This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3327583

PREVENT: An Unsupervised Approach to Predict Software Failures in Production

Giovanni Denaro[*], Rahim Heydarov[+], Ali Mohebbi[+], Mauro Pezzè[+,#]

[*] University of Milano-Bicocca
Milano, Italy

[+] Università della Svizzera Italiana (USI)
Lugano, Switzerland

[#] Constructor Institute
Schaffhausen, Switzerland

Email: giovanni.denaro@unimib.it, {rahim.heydarov, ali.mohebbi, mauro.pezze}@usi.ch

***Abstract*—This paper presents PREVENT, a fully unsupervised approach to predict and localize failures in distributed enterprise applications.**

Software failures in production are unavoidable. Predicting failures and locating failing components online are the first steps to proactively manage faults in production. Many techniques predict failures from anomalous combinations of system metrics with supervised, weakly supervised, and semi-supervised learning models. Supervised approaches require large sets of labelled data not commonly available in large enterprise applications, and address failure types that can be either captured with predefined rules or observed while training supervised models.

PREVENT integrates the core ingredients of unsupervised approaches into a novel fully unsupervised approach to predict failures and localize failing resources. The results of experimenting with PREVENT on a commercially-compliant distributed cloud system indicate that PREVENT provides more stable, reliable and timely predictions than supervised learning approaches, without requiring the often impractical training with labeled data.

I. INTRODUCTION

Good design and quality assurance practice cannot prevent software to fail in production [24]. The complexity of distributed enterprise applications further increases the risks of production failures.

Several approaches apply machine learning techniques to prevent system failures, following many studies that exploit artificial intelligence for software engineering problems [4], [5], [7], [12], [17], [25], [33], [35], [36], [41], [51], [53], [61], [65]–[67]. The two mainstream classes of approaches either predict failure prone modules based on metrics that reflect the complexity of the code or predict the occurrence of error states at runtime based on metrics that reflect the execution of software systems.

Approaches that predict failure prone modules feed prediction models with metrics from the code, and allow for fine-tuning testing activities for the modules that are more likely to contain defects [25], [67]. A notable case of approaches that predict failure prone modules is the Nam and Kim's CLAMI approach [48] that clusters software modules based on the similarity of code complexity metrics.

Approaches that predict the occurrence of error states at runtime feed prediction models with metrics monitored at runtime, and allow to take countermeasures before the failures actually manifest. These approaches draw on the observation that many failures occur in production when the execution of some faulty statements corrupts the execution state, and eventually the error state propagates to a system failure, that is, a deviation of the delivered service from the required functionality.

Current approaches for predicting failures exploit rule-based, signature-based, or semi-supervised strategies. *Rule-based approaches* rely on predicates that experts extract from data observed during operations [13]. *Signature-based approaches* rely on supervised learning models that leverage the information from historical records of previously observed failures [6], [15], [19], [23], [31], [40], [49], [50], [57], [62]. Signature-based approaches require large amounts of labeled failure data for training, which are rarely available and hard to collect. *Semi-supervised approaches* exploit signature-based models on top of synthetic data inferred with either semi-supervised, weakly supervised or unsupervised learning, to balance accuracy and required information [22], [27], [43], [63], [64]. Signature-based approaches for distributed applications often aim also to localize the components responsible for the failures [13], [28], [39], [42], [43], [56], [63].

This paper investigates a *purely unsupervised* approach to failure prediction. Whereas signature-based approaches identify failures of pre-defined failure types that occur in the training data, *unsupervised* approaches detect anomalies as deviations from some model of the nominal system behavior suitably inferred in absence of failures, and can thus reveal failures of any types, including types that do not necessarily correspond to training data. Previous work explored unsupervised anomaly detection for intrusion attacks [8], anomalies of streaming data [2], issues in computer networks [20], log file analysis [18] and detection of performance issues [29].

In this paper we propose PREVENT, an original application of unsupervised approaches to predict and localize failures in distributed enterprise applications. PREVENT is grounded on the lessons learned from PREMISE, EMBED and LOUD, our previous work on supervised and unsupervised approaches to predict failures and localize faults.

PREMISE [43] combines anomaly detection technique and Logistic Model Trees (LMT) to predict failures and localize failing resources in supervised fashion. EMBED [46], [47] proposes Restricted Boltzmann Machine (RBM) to predict failures in complex cloud systems. LOUD [42] combines Granger causality analysis and page rank centrality measures to localize failing components. While PREMISE offers a supervised end-to-end solution to predict failures and localize faults, EMBED and LOUD propose unsupervised approaches to predict failures and locate faults, respectively. Predicting the occurrence of failures at the system level without localizing the failing components in distributed applications (as in EMBED) does not offer developers and maintainers enough information to activate prevention mechanisms. Relying only on localization mechanisms (as in LOUD) results in many false alarms.

PREVENT originally combines failure prediction and failure localization in a coordinated and fully unsupervised approach. It exploits a deep autoencoder model to predict failures, and coordinates deep autoencoder, Granger causality and page rank centrality to locate failing components. In a nutshell, The deep autoencoder identifies anomalous states of the system, and discriminates normal and anomalous metrics. The Granger causality analysis combined with page rank centrality effectively ranks the values that the deep autoencoder reveals as anomalous, to spot the faulty components responsible for the predicted failures.

We introduce a new large dataset that we obtained by running many experiments on REDIS, a commercially-compliant, distributed cloud system. The experiments that we discuss in the paper compare the stability, reliability and earliness of PREVENT with PREMISE, EMBED and LOUD, in the context of failures that we seed in REDIS.

PREVENT recomputes the prediction at each time interval. We measure the stability of a prediction as the true positive rate after the first true prediction, that is, the continuity in correctly reporting the failing component in the presence of error states. Intuitively, predictions consecutively raised at all timestamps over a time interval are clearer messages of failures than intermittent predictions that appear and disappear in a time interval. We measure the reliability of the prediction with reference to the false positive rate, that is, the frequency of alarms that do not correspond to real error states or wrongly track the failures to non-failing components: The lower the false positive rate, the higher the reliability of the prediction. Intuitively, predictions are more effective when they occur only in error states than when they occur both in error and correct states. We measure the earliness of a prediction as the time interval between the first correct prediction and the actual failure, that is, the time interval for a proactive action on the failing component to prevent a failure before its occurrence.

This paper contributes to the research in software engineering by

- defining PREVENT, a new unsupervised approach that originally combines deep autoencoder, Granger causality and page rank centrality to predict failures and locate the corresponding faulty components.
- presenting a large set of data collected from a commercially-compliant, distributed cloud systems to evaluate and compare different approaches, data that we offer in a replication package,[1]
- comparing PREVENT with PREMISE, EMBED and LOUD, to indicate the advantages of a fully unsupervised approach for both predicting failures and locating faulty components with respect to supervised approaches.

This paper is organized as follows. Section II presents PREVENT. Section III describes the experimental setting, and discusses the results of the experiments that comparatively evaluate PREVENT with respect to PREMISE, EMBED and LOUD. Section IV discusses the main state-of-the-art approaches for predicting and diagnosing failures, and their relation with PREVENT. Section V summarizes the contribution of the paper and indicates novel research directions.

II. PREVENT

The core contribution of this paper is PREVENT, an unsupervised approach that both predicts failures in distributed enterprise applications and localizes the corresponding faulty components. PREVENT originally combines a deep autoencoder, Granger causality analysis and pagerank centrality analysis to predict failures and localize faulty components without requiring training with labeled data. With this original combination of unsupervised techniques, PREVENT overcomes the main obstacles of supervised approaches: the effort required to label the data for training and the difficulty of predicting failures of types that do not correspond to training data. The unsupervised nature of PREVENT allows to both predict failures of any type and locate the faulty components responsible for the failures. Efficiently locating the faulty components in large distributed enterprise applications provides enough information for state-of-the-art self healing approaches [60] to automatically activate healing actions before the occurrence of the failures.

Figure 1 shows the main components of PREVENT, the *State classifier* and the *Anomaly ranker*, that predict failures and localize the faulty components, respectively. As shown in the figure, the *State classifier* and the *Anomaly ranker* work on time series of Key Performance Indicators, *KPI*s, that are sets of metric values observed by monitoring the application at regular time intervals (every minute in our experiments). A *KPI* is a pair $\langle metric, node \rangle$ of a metric value collected at either a virtual or physical node of the monitored application. Our current prototype of PREVENT collects *KPI* series with a monitoring facility built on top of Elasticsearch [1].

PREVENT returns a list of anomalous nodes ranked by anomaly relevance, list that the *Anomaly ranker* produces in the presence of anomalous states inferred with the *State classifier*.

Both the *State classifier* and the *Anomaly ranker* include a deep autoencoder that requires unsupervised training with KPI data collected during normal (non-failing) executions of the application, without requiring any labels at training time. The deep autoencoder can be trained with a reasonably small amount of data. In our experiments we train PREVENT

[1] The replication package at https://star.inf.usi.ch/#/software-data/14

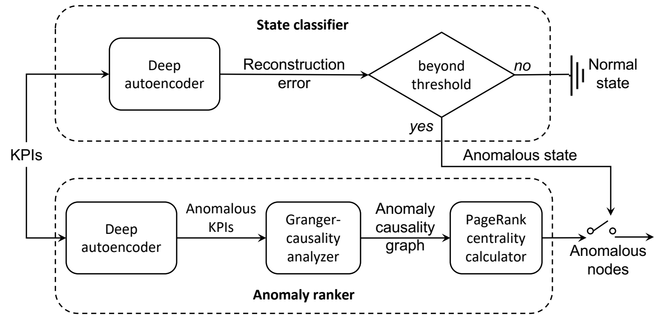

Fig. 1. Overview of PREVENT

with data collected in two weeks of normal execution. Thus, PREVENT is resilient to concept drifts that span over several weeks or more. Such drifts can be overtaken with both frequent short retraining sessions and continuous training in production, thanks to the unsupervised nature of PREVENT.

The *State classifier* crosschecks the KPI values observed in production against the normal-execution characteristics inferred during training, and accepts *normal states*, when there are no significant differences. It pinpoints *anomalous states*, otherwise. The *Anomaly ranker* identifies anomalous KPIs, that is, KPIs with values that significantly differ from values observed during training in normal execution conditions, and ranks anomalous nodes according to their relevance with respect to the anomalous KPIs. PREVENT reports the anomalous nodes only when the *State classifier* reveals anomalous states. We discuss the inference models that instantiate the two components later in this section.

We defined PREVENT by benefitting from the lessons learned with PreMiSe [43], EmBeD [46], [47], and LOUD [42], three representative techniques to predict and localize failures.

The supervised PreMiSe approach that we developed as a joint project with industrial partners, gives us important insights about the strong limitations of supervised approaches in many industrially-relevant domains. PreMiSe combines an unsupervised approach for detecting anomalous KPIs with a supervised signature-based approach for predicting failures. The signature-based approach requires long supervised training, and achieves good precision only for failures of types considered during supervised training. PreMiSe indicates that supervised approaches can indeed precisely and timely predict failures, localize faults and identify the fault types. It also highlights the strong limitations of training systems with seeded faults in production, as supervised approaches require.

EmBeD predicts failures by both computing the Gibbs free energy associated with the KPIs that represent the system state and monitoring anomalous energy fluctuations in production. EmBeD provides evidence about the correlation between anomalous energy values of the RBM and KPI anomalies that can lead to system failures. EmBeD predicts failures of the target application as a whole, without any information at the level of the application nodes.

LOUD localizes faults with an unsupervised observational strategy, by identifying the application nodes that are highly relevant with respect to the causal dependencies between the observed anomalies. It assumes the availability of precise anomaly predictions at system-level to limit the otherwise large amount of false alarms. LOUD shows how to combine Granger causality analysis and page rank centrality to effectively locate faulty components, in the presence of reliable failure predictions.

PREVENT introduces a deep autoencoder to improve the precision of detecting and classifying anomalous KPIs, and proposes an original combination of the approaches as an efficient end-to-end solution to predict failures and locate faulty components with lightweight unsupervised training.

We implemented two *State classifier*s that we refer to as PREVENT$_A$ ($Prevent_{Autoencoder}$) and PREVENT$_E$ ($Prevent_{Energy}$). PREVENT$_A$ implements PREVENT as illustrated in Figure 1. PREVENT$_E$ replaces the Deep Autoencoder of the *State classifier* in Figure 1 with a Restricted Boltzmann Machine (RBM) that implements the free-energy-based approach of EmBeD. We use PREVENT$_E$ to compare PREVENT with EmBeD. Both PREVENT$_A$ and PREVENT$_E$ are integrated with the *Anomaly ranker* that combines Granger-causality with eigenvector-centrality (page rank centrality calculator) to precisely localize failing nodes.

A. PREVENT$_A$ *State Classifier*

The PREVENT$_A$ State classifier uses a deep autoencoder to predict failures without requiring training with seeded faults. The deep autoencoder model identifies anomalous KPIs as KPIs with values that are anomalous with respect to the observations when training with normal executions.

A deep autoencoder (also simply referred to as autoencoder) is a neural network architected with two contiguous sequences of layers that mirror each other structure: A first sequence of layers of decreasing size up to an intermediate layer of minimal size, and a second sequence of layers of correspondingly increasing size up to a layer with the same size of the initial layer.

During training, the first half of the network learns how to encode the input data in incrementally condensed form, up to a minimal form in the intermediate layer. The second half of the network learns how to regenerate the input based on the information condensed in the intermediate layer. The difference between the input and output values is the *reconstruction error* of the autoencoder.

During training the neurons of the network learn functions that minimize the average reconstruction error on the training data. In production, the network returns small reconstruction errors for data similar to the training data in both absolute values and mutual correlations. It returns large reconstruction errors for data that significantly differ from the observations in the training phase.

Figure 2 illustrates the architecture of the PREVENT$_A$ autoencoder that is composed of seven layers with sizes n, $n/2$,

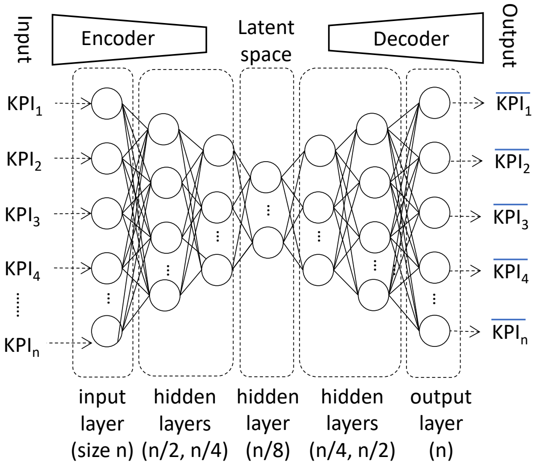

Fig. 2. Deep autoencoder as instantiated in PREVENT$_A$

$n/4$, $n/8$, $n/4$, $n/2$, and n, respectively, being n is the number of monitored KPIs.

We trained the PREVENT$_A$ autoencoder with the KPIs observed at regular time intervals on the distributed enterprise application executed in normal conditions, that is, without failures. The trained autoencoder reveals the anomalous states that emerge in production, as the states that correspond to reconstruction errors that significantly differ from the mean reconstruction error computed during training. In our current prototype this threshold is set to the reconstruction errors that differ from the mean reconstruction error for more than three times the standard deviation observed on the training set.

B. PREVENT$_E$ *State Classifier*

The PREVENT$_E$ State classifier infers anomalous combinations of KPI values from perturbations of the Gibbs free energy computed on time series of KPI values monitored from production.

The Gibbs free energy was originally introduced in statistical physics to model macroscopic many-particle systems as a statistical characterization of the properties of single particles that affect the global physical quantities such as energy or temperature [10], and is applied in many domains, such as the growth of the World Wide Web and the spread of epidemics [16].

We rely on the intuition that complex distributed enterprise software applications and physical systems share the dependencies of properties of interest of the global state of the system (energy and temperature in the case of physical systems, failures in the case of software applications) on the collective configuration of basic elements (particles in the case of physical systems, KPI values in the case of software applications). Intuitively, the execution of some faulty code produces some error states with anomalous KPI values that propagate through the execution, thus leading to a progressive alignment of anomalous KPI values. Following this intuition, we recast the problem of predicting failures in complex distributed enterprise software applications to the problem of revealing the collective alignment of the KPIs of an application to correlated anomalous values.

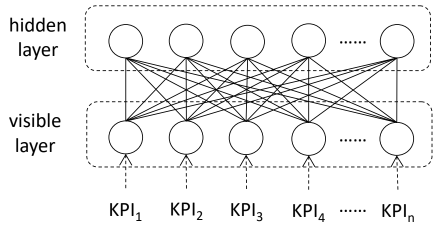

Fig. 3. RBM as instantiated in PREVENT$_E$

PREVENT$_E$ supersedes the intractability of analytically representing the physical dependencies that concretely govern the correlations among KPIs [9] by approximating the computation of the Gibbs energy with restricted Bolzmann machines, *RBM* [21], and signals anomalous states when the energy exceeds a threshold.

Figure 3 show the two convolutional layers of the RBM as instantiated in PREVENT$_E$: A visible layer with as many neurons as the number of KPIs monitored on the target application, and a hidden layer with an equivalent number of neurons. The visible layer takes in input the values of the KPIs monitored at each timestamp, while the hidden layer encodes the joint distributions of KPIs, based on the training-phase sampling of the conditional probabilities of the hidden nodes given the visible nodes.

PREVENT$_E$ trains the RBM with KPI values collected at regular time intervals in production to set the reference energy value. Our experiments indicate that a training with KPI values collected over two weeks produces good results. PREVENT$_E$ reports anomalies when the energy value observed at runtime exceed the reference energy value by some thresholds. The threshold values can be computed offline by training the RBM with datasets from different applications. PREVENT$_E$ implements the RBM neural network in Matlab.

C. PREVENT *Anomaly ranker*

The PREVENT Anomaly ranker:

(i) identifies sets of anomalous KPIs with a *deep autoencoder*,
(ii) builds a graph that represents the causality dependencies between the anomalous KPIs with *Granger-causality analysis* [3], [26], [52],
(iii) exploits the causality graph to calculate the *PageRank centrality* value of each anomalous KPI in the graph [34], [44], [58],
(iv) returns the *three nodes of the application with the highest numbers of anomalous KPIs*, selecting the top anomalous KPIs of the centrality ranking, up considering at most 20% of the KPIs.

Deep Autoencoder: The deep autoencoder of PREVENT Anomaly ranker is the same deep autoencoder of PREVENT$_A$ state classifier. It identifies anomalous KPIs as KPIs with locally high reconstruction errors that we implement as KPI reconstruction errors that differ from the corresponding mean observed on the training set for more than three times the corresponding standard deviation, in our prototype implementation.

Granger Causality Analysis: At each timestamp in production, the Granger-causality analyzer builds a causality graph that represents the causal dependencies among the KPIs with anomalous values at the considered timestamp. During training, PREVENT builds a *baseline causality graph* that represents the causality relations between the KPIs, as captured under normal execution conditions: For each pair of KPIs $\langle k_a, k_b \rangle$, there is an edge from from the corresponding node k_a to node k_b in the baseline casualty graph, if the analysis of the time series of the two KPIs reveals a causal dependency from the values of k_a on the values of k_b, according to the Granger causality test [26]. The weight of the edges indicates the strength of the causal dependencies.[2] At each timestamp in production, PREVENT Anomaly ranker derives the causality graph of the anomalous KPIs by pruning the baseline causality graph, to exclude the KPIs that autoencoder does not indicate as anomalous.

PageRank Centrality: At each timestamp in production, the PREVENT Anomaly ranker exploits the causality graph of the anomalous KPIs to weight the relative relevance of the anomalous KPIs as the PageRank centrality index. PageRank scores the graph nodes (KPIs) according to both the number of incoming edges and the probability of anomalies to randomly spread through the graph (teleportation) [34].

Top Anomalous Application Nodes: PREVENT Anomaly ranker sorts the anomalous KPIs according to the decreasing values of their centrality scores; It selects the top anomalous KPIs of the ranking up to considering at most 20% of the KPIs; It tracks these anomalous KPIs to the corresponding nodes of the application, and returns the three application nodes with highest number of top-anomalous KPIs that correspond to their location.

The PREVENT Anomaly ranker improves LOUD by (i) discriminating the anomalous KPIs with a deep autoencoder instead of time-series analysis, to improve the results, (ii) exploiting the top anomalous KPIs of the ranking up to 20% of KPIs, rather than simply referring to the top 20 anomalous KPIs, to scale to large systems with thousands KPIs and (iii) considering the KPIs that are anomalous at each specific timestamp, rather than the KPIs that are detected as anomalous at least once in the scope of a time window, to identify anomalous states and anomalous nodes at each timestamp.

III. EXPERIMENTS

A. Research Questions

Our experiments address three research questions:

*RQ*1: Can PREVENT predict failures in distributed enterprise applications?

*RQ*2: Does the unsupervised PREVENT approach improve over state-of-the-art (supervised) approaches?

*RQ*3: Does PREVENT improve over LOUD, that is, the Anomaly ranker used as a standalone component?

RQ1 studies the ability of PREVENT to predict failures. We consider both PREVENT$_A$ and PREVENT$_E$, and comparatively evaluate their ability to predict failures in terms of false alarm rate, prediction earliness, and stability of true predictions.

We compute the false alarm rate in terms of *false-prediction* and *false-location* alarms. The *false-prediction alarms* are the timestamps that correspond to states wrongly identified as anomalous during normal execution. The *false-location alarms* are the timestamps that correspond to states that are identified as anomalous, but with failures wrongly located in non-anomalous nodes, during failing execution. The lower the false alarm rate of either types is, the higher the reliability of the prediction is.

We compute prediction earliness as the number of timestamps between the first true prediction and the observed system failure, that is, the time interval for activating a healing action before a system failure. We compute the stability as the true positive rate after the first prediction, that is, the ratio between predictions after the first true prediction and timestamps before the observed system failure. Intuitively, the stability indicates the continuity in correctly reporting the failing component in the presence of error states.

RQ2 investigates the advantages and limitations of training without failing executions. PREVENT trains models with data from normal (non-failing) executions only, while *supervised* techniques rely on training with both normal and failing executions. The non-necessity of training with failing execution extends the applicability of PREVENT with respect to *supervised* techniques. PREVENT discriminates failure-prone conditions as executions that significantly differ from observations at training time, while *supervised* techniques can identify only failures of types considered during training. As a result, PREVENT is failure-type agnostic, that is, it can predict failures of any types, while *supervised* techniques foster effectiveness by limiting the focus on specific types of failures. We answer *RQ2* by comparing both PREVENT$_A$ and PREVENT$_E$ with PREMISE, a state-of-the-art supervised approach that we studied in the last years and successfully applied in industrial settings [43], in terms of true positive, true negative, and false alarm rates.

[2] The Granger causality test determines the existence of a causal dependency between a pair of KPIs k_a and k_b as a statistical test of whether the time series values of k_a provide statistically significant information about the evolution of the future values of k_b [3], [26], [52]. Specifically, we test the null-hypothesis that k_a does not Granger-cause k_b by (i) building an auto-regression model for the time series values of k_b, (ii) building another regression model that considers the values of both k_a and k_b as predictors for the future values of k_b, (iii) testing if the latter model provides significantly better predictions than the former one. If so, we reject the null-hypothesis in favor of the alternative hypothesis that k_a Granger-causes k_b, and compute the strength of the causality relation as the coefficient of determination R^2. We implemented the test with the Statsmodels Python library [45], [59].

A comparison with state-of-the-art approaches requires a conceivable effort to set up the experimental context and execute the experiments. By comparing PREVENT with PREMISE we offer a fair comparative evaluation while keeping the costs within an acceptable threshold. The original PREMISE paper [43] reports a comparative evaluation with both *Operation Analytics - Predictive Insights (OA-PI)*[3], a widely adopted industrial anomaly-based tool, and G-BDA [57], a state-of-the-art signature-based approach, both outperformed by PREMISE. Thus, the comparison with PREMISE offers a good data spectrum, within acceptable set-up costs because of our full access to PREMISE.

RQ3 evaluates the contribution of the synergetic combination of the State classifier and Anomaly ranker, to skim the false alarms that derive from ranking anomalous KPIs in either non-anomalous states or non-anomalous components. We evaluate the contribution of combining State classifier and Anomaly ranker by comparing both PREVENT$_A$ and PREVENT$_E$ with LOUD, the Anomaly ranker component standalone that locates faults failures for any state, by referring to the improved approach that we integrated in the PREVENT Anomaly ranker.

B. Experimental Settings

We report the results of experimenting with PREVENT on Redis Cluster, a popular open-source enterprise application that provides in-memory data storage.[4] A Redis Cluster deploys the in-memory data storage services on a cluster of computational nodes, and balances the request workload across the nodes to improve throughput and response time. Figure 4 illustrates the structure of the Redis Cluster that we deployed for our experiments: twenty computational nodes running on separate virtual machines, combined pairwise as ten master and ten slave nodes. We integrated the Redis Cluster with a monitoring facility built on top of Elasticsearch [1], to collect the 85 KPIs indicated in Table I for each of the twenty nodes of the cluster, on a per-minute basis, resulting in 1,700 KPIs collected per minute.[5]

We executed Redis Cluster on Google cloud with a workload consisting of calls to operations for both storing and retrieving data into and from the Cluster. The synthetic workload implements a typical workload of applications in a working environment, with a high amount of calls at daytime and a decreased workload at nighttime, peaks at 7 PM and 9 AM, low traffic in weekends and high traffic in workdays: between 0 and 26 requests per second in the weekends, between 0 and 40 in workdays [42].

C. Injecting failures

We executed the Redis cluster with either no or injected failures, to collect data during normal executions (*normal execution data*) and in the presence of failures (*failing execution data*), respectively. We reinitialized Redis and restarted the virtual machines before each execution, and dropped the first 15 timestamps (15 minutes) of execution under normal conditions, to collect data on completely separated runs and experiment in a stable and unbiased context, respectively. We injected failures after 15 timestamps of execution under normal conditions. In this way, we reproduce the occurrence of failures in production at any time in normal execution conditions. We injected a failure type at a time, to reproduce failures that appear rarely in production.

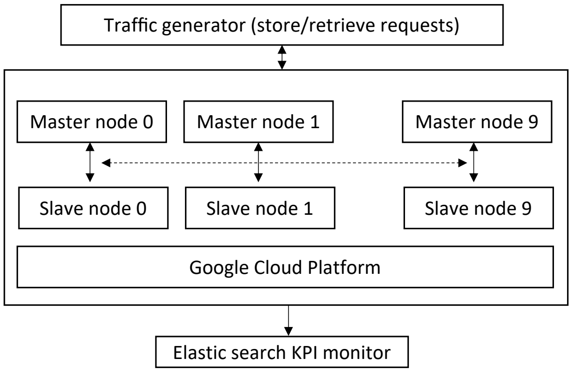

Fig. 4. Structure of the Redis Cluster

We injected failures with **Chaos Mesh**[6], a popular tool for injecting failures in the cloud. We experimented with five of the most common types of failures as defined in **Chaos Mesh**: *CPU stress*, *memory stress*, *network packet loss*, *network packet delay*, and *network packet corruption*.

We executed the Redis cluster with no injected failures for a total of three weeks. We used two weeks of continuous execution for training (80%) and validation (20%). We used the third week of execution, disjoint from the previous two weeks, to collect independent data to check for false positives. We injected each failure at a master-slave node pair of the Redis Cluster, and repeated each experiment three times, with failures injected in three different master-slave pairs to reduce statistical biases.

Experimenting with PREVENT

We trained the *Deep autoencoder* and the *RBM neural network*, and we built the granger-causality graph for the *granger-causality analyzer* with the unlabeled data collected in two weeks of normal execution. We preprocessed the collected KPIs, and discard the ones that kept a stable value all the time, which we identified as the ones for which the variance is below 10^{-5} for the entire time series, since these KPIs are not representative of the how the application reacts to input stimuli. We trained our models by considering 719 KPIs out of the 1,700 KPIs initially collected.

[3] IBM. Operation Analytics - Predictive Insights Last access: July 2019

[4] https://redis.io

[5] The monitoring infrastructure is commonly part of the deployment of distributed applications like Redis, and executing PREVENT at runtime has negligible overhead (less than 1 second) every minute.

[6] https://chaos-mesh.org/

TABLE I
KPIs collected at each node of the Redis cluster

KPI type	# of KPI	KPI names
Core	9	Id (CPU Core Identifier), idle.pct (Percentage of Idle time), iowait.pct (Percentage of CPU time spent in wait), irq.pct (Percentage of CPU time spent in handling hardware interrupts), nice.pct (Percentage of CPU time spent in low-priority processes), softirq.pct (Percentage of CPU time spent in handling software interrupts), steal.pct (Percentage of CPU time spent in involuntary wait by the virtual CPU while the hypervisor was servicing another processor), system.pct (Percentage of CPU time spent in kernel space), user.pct (Percentage of CPU time spent in user space)
CPU	19	cores (Number of CPU cores on the host), idle.pct, iowait.pct, irq.pct, nice.pct, softirq.pct, steal.pct, system.pct, total.pct (Percentage of CPU time spent in states other than Idle and IOWait), user.pct norm.idle.pct, norm.iowait.pct, norm.irq.pct, norm.nice.pct, norm.softirq.pct, norm.steal.pct, norm.system.pct, norm.total.pct , norm.user.pct (norm prefix means normalized value of the corresponding CPU metric without the norm prefix. They are normalized by considering number of cores)
File	5	count (Number of file systems), total files (Total number of files), total size.free (Total free space on disk), total size.total (Total disk space either used or free), total size.used (Total used disk space)
Load	7	load.1 (Load average for the last minute), load.5 (Load average for the last 5 minutes), load.15 (Load average for the last 15 minutes), cores (Number of CPU cores on the host), norm.1 (Load in the last minute divided by number of cores), norm.15, norm.5
Memory	18	actual.free, actual.used.bytes, actual.used.pct, free, hugepages.default size, hugepages.free, hugepages.reserved, hugepages.total, hugepages.used.pct, swap.used.bytes, swap.used.pct, total, used.bytes, used.pct
Process	8	dead (Number of dead processes), idle (Number of idle processes), running (Number of running processes), sleeping (Number of sleeping processes), stopped (Number of stopped processes), total (Total number of processes), un-known (Number of processes for which the state could not be retrieved or is unknown), zombie (Number of zombie processes)
Socket	11	all.count (All open connections), all.listening (All listening ports), tcp.all.close wait (Number of TCP connections in close_wait state), tcp.all.count (All open TCP connections), tcp.all.established (Number of established TCP connections), tcp.all.listening (All TCP listening ports), tcp.all.orphan (Number of all orphaned tcp sockets), tcp.all.time wait (Number of TCP connections in time_wait state), tcp.memory (Memory used by TCP sockets), udp.all.count (All open UDP connections), udp.memory (Memory used by UDP sockets)
Network	8	in.bytes (Number of bytes received), in.dropped (Number of incoming packets that were dropped), in.errors (Number of errors while receiving), in.packets (Number or packets received), out.bytes (Number of bytes sent), out.dropped (Number of outgoing packets that were dropped), out.errors (Number of errors while sending), out.packets (Number or packets received)
Total	85 KPI/node (1,700 KPI in the Redis Cluster with 20 nodes)	

TABLE II
Types of injected failures

We used the data collected during a third week of normal execution to evaluate the impact of *false-prediction alarms*, that is, timestamps that the prediction approaches might erroneously indicate as anomalous states during normal executions.

We used the failing execution data to evaluate (i) the ability of revealing failures, (ii) the ability to correctly locate the corresponding failing nodes, (iii) the impact of both *false prediction* and *false location alarms*, that is, the timestamps that correspond to states wrongly identified as anomalous during normal executions, and the timestamps that correspond to wrongly identified localizations during failing executions, respectively, as defined in Section II.

Experimenting with PreMiSe *and* Loud

The supervised PreMiSe approach requires training with labeled data. To train PreMiSe, we labeled normal execution data as *no-failure*, and we labelled the failing execution data as $\langle$*failure type*, *master-slave node pair*$\rangle$ to indicate the type of the injected failure and the pair of nodes at which the failures were injected, respectively. We augmented the training data set to include failing execution data for all nodes, synthesized by replicating failures on symmetric nodes. In this way, we obtained ten failing data sets for each original failing data set, one per node pair. For each failure type FT, we trained PreMiSe with the labeled data from both normal and failing executions, without the data from the failing executions corresponding to FT, and we evaluated the ability of PreMiSe to predict failures when induced by faults of type FT.

We compared Prevent with Loud, by considering the same embodiment of Loud that we exploit in Prevent, that is, the standalone Anomaly ranker component trained as described above.

Loud (the standalone Anomaly ranker) ranks anomalous KPIs for all execution states, regardless of any prediction, and thus suffers from many false-prediction alarms on normal execution data. For a fair comparison, we filtered out false-prediction alarms by signaling failure predictions only when Loud reports the same node at the top of the anomalous node ranking for N consecutive timestamps. This follows the intuition that anomalies increase in the presence of failures, and thus the failing nodes should likely persist as the top ranked nodes if the failing conditions keep occurring. We report the false-prediction alarm rate of Loud for $N = 3, 4, 5, 6$, after observing a huge amount of false alarms for values of N less than 3, and an unacceptable delay in signaling anomalous states for values of N greater than 6.

D. Results

Figures 5, 6, 7, 8, 9 visualize the results of Prevent$_A$, Prevent$_E$, and PreMiSe in the experiments with the injected failures. Each figure shows three plots that report the results of three replicas of the experiments for Prevent$_A$, Prevent$_E$, and PreMiSe, respectively. The plots of PreMiSe report the data obtained with the PreMiSe model trained without the failures of the type considered in the plot.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3327583

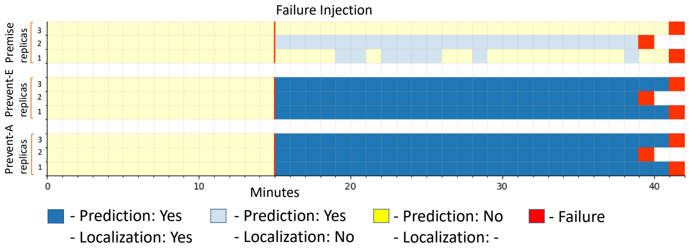

Fig. 5. Failure predictions for CPU stress

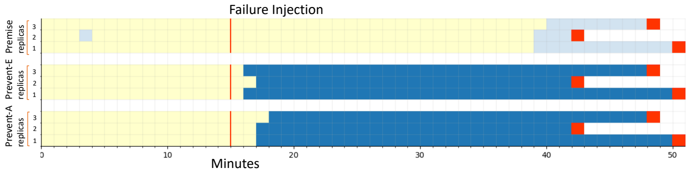

Fig. 6. Failure predictions for memory leak

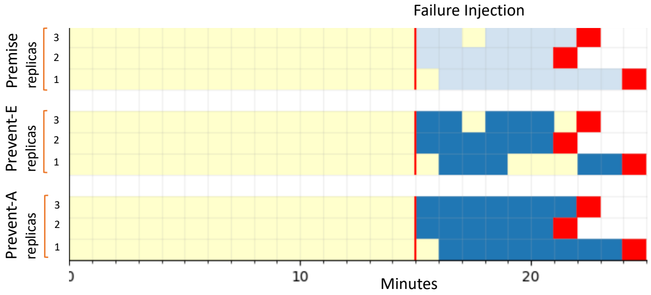

Fig. 7. Failure predictions for packet loss

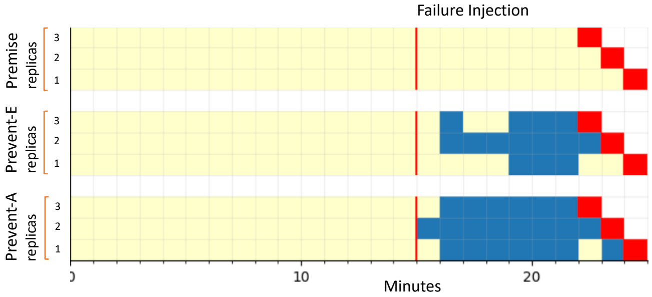

Fig. 8. Failure predictions for packet delay

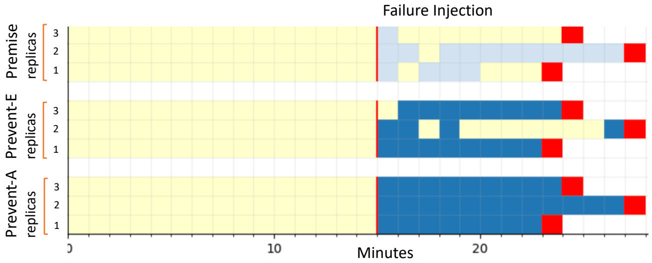

Fig. 9. Failure predictions for packet corruption

The x-axis indicates the timeline of each experiment in minutes: Each experiment includes a (not showed) initial 15 minutes lag to stabilize the system, a fault injection after additional 15 minutes of normal execution (the vertical red line in the plots), and a failing execution (the area after the red line) up to an observable system failure (a red square at the end of the plots).

We observe a Redis failures when the memory fragmentation ratio of Redis servers exceeds 1.5, following the Redis documentation that requires the memory fragmentation ratio of Redis servers to be lower than 1.5, and the servers to be restarted otherwise.[7]

Each plot shows three rows of colored squares, one row for each experiment replica. The colors visualize the strength of the predictions of the different approaches at each timestamp:

- Blue squares indicate successful failure predictions, that is, failure predictions reported along with correct localization results. In the case of Prevent this means that the node reported at the top of the localization ranking is a node in which we injected the failure.
- Grey squares indicate either false prediction alarms before the injection or false location alarms after the injection, that is, states wrongly identified as anomalous during normal executions and failures wrongly located in non-anomalous nodes, respectively.
- Yellow squares indicate the absence of prediction, which correspond to either true negatives before the start of the failure injection or false negatives after the injection.
- The red squares indicate the timestamps at which the injected failures manifest as observable failures of the cluster, and the experiment terminates.

We add the legend only to Figure 5 to reduce redundancy. In a nutshell, the more the yellow squares occur before the injection and the blue squares occur after the injection, the better the approach is. As we discuss below, the three experiment replicas yields consistent results in all cases, thus suggesting that three replicas are sufficient indeed.

RQ1: Effectiveness

RQ1 focuses on Prevent$_A$ and Prevent$_E$ (the bottom and mid plots in Figures 5, 6, 7, 8, 9). We comparatively evaluate the effectiveness of Prevent$_A$ and Prevent$_E$ to predict failures in terms of false prediction alarm rate, reaction time, prediction earliness and true positive rate that we define as follows:

- *False prediction alarm rate:* The portion of false predictions during normal execution, before the failure injection, that is, the portion of grey squares before the vertical red line;
- *Reaction time:* The time interval between the injection and the first true prediction, that is, the number of squares between the failure injection and the first blue square;

[7] https://redis.io/docs

- *Prediction earliness:* The time interval between the first true positive and the observable system failures, that is, the number of squares between the first blue square and the red square;
- *True positive rate:* The portion of true positives between the first true positive and the observable system failures, that is, the portion of blue squares between the first blue square and the red square.

The false prediction alarm rate quantifies the annoyance of operators receiving useless alerts. The reaction time quantifies the delay in alerting the operators. The prediction earliness quantifies the time offered to the operators to adopt countermeasures before the observable system failure. The true positive rate indicates the stability of the failure location alarms, once they are first raised. Predictions with high stability are most desirable since they indicate good sensitivity of the models after they start sensing the failure symptoms.

The plots in the figures show that both Prevent$_A$ (the bottom plots) and Prevent$_E$ (the mid plots) successfully predict all five types of failures, consistently across the experiment replicas. Prevent$_A$ predicts both slightly earlier and with slightly better stability than Prevent$_E$ both packet delay and packet corruption failures. The yellow squares before the injection of the failure indicate that both Prevent$_A$ and Prevent$_E$ do not suffer from false-prediction alarms during normal execution.

Tables III and IV report in detail the reaction time interval, the prediction earliness interval, and the true positive rate of predictions (TPR) per failure type and experiment replica, for Prevent$_A$ and Prevent$_E$, respectively. The data indicate that both Prevent$_A$ and Prevent$_E$ predict failures within the first minute after the failure injection in most cases (reaction interval $\leq$ 1), and in four minutes in the worst case. In details, Prevent$_E$ predicts the failure in less than a minute in 13 out of 15 cases, and Prevent$_A$ in 12 out of 15 cases. The predictions of Prevent$_A$ are more stable than the predictions of Prevent$_E$ for packet loss, packet delay and packet corruption. The TPR of Prevent$_A$ is 100% in 14 out of 15 cases, and 88% in a single replica of the packet delay experiment, while the TPR of Prevent$_E$ is 100% in 10 cases; it is between 60% and 71% in 4 cases; It is 33% in a replica of the packet corruption experiment.

TABLE III
PREDICTION EARLINESS FOR Prevent$_A$

Failure Type	Experiment replica	reaction interval	earliness interval	TPR
		(in minutes)		
CPU stress	r1	0	26	100%
	r2	0	24	100%
	r3	0	26	100%
Mem leak	r1	2	33	100%
	r2	2	25	100%
	r3	3	30	100%
Pckt loss	r1	1	8	100%
	r2	0	6	100%
	r3	0	7	100%
Pckt delay	r1	1	8	88%
	r2	0	8	100%
	r3	1	6	100%
Pckt corr	r1	0	8	100%
	r2	0	12	100%
	r3	0	9	100%

TABLE IV
PREDICTION EARLINESS FOR Prevent$_E$

Failure Type	Experiment replica	reaction interval	earliness interval	TPR
		(in minutes)		
CPU stress	r1	0	26	100%
	r2	0	24	100%
	r3	0	26	100%
Mem leak	r1	1	34	100%
	r2	2	25	100%
	r3	1	32	100%
Pckt loss	r1	1	8	63%
	r2	0	6	100%
	r3	0	7	71%
Pckt delay	r1	4	5	60%
	r2	1	7	100%
	r3	1	6	67%
Pckt corr	r1	0	8	100%
	r2	0	12	33%
	r3	1	8	100%

Overall both Prevent$_A$ and Prevent$_E$ predict failures early and with good stability in most cases. The two approaches have comparable performance, although Prevent$_A$ provides slightly more stable predictions than Prevent$_E$. These results confirm the ability of the general Prevent approach to predict failures in unsupervised fashion, and the relevance of the original Prevent$_A$ instantiation that we propose in this paper.

RQ2: Comparative Evaluation

RQ2 focuses on the advantages and limitations of the unsupervised Prevent approach with respect to state-of-the-art (supervised) approaches. Unsupervised approaches do not require training with data collected during failing executions, thus they can be used in the many industrially relevant cases where it is not possible to seed failures during operations, as required for supervised approaches.

We experimentally compare Prevent with PreMiSe, a representative supervised state-of-the-art approach as we discuss in Section II at page 3, to understand the impact of the lack of training with data from failing execution on earliness and true positive rate. The top plots in Figures 5, 6, 7, 8, 9 visualize the results of executing PreMiSe on the same data we use in the experiments with Prevent$_A$ and Prevent$_E$.

In the experiments PreMiSe rarely predicts failures and never localizes the faulty component (grey squares after failure injection, that is, false location alarms). The experiments confirm that the original combination of a state classifier with an anomaly ranker of both Prevent$_A$ and Prevent$_E$ is more effective than the PreMiSe supervised learning approach, in terms of both true and false positive rate.

TABLE V
FALSE ALARM RATE ON THE NORMAL EXECUTION DATA

Approach	False-prediction alarm rate
PREVENT$_A$	2%
PREVENT$_E$	5%
LOUD$N3$	57%
LOUD$N4$	43%
LOUD$N5$	32%
LOUD$N6$	22%

RQ3: state classifier and anomaly ranker interplay

RQ3 focuses on the effectiveness of the original combination of an anomaly ranker with a state classifier in both PREVENT$_A$ and PREVENT$_E$. We compare PREVENT$_A$ and PREVENT$_E$ with LOUD, the state-of-the-art anomaly ranker that inspired our anomaly ranker. We estimated the false-prediction alarm rate of PREVENT$_A$, PREVENT$_E$, and LOUD (with $N = 3, 4, 5, 6$) by executing each approach with the data that we collected during a week of normal execution of the Redis Cluster (the third week of our collected data, disjoint from the two weeks of data that we used for training).

Table V reports the false-prediction alarm rate, that is, the portion of false predictions during the week of normal data. The false-prediction alarm rate quantifies the annoyance of operators receiving useless alerts. PREVENT$_A$ and PREVENT$_E$ dramatically reduced the false-prediction alarm rate from unacceptable values between 22% and 57%, when using LOUD stand-alone, to 2% and 5%, respectively, by combining LOUD with failure predictors, as in PREVENT. PREVENT$_A$ outperforms all approaches.

Threats to validity

We evaluated PREVENT by executing a prototype implementation on a large dataset that we collected on a complex cluster. We carefully implemented the approach and carefully designed our experiments to avoid biases. We understand that the limited experimental framework may threaten the validity of the results, and we would like to conclude this section by discussing how we mitigated the main threats to the validity of the experiments that we identified in our work.

Threats to the external validity

Possible threats to the external validity of the experiments derive from the experimental evidence being available for a single system and five failure types, which in turn depend on the effort required to set up a proper experimental environment and collect data for validating the approach. The main threats derive from the experimental setting, and the set of failures.

Experimental setting
We experimented with a dataset collected from a Redis cluster that we implemented on the Google Cloud Platform, controlling the stability of the cluster and guaranteeing the same running conditions for all experiments, and the results may not generalise to other systems. We mitigated the threat that derive from experimenting with a single system by collecting data from a standard installation of a popular distributed application that is widely used in commercial environments, and by collecting a large set of data from several weeks of experiments. The data we used in the experiments are available in a replication package[8] for replicating the results and for comparative studies with new systems.

Set of failures
We experimented with five failure types, and the results may not generalize to other failure types. We limited the experiments to five failure types due to the large effort required to properly install and tune failure injectors and collect valid data sets for each failure. We mitigated this threat to the validity of our results by choosing types of failures which are both common and very relevant in complex cloud systems, and by injecting the failures with **Chaos Mesh**[9], a failures injector commonly used to study failure in cloud systems.

Threats to the internal validity

Possible threats to the internal validity of the experiments derive from the prototype implementation of the approaches and the traffic profile used in the experiments.

Prototype implementation
We carried on the experiments on an in-house prototype implementation of PREVENT, PREMISE and LOUD. We mitigated the threats to the validity of the experiments that may derive from a faulty implementation by carefully designing and testing our implementation, by (i) comparing the data obtained with our PREMISE and LOUD implementations with the data available in the literature, (ii) repeating the experiments three times, and (iii) making the prototype implementation available in a replication package, to allow for replicating the results.

Traffic profile
We mitigated the threats to the validity of the results that may derive from the traffic profile that we use to collect the experimental data, by experimenting with a seasonal traffic profile excerpted from industrial collaborations, and that we compared with analogous traffic profiles publicly available in the literature. We carried on the experiments by collecting data over weeks of continuous executions with profiles that correspond to traffic commonly observed in commercial settings.

IV. RELATED WORK

In this section we overview the applications of machine learning to software engineering, and discuss in detail the approaches for predicting software failures at runtime.

[8] The replication package at https://star.inf.usi.ch/#/software-data/14
[9] https://chaos-mesh.org/

A. Machine Learning for Software Engineering

Many studies address software engineering problems with supervised, unsupervised, weakly-supervised and semi-supervised machine learning strategies.

Supervised learning approaches require training samples annotated (labelled) with the prediction outcomes. The supervised training phase tunes the parameters of the model by minimizing the errors with respect to the outcomes encoded with the labels. There are many supervised learning approaches for predicting failures [40], [43], [49], [50], [57].

Unsupervised learning approaches work with unlabelled datasets, and either do not require training at all or tune models that represent the characteristics of the samples as a whole, as in our PREVENT approach. In production, unsupervised learning approaches either discriminate clusters of mutually similar samples [48] or identify anomalies as samples that notably differ from the samples used in the training phase [8], [18], [20], [47], as in PREVENT. Unsupervised approaches do not require the expensive effort of annotating large training samples, while supervised approaches precisely predict the outcomes observed in the training set.

Semi-supervised approaches are an interesting tradeoff between costs and precision, and have been recently applied to solve several software engineering problems, including predicting software fault-proneness [36], [65], [67].

Weak-supervised approaches aim to achieve the benefits of supervised learning, while reducing the costs of labelling [4], [35], [51], [53], [61], [66]. The popular *active learning* approaches require to label a relatively small set of training samples that the active learning algorithms select as the samples that contribute most to improve the quality of the predictions [17].

Semi-supervised learning approaches are weak-supervised approaches that rely on samples that the approaches automatically label from the characteristics of a small set of data that come with known labels [5], [12], [33], [41]. Bodo et al. propose a semi-supervised learning approach to analyze the performance indicators related to the software process, and show that the semi-supervised algorithm outperforms supervised learning approaches [7]. There are many semi-supervised learning approaches for software analytics, and in particular for predicting fault-prone software modules [36], [65], [67].

Classic approaches identify fault-prone modules with supervised models based on metrics that reflect the complexity of the code. These approaches require training datasets labeled either as defective or non-defective according to historical data, with a high cost for collecting the training samples [25]. Zhang et al. alleviate the labelling cost by exploiting the similarity among the code-level metrics of software modules. They propose a semi-supervised learning approach that propagate an initially small set of the labelled samples to unlabelled samples, by applying spectral clustering [67]. Tu and Menzies' FRUGAL approach extends the semi-supervised approach from fault-proneness predictions to other software analytics, like code warnings and issue close time [65].

The problem of identifying fault-prone software from both historical data on the defects identified in the software and code metrics that can be measured during software testing and maintenance radically differs from the problem of predicting failures at runtime from metrics that measure the execution of the software and not the code. To the best of our knowledge, there exist no semi-supervised or weak-supervised approaches to predict failures at runtime.

B. Failure Prediction and Diagnosis

Salfner et al.'s survey identifies *online failure prediction* as the first step to proactively manage faults in production, followed by *diagnosis*, *action scheduling* and *execution of corrective actions*, and defines *failure symptoms* as the out-of-norm behavior of some system parameters before the occurrence of a failure, due to side effects of the faults that are causing the failure [55]. In this section we focus on the failure prediction and diagnosis steps of Salfner et al.'s fault management process, the steps related to PREVENT. We refer the interested readers to Colman-Meixner et al.'s comprehensive survey for a discussion of mechanisms for scheduling and executing corrective actions to tolerate or recover from failures [14].

Detecting failure symptoms, often referred to as *anomalies*, is the most common approach to predict failures online. The problem of detecting anomalies has been studied in many application domains, to reveal intrusions in computer systems, frauds in commercial organizations, abnormal patient conditions in medical systems, damages in industrial equipments, abnormal elements in images and texts, abnormal events in sensor networks, failing conditions in complex software systems [11]. Approaches to detect anomalies heavily depend on the characteristics of the application domain. In this section, we discuss approaches to detect anomalies in complex software systems, to predict failures in production.

A distinctive characteristics of approaches for detecting anomalies is the model that the approaches use to interpret the data monitored in production, models that are either manually derived by software analysts in the form of rules that the system shall satisfy [13] or automatically inferred from the monitored data. Approaches that automatically infer models from monitored data work in either supervised or unsupervised fashion, and do or do not require labeled data to synthesize the models, respectively.

Rule-based approaches leverage analysts' knowledge about the symptoms that characterize failures, and rely on rules manually defined for each application and context. *Supervised approaches* that are also referred to as *signature-based approaches* [28] build the models by relying on previously observed anomalies, and offer limited support for detecting anomalies that were not previously observed. *Unsupervised approaches* derive models without requiring labeled data, thus better balancing accuracy and required information. Some approaches focus on failure predictions only, yet others approaches support both prediction and diagnosis, thus addressing the first two steps of Salfner et al.'s proactive fault management process.

The PREVENT approach that we propose in this paper is an unsupervised approach that both predicts and diagnoses failures, by detecting anomalies in the KPI values monitored on distributed enterprise applications. The PREVENT State classifier predicts upcoming failures as it observes system states with significant sets of anomalous KPIs. The PREVENT Anomaly ranker diagnoses the components that correspond to the largest sets of representative anomalies as the likely originators of the failures. In the remainder of this section we review the relevant automatic approaches to predict and diagnose faults in complex software systems.

Failure Prediction

The failure prediction approaches reported in the literature predict failures in either datacenter hosts that serve cloud-based applications and services or distributed applications that span multiple hosts.

Approaches that predict failures in datacenter hosts monitor metrics about resource consumption at host levels, like CPU, disk, network and memory consumption, to infer whether a host is likely to incur some failure in the near future, and proactively migrate applications that run on likely-failing hosts. Tehrani and Safi exploit a support-vector-machine model against (discretized) data on CPU usage, network bandwidth and available memory of the hosts, to identify hosts that are about to fail [19]. Both Islam and Manivannan and Gao et al. investigate the relationship between resource consumption and task/job failures, to predict failures in cloud applications [23], [31]. Both David et al. and Sun et al. exploit neural networks against data on disk and memory usage, to predict hardware failures, isolate the at-risk hardware, and backup the data [15], [62]. Approaches that predict failures in distributed applications, like PREVENT, address the complex challenge of sets of anomalies that span multiple physical and virtual hosts and devices.

Signature-based approaches leverage the information of observed anomalies with supervised learning models, to capture the behavior of the monitored system when affected by specific failures. Signature-based approaches are very popular and support a large variety of approaches. Among the most representative approaches, both *Seers* [50] and Malik et al.'s approaches [40] label runtime performance data as related to either passing or failing executions, and train classifiers to identify incoming failures by processing runtime data in production, Sauvanaud et al. classify service level agreement violations based on data collected at the architectural tier [57], The *SunCat* approach models and predicts performance problems in smartphone applications [49].

Signature-based approaches suffer from two main limitations: they both require labeled failure data for training, which may not be easy to collect, and foster predictions that overfit the failures available in the training set, without well capturing other failures and failure patterns.

Both *semi-supervised* and *unsupervised* approaches learn models from data observed during normal (non-failing) executions, and identify anomalies when the data observed in production deviate from the normal behaviors captured in the models, thus discharging the burden of labelling the training data.

Most approaches that claim a semi-supervised nature combine semi-supervised learning with signature-based models to yield their final prediction verdicts [22] [64] [63] [27]. As representative examples, Mariani et al.'s PREMISE approach uses a semi-supervised approach on monitored KPIs (IBM SCAPI [30]) to identify anomalous KPIs, and builds a signature-based model to identify failure-prone anomalies, by capturing the relation between failures observed in the past and the sets of anomalous KPIs [43]. Fulp et al.'s approach builds failure prediction models with a support vector machine (a signature-based approach) based on features distilled in semi-supervised fashion from system log files [22]. Tan et al.'s *PREPARE* approach predicts anomalies with a Bayesian networks trained in semi-supervised fashion, and feeds the anomalies to a 2-dependent Markov model (a signature based model) to predict failures [64]. Tan et al.'s ALERT approach refers to unsupervised clustering to map the failure data to distinct failure contexts, aiming to predict failures by using a distinct decision tree (a signature-based approach) for each failure context [63].

These approaches are in essence signature-based approach themselves, although they preprocess the input data in semi-supervised fashion. They may successfully increase the precision of the predictions, by training the signature-based models on pre-classified anomalies rather than plain runtime data, but share the same limitations of signature-based models of requiring labeled failure data for training, and fostering predictions that overfit the failures of the training set. Indeed, in our experiments, the PREMISE approach was ineffective against failures and failure patterns that did not correspond to the signatures considers during training.

Guan et al.'s approach [27] combines supervised and unsupervised learning in a different way. They exploit Bayesian networks to automatically label the anomalous behaviors that they feed to supervised learning models for training, thus potentially relieving the burden of labelling the data.

Only few approaches work in unsupervised fashion like PREVENT and EMBED [47] the energy-based failure prediction approach that PREVENT exploits as state classifier and that we have already described in Section II-B. Fernandes and al.'s approach [20] matches the actual traffic again models of the normal traffic built with either principal component analysis or the ant colony optimisation metaheuristic to detect anomalous traffic in computer networks. Ibidunmoye et al.'s unsupervised approach [29] estimates statistical and temporal properties of KPIs during normal executions, and detects deviations by means of adaptive control charts. Both Bontemps et al.'s [8] and Du et al.'s [18] approaches detect anomalies with long short-term memory recurrent neural networks trained on normal execution data. Ahmad et al.'s approach [2] uses hierarchical temporal memory networks to detect anomalies in streaming data time series.

Roumani and Nwankpa [54] propose a radically different approach that extrapolates the trend of occurrence of past incidents (extracted from historical data) to predict when other incidents will likely occur again in the future, without requiring data monitored in production.

Failure Diagnosis

Failure Diagnosis is the process of identifying the application components responsible for predicted failures. Most approaches proposed in the literature target performance bottlenecks in enterprise applications, and include kowledge-driven, depedency-driven, observational and signature-based approaches [28].

Knowledge-driven approaches rely on knowledge that analysts manually extract from historical records, and encode in rules that the inference engine processes to detect performance issues and identify the root-cause component. As representative example, Chung et al.'s approach [13], is designed to work at both development- and maintenance-time, to provide testing-as-a-service, and assumes that analysts frequently update the underlying rules when observing new issues.

Dependency-driven approaches analyze the communication flow among components, measure the frequency of the flows, and perform causal path analysis to detect anomalies and their causes. As a representative example, Sambasivan et al.'s Spectroscope [56] approach assists developers to diagnose the causes of performance degrades after a software change. It identifies significant variations in response time or execution frequency, by comparing request flows between two corresponding executions, observed before and after the change.

Observational approaches directly analyze the distribution of the profiled data to explain which application component correlates the most with a given system-level failure. As a representative example, Magalhaes and Silva's approach [37] identifies performance anomalies of Web applications, by analyzing the correlation between the workload and response time of the transactions: A response time variation that does not correlate with a workload variation is pinpointed as a performance anomaly. Then, they analyze the profiled data of each component with ANOVA (analysis of variance) to spot which data (and thus which components) explain the variance in the response time [38], [39].

Signature-based strategies use prior knowledge about the mapping between failures and components to diagnose failures of previously observed types [32]. As representative examples, both PREMISE and *ALERT* that we already mentioned above consider the failure location as an additional independent variable of their signature-based prediction models [43], [63].

All approaches suffer from restrictions that limit their applicability to distributed enterprise applications. Knowledge-driven and dependence-driven approaches, like Chung et al.'s approach [13] and Sambasivan et al.'s Spectroscope [56], are defined to assist developers in offline analyses, for instance, after a software change. Signature-based approaches suffer from the same limitations of signature-based failure prediction approaches: They require labeled failure data for training, and foster predictions that overfit the failures available in the training set, without well capturing other failures and failure patterns.

Magalhaes and Silva's observational approach [38] is the closest approach to PREVENT anomaly ranker, since both approaches identify failing components as the components related to data strongly connected to anomalies. Magalhaes and Silva's approach and PREVENT differ in their technical core: PREVENT uses Granger-causality centrality of the anomalous KPIs of the component, while Magalhaes and Silva rely on the ANOVA analysis. In this respect, PREVENT is more failure-type-agnostic than Magalhaes and Silva's approahc, since the Granger-causality models only causal relations among KPIs, while the ANOVA analysis refer to some specific failure metric, like poor response time in Magalhaes and Silva's approach.

Overall all approaches face challenges in balancing high detection rates with low false-alarm rates, and the effectiveness of the different strategies largely depends on the system observability, the detection mode (real-time or post-mortem), the availability of labelled data, the dynamics of the application workload, the underlying execution context, and the nature of the collected data.

V. CONCLUSIONS

Software failures in production are unavoidable [24]. Predicting failures and locating failing components online are the first steps to proactively manage faults in production [55]. In this paper we discuss advantages and limitations of the approaches reported in the literature, and propose PREVENT, a novel approach that overcomes the main limitations of current approaches.

PREVENT originally combines Deep autoencoder with Granger causality analysis and PageRank centrality, to effectively predict failures and locate failing components. By relying on a combination of unsupervised approaches, PREVENT overcomes a core limitation of supervised approaches that require seeding failures to gather labeled training data, activity hardly possible in commercial systems in production.

The experimental results that we obtained on data collected by monitoring for several weeks a popular distributed application used in commercial environments show that the original combination of unsupervised techniques in PREVENT outperforms supervised failure prediction approaches in the majority of the experiments, and largely reduces the unacceptable false positive rate of fault localizers used in isolation.

The main contribution of this paper are (i) PREVENT, an original combination of unsupervised techniques to predict failures and localize failing resources in distributed enterprise applications, (ii) a large set of data that we collected on Redis, a cluster widely used in commercial environments, data that we offer in a replication package,[10] (iii) a thorough evaluation

[10] The replication package at https://star.inf.usi.ch/#/software-data/14

that indicates the effectiveness of PREVENT with respect to the start of the art.

The results presented in this paper indicate the feasibility of unsupervised machine-learning-based approaches to predict failures and locate failing components in commercial environments, and open the horizon to study the effectiveness of unsupervised approaches for predicting failure in complex systems.

ACKNOWLEDGEMENT

This work is partially supported by the Swiss SNF project ASTERIx: Automatic System TEsting of InteRactive software applications (SNF 200021_178742), and by the Italian PRIN project SISMA (PRIN 201752ENYB).

REFERENCES

[1] Elasticsearch reference. https://www.elastic.co/elasticsearch/.

[2] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing*, 262:134–147, 2017.

[3] A. Arnold, Y. Liu, and N. Abe. Temporal causal modeling with graphical granger methods. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 66–75. ACM, 2007.

[4] S. H. Bach, B. He, A. Ratner, and C. Ré. Learning the structure of generative models without labeled data. In *International Conference on Machine Learning*, pages 273–282. PMLR, 2017.

[5] D. Berthelot, N. Carlini, I. Goodfellow, N. Papernot, A. Oliver, and C. A. Raffel. Mixmatch: A holistic approach to semi-supervised learning. *Advances in neural information processing systems*, 32, 2019.

[6] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *Proceedings of the 5th European conference on Computer systems*, pages 111–124, 2010.

[7] L. Bodo, H. Oliveira, F. A. Breve, and D. M. Eler. Semi-supervised learning applied to performance indicators in software engineering processes. In *International Conference on Software Engineering Research and Practice (SERP 2015), Las Vegas, EUA*, pages 255–261, 2015.

[8] L. Bontemps, J. McDermott, N.-A. Le-Khac, et al. Collective anomaly detection based on long short-term memory recurrent neural networks. In *International Conference on Future Data and Security engineering*, pages 141–152. Springer, 2016.

[9] M. Á. Carreira-Perpiñán and G. E. Hinton. On contrastive divergence learning. In *Proc. International Workshop on Artificial Intelligence and Statistics*. The Society for Artificial Intelligence and Statistics, 2005.

[10] D. Chandler. *Introduction to Modern Statistical Mechanics*. Oxford University Press, September 1987.

[11] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):15, 2009.

[12] O. Chapelle, B. Schölkopf, and A. Zien, editors. *Semi-Supervised Learning*. The MIT Press, 2006.

[13] I.-H. Chung, G. Cong, D. Klepacki, S. Sbaraglia, S. Seelam, and H.-F. Wen. A framework for automated performance bottleneck detection. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–7. IEEE, 2008.

[14] C. Colman-Meixner, C. Develder, M. Tornatore, and B. Mukherjee. A survey on resiliency techniques in cloud computing infrastructures and applications. *IEEE Communications Surveys & Tutorials*, 18(3):2244–2281, 2016.

[15] N. A. Davis, A. Rezgui, H. Soliman, S. Manzanares, and M. Coates. Failuresim: A system for predicting hardware failures in cloud data centers using neural networks. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 544–551. IEEE, 2017.

[16] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes. Critical phenomena in complex networks. *Reviews of Modern Physics*, 80:1275–1335, 2008.

[17] G. Druck, B. Settles, and A. McCallum. Active learning by labeling features. In *Proceedings of the 2009 conference on Empirical methods in natural language processing*, pages 81–90, 2009.

[18] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298, 2017.

[19] A. Fadaei Tehrani and F. Safi-Esfahani. A threshold sensitive failure prediction method using support vector machine. *Multiagent and Grid Systems*, 13(2):97–111, 2017.

[20] G. Fernandes Jr, L. F. Carvalho, J. J. Rodrigues, and M. L. Proença Jr. Network anomaly detection using ip flows with principal component analysis and ant colony optimization. *Journal of Network and Computer Applications*, 64:1–11, 2016.

[21] A. Fischer and C. Igel. An introduction to restricted boltzmann machines. In *Iberoamerican congress on pattern recognition*, pages 14–36. Springer, 2012.

[22] E. W. Fulp, G. A. Fink, and J. N. Haack. Predicting computer system failures using support vector machines. In *Proceedings of the USENIX conference on Analysis of system logs*, WASL'08, pages 5–5. USENIX Association, 2008.

[23] J. Gao, H. Wang, and H. Shen. Task failure prediction in cloud data centers using deep learning. *IEEE Transactions on Services Computing*, 2020.

[24] L. Gazzola, L. Mariani, F. Pastore, and M. Pezzè. An exploratory study of field failures. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '17, 2017.

[25] D. Giovanni, S. Morasca, and M. Pezzè. Deriving models of software fault-proneness. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2002)*, 2002.

[26] C. W. J. Granger. Investigating causal relations by econometric models and cross-spectral methods. *Econometrica*, 37:424–438, 1969.

[27] Q. Guan, Z. Zhang, and S. Fu. Ensemble of bayesian predictors and decision trees for proactive failure management in cloud computing systems. *Journal of Communication*, 7(1):52–61, 2012.

[28] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys*, 48(1):4:1–4:35, 2015.

[29] O. Ibidunmoye, A.-R. Rezaie, and E. Elmroth. Adaptive anomaly detection in performance metric streams. *Transactions on Network and Service Management*, 15(1):217–231, 2018.

[30] IBM Corporation. *SmartCloud Analytics - Predictive Insights 1.3 (Tutorial)*, 2014. Document Revision R2E2.

[31] T. Islam and D. Manivannan. Predicting application failure in cloud: A machine learning approach. In *2017 IEEE International Conference on Cognitive Computing (ICCC)*, pages 24–31. IEEE, 2017.

[32] H. Kang, X. Zhu, and J. L. Wong. Dapa: Diagnosing application performance anomalies for virtualized infrastructures. In *2nd {USENIX} Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*, 2012.

[33] S. Laine and T. Aila. Temporal ensembling for semi-supervised learning. *arXiv preprint arXiv:1610.02242*, 2016.

[34] A. N. Langville and C. D. Meyer. A survey of eigenvector methods for web information retrieval. *SIAM Review*, 47(1):135–161, 2005.

[35] P. Liang, M. I. Jordan, and D. Klein. Learning from measurements in exponential families. In *Proceedings of the 26th annual international conference on machine learning*, pages 641–648, 2009.

[36] H. Lu, B. Cukic, and M. Culp. Software defect prediction using semi-supervised learning with dimension reduction. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 314–317, 2012.

[37] J. P. Magalhaes and L. M. Silva. Detection of performance anomalies in web-based applications. In *2010 Ninth IEEE International Symposium on Network Computing and Applications*, pages 60–67. IEEE, 2010.

[38] J. P. Magalhaes and L. M. Silva. Adaptive profiling for root-cause analysis of performance anomalies in web-based applications. In *2011 IEEE 10th International Symposium on Network Computing and Applications*, pages 171–178. IEEE, 2011.

[39] J. P. Magalhaes and L. M. Silva. Root-cause analysis of performance anomalies in web-based applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 209–216, 2011.

[40] H. Malik, H. Hemmati, and A. E. Hassan. Automatic detection of performance deviations in the load testing of Large Scale Systems. In *Proceedings of the International Conference on Software Engineering*, ICSE '13, pages 1012–1021. IEEE Computer Society, 2013.

[41] G. S. Mann and A. McCallum. Generalized expectation criteria for semi-supervised learning with weakly labeled data. *Journal of machine learning research*, 11(2), 2010.

[42] L. Mariani, C. Monni, M. Pezzè, O. Riganelli, and R. Xin. Localizing faults in cloud systems. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '18, pages 262–273. IEEE Computer Society, 2018.

[43] L. Mariani, M. Pezzè, O. Riganelli, and R. Xin. Predicting failures in multi-tier distributed systems. *Journal of Systems and Software*, 161, 2020.

[44] T. Martin, X. Zhang, and M. E. J. Newman. Localization and centrality in networks. *Phys. Rev. E*, 90:052808, Nov 2014.

[45] W. McKinney, J. Perktold, and S. Seabold. Time series analysis in python with statsmodels. pages 96–102, 2011.

[46] C. Monni and M. Pezzè. Energy-based anomaly detection a new perspective for predicting software failures. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2019, Montreal, QC, Canada, May 29-31, 2019*, pages 69–72. IEEE / ACM, 2019.

[47] C. Monni, M. Pezzè, and G. Prisco. An RBM anomaly detector for the cloud. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 148–159. IEEE, 2019.

[48] J. Nam and S. Kim. Clami: Defect prediction on unlabeled datasets (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 452–463. IEEE, 2015.

[49] A. Nistor and L. Ravindranath. Suncat: Helping developers understand and predict performance problems in smartphone applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '14, pages 282–292. ACM, 2014.

[50] B. Ozcelik and C. Yilmaz. Seer: A Lightweight Online Failure Prediction Approach. *IEEE Transactions on Software Engineering*, 42(1):26–46, 2016.

[51] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[52] V. A. Profillidis and G. N. Botzoris. *Modeling of transport demand: Analyzing, calculating, and forecasting transport demand*. Elsevier, 2018.

[53] A. J. Ratner, C. M. De Sa, S. Wu, D. Selsam, and C. Ré. Data programming: Creating large training sets, quickly. *Advances in neural information processing systems*, 29, 2016.

[54] Y. Roumani and J. K. Nwankpa. An empirical study on predicting cloud incidents. *International journal of information management*, 47:131–139, 2019.

[55] F. Salfner, M. Lenk, and M. Malek. A survey of online failure prediction methods. *ACM Computing Surveys*, 42(3):1–42, 2010.

[56] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI*, volume 5, pages 1–1, 2011.

[57] C. Sauvanaud, K. Lazri, M. Kaâniche, and K. Kanoun. Anomaly detection and root cause localization in virtual network functions. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '16, pages 196–206. IEEE Computer Society, 2016.

[58] J. P. Scott and P. J. Carrington. *The SAGE Handbook of Social Network Analysis*. Sage Publications Ltd., 2011.

[59] S. Seabold and J. Perktold. Statsmodels: Econometric and statistical modeling with python. In *Proceedings of the 9th Python in Science Conference*, volume 57, page 61. Austin, TX, 2010.

[60] P. Stack, H. Xiong, D. Mersel, M. Makhloufi, G. Terpend, and D. Dong. Self-healing in a decentralised cloud management system. In *Proceedings of the 1st International Workshop on Next generation of Cloud Architectures, CloudNG@EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 3:1–3:6, 2017.

[61] R. Stewart and S. Ermon. Label-free supervision of neural networks with physics and domain knowledge. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[62] X. Sun, K. Chakrabarty, R. Huang, Y. Chen, B. Zhao, H. Cao, Y. Han, X. Liang, and L. Jiang. System-level hardware failure prediction using deep learning. In *2019 56th ACM/IEEE design automation conference (DAC)*, pages 1–6. IEEE, 2019.

[63] Y. Tan, X. Gu, and H. Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceedings of the Symposium on Principles of Distributed Computing*, PODC '12, pages 173–182. ACM, 2010.

[64] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 285–294. IEEE, 2012.

[65] H. Tu and T. Menzies. FRUGAL: unlocking semi-supervised learning for software analytics. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 394–406. IEEE, 2021.

[66] J. Zhang, C. Hsieh, Y. Yu, C. Zhang, and A. Ratner. A survey on programmatic weak supervision. *CoRR*, abs/2202.05433, 2022.

[67] Z.-W. Zhang, X.-Y. Jing, and F. Wu. Low-rank representation for semi-supervised software defect prediction. *IET Software*, 12(6):527–535, 2018.