Monitorless: Predicting Performance Degradation in Cloud Applications with Machine Learning

Johannes Grohmann
johannes.grohmann@uni-wuerzburg.de
University of Würzburg
Würzburg, Germany

Patrick K. Nicholson
pat.nicholson@nokia-bell-labs.com
Nokia Bell Labs
Dublin, Ireland

Jesus Omana Iglesias
jesus.omana@telefonica.com
Telefonica Alpha
Barcelona, Spain

Samuel Kounev
samuel.kounev@uni-wuerzburg.de
University of Würzburg
Würzburg, Germany

Diego Lugones
diego.lugones@nokia-bell-labs.com
Nokia Bell Labs
Dublin, Ireland

Abstract

Today, software operation engineers rely on application key performance indicators (KPIs) for sizing and orchestrating cloud resources dynamically. KPIs are monitored to assess the achievable performance and to configure various cloud-specific parameters such as flavors of instances and autoscaling rules, among others. Usually, keeping KPIs within acceptable levels requires application expertise which is expensive and can slow down the continuous delivery of software. Expertise is required because KPIs are normally based on application-specific quality-of-service metrics, like service response time and processing rate, instead of generic platform metrics, like those typical across various environments (e.g., CPU and memory utilization, I/O rate, etc.)

In this paper, we investigate the feasibility of outsourcing the management of application performance from developers to cloud operators. In the same way that the *serverless* paradigm allows the execution environment to be fully managed by a third party, we discuss a *monitorless* model to streamline application deployment by delegating performance management. We show that training a machine learning model with platform-level data, collected from the execution of representative containerized services, allows inferring application KPI degradation. This is an opportunity to simplify operations as engineers can rely solely on platform metrics – while still fulfilling application KPIs – to configure portable and application agnostic rules and other cloud-specific parameters to automatically trigger actions such as autoscaling, instance migration, network slicing, etc.

Results show that *monitorless* infers KPI degradation with an accuracy of 97% and, notably, it performs similarly to typical autoscaling solutions, even when autoscaling rules are optimally tuned with knowledge of the expected workload.

CCS Concepts • **Computer systems organization → Cloud computing**; **Self-organizing autonomic computing**; • **Computing methodologies →** *Supervised learning by classification.*

Keywords Cloud computing, Machine learning, Monitoring, DevOps.

ACM Reference Format:
Johannes Grohmann, Patrick K. Nicholson, Jesus Omana Iglesias, Samuel Kounev, and Diego Lugones. 2019. *Monitorless*: Predicting Performance Degradation in Cloud Applications with Machine Learning. In *20th International Middleware Conference (Middleware '19), December 8–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3361525.3361543

Middleware '19, December 8–13, 2019, Davis, CA, USA

ACM ISBN 978-1-4503-7009-7/19/12…$15.00
https://doi.org/10.1145/3361525.3361543

1 Introduction

Bringing new applications or product features to production requires time and expertise to properly manage the tradeoff between agility and reliability or product stability. One reason for delays is the extensive testing required to assess performance; for example, telecommunication operators can spend up to 40% of their time on testing activities [37]. Performance behavior is tightly related to the amount of resources allocated to applications [43]. Therefore, as software development progressively adopts loosely coupled architectures (e.g., microservices) to speedup continuous delivery and integration, operation engineers struggle to cope with the pace at which new features can be integrated into production [31]. Operation requires significant testing, which involves collecting metrics as well as analyzing and correlating resource usage with application *key performance indicators* (KPIs) to understand performance bottlenecks, scalability issues and degradation of service quality.

However, as the scale of cloud environments continues to grow and applications are built with tens or hundreds of microservices [4], the number of metrics increases drastically [18, 35, 51, 57], which makes the process of gaining performance insights very complex. To cope with such complexity, operation engineers need to design and architect frameworks to *automate* or even substitute the testing pipelines and streamline KPI control to fulfill service level objectives (SLOs) [3].

Recent research work relies on specific application KPIs for automating performance analysis and orchestration [38, 63, 65, 71]. We argue that application-specific metrics limit the generality of solutions and their applicability across applications and platforms. That is, operation teams need to proficiently re-evaluate and control critical KPIs and keep track of their specific target operating range for each running application, which reduces the agility of deployment.

In this paper, we leverage machine learning to loosen the dependencies between operation tasks and application-specific KPIs.

Intuitively, the idea is to use historical data from various typical services, labeled with information about bottlenecked resources and hardware configuration, to infer service degradation without the need to specifically monitor or analyze any KPIs during production. Thus, our approach uses only a standard set of application-agnostic, hypervisor-level, platform metrics (e.g., CPU and memory utilization) to infer KPI degradation. Analogous to the *serverless* paradigm, in which servers are hidden to end-users, our approach still has metrics, but only generic platform metrics, instead of application-specific ones. These metrics are autonomously processed by a machine learning algorithm to predict performance issues and accelerate the pace at which performance insights are obtained in production. We therefore call our approach "*monitorless*".

The reasoning behind *monitorless* is motivated by: 1) new software architectural patterns such as *microservices* that allow building applications from a collection of loosely coupled and fine-grained services; 2) cloud commodity hardware that allows for a more concrete and homogeneous set of platform-level metrics; and 3) the fact that application KPIs are directly related to the usage of underlying platform resources (physical or virtual), and finding this relation is a realistic machine learning task. In our proof-of-concept, *monitorless* relies on a binary classifier that works with a large set of features derived from combinations of platform-level metrics. The classifier is trained with data obtained during the execution of typical microservices, widely used by application developers (e.g., databases, load balancers, messaging, etc.), monitored as they experience resource saturation as well as in normal operation.

This paper introduces the *monitorless* framework and shows that *one* resource saturation model allows for detecting performance degradation of several complex applications, even when such applications are unknown to the trained model. We note this represents a significant divergence between *monitorless* and other solutions based on KPIs as we propose a generic approach with a single resource saturation model working for a heterogeneous set of different applications. Moreover, *monitorless* can be used as a building block that enables black-box services and applications deployed on a cloud platform to be autonomously managed. In particular, since our framework can be used to detect performance degradation in a service-agnostic way based on generic platform-level metrics, it can be used as a basis for autoscaling and consolidation decisions, as well as performance bottleneck analysis. Thus, we posit that in many cases *monitorless* can eliminate the need for (i) dedicated tests to analyze performance behavior and, (ii) specialized online service-level monitoring of KPIs.

In summary, the key paper contributions are the following:

- A methodology for creating the appropriate feature set for the robust operation of cloud environments driven by generic platform data. Features are selected according to the *USE* (Utilization, Saturation, Error) method [32] and extracted with the Performance Co-Pilot [10] tool.
- A pipelined architecture for model training and validation, as well as the components required to infer performance degradation without monitoring application KPIs, and to integrate the *monitorless* model into the cloud orchestrator.

The rest of the paper is organized as follows. Section 2 presents an overview of the *monitorless* architecture and design, whereas Section 3 elaborates on the feature selection and methodology used to create a robust machine learning model. Results are presented in Section 4. We discuss the limitations of *monitorless*, and the research needed to address them, in Section 5. In Section 6 we provide a taxonomy of related work and, finally, we conclude in Section 7.

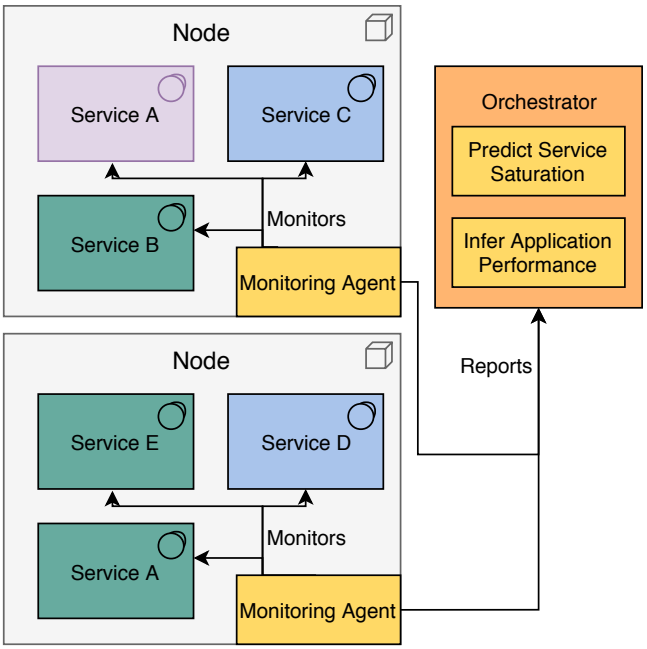

Figure 1. Overview of the *monitorless* components on a two-nodes platform with three applications composed of several microservices.

2 *Monitorless* Design

Figure 1 illustrates the components of *monitorless* in a simple deployment with two physical nodes running three applications (indicated with different colors), each composed of one or many instances of different microservices. *Monitorless* introduces two key components: a monitoring agent deployed on each cloud node, and the orchestrator function serving as a centralized repository for data collection and training. Both components can be integrated into an existing cloud environment to augment its functionality.

The monitoring agents run on each physical host to collect a set of predefined platform metrics, using standard monitoring tools. Examples of these metrics include CPU usage, RAM usage, and throughput of I/O-devices; all of them are measured at operating system level and include hypervisor information, e.g., namespace and cgroups interfaces for Linux containers. The orchestrator periodically receives metrics from the agents. First, the data collected is used to make performance prediction at each container to detect resource saturation; see section 3 for more details on the prediction model. Second, the orchestrator infers the overall application performance from the predictions at individual containers composing the application. Third, based on the inference, the orchestrator can decide to adapt the current deployment by migrating or scaling applications automatically as a remediation for performance problems. The orchestrator operates online, repeating the steps above at regular intervals.

Next, we describe the nomenclature used throughout the paper (§2.1), the methodology proposed for detecting saturation (§2.2), and

a formal definition of the machine learning problem *monitorless* is designed to solve (§2.3).

2.1 Nomenclature and definitions

A *cloud* is a set of connected nodes $C = \{c_1, \ldots, c_{|C|}\}$ in which multiple applications are executed, and where each *node* $c \in C$ is a computing entity able to host service instances in a virtualized environment (as VMs or Linux containers). An *application* $\mathcal{A} = \{\mathcal{S}_1, \ldots, \mathcal{S}_{|\mathcal{A}|}\}$ is composed of interconnected services running in the cloud, where each *service* $\mathcal{S} = \{\mathcal{I}_1, \ldots, \mathcal{I}_{|\mathcal{S}|}\}$ consists of one or more service instances. These *service instances* are functionally identical but can differ in terms of which node they are assigned to, the resources at that node, and the workload sent to the instance. Thus, each service instance $\mathcal{I}$ is assigned to run inside its own virtual environment on exactly one node c of the cloud environment.

At each point in time t, the monitoring agents collect a set of $k_{\mathcal{H}}$ *host metrics*, $\mathcal{H}_{c,t}$, as well as a set of $k_{\mathcal{V}}$ *virtual metrics*, $\mathcal{V}_{\mathcal{I},t}$, for each service instance $\mathcal{I}$ running on node c. Moreover, at each point in time t, it is possible to observe a Key Performance Indicator (KPI) value $\mathcal{P}_{\mathcal{A}}(t)$, which represents the performance of application $\mathcal{A}$ at time t. Each service instance $\mathcal{I}$ is associated with one set of host metrics $\mathcal{H}_{c,t} \in \mathbb{R}^{k_{\mathcal{H}}}$ as well as one set of virtual metrics $\mathcal{V}_{\mathcal{I},t} \in \mathbb{R}^{k_{\mathcal{V}}}$ for a given time t (of course assuming that $\mathcal{I}$ was running at time t). We denote the vector representing the concatenation of metrics $\mathcal{H}_{c,t}$ and $\mathcal{V}_{\mathcal{I},t}$, as $\mathcal{M}_{\mathcal{I},t}$.

2.2 Labeling resource saturation

We consider *service* instances (i.e., microservices composing the application) to be either *saturated* or *non-saturated* at a given time, and define the saturation state of an *application* $\mathcal{A}$ at time t according to its $\mathcal{P}_{\mathcal{A}}(t)$. We use application KPIs only for labeling data to *train* our models – note, though, that such KPIs are not required for *using* the resulting model. Depending on the application, examples of KPIs can be: response time or throughput of a web service, jitter in a video streaming application, or availability indicators in communication systems. In the following, we present a methodology for labeling throughput. However, this step can be analogously applied to similar KPIs; for other more sophisticated KPIs, manual modeling might be necessary. Note that this labeling is only required during training of different applications and implies the abstraction from the KPIs of the used application. Therefore, this abstraction of different KPIs enables *monitorless* to act application-agnostically. The actual training is done using labeled (and therefore KPI-indepedent) historical data from a set of selected representative applications without assuming that the target application is also used for training. Although for common use-cases (e.g., autoscaling) a binary classification is enough, note that one can also apply more complex state descriptions based on multiple classes.

In practice, finding resource saturation is difficult as metrics can have non-deterministic or noisy behavior – see *observed throughput* (blue dots) in Figure 2 – or important points in the dataset can be missing or not sampled frequently enough. Usually, an application serving increasing workloads will show a proportional increase in throughput until a saturation point is reached, from which a non-linear behavior is expected – see the *elbow/knee* around 700 requests/sec in the *smoothed curve* (orange line) of Figure 2. This behavior can be correlated to other KPIs; for example, the *response time* would rapidly increase when the throughput reaches the non-linear region; or there would be an increase in the number of *dropped requests*.

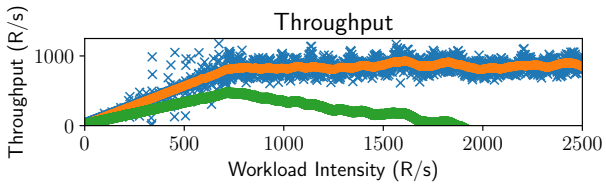

Figure 2. Observed throughput (blue), smoothed curve (orange) and differences $\beta_i - \alpha_i$ (green) of an example run.

To label the dataset properly, it is critical to ensure that the non-linear behavior of KPIs is due to resource saturation when detecting elbows or knees in KPIs. Although there is no mathematically unique "elbow" in an exponential curve, we use Satopaa's et al. [59] definition based on curvature and employ their *Kneedle* approach to find it.

Therefore, to create training and test data, we linearly increase the workload of a given target application $\mathcal{A}$. While applying the workload, we monitor the KPI $\mathcal{P}_{\mathcal{A}}(t)$. Relating this KPI to the workload intensity for time step i defines a discrete function f with $f(\alpha_i) = \beta_i$, where α_i is the workload intensity and β_i the corresponding KPI. Our implementation of Kneedle for the purpose of labeling the training and test data is outlined below:

1. Smooth f by applying a Savitzky-Golay filter [60] (see orange curve in Figure 2), or any similar filter.
2. We normalize the points of f to the unit square by setting: $\alpha_i \leftarrow (\alpha_i - \min_j\{\alpha_j\})/(\max_j\{\alpha_j\} - \min_j\{\alpha_j\})$, and $\beta_i \leftarrow (\beta_i - \min_j\{\beta_j\})/(\max_j\{\beta_j\} - \min_j\{\beta_j\})$.
3. We calculate the differences between β_i and α_i by setting: $\beta_i \leftarrow \beta_i - \alpha_i$. This yields the green curve in Figure 2.
4. The set of candidate saturation points are local maxima of this curve defined by the differences above, and we manually choose the local maximum (such as in Figure 2), defining the corresponding β_i as our threshold Υ.

Hence, we obtain a function $\tilde{\mathcal{P}}_{\mathcal{A}} : \mathbb{R} \rightarrow \{0, 1\}$, with

$$\tilde{\mathcal{P}}_{\mathcal{A}}(t) = \begin{cases} 0 & \text{if } \mathcal{P}_{\mathcal{A}}(t) \leq \Upsilon \quad (no\ saturation) \\ 1 & otherwise \quad\quad (saturation) \end{cases}$$

The function f above is assumed to have positive concavity. If the opposite is true, the same technique can be applied by setting $\beta_i \leftarrow \max_j(\beta_j) - \beta_i$ and $\alpha_i \leftarrow \max_j(\alpha_j) - \alpha_i$. We note that Savitzky-Golay filters have tunable parameters that should be adjusted depending on input data. However, its purpose is to provide a smoothed curve from which we can identify a knee or elbow in a reproducible manner, as we are applying machine learning algorithms to the function f. Thus, the procedure above need not be fully automated for arbitrary input data, and, indeed, we recommend that f is visually inspected as a sanity check.

2.3 A machine learning problem

The training dataset $\mathcal{T}$ is built from multiple instances running various services on different deployments and serving diverse workloads to increase the model robustness. We define the machine learning problem as follows. $\mathcal{T}$ is partitioned in many subsets $\mathcal{T}_{\mathcal{I}}$ for each

instance $\mathcal{I}$ from which we generated labeled data. Each $\mathcal{T}_{\mathcal{I}}$ is a set of pairs $\mathcal{T}_{\mathcal{I},t} = \{(\mathcal{M}_{\mathcal{I},t}, \tilde{\mathcal{P}}_{\mathcal{A},t})\}$. The vector $\mathcal{M}_{\mathcal{I},t}$ represents the system state of service instance $\mathcal{I}$ at time t, and $\mathcal{H}_{c,t} \subset \mathcal{M}_{\mathcal{I},t}$ are the platform metrics obtained from each host. Instead, $\mathcal{V}_{\mathcal{I},t} \subset \mathcal{M}_{\mathcal{I},t}$ are metrics specific to the service running in the instance $\mathcal{I}$. In case of a Linux container, this could be the CPU-time relative to the allocated maximum. Thus, multiple containers running on machine c at time t share the same feature values for $\mathcal{H}_{c,t}$ but have different values for $\mathcal{V}_{\mathcal{I},t}$. Note that this architecture enables the model to handle any variable number of service instances on different hosts.

Then, solving the machine learning problem consists of: i) processing $\mathcal{M}_{\mathcal{I},t}$ first to extract feature vectors $x_{\mathcal{I},t}$ and ii) training a binary classifier mapping $x_{\mathcal{I},t}$ to $y_{\mathcal{I},t} = \tilde{\mathcal{P}}_{\mathcal{A},t}$.

3 Modeling Process

This section provides implementation details of the prototyped *monitorless* binary classifier.

3.1 Metric collection

The *USE* method [32] allows for detecting performance bottlenecks by examining the **U**tilization, **S**aturation and the **E**rrors of each relevant platform resource. We include as much of the USE metrics as possible to detect resource bottlenecks without incurring monitoring overhead.

For collecting such metrics, we use the Performance Co-Pilot (PCP) monitoring tool [10], which provides a wide set of platform metrics, while being lightweight. PCP gathers usage and saturation indicators of CPUs (incl. vCPUs), memory, network and storage devices, controllers and buses, as well as interrupts and network errors. As a preprocessing step, metrics reporting counters must be converted into rates, and utilization metrics to a relative scale (i.e., percentage value). These steps are necessary to avoid overfitting our model to a particular hardware configuration, or system state (see Section 3.3 for more details). Measurements are produced every second, a default interval of PCP [10] and the Docker stats command [15]. This is a reasonable sampling time as it enables the algorithms to react quickly and start/stop Linux containers that usually instantiate in a few seconds.

3.2 Training data

The idea of the *monitorless* model is to classify service performance without any prior knowledge. For this purpose, it is necessary to create training data using services with different resource utilization patterns, performance demands, diverse usage of resource (e.g., *CPU-bound*) and intensity of traffic. In this work, we created training data using a small set of services widely used by application developers and discuss how to extend it in section 5.

3.2.1 Services and applications used for training

We used three different applications for training: Solr [48], Memcache [39], and Cassandra [21]. We selected these applications for two reasons: i) they are representative services in Cloudsuite [53], and; ii) they show different resource usage profiles and therefore different resource bottlenecks. To improve the reproducibility of our results, we used the Cloudsuite benchmarking tools to run these services [20, 53].

Apache Solr is an enterprise search platform. We use an index size of 12GB of content crawled from the Internet that comes with Cloudsuite, and the HTTPLoadGenerator [66] to generate HTTP varying workloads as specified by LIMBO [68]. The individual response times and failed request rates are logged every second to label the training data.

During the load generation, clients send search requests with a range between one and five terms. Each term is randomly selected from the top 10,000 most frequent words in the index. The server outputs a list of the top-10 relevant documents according to these terms. Our hardware configuration allows the index to fit into main memory, thus eliminating page faults and minimizing disk activity [40]. With this configuration, the benchmark is mostly CPU-bound. In addition, however, we also conduct experiments with different configurations in which the container resources were limited in the server to alter such CPU-bound behavior.

Memcache is a distributed memory object system, usually used to alleviate database loads via caching. Again, we used the load generator provided by Cloudsuite that uses a 10GB Twitter dataset to populate the cache. It then applies a constant target throughput with a configurable get/set rate parameter. Memcache is configured to be memory-bound. Thus, in our setup, we constrained CPU resources in the containers just for one dataset, and changed the memory configuration to either 8GB, 4GB, or unlimited.

Apache Cassandra is a large-scale NoSQL database system. We generate database workloads with the *Yahoo! Cloud Serving Benchmark* (YCSB) [13]. The database is populated with 30 million records, consuming about 30GB (plus additional indexing and log files). Several constant target loads are applied by changing the number of client threads and/or the required target throughput. As Cassandra stresses several resources, we can tune it to be either CPU or disk-bound. One setting with 20 cores and 30GB RAM limit creates an I/O bottleneck, while a setting with 6 cores and unlimited memory creates a CPU-bottleneck.

3.2.2 Generated datasets

Each application is monitored under different workloads and resource constraints. They run both in isolation, as the only instance running on the host, as well as in combination with other instances. The purpose of the latter is to create a model robust to interference caused by resource sharing in the physical host. Table 1 lists all the configurations used for training. All experiments were conducted on HP ProLiant DL380 Gen9 servers provisioned with a 48 Core Intel® Xeon® CPU E5-2680 v3 @ 2.50GHz processor and 125G of memory, connected by a 10Gbps switch and running CentOS 7.3. with Docker 17.06.1-ce and PCP 3.12.1.

The master orchestrator collects throughput, response times, and platform metrics generated throughout the execution of the specified workloads. An additional experiment with linear increasing load is conducted in order to determine the threshold value Υ, defined in Section 2.2. After acquiring this threshold, the labeling (saturated or not saturated) of samples is performed as described in Section 2.3.

If a test runs in parallel with other tests to create interference, we indicate this in the *Par* column, e.g., Solr (test 3) was run in parallel with Cassandra (test 18). For Memcache, we show the minimum and maximum request rate used. Cassandra workloads can be divided into four different classes: A, B, D, and F. These correspond to core workloads available in YCSB. A is an update-heavy workload (Read/Write: 0.5/0.5), B is read-heavy (Read/Write: 0.95/0.05), D is constantly inserting records and reading the most recent, and F

Table 1. List of datasets for the benchmarked *services*, CPU and Memory limits (*CPU/MEM*, a dash "–" indicates no container limitation), whether they ran in parallel (*Par*), the traffic pattern (*Traffic*) and the resource *bottleneck*.

#	Service	CPU, MEM	Par	Traffic	Bottleneck
1	Solr	3/–	–	sin1000	Container-CPU
2	Solr	–/–	–	sin1000	Host-CPU
3	Solr	–/8 GB	18	sinnoise1000	IO-Bandwidth
4	Solr	–/8 GB	19	sinnoise1000	IO-Bandwidth
5	Solr	3/8 GB	20	sinnoise1000	IO-Bandwidth
6	Solr	1.5/8 GB	22	sinnoise1000	Container-CPU
7	Memc.	–/–	–	2K-50K R/s	Mem-Bandwidth
8	Memc.	1/–	–	20K-85K R/s	Container-CPU
9	Memc.	–/8 GB	–	39K-45K R/s	IO-Queue
10	Memc.	–/4 GB	23	10K-65K R/s	IO-Queue
11	Cass.	–/–	–	A: 30K-100K R/s	Network-Util.
12	Cass.	–/–	–	B: 20K-70K R/s	Host-CPU
13	Cass.	–/–	–	D: 40K-90K R/s	Network-Util.
14	Cass.	20/30 GB	–	A: 300-1200 R/s	IO-Bandwidth
15	Cass.	20/30 GB	–	B: 100-900 R/s	IO-Bandwidth
16	Cass.	20/30 GB	–	B: 700 -1000 R/s	IO-Bandwidth
17	Cass.	20/30 GB	–	B: 100-1000 R/s	IO-Bandwidth
18	Cass.	6/–	3	A: 15K-25K R/s	Container-CPU
19	Cass.	6/–	4	B: 10K-15K R/s	Container-CPU
20	Cass.	6/–	5	D: 10K-25K R/s	Container-CPU
21	Cass.	6/–	–	A: 5K-20K R/s	Container-CPU
22	Cass.	6/–	6	B: 5K-20K R/s	Container-CPU
23	Cass.	6/–	10	B: 10K R/s	Container-CPU
24	Cass.	1/–	–	F: 200 R/s	IO-Wait
25	Cass.	1/–	–	F: 20 R/s	IO-Wait

reads a record, modifies it and then writes it. For Solr, we have two workload curves generated by LIMBO. The first (sin1000) is a simple sine function with a minimum request rate of 1 and a maximum request rate of 1000 requests per second. The second (sinnoise1000) has the same base structure but was massively modified by adding random noise to increase variability. For the sake of simplicity, we describe the used workload patterns in a very concise matter. However, we configured the different runs such that the training sets capture as many different contention scenario as possible.

Furthermore, we give an indication of the limiting factors and critical metrics in each dataset. If two datasets run in parallel, the goal is to learn also the isolation effects, i.e., how resource sharing impact multiple running applications.

3.2.3 Iterative improvement of the training set

We repeated the tests outlined in Table 1 multiple times, iterating over the various stack configurations, to improve the datasets with more representative use cases stressing most of the resources in the platform. As a result, we devised the following structure to create a robust and accurate model.

1. Normalize our training data and save the normalizing instance – We use the MinMaxScaler in Scikit-Learn [56].
2. Use the normalizing instance to analyze with a validation dataset. This is done by scaling the validation set with the known scaler instance. If any feature has its maximum or its minimum outside the scaling range of the trained scaler, we know that this feature was not sufficiently trained.
3. Analyze the features not covered by the training set and decide if they are critical for the model performance.
4. Design additional training cases in order to include other feature values in the training set and perform the additional measurements.
5. Add the training set to the others and repeat from step 1 in order to validate that the training has now improved.

3.3 Feature selection and optimization

We collect 1040 platform metrics using the PCP monitoring tool as described in Section 3.1. 952 of these platform metrics consider the host, 88 are specific to service instances (i.e., containers) running on the host. As expected, not all the metrics are relevant for the machine learning model and in many cases metric preprocessing is required such that they can be useful or leveraged by the algorithm. Next, we describe the preprocessing performed on these raw metrics.

3.3.1 Binary features

CPU and memory utilization are important indicators of saturation; thus, in order to improve the accuracy of the model, we introduce three additional boolean-valued (i.e., hot-encoded) features for both of these metrics. Namely: *LOW* indicates whether utilization is under 50%; *MED* indicates whether utilization is in the 50-80% range; and *HIGH* indicates whether it is above 80%. For CPU utilization, we also introduced two other boolean features called *VERYHIGH*, indicating utilization above 90%, and *EXTREME* indicating utilization above 95%. These new metrics are inserted both for the host and for the container-specific metrics resulting in a total of 16 additional binary features.

3.3.2 Scaling

All metrics having units in byte-values like KB or MB, and which are not convertible to a relative scale, e.g., the number of bytes read by an I/O-device where the maximum capacity is not known, are transformed to a logarithmic scale. The goal of this transformation is to improve accuracy by emphasizing the magnitude rather than a specific value and thus reduce hardware dependency. Nevertheless, these metrics are still prone to overfit to a specific hardware configuration as they are not on a normalized, and therefore portable, scale and need to be handled carefully.

3.3.3 Normalization

As the maximum values of some features are undefined, we opt to artificially limit the possible values to a range. We used the StandardScaler function of SciKit to transform feature values such that their distribution has a mean value of 0 and a standard deviation of 1. Hence, each monitored value would have the sample mean value subtracted and then divided by the standard deviation of all existing samples.

3.3.4 Filtering with random forest or PCA

We use the random forest algorithm [6] to filter the most relevant metrics, as it allows for a simple computation of the information gain of each feature and ranks them by importance. We trained the random forest on each of the datasets shown in Table 1 and took the union of the top 30 most important features of each dataset – below the top 30, the algorithm assigns features a weight lower than $1/\#Features$. This union set consists of 117 unique features.

We make three observations about this filtering step. The first is that the filtering does not decrease the cross-validation accuracy, which implies that resource saturation is detectable by looking at a small set of platform metrics; this finding is consistent with related work [11]. Second, while there is overlap between most

top-30 feature lists (e.g., CPU utilization), there are also metrics that are specific to one dataset, which encourages the inclusion of many different training applications to stress different platform resources in the future. Third, the binary features for memory were filtered out whereas the binary features for CPU were selected as highly important. The reasons for this can be: (i) the training set has not enough memory saturation samples, (ii) the binary labels are not actually required for training because memory problems are likely to be preceded by other symptoms (e.g., disk usage because of page thrashing), or (iii) the binary labels are not required because memory measurements are more insightful with the real values (i.e., non-binary).

An alternative method for feature selection is the Principle Component Analysis (PCA) [55], which provides a linear transformation to obtain orthogonal features. PCA can reduce the number of features in very large platforms. The side effect is that the transformed features no longer correspond to physical magnitudes, which makes it difficult to fully interpret the resulting model. Using the Scikit-learn implementation of PCA, we reduced the number of features to 50, which accounts for 99.99% of variability in the data.

3.3.5 Time-dependent features

Metrics are collected for a fixed time window of one second. However, time-dependent features can give some insights and reflect certain dynamics of performance, e.g., having a low CPU-utilization during the past 15 seconds and a peak at the present time *might not* actually imply that the resource is saturated. On the contrary, if CPU-utilization is high for the last 15 seconds, then the platform is more likely to be experiencing resource saturation.

Based on this intuition, we create the X-AVG and X-LAG variants of each metric. X-AVG takes the average over the last $X+1$ samples (seconds) including the current one, while X-LAG contains the value of the metric X samples ago. This helps to include context in an otherwise isolated one-second snapshot of the platform. We include X-AVG and X-LAG for each metric with values $X = 1, 5, 15$. A window of 15 seconds proved to be sufficient in our experiments regarding the applications we considered.

3.3.6 Combining features

It is a common practice in data science to create new features by multiplying existing features, as this often improves the models significantly by revealing relationships that are not visible by just analyzing linear combinations of the features separately. In our case, we multiply all pairs of features from different domains (e.g., CPU and memory), but in order to prevent an explosion of the size of the feature set, we omit all time-dependent features from this step. This refinement turned out to be crucial to capture performance problems across features, i.e., detectable by observing combination of metrics (cf. section 3.5 for more details).

3.3.7 Pipeline optimization

To guarantee the right order and configuration, as well as to streamline the execution of the aforementioned steps, we have implemented a pipeline. The pipeline has five steps:

1. Create binary features and scale required features.
2. Normalize the features.
3. Apply a first *reduction* step (either filtering or PCA).
4. Create new time-dependent and multiplicative features.
5. Apply a second *reduction* step (either filtering or PCA).
6. Remove features with 0 variance (provide no information).

We perform a grid search on steps 2 to 5 in order to find the best combination of features. Steps 3 and 5 can either do filtering, PCA, or none of them, while steps 2 and 4 can perform optional normalization and feature addition.

The grid search is performed with the training set described in Section 3.4, applying the same cross-validation scheme. We again use a random forest algorithm with default parameters as a prediction algorithm to evaluate the individual feature engineering steps. We do not include the combination of not applying a first feature selection in Step 3 and then adding the multiplicative features in Step 4, since this is practically unfeasible due to an exponential increase of the resulting features.

3.4 Training the *monitorless* model

Combining all datasets from Section 3.2 and the features described in Section 3.3, we get a total training set of 63086 samples with 4492 features each. The portion of saturated examples in the training set is 26%.

For algorithm training, we choose to compare six different classifiers: (1) binary logistic regression [14] based on SAG [61], (2) Support Vector Classification (SVC) based on LIBLINEAR [19], (3) AdaBoost [25] with decision trees [7], (4) XGBoost [8], (5) a three-layer, fully connected, sequential neural network [9], and (6) random forest [6]. Note that we use a linear kernel for SVC as any other kernel function increased the algorithm training time significantly.

We use the Python implementations of XGBoost [12], Keras [9] with a TensorFlow backend [1] for the neural network, and Scikit-learn for all other algorithms. We perform a 5-fold cross-validation with a grid search to select the hyper-parameters of each algorithm. The 5-fold cross-validation partitions the training sets from Table 1 using 20 sets for training and 5 sets for validation in the fold. By partitioning the training sets in this way, rather than using the union of all training sets, we aim to avoid overfitting. Table 2 lists the parameters considered during the hyper-parameter grid search, with underlined parameters as the chose ones for all considered algorithms. The naming of the parameters follows the convention of Scikit-Learn.

Table 3 lists the training times, the per-sample classification time, and the $F1_2$ score of our first validation set – see Section 4.1 for scoring metric definitions. Note that the training and classification time excludes the feature extraction described in the previous section, since that takes the same time for all algorithms and only accounts for $\sim$ 28ms per prediction on average.

We observe that random forest outperforms all other approaches with an F1 score of 0.99. There is a clear tradeoff between training and prediction time, on the one hand, and prediction accuracy, on the other hand. However, even for the slowest algorithm (random forest), the prediction time was still only 40ms per prediction on average. This is sufficiently fast to make such predictions online in production. Interestingly, we found that Logistic Regression, Neural Net, and AdaBoost only predict the majority label and, hence, a classifier that predicts "saturated" for all samples would receive the same score. While SVC does make some correct predictions, it does poorly overall and achieves an even lower score than simply predicting the majority label. The high F1-score of 0.997 can be

Table 2. Examined parameter space by the grid search for each applied algorithm.

Algorithm	Parameters	Values
Log. Regression	C	0.01, 0.1,1
	tol	0.1,0.01,0.001,0.0001
	class_weight	balanced, None
SVC	C	0.1,1,10
	tol	0.01,0.0001, 0.00001
	penalty	l1, l2
	class_weight	balanced, None
AdaBoost	n_estimators	50,250,500
	algorithm	SAMME, SAMME.R
	DT_criterion	gini, entropy
	DT_splitter	random, best
	DT_min_samples_split	5,10,20
XGBoost	min_child_weight	1,4,16,64
	max_depth	1,4,16,64
	gamma	0,1,4,16
NN	activation_function1	softmax, relu, sigmoid, linear
	activation_function2	softmax, relu, sigmoid, linear
	activation_function3	softmax, relu, sigmoid, linear
Random Forest	n_estimators	250,500,1000
	min_samples_leaf	5,10,20,30
	min_samples_split	5,10,20,30
	criterion	gini, entropy
	class_weight	balanced, subsample, None

Table 3. Performance of the applied algorithms.

Algorithm	Training Time	Class. Time	$F1_2$
SVC	837.8 s	0.2 ms	0.579
Logistic Regression	3.1 s	0.2 ms	0.858
AdaBoost	250.7 s	0.4 ms	0.858
Neural Net	76154.9 s	4.1 ms	0.858
XGBoost	5140.3 s	6.3 ms	0.944
Random Forest	68.3 s	40.6 ms	0.997

attributed to the amount of training samples including many clearly saturated or non-saturated examples.

Based on these results, we select the model created by random forest for the evaluation of *monitorless* in the next section. The hyper-parameter tuning of random forest resulted in 250 trees, trained with 20 samples in a leaf node using the information gain as splitting criterion and applying no weights to the different classes.

3.5 Model behavior compared to expert decisions

Here we analyze the trained model with focus on the filtered features to provide insights from a system perspective and compare the outcome to expert-made decisions. For this, we use the feature importances as given by the trained random forest model. Table 4 shows the 30 most important features, based on the trained random forest model.

First, we can observe that the combination of features is a non-trivial step as almost all used features are multiplications of two original metrics. Most of them are the multiplication of CPU-level metrics with a metric of another resource, e.g., CPU-HIGH multiplied with various network or memory related features. Different metrics like memory utilization are also included, but with a lower ranking and are therefore not contained in the table. Intuitively, this means that service saturation is captured analyzing more than one resource type at the same time. Additionally, lagged and average features are occasionally used. In fact, raw (un-engineered) metrics

Table 4. Top 30 features sorted by importances assigned by random forest. The ×-symbol denotes a multiplication of two features. The features S-MEM-U, C-CPU-MEDIUM, C-CPU-HIGH, C-CPU-VERYHIGH are derived relative utilizations, the suffix HIGH and VERYHIGH are binary features, an AVG-k or LAGGED-k denotes that the feature was averaged or lagged by k seconds. All other parameters follow the nomenclature of the monitoring tool (PCP).

Feature name
network.tcp.currestab × C-CPU-HIGH
hinv.ninterface × C-CPU-VERYHIGH
kernel.all.pswitch-AVG14
mem.vmstat.nr_inactive_anon × C-CPU-VERYHIGH
network.tcp.currestab × C-CPU-VERYHIGH
network.tcpconn.established × C-CPU-HIGH
C-CPU-HIGH
network.sockstat.tcp.inuse × C-CPU-VERYHIGH-AVG14
C-CPU-VERYHIGH × C-CPU-VERYHIGH
network.sockstat.tcp.inuse × C-CPU-VERYHIGH
cgroup.cpusched.periods × C-CPU-HIGH
C-CPU-VERYHIGH
C-CPU-SUPERHIGH-AVG14
C-CPU-HIGH-AVG4
mem.vmstat.nr_kernel_stack × C-CPU-VERYHIGH
cgroup.cpusched.throttled × C-CPU-VERYHIGH
kernel.all.nprocs × C-CPU-HIGH
hinv.ninterface × C-CPU-MEDIUM
C-CPU-SUPERHIGH-LAGGED15
S-MEM-U-mapped × C-CPU-VERYHIGH
C-MEM-U-usage × C-CPU-HIGH
cgroup.cpusched.throttled × C-CPU-HIGH
C-CPU-VERYHIGH-AVG4
C-CPU-HIGH × C-CPU-VERYHIGH
vfs.inodes.free × C-CPU-VERYHIGH
mem.vmstat.pgpgin × C-CPU-HIGH
mem.vmstat.nr_inactive_file × C-CPU-VERYHIGH
vfs.inodes.free × C-CPU-HIGH
disk.all.aveq-AVG4
S-MEM-U-active_file × C-CPU-VERYHIGH

are rather low-rated and are used seldomly. However, the chosen features heavily depend on the CPU-utilizations (in various levels), the number of network connections, the disk queue, the memory utilization, the number of throttled cgroup periods, etc, all of which intuitively make sense to a system engineer. Hence, we conclude that the system itself does not behave very differently from any human performance engineer with respect to the set of analyzed metrics. Nonetheless, there are also some interesting combinations chosen by the system which are surprising and not directly obvious. Furthermore, it is impossible for any human to monitor all of the considered metrics in parallel.

4 Evaluation

We evaluate *monitorless* with three applications that are not included in the training phase: a three-tier web service [53] and two

microservice-based e-commerce applications, (i) *TeaStore* [67] composed of seven microservices, and (ii) *Sockshop* [69] composed of sixteen microservices.

To measure the accuracy of *monitorless*, it is necessary to determine whether a predicted label is correct. The threshold for determining saturation of the overall application is discovered by running a linearly increasing load test, as described in Section 2.2, which yields a set of ground truth labels $y_{\mathcal{A},t} = \tilde{\mathcal{P}}_{\mathcal{A}}(t)$ for each time t. Contrary to the training phase where only single-container services were considered, here we apply *monitorless* to applications composed of multiple services. For scaling, our strategy to decide whether the application is experiencing resource saturation is to take the logical *OR* of the inferences over all instances of services comprising the application. In other words, our prediction vector is $\hat{y}_{\mathcal{A},t} = \bigvee_{I \in \mathcal{S}, \mathcal{S} \in \mathcal{A}} \hat{y}_{I,t}$. While other use cases might require different aggregations, *OR* should be sufficient for scaling instances. Although application performance may not show degradation when short saturation occurs in some components, scaling saturated instances is desirable – even if it does not directly influence the end-to-end latency.

Metrics We compare the prediction $\hat{y}_{\mathcal{A},t}$ to the ground truth label $y_{\mathcal{A},t}$ obtained by the threshold analysis $\tilde{\mathcal{P}}_{\mathcal{A}}(t)$, in order to evaluate the performance of our algorithm. We use the following standard metrics to assess accuracy:

- True Positives (TPs)/True Negatives (TNs) are correctly classified saturated/non-saturated samples, respectively.
- False Positives (FPs)/False Negatives (FNs) are incorrectly classified non-saturated/saturated samples, respectively.

Therefore, the goal is to minimize FPs and FNs. Standard metrics to summarize this goal are accuracy, computed as (TP+TN)/ (TP+TN+FP+FN), and the Sørensen-Dice coefficient or F1 score, computed as 2TP/ (2TP+FP+FN)). Typically, the costs associated with each FP and FN are inherently asymmetric in practice, because avoiding increasing latencies (FNs) could be more critical than avoiding unnecessary scaling decisions (FPs) – or the other way around. Our solution supports adaptation towards this asymmetry, by manipulating the prediction threshold of our classifier towards the direction of preference. In this work, we aim to minimize FNs by being more conservative with our predictions. Therefore, we set the prediction threshold of the random forest classifier to 0.4.

A critical issue discovered during the preliminary evaluation is that many FPs/FNs and ground truth "saturated" samples are close in time, but not precisely aligned, e.g., many FPs are followed by an FN within one or two samples. The reason for this delay is that saturated applications have increased response times. That is, during peak periods, we observe response times of up to three seconds. After three seconds a request is usually dropped by our load generators. Since requests take longer to arrive back to the load generator during these peak periods, a gap is introduced between the recorded platform metrics and the ground-truth labels, which are both monitored at a 1-second interval.

To fix this, we introduce *lagged metrics*. For a given lagged metric FP_k (to be read as *false positives at distance k*), a false positive prediction is classified as such only if there are no ground truth "saturated" occurrences within the next k samples. Analogously, for FN_k, a false negative for a ground-truth "saturated" sample occurs only if the previous k samples are predicted as "not saturated". That is, if a false positive occurs at time t, and a ground-truth "saturated" sample occurs in the time range $[t+1, t+k]$, then the sample at time t is classified as a true negative TN_k. If a false negative occurs at time t, and a positive prediction occurs in the time range $[t-k, t-1]$, then such samples are added to TP_k.

This modification allows early "saturated" predictions to be transferred to later "saturated" ground-truth samples, thereby handling the above-mentioned application latency issues. Importantly, the symmetric case of a late prediction, i.e., after the saturation was already observed at the client, is still classified as incorrect under this metric. Based on this definition and the fact that our peak response times were limited to three seconds, we perform our evaluation with $k = 2$ (note that k is specific to the applications we use in this evaluation). Thus, in all subsequent discussions, we use the $F1_2$-score and Acc_2, which are defined analogously to the F1-score and accuracy, but for the lagged metrics. This step is necessary for coping with monitoring delays in the real system that cause predictions to be misaligned with the ground truth. By doing so, we align the predicted data with the observed data in order to appropriately judge the behavior of the system.

Baseline comparisons As approaches from related work (see Section 6) follow a different paradigm, we compare *monitorless* against four optimal baseline approaches based on static CPU and memory thresholds. The first is CPU threshold, based on the relative CPU usage of each service instance. The second is MEM threshold, based on the relative memory usage of each service instance. We also look at a disjunctive CPU-OR-MEM and conjunctive CPU-AND-MEM combinations, where instances are predicted saturated if CPU or/and MEM indicate saturation, respectively. We use CPU and memory for demonstration as they are sufficient for the evaluated applications, but any other resource threshold can be used.

We note that the considered baseline approaches have an unfair advantage over *monitorless* in that they are configured with knowledge of the entire input data in advance, including ground-truth labels. These ground-truth labels are then used to choose the *optimal* threshold that maximizes the F1 score. The presented baselines therefore represent the best possible outcome for threshold-based approaches, given that the threshold would be optimally configured. In practice, these thresholds are unrealistic to find, since the need to be configured before deployment based on expertise and understanding the relationship between application performance and resource usage. This task is not needed by *monitorless* as it works application agnostic. In addition to the four platform-metric baselines, we use one application-specific baseline based on actual KPI-measurements in Table 7 serving as upper bound.

4.1 Evaluation using a three-tier web application

The first validation application is based on the social networking engine Elgg [23] included in Cloudsuite.

4.1.1 Setup of the three-tier web application

The hardware used was identical to that described in Section 3.2.2. The application consists of three tiers, each deployed in their own service instance: (1) the front-end Elgg web-server, (2) the InnoDB database, and (3) Memcache to speed up the database-driven application [50].

Given that Memcache and a similar database are used for training (see Section 3.2), we stressed the front-end tier by sending static requests to access the web-server's index page.

Table 5. Comparison of the baseline approaches to *monitorless* using a three-tier web-serving application.

Algorithm	TN_2	FP_2	FN_2	TP_2	$F1_2$	Acc_2
CPU (97%)	610	8	0	1838	0.999	0.997
MEM (43%)	544	74	14	1824	0.976	0.964
CPU-OR-MEM	538	80	0	1838	0.978	0.967
CPU-AND-MEM	616	2	14	1824	0.995	0.993
monitorless	607	11	0	1838	0.997	0.995

We deploy all service instances as containers using Docker on one physical machine. The Elgg-container is assigned with one CPU core and 4GB of memory. The PCP monitoring agent is running on this machine and sends metrics to a second orchestrator machine. The workload is generated by a third machine running HttpLoadGenerator [66]. The workload pattern is similar to sin-noise1000 (see Table 1), but scaled down to 1/10 the intensity, as the web server could handle fewer requests than Solr.

4.1.2 Results for the three-tier web application

Table 5 shows the results of all baseline approaches compared to our model evaluated using the lagged metrics. We observe that the test samples have a saturated/not-saturated ratio of about 75%, which is the inverse of the one in the training phase for *monitorless*. This indicates that the model is not biased towards the majority training label. As the front-end component is heavily CPU-based, the CPU detector can effectively flag saturation. Observe that *monitorless* is very accurate, achieving no false negatives and only a few false positives, even without being tuned for the application. However, as both CPU and CPU-AND-MEM threshold-based approaches also perform well, we provide next an evaluation with more complex applications.

4.2 Evaluation of a multi-tenant, microservice-based environment

In this section, we evaluate *monitorless* in a more realistic multi-tenant scenario, where several microservice-based applications are running in a distributed environment. We aim to show the accuracy of our model, as well as its robustness to changes in the hardware configuration and underlying operating system, e.g., the services are running on a different operating system than the one used in the training phase.

4.2.1 Setup of the multitenant environment

In contrast to the previous experiments, the hardware for this test is composed of three different HP ProLiant DL360 Gen9 servers equipped with 32 GB of RAM and a 10 core Intel® Xeon® CPU E5-2650 v3 (Haswell) @ 2.30GHz (M1), a 12 core Intel® Xeon® CPU E5-2650 v4 (Broadwell) @ 2.20GHz (M2) and, finally, an 8 core Intel® Xeon® CPU E5-2640 v3 (Haswell) @ 2.60GHz (M3). M1 and M2 run Debian 9, whereas M3 is running Ubuntu 16.04. We use two similar servers as workload drivers (M4, M5). These servers are connected to a 1Gb LAN; recall that we used a 10Gb network for training. We distribute the following two test applications among the three test machines M1, M2, and M3.

(1) TeaStore [67] is a novel open-source, online storefront application designed for testing and benchmarking [34]. A *WebUI* service replies to HTTP requests and creates a front-end user view supplied with images by a second service called *Image Provider*. A third service, *Auth*, handles data encryption and authentication, while the fourth service, called *Recommender*, applies machine learning algorithms to recommend certain products to specific users. A fifth *Data Persistence* service is in charge of giving access to permanent storage. There is also a *Registry* service to handle load-balancing and communication between instances, and a seventh *Database* service running MariaDB [24]) used by the persistence layer. All containers are available as docker images [33].

(2) Sockshop [69] is a similar online storefront application but larger than TeaStore. It consists of 14 different services: *Edge-Router, Front-end, Payment, Catalogue, Catalogue-DB, Carts, Carts-DB, User, User-DB, Order, Order-DB, Shipping, Queue, and Queue-Master.*

In order to represent a multi-tenant environment, we deployed all TeaStore and Sockshop services on M1, M2, and M3 servers as follows – entries marked with (T) are TeaStore services, all others belong to Sockshop:

- M1: Catalogue, Catalogue-DB, Front-end, Queue, Recommender (T), Auth (T), Registry (T).
- M2: Edge-Router, Carts, Carts-DB, Order, Order-DB, Payment, Queue-Master, DB (T), Persistence (T).
- M3: User, User-DB, Shipping, Web-UI (T), and Image-Provider (T).

All containers have a memory limit of 4GB. The Auth (T), Catalogue-DB, Carts-DB, Order-DB, and User-DB have 2 CPU cores, all other services are assigned 1 core.

Load generation for TeaStore. We stressed the TeaStore application with the *HttpLoadGenerator* [66]. User actions, which generate the content of requests, are defined using stateful user profiles. Each time a request is sent, an idle user from a pool is selected to execute a single action on the store. The user performs that action and returns to the pool. The actions available to the user are: 1) log in, 2) browse the store for products, 3) add these products to the shopping cart, and 4) log out. The user's distinct actions stress the services in different ways. The number of users is chosen depending on the maximum load intensity, such that it guarantees that an idle user is available each time a request is sent. The arrival rate profile represents a realistic, but worst-case workload for clouds [62] with more variance and multiple daily patterns within the experiment. It is depicted in Figure 3 (cf. right axis). Note that the training is done on mostly smooth workloads; we purposely choose this challenging workload in order to evaluate the robustness of *monitorless*.

Load generation for Sockshop. The load is generated via Locust [44] using the standard user profile delivered by Sockshop [46]. Users log in, browse the catalog, add items to their cart, and then place orders. The load intensity is controlled via the number of concurrent users issuing requests to the system. Locust usually applies a constant load, once all clients are hatched. In order to apply varying load patterns, we start three different Locust runs in parallel. Each run takes 1000 seconds and slowly hatches all clients until a maximum load of 700 concurrent clients is reached after 700 seconds. A constant load pattern is then applied for the remaining 300 seconds before one run finishes. We start such runs after 1000, 3000, and 5000 seconds.

Table 6. Comparing different threshold-based approaches to *monitorless* with the *TeaStore* dataset.

Algorithm	TN_2	FP_2	FN_2	TP_2	$F1_2$	Acc_2
CPU (95%)	6805	179	7	202	0.685	0.974
MEM (90%)	228	6756	4	205	0.057	0.060
CPU-OR-MEM	180	6804	1	208	0.057	0.054
CPU-AND-MEM	6853	131	10	199	0.738	0.983
monitorless	6820	164	3	206	0.712	0.977

4.2.2 Results for the *TeaStore* application

Applications containers are dimensioned such that most of the target load can be handled; that is, only large load peaks cause the application to saturate. This leads to a saturation/non-saturation ratio of 2.9%, far lower than the training data. By analyzing the predictions of *monitorless*, we note that most TP_2 are for the Auth service, the Web-UI, and the Recommender service. Figure 3 displays a detailed breakdown of the predictions made by *monitorless* over time, showing also the injected workload and the measured response times.

We observe that *monitorless* is able to detect saturation occurring in the different types of services (Web-UI, Recommender, Auth, Registry and Persistence), each with their own bottlenecks. Note that we cannot determine the distribution of FN_2 among the individual services as the KPI function $\mathcal{P}_{\mathcal{A}}$ is only observable at the application level, and not at the service level. However, we verify that the services that *monitorless* predicted to be saturated were indeed saturated, by subsequently scaling the saturated components and re-injecting the same load pattern. Thus, we repeat the same test with additional instances (scaling) of the Web-UI, the Recommender and the Auth service on M2, since those were the three services causing the most saturation predictions.

Table 6 shows the comparison between *monitorless* and the baseline approaches for the run in Figure 3. Note that *monitorless* achieves an $F1_2$-score of 0.712 compared to the best baseline approach with 0.738. Although the $F1_2$-score is lower than in the previous examples, it still amounts to 0.977 with only 3 false negatives. In contrast to the previous experiment, the CPU-AND-MEM has the best score but at expenses of a higher FN_2 count, which implies that it fails to detect crucial saturation samples.

To show this effect in more detail, we analyze the application performance end-to-end for an autoscaling scenario. We start with *baseline* deployment of each service and scale-out when saturation is predicted (see Figure 3). All replicated services have a lifespan of 120 seconds, after which scale-in occurs again in order to avoid the issue of endless out-scaling. This is the same for all analyzed approaches. Results are shown in Table 7, which reports the additional container provisioning related to the baseline non-scaled application (see second column), and the number of SLO violations (see third column) incurred by each technique (see first column). Using the approach in section 2.2, we indicate SLO violations when the average response time of all requests is higher than 750 ms, or if any request is dropped due to overload, or if more than 10% of requests fail during each one-second interval.

Note that we add the baseline worst-case "No Scaling" for reference. This baseline has static resources over the execution and

Table 7. Comparing different approaches to *monitorless* scaling on the *TeaStore* dataset. The average provisioning is calculated as the percentage of containers elastically added to the baseline case (i.e., the non-scaled initial deployment).

Algorithm	Provisioning (Ave)	SLO viol. (#)
A-posteriori CPU (95%)	+12%	12
A-posteriori MEM (90%)	+33%	9
CPU-OR-MEM	+39%	4
CPU-AND-MEM	+9%	17
monitorless	+10%	7
No Scaling (*baseline*)	0%	183
RT-based (*optimal*)	+7%	1

allows to understand how the different techniques provision containers upon scaling out events. Intuitively, the number of SLO violations should decrease with a larger provisioning (i.e., the system is more elastic); however a larger provisioning incurs a larger cost. Similarly, we also include the *optimal* response-time auto-scaler, which is based on postmortem end-to-end latency measurements. As the optimal auto-scaler does not know which instance causes a latency increase, we use our application knowledge to scale both the Recommender and the Auth service when the latency increases. For a fair comparison, all approaches are tied to scaling both Recommender and Auth at the same time, i.e., they scale both if one of them is predicted saturated.

Results are inline with Table 6. The optimal approach is able to reduce the SLO violations from 183 to 1 sample by using 7% more resources. This is expected as the RT-based approach directly observes the response time that is used as SLO. The remaining violations are due to the natural reaction time of the approaches and is therefore unavoidable.

We observe that *monitorless* manages to effectively reduce violations, while provisioning only 10% additional resources – consider that the optimal approach provisions 7% in excess. The only approach that is 'cheaper', in terms of resource provisioning, is the CPU-AND-MEM approach. However, CPU-AND-MEM allows more than twice as much SLO violations, while saving only 1% in comparison to *monitorless*, which is not a favorable trade-off. Both CPU-OR-MEM scaler and the MEM-based scaler provision 3 to 4 times more containers than *monitorless* due to the higher number of FPs_2 as shown in Table 6. The CPU-based scaler makes a reasonable trade-off between cost and SLO violations, but performs considerably worse than *monitorless* in both dimensions.

Although there is room for improvement with *monitorless*, we believe its performance on the TeaStore application is very impressive considering that: 1) it has never been trained with any of the TeaStore services; 2) it is running on slightly different hardware and operating systems than what it was trained on, 3) it is not using application metrics to make predictions; 4) it is operating in the presence of interference from another application, and, finally; 5) it achieves this performance with no knowledge of how the TeaStore behaves for this workload, unlike the threshold-based baseline approaches. Furthermore, the experiment shows that *monitorless* can detect bottlenecks on co-located services stressing different resources. Scaling these services results in a significant reduction of response time, where *monitorless* achieves results comparable to

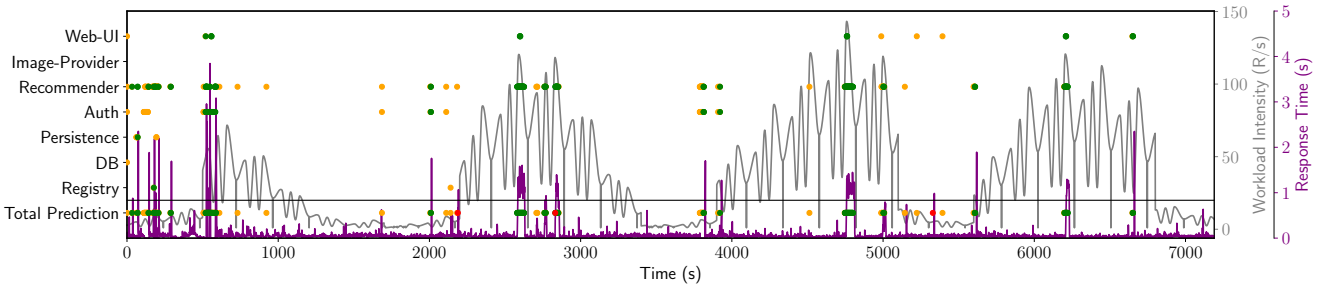

Figure 3. Predictions by service for the TeaStore over time. Green dots mark TP_2, yellow dots mark FP_2 and red dots are FN_2. TNs are not shown since are the most common and easily predicted. The gray curve displays the workload intensity (in requests per second) over time. The purple curve displays the measured average response time per second.

Table 8. Comparing different threshold-based approaches to *monitorless* using the *Sockshop* dataset.

Algorithm	TN_2	FP_2	FN_2	TP_2	$F1_2$	Acc_2
CPU (99%)	2036	657	3	301	0.447	0.780
MEM (10%)	683	2010	8	296	0.227	0.327
CPU-OR-MEM	595	2098	2	302	0.223	0.299
CPU-AND-MEM	2604	89	93	211	0.699	0.939
monitorless	2418	275	57	247	0.598	0.889

a simple autoscaler but without using any application knowledge. Note that even though the baseline approaches use perfect information, i.e., application-level metrics as ground-truth with a-posteriori knowledge of the evaluation set, there is no approach in the evaluation that outperforms *monitorless* in all scenarios. Moreover, the optimally-tuned thresholds differ for each alternative approach, whereas *monitorless* performs consistently across all experiments without tuning.

4.2.3 Results for the *Sockshop* application

Compared to TeaStore, the performance of the larger and more complex Sockshop application is more challenging to predict, especially with interference. The Sockshop runs are similar than those of TeaStore but contain only 999 samples each, resulting in a total of 2997 samples. The percentage of saturated samples is 10.1% (cf. Table 8). We observe that both the FP_2 and the FN_2 rates are higher than in the TeaStore experiment for *monitorless*. Overall, *monitorless* achieves an $F1_2$-score of 0.598 together with an accuracy of 89%, only surpassed by the CPU-AND-MEM baseline, with an $F1_2$-score of 0.699. However, note that the baseline approaches apply different thresholds than in the other experiments (97%/43%, 95%/90%, 99%/10%). In contrast, the *monitorless* model was unmodified through the evaluation. This illustrates that *monitorless* performs well on significantly more complex applications without having to manually adapt it different application-level metrics.

As our aggregation function over services is the logical OR of all predictions, this naturally creates more FPs as we increase the number of services. Hence, in order to tackle larger applications, this experiment motivates a more sophisticated approach for aggregation.

5 Discussion

We have shown that *monitorless* is generic and feasible in practice for predicting performance degradation. However, there are still some limitations requiring further research. In this section, we elaborate on how we plan to address them.

Generalize to broader set of services and platforms. We are aware of the diversity of microservices and cloud platforms that can generate mixed bottlenecks and more complex resource utilization patterns. To further generalize *monitorless* while improving its accuracy and robustness, we plan to incorporate datasets with different resource saturation samples, together with different and noisy workload patterns, applications with different resiliency, scaling and load balancing setups, as well as executions on different hardware platforms (e.g., GPUs and other accelerators). Additionally, we plan to experiment with different and automatically generated feature sets.

Refine the architecture. Our ongoing work is to improve the network overhead for large scale systems. A possible approach is to offload orchestrator functionality to the agents, e.g., the saturation prediction in Step 1 (see section 2). This allows network traffic to be reduced at the expense of higher CPU overhead in the agents and less data available at the orchestrator, possibly preventing elaborated decision-making.

Calibration. *Monitorless* may require additional calibration to infer the performance of applications with resource usage patterns significantly different from those in the training set. In machine learning, this is referred to as domain adaptation [54]. We plan to experiment with heuristics to best support domain adaptation, in the case where there there is no labeled data in the target domain (i.e., the new application) on which to calibrate the model.

Interpretability. A benefit of decision tree-based ensemble models is the possibility of generating user-interpretable scaling rules using *monitorless*. This may be in the interest of the application developers to identify bottlenecks in their software and make design decisions. We plan to experiment with depth-restricted decision trees or LIME [58] in order to simplify the model understanding in future work.

Using *monitorless* for autoscaling. We applied the predictions of *monitorless* to auto-scaling policies in the evaluation for demonstration purposes. Although, scaling down decisions are less critical from a system perspective, it is possible to extend

our approach training an additional classifier for detecting over-provisioned services and conservatively scale in to reduce costs. This makes it possible to recommend the exact amount of service instances required for any particular scenario.

6 Related Work

There are several approaches in both industry and academia for augmenting cloud operation with machine learning. The following taxonomy is based on the KPIs required by such approaches.

KPI-driven solutions. These approaches rely on application specific metrics. For example, Satopaa et al. [59] require KPIs to find valid operating ranges without resource saturation. In many cases, additional metrics are required, e.g., the workload type or intensity [26–28, 64, 73], or an indication of SLA fulfillment [29]. Other proposals [5, 45] need KPIs as input to machine learning models to predict performance degradation. Instead, our proposal is to infer KPI degradation without actually measuring it.

Black-box techniques. Other approaches treat applications as a black-box and use either online (e.g., resource pressure models [49]) or offline analysis (e.g., bytecode benchmarking [41]) to infer performance. Kundu et al. [42] use machine learning to create performance models. Yet, these approaches only work for the specific software running in the black-box and need to be (re)created for each target application. Emeakaroha [16, 17] relies on manually created mappings between low-level metrics and high-level SLAs. Wood et al. [70] achieve bottleneck detection based on fixed threshold rules build from a limited set of platform metrics. Similarly, commercial [2, 30, 47] and open-source [22, 52] autoscaling solutions rely only on platform metrics but require expertise to manually combine the scaling triggers properly.

Automated and application-agnostic approaches. Other sophisticated proposals apply machine learning to application-level metrics to predict performance. Hence, they do not need to measure application KPIs at runtime. These proposals are applied, for example, to a web-server for performance modeling during VM migrations [36]; or to a video-streaming service for KPI prediction [71, 72]; or for training Bayesian networks to classify SLO compliance of web-servers [11]. In contrast to *monitorless*, there is no evidence on how generic these models are, and whether they can be used autonomously across various applications and platforms.

Summarizing, approaches from literature are usually tuned towards a specific application. This is done either by specifically monitoring KPIs for that application or by analyzing, adapting and/or training a specific use-case of that application. There is no evidence that the trained model can be transferred to prior unseen applications. In contrast, our work aims at predicting application performance without measuring KPIs or any application-level metrics while creating *one* prediction model transferable to a large set of different applications.

7 Conclusions

We have introduced *monitorless*, a method for inferring application KPI degradation in cloud. The differentiating feature of *monitorless* is that only platform-level metrics are required as input, instead of application-specific metrics, allowing for accelerated software on-boarding and feature releases. With this approach, operation engineers can rely on a generic and stable set of metrics to streamline testing without application expertise or handcrafted configurations.

To realize this idea, we use machine learning to correlate KPI degradation to platform-level metrics and predict whether applications will exhibit performance bottlenecks. This information can be used for fast resource provisioning and allocation as *monitorless* can detect such bottlenecks in a matter of seconds.

We trained *monitorless* using workloads produced by four different benchmark services and evaluated its performance with three unknown applications composed of three, seven, and fourteen services, respectively. Even for complex micro-service applications, operating in the presence of interference, the accuracy achieved by our approach can be as high as 97%. This motivates the inclusion of even more test services, with diverse resource boundaries, to improve predictions and detect performance degradation across multiple resources.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Amazon. 2018. Amazon Web-Services Auto Scaling. Retrieved September 5, 2019 from https://aws.amazon.com/autoscaling/

[3] Amazon. 2018. DevOps and AWS. Retrieved September 5, 2019 from https://aws.amazon.com/devops/

[4] Appcentrica. 2016. "Deathstar Diagrams". Retrieved September 5, 2019 from "https://www.appcentrica.com/the-rise-of-microservices"

[5] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. 2009. Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing (HotCloud'09)*. USENIX Association, Berkeley, CA, USA, Article 12, 5 pages. http://dl.acm.org/citation.cfm?id=1855533.1855545

[6] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[7] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. 1984. *Classification and regression trees*. Routledge, New York, NY, USA.

[8] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[9] François Chollet et al. 2015. Keras. https://keras.io.

[10] Performance Co-Pilot. 2018. Performance Co-Pilot. Retrieved September 5, 2019 from http://pcp.io/

[11] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. 2004. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA, 16–16. http://dl.acm.org/citation.cfm?id=1251254.1251270

[12] Distributed (Deep) Machine Learning Community. 2018. Scalable, Portable and Distributed Gradient Boosting (GBDT, GBRT or GBM) Library, for Python, R, Java, Scala, C++ and more. Runs on single machine, Hadoop, Spark, Flink and DataFlow. Retrieved September 5, 2019 from https://github.com/dmlc/xgboost

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[14] David R Cox. 1958. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)* 20, 2 (1958), 215–232.

[15] DZone.com. 2018. Monitoring Docker Containers - Docker Stats, cAdvisor, Universal Control Plane. Retrieved September 5, 2019 from https://dzone.com/articles/monitoring-docker-containers-docker-stats-cadvisor

[16] V. C. Emeakaroha, I. Brandic, M. Maurer, and S. Dustdar. 2010. Low level Metrics to High level SLAs - LoM2HiS framework: Bridging the gap between monitored metrics and SLA parameters in cloud environments. In *2010 International Conference on High Performance Computing Simulation*. IEEE Computer Society,

Washington, DC, USA, 48–54. https://doi.org/10.1109/HPCS.2010.5547150
[17] Vincent C Emeakaroha, Marco AS Netto, Rodrigo N Calheiros, Ivona Brandic, Rajkumar Buyya, and César AF De Rose. 2012. Towards autonomic detection of SLA violations in Cloud infrastructures. *Future Generation Computer Systems* 28, 7 (2012), 1017–1029.
[18] Uber Engineering. 2017. Observability at Uber Engineering: Past, Present, Future. Retrieved September 5, 2019 from https://www.youtube.com/watch?v=2JAnmzVwgP8
[19] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A Library for Large Linear Classification. *Journal of Machine Learning Research* 9 (2008), 1871–1874.
[20] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *SIGPLAN Not.* 47, 4 (March 2012), 37–48. https://doi.org/10.1145/2248487.2150982
[21] Apache Software Foundation. 2018. Apache Cassandra. Retrieved September 5, 2019 from http://cassandra.apache.org/
[22] Apache Software Foundation. 2018. Apache CloudStack: Open Source Cloud Computing. Retrieved September 5, 2019 from https://cloudstack.apache.org/
[23] The Elgg Foundation. 2018. Elgg.org. Retrieved September 5, 2019 from https://elgg.org/
[24] The MariaDB Foundation. 2018. MariaDB.org - Supporting continuity and open collaboration. Retrieved September 5, 2019 from https://mariadb.org/
[25] Yoav Freund and Robert E Schapire. 1997. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *J. Comput. System Sci.* 55, 1 (1997), 119 – 139. https://doi.org/10.1006/jcss.1997.1504
[26] Yu Gan, Meghna Pancholi, Siyuan Hu, Dailun Cheng, Yuan He, and Christina Delimitrou. 2018. Seer: Leveraging Big Data to Navigate the Increasing Complexity of Cloud Debugging. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. USENIX Association, Boston, MA, 15. https://www.usenix.org/conference/hotcloud18/presentation/gan
[27] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. 2014. Adaptive, Model-driven Autoscaling for Cloud Applications. In *11th International Conference on Autonomic Computing (ICAC 14)*. USENIX Association, Philadelphia, PA, 57–64. https://www.usenix.org/conference/icac14/technical-sessions/presentation/gandhi
[28] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. 2012. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Trans. Comput. Syst.* 30, 4, Article 14 (Nov. 2012), 26 pages. https://doi.org/10.1145/2382553.2382556
[29] Daniel Gmach, Stefan Krompass, Andreas Scholz, Martin Wimmer, and Alfons Kemper. 2008. Adaptive Quality of Service Management for Enterprise Services. *ACM Trans. Web* 2, 1, Article 8 (March 2008), 46 pages. https://doi.org/10.1145/1326561.1326569
[30] Google. 2018. Google Compute Engine Auto Scaler. Retrieved September 5, 2019 from https://cloud.google.com/compute/docs/autoscaler/
[31] Google. 2018. Site Reliability Engineering (SRE). Retrieved September 5, 2019 from https://landing.google.com/sre/
[32] Brendan Gregg. 2013. Thinking Methodically About Performance. *Commun. ACM* 56, 2 (Feb. 2013), 45–51. https://doi.org/10.1145/2408776.2408791
[33] Descartes Research Group. 2018. descartesresearch - Docker Hub. Retrieved September 5, 2019 from https://hub.docker.com/r/descartesresearch/
[34] Descartes Research Group. 2018. A micro-service reference test application for model extraction, cloud management, energy efficiency, power prediction, multi-tier auto-scaling. Retrieved September 5, 2019 from https://github.com/DescartesResearch/TeaStore
[35] B. Harrington and R. Rapoport. 2014. "Introducing Netflix's Primary Telemetry Platform". Retrieved September 5, 2019 from "http://techblog.netflix.com/2014/12/introducing-atlas-netflixs-primary.html"
[36] Helmut Hlavacs and Thomas Treutner. 2011. Predicting web service levels during vm live migrations. In *2011 5th International DMTF Academic Alliance Workshop on Systems and Virtualization Management: Standards and the Cloud (SVM)*. IEEE, IEEE Computer Society, Washington, DC, USA, 1–10.
[37] Huaweii. 2017. TestCraft. Testing as a Service to Accelerate SDN/NFV Service Deployment. Retrieved September 5, 2019 from http://carrier.huawei.com/~/media/CNBG/Downloads/Services/nfv/Testing%20as%20a%20Service%20to%20Accelerate%20SDN%20NFV%20Service%20Deployment.pdf
[38] Jesus Omana Iglesias, Jordi Arjona Aroca, Volker Hilt, and Diego Lugones. 2017. ORCA: an ORChestration Automata for Configuring VNFs. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17)*. ACM, New York, NY, USA, 81–94. https://doi.org/10.1145/3135974.3135982
[39] Danga Interactive. 2018. Memcached - A distributed object memory caching system. Retrieved September 5, 2019 from https://memcached.org/
[40] Vijay Janapa Reddi, Benjamin C Lee, Trishul Chilimbi, and Kushagra Vaid. 2010. Web search using mobile cores: quantifying and mitigating the price of efficiency. *ACM SIGARCH Computer Architecture News* 38, 3 (2010), 314–325.
[41] Klaus Krogmann, Michael Kuperberg, and Ralf Reussner. 2010. Using genetic search for reverse engineering of parametric behavior models for performance prediction. *IEEE Transactions on Software Engineering* 36, 6 (2010), 865–877.
[42] Sajib Kundu, Raju Rangaswami, Ajay Gulati, Ming Zhao, and Kaushik Dutta. 2012. Modeling Virtualized Applications Using Machine Learning Techniques. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2151024.2151028
[43] Philipp Leitner and Jurgen Cito. 2016. Patterns in the Chaos: A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Trans. Internet Technol.* 16, 3, Article 15 (April 2016), 23 pages. https://doi.org/10.1145/2885497
[44] Locust. 2018. Scalable user load testing tool written in Python. Retrieved September 5, 2019 from https://github.com/locustio/locust
[45] Andréa Matsunaga and José A. B. Fortes. 2010. On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID '10)*. IEEE Computer Society, Washington, DC, USA, 495–504. https://doi.org/10.1109/CCGRID.2010.98
[46] microservices demo. 2018. Deployment scripts and config for Sock Shop. Retrieved September 5, 2019 from https://github.com/microservices-demo/microservices-demo
[47] Microsoft. 2018. Microsoft Azure Autoscale. Retrieved September 5, 2019 from http://azure.microsoft.com/features/autoscale/
[48] Apache Software Foundatio n. 2018. Apache Solr. Retrieved September 5, 2019 from https://lucene.apache.org/solr/
[49] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. 2013. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. USENIX, San Jose, CA, 69–82. https://www.usenix.org/conference/icac13/technical-sessions/presentation/nguyen
[50] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (nsdi'13)*. USENIX Association, Berkeley, CA, USA, 385–398. http://dl.acm.org/citation.cfm?id=2482626.2482663
[51] Openstack. 2016. Openstack Kolla. Retrieved September 5, 2019 from http://docs.openstack.org/developer/kolla/
[52] OpenStack. 2018. Open Stack is open source software for creating private and public clouds. Retrieved September 5, 2019 from https://www.openstack.org/
[53] T. Palit, Yongming Shen, and M. Ferdman. 2016. Demystifying cloud benchmarking. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, Washington, DC, USA, 122–132. https://doi.org/10.1109/ISPASS.2016.7482080
[54] Sinno Jialin Pan and Qiang Yang. 2010. A Survey on Transfer Learning. *IEEE Trans. Knowl. Data Eng.* 22, 10 (2010), 1345–1359. https://doi.org/10.1109/TKDE.2009.191
[55] Karl Pearson. 1901. LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2, 11 (1901), 559–572.
[56] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[57] Quantcast. 2014. Monitoring at Quantcast. Retrieved September 5, 2019 from https://www.quantcast.com/wp-content/uploads/2013/10/Wait-How-Many-Metrics_-Quantcast-2013.pdf
[58] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, NY, USA, 1135–1144.
[59] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. 2011. Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems Workshops (ICDCSW '11)*. IEEE Computer Society, Washington, DC, USA, 166–171. https://doi.org/10.1109/ICDCSW.2011.20
[60] Abraham Savitzky and Marcel JE Golay. 1964. Smoothing and differentiation of data by simplified least squares procedures. *Analytical chemistry* 36, 8 (1964), 1627–1639.
[61] Mark Schmidt, Nicolas Le Roux, and Francis Bach. 2017. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming* 162, 1-2 (2017), 83–112.
[62] S. Shen, V. Beek, and A. Iosup. 2015. Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE Computer Society, Los Alamitos, CA, USA, 465–474. https://doi.org/10.1109/CCGrid.2015.60
[63] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2015. CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. ACM, New York, NY, USA, Article 23, 7 pages. https://doi.org/10.1145/2774993.2775066
[64] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. 2011. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements. In *Proceedings of the*

9th USENIX Conference on File and Stroage Technologies (FAST'11). USENIX Association, Berkeley, CA, USA, 12–12. http://dl.acm.org/citation.cfm?id=1960475.1960487

[65] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-scale Advanced Analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, Berkeley, CA, USA, 363–378. http://dl.acm.org/citation.cfm?id=2930611.2930635

[66] Jóakim von Kistowski. 2018. HTTP Load Generator for variable load intensities. Retrieved September 5, 2019 from https://github.com/joakimkistowski/HTTP-Load-Generator

[67] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. 2018. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '18)*. IEEE Computer Society, Washington, DC, USA, 223–236. https://doi.org/10.1109/MASCOTS.2018.00030

[68] Jóakim von Kistowski, Nikolas Herbst, Samuel Kounev, Henning Groenda, Christian Stier, and Sebastian Lehrig. 2017. Modeling and Extracting Load Intensity Profiles. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 11, 4, Article 23 (January 2017), 28 pages. https://doi.org/10.1145/3019596

[69] Inc Weaveworks. 2018. Sock Shop – A Microservices Demo Application. Retrieved September 5, 2019 from https://microservices-demo.github.io/

[70] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. 2009. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks* 53, 17 (2009), 2923–2938.

[71] R. Yanggratoke, J. Ahmed, J. Ardelius, C. Flinta, A. Johnsson, D. Gillblad, and R. Stadler. 2015. Predicting real-time service-level metrics from device statistics. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE Computer Society, Washington, DC, USA, 414–422. https://doi.org/10.1109/INM.2015.7140318

[72] Rerngvit Yanggratoke, Jawwad Ahmed, John Ardelius, Christofer Flinta, Andreas Johnsson, Daniel Gillblad, and Rolf Stadler. 2015. Predicting Service Metrics for Cluster-based Services Using Real-time Analytics. In *Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM) (CNSM '15)*. IEEE Computer Society, Washington, DC, USA, 135–143. https://doi.org/10.1109/CNSM.2015.7367349

[73] Li Yin, Sandeep Uttamchandani, and Randy Katz. 2006. An Empirical Exploration of Black-Box Performance Models for Storage Systems. In *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS '06)*. IEEE Computer Society, Washington, DC, USA, 433–440. https://doi.org/10.1109/MASCOTS.2006.12