2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)

Maintaining SLOs of Cloud-native Applications via Self-Adaptive Resource Sharing

Vladimir Podolskiy*, Michael Mayo†, Abigail Koay†, Michael Gerndt*, Panos Patros†
*Chair of Computer Architecture and Parallel Systems, Technical University of Munich, Germany
v.podolskiy@tum.de, gerndt@in.tum.de
†Department of Computer Science, University of Waikato, New Zealand
{michael.mayo, abigail.koay, panos.patros}@waikato.ac.nz

***Abstract*—With changing workloads, cloud service providers can leverage vertical container scaling (adding/removing resources) so that Service Level Objective (SLO) violations are minimized and spare resources are maximized. In this paper, we investigate a solution to the self-adaptive problem of vertical elasticity for co-located containerized applications. First, the system learns performance models that relate SLOs to workload, resource limits and service level indicators. Second, it derives limits that meet SLOs and minimize resource consumption via a combination of optimization and restricted brute-force search. Third, it vertically scales containers based on the derived limits. We evaluated our technique on a Kubernetes private cloud of 8 nodes with three deployed applications. The results registered two SLO violations out of 16 validation tests; acceptably low derivation times facilitate realistic deployment. Violations are primarily attributed to application specifics, such as garbage collection, which require further research to be circumvented.**

I. Introduction

Deploying applications on lightweight containers enables fine-grained control of the distribution of resources on multi-tenant clouds. This increases the density of cloud systems and improves their capacity to dynamically resize resources based on levels of load, which is referred to as elasticity.

However, using containers lowers isolation and increases the risk of violating service requirements. Acceptable levels of performance are specified by Service Level Objectives (SLOs) which define acceptable performance thresholds. Cloud providers aim to 1) satisfy application owners' by enforcing SLOs; 2) decrease the overprovisioning of hardware resources to make room for new tenants and decrease operational costs; and, crucially, 3) continue doing so while SLOs, demand levels, deployed applications and hardware change dynamically. Contemporary container orchestration platforms, e.g. Kubernetes, expose an API to manage the number of containers (horizontal scaling) and the size of resources they receive (vertical scaling). Horizontal scaling is preferred but it is not as effective when the cloud has become saturated [1].

In this paper, we study autonomic vertical scaling for co-located applications on saturated containerized clouds. We investigate the connection between Service Level Indicators (SLIs) such as response time, and the workload, resource limits and SLIs of other tenants. The research question is how should resources be shared among co-located containerized applications if adding extra instances is no longer possible.

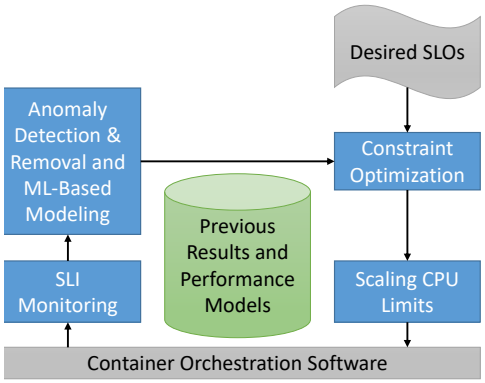

Fig. 1. MAPE-K-Inspired Architecture of the Proposed Technique

The research aims were achieved by 1) deploying a private Kubernetes cluster, 2) stress-testing it with sample applications to create a dataset, 3) building machine learning-powered models to predict SLIs, 4) applying single- and multi-objective optimization on the models to derive CPU limits such that SLOs are met while resource consumption is minimized, and 5) conducting validation tests of the derived resource limits.

Our proposed solution (Fig. 1) is inspired by the Monitor-Analyze-Planning-Execution-Knowledge (MAPE-K) architecture for self-adaptive software [2] and aims to facilitate goal- and constraint-driven adaptation under uncertainty caused by co-location-induced performance interference. The main contribution of this work is a novel approach for deriving SLO-compliant resource limits for application containers in multi-tenant clouds based on a) machine learning techniques for constructing performance models and b) single- and multi-objective optimization. More broadly, the paper also studies the synergy of machine learning models and anomaly-removal that take the role of the *analysis* (A) component of MAPE-K as well as multi-target optimization models serving as its *plan* (P) component. The paper also contributes to the design of testbed cloud environments.

II. Background and Related Work

Containers are managed by a runtime, such as Docker Engine. Dockerfiles describe a set of commands that bring a container to a desirable state, which is subsequently stored

1949-3681/19/$31.00 ©2019 IEEE
DOI 10.1109/SASO.2019.00018

as an image in a repository. Docker can start a new container based on an image and apply desired resource limits[1].

Cloud Services Providers (CSP) rely on container orchestration software, such as Kubernetes, to autonomically manage the life-cycle of containers. Kubernetes manages a cluster of physical and/or virtual nodes. A subset of these nodes take the role of a master, the remaining are workers. Each worker node contains a controller, called `kubelet`, which receives instructions from a master regarding the desired state it needs to maintain in its node and then uses a container runtime software (commonly Docker) to create, manage and interconnect containers inside pods. Kubernetes limits the CPU a pod receives in the following two ways: (i) **Soft limits** using CPU shares, which guarantee that at least a certain percentage of the CPU will be available to the processes of a container; and (ii) **hard limits** using CPU slices, which guarantee that no more than a given percentage of the CPU will be awarded to a container. In both cases, the limits are expressed in millicores or milliCPUs (mCPUs), a milliCPU being one $\frac{1}{1000}$ of a worker node's CPU.[2]

The area of Self-Adaptive Systems studies such self-managing technologies. These technologies started with "haphazard" scripts that automated infrastructure management and progressed with the establishment of Self-Adaptive architectures, such as the flagship Monitor-Analyze-Plan-Execute-Leveraging-Knowledge MAPE-K Loop. Later, runtime performance models were identified as means to adapt the infrastructure such that SLOs are satisfied and guarantees are provided. Control-theory has also been proposed as an alternative; however, it is beyond the scope of this paper [2].

Optimal application placement in the cloud and cloud/edge environments is a recent research topic. [3] proposes to use cost and round trip time metrics to decide whether to place latency-critical Internet of Things applications to edge or to the central data center. An approach based on the notion of *technical debt* was proposed in [4] to manage the elasticity of SaaS applications by establishing coalitions of applications sharing the cloud resources to fulfill the corresponding SLOs.

Studies in resource allocation for placing virtual machines in data centers address similar cloud resource management challenges but on the level of virtual infrastructure. In [5], the authors present an approach to SLO- and performance-aware resource allocation in overbooked data centers which is also based on a constrained optimization task. The problem of the scalability of resource allocation for virtual machines can be addressed by solving an optimization problem in a distributed peer-to-peer manner over a scale-free overlay as shown in [6]. The vertical elasticity of VMs to meet SLOs imposed on cloud applications was studied from the control-theoretic point of view in [7]. The proposed approach to vertical scaling was extended with probabilistic cloning of requests in [8], which improved SLO-compliance of running applications and reduced over-provisioning in comparison to [9]. The effects of performance interference between co-located VMs on SLOs were thoroughly studied in [4], resulting in a technique that, depending on unmet demand and spare capacity, reconfigures the load balancer and adjusts the scaling actions.

Container performance interference for PaaS clouds has additionally been investigated in [1] and [10]: the authors measured slowdowns inflicted on PaaS tenants based on the resource-intensiveness, calculated performance models for CPU utilization, and proposed request-reordering to maximize SLO compliance, but did not leverage data-driven adaptation.

Our study conceptually differs from the previous works by focusing on containerized applications instead of VMs, by also considering soft resource limits[3], and by incorporating the data-based SLIs prediction models into the SLO-compliant resource limits derivation via an optimization-based technique.

III. Dataset

To investigate the efficacy of our approach, we collected experimental data consisting of **8,864** observations from a private cloud we created for this purpose. We decided against experimenting on public clouds because of their unpredictability in consistently allocating the same types of physical resources and their loose performance isolation guarantees[4]. Load to our cloud was driven by a separate machine, which also recorded SLIs. Multiple configurations of the deployed applications were repeated 500 times, each time randomly selecting the available CPU resources and the level of concurrency (number of parallel requests) driving their load.

A. Private Kubernetes Cloud

We used a single physical machine with an Intel(R) Xeon(R) CPU E5-2670 @ 2.60GHz with 32 virtual cores, 256GB of RAM and running Ubuntu 18.04.1 LTS (Bionic Beaver). Using a Vagrant and Virtual Box script from Oracle[5], we deployed a Kubernetes private cluster with 8 virtual nodes. Each node was a VMBox Virtual Machine running Ubuntu with 4 VCPUs and 15GB of RAM. One of the nodes was selected as the master node and the remaining 7 as worker nodes. To minimize the effect of performance interference, no other user accessed the machine while our experiments were running. The single-site single-machine configuration of the private cloud for the experiments was selected to avoid the influence of the network effects on the model for the co-located applications.

B. Deployed Applications

We started the following three deployments on our private cloud: **Nginx**, **Liberty** and **Guestbook**. First of all, **Nginx** (*App1*) exposes a Kubernetes Load Balancer service, which forwards incoming requests to one of the available replicas of an nginx:1.15.6 container that produces a hello-world response.

[1]https://www.docker.com/

[2]https://kubernetes.io/

[3]The amount of resources that is guaranteed for the container, also known as *resource request* in the Kubernetes' terms. The actual amount of allocated resources could be higher, but not lower than the soft limit.

[4]Our investigation happens from the point of view of the cloud provider.

[5]https://github.com/oracle/vagrant-boxes

Secondly, **Liberty** (*App2*) exposes a Kubernetes Load Balancer service, which in turn forwards incoming HTTP(s) requests to one of the available replicas of a websphere-liberty:18.0.0.3-javaee8 Docker container. A default hello-world response is produced by each liberty pod when a request is made at it. However, unlike nginx pods, liberty pods run a more complex stack of applications. Each pod executes the Eclipse OpenJ9 Java Virtual Machine, which is a runtime for Java. On top of OpenJ9, the Java application Liberty profile is running, which is a Java EE application server.

Thirdly and finally, **Guestbook** (*App3*) is a three-tier Kubernetes tutorial application. Its PHP front-end enables users to view and add comments that are stored on a Redis backend. A Kubernetes Load Balancer service is exposed to forward incoming requests to replicas of a Guestbook container implementing the frontend part of the application. The PHP pods are connected through an internal Kubernetes service to a single Redis-master pod, which drives replicated Redis-slave pods.

C. Load-Driving

Our load-driving script was executed on a separate machine with an Intel(R) Xeon(R) CPU E5-1620 v3 @ 3.50GHz with 8 virtual cores, 16GB of RAM and running Ubuntu 16.04.5 LTS (Xenial Xerus). We placed our load-driving machine on the same network and building as our Kubernetes cluster to minimize network-related performance unpredictability. The script also controlled the vertical scaling of the deployed containers via remotely accessing the Kubernetes API.

The goal of our load-driving script was to explore various combinations of levels of concurrency and allocated CPU resources for each of the three deployed applications. The ranges explored were between 100 and 700 mCPUs per pod and between 20 and 80 clients repeatedly firing requests in parallel. For all applications, we set the number of replicas to 7, to match the available worker nodes and saturate the system.

The script used the load-tool Apache `ab` to send requests. The experimental results were averaged across the 16 cycles of each configuration and accumulated in a `csv` file with `awk`. Each request-firing cycle lasted for 60s. Each random configuration of concurrencies and mCPUs was repeated for 16 cycles and 500 random configurations were tested. Because Kubernetes restarts services to perform vertical scaling, our script waited until all pods were ready. We conducted two sets of experiments using hard and soft CPU limits.

IV. Deriving SLO-compliant Resource Limits

Derivation of resource limits for co-located containerized applications such that SLOs imposed on these applications are met (further addressed as SLO-compliant resource limits) is challenging because of several reasons. First, the specifics of the particular application or technology stack might produce sporadic artifacts in the collected data. Second, accurate resource limits might only be identified with trial-and-error for real workloads in the production environment on all the potential deployment settings, which is unfeasible. Third, the effects of co-location of applications might interfere with the collected SLIs (response time and throughput) in unexpected ways. Finally, SLO-compliance should be ensured for all co-located applications at the same time.

The outlined challenges in SLO-compliant resource limit allocation for containerized applications in dynamic environments are addressed with the following steps. Firstly, by **removing anomalies** that cannot be explained by available features; and secondly, by **learning prediction models** that relate available features (resource limits, request rates, SLIs of other applications observed simultaneously) to SLIs. Various types of prediction model are considered, evaluated and selected for further use based on their efficacy in predicting SLIs: independent (each SLI predicted independently), application-wise (all SLIs are predicted per application), SLI-wise (SLI is predicted based on all applications), all-targets (all SLIs are predicted for all application). The third way that the challenges are addressed is by **deriving resource limits** for co-located containerized applications, optimizing SLIs and ensuring that predefined SLOs are met. We envision these steps in a MAPE-K loop that periodically applies the derived limits to perform autonomic vertical scaling along side load forecasting.

The evaluation of the proposed steps can be–and were–conducted by deploying the derived configuration, generating load, and comparing the actual SLIs with the SLOs set. Each step of the proposed approach as well as its evaluation is studied in detail in the following sections of the paper.

V. Models to Predict SLIs

The target variables for prediction are SLIs for: 1) The 99%-tile of response time (ms), which is defined as the time 99% of requests were processed. 2) Throughput (requests per second, RPS), which is the number of requests processed per second.

The model features can be divided into three groups: First, **unmanageable features** representing user demand, denoted as *Concurrency_i* for the ith application and measured in RPS. Though these features can be set during stress-testing, when we apply the model these features will become the workload of the end users, which can only be *forecasted*.

Second, **manageable features** represent the resource limits that are allocated to applications: $mCPUs_i$ for the ith application and measured in millicores (a.k.a., mCPUs or shares of processor time). We note that other resource types could be considered considered here but we focus on just one.

Third, **partially-manageable features** represent SLIs of other co-located applications. These SLIs can be partially influenced by the amount of allocated resources; the other influence is through unmanageable features, such as the load of the target and any co-located applications. Crucially, introducing such features enables modelling the effects of co-location.

A. Selecting a Modeling Approach

The aim of our initial modelling attempt was to select an ideal machine learning technique for predicting SLIs from the combination of features. To this end, we tested three common approaches: linear regression, which performs ordinary least-squares regression and produces a linear model; lasso

Algorithm	*Prediction Target*		
	Resp 1	Resp 2	Resp 3
Linear reg.	0.44±0.05	0.04±0.03	0.47±0.05
Linear reg. (poly)	0.56±0.06	0.06±0.05	**0.59±0.06**
Lasso reg.	0.44±0.05	0.04±0.03	0.47±0.04
Lasso reg. (poly)	**0.57±0.06**	0.08±0.03	**0.59±0.05**
Random forest	0.52±0.06	0.09±0.05	0.55± 0.06
Random forest (poly)	0.53±0.07	**0.10±0.04**	0.57±0.06

Algorithm	*Prediction Target*		
	Thru 1	Thru 2	Thru 3
Linear reg.	0.90±0.02	0.65±0.10	0.92±0.02
Linear reg. (poly)	**0.93±0.02**	**0.68±0.10**	**0.95±0.01**
Lasso reg.	0.90±0.02	0.65±0.10	0.92±0.02
Lasso reg. (poly)	**0.93±0.02**	**0.68±0.09**	**0.95±0.01**
Random forest	0.89±0.03	0.65±0.09	0.90±0.03
Random forest (poly)	0.88±0.03	0.65±0.09	0.89±0.02

TABLE I
COEFFICIENT OF DETERMINATION (R^2) RESULTS BY PREDICTION TARGET (COLUMN) AND ALGORITHM (ROW). ENTRIES IN THE TABLE GIVES MEAN PREDICTIVE PERFORMANCE (± STANDARD DEVIATION) OVER ONE DEPLOYMENT-GROUPED 10-FOLD CROSS VALIDATION.

regression, which also produces a linear model but via a more sophisticated method and using an L1 regularization [11], [12]; and random forests [13], an ensemble-based decision-tree technique that usually works well "out of the box". Furthermore, since the total number of manageable and unmanageable features is small, we also considered each of the above algorithms in conjunction with second order polynomial features, such as *Concurrency_2*$\times mCPUs_1$ and $(mCPUs_3)^2$. The rationale for including such additional features is that they better enable linear models to capture non-linear patterns in the data.

The results of our initial explorations in this direction (Table I) indicate that lasso-based regression with polynomial features is the best performer in terms of R^2. Furthermore, we observed that the lasso models, due to the typically heavier degree of regularization of the model, were substantially simpler (i.e., had significantly more zero coefficients) than the linear regression models. The only SLI where lasso (and all other methods, in fact) fail is 99%-tile response time for **Liberty**. In light of this, we proceeded with subsequent machine learning experiments using the lasso model (and its multi-target variant) with second order polynomial features.

B. Removing the Anomalies

Variation of observations is usually considered an asset in machine learning tasks [14]. However, the absence of features in a dataset might render so-called anomalous samples deleterious to the quality of the model. Ideally, anomalous features need to be included in the training set to improve the accuracy of predicting the normal behaviour of the system. In the following paragraphs we evaluate the impact of anomalous samples in the collected dataset on predicting SLIs and introduce an approach to prune the anomalies.

1) Influence of Anomalies on the Quality of Models: Preliminary analysis of distributions for the 99%-tile response time and throughput for all three applications clearly points out anomalies among the observations of the 99%-tile response time for application 2, **Liberty** (Fig. 2). It turns out that around **8.2%** of all observations for the response time are higher than

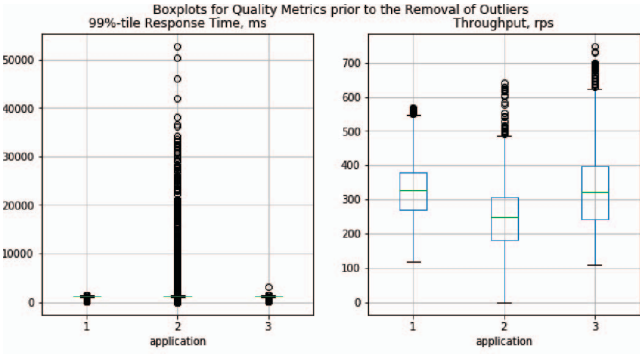

Fig. 2. Distribution of SLIs values in the dataset, five number summary.

Fraction of removed observations	*Averaged R^2 Score*	*R^2 Score variance*
0.00	-0.07	0.2391
0.01	0.16	0.0100
0.02	0.24	0.0128
0.03	0.33	0.0116
0.04	0.36	0.0100
0.05	0.43	0.0127
0.06	0.51	0.0083
0.07	0.57	0.0054
0.08	0.58	0.0050
0.09	0.59	0.0044
0.10	**0.61**	**0.0036**
0.11	**0.61**	**0.0036**
0.12	**0.61**	**0.0038**
0.13	0.60	0.0037
0.14	0.59	0.0035
0.15	0.59	0.0033
0.16	0.58	0.0030
0.17	0.57	0.0026
0.18	0.56	0.0025
0.19	0.56	0.0026

TABLE II
IMPACT OF THE FRACTION OF REMOVED ANOMALOUS OBSERVATIONS ON THE R^2 SCORE OF THE FITTED 6-TARGET LASSO REGRESSION MODEL

1.5s, which in practice means failed requests as an end-user is unlikely to wait for so long to receive a response. Although observations above 1.5s can be used as a threshold to identify anomalies, there are also anomalies that might impact the quality of the model that are below the threshold, such as network issues that result into a near-zero response time.

The influence of anomalies on the quality of prediction models is easy to track by removing an increasing fraction of anomalies from the dataset and evaluating the R^2 score on the pruned dataset. Table II shows that the most significant improvement in 10-fold cross-validated R^2 scores for the 6-target prediction model of order 2 is for a removed anomalies fraction between **0.10** and **0.12**. The identification of anomalies is done using Isolation Forest [15]. Crucially, setting the fraction of removed anomalies to **0.11**, causes the distribution of the 99%-tile response time for the second application (**Liberty**) to approach normality (see Fig. 3).

2) Unsupervised Anomaly Identification and Removal: The approach proposed in the previous subsection could, in principle, be used for removing anomalies. However, the

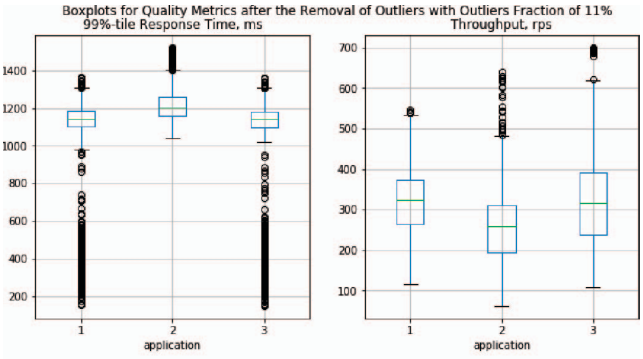

Fig. 3. Distribution of SLIs values in the dataset with 11% of anomalies removed with Isolation Forest, five number summary.

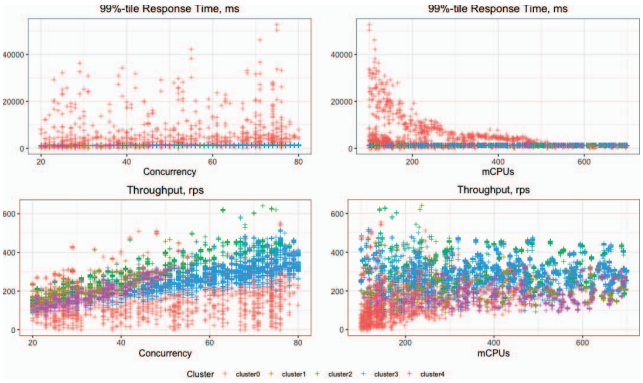

Fig. 4. Clusters Created using EM Algorithm

parameters of Isolation Forest depend on the actual data, which generally differ among applications. Accurate manual tuning of hyperparameters for Isolation Forest is not compatible with the autonomic goals of this paper. Hence, to automate anomaly-removal, we propose utilizing clustering algorithms. Unlike Isolation Forest, clustering avoids using a fixed anomaly fraction. Instead, it results in grouping subsets of the data by identifying latent similarities.

Thus, we used an Expectation-Maximization (EM) algorithm [16] to perform 6-target clustering. EM creates a number of clusters that best capture similarities between observations. We set EM to provide 10-fold cross-validation and finish in 100 iterations. This clustering method is performed on pre-processed data where the corresponding median of each deployment is used instead of the individual SLI values.

The results of applying EM (Fig. 4) clearly show 5 clusters: *cluster0* represents the anomalies of the dataset, capturing *"too high"* or *"too low"* SLI values. Also, this cluster contains **13%** of the data and is considered anomalous as it had the highest standard deviation (**582.78ms**) among the other clusters whose standard deviation was **34–38ms**. Crucially, the fraction of the data in *cluster0* is close to the optimal fraction of removed anomalies (0.10–0.12) as shown in Table II.

3) Use of Anomalies for Search-Space Reduction: Knowing that anomalies might be related to the configuration parameters, one might try to use the identified anomalies to limit the search space of potential applications' resource allocations. For that, it is necessary to derive a model of an anomaly, i.e., to relate the resource allocation parameters to the SLIs.

Model Degree	*Averaged R^2 Score*	*R^2 Score variance*
1	0.43	0.0119
2	0.66	0.0040
3	0.76	0.0049

TABLE III
IMPACT OF SLIS AS PREDICTORS ON ANOMALIES MODELS

Anomaly detection on the given dataset resulted in observations of two types being labeled as anomalies: observations with very low response time for Application 2 and observations with high response time for the same application. In the absence of data about error codes, one should focus on the requests that take the longest to be processed. In the dataset, there are **710** observations that took longer than **1,521** ms to complete. This threshold is the maximal response time after filtering out the **11%** of outliers with Isolation Forest. Leaving only these observations for modeling, we are able to cover **72.8%** of all the anomalies in the dataset. These anomalies are important to model as it is not known whether low response times that are also considered anomalies are due to the response being completed with error.

Deriving a lasso regression model for the filtered anomalies is not straightforward as the dataset does not contain the direct predictors of anomalies. Indeed, fitting a single-target model to predict the 99%-tile response time for the second application ends up with an R^2 score lower than **0.07** for model degrees of 1, 2, 3 using 10-fold cross-validation. However, including the target variables for co-located applications 1 and 3, increases the R^2 score to **0.76** for the degree-3 model (Table III).

Essentially, these results yield two ideas: 1) with the limited data on application-specific parameters (e.g., garbage collection invocations) one can still extract these features indirectly by using SLIs of co-located applications; 2) when building a prediction model for single or multiple applications simultaneously, it might be worthwhile to include SLIs as predictors—that way, one is forced to use the prediction model at runtime, which imposes strict limitations on the computational complexity of the model; however, the accuracy of predictions increases dramatically. The proposed regression-based approach to anomaly modeling might also be changed to the classification-based approach, if the data format allows.

When the regression model is fit to the anomalies, it is possible to specify a threshold on a target variable (SLI) and then to derive valid intervals for predictors. Following, these intervals can be used during resource allocation, limiting the search space for the deployment configuration. Due to size limitations, the derivation of parameters' boundaries according to these ideas was shifted to a future work.

C. Models' Evaluation

The evaluation of the models' quality was done using 10-fold cross-validation and max iterations number of **10,000**

Application	*SLI*	*Degree = 1*		*Degree = 2*		*Degree = 3*	
		$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$
Nginx	RT	0.43	0.0022	0.56	0.0029	0.57	0.0033
	T	0.90	0.0008	0.93	0.0007	0.94	0.0007
Liberty	RT	0.60	0.0199	0.60	0.0177	0.61	0.0173
	T	0.65	0.0177	0.69	0.0182	0.69	0.0191
Guestbook	RT	0.45	0.0035	0.58	0.0037	0.59	0.0045
	T	0.92	0.0004	0.95	0.0003	0.96	0.0002

TABLE IV
EVALUATION OF INDEPENDENT MODELS

Application	*SLI*	*Degree = 1*		*Degree = 2*	
		$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$
Nginx	Response time	0.80	0.0033	0.80	0.0073
	Throughput	0.95	0.0001	0.99	0.0000
Liberty	Response time	0.76	0.0037	0.84	0.0021
	Throughput	0.89	0.0019	0.91	0.0010
Guestbook	Response time	0.80	0.0028	0.81	0.0023
	Throughput	0.95	0.0001	0.99	0.0000

TABLE V
EVALUATION OF INDEPENDENT MODELS WITH TARGET VARIABLES

with grouping by deployment on the anomaly-free dataset. The following notation is used: **RT**, 99% response time; **T**, throughput; $\overline{R^2}$, average R^2 score; and $V[R^2]$, variance of R^2. The evaluation was done on a machine with Intel Core i7-6700HQ (2.60 GHz) and 16 GB of RAM under 64-bit OS Windows 10 HE. The scripts were run on Jupyter Notebook.

1) Independent Models: Independent models predict each SLI individually; thus, for three applications with SLI of 99%-tile response time and of throughput there are six such models. The advantage of these models is their high discriminatory power. However, using such models to derive an SLO-compliant resource allocation might lead to diverging results.

Independent models without target variables as predictors. Table IV shows that independent lasso regression models have a good potential for predicting the throughput of some applications. An additional advantage of the independent lasso regression is its short time needed to fit the model: even for models of degree 3, the fitting time stayed below 1 minute.

Independent models that include target variables as predictors. Including target variables as predictors into independent models improves their predictive power. Each model includes only those target variables as predictors that were not used as their own target variable. Crucially, the average R^2 score of the the degree-2 model does not fall below **0.80** for any target (Table V). Nevertheless, this improvement does not yield consistent resource limits. Moreover, with the introduction of target variables as predictors, the fitting time increased up to 1 min for degree 2. This rendered the use of higher-degree models unfeasible for the most of the tasks requiring fast resource allocation to co-located applications.

2) Application-wise Models: A single multi-SLI prediction model per application is able to produce resource allocation that is consistent for all the SLOs imposed on an application. However, per-application prediction models will still fail in supporting consistent resource allocation across multiple applications. Nevertheless, the use of such models might be justified, since per-application prediction models can easily

Application	*Degree = 1*		*Degree = 2*		*Degree = 3*	
	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$
Nginx	0.47	0.0014	0.59	0.0023	0.60	0.0025
Liberty	0.74	0.0093	0.75	0.0087	0.76	0.0081
Guestbook	0.51	0.0025	0.62	0.0028	0.63	0.0033
	$\overline{t} = 1.44$ min		$\overline{t} = 6.44$ min		$\overline{t} = 68.36$ min	

TABLE VI
EVALUATION OF APPLICATION-WISE MODELS

Application	*Degree = 1*		*Degree = 2*	
	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$
Nginx	0.81	0.0027	0.81	0.0071
Liberty	0.79	0.0046	0.83	0.0032
Guestbook	0.82	0.0021	0.83	0.0017
	$\overline{t} = 5.79$ min		$\overline{t} = 130.91$ min	

TABLE VII
EVALUATION OF APPLICATION-WISE MODELS WITH TARGET VARIABLES

scale with the increasing number of applications due to the mutual independence of such models.

Application-wise models without target variables as predictors. Table VI clearly shows that the combined prediction of two SLIs per application loses accuracy of individual predictions for throughput. In essence, each prediction model's R^2 score is limited by the lowest R^2 score of the target SLIs, what can be inferred by comparing these results with Table IV.

The joint consideration of R^2 scores with the average models fitting times for different degrees (given at the bottom of the table) indicates that it is pointless to increase the degree for application-wise models beyond 2 — the increase of R^2 score by 0.01 is not worth an additional hour of fitting time.

Application-wise models that include target variables as predictors. Including target variables as predictors into application-wise models enables fairly high predictive power. As shown in Table VII, the R^2 score stayed above **0.8** for models with degree of 2. This is never achieved for application-wise models without target variables as predictors, even though the maximal degree of a model is higher. High average fitting time (more than two hours), makes these models impractical.

3) SLI-wise Models: Contrasting, SLI-wise models are able to span multiple applications, hence they can be considered as globally consistent. As each such model uses only one SLI as a target variable (either response time or throughput), the resulting resource limits acquired using these models will be inconsistent for a single application. Such models could scale well for more SLIs in case there are other mechanisms to resolve the inconsistency in derived resource limits.

SLI-wise models without target variables as predictors. Table VIII indicates that the 99%-tile response time is not well-explained by the available features; in contrast, throughput is well-described. This follows from the long-tail distribution of response times in comparison to throughput, with the latter being closer to normally-distributed.

SLI-wise models that include target variables as predictors. Inclusion of the target variables as predictors does not help to significantly increase the predictive power of SLI-wise models as is demonstrated by the average R^2 score in Table IX

SLI (target)	*Degree = 1*		*Degree = 2*		*Degree = 3*	
	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$
Response time	0.41	0.0056	0.54	0.0079	0.57	0.0066
Throughput	0.83	0.0034	0.86	0.0035	0.87	0.0035
	$\overline{t} = 1.62$ min		$\overline{t} = 6.79$ min		$\overline{t} = 60.38$ min	

TABLE VIII
EVALUATION OF SLI-WISE MODELS

SLI (target)	*Degree = 1*		*Degree = 2*	
	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$
Response time	0.48	0.0094	0.59	0.0093
Throughput	0.88	0.0013	0.93	0.0008
	$\overline{t} = 5.71$ min		$\overline{t} = 134.70$ min	

TABLE IX
EVALUATION OF SLI-WISE MODELS WITH TARGET VARIABLES

being only slightly better than that of Table VIII for SLI-wise models that do not incorporate target variables as predictors (by **0.05–0.07**). The significant overhead in fitting time even for the degree of 2 renders use of such a model impractical.

4) All-targets Model: Though the all-targets model allows to predict all targets at once (both SLIs for all three applications in our case), it has two significant drawbacks that render its use questionable: 1) this model scales poorly as new SLIs and applications will make it more complex and will require longer fitting; 2) its predictive capacity is susceptible to the changes even for a single parameter. In order to compensate for the increase in the variety of application types, one would have to include more application-specific predictors that would allow to capture the performance model accurately and to produce reliable resource limits after optimization.

As the model under discussion already predicts all possible target variables, there is only a single case for its evaluation whose results are presented in the Table X. Poor predictive properties of the model can be explained by the presence of the 99%-tile response time for all three applications among the target variables, which was previously shown to be the problematic case for Lasso regression models when considered without other target variables as predictors.

5) Models Comparison: Tables IV, VI, VIII and X clearly show that using lasso regression degrees higher than 2 does not significantly improve the R^2 score and also, results in a larger duration of model fitting step. Hence, lasso regression-based models of degree 2 suffice for predicting SLIs.

Including target variables as predictors into the Lasso regression models increased R^2 (Tables IV–IX). In some cases (e.g. Table VI vs Table VII) this yielded a two-fold increase in R^2; thus, co-location effects strongly influence SLIs.

Evaluating the All-targets model revealed a low predictability of SLIs even with anomalies removed (Table X). In contrast, the available predictors of SLI-wise models suffice to infer throughput but not response time (Tables VIII and IX), which makes them unsuitable. The case of independent models (one model per SLI-application combination) is well captured by Lasso regression; however, it is not applicable due to inconsistencies in the results. Application-wise models with target variables as predictors are the most well-balanced models for our dataset. All these models simultaneously predict both 99%-tile response time and throughput, which makes them both consistent for resource allocation on the level of the application and easily scalable. With the small difference in average R^2 estimates for degree 1 and 2 in Table VII as well as a large difference in fitting times, one could consider using the model of degree 2 while at the same time reducing the number of iterations for the model fit and/or loosening the termination condition. Consequently, this type of model was selected for SLO-compliant resource allocation.

Degree = 1		*Degree = 2*		*Degree = 3*	
$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$
0.50	0.0020	0.61	0.0036	0.63	0.0034
$\overline{t} = 2.16$ min		$\overline{t} = 7.56$ min		$\overline{t} = 62.57$ min	

TABLE X
EVALUATION OF THE ALL-SIX-TARGETS SIMULTANEOUSLY MODEL

VI. SLO-COMPLIANT RESOURCE ALLOCATION

Having models that relate resource limits and workload to SLIs while being purified of anomalies, we now have an *Analyze* (A) component placed in our MAPE-K loop. The next step is the *Plan* (P) stage, which requires utilizing the derived models to acquire resource limits based on the required SLOs satisfaction requirements. In essence, planning is formulated as an SLO-compliant resource allocation problem. Crucially, a Self-Adaptive solution is required to compensate the uncertainty created by the performance interference that co-located containers induce to each other based on their levels of load.

The task of SLO-compliant resource allocation to co-located applications is a multi-objective constrained optimization task. It could be formulated both as an integer programming problem as well as a continuous optimization problem. In our work, we solve the continuous optimization problem 1) to allow the maximum possible flexibility in the resource limits and 2) to minimize the duration of the optimization.

A. Problem Formulation

Consider n co-located applications as well as the throughput SLO (T_i) as $T_i \geq T_i^{(SLO)}$ and the 99%-tile response time SLO (RT_i) as $RT_i \leq RT_i^{(SLO)}$ of the i^{th} application. Also, consider the limitation on the available resources: $\sum_{i=1}^{n} mCPUs_i \leq \sum_{j=1}^{m} cores_j \cdot 1000$ where m is the number of hosts (VMs or physical servers) and $cores_j$ is the number of cores available at j^{th} host. The number of cores is normalized to milliCPUs by multiplying by 1000, hence we consider the homogeneous case where hosts have the same cores. Resource limits on milliCPUs for all the applications under consideration can be grouped into the single vector $\mathbf{mCPUs} = [mCPUs_1, mCPUs_2, ...mCPUs_n]^T$. SLI values predicted with the trained lasso models for the above vector of resource limits are denoted with $\hat{RT}_i(\mathbf{mCPUs})$ and $\hat{T}_i(\mathbf{mCPUs})$ for the 99%-tile response time and throughput respectively. Based on the introduced notations, the following objective functions can be formulated for each application:

$$f_i^{(RT)}(mCPUs) = \frac{\hat{RT}_i(\mathbf{mCPUs})}{RT_i^{(SLO)}} \quad (1)$$

$$f_i^{(T)}(mCPUs) = \frac{\hat{T}_i(\mathbf{mCPUs})}{T_i^{(SLO)}}^{-1} \quad (2)$$

The multi-objective cost function for application-wise, SLI-wise, and all-targets options respectively, is:

$$g_i^{(App)} = f_i^{(RT)}(\mathbf{mCPUs}) \cdot f_i^{(T)}(\mathbf{mCPUs}) \quad (3)$$

$$g_j^{(SLI)} = \prod_{i=1}^{n} f_i^{(SLI)}(\mathbf{mCPUs}), SLI \in \{RT, T\} \quad (4)$$

$$g^{(All)} = \prod_{i=1}^{n} f_i^{(RT)}(\mathbf{mCPUs}) \cdot f_i^{(T)}(\mathbf{mCPUs}) \quad (5)$$

Thus, our problem is formulated as the following multi-objective constrained optimization problem:

$$\begin{cases} \mathbf{mCPUs}_i^* = \arg\min g_i^{(App)} \\ ||\mathbf{mCPUs}_i^*||_1 \leq \sum_{j=1}^{m} cores_j \cdot 1000 \\ \hat{RT}_i(\mathbf{mCPUs}_i^*) \leq RT_i^{(SLO)} \\ \hat{T}_i(\mathbf{mCPUs}_i^*) \geq T_i^{(SLO)} \\ mCPUs_k \geq 100 \, \forall \, mCPUs_k \in \mathbf{mCPUs}_i^* \end{cases} \quad (6)$$

The solution of the optimization problem, as specified in Eq. 6, yields potential discrepancies between optimal resource allocations acquired for each application as $\forall\,(i,j) : i \in 1,2,...n, j \in 1,2,...n, i \neq j$ we have:

$$|\mathbf{mCPUs}_i^* - \mathbf{mCPUs}_j^*| \geq 0 \quad (7)$$

The only multi-objective cost function that would avoid such a discrepancy is $g^{(All)}$, but its use is infeasible due to a linear growth in size the vector **mCPUs**, and the poor predictive properties of the all-target model (Table X).

Next, we introduce an approach to alleviate the discrepancy between the solutions provided by application-wise optimization at the cost of sub-optimality. It is important to note that applications might consist of multiple containerized microservices (as in case of **Guestbook** application)—in that case the resource limits should be allocated to each microservice individually. In the following section we evenly distribute the derived resource limit of the **Guestbook** application among its services, but in practice one should take into account the features of each microservice via sandboxing of microservices for capacity and performance profiles derivation [17].

B. Approach to SLO-compliant Resource Allocation

1) Pure Continuous Constrained Optimization: The non-linear integer programming formulation of the same problem is NP-hard and so far does not have any practically-feasible (i.e. non-time consuming) solution [18]. Hence, the most straightforward and feasible approach to the solution of the optimization problem in Eq. 6 is to solve it as a continuous constrained optimization problem. For that, one can use trust region-based methods for nonlinear constrained optimization [19]. Such a method is conveniently implemented in Python's **scipy.optimize**[6]. Due to the dependence of the form of SLI prediction models on the collected data, the use of the 2-point numerical Jacobian approximation and symmetric-rank-1 (SR1) Hessian update strategy in optimization is justified. The results of the continuous constrained optimization acquired for each application should be summarized (e.g. averaged) and rounded or made integer in some other way.

An initial guess of the parameters to be optimized may severely impact the optimality of the solution as the cost function might have multiple local minima over the parameters' search space. In order to overcome this issue, continuous constrained optimization was performed multiple times for a random $\mathbf{mCPUs_0}$; the final result is the rounded median of results of the conducted random runs. Preliminary tests of continuous optimization showed that **15** random runs are enough to acquire stable optimization results. The proposed adjustment is an adapted version of the Basin-Hopping algorithm [20].

2) Constrained Optimization with Limited Brute Force: The accuracy of the solution provided by the pure continuous constrained optimization might be improved by conducting a brute force search over a small neighborhood of the acquired solution. The following parameters can be specified for such an approach: 1) the search step Δ, e.g., 50 mCPUs, 2) the number of search steps ϕ in the direction of increase and decrease of the proposed solution, which can be approximated by:

$$\phi = \left\lceil \frac{\sum_{j=1}^{m} cores_j \cdot 1000}{2 \cdot n \cdot \Delta} \right\rceil \quad (8)$$

Considering $\mathbf{mCPUs}^*$ as the result of continuous constrained optimization (averaged and rounded over all applications), the search space is limited by: $\mathbf{mCPUs}^* \pm [(\phi \cdot \Delta)_{\times n}]$.

3) Evaluation: The baseline for evaluating the optimality of the solutions produced by the optimization approaches described above was the result of a brute force (exhaustive) search with a grid cell size of $10 \times 10 \times 10$, **BF-10**. Brute force with coarse-grained grid (cell size of $50 \times 50 \times 50$), **BF-50**, was used to highlight the tradeoff between the quality of the acquired parameters' vector and the running time of brute force search depending on the cell size. Both brute force cases used a pre-trained all-target model as in Section V-C4 to acquire an SLI prediction for subsequent global optimization. The results acquired with brute force were compared to the results of the continuous optimization approach, **CO**, from Section VI-B1 and the continuous optimization with limited coarse-grained brute force search on top, **Hyb**, from Section VI-B2.

In evaluation, the following parameters were used: $RT_i^{(SLO)} = 800$ and $T_i^{(SLO)} = 200 \, \forall \, i \in 1,2,3$. The joint limit on CPU was **3000** mCPUs. Concurrencies and SLIs were generated randomly for each test in allowed boundaries; **10** tests were conducted for both the soft and hard limit cases.

[6]docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html

Limit Type	Distance (median)			Execution Time, s (median)			
	BF-50	CO	Hyb	BF-10	BF-50	CO	Hyb
Soft	56.6	1229.2	515.9	209.2	1.68	69.1	71.4
Hard	40.0	1310.6	515.8	121.8	0.95	44.0	45.2

TABLE XI
QUALITY OF SOLUTIONS BY SLO-COMPLIANT RESOURCE ALLOCATION APPROACHES AS COMPARED TO THE BRUTE FORCE

Limit Type	Concurrencies			Resource Limits		
	App1	App2	App3	App1	App2	App3
Soft	58	61	53	200	550	120
Hard	58	63	75	300	500	100

TABLE XII
SETTINGS FOR THE PRELIMINARY VALIDATION TEST

The evaluation results are shown in Table XI. The *Distance* column addresses the Euclidean distance between the optimization result for the considered approach and the result of the fine-grained brute force (**BF-10**). A high execution time of 2–4 minutes of the fine-grained brute force clearly prohibits its use for dynamic reconfiguration of the co-located applications, especially when the count of applications increases beyond 3. The coarse-grained brute force search gives results that are close to that of the fine-grained approach, but significantly faster (only 1–2 seconds are required). Despite the observed small duration of the coarse-grained brute force search **BF-50**, it scales poorly for environments with a high number of co-located applications (e.g. 50-100) and higher numbers of resource limits configurations.

Methods that are based on *continuous optimization* using trust regions show similar execution time but significantly different quality of results. Adding subsequent coarse-grained brute force search on a limited neighborhood improves quality (the distance from the baseline decreased more than 2-fold); the main contributor to the execution time of 1–1.5 minutes for these methods were the multiple runs of the optimization for random values of the initial parameter vector. The execution time might be decreased via a heuristic that estimates the initial parameters vector or decreases its generation time.

VII. EVALUATION

A. Validation Test Design & Settings

The validation of our approach includes two steps: First, a **Preliminary Validation Test (PVT)** to acquire the values of SLIs used as features in application-wise prediction models—in a production environment these values should be collected dynamically and applied in the scope of a continuous resource allocation process. The test uses fixed request rates and an initial guess for resource allocation; a load test is repeated 16 times with **ab** as described in Section III-C. Second, an **Evaluation Validation Test (EVT)** to conduct the real testing similar to **PVT** but with the following major changes: 1) SLIs prediction models are acquired by applying the application-wise lasso regression on the same concurrencies as in **PVT** with medians of SLIs values received as a result of **PVT**; 2) continuous constrained optimization with limited brute force (**Hyb**) is applied to the acquired models and the desired SLOs.

A single validation test according to the description above was conducted to validate our approach. The test was conducted both for hard and soft limits on our Kubernetes cluster. The settings for **PVT** are given in Table XII. The settings for **EVT** are shown in Table XIII (partially acquired as the results of **PVT**). The same SLOs were set for all applications during **EVT**: $RT_i^{(SLO)} = 800$ ms and $T_i^{(SLO)} = 200$ RPS $\forall\, i \in \{1, 2, 3\}$. These SLOs were selected as realistic values based on the collected dataset.

B. Validation Results

The conducted validation test showed the overall validity of the approach proposed in the paper with at most two cases out of 16 trials violating SLOs as shown in Table XIV (marked as N_v in table). Almost all the SLO violation cases (9 out of 10) are attributed to the SLO on the 99%-tile response time, which is more unstable than the throughput. Among these violations, 4 cases were identified for the second application **Liberty** data that contained anomalies due to garbage collection invoked periodically, which is also pointed out by related work [21], [22]. For the selected test environment, soft and hard limits on Kubernetes pods' resources do not seem to significantly influence the amount of SLO violations.

The conducted validation test demonstrated the validity of the designed approach to SLO-compliant resource allocation. The presence of a few SLO violations points out the necessity to include additional features in the SLI prediction models that should capture unique properties of particular applications and their runtimes, such as garbage collection delays.

VIII. CONCLUSION AND FUTURE WORK

The presented approach to the SLO-compliant resource allocation problem for co-located containerized applications demonstrated its validity both for soft and hard limits on CPU resources allocated to Kubernetes pods with at most **2** SLO violations out of **16** trials for a highly-volatile 99%-tile response time SLI. This approach includes four major steps: 1) collecting SLI values for various resource limits and workload rates; 2) removing anomalies that cannot be explained through available features via clustering; 3) learning prediction models relating SLIs to parameters of workload and resource limits via Lasso regression; and 4) deriving the resource limits for applications deployment via continuous optimization and limited brute force search for known SLOs.

Aside from the proposed approach, the paper also offers an approach to select the model features and parameters thereof in order to increase the accuracy of the SLI prediction model. It was also demonstrated that the application-wise SLIs prediction model is the best for the case of co-located applications due to a good balance between its prediction properties (R^2 score) and its scalability to maintain the service availability for a higher number of applications and SLIs. From the machine-learning and MAPE-K planning point of view, one example finding of our work is that lasso regression-based models of degree 2 seem to suffice for predicting SLIs.

Limit	*Concurrencies*			*Resource Limits*			*99%-tile Response Time*			*Throughput*		
Type	*App1*	*App2*	*App3*	*App1*	*App2*	*App3*	*App1*	*App2*	*App3*	*App1*	*App2*	*App3*
Soft	58	61	53	1250	500	130	414.0	466.0	421.0	266.9	254.3	241.2
Hard	58	63	75	200	550	120	562.0	627.0	571.5	231.1	227.6	292.9

TABLE XIII
SETTINGS FOR THE EVALUATION VALIDATION TEST; RESOURCE LIMITS ARE RESULTS OF PROPOSED APPROACH

99%-tile Response Time					
App1		*App2*		*App3*	
Value	N_v	*Value*	N_v	*Value*	N_v
With Soft Limits					
524.9 ± 142.7	1	568.8 ± 130.2	**2**	537.9 ± 149.6	**2**
With Hard Limits					
507.9 ± 108.4	1	705.2 ± 592.8	**2**	294.5 ± 109.3	1
Throughput					
App1		*App2*		*App3*	
Value	N_v	*Value*	N_v	*Value*	N_v
With Soft Limits					
264.6 ± 5.4	0	252.2 ± 4.1	0	237.9 ± 4.6	0
With Hard Limits					
231.9 ± 5.3	0	225.1 ± 9.9	1	294.5 ± 7.2	0

TABLE XIV
RESULTS OF THE VALIDATION TEST

A limitation of the current study is its simplistic dataset. However, the focus of this paper, was to come up with a suitable self-adaptive technique and establish a methodology for testing it: the next step is to repeat with a larger and more realistic application dataset. Other future work includes: 1) evaluation of the impact of runtime/technology-specific behaviors like garbage collection on SLIs and search for predictors for such behaviors; 2) evaluation of artificial neural networks for predicting SLIs due to cheap forward pass; 3) SLO-compliant resource allocation for individual microservices of compound applications; 4) derivation of models for allocation of other resources like RAM; 5) joint resource allocation for pods and virtual machines; and 6) investigation of anomaly detection techniques to improve outlier identification.

REFERENCES

[1] P. Patros, S. A. MacKay, K. B. Kent, and M. Dawson, "Investigating resource interference and scaling on multitenant paas clouds," in *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 2016, pp. 166–177.

[2] D. Weyns, "Software engineering of self-adaptive systems: an organised tour and future challenges," *Chapter in Handbook of Software Engineering*, 2017.

[3] A. Mehta and E. Elmroth, "Distributed cost-optimized placement for latency-critical applications in heterogeneous environments," in *2018 IEEE International Conference on Autonomic Computing (ICAC)*, Sep. 2018, pp. 121–130.

[4] N. Rameshan, Y. Liu, L. Navarro, and V. Vlassov, "Augmenting elasticity controllers for improved accuracy," in *2016 IEEE International Conference on Autonomic Computing (ICAC)*, July 2016, pp. 117–126.

[5] L. Toms, E. B. Lakew, and E. Elmroth, "Service level and performance aware dynamic resource allocation in overbooked data centers," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 42–51.

[6] M. Sedaghat, F. Hernndez-Rodrguez, and E. Elmroth, "Autonomic resource allocation for cloud data centers: A peer to peer approach," in *2014 International Conference on Cloud and Autonomic Computing*, Sep. 2014, pp. 131–140.

[7] E. B. Lakew, A. V. Papadopoulos, M. Maggio, C. Klein, and E. Elmroth, "Kpi-agnostic control for fine-grained vertical elasticity," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 589–598.

[8] E. B. Lakew, R. Birke, J. F. Perez, E. Elmroth, and L. Y. Chen, "Small-tail: Scaling cores and probabilistic cloning requests for web systems," in *2018 IEEE International Conference on Autonomic Computing (ICAC)*, Sep. 2018, pp. 31–40.

[9] R. Birke, J. F. Prez, Z. Qiu, M. Bjrkqvist, and L. Y. Chen, "Power of redundancy: Designing partial replication for multi-tier applications," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, May 2017, pp. 1–9.

[10] P. Patros, K. B. Kent, and M. Dawson, "Slo request modeling, reordering and scaling," in *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 2017, pp. 180–191.

[11] J. Friedman, T. Hastie, and R. Tibshirani, "Regularization paths for generalized linear models via coordinate descent," *Journal of Statistical Software, Articles*, vol. 33, no. 1, pp. 1–22, 2010. [Online]. Available: https://www.jstatsoft.org/v033/i01

[12] S.-J. Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky, "An interior-point method for large-scale l_1-regularized least squares," *IEEE journal of selected topics in signal processing*, vol. 1, no. 4, pp. 606–617, 2007.

[13] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[14] V. S. Sheng, F. Provost, and P. G. Ipeirotis, "Get another label? improving data quality and data mining using multiple, noisy labelers," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2008, pp. 614–622.

[15] F. T. Liu, K. M. Ting, and Z. Zhou, "Isolation forest," in *2008 Eighth IEEE International Conference on Data Mining*, Dec 2008, pp. 413–422.

[16] T. K. Moon, "The Expectation-Maximization Algorithm," *IEEE Signal Processing Magazine*, vol. 13, no. 6, pp. 47–60, 1996.

[17] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: ACM, 2019.

[18] R. Hemmecke, M. Köppe, J. Lee, and R. Weismantel, *Nonlinear Integer Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 561–618. [Online]. Available: https://doi.org/10.1007/978-3-540-68279-0_15

[19] R. Byrd, R. Schnabel, and G. Shultz, "A trust region algorithm for nonlinearly constrained optimization," *SIAM Journal on Numerical Analysis*, vol. 24, no. 5, pp. 1152–1170, 1987. [Online]. Available: https://doi.org/10.1137/0724076

[20] D. J. Wales and J. P. K. Doye, "Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms," *The Journal of Physical Chemistry A*, vol. 101, no. 28, pp. 5111–5116, 1997. [Online]. Available: https://doi.org/10.1021/jp970984n

[21] P. Patros, K. B. Kent, and M. Dawson, "Investigating the effect of garbage collection on service level objectives of clouds," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2017, pp. 633–634.

[22] ——, "Why is garbage collection causing my service level objectives to fail?" *International Journal of Cloud Computing*, vol. 7, no. 3-4, pp. 282–322, 2018.

ACKNOWLEDGEMENT

We thank the NZ Ministry of Business, Innovation and Employment (**MBIE**) and Security Technologies Returning Accountability, Trust and User-centric Services in the Cloud (**STRATUS**) for providing funding, **Clint Dilks** for setting up the hardware and our **Anonymous Reviewers** for their useful comments. This work was conducted at the Oceania Researchers in Cloud and Adaptive-systems (**ORCA**—*Ohu Rangahau Kapua Aunoa*) lab at the University of Waikato.