based models previously used in software engineering (Tian, 1995; Kiciman and Fox, 2005) to diagnose risky cloud executions then proactively terminate them. The impact of these actions on reliability and efficiency was analyzed, simulated, and compared to the baseline results. Results on the Google cluster dataset demonstrated our method to be consistently effective in both CCS reliability and efficiency improvement from both the CCS user's and provider's perspectives. We also developed a variation of our method to diagnose risks early to provide a more practical, light-weight implementation of our method.

To the best of our knowledge, our method presented in this paper is the first complete solution integrating cloud failure risk prediction and follow-up proactive actions to address both CCS reliability and efficiency improvement from both CCS user's and provider's perspectives. However, this study has its own limitations. The representativeness of the Google cluster dataset is a concern, given the dataset showing an unusually high failure rate comparing to traditional commercial systems. To demonstrate the generalizability, adaptability, and effectiveness of our method, we plan to tailor our method to make it applicable to and actually apply it to other representative CCS environments, including those involving fog computing or edge computing (Numhauser, 2012; Pereira et al., 2019; Xu et al., 2016), and measuring its impact on reliability and efficiency based on actual data instead of simulation. We will explore the use and integration of other risk identification techniques (Tak et al., 2016; Chen et al., 2014; Islam and Manivannan, 2017; Liu et al., 2017) and data attribute selection methods (Guan and Fu, 2013). We also plan to explore the use of alternative proactive actions, such as replication, recovery and VM migration (Yu et al., 2015; Bonvin et al., 2010; Elnozahy et al., 2002; Liu et al., 2009; Belalem and Limam, 2013; Hines et al., 2009; Shribman and Hudzia, 2012), instead of merely terminating risky executions to further improve the overall CCS reliability and efficiency.

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.

CRediT authorship contribution statement

Yuli Tian: Conceptualization, Methodology, Software, Formal analysis, Writing - original draft, Investigation. **Jeff Tian:** Conceptualization, Methodology, Supervision, Writing - review & editing, Resources. **Ning Li:** Conceptualization, Methodology, Writing - review & editing, Resources.

Acknowledgment

This study was supported in part by the National Basic Research Program (China 2018YFB1003403), Natural Science Basic Research Plan in Shaanxi Province of China (2018JM6086), and NSF Net-Centric Software and Systems IUCRC (U.S.). Yuli Tian would thank China Scholarship Council for sponsoring his visiting to Southern Methodist University from Nov. 2017 to Apr. 2019 while working on this research.

References

Belalem, G., Limam, S., 2013. An approach to fault tolerance in the cloud using the checkpointing technique. Int. J. Commun. Netw. Distrib. Syst. 11 (3), 236–249.
Bonvin, N., Papaioannou, T.G., Aberer, K., 2010. A Self-organized, Fault-tolerant and Scalable Replication Scheme for Cloud Storage. In: ACM Symposium on Cloud Computing, pp. 205–216.
Chen, X., Lu, C.-D., Pattabiraman, K., 2014. Failure Prediction of Jobs in Compute Clouds: A Google Cluster Case Study. In: IEEE International Symposium on Software Reliability Engineering Workshops, pp. 341–346.
Cover, T.M., Thomas, J.A., 2012. Elements of Information Theory. John Wiley & Sons.
Elnozahy, E.N., Alvisi, L., Wang, Y., Johnson, D.B., 2002. A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34 (3), 375–408.
Fu, S., 2010. Failure-aware resource management for high-availability computing clusters with distributed virtual machines. J. Parallel Distrib. Comput. 70 (4), 384–393.
Garraghan, P., Townend, P., Xu, J., 2014. An Empirical Failure-analysis of a Large-scale Cloud Computing Environment. In: IEEE International Symposium on High-Assurance Systems Engineering, pp. 113–120.
Guan, Q., Fu, S., 2013. Adaptive Anomaly Identification by Exploring Metric Subspace in Cloud Computing Infrastructures. In: IEEE International Symposium on Reliable Distributed Systems, pp. 205–214.
Hines, M.R., Deshpande, U., Gopalan, K., 2009. Post-copy live migration of virtual machines. ACM SIGOPS Oper. Syst. Rev. 43 (3), 14–26.
Islam, S., Keung, J., Lee, K., Liu, A., 2012. Empirical prediction models for adaptive resource provisioning in the cloud. Future Gener. Comput. Syst. 28 (1), 155–162.
Islam, T., Manivannan, D., 2017. Predicting Application Failure in Cloud: A Machine Learning Approach. In: IEEE International Conference on Cognitive Computing, pp. 24–31.
ISO/IEC 25010, 2011. Systems and software engineering - systems and software quality requirements and evaluation - system and software quality models.
Kiciman, E., Fox, A., 2005. Detecting application-level failures in component-based internet services. IEEE Trans. Neural Networks 16 (5), 1027–1041.
Liu, C., Han, J., Shang, Y., Liu, C., Cheng, B., Chen, J., 2017. Predicting of job failure in compute cloud based on online extreme learning machine: a comparative study. IEEE Access 5, 9359–9368.
Liu, H., Jin, H., Liao, X., Hu, L., Yu, C., 2009. Live Migration of Virtual Machine Based on Full System Trace and Replay. In: ACM International Symposium on High Performance Distributed Computing, pp. 101–110.
Mell, P., Grance, T., 2011. The NIST definition of cloud computing. Tech. Rep. Special Publication 800-145, National Institute of Standards and Technology.
Lyu, M.R., 1996. Handbook of Software Reliability Engineering. IEEE computer society press CA.
Moreno, I.S., Garraghan, P., Townend, P., Xu, J., 2014. Analysis, modeling and simulation of workload patterns in a large-scale utility cloud. IEEE Trans. Cloud Comput. 2 (2), 208–221.
Musa, J., Iannino, A., Okumoto, K., 1987. Software Reliability: Measurement, Prediction, Application. McGraw-Hill, New York.
Nelson, E., 1978. Estimating software reliability from test data. Microelectron. Reliab. 17 (1), 67–73.
Numhauser, J.B.-M., 2012. Fog Computing: Introduction to a New Cloud Evolution. In: Cies III Congress, p. 111126.
Oivo, M., Basili, V.R., 1992. Representing software engineering models: the TAME goal oriented approach. IEEE Trans. Software Eng. 18 (10), 886–898.
Pereira, J., Ricardo, L., Luís, M., Senna, C.R., Sargento, S., 2019. Assessing the reliability of fog computing for smart mobility applications in VANETs. Future Gener. Comput. Syst. 94, 317–332.
Ponemon Institute, 2016. Cost of data center outages. 1–20.
Reiss, C., Wilkes, J., Hellerstein, J.L., 2011. Google cluster-usage traces: format + schema. Tech. rep., Google Inc..
Salfner, F., Lenk, M., Malek, M., 2010. A survey of online failure prediction methods. ACM Comput Surv 42 (3), 10.

Sharma, Y., Javadi, B., Si, W., Sun, D., 2016. Reliability and energy efficiency in cloud computing systems: survey and taxonomy. J. Netw. Comput. Appl. 74, 66–85.
Shribman, A., Hudzia, B., 2012. Pre-copy and post-copy VM live migration for memory intensive applications. In: European Conference on Parallel Processing, pp. 539–547.
Tak, B.C., Tao, S., Yang, L., Zhu, C., Ruan, Y., 2016. Logan: problem diagnosis in the cloud using log-based reference models. In: IEEE International Conference on Cloud Engineering, pp. 62–67.
Tian, J., 1995. Integrating time domain and input domain analyses of software reliability using tree-based models. IEEE Trans. Softw. Eng. 21 (12), 945–958.
Xu, Y., Mahendran, V., Radhakrishnan, S., 2016. Towards SDN-based fog computing: MQTT broker virtualization for effective and reliable delivery. In: IEEE International Conference on Communication Systems and Networks, pp. 1–6.
Yu, X., Ning, P., Vouk, M.A., 2015. Enhancing security of hadoop in a public cloud. In: IEEE International Conference on Information and Communication Systems, pp. 38–43.
Zhu, X., Yang, L.T., Chen, H., Wang, J., Yin, S., Liu, X., 2014. Real-time tasks oriented energy-aware scheduling in virtualized clouds. IEEE Trans. Cloud Comput. 2 (2), 168–180.