Predicting Failures of Autoscaling Distributed Applications

GIOVANNI DENARO, University of Milano-Bicocca, Italy
NOURA EL MOUSSA, Università della Svizzera Italiana (USI), Switzerland and Constructor Institute Schaffhausen, Switzerland
RAHIM HEYDAROV, Università della Svizzera Italiana (USI), Switzerland
FRANCESCO LOMIO, Constructor Institute Schaffhausen, Switzerland
MAURO PEZZÈ, Università della Svizzera Italiana (USI), Switzerland and Constructor Institute Schaffhausen, Switzerland
KETAI QIU*, Università della Svizzera Italiana (USI), Switzerland

Predicting failures in production environments allows service providers to activate countermeasures that prevent harming the users of the applications. The most successful approaches predict failures from error states that the current approaches identify from anomalies in time series of fixed sets of KPI values collected at runtime. They cannot handle time series of KPI sets with size that varies over time. Thus these approaches work with applications that run on statically configured sets of components and computational nodes, and do not scale up to the many popular cloud applications that exploit autoscaling.

This paper proposes Preface, a novel approach to predict failures in cloud applications that exploit autoscaling. Preface originally augments the neural-network-based failure predictors successfully exploited to predict failures in statically configured applications, with a Rectifier layer that handles KPI sets of highly variable size as the ones collected in cloud autoscaling applications, and reduces those KPIs to a set of *rectified-KPIs* of fixed size that can be fed to the neural-network predictor. The Preface Rectifier computes the *rectified-KPIs* as descriptive statistics of the original KPIs, for each logical component of the target application. The descriptive statistics shrink the highly variable sets of KPIs collected at different timestamps to a fixed set of values compatible with the input nodes of the neural-network failure predictor. The neural network can then reveal anomalies that correspond to error states, before they propagate to failures that harm the users of the applications. The experiments on both a commercial application and a widely used academic exemplar confirm that Preface can indeed predict many harmful failures early enough to activate proper countermeasures.

CCS Concepts: • **Software and its engineering → Software reliability**; • **Computing methodologies → Neural networks**.

Additional Key Words and Phrases: Failure Prediction, Fault Localization, Kubernetes

*Ketai Qiu is the corresponding author.

Authors' addresses: Giovanni Denaro, University of Milano-Bicocca, Milan, Italy, giovanni.denaro@unimib.it; Noura El Moussa, Università della Svizzera Italiana (USI), Lugano, Switzerland and Constructor Institute Schaffhausen, Schaffhausen, Switzerland, noura.el.moussa@usi.ch; Rahim Heydarov, Università della Svizzera Italiana (USI), Lugano, Switzerland, rahim.heydarov@usi.ch; Francesco Lomio, Constructor Institute Schaffhausen, Schaffhausen, Switzerland, francesco.lomio@usi.ch; Mauro Pezzè, Università della Svizzera Italiana (USI), Lugano, Switzerland and Constructor Institute Schaffhausen, Schaffhausen, Switzerland, mauro.pezze@usi.ch; Ketai Qiu, Università della Svizzera Italiana (USI), Lugano, Switzerland, ketai.qiu@usi.ch.

ACM 2994-970X/2024/7-ART87
https://doi.org/10.1145/3660794

ACM Reference Format:
Giovanni Denaro, Noura El Moussa, Rahim Heydarov, Francesco Lomio, Mauro Pezzè, and Ketai Qiu. 2024. Predicting Failures of Autoscaling Distributed Applications. *Proc. ACM Softw. Eng.* 1, FSE, Article 87 (July 2024), 22 pages. https://doi.org/10.1145/3660794

1 INTRODUCTION

Failures in production are unavoidable due to both the combinatorial explosion of execution conditions that limits the effectiveness of testing on testbeds and the execution scenarios that emerge only in production [Gazzola et al. 2017]. Many failures stem from software bugs that corrupt the execution state, and propagate to disruptive failures, that is, system failures perceived by the users. Error states may remain hidden to the users for some time, and produce incrementally worse errors in the affected software components with minor external impacts on the overall functionality of the application, until they eventually manifest as disruptive failures. The interval between the time a bug starts producing error states and the the time the application fails offers the opportunity to predict the incoming system failures from anomalies in the error states. Predicting the failures and locating the responsible components allows for activating timely actions to avoid the impact of the failures on systems in production.

In this paper we consider microservice-based distributed applications that exploit containerized-deployment platforms [KubernetesDocs2022 2022; Merkel 2014], which are increasingly popular for providers of distributed applications and online services. These platforms optimize the deployment of applications on pools of physical machines. For instance, Kubernetes [KubernetesDocs2022 2022] runs microservices in computational nodes called *pods* that it deploys at runtime by exploiting container-based technology. Kubernetes features *autoscaling*, meaning that it can dynamically take autonomous decisions on the set of computational resources allocated to the microservices at runtime [Merkel 2014]. It monitors the resources used by the microservices (CPU, disk and memory usage) at runtime, and dynamically replicates the microservices that experience high resource consumption, by instantiating those microservices on additional pods. Thus, a microservice can run on a pool of pods, according to the required resources. Similarly, Kubernetes downscales the resources allocated to those microservices that are experiencing low stress at given time points. Kubernetes offers some self-healing features: It replaces and reschedules containers when nodes die, and kills containers that do not respond to user-defined health checks [Haja et al. 2019]. These features can mitigate some failures, but they are blind on which components are truly going to cause system failures to occur and when.

The failure prediction techniques studied so far assume a static configuration of the computational resources, and cannot cope with autoscaling distributed applications. The problem of automatically predicting failures has been widely studied in the last two decades [Chung et al. 2008; Fulp et al. 2008]. The most recent results indicate that unsupervised machine learning based approaches can precisely predict failures in complex software systems, without requiring labeled data, often difficult to gather in production [Ahmad et al. 2017; Bontemps et al. 2016; Du et al. 2017; Fernandes Jr et al. 2016; Ibidunmoye et al. 2018; Monni et al. 2019]. The machine-learning approaches proposed so far rely on neural networks that process time series of sets of key performance indicators, KPIs, that is, values of metrics collected at the computational nodes that comprise the target application. They assume a constant set of KPIs monitored at regular time intervals. Such approaches cannot cope with dynamic configurations and autoscaling, where the set of KPIs varies according to the dynamic configuration, as in the case of the many systems developed with Kubernetes.

In this paper, we propose PREFACE, *PREdicting Failures in AutosCaling distributEd Applications*, an approach to predict failures in autoscaling distributed applications. PREFACE combines descriptive statistics with a deep neural network (autoencoder) to reveal anomalous KPI values that are symptoms of incoming system failures, and ranks the microservices that are likely responsible for the failures. While state-of-the-art approaches work on time series of a constant number of KPIs, collected at regular timestamps, Preface introduces an original preprocessing step, by exploiting descriptive statistics, to deal with time series of KPI sets with size that varies over time, as in autoscaling distributed applications. Preface computes the descriptive statistics of the homologues KPIs of the pool of pods of each microservice, a step that we refer to as the Rectifier, thus reducing the varying KPI sets, to a constant set of descriptive statistics, which can be effectively handled with the autoencoder to identify anomalies.

We experimentally validated Preface on two applications, Alemira[1] which is a commercial Learning Managing System developed in Constructor Tech[2] and currently in use in several educational institutions, and TrainTicket[3] which is a microservice applications widely used in research projects. Both Alemira and TrainTicket are microservice-based applications that take advantage of the autoscaling mechanisms of Kubernetes. The results that we report in Section 4 indicate that Preface predicts failures in autoscaling distributed applications with an overall success rate between 41% and 99% in predicting error states and correctly locating the failing micro-service in the top-3 of the ranking. The results indicate a reaction time interval (that is, the interval between the first occurrence of a failure and the first correct localization of the failure) between 0 and 35 minutes, and an earliness interval (that is, the interval between the first correct localization of the failure and the disruptive failure of the system) between 13 and 102 minutes. The results that we discuss in Section 4 indicate that Preface successfully predicts also multiple failures, that is simultaneous failures of different types in different services, and locates the corresponding faults.

Thus, Preface predicts failures and locates the responsible microservices early enough to activate the self-healing mechanisms of Dockers and Kubernetes to prevent disruptive failures.

This paper contributes to the research in software engineering by both defining Preface, the first approach to predict failures in autoscaling distributed applications, and presenting the results of a set of experiments on two applications deployed on Kubernetes. Our results are available in a replication package.[4]

The paper is organized as follows. Section 2 recalls the concepts of containerized applications, and introduces the characteristics of Kubernetes that are relevant to the paper, in particular autoscaling, to make the paper self-contained. It also presents both Alemira and TrainTicket, the applications that we use for our experiments and as running examples, and discusses the main challenges that derive from autoscaling distributed applications. Section 3 presents Preface, the approach that we propose to predict failures in autoscaling applications, and shows how we can address the dynamic configurations of autoscaling distributed applications by using descriptive statistics. Section 4 presents the results of our experiments with Preface on Alemira and TrainTicket. Section 5 overviews the related work and discusses the original contributions of Preface. Section 6 summarizes the core contribution of this paper, and indicates the research directions that the results presented in this paper open for future studies.

2 WORKING EXAMPLE AND PROBLEM STATEMENT

In this section, we briefly recall containerized applications and Kubernetes, the technology we refer to in our experiments, by focusing on the core aspects relevant for this paper. We introduce Alemira and TrainTicket, the containerized applications that we use both in our experiments and as running examples, and we discuss the main challenges that derive from autoscaling.

[1] Alemira Learning Management Systems LMS available at https://constructor.tech/products/learning/lms

[2] https://constructor.tech/

[3] https://github.com/ovkulkarni/train-ticket

[4] Replication package available at https://doi.org/10.5281/zenodo.11160861

2.1 Containerized Applications with Kubernetes

Containerization optimizes the deployment of applications on pools of physical machines by dynamically allocating computational resources to microservices at runtime [Merkel 2014]. Kubernetes [KubernetesDocs2022 2022], runs microservices in computational nodes called *pods* that Kubernetes deploys with container-based technology. It monitors the resources used by the microservices (CPU, disk and memory usage) at runtime, and dynamically replicates the microservices that experience high resource consumption, by replicating the microservices on additional pods. Thus, a microservice runs on a pool of pods, according to the required resources. In this paper we use the terms *autoscaling* to refer to the *horizontal autoscaling* mechanism of Kubernetes that dynamically allocates and dismisses pods to microservices while they undergo high or low workload overtime. We use the term *autoscaler* to refer to the component of Kubernetes that implements this mechanism, consistently with the terminology of Kubernetes.

Kubernetes offers self-healing mechanisms that restart containers that fail, replace containers when nodes die, and kill containers that do not respond to health checks. In all cases, Kubernetes activates self-healing mechanisms after containers or nodes fail, and targets the actual failure, regardless of the root cause. The Preface approach that we introduce in this paper predicts failures and locates the microservices responsible for the failures before the failures occur, and thus both complements and is synergistic with the self-healing mechanisms of Kubernetes. On one side, Preface can activate the Kubernetes self-healing mechanisms before the occurrence of the failures, thus further reducing the impacts of errors on the users. On the other side, Preface predicts failures longly before disruptive failures, and can predict failures that are not in the scope of the current self-healing mechanisms of Kubernetes. Thus it can enable novel effective node- and pod-specific self-healing mechanisms, based on the information about the nods or pods ultimately responsible for the failures.

2.2 Working Example

We experimented with Alemira and TrainTicket, two applications designed with a microservice-based architecture. Table 1 lists the main microservices that comprise Alemira, and that are deployed on a pool of containers handled with the Kubernetes technology. The `rabbitmq` microservice and the `redis` container are represented after a separation line because they are the middleware of the Alemira system. The `rabbitmq` microservice manages a FIFO buffer of the messages from the clients. The `redis` container offers an in-memory cache, by interfacing with a `Redis` server for frequently retrieved data. The other microservices offer the common services of a learning management system, as most names intuitively suggest. Figure 1 reports the microservice architecture of the TrainTicket application that we use in our experiments.

Table 1. Main services of Alemira

author	mail-sender-web	ui	learner
author-react	userapi	ztool	scormhandlers
fileapi	userhandlers	gradeservice	rui
identiy	scorm	scorm-fe	identityapi
	rabbitmq	redis	

In our experiments with Alemira, we configured the Kubernetes autoscaler to allocate up to three pods for each microservice, with pods running on a pool of physical nodes ranging from two to eight nodes, under the control of the autoscaler itself. Thus, our Alemira configurations can autoscale from a minimum of 18 pods (one pod for each service of Table 1) running on two nodes, to a maximum of 54 pods (three pods per service) running on eight nodes. In our experiments

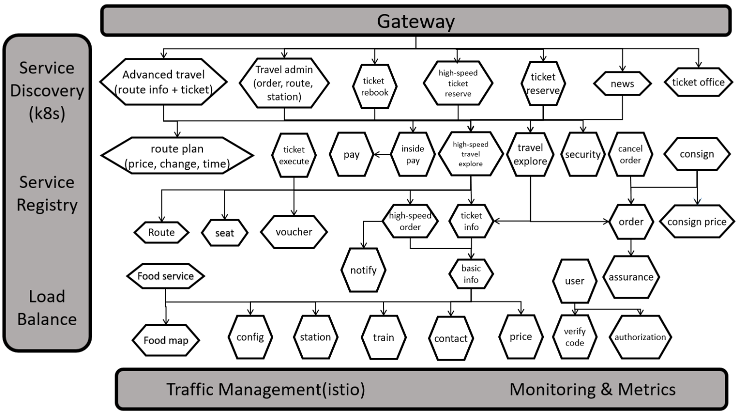


Fig. 1. The microservice architecture of TrainTicket (The figure is taken from the train-ticket github repository)

with TrainTicket, we configured the Kubernetes autoscaler to allocate up to 30 pods for each microservice, with pods running on a pool of physical nodes ranging from 1 to 10 nodes. Thus, our TrainTicket configurations can autoscale from a minimum of 42 pods running on 1 node to a maximum of 1,260 pods running on 10 nodes. Besides, there are a total of 24 database containers that do not autoscale in the case of TrainTicket, which means they always have only 1 pod. However, while Kubernetes executes the autoscaler, some pods may remain for a while in pending state, but still alive, thus at runtime we may see much more running pods occasionally. As we report in Section 4, in our experiments the distribution of the number of total running pods varied between 29 and 49 pods per timestamp (after excluding the outlier values) over two weeks of execution of Alemira. The distribution of the number of total running pods varied between 83 and 86 pods per timestamp over two weeks of execution of TrainTicket.

Preface predicts failures and locates faults from error states that reflect in anomalous metrics. We collect Key Performance Indicators (KPI series), that is, series of values of metrics from the nodes, the pods and the cloud platform, by using commonly available runtime monitoring tools, namely, `Prometheus`[5] (for Alemira only), `Google Cloud Monitoring`[6] (GCM) and `Locust`[7], for both Alemira and TrainTicket. Table 2 summarizes the types of KPI metrics that we collected from Alemira and TrainTicket, indicating how many metrics we collected for each type with the different tools, and at the level of pods, nodes or cloud platform. We report the detailed list of collected metrics in the replication package.

2.3 New Challenges from Autoscaling

The number of pods and nodes, and thus the number of KPIs that we monitor at each timestamp varies dramatically over time, due to the autoscaling mechanism. In our experiments the configurations vary dynamically from 18 pods running on two nodes, to 54 pods running on 8 nodes for Alemira, and from 42 pods on 1 node to 1,260 on 10 nodes for TrainTicket plus additional 24 pods for database containers. Consequently the KPIs to be collected at each timestamp can vary from 797 KPIs, namely, 40 KPIs × 18 pods + 29 KPIs × 2 nodes + 19 platform level KPIs (cfr. Table 2), to 2,411 KPIs, namely, 40 KPIs × 54 pods + 29 KPIs × 8 nodes + 19 platform level KPIs, across

[5] https://prometheus.io
[6] https://cloud.google.com/monitoring
[7] https://locust.io

Table 2. Summary of metrics collected by type, monitoring tool and monitoring level

	Monitoring tool					**Monitoring level**							
Metric Type	**Prometh.**	**GCM**		**Locust**		**Pod**		**Node**		**Platform**		**Total**	
	A	A	TT	A	TT	A	TT	A	TT	A	TT	A	TT
CPU	4	6	21	-	-	5	6	5	5	-	10	10	21
Memory	18	7	27	-	-	12	10	13	8	-	9	25	27
Network	10	10	44	-	-	12	8	6	7	2	29	20	44
Disk	-	-	33	-	-	-	7	-	5	-	21	-	33
Process	5	-	-	-	-	2	-	3	-	-	-	5	-
Requests Statistics	-	-	-	6	6	-	-	-	-	6	6	6	6
Response Time Statistics	-	-	-	11	11	-	-	-	-	11	11	11	11
System	8	3	46	-	-	9	15	2	9	-	22	11	46
Total	45	26	171	17	17	40	46	29	34	19	108	*88*	*188*

`Prometh.` stands for Prometheus, `A` stand for Alemira, `TT` stands for TrainTicket. We use Prometheus only for Alemira.

different timestamps for Alemira. The KPIs to be collected from TrainTicket can vary from 3,178 to 59,512. Also in this case, we may observe some outlier values due to pods in pending status. As we report in Section 4, in our experiments the distribution of the number of KPIs collected per timestamp varied between 892 and 1,628 for Alemira and from 3,444 to 3,636 for TrainTicket (after excluding the outlier values) over two weeks of execution.

The high variability of KPIs collected at each timestamp challenges neural-network-based approaches, since neural networks are trained and executed with constant sets of values per timestamp, a value for each input node of the neural network.

A neural network approach to predict failures in autoscaling Kubernetes applications needs to (i) map a number of KPIs that vary overtime to a constant number of input nodes of the neural network, (ii) map the KPIs monitored on pods that Kubernetes dynamically allocates to different microservices to external nodes of the neural network, by taking into consideration that the KPIs of a pod may refer to different microservices overtime, and (iii) cope with a number of KPIs that dramatically changes over time, due to the autoscaling mechanisms.

3 THE PREFACE SOLUTION

We define Preface to address the main challenge of KPI sets that vary over time. We designed Preface by extending the common architecture of run-time fault localizers that combines a classifier and a localizer, by adding a Rectifier. The classifier identifies anomalous states. The localizer locates the faulty elements. The Rectifier reduces the input KPI sets of variable sizes into KPI sets of constant size, thus compatible with an autoencoder neural network that provides failure predictions in unsupervised fashion.

As shown in Figure 2, Preface combines three main components, the Rectifier, the Deep autoencoder, and the Localizer. The Rectifier reduces the set of variable size of KPIs collected from the application into a set of fixed size of KPIs, to suitable feed the Deep autoencoder with an input vector of fixed size. The Deep autoencoder [Goodfellow et al. 2016] is a deep neural network that takes the KPI vector as input, and scores the level of anomaly of both that set of KPIs as a whole and each KPI separately. As explained in further detail below, those anomaly scores are technically referred to as the *reconstruction errors.* The Deep autoencoder classifies states as either correct or anomalous, by comparing the reconstruction error of the reduced KPIs with a threshold that we tune at training time on an *unlabeled* dataset comprised of a set of observations of KPI vectors that we collect during normal (non-failing) execution. If the reconstruction error is below the non-failing-execution threshold, Preface does not generate any alarm, otherwise it

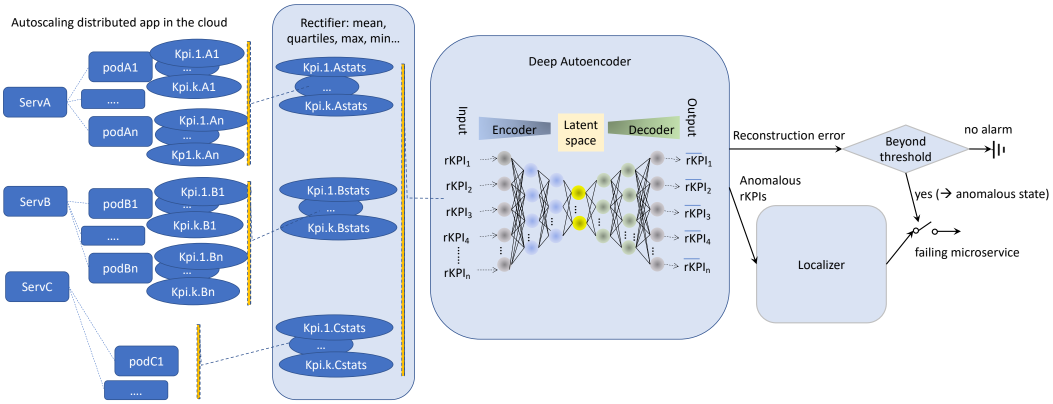


Fig. 2. The Preface architecture

activates the output from the Localizer, to signal an alarm with information on the likely source of the failure. Preface feeds the KPIs that the Deep autoencoder identified as anomalous to the Localizer, along with the local reconstruction error of each anomalous KPI. The Localizer tracks the anomalous KPIs to their corresponding microservices, and assigns the failure to the microservice that ranks as top according to the extent to which its corresponding anomalous KPIs contribute to the cumulative reconstruction error. Preface returns the failing microservice only when the overall reconstruction error signals an anomalous state.

Below we discuss in detail the main components of Preface: the Rectifier, the Deep autoencoder and the Localizer.

The core component of Preface is the Rectifier (second box in the figure) that reduces the input KPI series (first box) into a constant KPI set to be sent to the Deep autoencoder (third box), to enable a neural-network-based approach to work with a variable set of monitored KPIs.

Preface processes time series of KPIs that it collects from the microservices that comprise the target application ($ServA$, $ServB$, $ServC$, in Figure 2). The monitors collect KPIs from all Kubernetes pods that are running an instance of each given microservice at each timestamp. For instance in the figure, $podA1, \ldots, podAn$ indicate n pods that are running instances of $ServA$. For each pod and each considered KPI, Preface produces the time series of the KPI sampled at constant intervals, every minute in our experiments. In the figure, $kpi.(1..k).A1$ indicate k distinct KPIs collected from $podA1$ that is running an instance of $ServA$. As discussed in the former section, the set of collected KPIs varies over time in both size and service-type, depending on the number of the pods that are running at the time of the sample, and which microservice is being instantiated in each pod.

The Rectifier reduces the variable sets of KPIs collected at each timestamp to a set of KPI values that has constant size at every timestamp, by computing summary statistics (mean, quartiles, max, min, and count) of the homologous KPIs from all pods that correspond to the same microservice. In the examples in Figure 2, $kpi.k.Astats$ are the statistics ($stats \in \{mean, quartiles, max, min, count\}$) of $kpi.k$ for microservice $ServA$, based on the values of $kpi.k$ that are monitored across all pods that are running $ServA$ at a timestamp. For instance, with reference to the actual KPIs that we collected for Alemira, $used.bytes.Amean$ indicates the mean of the allocated memory (the KPI $used.bytes$) and can be computed with respect to all active pods of type $ServA$ ($podA*$) at the considered timestamp: if microservice $ServA$ is currently replicated in four pods, then the current sample from the monitoring facility includes the values $used.bytes.A1$, $used.bytes.A2$, $used.bytes.A3$ and $used.bytes.A4$, i.e., the allocated memory for each of the four pods. Then, at that timestamp, the Rectifier yields the KPI value $used.bytes.Amean = (used.bytes.A1 + used.bytes.A2 + used.bytes.A3 + used.bytes.A4)/4$.

Since we monitor a fixed number of KPIs from each microservice, each KPI (for instance, $used.bytes.A$) is summarized with a fixed set of statistics (mean, quartiles, max, min, and count), and the application consists of a fixed number of microservices (even if there can be a variable number of pod service replicas), the Rectifier produces the same number of KPI statistics, namely, $|KPIs| \times |stats| \times |Microservices|$ at all timestamps. In the visualization of Figure 2 the Rectifier yields as output a fixed set of *rectified KPIs* (rKPIs).

The Deep autoencoder identifies both anomalous states and anomalous rKPIs. As illustrated in Figure 2, a Deep autoencoder [Goodfellow et al. 2016] is a deep neural network with two contiguous sequences of layers (*Encoder* and *Decoder*) that mirror each other structure, share a common layer (*Latent space*), and transform an input set of n rKPIs ($rKPI_i$) into an an output set of n rKPIs ($\overline{rKPI_i}$). Namely, the *Encoder* and *Decoder* of the Preface Deep autoencoder are composed of three layers of sizes n, $\frac{n}{2}$, $\frac{n}{4}$, respectively, and a *Latent space* of $\frac{n}{8}$ nodes, being n is the number of rKPIs.

We train the Deep autoencoder on a set of observations of the rKPIs collected during normal (non-failing) execution. For example, in the experiments reported in Section 4, we collected observations of the rKPIs every minute for two weeks. During training, the *Encoder* learns how to encode the input data in incrementally condensed form, while the *Decoder* learns how to regenerate the input from the condensed information. The Deep autoencoder computes the *reconstruction error* as the difference between the input and output values. 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 (the Beyond threshold check in Figure 2). Namely, Preface identifies an anomalous state when the mean-squared reconstruction error of all rKPI is above the threshold of $m_e + 3s_e$, where m_e and s_e are the mean and the standard deviation, respectively, of the mean-squared reconstruction errors observed during training. Similarly, for the anomalous states, Preface identifies the value of a specific $rKPI_a$ as anomalous when the reconstruction error of $rKPI_a$ is above the threshold of $m_e^a + 3s_e^a$, where m_e^a and s_e^a are the mean and the standard deviation of reconstruction errors observed during training for $rKPI_a$.

The Localizer scores the anomalous rKPIs by the contribution of their local reconstruction errors with respect to the overall reconstruction error, and their help in discriminating the anomalous states. For each anomalous rectified KPI, say $rKPI_a$, it first computes the Z-score,

$$zscore = (reconstruction\_error(rKPI_a) - training\_threshold(rKPI_a))/training\_std\_dev(rKPI_a)$$

of the reconstruction error of $rKPI_a$, to quantify the extent of the anomaly with respect to the typical variance (known from training) of that rKPI. Then, it further adjusts the score by computing to what extent the Z-score of $rKPI_a$ differs with respect to the Z-score values observed for $rKPI_a$ in the latest 20 consecutive timestamps for which Preface did not predict failures,

$$score_{adjusted} = (zscore - mean(zscore\_latest\_20\_ok))/std\_dev(zscore\_latest\_20\_ok).$$

This step acknowledges that some KPIs can sometimes experience some drifts of their values in production with respect to the observations made in the training phase, and produce anomalies that do not discriminate the anomalous states from the non-anomalous ones.

The Localizer aggregates the scores of the anomalous rKPIs that belong to the same microservices, and signals the top ranked microservices as failing microservices at each timestamp for which Preface predicted an anomalous state. As the rKPIs are aggregated statistics of the metrics from different instances of the same microservice, the Preface's Localizer is able to localize the failures at the microservice level, however it does not indicate the specific failing instance.

4 EXPERIMENTAL EVALUATION

In this section we discuss the experimental setting and the results of our experimental evaluation. In Section 4.1 we introduce the main research questions that drove our evaluation, present the relevant details of the Alemira and TrainTicket case studies, describe the workload generator, and outline the experimental plan. In Sections 4.2, 4.3, 4.4 and 4.5 we discuss our results and findings for our research questions. In Section 4.6 we discuss the main threats to the validity of the results presented in this paper.

4.1 Experimental Setting

4.1.1 Research Questions. Our experiments address four main research questions:

- **RQ1:** What is the variation in the size of the KPI set that a failure prediction model shall handle, for distributed applications with dynamically-scaling deployment?
- **RQ2:** Can Preface predict failures in distributed applications that take full advantage of Kubernetes autoscaling?
- **RQ3:** Can Preface predict multiple failures that occur simultaneously in different components of distributed applications?
- **RQ4:** Can Preface be effective also by using a threshold-based failure detector rather than the Deep autoencoder neural network?

RQ1 studies the extent of the problem that we address with the Preface approach, that is, the need of a failure-prediction approach that handles KPI sets of dynamically varying size. We investigate RQ1 by monitoring several days of execution of both Alemira and TrainTicket, and quantifying the variation in the number of running pods and KPIs collected over time.

RQ2 and RQ3 study the effectiveness of Preface, and assess the contribution of our approach in solving the failure-prediction problem for distributed applications with dynamically-scaling deployment. We answer RQ2 and RQ3 by training Preface on the data monitored against two weeks of execution of both Alemira and TrainTicket, further executing two applications with different types of failures injected in either different microservices (RQ2) or simultaneously in multiple microservices (RQ3), and quantifying the precision of Preface in predicting those failures.

RQ4 investigates the specific contribution of the Deep autoencoder neural network in Preface, by means of an ablation study. We reshape the design of Preface by replacing the Deep autoencoder with a no-neural-network anomaly detector, which directly exploits the dynamics of the rKPIs, rather than referring to the reconstruction errors. Then we compare the effectiveness of Preface with respect to this baseline. The no-neural-network anomaly detector calculates a threshold for each rKPI computed by the Preface Rectifier, by referring to the training data. It detects the anomalous rKPIs at runtime, as the rKPIs with value greater than the corresponding threshold. The threshold for each rKPI is computed as $m + 3s$, where m is the mean and s is the standard deviation of the normalized value of the rKPI in the training data, which is consistent with the approach of the Preface Deep autoencoder that detects anomalies at runtime when the reconstruction error exceeds the threshold of $m_e + 3s_e$, with respect to the mean and the standard deviation of the reconstruction errors observed during training. The no-neural-network anomaly detector identifies anomalous states when the average value of the normalized rKPIs exceeds the threshold of the average value observed in the training data, and then feeds the Localizer by scoring the anomalous rKPIs by their corresponding extra-threshold deltas.

4.1.2 Metrics. We evaluate the ability of PREFACE to predict failures in terms of localization rate and timing, with a set of experiments, in which we execute both ALEMIRA and TRAINTICKET with failures injected at selected microservices. We compute the localization rate with respect to the error interval, that is, the interval between the timestamp when we inject the failure and the early timestamp of a disruptive failure. We identify a disruptive failure in terms of response time and HTTP failure rate, following the common perception of Kubernetes users. We compute the response time as the aggregated 95^{th} percentile of the response time of all the requests in a minute, and the HTTP failure rate as the average rate of HTTP failures of all requests within a minute. We performed the Mann-Whitney U rank test [Mann and Whitney 1947] on the faulty dataset and normal dataset to confidently reject the null hypothesis that there is no difference between two datasets in terms request response time and HTTP failures. We also compute the corresponding Vargha-Delaney A^12 effect size between the two datasets to investigate the significance of differences. We identify a disruptive service at the timestamps at which the difference of the request response time and HTTP failure rate between the normal dataset and the faulty dataset is statistically significant.

We collected the following metrics:

- *Strong localization rate:* We compute the *strong localization rate* as the portion of timestamps at which PREFACE correctly predicts a failure and correctly locates the microservice where we injected the failure, by referring to the error interval.
- *Weak localization rate:* We compute the *weak localization rate* as the portion of timestamps at which PREFACE correctly predicts a failure, and locates either the failing microservice as second or third in the ranking obtained from PREFACE's LOCALIZER or a proxy-microservice of the failing microservice in the top position of the ranking. A proxy-microservice of the failing microservice is a microservice that is directly related to the failing microservice in the architecture, and thus offers a good approximation of the localization of the failing service, by referring to the error interval.
- *Overall localization rate:* We define the *overall localization rate* as the union of strong and weak localizations, which quantifies the true positives yielded by PREFACE.
- *False alarm rate:* We compute the *false alarm rate* in terms of *false-prediction* and *false-localization alarms*. The *false-prediction alarms* are the timestamps that PREFACE wrongly identifies as anomalous states during normal execution, that is, at timestamps that occur before the failure injection. The *false-localization alarms* are the timestamps that correspond to states that PREFACE correctly identified as anomalous albeit without locating the failure, that is, it locates the failure in a microservice that is neither the failing one nor an immediate proxy of the failing one, in the error interval.
- *Reaction interval:* We compute the *reaction interval* as the number of timestamps between the fault injection and the first correct localization (either strong or weak).
- *Earliness interval:* We compute the *earliness interval* as the number of timestamps between the first correct localization and the disruptive failure. The earliness intervals quantify the usefulness of PREFACE, in terms of the time interval for activating a healing action before a system failure.

4.1.3 Experimental setup. We experimented PREFACE for predicting failures on instances of both ALEMIRA and TRAINTICKET deployed on a Google Cloud Kubernetes cluster equipped with `Google Cloud Monitoring`. We collected the metrics from the load generators that we used for both ALEMIRA and TRAINTICKET from `Locust`. We collected additional metrics for ALEMIRA only with `Prometheus` that we deployed on the cluster where ALEMIRA runs, by automatically scraping for metrics from the monitored application.

Table 3. Alemira and TrainTicket configuration details

	CPU and mem. per pod	CPU and mem. threshold	pods per microservice	node autoscaling	node config.	initialization
Alemira	1 CPU, 500 MB	80%	1–3	2–8	6 CPU 10 GB mem. 100 GB disk	1 pod per service, 3 nodes
TrainTicket	1 CPU, 100 MB–2 GB	80%	1–30	1–10	4 CPU 10 GB mem. 100 GB disk	1 pod per service, 3 nodes

We collected metrics at a time granularity of a minute, in the form of time series, by aggregating the values of the metrics within a specific minute via a Python script. We collected the KPIs that we discuss in Section 2 and report in details in the replication package.

We used the Google Kubernetes Engine (GKE) service as a platform to create and manage a Docker-based environment for building a Kubernetes cluster to deploy both the Alemira and the TrainTicket microservices. GKE is an industry-scale service that implements complete Kubernetes API with horizontal pod and node autoscaling. Table 3 summarizes the parameters of the Alemira and TrainTicket setups.

4.1.4 Workload Generator. We implemented two workload generators, one for Alemira and one for TrainTicket, respectively. Alemira is a relatively new product, and we still have limited data about the usage. We designed the workload generator based on data of the usage of `iCorsi`[8], a learning management system widely used in many academic institutions in Switzerland. We refer to the data collected from 2017 to 2022 of the usage of over thousand courses offered to tens of thousands users in `iCorsi`. We designed the workload generator for TrainTicket, by referring to the publicly available statistics of the Swiss railway system. We generated workload according to profiles illustrated in Figure 3. We implemented the workload generators in Python with `Locust`[9]. The generators periodically send API calls to some microservices and wait for the responses. Both Alemira and TrainTicket offer hundreds of APIs.

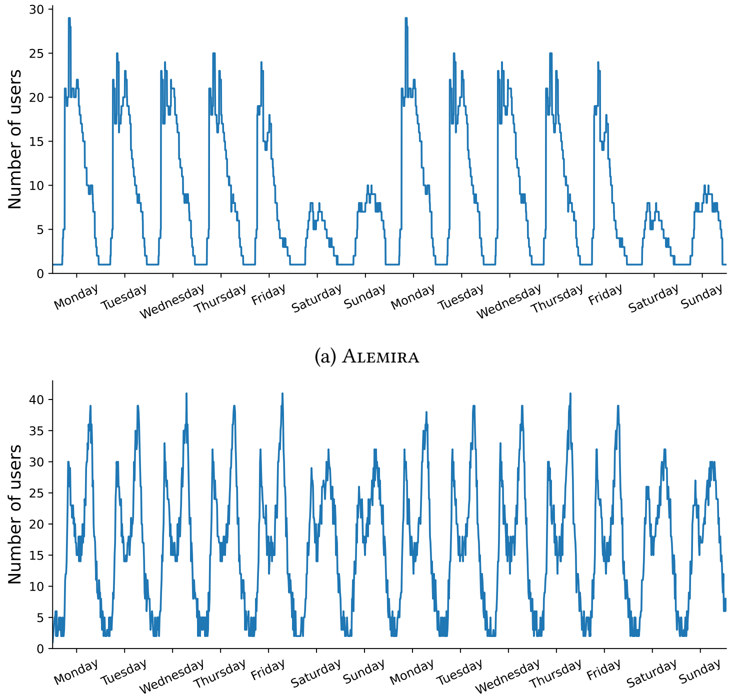
We selected the 64 Alemira APIs that correspond to the mostly used *services* according to the data from `iCorsi`, and that are offered in both `iCorsi` and Alemira, to generate realistic workload conditions. There are three main types of users of Alemira, students, instructors and administrators. We characterize each type of user with a set of tasks, each composed of a sequence of *api* calls. For example, students can authenticate, visit a course, upload an assignment, etc. A visit task calls the *api* `get_user_permissions`, `get_user_roles`, and `get_user_objective_workflow_aggregates`.

We selected the 117 TrainTicket APIs that correspond to the mostly used *services* according to the data from the statistics of the Swiss railway system, to generate realistic workload conditions. There are two main types of users of TrainTicket, passengers and sellers, and a total of nine profiles that correspond to different kinds of passengers and sellers. We characterize each type of user with a set of tasks, each composed of a sequence of API calls.

The traffic generators associate a weight to each task, according to the data from `iCorsi` and the Swiss railway system, respectively. At each step, the traffic generators select a task according to the weights. We repeatedly generated workloads to tune the parameters of the workload generator, and generate workloads consistent with the data of `iCorsi3` and the Swiss railway system, respectively.

4.1.5 Experiment plan. We executed the case studies, Alemira and TrainTicket, both in normal conditions and with injected failures. We executed both case studies for two weeks in normal

[8] https://www.icorsi.ch
[9] https://locust.io/



(a) Alemira

(b) TrainTicket

The X-axis indicates the weekdays for two consecutive weeks. The Y-axis indicates the workload as number of users.

Fig. 3. Workload profiles for the Alemira and TrainTicket traffic generators

conditions, with the workloads that we present in Section 4.1.4 to collect data for training the Deep autoencoder. Based on the KPI data collected every minute for two weeks, the training of the Deep autoencoder required between 10 and 20 minutes in both case studies. We remark that the training of the Deep autoencoder has no impact at runtime, as it happens offline. In practical settings, the collection of the training data and the training of the Deep autoencoder can be repeated periodically to account for possible drifts in the behavior of the applications.

We executed both case studies with injected failures up to a disruption point, to study the effectiveness of Preface in predicting failures. We injected CPU stress, Memory stress and Network delay failures that correspond to the most common disruptive failures in Kubernetes. We executed the case studies with both single failures injected in a microservice, and two failures of different types injected in two distinct microservices. We injected the failures with `Chaos Mesh` [10] with the default settings, a popular tool for injecting failures, commonly used in research and commercial studies, to reduces biases. We injected failures in five different services, `userapi` and `redis` of Alemira, `train`, `station` and `basic` of TrainTicket. We executed the microservice up to stable execution conditions that we kept for 30 minutes before injecting the failure. We then continue executing the microservice up to a disruptive state that we observed in all experiments at different time intervals.

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

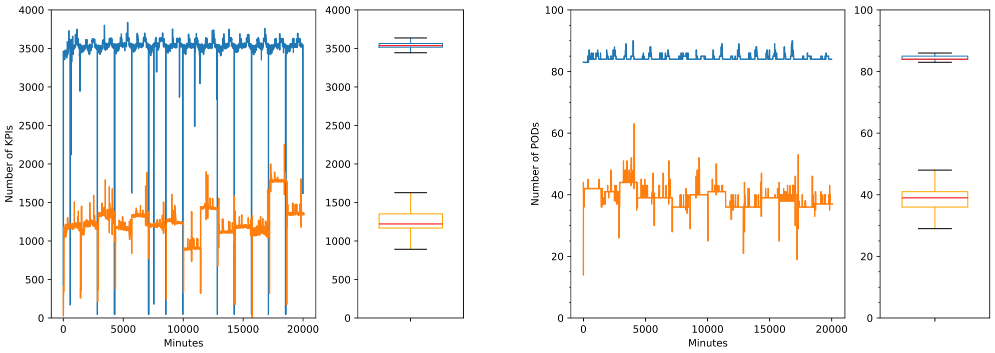
The experiments with injected failures lasted between 44 and 139 minutes, including 30 minutes before the first injection. We then measured the false alarm rate of Preface in the intervals before the failure injections in the 15 experiments, for a total of 450 minutes. Executing Preface at runtime requires just few seconds for each prediction, which has negligible impact as we monitor the applications with sampling periods of one minute.

4.2 RQ1: Impact of Dynamic-Scaling Deployment

Our study grounds on the observation that state-of-the-art failure-prediction techniques cannot analyze distributed applications with dynamic-scaling deployment, like microservice-based applications on Kubernetes, because they cannot deal with sets of KPIs with sizes that dynamically vary over time. Our first research question investigates the relevance of this phenomenon to confirm the need of new approaches, e.g., an approach like Preface. We studied the extent of this variation in our case-study applications Alemira and TrainTicket. We analyzed both the number of KPIs that we monitored and the number of pods that were instantiated during two weeks of execution of both Alemira and TrainTicket with the traffic profiles described in Section 4.1.

Figures 4a and 4b plot the number of KPIs collected and pods instantiated during the two weeks of execution and the box plots of the corresponding summary statistics (mean, quartiles, minimum, and maximum), respectively, after excluding the outliers. As already commented in Section 2, the outliers arise because there can occasionally be pods with pending-status and KPIs collected from those pending-status pods.

In the figures we observe that the number of collected KPIs ranges between 892 and 1,628 for Alemira and between 3,444 and 3,636 for TrainTicket (boxplots at the right side of Figure 4a with outliers excluded) with continuous variations from low to high values and back (plot at the left side of the figure) across the two weeks of our experiment, due to the variation in the number of instantiated pods that ranges between 29 and 49 for Alemira and between 83 and 86 for TrainTicket (boxplots at the right side of Figure 4b with outliers excluded) with continuous variations (plot at the left side of the figure). The data confirm the need for a novel type of prediction model to cope with the high variability in the number of KPIs of this class of applications.



(a) Number of KPIs collected

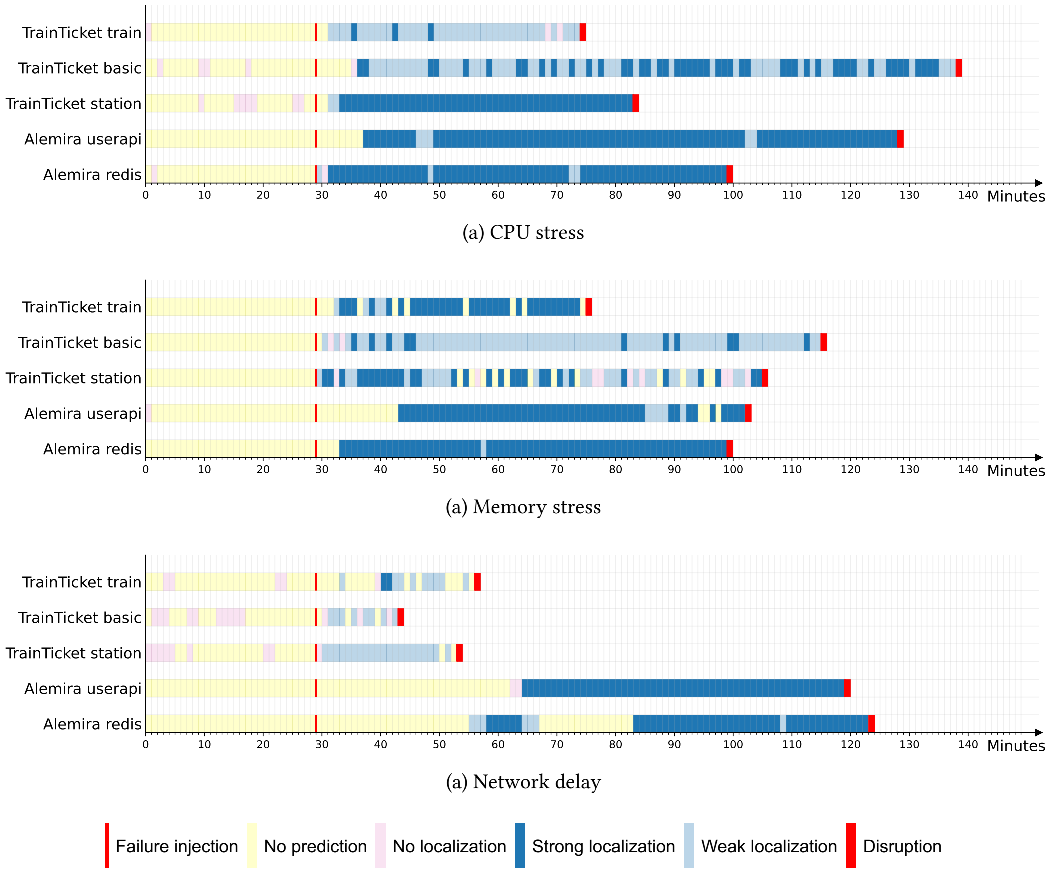
(b) Number of pods instantiated

Fig. 4. Dynamic-scaling of resources while executing Alemira (orange) and TrainTicket (blue) for two weeks

4.3 RQ2: Effectiveness of Preface

Figure 5 shows the results of the experiments with Preface on both Alemira and TrainTicket. For each failure type, the figure shows a plot that reports five data lines. The bottom two lines correspond to experiments with failures injected in two different microservices of Alemira, `userapi` and `redis`. The top three lines correspond to experiments with failures injected in three different microservices of TrainTicket, `train`, `basic`, and `station`.

The x-axes indicate the timeline of each experiment in minutes. In each experiment, time 0 is an instant the system reached a stable state in normal execution conditions. We start the fault injector after 30 minutes of normal execution in a stable state (the vertical red line in the plots, labeled as *failure injection*). Then we monitor the failing execution (the time points after the red line) up to an observable disruptive failure (the red square at the end of the plots, labeled as *disruption*).



(a) CPU stress

(a) Memory stress

(a) Network delay

Fig. 5. Failure predictions with Preface on both Alemira and TrainTicket

Each plot includes five data rows represented as sequences of colored squares, one row for each experiment replica. The colors visualize the results of Preface at each timestamp:

- blue squares indicate strong localization, that is, failure predictions reported along with correct localization results;
- light blue squares indicate weak localization, that is, failure predictions in which the failing microservice is reported as second or third of the ranking or a proxy-microservice of the failing microservice in the top position of the ranking from the Localizer;

- pink squares indicate either false-prediction alarms before the injection or false-localization 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.

In a nutshell, the more the yellow squares occur before the injection and the more blue and light blue squares occur after the injection, the better the precision of the predictions of Preface is.

Table 4 summarizes the data from the experiments for each failure type and each microservice in terms of *reaction interval*, *earliness interval*, *strong localization rate*, *weak localization rate*, and *overall localization rate*.

The plots and the table show that Preface successfully predicts all three types of failures, consistently across the experiments with different failing microservices. It suffered only a negligible number of false-prediction alarms during normal execution. We observe an average false-prediction alarm rate of 8% (percentage of pink over yellow squares before the failure injection across all the diagrams in Figure 5). The data indicate that Preface predicts the failures for the first time with a reaction time interval that ranges from 0 to 35 minutes after the injection, and at least 13 minutes before the disruptive failure (disruption in the plots). The percentages in column *earliness interval* indicate the percentage positions of the first localization in the intervals between the failure seeding and the disruptive failure. We see that Preface predicts the failure in the first ten percent of the time slots after the injection (values above 90% in most cases), and always in the first half of the interval (values above 50%). Overall the successful predictions range between 41% and 99% of the time points in the interval between the beginning of failure injection and the system failure.

In our experiments, Preface has higher precision in the experiments with CPU stress and Memory stress failures, in which the overall localization rate ranges between 72% and 99%, than in the experiments with Network delay failures, in which the overall localization rate is between 41% and 88%. The manual inspection of the data indicates that the network failures have less disruptive impacts on the functionality of the microservices, and can generally be tolerated better by the application, thus resulting in lower impact on the monitored KPIs than in the case of CPU stress and Memory stress failures. Nonetheless, Preface is able to predict those failures at least 13 minutes before the disruption point.

4.4 RQ3: Multiple failures

Figure 6 shows the results of the experiments with two failures of different types injected in distinct microservices in TrainTicket. The blue squares indicate that the localizer ranks first either of the two failing services (strong localization). The light blue squares indicate that the localizer ranks either second or third at least one of the two failing services, with a non failing service ranked first (weak localization). Yellow squares indicate absence of prediction (true positives before and false negatives after the injection). Pink squares indicate that no failing services is ranked in the top three positions in the presence of a localization (false positives).

Table 5 reports the reaction time, earliness and localization rate for the experiments described in Figure 6. The results are consistent with the results of failures injected in single microservices (Table 4): reaction time within 0-2 minutes (0-35 for single failing service), earliness within 12 and 64 minutes (13-102 for single failing service) with all predictions in the first ten percent of the interval (values greater than 90% in column *earliness interval*), overall localization rate within 40% and 96% (41%-99% for single failing service). The improved reaction time may reflect the stronger impact of multiple vs. single failing microservices. Overall, the experiments with failures of all

Table 4. Reaction time, earliness interval and localization rates

Failure Type	Target Application	Failing Service	reaction interval (min)	earliness interval (min)	Strong Localization Rate	Weak Localization Rate	Overall Localization Rate
CPU stress	ALEMIRA	userapi	8	91 (92%)	87%	5%	92%
		redis	0	70 (100%)	93%	6%	99%
	TRAINTICKET	train	2	42 (96%)	7%	84%	91%
		station	2	51 (96%)	93%	3%	96%
		basic	6	102 (94%)	44%	49%	94%
Memory stress	ALEMIRA	userapi	14	59 (80%)	70%	7%	77%
		redis	4	66 (94%)	93%	1%	94%
	TRAINTICKET	train	3	42 (93%)	69%	9%	78%
		station	0	84 (100%)	33%	39%	72%
		basic	1	59 (98%)	13%	84%	97%
Network delay	ALEMIRA	userapi	35	55 (61%)	61%	0%	61%
		redis	26	68 (72%)	48%	7%	55%
	TRAINTICKET	train	4	23 (85%)	8%	33%	41%
		station	0	24 (100%)	0%	88%	88%
		basic	1	13 (93%)	0%	57%	57%

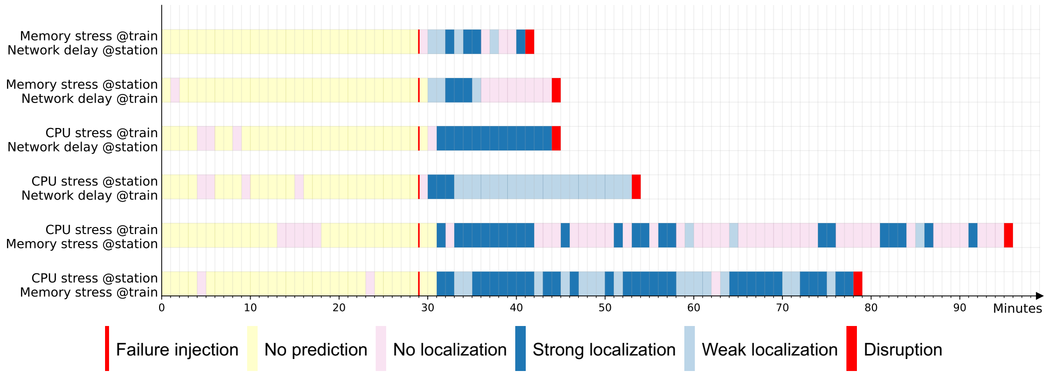


Fig. 6. Failure predictions with PREFACE on simultaneous failure injection in TRAINTICKET

types injected pairwise in two different services indicate a consistent behavior of PREFACE in the multiple failures of different types occurring in different services.

4.5 RQ4: Contribution of the DEEP AUTOENCODER

Table 6 compares the results of PREFACE (columns *P* in the table) with the baseline (columns *B*) of the no-neural-network variant of PREFACE, by reporting the results of experiments with ALEMIRA and TRAINTICKET. The columns *False Alarm Rate* indicate the portion of false-prediction alarms during normal execution, that is, before the failure injection. The columns *Missed Alarm Rate*, *False Localization Rate* and *Overall Localization Rate* indicate the precision during failing executions, that is, after failure injection, when an approach may incur a *missed alarm* (if it raises no alarm), a *false localization* (if it localizes the raised alarm in a non-faulty service) or a *valid localization* (if it provides a correct –weak or strong– localization). The values in column *Overall Localization Rate/P* are the values of *Overall Localization Rate* that we reported for PREFACE in the last column of Table 4.

Table 5. Reaction time, earliness interval and localization rates for injections of multiple failures

Injected Failures		reaction interval (min)	earliness interval (min)	Strong Localization Rate	Weak Localization Rate	Overall Localization Rate
CPU stress Memory stress	@station @train	2	47 (96%)	61%	33%	94%
CPU stress Memory stress	@train @station	2	64 (97%)	35%	5%	40%
CPU stress Network delay	@station @train	0	24 (100%)	13%	83%	96%
CPU stress Network delay	@train @station	1	14 (93%)	87%	0%	87%
Memory stress Network delay	@station @train	1	14 (93%)	20%	20%	40%
Memory stress Network delay	@train @station	0	12 (92%)	33%	33%	67%

Table 6. Comparison between Preface (P) and the no-neural-network Preface baseline (B)

Experiment			False Alarm Rate		Missed Alarm Rate		False Localization Rate		Overall Localization Rate	
			B	P	B	P	B	P	B	P
CPU stress	Alemira	userapi	0%	0%	25%	8%	0%	0%	75%	92%
		redis	3%	3%	1%	0%	0%	1%	99%	99%
	TrainTicket	train	3%	3%	67%	4%	0%	4%	33%	91%
		station	67%	23%	0%	4%	4%	0%	96%	96%
		basic	50%	13%	3%	6%	3%	1%	94%	94%
Memory stress	Alemira	userapi	0%	3%	100%	23%	0%	0%	0%	77%
		redis	0%	0%	23%	6%	0%	0%	77%	94%
	TrainTicket	train	100%	0%	0%	22%	98%	0%	2%	78%
		station	100%	0%	0%	16%	100%	12%	0%	72%
		basic	100%	0%	0%	1%	9%	2%	91%	97%
Network delay	Alemira	userapi	0%	0%	100%	37%	0%	2%	0%	61%
		redis	0%	0%	94%	45%	3%	0%	3%	55%
	TrainTicket	train	3%	13%	78%	56%	15%	4%	7%	41%
		station	100%	27%	8%	8%	25%	4%	67%	88%
		basic	100%	33%	0%	21%	93%	21%	7%	57%

The *False Alarm Rate* of Preface is lower or equal than the baseline in 13 out of 15 experiments. It's slightly higher than the baseline only in two experiments, Memory-stress-Alemira-userapi and Network-delay-TrainTicket-train. The *False Alarm Rate* of the baseline is the worst possible (100%) in 5 out of 15 experiments, while the *False Alarm Rate* of Preface is consistently and acceptably low. The *Overall Localization Rate* of Preface is consistently better than the baseline, up to large extent, in most cases, and equal to only in three CPU stress experiments. *The False Localization Rate* of Preface is always lower than or comparable with the baseline. These results confirm the relevant role of the Deep autoencoder in the Preface approach.

4.6 Threats to Validity

We experimented for several months with a prototype that we carefully designed, by relaying on the mature libraries available for some of the core functionalities, namely the autoencoder and the monitoring infrastructure. We acknowledge that the limited experimental framework may threaten the validity of the results, despite our careful execution of the experiments. In this section, we identify the main threats to the validity of the results, and we discuss how we mitigated them.

Threats to the external validity. The main threat to the external validity of the results derives from experimenting with a single system and with a limited amount of failures injected in few services. We experimented with datasets collected from both a research-benchamark application, TrainTicket, and a commercial system, Alemira, currently in use in many institutions. We collected the data from our own installation of TrainTicket on Google Cloud Kubernetes and from an Alemira instance originally deployed by the developers for load and stress testing. We defined the workload profile for Alemira from the data monitored on a learning management system widely used in many academic institutions in Switzerland, and for TrainTicket by referring to the publicly available statistics of the Swiss railway system. We injected failures of the three types that occur more often in cloud systems, according to our industrial partners. We used a state-of-the-art injector commonly used in both academic and industrial experiments. The results are not statistically generalizable, but they offer a clear vision of the potentiality of Preface.

Threats to the internal validity. The main threat to the internal validity of the results derive from the reliability of the data collected during the experiments, the tools used to inject the failures, and the prototype implementation of Preface that we developed to analyze the data. We monitored the KPIs, collected the data and injected failures with standard tools commonly used in academic and commercial studies. We strongly relied on mature libraries to implement the core functionalities of the prototype implementation of Preface. We made the prototype implementation and the data available in a replication package for replicating the experiments.[11]

5 RELATED WORK

In this section we overview the work on failure prediction and fault localization for distributed applications, and discuss the main research efforts related to detect anomalies with dynamically varying sets of KPIs.

Current approaches for predicting failures target applications deployed on statically-sized sets of resources, with combinations of rule-based, supervised, semi-supervised or unsupervised strategies. *Rule-based approaches* depend on detection rules manually defined by experts, by leveraging on their experience about the application-specific symptoms that characterize the failures [Chung et al. 2008]. The major limitation of these approaches is the low degree of automation. *Supervised approaches* characterize most popular techniques that automatically infer failure-detection models from data monitored at runtime. These approaches train machine-learning models based on data from both executions in which specific failures manifested, and executions without failures. Previous studies reported on using supervised classifiers for many types of failures, including performance failures and service level agreement violations [Bodik et al. 2010; Davis et al. 2017; Fadaei Tehrani and Safi-Esfahani 2017; Gao et al. 2020; Islam and Manivannan 2017; Malik et al. 2013; Nistor and Ravindranath 2014; Ozcelik and Yilmaz 2016; Sauvanaud et al. 2016; Sun et al. 2019]. Supervised approaches generally suffer from requiring large amounts of labeled failure data for training, which are hardly available in many practical application scenarios. *Semi-supervised approaches* exploit supervised models on top of synthetic data inferred with either semi-supervised, weakly supervised or unsupervised learning, to balance accuracy and required information [Fulp et al. 2008; Guan et al. 2012; Mariani et al. 2020; Tan et al. 2010, 2012]. Purely *unsupervised approaches* work without requiring labeled data, e.g., by deriving models that capture the characteristics of the

[11] Replication package available at https://doi.org/10.5281/zenodo.11160861

executions without failures, and can thus discriminate the anomalous executions that significantly differ from those ones [Ahmad et al. 2017; Bontemps et al. 2016; Du et al. 2017; Fernandes Jr et al. 2016; Ibidunmoye et al. 2018; Monni et al. 2019]. Failure prediction approaches for distributed applications often aim also to localize the components responsible for the failures [Chung et al. 2008; Ibidunmoye et al. 2015; Magalhaes and Silva 2011; Mariani et al. 2018, 2020; Sambasivan et al. 2011; Tan et al. 2010]. Similarly to Preface, the recent `Prevent` approach of Denaro et al. [Denaro et al. 2022] uses an autoencoder to predict failures and locate faults. Differently from Preface, `Prevent` combines an autoencoder with Granger causality and page ranking to predict failures and locate faults in application executed on a statically configured set of computational resources. Thus, `Prevent` cannot deal with autoscaling distributed applications.

The approach discussed in this paper is itself unsupervised, and includes both a failure prediction and a failure localization step, but is unprecedented in addressing failure prediction for distributed applications with dynamically-scaling deployments.

To the best of our knowledge, there has been no work so far on predicting failures in distributed applications with dynamically-scaling deployments, as it is the case of Kubernetes. We exploit descriptive statistics [Nicholas 1990] to design the Rectifier for producing sets of the same size at all timestamps from sets of dynamically changing size. The most relevant alternative approaches to reconciling a varying set of KPIs with prediction models with fixed number of inputs (neural networks) can be devised by either preprocessing the KPIs with padding [Jadhav et al. 2019] or exploiting recurrent neural networks [Lipton et al. 2015; Mesnil et al. 2013].

The *padding* approaches use neural network models with as many inputs as the maximum number of KPIs, that is, the set of KPIs monitored on the largest configuration of computational units. The simple *zero-padding* approach sets the missing inputs to zeroes, at timestamps when the number of KPIs is less than the number of inputs [Albawi et al. 2017]. However, for our problem instance, zero-padding results in unrealistic values for the missing KPIs during downscaled executions. In our project, we considered the idea of combining padding with imputation methods [Jadhav et al. 2019], assigning the missing KPIs, e.g., due to unallocated computational units for a component, with synthetic values computed as functions of the KPIs from the currently allocated computational units. Other work combines padding with imputation methods [Jadhav et al. 2019] to generate the missing inputs with synthetic values computed as functions of some available KPIs. For example, the missing KPIs due to unallocated computational units for a given component could replicate the KPIs of computational units currently allocated to that component, thus using plausible values for those computation units.

To the best of our knowledge, no research effort so far has investigated the application of padding and imputation methods for instantiating failure prediction models in dynamically-scaling deployments of distributed applications. The effectiveness of padding depends on the percentage of missing values and the regularity of the distribution of values. In highly dynamic Kubernetes application, the largest configuration may be dramatically larger than many common configurations, thus resulting in large percentages of missing values at most timestamps. Our preliminary experiments with padding augmented with suitable imputation methods confirmed our concerns. This is why eventually we privileged a descriptive statistics approach over padding.

Recurrent neural networks have an internal memory that encodes the meaning of a sequence of input vectors [Lipton et al. 2015; Mesnil et al. 2013]. We could merge the KIPIs of each computational unit into separate input vectors of fixed size, and handle the KPIs of multiple computational units allocated to a component as multiple vectors that the network handles in sequence. The computational cost of *recurrent neural networks* increases with the number of units, and may deserve specific care in the presence of large numbers of units, as in wide cloud applications. This is why we privileged a descriptive statistics approach over recurrent neural networks that we plan to study next, taking advantage of the experience and excellent results of Preface.

6 CONCLUSIONS

In this paper we present the first approach that predicts failures and locates the failing component in autoscaling distributed applications, and discuss the results that we obtained on both a large commercial application, Alemira, and a widely used academic exemplar, TrainTicket.

Software fails in production due to both the impossibility of exhaustively sampling the ultra large execution space of large software systems and execution conditions that emerge only in production and are not available in testbeds. [Gazzola et al. 2017]. Very often, failures originate from bugs in the code that produce error states that propagate through the system execution up to disruptive effects that are perceived by the final users [Avizienis et al. 2004]. Early detecting error states before disruptive effects enables both manual and automatic healing, as the self-healing mechanisms that modern platforms currently activate only after disruptive failures [Haja et al. 2019].

Many approaches predict failures by analyzing time series of values of Key Performance Indicators, KPIs, that reflect error states. The most recent approaches use unsupervised machine learning, and specifically neural networks, to reveal anomalies in time series of KPI, and predict failures [Ahmad et al. 2017; Bontemps et al. 2016; Du et al. 2017; Fernandes Jr et al. 2016; Ibidunmoye et al. 2018; Monni et al. 2019]. The approaches based on neural networks feed the fixed set of input nodes of a neural network with a fixed number of KPI values monitored at each time sample. They can effectively predict failures in applications that run with statically configured computational resources, however, they cannot deal with the autoscaling mechanisms of containerized-deployment platforms [KubernetesDocs2022 2022; Merkel 2014].

This paper introduced Preface, an approach that predicts failures and locates the failing component by analyzing time series of KPIs with a number of KPIs that vary over time. Preface originally combines a deep neural network (autoencoder) with descriptive statistics and ranking. The descriptive statistics reduce KPI sets of variable size to the fixed number of input neurons of the autoencoder. The ranking aggregates the scores of the anomalies that belong to the same microservices, and signals the top ranked microservices as failing microservices to predict failures.

In the paper, we showed that the variation in number of both executing units and collected metrics can be very large and pervasively frequent for a Kubernetes-based application like our case studies. We discussed how Preface overcomes the limitations of current approaches to deal with the autoscaling mechanisms of containerized-deployment platforms, and we presented the results of a set of experiments that we executed on a commercial platform with workload obtained from profiles collected on a five years period, and common failures that we injected with a popular failure injector. The results presented in this paper confirm the suitability of the approach, and open new research directions toward effective approaches to improve the reliability of autoscaling distributed applications that run on containerized platforms.

ACKNOWLEDGMENTS

This work is partially supported by the Big Sistah national research project (MUR, PRIN 2022, 2022EYX28N) and by the Swiss SNF project ASTERIx: Automatic System TEsting of InteRactive software applications (200021_178742).

Ketai Qiu played a key role in experimenting both with Alemira, as reported in Ketai's master thesis titled *Experimental Evaluation of Failure Predictors for Industrial-scale Applications*, and with TrainTicket to significantly enrich the final results reported in the paper.

REFERENCES

Subutai Ahmad, Alexander Lavin, Scott Purdy, and Zuha Agha. 2017. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing* 262 (2017), 134–147.

Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. 2017. Understanding of a convolutional neural network. In *2017 international conference on engineering and technology (ICET)*. Ieee, 1–6.

A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 11–33. https://doi.org/10.1109/TDSC.2004.2

Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B Woodard, and Hans Andersen. 2010. Fingerprinting the datacenter: automated classification of performance crises. In *Proceedings of the 5th European conference on Computer systems*. 111–124.

Loïc Bontemps, James McDermott, Nhien-An Le-Khac, et al. 2016. Collective anomaly detection based on long short-term memory recurrent neural networks. In *International Conference on Future Data and Security engineering*. Springer, 141–152.

I-Hsin Chung, Guojing Cong, David Klepacki, Simone Sbaraglia, Seetharami Seelam, and Hui-Fang Wen. 2008. A framework for automated performance bottleneck detection. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–7.

Nickolas Allen Davis, Abdelmounaam Rezgui, Hamdy Soliman, Skyler Manzanares, and Milagre Coates. 2017. Failuresim: A system for predicting hardware failures in cloud data centers using neural networks. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. IEEE, 544–551.

Giovanni Denaro, Rahim Heydarov, Ali Mohebbi, and Mauro Pezzè. 2022. PREVENT: An Unsupervised Approach to Predict Software Failures in Production. *CoRR* abs/2208.11939 (2022). https://doi.org/10.48550/arXiv.2208.11939 arXiv:2208.11939

Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1285–1298.

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

Gilberto Fernandes Jr, Luiz F Carvalho, Joel JPC Rodrigues, and Mario Lemes Proença Jr. 2016. Network anomaly detection using IP flows with principal component analysis and ant colony optimization. *Journal of Network and Computer Applications* 64 (2016), 1–11.

Errin W. Fulp, Glenn A Fink, and Jereme N Haack. 2008. Predicting Computer System Failures Using Support Vector Machines.. In *Proceedings of the USENIX conference on Analysis of system logs (WASL'08)*. USENIX Association, 5–5.

Jiechao Gao, Haoyu Wang, and Haiying Shen. 2020. Task failure prediction in cloud data centers using deep learning. *IEEE Transactions on Services Computing* (2020).

Luca Gazzola, Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. 2017. An Exploratory Study of Field Failures. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '17)*.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

Qiang Guan, Ziming Zhang, and Song Fu. 2012. Ensemble of Bayesian predictors and decision trees for proactive failure management in cloud computing systems. *Journal of Communication* 7, 1 (2012), 52–61.

David Haja, Mark Szalay, Balazs Sonkoly, Gergely Pongracz, and Laszlo Toka. 2019. Sharpening Kubernetes for the Edge. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos* (Beijing, China) *(SIGCOMM Posters and Demos '19)*. Association for Computing Machinery, New York, NY, USA, 136–137. https://doi.org/10.1145/3342280.3342335

Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. 2015. Performance Anomaly Detection and Bottleneck Identification. *Comput. Surveys* 48, 1 (2015), 4:1–4:35.

Olumuyiwa. Ibidunmoye, Ali-Reza Rezaie, and Erik Elmroth. 2018. Adaptive Anomaly Detection in Performance Metric Streams. *Transactions on Network and Service Management* 15, 1 (2018), 217–231.

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

Anil Jadhav, Dhanya Pramod, and Krishnan Ramanathan. 2019. Comparison of performance of data imputation methods for numeric dataset. *Applied Artificial Intelligence* 33, 10 (2019), 913–933.

KubernetesDocs2022 2022. Kubernetes Documentation. https://kubernetes.io/docs. [Online; accessed Aug-2022].

Zachary C Lipton, John Berkowitz, and Charles Elkan. 2015. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019* (2015).

Joao Paulo Magalhaes and Luis Moura Silva. 2011. Root-cause analysis of performance anomalies in web-based applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. 209–216.

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

Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

Leonardo Mariani, Cristina Monni, Mauro Pezzè, Oliviero Riganelli, and Rui Xin. 2018. Localizing Faults in Cloud Systems. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '18)*. IEEE Computer Society, 262–273.

Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Rui Xin. 2020. Predicting failures in multi-tier distributed systems. *Journal of Systems and Software* 161 (2020). https://doi.org/10.1016/j.jss.2019.110464

Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

Grégoire Mesnil, Xiaodong He, Li Deng, and Yoshua Bengio. 2013. Investigation of recurrent-neural-network architectures and learning methods for spoken language understanding.. In *Interspeech*. 3771–3775.

Cristina Monni, Mauro Pezzè, and Gaetano Prisco. 2019. 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*. IEEE, 148–159. https://doi.org/10.1109/ICST.2019.00024

Jackie Nicholas. 1990. *Introduction to descriptive statistics*. Mathematics Learning Centre, University of Sydney.

Adrian Nistor and Lenin Ravindranath. 2014. SunCat: Helping Developers Understand and Predict Performance Problems in Smartphone Applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM, 282–292.

Burcu Ozcelik and Cemal Yilmaz. 2016. Seer: A Lightweight Online Failure Prediction Approach. *IEEE Transactions on Software Engineering* 42, 1 (2016), 26–46.

Raja R Sambasivan, Alice X Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R Ganger. 2011. Diagnosing Performance Changes by Comparing Request Flows.. In *NSDI*, Vol. 5. 1–1.

C. Sauvanaud, K. Lazri, M. Kaâniche, and K. Kanoun. 2016. Anomaly Detection and Root Cause Localization in Virtual Network Functions. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '16)*. IEEE Computer Society, 196–206.

Xiaoyi Sun, Krishnendu Chakrabarty, Ruirui Huang, Yiquan Chen, Bing Zhao, Hai Cao, Yinhe Han, Xiaoyao Liang, and Li Jiang. 2019. System-level hardware failure prediction using deep learning. In *2019 56th ACM/IEEE design automation conference (DAC)*. IEEE, 1–6.

Yongmin Tan, Xiaohui Gu, and Haixun Wang. 2010. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC '12)*. ACM, 173–182.

Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. 2012. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*. IEEE, 285–294.

Received 2023-09-28; accepted 2024-04-16