Predicting cloud-native application failures based on monitoring data of cloud infrastructure

Laszlo Toka
MTA-BME Network Softwarization Research Group
Budapest University of Technology and Economics
Budapest, Hungary
toka.laszlo@vik.bme.hu

Gergely Dobreff, David Haja, Mark Szalay
Faculty of Electrical Engineering and Informatics
Budapest University of Technology and Economics
Budapest, Hungary
{dobreff.gergely, david.haja, szalaymarkpeter}@edu.bme.hu

***Abstract*—The quality of service provided by cloud-deployed online applications is often affected by faults in the underlying cloud platform and infrastructure. In order to discover the cause and effect at application failures, a cloud monitoring system must be in place. The sheer amount of the produced monitoring data calls for smart and automatic handling in order to find the patterns that can be used for fault management. In this paper we present an open source, cloud-native, lightweight cloud monitoring system, and a data analytics pipeline that efficiently processes the gathered data and is able to discover useful inference between infrastructure-, and application-level metrics. We apply time series clustering steps within the pipeline to compress the collected data for fast and lightweight data mining. We show the capabilities of our proposed system in a reactive and a proactive use case. The results prove that the proposed system brings precious insights for root-cause analysis and proactive fault management frameworks of cloud applications.**

***Index Terms*—cloud-native, monitoring, fault management, artificial intelligence, proactive, Kubernetes**

I. Introduction

The third millennium has brought the rise of cloud computing; online applications are now mostly provisioned in data centers, either in private, or in public clouds. Along the transition to cloud, the programming paradigm has also changed significantly, monolithic, then 2-, or 3-tier application designs have been swept away by the microservice approach of continuously developing and rolling out sub-components of complex applications [1]. The current way of working, i.e., design, implementation and operation, of cloud-based applications enables the application providers to deliver services without huge capital expenditures into infrastructure, offers adaptive and fast scaling to the current customer base, and high reliability and availability, all thanks to the cloud context.

Deploying applications into the cloud has also the benefit of low operating expenses. The reason behind the effective operation is the economy of scale of compute infrastructure in data centers, and the shared resources among many tenants. The downside of the co-location and the relatively complex infrastructure though is that fault management has become even more challenging than before. Top cloud providers offer comprehensive monitoring services, e.g., Amazon CloudWatch [2], but in smaller public clouds and particularly in private clouds, the fault management of the deployed applications require the monitoring of the infrastructure too.

Fortunately many open source tools are available for this purpose. The selection of the appropriate monitoring apparatus is governed by the fault management goal: the set of metrics and the frequency of data collection determine the capability of failure detection, and potentially its prediction. Obviously the larger the monitoring target set and the higher the collection frequency are, the more resources the monitoring system takes away from the deployed applications. When there is no pinpointed goal of the monitoring data collection, it is best not to limit the scope of the monitored metrics, and also to allow the highest possible frequency. Then artificial intelligence (AI)-based methods can be used to tackle the data set, and to derive observations useful to uncover the causes of, and/or to completely avoid application failures.

Our contribution is two-fold. First, we present a monitoring system, built on open source components, to be applied with Kubernetes [3], the most widely used cloud management platform today. We show that the system is cloud native, and it is lightweight enough for wide adoption. Second, we propose an AI pipeline that takes the large amount of collected monitoring data, and after important steps of compacting the data set for fast and lightweight operation, highlights inference between infrastructure and application metrics. This knowledge then can be used in root-cause analysis of application failures, or in the proactive management of quality of service.

This paper is organized as follows. In Sec. II we highlight applied methods and related work in cloud monitoring, then in Sec. III we present our monitoring solution. We describe the AI process that we implement on the monitoring data for fault management in Sec. IV. We show illustrative use cases in Sec. V to reflect the power of the proposed framework. We conclude the paper in Sec. VI.

II. Related work

In an ordinary cloud monitoring and fault management system, infrastructure and application descriptive metrics are continuously recorded, and the values are treated as variables in correlation and/or regression analysis. [4] presents

The authors are members of HSNLab (hsnlab.hu) and their research project was funded by Ericsson.

978-3-903176-32-4 © 2021 IFIP

a lightweight anomaly detection tool for data centers, which uses the rigorous correlation of system metrics without the need for training or complex infrastructure set up. They hypothesize that the infrastructure nodes' metrics and the virtual machines where the applications run are strongly correlated in an anomaly-free system. Their tool detects a node-level anomaly if the correlation drops below a threshold value. We argue that such a correlation analysis does not consider the time dimension of the measured metrics, and finding nonlinear relationships can be challenging. In [5], [6] the authors use Canonical Correlation Analysis (CCA) [7]–[9] to model the correlation between workloads and the metrics of application performance to detect anomalies by discovering the abrupt change of the correlation coefficients. The advantage CCA brings compared to a regular correlation analysis is the impact weight calculation, however, CCA still does not consider the measured metrics' time dimension. In the anomaly detection system designed by Farshchi et al. [10], [11], the authors set the goal to find correlation between an application's logs and its activity's effect on cloud resources by an approach that adopts a regression-based analysis technique. In our work we tackle the opposing direction of cause and effect: we strive to predict application failures that are due to some anomalies in the cloud infrastructure, possibly caused by other applications running in the platform. The study in [12] states that correlation-based methods are although widely used for detecting root causes of failures in large-scale networks, they often contain spurious correlations, which buries the truly important information. To tackle this issue, the authors propose a method combining a graph-based causal inference algorithm and a pruning method based on domain knowledge.

Time Lagged Cross Correlation (TLCC) [13] is an applicable method to identify the leader-follower relationship, calculate the correlation value of two time-series, and find the delay between them. Another similar technique, Dynamic Time Warping (DTW) [14] calculates the distance between two time-series, which may have different lengths. By DTW one might expose the delays between two metrics' time-series, measure how close the two metrics are to each other, and cluster them according to their behavioral pattern. Consequently, DTW is a suitable method to detect how an infrastructure failure affects the application metrics in time.

Many research works have focused on monitoring enterprise infrastructure systems and diagnosing their effects on the applications' health. It is often assumed that the KPI (Key Performance Indicator) data of the examined systems are available. Munawar and Ward [15] argue that in practice, collecting such data is typically too expensive since it comes at the cost of reduced system efficiency. To avoid this complexity, they propose adaptive monitoring, i.e., their solution identifies relationships in the monitored data, characterizes the normal operation, and in the case of failure, it identifies the areas that need to be monitored in more detail. However, not just monitored KPIs, but network log messages can be valuable and useful information to detect unexpected or anomalous behavior. Kobayashi et al. [16] argue that it is challenging to extract pinpoint system failures and identify their causes from the extremely huge amount of system log data of a distributed network. Thus, they propose a methodology relying on causal inference which processes the network syslog data and significantly reduces the number of pseudo correlated events compared with the traditional methods.

Authors of [17] introduce the concept of functional connectivity as an alternative approach to monitoring events resulting from the interaction of various services operating on complex, heterogeneous and evolving networks. They present a novel inference approach whereby two nodes are defined as forming a functional edge if they emit substantially more coincident or short-lagged events than would be expected if they were statistically independent. The output of their method is an undirected weighted graph modeling the strength of the statistical dependencies between the nodes.

In [18] the authors claim that analysis of the runtime dependencies among infrastructure, platform, and application layers of the cloud software stack is essential to performing cloud resource management, detecting anomalous behavior of cloud applications, and meeting customer Service Level Agreements (SLAs). However, due to the non-linear nature of microservices interactions, aberrant data measurements and lack of domain knowledge, finding dependencies among the data is challenging. They propose the use of Long-Short Term Memory (LSTM) recurrent neural networks, which excel in capturing temporal relationships in multi-variate time series data and show resiliency to noisy pattern representations. They consider three use-cases of their proposed technique: finding the strongest performance predictors, discovering lagged/temporal dependencies, and improving the accuracy of forecasting for a given metric.

In this paper, we strive to predict application failures - from non-stop application and infrastructure KPIs monitoring - that are possibly caused by cloud infrastructure anomalies. Our proposed method considers the time dimension of the measure metrics (applies DTW), detects linear and nonlinear relationships, operates with low system footprint and uses classical statistical methods to discover inference between infrastructure-, and application-level metrics.

III. Cloud-native monitoring system

In this section we present our lightweight, cloud-native monitoring system for Kubernetes. Our proposed solution is able to collect and store sufficient amount of data from a Kubernetes cluster used for advanced analysis in order to detect different anomalies that can occur in cloud applications. This monitoring system is capable of collecting tens of thousands of infrastructure and application KPIs without consuming large amounts of extra resources.

A. Goals and requirements

The goal of our solution is to correlate infrastructure monitoring data with application failures. To this end, we set up an advanced analytics pipeline that collects data, automatically analyses inference between infrastructure and application

monitoring data and proposes actions based on the analysis. To be successful in performing such an advanced analysis, our system must monitor node metrics, like generic Linux properties, e.g., CPU, memory, I/O, logs, daemons, etc., and Kubernetes-related status properties and events.

Here are the requirements that our proposed solution must fulfill. First, we wanted to keep the monitoring system as lightweight as possible. It is important for such monitoring solutions to prevent consuming the available cloud resources from other applications. It is also expected from the monitoring ecosystem to be cloud-native. Regarding this latter, the components should have the following properties: i) easy deployment, e.g., with Helm charts [19]; ii) scalability: monitor the dynamically changing number of application instances and nodes with adaptive apparatus; iii) resiliency: both for the collected data and the monitoring system components. Fully compatible with Kubernetes, we selected CNCF [20] compliant tools.

B. System components

Our monitoring solution is a 4-tier system with the following class of components: exporters, collectors, database, visualization. Key parts are shown in Fig. 1.

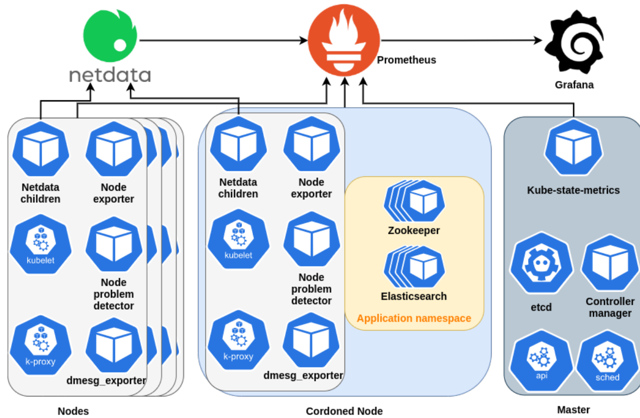

Fig. 1. Architecture of monitoring system

Based on our findings of the capabilities of open source monitoring tools, we learned that at least two collectors are needed to cover all required metrics. Therefore, respecting CNCF-compliance, we chose Netdata [21] and Prometheus [22] as our data collectors. Netdata is an open source monitoring tool that is able to dynamically monitor thousands of metrics and turn monitoring data into live visualizations. In our setup we used Netdata with its streaming feature: Netdata instantiated children agents on each node and one parent agent per cluster. Each child is running as a DaemonSet and pushes its data to the Netdata parent instance, which is then collected by our Prometheus instance. Prometheus is the mainstream monitoring tool in Kubernetes clusters. It scrapes standard Kubernetes components through the default Kubernetes API. Beside the default metrics, with additional collector extensions, Prometheus can monitor more, both operating system and Kubernetes related KPIs. We used the following extensions with Prometheus: i) Node exporter [23]; ii) Dmesg exporter [24]; iii) Node problem detector [25]; iv) Kube-state-metrics [26].

In our proposed solution we rely on Prometheus' time series data storage capability not only because of its efficient store mechanism, but also because of its advanced query language and alerting rules. PromQL (Prometheus Query Language) is the official functional query language of Prometheus. We also take advantage of its potential during the pre-process phase (see Sec. IV). We deployed Grafana [27] as our graphical visualization tool.

C. Fine-tuning the resource footprint vs. data trade-off

We strived to find the balance between the necessary resource consumption and the amount of collected data. We have measured the resource footprint of the monitoring system in our Kubernetes cluster with 3 master nodes and 16 worker nodes for a set of sampling rate settings. The sampling rate (or scrape interval) vs. resource footprint is the main trade-off to be addressed in the monitoring system: the higher rate we apply, the more information on the cluster status we gain, but the larger the footprint of data collection is. The CPU and memory consumption of the exporters (Dmesg-exporter, Kube-state-metrics, Node-problem-detector, Prometheus-node-exporter, Netdata-child) are around a few millicores and under 100MB, respectively, and they do not significantly vary when applying different sampling rates. We note that the variance of resource consumption of Netdata-child is higher than the other components', but it is still negligible compared to that of the collectors. The computational resource consumption for both collectors (Netdata and Prometheus) are decreasing as the scrape interval grows. However, our monitoring solution is lightweight: the Prometheus exporters and the Netdata-children, which burden all cluster nodes, consume little amount of computational resources, while the collectors consume 0.5 vCPUs and 4GB of memory altogether at a sampling rate of 15s.

IV. Analytics pipeline

Our data analytics pipeline is depicted in Fig. 2, in this section we describe the most important steps in details.

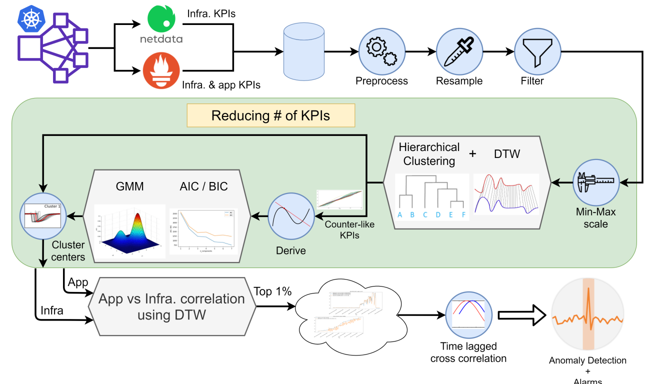

Fig. 2. Data analytics pipeline's flow chart

A. Data preparation

After preprocessing the measurements from the two collectors (e.g., unifying naming schemes, handling modifiers), the time series are resampled. As a result, the timestamp of the data points in each time series will be uniform, they will have the same frequency, and the same interval. This is necessary because the two monitoring systems may not have sampled the KPIs at exactly the same timestamps. Additionally, we can set arbitrary sampling frequency. Resampling is performed by linear interpolation.

Then a filtering is performed to greatly reduce the amount of data to be processed without losing valuable information. During this filtering, we discard the measurements that had zero variance during the measured period.

B. Time series clustering for dimension reduction

In the data analytics process we want to compare application and infrastructure KPIs. Due to the high number of KPIs collected, we need to reduce their number in order to reach an acceptable runtime of the pipeline. The size of KPI sets is reduced by eliminating redundancy, we use clustering for this purpose. The KPI set reduction method is performed separately for application and infrastructure KPIs. The first step in the KPI set reduction process, as shown in Fig. 2, is a Min-Max scaling: each time series is mapped to the 0-1 interval. This is necessary because in the following we are only interested in the shape of the time series, i.e., their characteristics, but not in the absolute value of the measurements. After scaling, these time series become comparable.

As a next step, Hierarchical Agglomerative Clustering is used to cluster the time series. This algorithm scales well with the number of KPIs, which makes it efficient for large number of KPIs. To use this algorithm, we need to utilize a distance function to describe the similarity of two time series. The distance between two time series was calculated using Dynamic Time Warping (DTW). This algorithm examines the distance between each data point in the time series, taking into account that there may be time delay between them or they might be changing at different pace. To cluster the time series, we need to determine how many clusters we want to create. There are several algorithms for determining this value, e.g., Silhouette-score, Elbow-method. The fewer clusters we choose, the more likely we group two time series that are not similar. We used the Silhouette-score, which in our experience, shows well the ideal number of clusters. In our case, we prefer to create a little bit more clusters, thus KPI set reduction is less effective, but we lose less information.

The clustering step results in many small clusters and one large cluster. This large cluster contains all KPIs that are monotonically increasing, such as KPIs that calculate the time elapsed since a particular event, or KPIs that aggregate or count events. There are several similar ones among these KPIs, but it is not practical to cluster them together. Therefore, the next step is to separate this larger cluster and break it down into further smaller clusters using another method. We derive all time series in the cluster (e.g., with the PromQL function *rate()*) so that we can compare them based on their slope. Then the first derivative of these time series are further clustered using the Gaussian Mixture Model. Although this algorithm does not scale well, it is a smaller cluster than the original, so it does not slow the pipeline. This algorithm has the advantage over simpler (e.g. K-means) algorithms that clusters can have more complex shapes. The number of clusters is determined by the AIC/BIC method.

The result of this second clustering and the result of the first clustering (except for the largest cluster) are merged. The time series in each cluster are reduced to one time series by averaging the data points of the time series, i.e., by calculating the cluster centers. Our KPI-reducing clustering method takes into account delays between measurements, and separately addresses monotonically increasing metrics that often occur in cloud infrastructure monitoring systems. The only hyperparameter of the clustering methods, i.e., the number of clusters, is determined using a heuristic approach, thus the process does not require the involvement of an expert at this point.

C. Correlation analysis

Next, we seek similar pairs between application and infrastructure KPI clusters. To this end, we calculated the distance between each application and infrastructure cluster pair using DTW, which allows for a delay between the two time series. Our ultimate goal is to look for causal cases where a change in the infrastructure metrics implies a change in an application KPI, possibly after some delay. After calculating the distance between each cluster pair, the closest 1% of all pairs were examined. Among the selected top 1% of discovered similarity cases there are many pairs that move together for an evident reason, e.g., reflecting the same performance metrics. Therefore, a domain expert must examine each of the most promising set of pairs to find the correlation cases that are of interest, and can drop correlated, but inexplicable ones. Moreover, clusters provide additional information about which KPIs describe the same behavior. With this method, the expert can also test hypotheses by comparing the values obtained.

We calculated a Time-Lagged Cross Correlation between the found and validated pairs, which is used to determine the amount of delay between the two KPI clusters and the degree of correlation between them in the case of this delay. Based on this knowledge, we can create predicting alerts for application failures triggered by an anomaly detection system that signals faults in the infrastructure.

V. Use cases and results

In our measurement scenarios we generated test traffic to a selected application, i..e., Elasticsearch, and emulated various infrastructure failures to find correlations between the behavior of application and infrastructure KPIs. In this section we present our infrastructure performance deterioration implementation, the selected application and our results.

A. Data collection during performance deterioration

We emulated four performance deterioration scenarios, where we consumed or spoiled some of the infrastructure

capabilities. We observed that in the case of deterioration of certain physical properties, which quality of service KPIs deteriorate on the application side. The emulated performance deterioration scenarios consisted of: network latency and packet loss increase; CPU and memory resource consumption with an emulated "noisy neighbor".

The application's hosting node had 8 CPU cores, 32GB memory as computational resources. We increased the latency and packet loss percentage using the *tc* command. We emulated increasing latency and also packet loss on a particular network interface on the application's node, resulting in degraded performance of all applications on that node. A noisy neighbor application was also simulated on the same node as the monitored application. The purpose of this noisy application was to consume the node's available CPU and memory resources. Both the CPU and memory consumption were realized with the application *stress-ng*.

We performed measurements with the deterioration of the previously described infrastructural properties, during which our monitoring system collected infrastructure and application data. Each measurement began with a short period of data collection when the system was operating in "normal" state, i.e., without any deterioration. This was followed by longer periods of data collection with deteriorating values of a given infrastructural property. At the end of the measurement, another shorter period of data collection followed, in which case the system returned to its "normal" state.

B. Load generation for the selected application

In order to experience a decrease in application performance during an infrastructure property deterioration, we generated load for the chosen application. Elasticsearch [28] is a distributed, search and analytics engine. We generated tens of thousands of objects that were submitted to the application in just a few seconds. Elasticsearch stores and indexes these objects in its database. During the measurements, after some time we deleted all the stored objects from Elasticsearch, then started to submit them again. With this generated load Elasticsearch performed some internal work, which we could observe with the monitored application data.

C. Results

In this section we present two cases that were yield by our monitoring system and analytics pipeline. In order to validate the real correlations between infrastructure and application data, we examined whether the discovered correlations existed in several measurements, so we present the results of 3 experiments for each case.

In the first case, network latency was deteriorated periodically. We set the latencies on the node's network interfaces to 50ms, 100ms, 250ms, 500ms and let the system run for 5 minutes with each latency value. In the meantime, we examined whether this error shows in any infrastructure KPI and whether that infrastructure KPI correlates with an application KPI. We present the most promising pair of potentially correlated KPIs resulting from our data analysis pipeline. From the infrastructure side, we found two KPIs:

1) '*scrape_duration_seconds*': Prometheus' own KPI, which measures how long it takes to scrape the pod metrics;
2) '*netdata.execution_time_of_coredns_time*': reported by Netdata that tells us how long the execution of the CoreDNS resolve queries take.

From the application side, we found the following KPI reported about the *search-engine-data* pod:

1) '*http_requests_duration_microseconds*': shows the various percentiles of time duration of ElasticSearch queries: the 99th, 90th and 50th percentiles.

Fig. 3 shows the results of 3 measurements. The values of KPIs are shown normalized for easier comparison. The value of infrastructure KPIs (dashed lines) increase with the value of application KPIs, sometimes a little earlier. However, it can be observed that the purple dashed line breaks at the highest latency deterioration and only reappears at the end of the measurement when the system returns to normal operation: Netdata was unable to retrieve system metrics due to a timeout, so it did not report any data. In the meantime, Prometheus kept reporting the selected metrics. An important lesson can be drawn from this: it is worth using several monitoring data collectors in parallel, because by doing so, we have higher chance to get information about the state of the system during critical conditions.

In the second case the phenomenon of noisy neighbors was tested where we consumed the resources under the running pods. In this scenario we periodically increased the memory consumption on a cluster node: 15G, 16G, 17G and 18G for 5 minute periods each. During these tests we observed that containers shut down due to memory starvation. In each of the 3 tests we performed, the *search-engine-ingest* and the *search-engine-master* pod shut down simultaneously. On the infrastructure side we examined the memory failures reported by other pods. To this end, we examined the '*container_memory_failures_total*' KPI reported by Prometheus. On the application side we examined the following KPIs:

1) '*elasticsearch_cluster_health_status*': shows the health status of each node in the Elasticsearch cluster (not the Kubernetes cluster). It has 3 colors: red, yellow, green: if the red status flag is raised, it means that the container is shut down.
2) '*elasticsearch_cluster_health_unassigned_shards*': shows the number of unassigned shards in the Elasticsearch cluster.

As it can be seen in Fig. 4, in each measurement short before the Elasticsearch pods shut down, other containers in the cluster reported memory errors.

In summary, we see in the two cases that our data analysis method can be used for both reactive and proactive fault management. In the first case, the results can be used in a root-cause analysis framework. This solution is reactive fault management, as we can only obtain information about

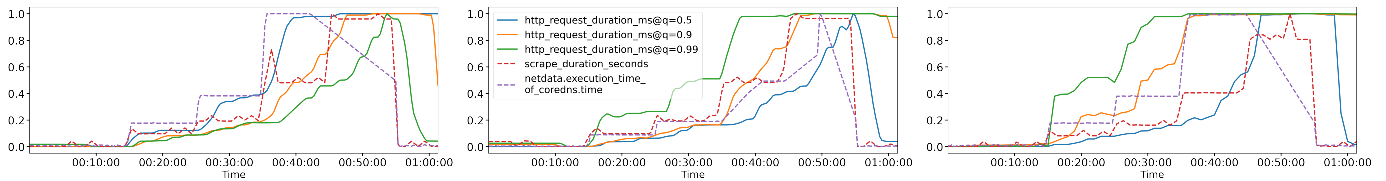

Fig. 3. Latency deterioration measurements

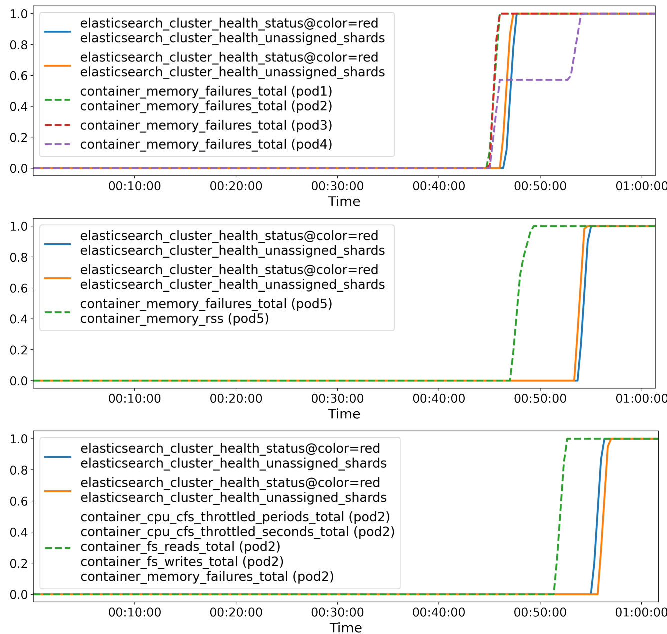

Fig. 4. Memory surge measurements

the error from the examination of the infrastructure KPIs when it already has a critical impact on the applications. However, in the second case, by monitoring memory errors of other containers, we can build a proactive system, in which the system tells the application pods to be aware that other containers report errors, thus it will most likely be affected, too. Then the application can prepare accordingly, e.g., save its state, before shutting down due to memory starvation.

VI. Conclusion

In this paper we presented a low footprint cloud monitoring framework to be deployed in Kubernetes managed data centers. Our goal was to cover as many infrastructure and cloud platform-related metrics as possible, while keeping the resource footprint of the monitoring apparatus low. Hindered by the large amount of collected data, we turned to machine learning for discovering patterns of interest: we sought inference between infrastructure-, and application-level metrics especially in times of distress. To this end we trained AI models for the most frequent failure events in the cloud: network issues and noisy neighbors. Via the showcased experiments, we propose this monitoring system and data analytics pipeline root-cause analysis frameworks, and for real-time, proactive failure avoidance of applications, e.g., the learned models can trigger Prometheus alerts for the application provider.

References

[1] D. Haja *et al.*, "Location, Proximity, Affinity – The key factors in FaaS," *Infocommunications Journal*, vol. 12, no. 4, pp. 14–21, 2020.

[2] "Amazon CloudWatch," https://aws.amazon.com/cloudwatch/.

[3] "Kubernetes," https://kubernetes.io/.

[4] S. Barbhuiya *et al.*, "A lightweight tool for anomaly detection in cloud data centres." in *CLOSER*, 2015.

[5] T. Wang *et al.*, "Fault detection for cloud computing systems with correlation analysis," in *IFIP/IEEE IM*, 2015.

[6] ——, "FD4C: Automatic fault diagnosis framework for web applications in cloud computing," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 46, no. 1, pp. 61–75, 2015.

[7] H. Hotelling, "Relations between two sets of variates," in *Breakthroughs in statistics*. Springer, 1992, pp. 162–190.

[8] R. P. Bagozzi *et al.*, "Canonical correlation analysis as a special case of a structural relations model," *Multivariate Behavioral Research*, vol. 16, no. 4, pp. 437–454, 1981.

[9] D. R. Hardoon *et al.*, "Canonical correlation analysis: An overview with application to learning methods," *Neural computation*, vol. 16, no. 12, pp. 2639–2664, 2004.

[10] M. Farshchi *et al.*, "Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis," in *IEEE ISSRE*, 2015.

[11] ——, "Metric selection and anomaly detection for cloud operations using log and metric correlation analysis," *Journal of Systems and Software*, vol. 137, pp. 531–549, 2018.

[12] S. Kobayashi *et al.*, "Causal analysis of network logs with layered protocols and topology knowledge," in *2019 15th International Conference on Network and Service Management (CNSM)*. IEEE, 2019, pp. 1–9.

[13] C. Shen, "Analysis of detrended time-lagged cross-correlation between two nonstationary time series," *Physics Letters A*, vol. 379, no. 7, pp. 680–687, 2015.

[14] D. J. Berndt *et al.*, "Using dynamic time warping to find patterns in time series." in *ACM KDD workshop*, 1994.

[15] M. A. Munawar *et al.*, "Adaptive monitoring in enterprise software systems," *SysML*, 2006.

[16] S. Kobayashi *et al.*, "Mining causality of network events in log data," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 53–67, 2017.

[17] A. Messager *et al.*, "Inferring functional connectivity from time-series of events in large scale network deployments," *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 857–870, 2019.

[18] S. Y. Shah *et al.*, "Dependency analysis of cloud applications for performance monitoring using recurrent neural networks," in *IEEE Big Data*, 2017.

[19] "Helm: The package manager for Kubernetes," https://helm.sh/.

[20] "Cloud Native Computing Foundation," https://www.cncf.io/.

[21] "Netdata," https://www.netdata.cloud/.

[22] "Prometheus," https://prometheus.io/.

[23] "Prometheus Node exporter," https://github.com/prometheus/node_exporter.

[24] "Dmesg exporter," https://github.com/cirocosta/dmesg_exporter.

[25] "Kubernetes Node-problem-detector," https://github.com/kubernetes/node-problem-detector.

[26] "Kubernetes Kube-state-metrics," https://github.com/kubernetes/kube-state-metrics.

[27] "Grafana," https://grafana.com/.

[28] "Elasticsearch," https://www.elastic.co/elasticsearch/.