Automated Intelligent Healing in Cloud-Scale Data Centers

Rui Li[1*], Zhinan Cheng[2*], Patrick P. C. Lee[2], Pinghui Wang[3∧], Yi Qiang[1],
Lin Lan[3], Cheng He[1], Jinlong Lu[1], Mian Wang[1], Xinquan Ding[1]
[1]*Alibaba Group* [2] *The Chinese University of Hong Kong* [3] *Xi'an Jiaotong University*

***Abstract*—Modern cloud-scale data centers necessitate *self-healing* (i.e., the automation of detecting and repairing component failures) to support reliable and scalable cloud services in the face of prevalent failures. Traditional policy-based self-healing solutions rely on expert knowledge to define the proper policies for choosing repair actions, and hence are error-prone and non-scalable in practical deployment. We propose AIHS, an automated intelligent healing system that applies machine learning to achieve scalable self-healing in cloud-scale data centers. AIHS is designed as a full-fledged, general pipeline that supports various machine learning models for predicting accurate repair actions based on raw monitoring logs. We conduct extensive trace-driven and production experiments, and show that AIHS achieves higher prediction accuracy than current self-healing solutions and successfully fixes 92.4% of the total of 33.7 million production failures over seven months. AIHS also reduces 51% of unavailable time of each failed server on average compared to policy-based self-healing. AIHS is now deployed in production cloud-scale data centers at Alibaba with a total of 600 K servers. We open-source a Python prototype that reproduces the self-healing pipeline of AIHS for public validation.**

I. Introduction

Cloud-scale data centers in production are susceptible to various types of component failures (e.g., hardware crashes, software bugs, network disconnection, and performance anomalies). To maintain high availability of commercial cloud services at scale, modern cloud-scale data centers often support *self-healing* [4], [14], which refers to the automation of the detection and repair of component failures with limited human intervention.

Enabling efficient and scalable self-healing in production environments is challenging. First, the accurate diagnosis of failures is non-trivial in the face of numerous types of failures, and such failures can also occur abruptly at any time. Second, automating the repair process in a timely manner is difficult, since multiple repair actions are possible for each type of failure. Choosing an incorrect repair action inevitably increases the repair cost. For example, for a transient server failure, it is preferable to simply reboot the server instead of sending human operators to manually check and fix the server, as the latter unnecessarily delays the repair process and adds extra manual efforts.

Traditional self-healing solutions are *policy-based*, in which human experts specify the policies that map failures to the appropriate repair actions. Such policies heavily rely on expert knowledge based on the observed historical failures, and hence cannot be readily updated to reflect emerging failures. To mitigate the dependence on expert knowledge in repair decision making, some production self-healing solutions (e.g., Microsoft AutoPilot [8], [15] and Facebook FBAR [18], [19]) reportedly adopt *machine learning* to *predict* repair actions based on historical data (see Section VI for details). While machine learning is widely believed as an effective tool for data-driven prediction, there remains limited analysis of machine-learning-based self-healing solutions in real-world cloud-scale data center deployment. In particular, some key deployment questions are unexplored, such as: (i) how a complete machine-learning-based self-healing pipeline should be deployed; (ii) how the prediction accuracy of self-healing varies across different machine learning models; and (iii) how different stages of a machine-learning-based self-healing pipeline affect the overall prediction accuracy of self-healing.

In this practical experience report, we propose AIHS, an Automated Intelligent Healing System that applies machine learning to self-healing in cloud-scale data centers. AIHS provides a full-fledged, general pipeline that supports various machine learning models, and predicts a repair action for a given failure based on the learning of raw monitoring logs and historical repair actions. Specifically, it employs a three-stage workflow: (i) it first uses unsupervised machine learning to transform the raw monitoring logs in text form into numerical features; (ii) it feeds the numerical features into supervised machine learning for the prediction of a repair action; and (iii) it further uses supervised machine learning to predict how likely a predictive repair action can fix the failures, such that only the repair actions with a high probability of success will be executed.

AIHS is currently deployed in production cloud-scale data centers with about 600 K servers at Alibaba. To validate the effectiveness of AIHS, we conduct extensive evaluation via both trace-driven and production experiments. Our evaluation shows that AIHS outperforms state-of-the-art self-healing solutions and can successfully repair 92.4% of a total of 33.7 million production failures over a seven-month span. In particular, AIHS reduces 51% of unavailable time of each failed server compared to policy-based self-healing. AIHS also effectively addresses emerging failures that cannot be readily solved by policy-based self-healing solutions. Finally, we discuss the insights and lessons learned from our design and deployment of AIHS, so as to help the community better understand machine-learning-based self-healing.

Due to proprietary concerns, we cannot publicize the production traces and the production implementation of AIHS. Instead, we built an AIHS prototype in Python

∗ The two authors contribute equally.
∧ Corresponding author: Pinghui Wang (phwang@mail.xjtu.edu.cn).

that can reproduce the complete three-stage workflow of AIHS based on synthetic traces for public validation. The source code of our AIHS prototype now is available at the repository: **https://github.com/alibaba-edu/dcbrain/tree/master/AIHS_prototype**.

II. Background

In this section, we introduce our cloud infrastructure and its challenges of supporting self-healing.

Cloud infrastructure. We first provide an overview of our cloud infrastructure atop which AIHS is deployed. Our cloud infrastructure comprises 22 data centers and runs about 150 applications. Each data center hosts multiple servers that are organized in clusters, each of which serves one application. Our cloud infrastructure includes 5,100 clusters with a total of 1.1 million servers. Currently, AIHS is deployed across 600 K servers in 2,200 of the 5,100 clusters.

Monitoring system. Our cloud infrastructure includes a *monitoring system* that centrally monitors all servers in real time and collects *raw monitoring logs* from each server. The raw monitoring logs are represented in the form of *attributes*, which describe the operational status and error information of a server based on multiple sources of logs, such as system logs and key performance indicators (KPIs). Currently, the monitoring system collects 165 attributes that cover different dimensions of operational status and can be categorized as follows:

- *Hardware attributes*: Each hardware attribute describes the operational status of a hardware component. For example, the `DiskHang` attribute reports whether the attached disk of a server is healthy.
- *Network attributes*: Each network attribute describes the network-related issues. For example, the `Ping` attribute reports whether a server can be reached by ping messages.
- *Environment attributes*: Each environment attribute describes the software environment issues. For example, the `PythonEnvCheck` attribute reports whether the Python environment of a server works normally.
- *Others*: They refer to the attributes that do not belong to the above three classes. For example, the `Load` attribute reports the CPU utilization.

Each attribute is associated with one of the six levels of failure severity (in the increasing order of severity): `info`, `good`, `warning`, `error`, `critical`, and `fatal`. For the latter three (i.e., `error`, `critical`, and `fatal`), the server is considered to be failed, and the monitoring system will trigger different types of repair actions (see below) on the failed server.

Repair actions. Our cloud infrastructure supports different repair actions to fix a failed server (similar to Microsoft AutoPilot [15]), including:

- *No operation (NOP):* It waits and queries the status of the server after a fixed time interval (e.g., 20 minutes) to see if the server returns to normal.
- *Management service restart (MSR):* It restarts the management service on the server.
- *Environment reinstallation (ER):* It reinstalls the runtime environment for software (e.g., Java Runtime Environment (JRE)) on the server. Note that any running service on the server remains unaffected.
- *Reboot (RB):* It reboots the server; before the reboot, any service hosted on the server is migrated to a different server.
- *Re-imaging (RI):* It re-images the server, by first migrating the services hosted on the server to a different server and then reinstalling the operating system of the server.
- *Return merchandise authorization (RMA):* It creates a repair ticket that notifies hardware experts to fix the failure (e.g., by replacing the hardware component).

We point out that all the above repair actions are currently supported in our cloud infrastructures. They are also commonly used for self-healing in other production data centers [8], [15].

Repair actions have different repair costs. It is thus critical to choose the right repair action with the minimum possible repair cost. While we do not specify the absolute cost of each repair action, we can compare different repair actions by their relative costs. The above repair actions are listed in the ascending order of the relative costs provided by operators.

It is infeasible to traverse and try each repair action to fix a failure, as executing a wrong repair action adds extra overhead. To understand the importance of choosing the right repair action for each failure, we consider an example when a server experiences a high CPU load. The server can actually restore to normal by itself without further repair actions. If we reboot the server (i.e., RB), the server can also restore to normal, but this incurs extra overhead of migrating the services hosted on the server to a different server and hence increases the downtime of the server. Note that a higher-cost repair action does not guarantee to repair a failure that can be repaired by a lower-cost repair action. For example, RB cannot fix software-environment-specific failures, which can be addressed by ER. Furthermore, the first five repair actions (i.e., NOP, MSR, ER, RB, and RI) can be executed automatically, while the last repair action (i.e., RMA) requires human intervention and should be avoided whenever possible.

Policy-based self-healing. Our cloud infrastructure has adopted a policy-based self-healing solution. Our experts empirically propose policies on the repair actions for different types of failures (based on the levels of failure severity of the 165 attributes), and refine the policies based on experience. Each policy triggers a specific repair action if the raw monitoring logs match the predefined text patterns of the policies. For example, one of the policies is to reboot the server when the monitoring system shows the `DiskHang` attribute with the severity `error`.

Our policy-based self-healing solution has been deployed on all 5100 clusters before AIHS is launched. The monitoring system collects a *repair record* of each triggered repair action. Each repair record contains the following fields: (i) the server ID, (ii) the raw monitoring logs, (iii) the triggered repair action, and (iv) the repair result. The repair result is marked as either *successful* if the repair action successfully recovers the server

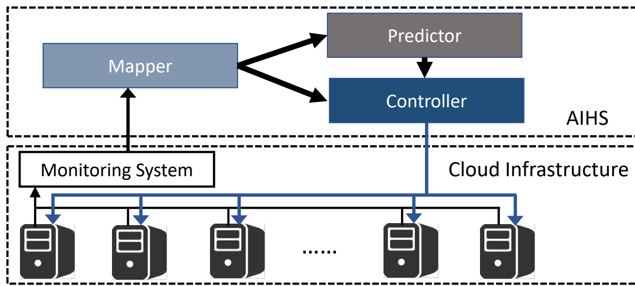


Fig. 1: AIHS architecture.

and the server works normally for more than 20 minutes, or *unsuccessful* otherwise.

Our deployment experience is that policy-based self-healing is ineffective, due to the huge number of attributes and different severity levels of each attribute. Specifically, there are a total of 165^6 possible combinations of severity levels in the attributes (recall that each of the 165 attributes has six possible levels of failure severity). Our policies currently can only cover a small fraction of failures for some combinations of severity levels, but cannot cover any emerging failure that we have not observed. It is infeasible to manually examine each failure and specify the corresponding repair action, due to the extremely large number of combinations of severity levels. If we later have new sets of attributes in different deployment cases, applying manual specification will be expensive. Even worse, our cloud infrastructure includes many clusters with varying configurations. This requires us to configure different policy sets for different clusters even for the same failure. This motivates us to explore machine-learning-based solutions to automate the entire self-healing workflow.

III. AIHS Design

In this section, we describe the complete workflow of AIHS, a machine-learning-based self-healing system for cloud-scale data centers. AIHS is designed as a general framework that supports various machine learning models. It also applies machine learning to different stages of the self-healing workflow. Figure 1 shows the architecture of AIHS, which comprises three components, namely *Mapper*, *Predictor*, and *Controller*. In the following, we provide the details of each of the components. We conclude with the deployment details of AIHS.

A. Mapper

Recall that our raw monitoring logs comprise various attributes (Section II). However, the attributes are in unstructured raw texts, while machine learning models require numerical features for prediction. Thus, we design the *Mapper* that transforms the raw monitoring logs into numerical features. Once the monitoring system detects a failure, it pushes the raw monitoring logs of the failed server to the Mapper, which generates numerical features from the raw monitoring logs and sends the numerical features to the Predictor and the Controller for choosing repair actions.

The Mapper formulates the transformation as an embedding problem in natural language processing, where the raw text is processed by unsupervised learning models and converted into numerical vectors. In our case, the Mapper treats each attribute name as a single word, and converts the attributes into feature vectors. It now supports three unsupervised embedding methods, namely Term Frequency-Inverse Document Frequency (*TF-IDF*) [16], Latent Dirichlet Allocation (*LDA*) [1], and Bidirectional Encoder Representations from Transformers (*BERT*) [5] (see Section IV-A for their comparisons).

We use LDA [1] as an example to show how the Mapper works. LDA treats a raw monitoring log as a document. It uses unsupervised machine learning to cluster attributes with similar semantics into a configurable number of groups (called *topics*). It transforms the document into a topic vector, in which each element is the probability of a topic occurring in the document. The probability of each topic is calculated based on the failure severity level of each attribute that is related to the topic.

B. Predictor

We design the *Predictor* to recommend one of the repair actions (i.e., NOP, MSR, ER, RB, RI, and RMA) for each detected failure. Specifically, it receives the numerical features of a failure from the Mapper as input. It then predicts a repair action, which is sent to the Controller as a recommendation.

The Predictor formulates a multi-class classification problem, in which each class corresponds to a repair action. The Predictor uses the triggered repair actions of historical successful repair records as labels to train the multi-class classifier, so that it can predict a repair action that fixes the similar types of failures in the past. We design the Predictor as a general framework to support various multi-class classification models, including: (i) Logistic Regression (LR), (ii) Support Vector Machine (SVM), (iii) Random Forest (RF), (iv) Gradient Based Decision Tree (GBDT), and (v) Bayes Network (BN). We compare different multi-class classifiers via trace-driven evaluation in Section IV-A.

C. Controller

The Controller takes the numerical features of a failure from the Mapper and the recommended repair action from the Predictor as input, and determines the final repair action to fix the failure. However, even if the Predictor achieves high prediction accuracy, the recommended repair action is predicted based on historical successful repair records for the past failures and it remains uncertain how likely the recommended repair action can actually fix the failure. To solve this issue, we design the *Controller* to *assess* the recommended repair action on the failure to see how likely the repair action can fix the failure. The Controller then chooses the final repair action based on the assessment.

To assess a repair action on a failure, the Controller employs six trained binary classifiers, one for each of the six repair actions (Section II). Here, the binary classifier for each repair action is trained using the repair results of all the historical repair records (both successful and unsuccessful) of the repair action as labels.

Specifically, the Controller executes the assessment workflow in three steps as follows:

- *Step 1:* It feeds the numerical features of the failure to the corresponding classifier of the recommended repair action made by the Predictor, and predicts the repair outcome (i.e., successful or unsuccessful). If the repair outcome is successful, the Controller directly triggers the recommended repair action and quits.
- *Step 2:* If the repair outcome is predicted to be unsuccessful, the Controller assesses other repair actions. It first assesses the repair action that has the lowest repair cost (Section II). If the repair outcome is predicted to be successful, the Controller triggers the repair action and quits; otherwise, it repeats the assessment on the next repair action with a higher repair cost.
- *Step 3:* If the Controller finds no repair action that returns a successful outcome, it calls the administrator to manually diagnose the failure.

The Controller can effectively solve emerging failures. Our experience is that the Predictor may not find the suitable repair action for emerging failures if the supervised machine learning models in the Predictor have not observed such failures before. By learning both successful and unsuccessful repair records of each repair action, the machine learning models in the Controller can develop knowledge about what repair actions can solve different kinds of failures. Thus, by assessing each repair action, the Controller can find the repair actions that are likely to fix emerging failures.

D. Production Deployment

AIHS is now deployed in our cloud infrastructure. We provide the deployment details of AIHS as follows.

Model training and updates. We train the machine learning models offline for AIHS before its deployment. For the Mapper, we learn the necessary hyper-parameters of the unsupervised learning models (e.g., the probability of each word belonging to which topic in LDA) based on collected repair records in the monitoring system. For the Predictor, we train its multi-class classifier using successful repair records (Section III-B). For the Controller, we train each binary classifier using both successful and unsuccessful repair records (Section III-C).

Also, we periodically update the models of AIHS in deployment. The reason is that we can collect more repair records that may improve the model accuracy of AIHS during deployment. We now check whether the model accuracy can be improved with the new repair records on a monthly basis. If the new models show improved accuracies than the old ones, we update the deployed AIHS with the updated models; otherwise, we discard the repair records.

Deployment. We have deployed AIHS on 2,200 out of 5,100 clusters (Section II). We divide the 2,200 clusters into five groups based on the services hosted by the clusters. Each group runs an AIHS instance independently (i.e., we run five AIHS instances in total), and each instance serves one type of application. For each group, we run 10 virtual machines (VMs) (with 4 vCPUs each) to respond to self-healing requests. Once a VM receives a raw monitoring log from the monitoring system, it launches a thread to sequentially run the Mapper, the Predictor, and the Controller (all of which are given the trained models) to determine the final repair action. Note that there is no interaction among different repair actions. Thus, AIHS can launch multiple threads in a VM to simultaneously handle the failures of different servers.

In our deployment, we use the same machine learning models and same model parameters for all AIHS instances and avoid manual tuning on the fly. For the monthly model updates, each instance only re-trains the models using its own collected repair records (i.e., not using the repair records from other instances). Finally, for most production failures (e.g., 98.8%; see Experiment B.3), AIHS can address them automatically without calling the manual repair action RMA. Overall, AIHS can achieve fully automated repair without human intervention for most failures.

IV. Evaluation

We conduct both trace-driven and production experiments to validate the effectiveness of AIHS. We summarize our major findings on AIHS.

- LDA achieves the highest accuracy in the Mapper, while GBDT achieves the highest accuracy in most cases in the Predictor and the Controller.
- AIHS outperforms state-of-the-art self-healing approaches [8], [19].
- AIHS achieves high production accuracy. It successfully fixes 92.4% of 33.7 million production failures over seven months. It also reduces 51% of unavailable time of each failed server on average compared to policy-based self-healing.
- AIHS effectively repairs emerging failures that cannot be solved by policy-based self-healing.
- AIHS maintains high accuracy through monthly model updates. Each AIHS instance also maintains high accuracy through the retraining of models using the repair records collected by the instance itself.

A. Analysis of Machine Learning Models

We compare different machine learning models in different components of AIHS via trace-driven experiments.

Traces. We have collected traces of repair records during the deployment of our policy-based self-healing solution (Section II) from June 2019 to October 2019. Our policy-based self-healing solution was deployed on all 5,100 clusters, from which we collected the traces. Our traces include 4.25 million repair records, whose fields are defined in Section II. The traces show that a total of 270 K servers suffer from failures during the trace period, and about 5.7% of these failed servers have unsuccessful repair records. For the failed servers with unsuccessful repair records, we observe that a single server can report up to 3,072 records during the period. The main reason is that some failures keep occurring in a server when the policy-based solution cannot repair these failures. On the other hand, 42.9% of failed servers only report a single failure during

	NOP	MSR	ER	RB	RI	RMA
Successful	2021 K	185 K	0.4 K	1.3 K	13 K	2.2 K
Unsuccessful	1501 K	520 K	1.3 K	0.3 K	1.2 K	1.6 K
Total	3522 K	705 K	1.7 K	1.6 K	13.2 K	3.8 K

TABLE I: Number of samples of each repair action in the trace.

the period and most of them (98.9%) can be successfully fixed in time.

Table I shows the number of samples of each repair action in our collected traces. Among the 4.25 million repair records, there are 2.22 million successful repair records and 2.03 million unsuccessful repair records. For different repair actions, we observe *sample imbalance* in the trace. For example, among all successful repair records, about 99% of them trigger NOP or MSR, while only less than 1% of repair records choose the remaining four repair actions (i.e., ER, RB, RI, or RMA). Considering both successful and unsuccessful repair records, there are also more than 99% of repair records choosing NOP or MSR. Finally, we observe that our policy-based self-healing approach triggers NOP for more than 83% of failures in the repair records. One main reason is that our policy-based self-healing approach triggers NOP if it cannot find any policy for the failure.

Recall that our monitoring system monitors different categories of attributes (Section II). We observe that hardware attributes and environment attributes occur in a large fraction of repair records. Specifically, 57.8% and 21.3% of repair records include hardware and environment attributes, respectively. Also, 35.8% of repair records report errors (i.e., `error` or `critical` or `fatal`) in only one attribute, while the remaining repair records have multiple attributes reporting errors. In particular, a repair record can have up to nine attributes reporting errors in the monitoring logs. This also indicates that some failures may trigger multiple attributes to report errors. Thus, it is important to understand the correlation between different attributes.

We finally verify that the same failure can be fixed by multiple repair actions. We focus on the successful repair records that only report the `Ping` attribute with value `error` and show what repair actions are adopted in these records. We observe that three repair actions can lead to successful repair. Among all records, 67.5% of records adopt MSR, 29.5% of records adopt NOP, and 3% of records adopt RB. This demonstrates that the same failure can be fixed by multiple repair actions. It is important to find the proper repair action with the minimum cost for a failure.

Methodology. We sort the repair records by time. We use the first 80% of repair records over time as the training set, while we use the remaining 20% as the testing set. Note that such a proportion of training-testing data is also used in prior work on failure prediction [2]. We consider the following metrics:

- *Precision*: It is the fraction of correctly predicted successful repair records over all (successful or unsuccessful) repair records that are predicted as successful repair records.
- *Recall*: It is the fraction of correctly predicted successful repair records over all successful repair records.
- *F1-score:* It is $\frac{2\times precision\times recall}{precision+recall}$.

	Predictor			Controller		
	Prec.	**Recall**	**F1-score**	**Prec.**	**Recall**	**F1-score**
TF-IDF	**0.97**	0.92	0.94	0.54	0.92	0.59
LDA	**0.97**	**0.97**	**0.97**	**0.87**	**0.94**	**0.90**
BERT	0.96	0.95	0.96	0.70	0.66	0.59

TABLE II: Experiment A.1 (Comparison of unsupervised learning models in the Mapper).

	Precision	Recall	F1-score
LR	0.93	0.84	0.87
SVM	0.95	0.94	0.94
GBDT	**0.97**	**0.97**	**0.97**
RF	0.94	0.91	0.92
BN	0.83	0.74	0.75

TABLE III: Experiment A.2 (Comparison of multi-class classifiers in the Predictor).

Experiment A.1 (Comparison of learning models for the Mapper). We compare the unsupervised learning models in the Mapper (i.e., TF-IDF [16], LDA [1], and BERT [5]) (Section III-A), by studying how they affect the prediction accuracies in the Predictor and the Controller. Specifically, we fix GBDT (also used in [19]) as the multi-class classifier in the Predictor and the binary classifier for each repair action in the Controller. For each GBDT classifier, we set the number of trees as 2,000. Note that we observe similar results for the classifiers other than GBDT.

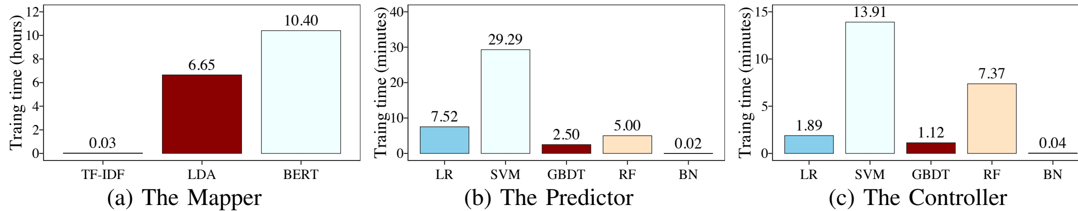
We train each learning model on the training set as follows. For TF-IDF, we use the training set to compute the IDF value for each word, which is later used to compute the TF-IDF value for each repair record. For LDA, we use Gibbs Sampling [6] to sample the topic distribution based on the training set. We set the two parameters α and β of the Dirichlet distribution in LDA as 0.1 and 0.01, respectively. We fix the number of topics as 10 and the number of sampling iterations as 1,024. For BERT, we configure the number of transformers as 10 and each transformer has five layers of neural networks. The above parameters provide good convergence in our preliminary evaluation.

Table II shows the results; in the interest of space, we only report the average of the six classifiers in the Controller. For the Predictor, all TF-IDF, LDA, and BERT achieve high prediction accuracies, with F1-scores 0.94, 0.97, and 0.96, respectively. For the Controller, LDA significantly outperforms both TF-IDF and BERT, mainly because LDA can better interpret the correlation among the attributes and allows the Controller to avoid under-fitting (see our trace analysis in Section IV-A). TF-IDF has a low precision as it does not interpret the correlations among the attributes, while BERT has both low precision and recall as it causes under-fitting in the classification models of the Controller. In summary, LDA is the most suitable model for the Mapper and provides high prediction accuracies in both the Predictor and the Controller.

Experiment A.2 (Comparison of multi-class classifiers in the Predictor). We compare different multi-class classifiers in the Predictor. We use LDA in the Mapper and set the same parameters for LDA as in Experiment A.1. For each

	NOP			MSR			ER			RB			RI			RMA		
	Pre.	**Rec.**	**F1.**	**Pre.**	**Rec.**	**F1.**	**Pre.**	**Rec.**	**F1.**	**Pre.**	**Rec.**	**F1.**	**Pre.**	**Rec.**	**F1.**	**Pre.**	**Rec.**	**F1.**
LR	0.99	0.96	0.98	0.99	0.93	0.96	0.02	0.23	0.04	0.44	0.36	0.40	0.77	0.70	0.73	0.51	0.42	0.46
SVM	**1.00**	0.96	0.98	0.97	**0.93**	0.95	0.03	0.37	0.05	0.37	0.62	0.46	0.79	0.83	0.81	0.61	0.47	0.53
GBDT	0.99	**0.98**	**0.98**	**1.00**	0.93	**0.96**	0.83	**0.98**	0.90	0.68	**0.93**	**0.79**	**0.95**	0.91	0.93	**0.75**	0.90	**0.82**
RF	0.96	0.98	0.97	**1.00**	0.93	0.96	**0.86**	**0.98**	**0.92**	**0.74**	0.73	0.74	0.94	0.94	**0.94**	0.71	0.92	0.80
BN	0.78	0.95	0.86	0.94	0.90	0.92	0.86	0.88	0.87	0.56	0.91	0.70	0.90	**0.97**	0.93	0.53	**1.00**	0.69

TABLE IV: Experiment A.3 (Comparison of classifiers in the Controller).



(a) The Mapper (b) The Predictor (c) The Controller

Fig. 2: Experiment A.4 (Training time of models in different components of AIHS).

classifier, we tune its learning rate and the number of iterations to make it convergent on the training set. Recall that we use the successful repair records for training a multi-class classifier (Section III-B). In our evaluation, we use the classifiers to predict a repair action for each successful repair record in the testing set and compare the predicted result with the original repair action in the repair record as the ground truth.

Table III shows the results for different classifiers. GBDT achieves the highest precision, recall, and F1-score in all cases. The result is also consistent with [19]. One major reason is that GBDT can better solve the sample imbalance problem of our collected traces (Section IV-A).

Experiment A.3 (Comparison of binary classifiers in the Controller). We compare different binary classifiers in the Controller. We again use LDA in the Mapper as in Experiment A.1. For each repair action, we train the corresponding binary classifier using all the related repair records. We evaluate the prediction accuracy of each binary classifier on the testing set. Here, we predict whether a repair action in a repair record can successfully fix the failure, while we use the repair result of the repair record as the ground truth.

Table IV shows the results for different binary classifiers on each repair action. Both GBDT and RF generally achieve the highest accuracy. Nevertheless, for the sake of simplicity and ease of maintenance, we tend to choose a single model for all repair actions. We find that GBDT performs the best for most repair actions. Thus, we also choose GBDT for the Controller in our deployment.

Experiment A.4 (Training time). We study the training times of different machine learning models for each component of AIHS. We train all models in a single machine equipped with an Intel Xeon 2.6 GHz 32-core E5-2650 CPU, 256 GB RAM, and an NVIDIA Tesla P4 GPU. Note that the dataset is loaded into memory before all experiments. We train all the models without using the GPU except for the BERT model. For machine learning models in the Mapper, Predictor, and Controller, we set the parameters of each model the same as in Experiments A.1, A.2, and A.3, respectively.

Figure 2 shows the training times of machine learning models in different components of AIHS. We observe that the models in the Mapper take up to 10 hours for training, while the models in the Predictor and Controller only take less than one hour for training. For the models we choose for our business, LDA takes 6.65 hours, while GBDT takes less than 3 minutes for training. Note that training is done offline, so the overhead of the model training for AIHS is not a burden in our case.

B. Analysis of AIHS in Deployment

We now evaluate the accuracy of AIHS in deployment using both trace-driven and production experiments.

Methodology. We consider the following metrics for evaluating the accuracy of AIHS. All metrics vary from zero to one; the higher is better.

- *HitRate:* It is the fraction of successful repair records that are correctly predicted by AIHS (i.e., AIHS predicts the same repair action as indicated in the repair record) over all successful repair records.
- *ReverseRate:* It is the fraction of unsuccessful repair records that are predicted differently by AIHS (i.e., AIHS predicts a different repair action as indicated in the repair record) over all unsuccessful repair records. Note that the ReverseRate only indicates how likely AIHS can avoid a wrong repair action, while it cannot guarantee the predicted repair action can successfully fix the failure.
- *OverallRate:* It combines HitRate and ReverseRate, i.e., $\frac{2 \times HitRate \times ReverseRate}{HitRate + ReverseRate}$.
- *SuccessRate:* It is the fraction of production failures that are successfully fixed over all production failures that are handled. Note that for a failure that cannot be fixed but continues reporting the same monitoring logs, we count it as one failure.

By default, we adopt LDA in the Mapper and GBDT in both the Predictor and Controller. We set the parameters for LDA and GBDT as described in Section IV-A.

Experiment B.1 (Micro-benchmarking). We first benchmark different components of AIHS. As running micro-benchmarking in production data centers can affect our services, we instead simulate the online decisions of AIHS using the testing set in Section IV-A. We consider the following settings.

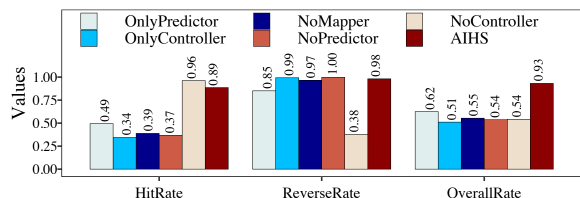


Fig. 3: Experiment B.1 (Micro-benchmarking).

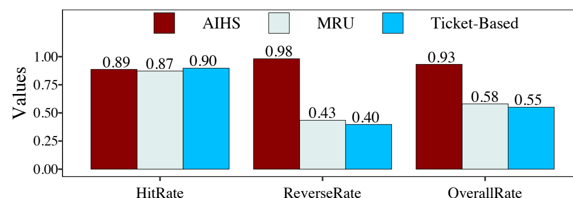


Fig. 4: Experiment B.2 (Comparisons with state-of-the-art approaches).

- *AIHS*: It is the whole system with all components.
- *NoController:* It disables the Controller and directly uses the recommended repair actions from the Predictor.
- *NoPredictor:* It disables the Predictor, meaning that the Controller directly assesses each repair action based on its repair cost to choose a repair action.
- *NoMapper*: It disables the Mapper, in which the raw monitoring log is directly transformed into a vector with 165 elements. Each element corresponds to an attribute, and has value one if the attribute shows a failure severity of `error`, `critical`, or `fatal`, or has value zero otherwise.
- *OnlyPredictor:* It processes the raw monitor log as *NoMapper* and only uses the Predictor to find a repair action.
- *OnlyController:* It processes the raw monitor log as *NoMapper* and only uses the Controller to choose a repair action.

Figure 3 shows the results. We first observe that the Mapper and Predictor are important for a high HitRate. Once we disable one or both of them, the HitRate drops to lower than 50%. Disabling the Controller can also improve the HitRate, mainly because some unsuccessful repair records are similar to successful repair records and the Controller inevitably drops the HitRate when trying to avoid them. On the other hand, the Controller is important for the ReverseRate. The ReverseRate drops to 38% once we disable the Controller. In summary, AIHS achieves high HitRate, ReverseRate, and OverallRate by combining all components to learn both successful and unsuccessful repair records.

Experiment B.2 (Comparison with state-of-the-art solutions). We compare AIHS with two state-of-the-art self-healing solutions [8], [19]:

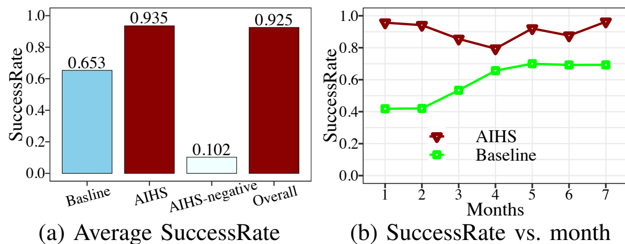
- *Most Recent Used (MRU)* [8], in which the most recent successful repair action that fixes the same types of failures are adopted.
- *Ticket-based* [19], in which the repair action is predicted by GBDT based on the successful historical repair records for the same types of failures. It also uses TF-IDF to transform raw monitoring logs into numerical features.

Since the source codes of both solutions are not public, we implement the two solutions based on their descriptions in the paper for comparison. For MRU, we trace the repair records of the last 30 days and choose a repair action based on the successful repair record that has the most similar monitoring logs with the current failure. For Ticket-based, we compute the TF-IDF value as described in Section IV-A and train the GBDT the same as in the Predictor of AIHS. To avoid affecting our services in production data centers, we again simulate the online decisions of different approaches using the testing set in Section IV-A.

Figure 4 shows the results. We observe that all approaches achieve a similar HitRate. For example, MRU, Ticket-based, and AIHS achieve a HitRate of 87%, 90%, and 89%, respectively. However, both MRU and Ticket-based achieve a very low ReverseRate (i.e., less than 45% in both cases), mainly because they only predict the repair action without assessing how likely a repair action can fix the failures. In contrast, AIHS not only predicts a repair action for the failure, but also assesses how likely the repair action can fix the failure. Thus, AIHS achieves a ReverseRate of up to 98%, resulting in an OverallRate of 93%.

Experiment B.3 (Production accuracy). We now study the effectiveness of AIHS on our 2,200 production clusters. Due to stability concerns and business requirements, we currently limit the permission of AIHS to execute only a subset of repair actions, although we finally target to allow AIHS to execute all repair actions. Until now, AIHS is only allowed to execute three types of repair actions, namely NOP, MSR, and RB. The 2,200 clusters now run both AIHS and a policy-based solution. If AIHS recommends other unavailable repair actions (i.e., ER, RI, and RMA), the raw monitoring logs are passed to the policy-based solution for finding a repair action; we call this approach *AIHS-negative*. Note that the policy-based solution under AIHS-negative may still recommend NOP, MSR, or RB for fixing these failures. Nevertheless, the three actions for AIHS cover the majority of failures. Among all 33.7 million production failures produced by the 2,200 clusters in our AIHS deployment, 98.8% of failures can be handled by AIHS and only 1.2% of failures are handled by AIHS-negative.

Figure 5(a) shows the average SuccessRate of AIHS during its seven-month deployment, compared to the average SuccessRate of the policy-based solution (called the *Baseline*). The Baseline reports the average SuccessRate on the remaining 2,900 clusters that purely adopt our policy-based self-healing approach without AIHS. We observe that the Baseline only fixes 65.3% of failures, as our policies cannot cover emerging failures. In contrast, among the failures that can be handled by AIHS, 93.5% of them can be fixed. In other words, AIHS itself successfully fixes 92.4% of production failures (i.e., 98.8%$\times$93.5%) during its deployment. For the remaining failures handled by AIHS-negative, 10.2% of them can be further fixed. Combining AIHS and AIHS-negative together (called the *Overall*), we can solve 92.5% of production failures (i.e., 98.8%$\times$93.5%+1.2%$\times$10.2%).



(a) Average SuccessRate (b) SuccessRate vs. month

Fig. 5: Experiment B.3 (Production accuracy).

Figure 5(b) further compares the monthly SuccessRate of AIHS with that of the Baseline. We observe that the SuccessRate of policy-based self-healing slightly increases in later months of deployment. The reason is that our experts propose new policies that help handle some of the new failures occurring in earlier months. On the other hand, the SuccessRate of AIHS shows a slight drop in some months, mainly because we gradually increase the number of clusters to deploy AIHS during that period. Nevertheless, AIHS significantly outperforms the policy-based solution at all times.

We also evaluate how AIHS reduces the unavailable time of a failed server (i.e., the duration that the failed server cannot serve the hosted service) compared to Baseline. We focus on a specific instance. For the six months before the clusters of the instance launch AIHS (i.e., Baseline), we observe 1,853 failed servers (note that a server may fail multiple times) and the unavailable time of each failed server is 20.1 hours on average. While for the first six months after the clusters launch AIHS (i.e., Overall), we observe 1,547 failed servers and the average unavailable time of each failed server is only 9.8 hours. The results indicate that AIHS can reduce 51% of unavailable time of each failed server (i.e., (20.1-9.8)/20.1=51%) by triggering a more accurate repair action for each failure compared to Baseline.

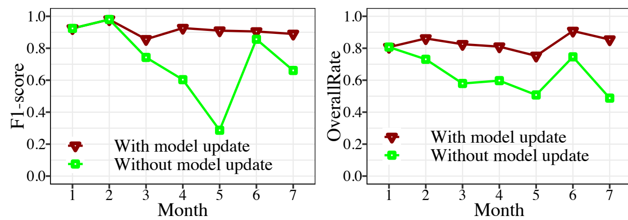
Finally, we also observe that Baseline takes less than 1 ms to determine a repair action for a failure, while AIHS needs 7 ms on average. Nevertheless, the response time is still negligible in production.

Experiment B.4 (Case study). We study how AIHS outperforms the policy-based solution via a case study. We focus on a specific type of failure, the `Ping` failure, i.e., when the `Ping` attribute reports failures. We compare the average SuccessRate of the `Ping` failure of AIHS with that of the policy-based self-healing during the seven-month deployment.

Table V shows the results. We observe that AIHS fixes 94.2% of `Ping` failures while the policy-based self-healing only fixes 54.8% of `Ping` failures over seven months. To examine the reason, we decompose the proportion of repair actions that each approach takes to fix the `Ping` failures (here we only focus on NOP, MSR, and RB). We observe that the policy-based self-healing triggers NOP in most of the cases, e.g., in 92.4% of `Ping` failures. The reasons are two-fold: (i) our experts tend to believe that some `Ping` failures can be automatically self-recovered, so some rules map the failures to NOP; (ii) our policy-based self-healing approach always triggers NOP if there is no policy matching the emerging failures. Nevertheless,

	SuccessRate	Proportion of each repair action		
		NOP	MSR	RB
Policy-based	0.548	0.924	0.067	0.009
AIHS	0.942	0.643	0.345	0.012

TABLE V: Experiment B.4 (Case Study): SuccessRate of the `Ping` failure and proportion of repair actions for the `Ping` failure



(a) F1-score of the Predictor (b) OverallRate

Fig. 6: Experiment B.5 (Effectiveness of monthly model update).

for the latter cases, NOP generally cannot fix the failures and results in a low SuccessRate of policy-based self-healing. In contrast, AIHS shows that about 35% of `Ping` failures should be handled by taking effective repair actions such as MSR or RB. As a consequence, AIHS can effectively fix up to 94.7% of `Ping` failure.

We further trace the three-stage workflow of AIHS to understand how it works on emerging failures. We observe that the Predictor also recommends NOP to the Controller for most emerging failures in our case. However, the assessment results of NOP in the Controller indicate that NOP cannot fix most emerging failures. As a result, the Controller upgrades the repair action to MSR or RB, which can effectively repair these emerging failures related to the `Ping` failure. In summary, our case study shows that AIHS can effectively solve emerging failures that are not covered by our expert policies.

Experiment B.5 (Effectiveness of monthly model updates). We study how monthly model updates (§III-D) improve the accuracy of AIHS. We focus on a specific AIHS instance, and compare the accuracy when we enable or disable monthly model updates in the instance.

Figure 6 shows the results. We first compare the F1-score of the Predictor in AIHS (Figure 6(a)). We observe that monthly model updates can effectively maintain a high F1-score of the Predictor by allowing the Predictor to learn the successful repair records of the emerging failures in the last month and accurately predict repair actions for these failures. On the other hand, if we disable monthly model updates, the F1-score of the Predictor drops dramatically. For example, the F1-score of the Predictor drops from 0.93 in the first month to 0.35 in the fifth month due to emerging failures.

We further compare the OverallRate of AIHS during the seven-month deployment (Figure 6(b)). We observe that disabling monthly model updates reduces the OverallRate of AIHS by 23% on average. In contrast, enabling monthly model updates allows AIHS to maintain a high OverallRate in all months. In summary, AIHS effectively mitigates the impact of emerging failures and maintains high prediction accuracy via monthly model updates.

Finally, we note that the time to deploy an updated model

Instance	OverallRate	
	Trained with repair records from own instance	Trained with repair records from all instances
A	0.971	0.835
B	0.978	0.949
C	0.912	0.776
D	0.985	0.881
E	0.946	0.846

TABLE VI: Experiment B.6 (AIHS instance accuracy): OverallRate of each instance when re-training models using repair records collected from a single instance and all instances.

in our production takes about 5 minutes. The old model can still serve self-healing requests while a new model is being deployed. Thus, the model update operation has a limited impact on production.

Experiment B.6 (AIHS instance accuracy). Our current deployment treats each AIHS instance independently, such that the model training of each instance is independent of the other instances. We justify this deployment strategy by studying how the repair records collected from different AIHS instances affect the accuracy of each AIHS instance. We focus on all five AIHS instances (denoted by A, B, C, D, and E) that are currently deployed for our business (Section III-D). For each AIHS instance, we compare the OverallRate of the following two cases: (i) the models of the instance are re-trained only using the repair records collected from the instance itself; (ii) the models are re-trained using the repair records aggregated from all five deployed instances.

Table VI shows the results. We observe that each instance achieves a higher OverallRate when only using the repair records collected from itself to re-train the models. Aggregating repair records from all five instances to re-train the models actually reduces the OverallRate of each instance (e.g., by 0.1 on average), since the repair records collected from different AIHS have different statistical properties.

V. Lessons Learned

In this section, we summarize the insights and lessons that we learned from the design and deployment of AIHS.

Designing AIHS as a general framework. The earlier version of AIHS is designed with specific machine learning models as in existing studies [8], [19]. However, we find that such a design is unfriendly when we need to compare different models for self-healing, especially when we compare different combinations of machine learning algorithms in the self-healing pipeline. Thus, we turn to design AIHS as a general framework to support different machine learning algorithms in each component. Such a design allows us to find the appropriate machine learning models for self-healing for different deployment environments in our business.

Machine learning model selection. In our business, we choose LDA for the Mapper, as LDA can better interpret the correlation among different attributes (§IV-A). For other cases, such as when there is no correlation between different attributes, our experience is that TF-IDF works better in the Mapper. Finally, for BERT, we find that it is suitable for the cases with a large training set, so as to avoid over-fitting the models. On the other hand, for the Predictor and Controller, we find that almost all machine learning models achieve high accuracy. For example, SVM, GBDT, and RF all have an F1-score larger than 0.9 in the Predictor. Our business finally uses GBDT in both the Predictor and the Controller, as GBDT better fits our dataset that shows sample imbalance (§IV-A). GBDT also shows high accuracy in other studies [18], [19].

Incremental deployment of AIHS. We deploy multiple AIHS instances in our business, in which each instance serves one type of application. Our goal is to gradually deploy AIHS in our production data centers, such that only when existing AIHS instances work as expected, we launch new instances for new groups of applications. One observation is that the accuracy of each AIHS instance can only be improved using the new repair records collected from the instance itself, while aggregating the repair records from all AIHS instances cannot improve the accuracy. The main reason is that the statistical properties of the repair records collected from the instances are different, due to the use of different applications (Experiment B.6). Thus, we keep multiple AIHS instances in our business, and conduct monthly model updates for each instance to maintain high accuracy (Experiment B.5).

Scalability and fault tolerance. We argue that AIHS also achieves scalability and fault tolerance. In our current scale, we use 10 VMs for each AIHS instance, where the self-healing requests are distributed to different VMs via a load-balancer. Thus, we can simply increase the number of VMs when the scale of the data centers increases. Also, if any VMs crash, the load-balancer can redirect the requests to other VMs for fault tolerance.

VI. Related Work

Failure detection and diagnosis. Prior studies explore failure detection and diagnosis in distributed systems [9], [11], [13], [17], [22], [23], [25], [26]. Gunter et al. [9] treat failure detection as a network anomaly detection problem, by storing logs in a summary data structure and using anomaly detection to find the failures. Xu et al. [26] combine log parsing with text mining to extract features from logs using principal component analysis. Falcon [17] enables fast detection by coordinating spy modules to monitor different system layers. CloudDiag [22] applies statistical techniques to pinpoint the errors for cloud performance diagnosis. Panorama [13] enhances system observability based on the interactions across system components. The above systems mainly focus on detecting and diagnosing failures in distributed systems, but do not address how to pick the proper repair actions to fix failures as in AIHS.

Machine learning for improving the reliability of data centers. Machine learning has recently been widely used to improve the reliability of data centers, such as in disk management and network management. For example, some studies explore machine learning to predict hard disk failures in data centers based on disk logs [2], [10], [21]. Some studies

use machine learning to predict network anomalies or classify network faults in Internet-based services [20], cellular networks [24], and cable broadband networks [12]. The above studies focus on applying machine learning to predict the presence of failures, while AIHS uses machine learning to predict repair actions for the detected failures.

Self-healing in data centers. Some studies propose new self-healing approaches for data centers in the literature. SymbioticSphere [3] applies biological principles to improve the adaptability and survivability of server farms. Dai et al. [4] propose a hybrid diagnosis tool to characterize different failure symptoms. Xue et al. [27] propose a prediction model to predict performance tickets and use the results to guide the self-healing of performance issues in data centers. RADAR [7] includes a self-repairing component to monitor alert events of CPU and memory and fixes errors based on some predefined repair actions. However, all these approaches are only evaluated in simulated environments, but not deployed in production.

Both Microsoft [8], [15] and Facebook [18], [19] provide automated healing services for their production data centers. Microsoft Autopilot [15] initially provides automated repair services by choosing repair actions to fix the server failures based on simple heuristics. It is later extended with the logistic regression model to predict if a repair action is likely to fix a server failure [8]. Facebook FBAR and its extensions [18], [19] consider different automated repair actions without human intervention to fix a failure. If the failure cannot be solved by any automated repair action, a repair ticket is created for human experts to further investigate the failure. To help experts debug the undiagnosed failure in a ticket, FBAR employs machine learning to predict the repair actions for a ticket by learning the similarity of various repair tickets [19]. AIHS provides a general framework that applies various machine learning models in different stages of a self-healing pipeline. In addition to predicting repair actions as in Microsoft AutoPilot and Facebook FBAR, AIHS also assesses how likely a repair action fixes a failure (Section III-C).

VII. Conclusion

This paper presents AIHS, a machine-learning-based automated intelligent healing system for cloud-scale data centers. AIHS provides a general framework to support various machine learning models for self-healing. We conduct both trace-driven experiments and production experiments to validate the effectiveness of AIHS. Our evaluation shows that AIHS outperforms state-of-the-art self-healing solutions and successfully fixes 92.4% of production failures of 600 K production servers over seven months. Our AIHS prototype is now open-sourced.

Acknowledgments. This work was supported by Alibaba Group via the Alibaba Innovation Research (AIR) program.

References

[1] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3(Jan):993–1022, 2003.

[2] M. M. Botezatu, I. Giurgiu, J. Bogojeska, and D. Wiesmann. Predicting disk replacement towards reliable data centers. In *Proc. of ACM SIGKDD*, pages 39–48, 2016.

[3] P. Champrasert and J. Suzuki. A biologically-inspired autonomic architecture for self-healing data centers. In *Proc. of IEEE COMPSAC*, pages 103–112, 2006.

[4] Y. Dai, Y. Xiang, and G. Zhang. Self-healing and hybrid diagnosis in cloud computing. In *Proc. of IEEE CLOUD*, pages 45–56, 2009.

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-raining of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[6] E. I. George and R. E. McCulloch. Variable selection via gibbs sampling. *Journal of the American Statistical Association*, 88(423):881–889, 1993.

[7] S. S. Gill, I. Chana, M. Singh, and R. Buyya. Radar: Self-configuring and self-healing in resource management for enhancing quality of cloud services. *Concurrency and Computation: Practice and Experience*, 31(1):e4834, 2019.

[8] M. Goldszmidt, M. Budiu, Y. Zhang, and M. Pechuk. Toward automatic policy refinement in repair services for large distributed systems. *ACM SIGOPS Operating Systems Review*, 44(2):47–51, 2010.

[9] D. Gunter, B. L. Tierney, A. Brown, M. Swany, J. Bresnahan, and J. M. Schopf. Log summarization and anomaly detction for troubleshooting distributed systems. In *Proc. of EEE/ACM GRID*, pages 226–234, 2007.

[10] S. Han, P. P. Lee, Z. Shen, C. He, Y. Liu, and T. Huang. Toward adaptive disk failure prediction via stream mining. In *Proc. of IEEE ICDCS*, 2020.

[11] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The ϕ accrual failure detector. In *Proc. of IEEE SRDS*, pages 66–78, 2004.

[12] J. Hu, Z. Zhou, X. Yang, J. Malone, and J. W. Williams. Cablemon: Improving the reliability of cable broadband networks via proactive network maintenance. In *Proc. of USENIX NSDI*, pages 619–632, 2020.

[13] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *Proc. of USENIX OSDI*, pages 1–16, 2018.

[14] Y. Huang and C. M. R. Kintala. Software implemented fault tolerance technologies and experience. In *Proc. of IEEE FTCS*, pages 2–9, 1993.

[15] M. Isard. Autopilot: Automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[16] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 1972.

[17] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proc. of ACM SOSP*, pages 279–294, 2011.

[18] F. Lin, M. Beadon, H. D. Dixit, G. Vunnam, A. Desai, and S. Sankar. Hardware remediation at scale. In *Proc. of IEEE/IFIP DSN*, pages 14–17, 2018.

[19] F. Lin, A. Davoli, I. Akbar, S. Kalmanje, L. Silva, J. Stamford, Y. Golany, J. Piazza, and S. Sankar. Predicting remediations for hardware failures in large-scale datacenters. In *Proc. of IEEE/IFIP DSN*, pages 13–16, 2020.

[20] D. Liu, Y. Zhao, H. Xu, Y. Sun, D. Pei, J. Luo, X. Jing, and M. Feng. Opprentice: Towards practical and automatic anomaly detection through machine learning. In *Proc. of ACM IMC*, pages 211–224, 2015.

[21] F. Mahdisoltani, I. Stefanovici, and B. Schroeder. Proactive error prediction to improve storage system reliability. In *Proc. of USENIX ATC*, pages 391–402, 2017.

[22] H. Mi, H. Wang, Y. Zhou, M. R.-T. Lyu, and H. Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Trans. on Parallel and Distributed Systems*, 24(6):1245–1255, 2013.

[23] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proc. of USENIX NSDI*, pages 353–366, 2012.

[24] L. Pan, J. Zhang, P. P. Lee, H. Cheng, C. He, C. He, and K. Zhang. An intelligent customer care assistant system for large-scale cellular network diagnosis. In *Proc. of ACM SIGKDD*, pages 1951–1959, 2017.

[25] C. Schneider, A. Barker, and S. Dobson. Autonomous fault detection in self-healing systems: Comparing hidden markov models and artificial neural networks. In *Proc. of International Workshop on Adaptive Self-tuning Computing Systems*, pages 24–31, 2014.

[26] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan. Mining console logs for large-scale system problem detection. *SysML*, 8:4–4, 2008.

[27] J. Xue, R. Birke, L. Y. Chen, and E. Smirni. Spatial-temporal prediction models for active ticket managing in data centers. *IEEE Trans. on Network and Service Management*, 15:39–52, 2018.