IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 9 February 2023, accepted 16 March 2023, date of publication 21 March 2023, date of current version 28 March 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3260069

RESEARCH ARTICLE

Trace-Driven Scaling of Microservice Applications

VAHID MIRZAEBRAHIM MOSTOFI[1], EVAN KRUL[1], DIWAKAR KRISHNAMURTHY[1], (Member, IEEE), AND MARTIN ARLITT[2], (Senior Member, IEEE)

[1]Department of Electrical and Software Engineering, University of Calgary, Calgary, AB T2N 1N4, Canada
[2]Department of Computer Science, University of Calgary, Calgary, AB T2N 1N4, Canada

Corresponding author: Diwakar Krishnamurthy (dkrishna@ucalgary.ca)

This work was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

ABSTRACT The containerized microservices architecture is being increasingly used to build complex applications. To minimize operating costs, service providers typically rely on an auto-scaler to ''right size'' their infrastructure amid fluctuating workloads. The agile nature of microservice development and deployment requires an auto-scaler that does not require significant effort to derive resource allocation decisions. In this paper, we investigate reducing auto-scaler development effort along a number of dimensions. First, we focus on a technique that does not require an expert to develop a model, e.g., a queuing model or machine learning model, of the system and tweak the model as the underlying microservice application changes. Second, we explore ways to limit the number of workload patterns that need to be considered. Third, we study techniques to reduce the number of resource allocation scenarios that one has to explore before deploying the auto-scaler. To address these goals, we first analyze the workload of 24,000 real microservice applications and find that a small number of workload patterns dominate for any given application. These results suggest that auto-scaler design can be driven by this small subset of popular workload patterns thereby limiting effort. To limit the number of resource allocation scenarios explored, we develop a novel heuristic optimization technique called MOAT, which outperforms Bayesian Optimization often used for such exercises. We combine insights obtained from real microservice workloads and MOAT to realize an auto-scaler called TRIM that requires no system modeling. For each popular workload pattern identified for an application, TRIM uses MOAT to pre-compute a near minimal resource allocation that satisfies end user response time targets. These resource allocations are then used at runtime when appropriate. We validate our approach using a variety of analytical, on-premise, and public cloud systems. From our results, TRIM in consort with MOAT significantly improves the performance of the industry-standard HPA auto-scaler by achieving up to 92% fewer response time violations and up to 34% lower costs compared to using HPA in isolation.

INDEX TERMS Auto-scalers, containers, microservice architecture, resource allocation, software performance.

I. INTRODUCTION

Microservice architectures are being increasingly used to build complex enterprise applications. With such an architecture, an application is developed as a collection of lightweight and loosely coupled services that interact with one another. Often, a microservice is packaged using containerization technology [1]. There are several reasons for the increased use of microservice architectures. For example, designing a complex application service by breaking it down into simple microservices can enable Continuous Integration-Continuous Delivery (CI-CD). The CI-CD paradigm places emphasis on simplifying application development, maintenance, and operation while enabling agile and frequent roll outs of changes. In a survey done by Shahin et al. [2], 53.7% of participants mentioned that the applications they had some role in the development of were in a deployable state *one or more times per day*.

The associate editor coordinating the review of this manuscript and approving it for publication was Michael Lyu.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

Many microservice applications are customer-facing and hence have stringent performance requirements. Specifically, these applications are expected to satisfy predefined response time targets for requests submitted by their end users. These response time targets are usually included as part of Service Level Agreements (SLAs) operators of an application service have with end users of that service. Customer-facing services often experience workload fluctuations [3]. Hence, a microservice application needs to use scaling techniques to handle such fluctuations. Scaling techniques ensure enough resources are allocated to an application so that SLA targets continue to be met in spite of workload fluctuations. The survey by Salah et al. [4] mentions the ability to simplify scaling as a key reason for the adoption of the microservice architecture.

Modern enterprise applications often use auto-scaling to ensure performance requirements are satisfied continuously. Auto-scaling is a technique where the resources, e.g., CPU and memory, allocated to an application are changed at runtime so that the application continues to satisfy its SLA objectives. Two key requirements in auto-scaler design are SLA Awareness, i.e., the ability to satisfy pre-defined response time targets, and efficiency, i.e., allocating just the right amount of resources to achieve these targets in a cost-efficient manner. While not allocating enough resources can lead to response time violations, over-provisioning leads to cost overruns for the service operator. We term the module within the auto-scaler that performs this operation as the resource allocation module.

Microservice applications impose a third unique requirement of CD compatibility, i.e., the ability to support agile and frequent software releases. The auto-scaler design should be consistent with this requirement. Specifically, CD compatibility can be assessed along several dimensions based on the effort needed to develop and maintain the auto-scaler. First, many existing approaches require an expert to develop a model, e.g., a queuing model or machine learning (ML) model, of the system and many organizations may not have access to such experts or may wish to avoid the costs or delays that hiring a consultant to provide such expertise might entail. In addition to requiring an expert, model development typically involves significant effort. For example, ML models may require feature engineering and hyper-parameter tuning initially. Some of these tasks need to be repeated when the underlying microservice application changes, e.g., when existing microservices undergo extensive modifications or additional types of requests and microservices are introduced to support new features [5], [6]. Second, to be accurate, models in general need to be validated or trained against a large number of workload scenarios. Third, the modeling exercise should also consider a large number of resource allocation scenarios to be effective. With these two considerations, adapting the model in response to application changes can incur a prohibitively large effort and cost.

Although there have been many studies on developing auto-scalers for microservice applications [7], [8], [9], [10], [11], these existing techniques do not satisfy the three requirements of SLA awareness, resource efficiency, and CD compatibility simultaneously. In particular, many existing techniques require model-building and the concomitant extensive exploration of workload patterns and resource allocation scenarios, which can hinder agile deployment. To address this issue, we explore a CD compatible approach that does not require system modeling, relies on only a limited number of workload patterns, and reduces the number of resource allocations that need to be explored prior to auto-scaler deployment. We also design the approach such that it is SLA-aware and resource efficient.

Specifically, we consider the following research questions:

- RQ1 - Do real-world microservice workloads have properties that can reduce the number of workload patterns that one needs to consider while devising an auto-scaler?
- RQ2 - How can knowledge of an application's service architecture and bottlenecks be exploited to limit the number of resource allocations explored while identifying a cost-effective resource allocation, prior to auto-scaler deployment?
- RQ3 - How does an auto-scaler that exploits the answers to RQ1 and RQ2 perform when compared to an industry-standard, CD-compatible auto-scaler?

The three novel contributions that underpin our effort are described as follows. First, we analyze workloads of over 24, 000 real-world microservice applications [12] to discover a novel insight that enables the design of a CD-compatible auto-scaler. Specifically, we find that a small number of workload patterns occur repeatedly, i.e., workloads exhibit skewed popularity (RQ1). Furthermore, popular workloads observed during a time period also tend to be popular in the subsequent time period, i.e., workload patterns exhibit good temporal locality (RQ1). This result suggests that auto-scaler design can be driven by this small set of popular workload patterns for CD compatibility.

Second, we develop a novel resource allocation module called the Microservices Application Tuner (MOAT). MOAT limits the number of resource allocation scenarios explored by quickly identifying an efficient resource allocation for any given workload in the set of popular workload patterns. The technique employs an iterative performance testing approach during a profiling phase. It accepts as inputs the workload and response time targets for various classes of requests supported by the application. It first generates initial resource allocations, e.g., CPU shares, for individual microservices. MOAT refines the resource allocation incrementally by monitoring service response times and resource usage during this test and by exploiting knowledge of the application's service architecture. A new performance test is then launched with the refined allocation. The process of refining and testing allocations is continued iteratively until all response time targets are met. A key feature of this algorithm is that it determines resource allocations without resorting to a large

number of performance tests thereby speeding up the tuning process and enabling CD compatibility.

Third, we build a trace-driven microservices auto-scaler (that we call TRIM) that leverages the above two contributions to simultaneously realize SLA awareness, efficiency, and CD compatibility. TRIM does not require model development. It considers the popular workload patterns observed at a system during a time period, e.g., a day, and pre-computes resource allocations for this small set of patterns in the profiling phase by using MOAT. The popularity skew ensures that the profiling phase is short thereby supporting the agility demanded by CD. At runtime the auto-scaler measures the workload expected at any given time instant. If the measured workload closely matches one of the workloads explored during the profiling phase, the corresponding resource allocation is applied to the system. Otherwise, the auto-scaler falls back on a default rule-based algorithm used currently by practitioners [13]. The most frequent patterns can be updated periodically, e.g., once per day, and the process repeated to track subtle workload changes over time.

A comprehensive evaluation that considers real-world, analytical, on-premise, and public cloud systems shows that our approach outperforms existing baseline techniques. For example, considering the busiest of the 24, 000 applications in the Azure function traces [12], the top 5 workload patterns in a day account for 78% of the observations that day (RQ1). These patterns capture 75% of the patterns the next day suggesting good promise for our approach (RQ1). Furthermore, MOAT promotes CD compatibility by significantly cutting down the number of resource allocation scenarios explored compared to a Bayesian Optimization (BO) approach, commonly used for similar exercises. Specifically, on a public cloud testbed, MOAT requires 4.8 times fewer iterations than BO to converge to a near optimal resource allocation (RQ2). Furthermore, MOAT identifies more efficient, i.e., cost-effective, allocations. For the same cloud system, MOAT's resource allocations reduced operational cost by 48% compared to those identified by BO (RQ2). Next, we evaluate TRIM by considering a representative application from the Azure function traces and emulating its workload over a 24 hour period on an on-premise system. MOAT quickly pre-computes the resource allocations for the top 10 patterns during a 13 minute profiling phase. When we only employ the industry-standard and CD compatible HPA auto-scheduler [13], we notice significant SLA violations and excessive resource usage. By leveraging the pre-computed resource allocations from MOAT, TRIM is able to reduce *both* SLA violations and resource usage compared to HPA. Considering all experiments done on this system, TRIM reduces SLA violations and resource usage by up to 92% and 34%, respectively (RQ3).

We have previously published a preliminary version of MOAT's resource allocation algorithm (named ''the basic algorithm'' in this paper) [14]. This paper enhances the algorithm and leverages the enhancement in the design of TRIM. The workload analysis driving the design of TRIM is also a significant addition to this paper. Parts of this paper can be found in an expanded form in the masters thesis document of the first author [15].

The rest of the paper is organized as follows. Sec. II discusses related work. We present our workload analysis to motivate the design of TRIM in Sec. III. Sec. IV outlines the design of TRIM. Sec. V describes the MOAT resource allocation module. Evaluation and results of MOAT and TRIM are presented in Sec. VI and Sec. VII, respectively. Sec. VIII concludes the paper.

II. RELATED WORK

The surveys done by Qu et al. [16] and Singh et al. [17] provide comprehensive discussions on existing auto-scaling research. Auto-scaling techniques can be broadly grouped into *Rule-Based*, *Profiling Based*, *Analytical Modeling Based* and *ML Based* approaches. We now provide a brief discussion of these approaches.

Many public cloud providers [18] and open source container orchestrating tools [1] provide simple, static, rule-based solutions [13], [18] for scaling microservices. A rule specifies the allocation or removal of resources based on whether a chosen performance metric, e.g., CPU utilization, exceeds a static threshold or not. These techniques are CD-compatible since they do not require a lengthy profiling phase where a large number of workload patterns and resource allocation scenarios are explored and are convenient to use. However, setting the appropriate thresholds can be challenging. Improper selections can cause response time target violations or resource over-allocation [19], [20]. Extensions of these techniques use dynamic thresholds [21], [22] and fuzzy rules [16], [23], [24]. However, these techniques in general require complex tuning to ensure simultaneous SLA awareness and resource allocation efficiency.

Profiling based approaches collect experimental data from a target system to infer resource allocation decisions that can be applied when that system is deployed [17]. Many techniques conduct the profiling phase offline, prior to deploying the system [25], [26], [27], [28]. While these techniques can be designed to be SLA aware and efficient, the profiling experiments need to be repeated every time the application is updated. Since microservice applications tend to get updated frequently, care must be exercised to limit profiling experiments so that the technique is CD compatible. However, achieving a balance between profiling effort and resource estimation accuracy is challenging and is still an open problem. Other approaches rely on profiling data collected after a system has been deployed [29], [30]. Since it is undesirable to degrade the performance of a system, particularly for an extended period of time, only a limited number of experiments can be executed. Consequently, these techniques are less likely to identify optimal resource allocation decisions compared to the offline techniques.

Analytical modeling approaches [9], [31] build a mathematical model of a target system that can be solved using

queuing theory. They use a model to predict a system's performance under any given workload with different resource allocations. These predictions are used to arrive at an optimal resource allocation. A key challenge with this approach is the need for an expert to construct an accurate queuing model. An additional challenge is that the model has to be validated against a large number of workload patterns and resource allocation scenarios to ensure accuracy. Furthermore, the model has to be updated whenever the system changes [17], which impacts agile software roll outs and hence CD compatibility. Some studies address model building complexity by representing similar services using a single type of model [32]. Others simplify the modeling by building models for each microservice in an application instead of one complex model that captures the combined impact of all microservices [33], [34]. However, such an approach requires response time targets defined for the whole application to be decomposed into targets for each microservice, which can be a complex task [7].

Several studies leverage ML models to drive auto-scaling decisions. The learning process can be performed before or after deployment. Reinforcement Learning (RL) based techniques allow a system to learn how to react in a specific environment to maximize predefined rewards. RL models are trained while the system faces changing workloads. Gains and rewards are defined such that the system is nudged to find efficient solutions that meet SLA targets [35]. Regression based techniques [36], [37], [38] concentrate on learning the pattern between various system and workload variables to predict an appropriate number of resources. The trained model can be used for many purposes, such as running simulations or optimization. Both RL and regression based approaches require expertise to build and tune a model. Furthermore, they generally require extensive training data, e.g., workload patterns and resource allocations, to be effective. For example, RL models need time to converge, and an auto-scaler based on such models reacts poorly while the models are in the learning stage [16]. Furthermore, the models need to be tuned again with every application update. For example, these models might lose their accuracy if a microservice update has resulted in a significant change in the service's performance behaviour or if a completely new service has been introduced into the application. These reasons motivate our current effort of exploring an alternative technique that does not require model building, that can be effective with data from a limited number of workload patterns, and that does not need to explore a large number of resource allocation scenarios.

Commercial cloud and container orchestration systems support reactive auto-scaling approaches, e.g., the HPA auto-scaler [13] offered by Kubernetes. In contrast, Chameleon [34] and ASFM [11] are proactive auto-scaling approaches. These approaches train an ML model to predict the rate of arrival of requests to the system at a future time instant. The auto-scaler then proactively applies resource allocation for the predicted rate. Since training such workload prediction models with high accuracy is not trivial, we explore an alternative technique that leverages other characteristics such as workload popularity.

Table 1 compares prior works presented in this section that focus explicitly on microservice applications. We also include TRIM as the last line of the table. Most papers in the table did not publish their auto-scaler source code. While Taherizadeh and Stankovski [21] share their source code, their auto-scaler requires a specific platform called SWITCH [39] that makes using it as a baseline difficult. The other project that offers source code is FIRM. However, the auto-scaler requires a specific generation of processors [7] we do not have access to thereby making comparison impossible.

From the table, many of the approaches require a pre-deployment profiling phase. As discussed previously, the profiling needs to be repeated whenever an application is updated thereby necessitating new approaches that do not require significant profiling effort. Similarly, as discussed previously, approaches requiring queuing model development and ML model training need modeling experts, extensive evaluation of workload patterns and resource allocation scenarios, and model re-tuning due to updates. As shown in the table, two of the three rule-based techniques do not suffer from this limitation. However, these techniques are not SLA aware since they do not explicitly consider SLA targets. Such approaches are prone to SLA violations or resource overprovisioning.

TRIM addresses these limitations. It does not require model building, which requires expertise that might be difficult to access in many organizations. It exploits workload patterns observed in real world microservice workloads to limit the number of workload patterns that need to be evaluated. Furthermore, TRIM employs the MOAT resource allocation module to identify an efficient resource allocation using fewer iterations than the BO-based baseline considered in existing work. These features make TRIM SLA-aware, efficient, and CD-compatible. To promote repeatable research and future empirical comparisons of auto-scaling approaches, we have open-sourced TRIM and MOAT [40].

From Table 1, the PEMA [28] auto-scaler is similar to our approach in that it is SLA aware and requires no system modeling. However, unlike TRIM that supports allocation of multiple resources and considers popularity of workload patterns, PEMA focuses only on CPU provisioning and relies on a trial and error allocation approach. Furthermore, while the source code of PEMA has been published, there are no instructions to deploy and execute the system. The only other auto-scaler that does not require system modeling and is open-source (with documentation) is the industry-standard HPA auto-scaler implemented within the Kubernetes container orchestration system [1]. Given our emphasis on CD-compatible techniques that obviate the need for model building and maintenance, we select HPA as the baseline to evaluate TRIM's performance.

TABLE 1. Comparison of different microservice auto-scalers.

Work	Resource estimation	Scaling timing	SLA aware?	Requires profiling?	Requires modeling?	Source code
FIRM [7]	ML, RL	Reactive	Yes	Yes	Yes	Yes
Autopilot [8]	ML, Heuristics, Past data	Reactive	No	Yes	Yes	No
ATOM [9]	QueML, Simulationuing	Reactive	Yes	Yes	Yes	No
Microscaler [38]	Queuing, Heuristics	Reactive	Yes	Yes	Yes	No
ASFM [11]	Regression	Proactive	Yes	Yes	Yes	No
Liu et al. [41]	Fuzzy rule-based	Reactive	Yes	Yes	No	No
Abdullah et al. [37]	ML, Simulation	Proactive	Yes	Yes	Yes	No
Elascale [42]	Rule-based	Reactive	No	Yes	No	No
Abdullah et al. [43]	ML, Past data	Proactive	Yes	Yes	Yes	No
Khaleq et al. [35]	RL	Reactive	Yes	Yes	Yes	No
Kubernetes HPA [13]	Rule-based	Reactive	No	No	No	Yes
AWS Autoscaling [18]	Rule-based	Reactive	No	No	No	No
Chameleon [34]	Queuing, Past data	Proactive	No	Yes	Yes	No
me-kube [33]	Queuing	Both	Yes	Yes	Yes	No
Vondra et al. [31]	Queuing, Simulation	Reactive	Yes	Yes	Yes	No
Taherizadeh et al. [21]	Dynamic rule-based	Reactive	No	No	No	Yes
Fernandez et al. [25]	Runtime profiling	Proactive	Yes	Yes	No	No
PEMA [28]	Runtime profiling	Reactive	Yes	Yes	No	Yes
TRIM	Heuristics, Past data	Reactive	Yes	Yes	No	Yes

III. WORKLOAD ANALYSIS

In this section, the publicly available dataset of Azure Function Traces [12] is introduced and analyzed. This dataset contains information about invocations of serverless functions that are building units of microservice applications. The objective is to determine whether there are patterns in the workload that can be leveraged to realize a CD compatible auto-scaler that requires only a short profiling phase.

A. DATASET OVERVIEW AND PRELIMINARY ANALYSIS

The dataset used in this research is from microservice applications built using serverless functions and deployed on the Microsoft Azure public cloud. An application is built as a collection of functions supplied by an owner, i.e., a subscriber, of the platform. Functions are invoked by various triggers. The triggers are grouped into seven classes, namely HTTP, Event, Queue, Timer, Orchestration, Storage, and Others [12]. *HTTP* requests are the most common method for invoking functions. Functions that are invoked based on *Timer* triggers are similar to cron jobs, i.e., they have a predefined frequency in their invocations.

The dataset captures about 24, 000 applications cumulatively supporting more than 70, 000 functions. It covers a two week period from July 15, 2019 to July 28, 2019. Each day is divided into 1-minute intervals. The number of times each function is invoked in each of these 1-minute intervals is recorded along with information on the application and owner associated with that function.

More than 12 billion invocations are recorded in the dataset spanning all days and all applications. 54% of applications only have a single function while 95% of applications have 10 functions or less. Very few applications use a lot of functions. For example, only 0.04% of applications used more than 100 functions and only 4 applications used more than 1, 000 functions during the two-week period [12]. Also, only a small subset of applications are popular. Specifically, 99.6% of all invocations are directed to only 18% of applications.

Our analysis shows that in general applications from this dataset are likely to benefit from auto-scaling. Figure 1 and 2 characterizes the request rate, i.e., the rate at which functions belonging to an application are invoked, for three randomly selected applications. For each application, Figure 1 plots hourly request rates normalized to the peak hourly rate for a two-week period. To consider a finer timescale, Figure 2 also plots the request rate over a minute for the first three hours of the dataset normalized by the peak over that period. It can be observed that all three applications encounter workload fluctuations at both long and short timescales. Allocating resources to handle the peak request rate will result in over-allocation and cost overruns for other request rates. Provisioning based on low request rates, in contrast, will lead to SLA violations. Thus, the resource allocation should be modulated at runtime by an auto-scaler based on the workload pattern.

B. ANALYSIS OF POPULARITY OF WORKLOAD PATTERNS

We now analyze whether an application's workload is dominated by a few popular workload patterns. Employing performance analysis terminology, each function in an application represents a class of request. The performance of an application is affected both by the request rate and the workload mix, i.e., the proportions of different request classes. We use a metric called *workload intensity* that captures both these aspects. Workload intensity of an application is defined as a vector where each entry corresponds to the number of requests belonging to a specific request class, i.e., number of invocations of a specific application function. We identify frequently observed ranges of workload intensities during a given day. We then analyze the fraction of intensities observed in the subsequent day that fall into these popular ranges.

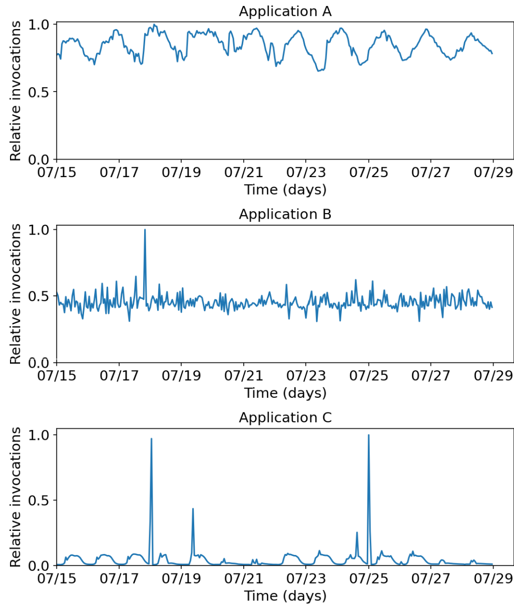

FIGURE 1. Two weeks, invocations per hour for three applications.

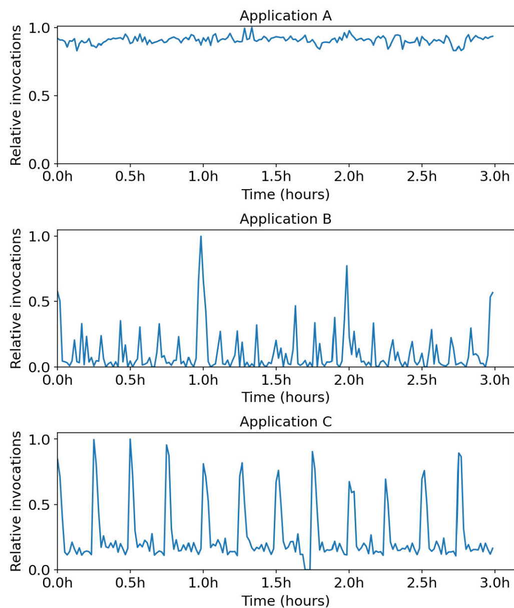

FIGURE 2. First hours of 07/15, invocations per minute for three applications.

As discussed previously, a higher fraction simplifies the design of a CD compatible auto-scaler. Sec. III-B1 formalizes our analysis procedure. The results of the analysis are presented in Sec. III-B2.

1) FORMULATION OF ANALYSIS

The intensity of workloads an application faces can be analyzed at different timescales. We consider a time period T_j to characterize workload intensities. Within this interval, we characterize workload intensities at finer, equal-sized intervals. The ith such interval within T_j is denoted as $T_{i,j}$. The workload intensity for an application over the period $T_{i,j}$ is represented as $\overrightarrow{W_{i,j}}$. It is a vector with length equal to the number of request classes in that application. Each element of the vector presents the number of requests in that fixed interval for one of the request classes. Specifically, for an application with C request classes, $\overrightarrow{W_{i,j}}$ is a C dimensional vector ($|\overrightarrow{W_{i,j}}| = C$). $\overrightarrow{W_{i,j}} = (w^1_{i,j}, w^2_{i,j}, \ldots, w^C_{i,j})$ and each $w^c_{i,j}$ is the number of invocations of the cth request class during $T_{i,j}$. Each application might face various workload intensities at the sub-intervals $T_{i,j}$ within T_j. The set of all workload intensities observed over T_j is represented as $\mathbf{W}_j$.

We define *intensity range* to group similar workload intensities together. An intensity range is a bin in the C dimensional histogram of all $\{\overrightarrow{W_{i,j}} \forall i \in \mathbf{W}_j\}$. As with a single dimensional histogram, several strategies can be used to find the bin size. We use the well-known Freedman–Diaconis rule [44] to find the bin size. This leads to a set of C dimensional bins, denoted as $\mathbf{B}_j$, extracted from $\mathbf{W}_j$. The nth bin in $\mathbf{B}_j$ is denoted as $b_{n,j}$. An intensity range can encompass various workload intensities. A workload intensity $\overrightarrow{W_{i,j}}$ falls into an intensity range if and only if Eq. 1 holds. The equation specifies that the number of requests belonging to any given request class must be within the lower and upper bounds for that class as defined by the bin.

$$l^c_{n,j} \leq w^c_{i,j} < h^c_{n,j} \forall c = 1, 2, \ldots C \tag{1}$$

We now define the popularity of intensity ranges. Each intensity range $b_{n,j}$ has a corresponding frequency $f_{n,j}$ indicating how many of the $\{\overrightarrow{W_{i,j}} \forall i \in \mathbf{W}_j\}$ vectors fall into bin $b_{n,j}$. We define the $\mathbf{Top}^{count}_j$ as the top *count* bins in $\mathbf{B}_j$ with the highest $f_{n,j}$ values. To quantify the combined popularity of a set of bins, we define a metric called coverage. Coverage represents the percentage of the total number of observed workload intensities that fall into a specified set of intensity ranges. Eq. 2 shows how the coverage of a set of popular intensity ranges (Z) on the interval j is calculated. In this calculation, $\overrightarrow{W_{i,j}} \in b$ is evaluated using Eq. 1 by considering the lower and upper bounds for each dimension, i.e., the l^c and h^c values, respectively. A large value of $coverage(j, Z)$ indicates that the Z bins capture most of the workload intensities observed during the time period T_j.

We also characterize how the popularity of intensity ranges in a time period T_j changes during the subsequent time period T_{j+1}. Specifically, we calculate the intensity ranges $\mathbf{B}_j$ based on the workload intensities $\mathbf{W}_j$ observed during T_j. We can then calculate $coverage(j+1, Z)$, i.e., the fraction of workload intensities observed in T_{j+1} that are captured by the subset Z of popular ranges in the previous time period T_j. Gaining such information about future workload intensities can be exploited for better resource allocations during the auto-scaling process.

$$coverage(j, Z) = \frac{|\{\overrightarrow{W_{i,j}} \forall i \in \mathbf{W}_j | \exists b \in Z : \overrightarrow{W_{i,j}} \in b\}|}{|\{\overrightarrow{W_{i,j}} \forall i \in \mathbf{W}_j\}|} \times 100 \tag{2}$$

2) ANALYSIS RESULTS

We apply the process defined previously on each application. We consider T_j as 24 hours. Since the workload intensities are provided in 1-minute bins, we consider $T_{i,j}$ as 1-minute intervals. Since the dataset contains data from 14 consecutive days, there are 13 pairs of consecutive days. In each pair of days, we use the first day to find the $\mathbf{Top}_j^{count}$ popular ranges and compute the coverage of these ranges for both the same day and the next day. Consider a scenario where $\mathbf{Top}_j^5$ has high coverage for the next day. This indicates that an auto-scaler can profile the system for just the top 5 workloads and use the resource allocations inferred for these workloads to drive the scaling over the next 24 hours.

Since the results of this analysis are used for designing an auto-scaler for microservice applications, we filter out some applications and functions. The filtering rules we use are as follows:

- Functions that are invoked using a *Timer* trigger are ignored. The predefined behaviour of invocation patterns of these functions is in contrast with the unpredictable workload of microservice applications. These functions may not benefit from auto-scaling as resources can be assigned to them based on the timing of invocations.
- On each pair of consecutive days, applications that do not use the same set of functions are not considered in this analysis. The process defined in the previous section needs the same functions, i.e., request classes, in both T_j and T_{j+1}. In the 13 pairs of consecutive days, on average 82% of the applications have the same set of functions and hence 18% of applications are filtered out.

We run the process on the subset of applications that are invoked at least once per minute on average, i.e., we exclude applications with very light workloads. We remove these applications since they are not likely to benefit from auto-scaling. We call the retained subset *popular applications*. There are 2, 130 popular applications. The average number of functions hosted per popular application is 2.7.

Table 2 shows the coverage values for the applications considered. The first column shows different *count* values in $\mathbf{Top}_j^{count}$ indicating the number of popular intensity ranges considered when computing the coverage. The second column represents the coverage of the $\mathbf{Top}_j^{count}$ on the T_j, i.e., coverage of the top intensity ranges on the same day. The values in this column are computed using $coverage(j, \mathbf{Top}_j^{count})$ based on Eq. 2. The last column is the coverage of $\mathbf{Top}_j^{count}$ on the T_{j+1}, computed using $coverage(j+1, \mathbf{Top}_{j+1}^{count})$.

The average number of unique workload intensity ranges observed for any given application is about 10^{100}. The average is skewed by certain applications having more than 1, 000 functions thereby resulting in extremely large number of histogram bins. Considering the median to avoid this skew, the median numbers of unique intensity ranges is 165.0 for the applications considered. From Table 2, by considering only the most popular intensity range of all unique ranges it is possible to achieve 51% coverage on the next day. Considering the top 5 ranges, one can obtain a coverage of 75% for the subsequent day. As the tables shows, having more intensity ranges increases the coverage. However, the rate at which coverage increases diminishes as the number of intensity ranges is increased. We find similar results when we expand the analysis to include the less popular applications in the dataset.

TABLE 2. Coverage on the same day and the next day.

Count	Coverage - same day	Coverage - next day
1	52%	51%
2	63%	61%
3	70%	68%
4	74%	72%
5	78%	75%
6	80%	78%
7	83%	80%
8	84%	82%
9	86%	83%
10	87%	84%

Summarizing, our results show that workload intensity ranges have a very skewed popularity distribution. Specifically, a small number of workload intensity ranges encompass a large fraction of workload intensities observed in a day. Furthermore, these popular workloads exhibit good temporal locality since they are likely to occur frequently in the subsequent day as well. This behaviour of applications can be exploited to design an auto-scaler that can perform a similar analysis and find the frequent intensity ranges. We can quickly find optimal or near-optimal resource allocations for these popular intensity ranges during a profiling phase. The auto-scaler can apply these allocations at runtime on the subsequent day whenever the system faces a workload intensity similar to one of the workload intensities in the popular set. Since the profiling phase has to only investigate the popular intensity ranges, the profiling time is small thereby making the auto-scaler CD compatible. We describe the design of an auto-scaler that embodies these principles in the next section.

We note that the performance of a function might change depending on the parameters passed to it. Since the traces do not include parameters, we are not able to characterize the distributions of parameter values. While profiling the popular intensity ranges, care must be exercised to ensure that the distributions of function parameter values used during profiling matches those observed in the real deployment.

IV. TRIM: TRACE DRIVEN AUTO-SCALER

A. OVERVIEW

The key idea that TRIM exploits is that future workload behaviours are expected to be similar to recent behaviours. To act on this idea, TRIM exploits workload trace data from a recent time period. Fig. 3 shows the high level architecture of TRIM. TRIM collects workload trace data from a deployed system through end point monitoring. During a profiling phase conducted in a sandbox environment, it analyzes the

traces using the methodology proposed in Sec. III to identify the most frequent workload intensity ranges. For each of these ranges, it uses a resource allocation module called MOAT that we developed. MOAT computes resource allocations to individual microservices such that operator specified response time targets for request classes are satisfied. The resource allocation plans computed by MOAT are stored in a repository for use in auto-scaling.

During the subsequent time period, TRIM reuses the resource allocation plans computed using the workload intensity ranges of the previous time period to perform auto-scaling. TRIM measures the workload intensity at any given time instant using end point monitoring. It uses this information to perform reactive auto-scaling. Specifically, it checks whether the current workload intensity falls into any of the ranges present in the resource allocation repository. If there is a match, TRIM looks up the corresponding resource allocations for the individual microservices. It then applies these resource allocations to the system. If there is no match, TRIM falls back on a default, rule-based auto-scaling algorithm.

The design of TRIM is flexible to accommodate changes. For example, although we have implemented reactive auto-scaling, the design can permit proactive auto-scaling as well. This only requires a workload prediction engine [11] that can predict in advance the workload intensity the system is likely to encounter at a future time instant. Furthermore, resource allocation engines other than MOAT can be accommodated. It is also straightforward to specify different default auto-scaling algorithms to handle workloads not captured by the popular intensity ranges. TRIM allows the operator to specify the period over which workload analysis is performed to identify the popular workload intensities. It also employs best practices [13] to prevent issues such as auto-scaler oscillation and to control other aspects of auto-scaler behaviour. Since the profiling phase explores only a handful of workloads, it can be repeated quickly whenever the system is updated, making it well-suited for microservice applications.

B. DESIGN

TRIM has two primary states. The first profiling state can happen under two scenarios. In scenario 1, the system has reached the end of T_j, e.g., end of a day assuming the workload is analyzed over a day. Scenario 2 happens when the system is updated. Since both scenarios are handled conceptually similarly, we only describe scenario 1. The profiling state is depicted by the dashed box in Fig. 3. In this step, TRIM finds a new set of popular workload intensity ranges $\mathbf{Top}_j^{count}$ based on $\mathbf{W}_j$ using the process introduced in Sec. III to analyze workload traces collected from the system (step P1). Each element of $\mathbf{Top}_j^{count}$ is an intensity range $b_{n,j}$ that specifies the range for each request class c in the application as defined by the lower bound $l_{n,j}^c$ and the upper bound $h_{n,j}^c$. Then, $\mathbf{Top}_j^{count}$ is passed to the MOAT module (step P2). By running performance tests on a sandbox environment, MOAT obtains resource allocations for each intensity range

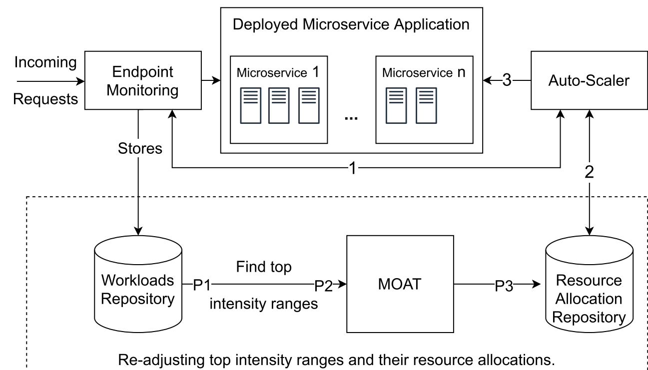

FIGURE 3. Overall approach of TRIM.

and stores them as $\mathcal{A}_{n,j} \forall b_{n,j} \in \mathbf{Top}_j^{count}$ in a resource allocations repository (step P3). As a result, each $b_{n,j} \in \mathbf{Top}_j^{count}$ has a corresponding resource allocation denoted by $\mathcal{A}_{n,j}$ that specifies resource allocation for each microservice when the application is facing a workload intensity that falls in $b_{n,j}$. TRIM uses $\mathcal{A}_{n,j}$ for scaling the application during T_{j+1}.

The second state represents TRIM scaling the current deployment of the system. This is shown outside the dashed box in Fig. 3. Inputs for this state are outputs of the profiling phase. An additional input I represents the time in seconds between TRIM's successive measurements of the system. It specifies how frequently auto-scaling is attempted. Assume the system is in the beginning of T_{j+1}. The incoming workload intensity at instant i $\overrightarrow{W_{i,j+1}}$ is measured (step 1). TRIM checks if it falls into one of $b_{n,j} \in \mathbf{Top}_j^{count}$ calculated in state 1 (step 2). Then, TRIM updates the resource allocations based on pre-computed values for this range using the corresponding resource allocation $\mathcal{A}_{n,j}$ (step 3). If the current workload does not fit into any of the ranges, TRIM scales the system based on a simple rule-based approach (step 3). In this work, we use the industry-standard HPA auto-scaler [13] as the default.

MOAT is in charge of deciding about the amount of resources allocated to each microservice for each intensity range ($\mathcal{A}_{n,j} \forall b_{n,j} \in \mathbf{Top}_j^{count}$). MOAT needs to consider request class response time targets specified by the operator while performing resource allocation. Another feature this module needs to satisfy is to offer quick resource allocation estimates to enable use within CI-CD pipelines. Techniques that require long model training or manual hyperparameter tuning are inconsistent with this requirement. Furthermore, the resource allocations discovered by MOAT need to be efficient, i.e., MOAT should allocate just the right amount of resources to each microservice. In the next section, we describe how MOAT addresses these requirements.

V. MOAT: RESOURCE ALLOCATION MODULE

A. OVERVIEW

MOAT uses a sandbox performance testing environment containing the application to be deployed. It considers as input a

popular workload intensity range that has been identified by TRIM through trace analysis. It also takes as input operator specified response time targets for individual request classes. MOAT employs an iterative performance testing approach to quickly infer cost-effective microservice container resource allocations that satisfy these response time targets. MOAT first generates initial resource allocations, e.g., CPU shares, for individual microservices and launches a performance test. It then refines the resource allocation incrementally by monitoring service response times and resource usage during this test and by exploiting knowledge of the application's service architecture. MOAT determines resource allocations without resorting to a large number of performance tests thereby speeding up the tuning process. It employs heuristics to reduce over-allocation of resources leading to efficient allocations. As described in Sec. IV, TRIM invokes MOAT for each popular workload intensity range and uses the resulting resource allocations at runtime.

B. DESIGN

Figure 4 depicts the overall design of MOAT. It also shows the steps involved in each iteration. MOAT deploys the application on an on-premise or cloud platform attached to the sandbox environment (step 1) and starts the *Load Generator* (step 2). While the performance test runs, the *Data Aggregator* gathers different measurements (step 3). The Aggregated Data is passed on to a *Tuning Agent* (step 4). The *Tuning Agent* uses this information to estimate new resource allocations for each microservice. These configurations are then stored in the *Resource Allocation Database* (step 5). The *Orchestrator Connector* deploys the application using this configuration to initiate another iteration (step 6). The iterations continue until a configuration that achieves all response time targets is identified or an operator-specified upper limit on the number of iterations is reached. We next describe the individual components of MOAT.

1) ORCHESTRATOR CONNECTOR

This module is responsible for starting a new deployment and altering the existing deployment for subsequent iterations. It is designed to work with both Docker Swarm and Kubernetes [1] container orchestration systems. It is also possible to add support for other container orchestration systems.

2) LOAD GENERATOR

This module provides the ability to emulate the desired workload intensity on the system. It also manages the details of the load test such as the request class mix and a warm-up period. It executes on a dedicated physical machine.

3) DATA AGGREGATORS

There are different types of raw measurements that MOAT can exploit. Each of these measurements might be available through various sources as microservice applications can be deployed differently with a wide range of available tools.

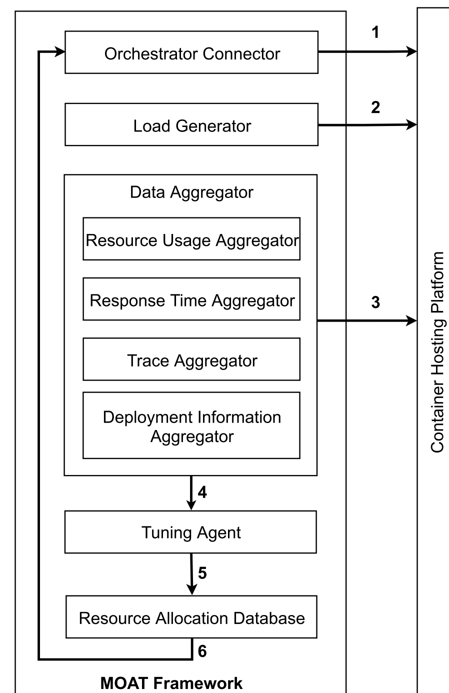

FIGURE 4. Overall design of MOAT.

MOAT focuses on commonly available data sources that are usually available from cloud providers or are possible to record by ready-to-use tools. The data from these sources sometimes need additional parsing and cleaning, which is also done by the *Data Aggregator*.

4) TUNING AGENT

This module uses the aggregated data that is passed to it and the algorithm that is chosen to suggest a resource allocation for the next iteration. In this research, we have developed two tuning algorithms, which we describe in Section V-C.

5) RESOURCE ALLOCATION DATABASE

The tuned resource allocations are stored in a database for further use by TRIM. TRIM can query this database to choose an appropriate resource allocation for the incoming workload at runtime.

We have designed MOAT to be modular and easily adaptable by others. The source code has been made public [40]. The implementation can be used to tune any microservice application with the appropriate configuration file to control MOAT. Our design allows the integration of a load generator customized for any given application. Furthermore, any module that accepts the sequence of service calls, i.e., list of microservices involved to satisfy each request class, request class response time percentile targets, and mean resource utilization as input and generates container resource allocations as output can be integrated as the *Tuning Agent*. We next describe our two different implementations of the tuning algorithm.

C. TUNING ALGORITHMS

This section is organized as follows. Sec. V-C1 formulates the tuning problem. Sec. V-C2 explains the details of the basic version of our algorithm. We identify shortcomings of this algorithm and address them in an enhanced version, which is described in Sec. V-C3.

1) FORMULATING THE TUNING PROBLEM

Performance tuning can be formulated as an optimization problem. MOAT determines resource allocations for a specific workload intensity $\overrightarrow{W}$. However, the trace analysis process described in Sec. III that is leveraged by TRIM yields popular workload intensity ranges. Hence, each workload intensity range needs to be first converted to a workload intensity for use by MOAT. We take an approach of using the heaviest workload intensity in a given range as a representative of that range. Assuming $b_{n,j}$ is the intensity range that needs to be configured by MOAT, the heaviest workload intensity can be constructed as $\overrightarrow{W} = (h^1_{n,j}, h^2_{n,j}, \ldots, h^C_{n,j})$, where $h^c_{n,j}$ is the upper bound of invocation count for request class c as defined in Sec. III. None of the workload intensities in $b_{n,j}$ can impose heavier stress on the system than this choice. Hence, a resource allocation that satisfies response time targets for this choice will automatically satisfy the targets for all intensities in the range.

Our tuning approach can accommodate multiple types of resources. However, we focus on CPU for two main reasons. First, all the on-premise and public cloud microservice applications we consider are CPU intensive. Second, focusing only on CPU simplifies the discussions of the problem formulation and the tuning algorithms. To establish the generalizability of the approach, we have explored in Appendix A an implementation of MOAT that considers both CPU and memory resource types.

We consider an application with K microservices, each packaged in its own container. The application supports C distinct request classes. Each request class c has an associated response time target RT_c. This target is typically defined as a percentile, e.g., 95th percentile, of response time values observed while servicing the workload $\overrightarrow{W}$. Each request class c invokes a sequence of microservices S_c to get its response. Thus, each service k in this sequence incurs a CPU demand. The amount of CPU resources, i.e., CPU shares, allocated to microservice k is denoted as α_k and $\mathcal{A} = \{\alpha_k : \forall k \in K\}$ is resource allocation for all microservices.

The tuning problem translates to determining the minimum CPU shares allocated to each service such that all response time targets are met. Specifically, Eq. 3 minimizes the cumulative CPU shares allocated to the application and hence the cost of operating the application on the cloud. Eq. 4 stipulates that just enough resources need to be allocated such that the request class's response time R_c is less than or equal to the corresponding target RT_c for all $c \in C$. Eq. 5 shows that the response time R_c is a function of the workload and the K CPU share values captured by $\mathcal{A}$. Eq. 6 incorporates an optional constraint that places an upper bound α_{max} on the total CPU shares available to the application. While many approaches can be used to solve this optimization problem, we need to focus on methods that can identify solutions quickly to support CD practices and to reduce the cost of renting the resources needed for the tuning.

$$\min \sum_{k \in K} \alpha_k \tag{3}$$

$$\text{s.t.} \quad R_c \leq RT_c \; \forall \, c \in C \tag{4}$$

$$\text{s.t.} \; R_c = f_c(W, \mathcal{A}) \; \forall \, c \in C \tag{5}$$

$$\text{s.t.} \sum_{k \in K} \alpha_k < \alpha_{max} \tag{6}$$

2) BASIC TUNING ALGORITHM

We developed the basic tuning algorithm presented in this section as preliminary work [14]. We summarize it here to aid explanations of our subsequent enhancements to the algorithm. The key idea of the basic tuning algorithm is to incrementally alleviate the resource bottlenecks experienced by each request class that does not meet its response time target. For each request class c that violates its target, we identify the service in its service sequence S_c that is utilized the most. Additional CPU shares are allocated to the services identified in this manner and the effect of these updated allocations is measured iteratively through further performance tests. An iterative approach allows the algorithm to closely track how bottlenecks shift across services so that allocations can be tailored accordingly.

Algorithm 1 describes the basic approach in detail. The algorithm takes as input Δ, which specifies the incremental amount of CPU shares to be added to each microservice when it is detected to be the bottleneck, i.e., has the highest CPU utilization, for a request class. It also takes as input an initial configuration, i.e., initial CPU shares, for all microservices (represented as $\alpha_k^{start} \forall k \in K$) and S_c, the service sequence. As outlined previously, the algorithm refines these initial shares iteratively using performance tests by continuously tracking bottlenecks. When all response time targets are met, the algorithm breaks out of the loop and reports the last employed CPU shares as the output.

In the main loop of Algorithm 1 at line 2, different approaches can be taken to deploy a configuration. One option is to allocate all resources assigned to each microservice to a single container. Another option is to split the resources between multiple container instances. With both approaches, the amount of CPU shares allocated to microservice k is equal to α_k.

We now focus on the factors affecting the time taken for the tuning process. The time spent by Algorithm 1 generating the CPU shares is negligible compared to the time taken to deploy the application and conduct the performance test. This motivates the need to cut down on the number of performance tests by reducing the number of iterations. Using a small Δ value allows the algorithm to track bottleneck shifts in a

fine-grained manner. This leads to more efficiency in terms of the total CPU shares allocated to services. However, this may require a large number of iterations. On the other hand, a larger Δ value leads to finding the final configuration faster but often leads to over-provisioning. Similar to the problem of choosing the learning rate in stochastic gradient descent [45], finding the perfect Δ value may be challenging. The other issue with Algorithm 1 is that a single microservice might be involved in different request classes and be the bottleneck for all of them. If we add Δ CPU shares for each request class to this microservice (line 12 of Algorithm 1), we might end up over-provisioning this service. We next refine the basic algorithm to address these issues thereby achieving both fast convergence and efficient allocations.

Algorithm 1 Basic Algorithm

```
   Result: A (α_k ∀k ∈ K)
   Input: initial CPUshares(α_k^start ∀k ∈ K), Δ, S_c ∀c ∈ C
1  while True do
2  |  Deploy the system with the current configuration
3  |  Run the performance test and gather:
4  |  - CPU utilization of all services
5  |  - Response times of all request classes (R_c)
6  |  if R_c ≤ RT_c ∀c ∈ C then
7  |  |  break
8  |  end
9  |  for c ∈ C do
10 |  |  if R_c > RT_c then
11 |  |  |  bottleneck ← bottleneck of S_c
12 |  |  |  add Δ CPU shares to bottleneck
13 |  |  end
14 |  end
15 end
```

Algorithm 2 Enhanced Algorithm

```
   Result: A (α_k ∀k ∈ K)
   Input: initial CPUshares(α_k^start ∀k ∈
          K), Δ, Δ_max, S_c ∀c ∈ C
1  for c ∈ C do
2  |  Δ_c ← Δ
3  end
4  while True do
5  |  Deploy the system with the current configuration
6  |  Run the performance test and gather:
7  |  - CPU utilization of all services
8  |  - Response times of all request classes (R_c)
9  |  if R_c < RT_c ∀c ∈ C then
10 |  |  pruneStageFlag ← True
11 |  end
12 |  for c ∈ C do
13 |  |  if R_c > RT_c then
14 |  |  |  bottleneck ← bottleneck of S_c
15 |  |  |  add Δ_c CPU shares to bottleneck
16 |  |  end
17 |  end
18 |  if pruneStageFlag then
19 |  |  K_prune ← choosePruningMicroservice()
20 |  |  Δ_c ← Δ_c/2
21 |  |  reduce Δ_c CPU shares from K_prune
22 |  end
23 |  for k ∈ K do
24 |  |  don't allow α_k to increase more than Δ_max in
25 |  |  a single iteration
26 |  end
27 |  if stopCondition() then
28 |  |  return best tried configuration and break
29 |  end
30 end
```

3) ENHANCED TUNING ALGORITHM

The enhanced algorithm simultaneously achieves fast convergence and efficiency by adaptively changing the incremental CPU share Δ. It initially employs a large value of Δ to quickly obtain a valid but over-provisioned solution. It then embarks on a pruning phase where allocations to selected services are reduced to offset the over-provisioning. As described shortly, several heuristics are used to drive this selection. Inspired by binary search, the Δ during the pruning phase, i.e., the reduction in CPU share, is progressively halved. If the pruning process causes response time targets to be violated again, the algorithm heals itself through additional allocations using the reduced Δ value. The use of smaller Δ values during pruning and healing prevents instability, which can delay convergence. The use of adaptive Δ allows the enhanced algorithm to reap the efficiency of employing the basic algorithm with a small Δ while also achieving the quick convergence of using the basic algorithm with a large Δ.

Algorithm 2 explains our enhanced approach. The main inputs for the algorithm are $\mathcal{A}$ the initial CPU shares $\alpha_k^{start} \forall k \in K$, the service sequence S_c for each request class c as well as Δ and Δ_{max}, which is the largest permissible increase in CPU share in a single iteration for any given service. Services that handle request classes whose response times are much lower than targets need more pruning than others. Accordingly, Δ has to be adapted for individual request classes. Consequently, the algorithm employs request class specific Δ_c values, all initialized to Δ (line 2).

Lines 23-26 address the basic algorithm's tendency to allocate too many resources to a service that is a bottleneck for multiple request classes. Specifically, the additional shares allocated to any given service in a single iteration is limited to Δ_{max}.

With the exception of enforcing Δ_{max}, the algorithm initially uses the basic approach until it realizes a configuration where all targets are met. It then enters the pruning phase by setting the *pruneStageFlag* to *True* (line 10). From this point,

we prune resource allocations at the end of every iteration (lines 18-22). If response time targets are violated due to the pruning in the previous iteration, more resources are allocated to the appropriate bottleneck services in the usual manner albeit with the updated Δ_c values (lines 12-17).

Through empirical observations, we identify several heuristics to select a candidate service K_c to prune for request class c. We validate the effectiveness of these heuristics in Sec. VI. These heuristics are as follows:

1) The candidate service should be the least utilized service in at least one service sequence in the set of C possible service sequences.
2) The candidate service should not be the most utilized service, i.e., the bottleneck, in any of the C service sequences. The reason both conditions 1 and 2 need to be checked is because one service might be the least utilized in one service sequence and the most utilized in another service sequence.
3) The service should not be involved in the service sequence of any request class whose response time is very close to the corresponding target. The closeness of a request class c to its target is specified as a percentage ϵ of RT_c (omitted from Algorithm 2 for clarity). Services whose response times deviate beyond this bound are considered for pruning in this heuristic.

The algorithm calls *stopCondition*() to check if more iterations are needed. It stops and returns the previously encountered configuration with the least cumulative CPU shares if any of the following conditions are met. The stopping conditions are as follows:[1]

1) The number of iterations in the pruning stage passes the number of iterations before the pruning stage started.
2) The algorithm wants to evaluate a configuration that has been tried previously.
3) The sum of Δ_c values falls below a pre-defined threshold (Δ_{stop}) indicating that the pruning has become too fine-grained to yield further improvements.
4) The cumulative CPU share budget has been reached.
5) Number of iterations reaches the operator-specified upper limit.

We now analyze of the time complexity of the enhanced algorithm. Considering the inputs of I, C, and K, the time complexity of the enhanced algorithm is $\mathcal{O}(I(C+K\log(K))$. The logarithmic term occurs due to the binary search technique used while adding resources. The linear and logarithmic terms are known given the inputs. However, we can only evaluate the value of the number of iterations I with heuristic analysis. Results that will be presented in subsequent sections covering a range of systems shows that the number of iterations needed by the algorithm is much lower than that needed by a state of the art search technique. Thus, the linear and logarithmic terms in combination with our empirically observed small values of I attest to the quickness of MOAT as a tuning technique.

[1] These are not shown in Algorithm 2 for the sake of clarity.

VI. EVALUATION OF MOAT

A. EVALUATION SETUP

We evaluate MOAT on analytical and experimental systems. The analytical systems helps ascertain how close the solutions of MOAT are to the optimal solutions. The on-premise system allows us to evaluate MOAT with realistic microservice applications. The public cloud system helps us generalize the applicability of MOAT to other cloud platforms.

The internal parameters Δ_{stop} and ϵ of the enhanced algorithm are set to 0.2 and 90, respectively. Also, Δ_{max} in the enhanced algorithm is equal to Δ in all experiments. When using Algorithm 1 and Algorithm 2, we split α_k equally between multiple container instances. If microservice k started the process with the initial configuration α_k^{start}, then the number of instances created based on α_k is $\lceil \frac{\alpha_k}{\alpha_k^{start}} \rceil$. This approach ensures that each service is scaled by merely adding more instances. It does not require the resource allocation of existing instances in the tuning process to be modified. The values of α_k^{start} are specified for each experiment separately.

We compare our approach with an existing Multi Objective Bayesian Optimizer solver [46]. In the rest of this section, this approach is referred as BO (Bayesian Optimizer). We configure the BO solver to minimize the total CPU shares while ensuring all response time targets are met. The BO solver launches a sequence of performance tests in an attempt to estimate the optimal CPU shares. It uses the measured response times from tests to guide its search in the tuning space. We set the internal parameters of the solver to their default values.

1) ANALYTICAL SYSTEMS

A key objective of this study is to evaluate how MOAT's resource allocations compare to optimal resource allocations. To achieve this objective, we require a system whose performance behaviour can be expressed using closed form analytical expressions. Such a system would allow us to determine the optimal resource allocations using analytic optimization solvers. Consequently, we consider analytical systems whose behaviour can be captured using product form open QNMs [47]. Each service k is represented as a single server queue. The CPU service demand placed by request-class c on service k when the service is allocated a CPU share $\alpha_k = 1.0$ is denoted by $D_{c,k}$. The service demand decreases linearly with an increase in CPU share. The workload intensity W is represented as a vector of the mean rate of arrivals for each request class with λ_c denoting the arrival rate in requests per second (rps) for request-class c. Given these assumptions, Eq. 7 and Eq. 8 can be used to calculate the mean utilization and mean response time of request classes, respectively. Eq. 7 and 8 can be used with Eq. 3, 4, and 6 to define a non-linear optimization problem.

$$U_k = \sum_{c \in C} \frac{\lambda_c D_{c,k}}{\alpha_k} \ \forall k \in K \tag{7}$$

TABLE 3. Properties of randomly generated systems.

K	C	T (ms)	Number of systems
10	12	300	100
10	25	450	37
20	30	300	44
20	30	500	45
30	40	300	42
30	90	600	13
40	90	1000	19

$$R_c = \sum_{k \in K} \frac{\frac{D_{c,k}}{\alpha_k}}{1 - U_k} \; \forall c \in C \tag{8}$$

We conduct an extensive validation study using randomly generated microservice systems with different numbers of request classes and services with randomly generated workload intensities. Since Eq. 8 estimates mean response times, we consider mean response time targets for these systems. We ensure that the generated systems are stable, i.e., utilizations do not exceed 1.0. To achieve stability, we first randomly generate utilization (between 0.0 and 1.0), demand, and arrival rate values. The demands and arrival rates are then scaled to ensure Eq. 7 holds. We additionally constrain the minimum α_k value to 1.0 in experiments with this system to avoid negative response times. The systems generated in this manner are both stable as well as random. We set $\alpha_k^{start} = 1.0 \forall k \in K$ in this experiment.

Table 3 provides details on the 300 random systems we consider. We use the same response time target T for all request classes in any given system. We use AMPL [48] and CONOPT [49], [50] solvers to obtain the solution for the optimization problem corresponding to each randomly generated system. This solution is then compared with the iterative solutions identified by the basic and enhanced algorithms. For our algorithms, we also keep track of the number of iterations needed to obtain the final solutions.

2) EXPERIMENTAL SYSTEMS

We expand the validation to include an on-premise system and a public cloud system. We note that the profiling environment used by MOAT must match the environment where the microservices application will be deployed. Hence, we conduct separate sets of profiling experiments for both the on-premise and public cloud systems. We also note that a dedicated profiling environment is not needed. The profiling environment can be leased when needed and released when the profiling is done.

In the on-premise system, the host machine has Ubuntu 16.04.6 LT as the operating system. It has an Intel(R) Xeon(R) CPU E5645@2.140GHz with 24 cores and 64 GB of memory. The Docker version on the host machine is 16.03.5. For generating loads, K6 v0.26.2 [51] is used. It is installed on a different machine with similar characteristics but on the same network. The load generator machine and the host machine have a dedicated 1 Gigabit/sec connection between them. All the microservice applications are developed using Node.js v13.7.0. We note that we do not explore core affinity, i.e., pinning containers to specific cores, in our experiments to prevent under utilization of cores. However, MOAT could also be used to configure systems where pinning is employed.

To evaluate our approach on a public cloud platform, we conduct a larger scale experiment on AWS Fargate. AWS Fargate supports on-demand use of containers. Customers are charged based on the amount of resources (CPU shares and memory) used per minute by their containers. AWS Fargate allows customers to provision container resources in a fine-grained manner, e.g., 0.25 CPU shares. This feature can be exploited by tools such as MOAT. We use Amazon Elastic Kubernetes Service (EKS) as the container orchestration framework. Kubernetes manages the containers and uses AWS Fargate to instantiate containers when needed.

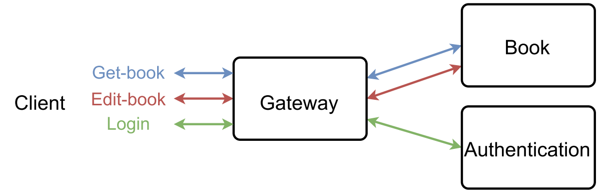

FIGURE 5. Bookstore with 3 request classes. Login is green, Edit-Book is red, Get-Book is blue.

We consider the Bookstore microservice application developed by us as shown in Fig. 5. This application models an online store that supports managing and selling an inventory of books. It contains three microservices and supports three request types. Response time targets are defined based on the 95th percentile of response times. All request classes have the same threshold of $250ms$. We consider four distinct workload intensities with the same request class mix but different request arrival rates, $\lambda = 75, 100, 125, 150$ requests per second. The chosen arrival rates allowed us to observe the behaviour of MOAT from low to high load conditions. Furthermore, the value of α_k^{start} is set to 1.0.

Each performance test has a 15 s warmup period. Post warmup, each test is run long enough to ensure that the 95th percentile of response time for any request class is contained within its 90% confidence interval. With this criterion, we needed to run the Bookstore tests for 1 minute. We consider response time and utilization metrics only from the post warmup period.

B. RESULTS

1) ANALYTICAL SYSTEMS

Fig. 6 shows for the basic algorithm the number of iterations for convergence and the percentage of CPU share over-allocation with respect to the AMPL solver's theoretical solution. Specifically, we plot the mean values of these metrics achieved over the 300 systems we evaluate. Regardless of the Δ settings, the basic algorithm's solution is always less efficient than the optimal AMPL solution. However, with

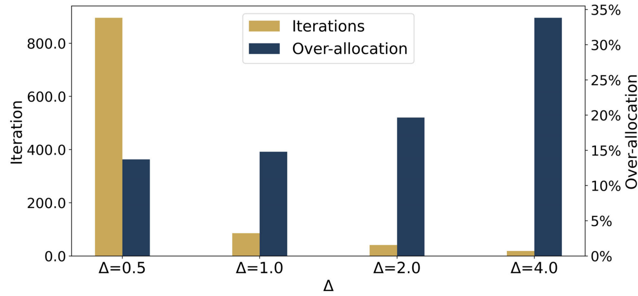

FIGURE 6. Performance of the basic algorithm.

low Δ values, the basic algorithm identifies near-minimal solutions. For example, it only over-allocates CPU shares by approximately 15% when $\Delta = 0.5$. As expected, the number of iterations drops with increasing Δ but the solutions become less efficient.

The enhanced algorithm significantly outperforms the basic algorithm by quickly identifying solutions that are closer to the optimal solutions obtained from AMPL. Table 4 shows the mean number of iterations and mean over-allocation based on all 300 systems. The lowest values of over-allocation and number of iterations are indicated in bold. The enhanced algorithm outperforms the basic variants with $\Delta = 0.5$, 1.0, and 2.0 with respect to *both* the number of iterations and over-allocation. The basic algorithm converges quicker with $\Delta = 4.0$ but provisions more than twice the CPU shares as that of the enhanced algorithm. The enhanced algorithm's solution on average provisions 13.5% more resources than the optimal AMPL solution. However, it has better real-world applicability since it only relies on commonly available measurements from performance tests and does not require mathematical system models. The behaviour of the basic and enhanced algorithms are similar for the experimental systems we explore, Hence, we do not present the results pertaining to the basic algorithm.

We next perform an external comparison of the enhanced algorithm with BO. We configure BO with five initial iterations for it to obtain a random sample of the tuning space. After the initial iterations, to make the comparison fair to BO, we configure BO to execute for the same number of iterations as the enhanced algorithm with $\Delta = 4.0$. For 140 out of 300 systems, BO is unable to find any feasible solutions. For the other 160 systems, BO is unable to find a more efficient solution than the enhanced algorithm. On average, BO suggests 75% more CPU shares than the enhanced algorithm.

BO only uses response times as feedback to its objective function. Consequently, it requires a long time to identify efficient allocations. With time and cost constraints, it typically cannot identify feasible solutions or identifies inefficient solutions. BO-based techniques are hence not appropriate for CI-CD and public cloud environments. In contrast, MOAT exploits knowledge of the application's architecture and commonly available resource utilization metrics to identify efficient allocations quickly.

TABLE 4. Basic vs Enhanced.

	Enhanced	Basic			
	$\Delta = 4.0$	$\Delta = 4.0$	$\Delta = 2.0$	$\Delta = 1.0$	$\Delta = 0.5$
CPU	**13.5%**	33.8%	19.7%	14.8%	13.7%
Iterations	34.6	**18.9**	41.2	85.7	896.0

TABLE 5. Enhanced Algorithm vs. BO, Bookstore.

	Enhanced		BO 1x		BO 2x	
Workload	CPU	Iterations	CPU	Iterations	CPU	Iterations
W_1 ($\lambda = 75$)	4	3	12.1	3	12.1	6
W_2 ($\lambda = 100$)	6	5	-	5	-	10
W_3 ($\lambda = 125$)	8	7	-	7	-	14
W_4 ($\lambda = 150$)	10	7	-	7	-	14

2) EXPERIMENTAL SYSTEMS

Table 5 compares the enhanced algorithm with tuning using the BO solver for the four workload intensities of the Bookstore application. The enhanced algorithm outperforms BO for the on-premise experimental systems as well. After the initial five sampling iterations, we configure BO to explore the tuning space for as many additional iterations as the total number of iterations taken by the enhanced algorithm to converge. This scenario is referred to as BO $1\times$ in the table. From the table, BO $1\times$ is only able to find a feasible solution for W_1. The same behaviour is observed when we double the number of iterations (BO $2\times$). For W_1, BO suggests three times the total CPU share estimated by the enhanced algorithm. BO finds feasible CPU share values for W_2 and W_3 after 14 and 17 iterations, respectively. The total CPU shares identified by BO for W_2 and W_3 exceed those of the enhanced algorithm by 16.7% and 35.0%, respectively. For W_4, BO is unable to identify feasible solutions even after 25 iterations. Thus, BO is typically unable to find feasible solutions even when it iterates more than the enhanced algorithm.

Results from the AWS public cloud Bookstore deployment show similar trends. We limit ourselves to a single experiment to keep costs down. We set $\lambda = 170$ for this experiment. Each load test lasted 90 sec and the 95th percentile of response times for this duration is compared against the threshold for each request class. We set $\alpha_k^{start} = 0.5 \forall k \in K$ in our experiments. Table 6 compares the MOAT's enhanced algorithm with BO. From the table, MOAT requires 4.8 times less iterations than BO. With the pricing scheme of Fargate and EKS, the cost incurred in renting resources for conducting the load tests in the tuning process is 15 times less with MOAT.

Furthermore, as shown in Table 6, the configuration discovered by MOAT is more cost-efficient than the optimal solution identified by BO. Considering a 24-hour time period, it costs 43% less to operate the service using the configuration identified by MOAT. Cost savings from MOAT can be considerable for organizations that deploy a large number of microservice applications on public cloud platforms such as AWS.

TABLE 6. MOAT vs BO on AWS.

Aspect	MOAT	BO
Duration of tuning	**6 iterations**	29 iterations
Cost of tuning	**$0.16**	$2.44
24-hour operational cost	**$14.55**	$25.58

TABLE 7. MOAT's enhanced algorithm parameters.

Parameter	Value
Δ	2.0
Δ_{max}	2.0
Δ_{stop}	0.5
α_{start}	0.5
ϵ	0.9

VII. EVALUATION OF TRIM

A. EXPERIMENT SETUP

The experiments in this section use the same on-premise infrastructure described in Sec. VI. We use the Bookstore application for this study. The application is hosted on a separate system than the one executing TRIM and the load generator.

1) TESTBED

Our objective is to evaluate TRIM's auto-scaling performance when the system faces workload fluctuations. To emulate such a workload, we select from the Azure dataset a workload for a representative application that encounters workload fluctuations.[2] The application supports the same number of request classes as the Bookstore application. We subject the Bookstore application to the workload faced by the selected Azure application on July 16, i.e., the second day in the dataset. We then deploy different auto-scalers to handle this 24 hour workload. The workload intensities recorded in the trace do not stress our on-premise system significantly. Hence, to observe auto-scaling behaviour, we multiply each 1-minute workload intensity vector, i.e., request class invocation counts, by 120. Fig. 7 shows a four-hour snapshot of the scaled workload the Bookstore application faces during a single day. The vertical, gray shadowed strips show the intervals that are covered with one of the $\mathbf{Top}^5$ workload intensities extracted from the previous day, i.e., July 15. TRIM exploits the popular intensity ranges observed in the July 15 workload to drive auto-scaling for the July 16 workload.

2) BASELINE AUTO-SCALER

In these experiments, TRIM is paired with HPA [13] as the default auto-scaler when TRIM encounters a workload intensity not contained by $\mathbf{Top}^{count}$. Furthermore, HPA is used as a standalone baseline to evaluate the impact of using TRIM. HPA depends on a parameter *desiredMetricValue* that specifies how the number of instances should be changed. In this evaluation, we consider mean CPU utilization as the metric value for HPA. HPA periodically re-evaluates the number of container instances of a microservice to keep the mean CPU utilization close to a target specified by *desiredMetricValue*. For example, by setting the *desiredMetricValue* to 50%, HPA tries to change the number of instances to keep the mean CPU utilization as close as possible to 50%. Choosing a lower value as the *desiredMetricValue* leads to allocating more resources, resulting in a higher cost of deployment. However, it reduces the chance of violating response time targets. We use a value of 30% to emulate a scenario where a system operator tries to preempt SLA violations through over allocation. A value of 50% is selected to observe a regime that is more relaxed with respect to SLA violations.

[2] HashApp value in dataset: *cd*05*d*7*b*4445349*ee*645*ea*29058 6*fd*28*c*0*c*675*a*155*eb*1522485535*c*5*c*0329*a*908.

3) MOAT PARAMETERS

We run multiple experiments with various parameters to investigate the performance of TRIM. All configurations that are found by MOAT to be used by TRIM exploit the enhanced algorithm with parameters listed in Table 7. With these values, MOAT's resource allocation for any given microservice consists of one or multiple container instances with each instance having a 0.5 CPU share. To facilitate comparison, HPA also scales horizontally using containers with 0.5 CPU shares.

4) EXPERIMENTS

To understand the benefit of using TRIM, we first run HPA with *desiredMetricValue* = 50% for the 24-hour workload period. This experiment scenario is called HPA-50. Then, we run experiments using the top three, five, and ten workload intensity ranges from the previous day, i.e., $\mathbf{Top}^3$, $\mathbf{Top}^5$, and $\mathbf{Top}^{10}$. In all TRIM runs, in case the measured current workload does not fit into any of the top intensity ranges, we use HPA with *desiredMetricValue* = 50% as the default auto-scaler. These scenarios are called $\mathbf{Top}^3$ + HPA-50, $\mathbf{Top}^5$ + HPA-50, and $\mathbf{Top}^{10}$ + HPA-50. We repeat this process once more using HPA with *desiredMetricValue* = 30% yielding the scenarios HPA-30, $\mathbf{Top}^3$ + HPA-30, $\mathbf{Top}^5$ + HPA-30, and $\mathbf{Top}^{10}$ + HPA-30. We note that the time taken by TRIM to identify popular workload patterns is negligible.

5) OSCILLATION MITIGATION

Scaling decisions are made every s s. To prevent the oscillation of instances, i.e., adding and removing instances too frequently, we use the same technique implemented by the Kubernetes orchestrator [13]. With this technique, a moving window of the last m s where $m > s$ is considered. The auto-scaler records the number of container instances suggested for any given microservice every s s. However, it uses the maximum of these values recorded over the last m s as the actual scaling decision for that service. This strategy reduces oscillations by discouraging premature scale-in operations, which reduce resource allocations. In our experiments, we use $s = 20$ s and $m = 120$ s.

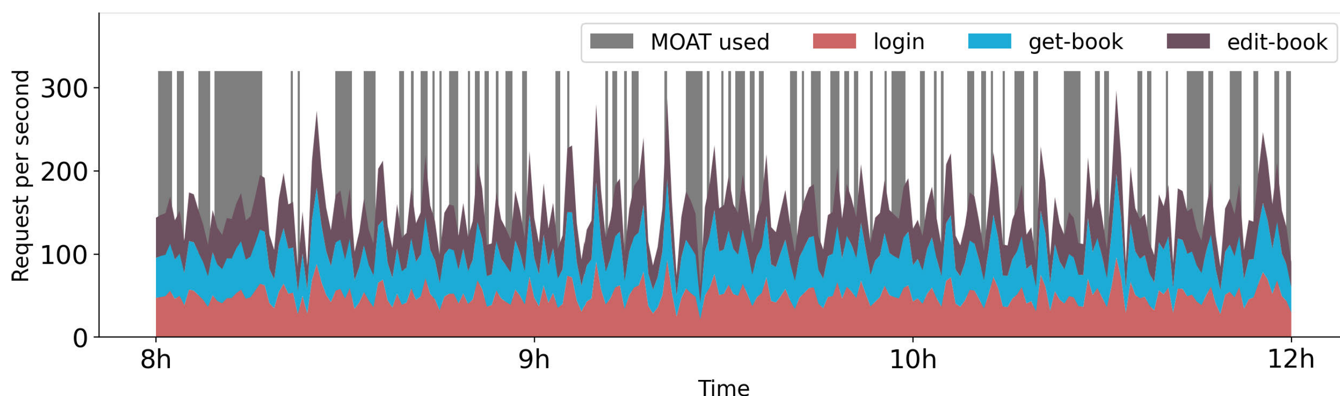

FIGURE 7. 4 hour snapshot of Bookstore workload during the 24 hour test.

6) MEASUREMENTS

In all experiments and scenarios, we monitor total number of instances and the number of SLA, i.e., response time, violations. Since each container instance used for scaling the system has 0.5 CPU shares, we use the total number of instances in the application to compare the efficiency of the various techniques. We track the number of instances used by the application every $s = 20$ s. We report the mean of these numbers over the 24-hour experiment period.

SLAs are defined based on 95th percentile of response times recorded over the past 1 minute. The threshold for the Login, Get-Book, and Edit-Book request classes are 250 ms, 25 ms, and 25 ms, respectively. We count the number of times each request class has violated the response time target and represent the total as *SLA violations*. We note that TRIM allows an operator to specify the granularity of the timescale over which response time targets are measured. Our experiments reflect a scenario where an operator seeks to enforce targets over a very fine timescale of 1 minute. However, TRIM allows coarser timescales, e.g., 5 minutes, to be specified as well.

B. RESULTS

First, we report the time taken by MOAT to estimate the resource allocations for the top 10 workload intensities. MOAT required 34 iterations for these 10 workloads. On average, each iteration consumes 129 s. Hence, determining the resource allocations for the top 10 workloads takes 73 minutes if done sequentially. However, the evaluation of each workload can be done independently in parallel. Thus, in this scenario the total time needed by MOAT equals the time to handle the workload intensity that consumes the most iterations. Considering parallelization, MOAT took less than 13 minutes to estimate the resource allocations. This establishes the feasibility of using our approach within CI-CD pipelines. We note that the profiling step is offline and hence does not disrupt the normal operation of a system. We also note that the time consumed by MOAT is negligible and almost the entire time comprises the time required to execute the underlying performance tests.

Figure 8 compares using only HPA with using TRIM. The figure shows the number of SLA violations, mean number of instances, and mean CPU utilization per instance for HPA in isolation as well as when TRIM is paired with HPA-50 and HPA-30. The coverages obtained with $\mathbf{Top}^3$, $\mathbf{Top}^5$, and $\mathbf{Top}^{10}$ are 33%, 62%, and 83%, respectively. Post experiment analysis shows that the $\mathbf{Top}^3+$HPA-30 auto scaler used HPA only 74% of the time. The use of HPA decreases with better coverage. For example, $\mathbf{Top}^5+$HPA-30 and $\mathbf{Top}^{10}+$HPA-30 use HPA 44% and 22% of the time, respectively. This pattern is also seen when TRIM is paired with HPA-50.

From Figure 8, using only HPA-50 results in many SLA violations suggesting that a target of 50% CPU utilization is too high to preempt SLA violations. From the figure, reducing the threshold to 30% reduces SLA violations at the cost of increased resource allocation. However, there are still a significant number of SLA violations. This reinforces the difficulty in using static resource utilization thresholds to manage SLA targets.

In all cases depicted in Figure 8, TRIM significantly decreases *both* the number of SLA violations and total replica count. Specifically, TRIM uses lesser instances and leads to increased per-instance CPU utilization thereby using resources more efficiently. Furthermore, expanding the number of popular intensity ranges leads to further reductions in SLA violations. For example, by considering only the top 3 ranges having a 33% coverage, the total number of container instances, i.e., cost of deployment, decreases by 10.5% and SLA violations decrease by 53% with HPA-50. When we increase the coverage to 62% by adding two more ranges, the cost reduces by about 22% and SLA violations decrease by 63%. Using the top 10 ranges reduces cost and SLA violations by 20% and 84%, respectively. The results for HPA-30 also shows similar patterns. Our results show that TRIM can achieve fewer SLA violations than merely using HPA while requiring fewer instances and having better utilization of the service's resources. Considering all experiments, TRIM can

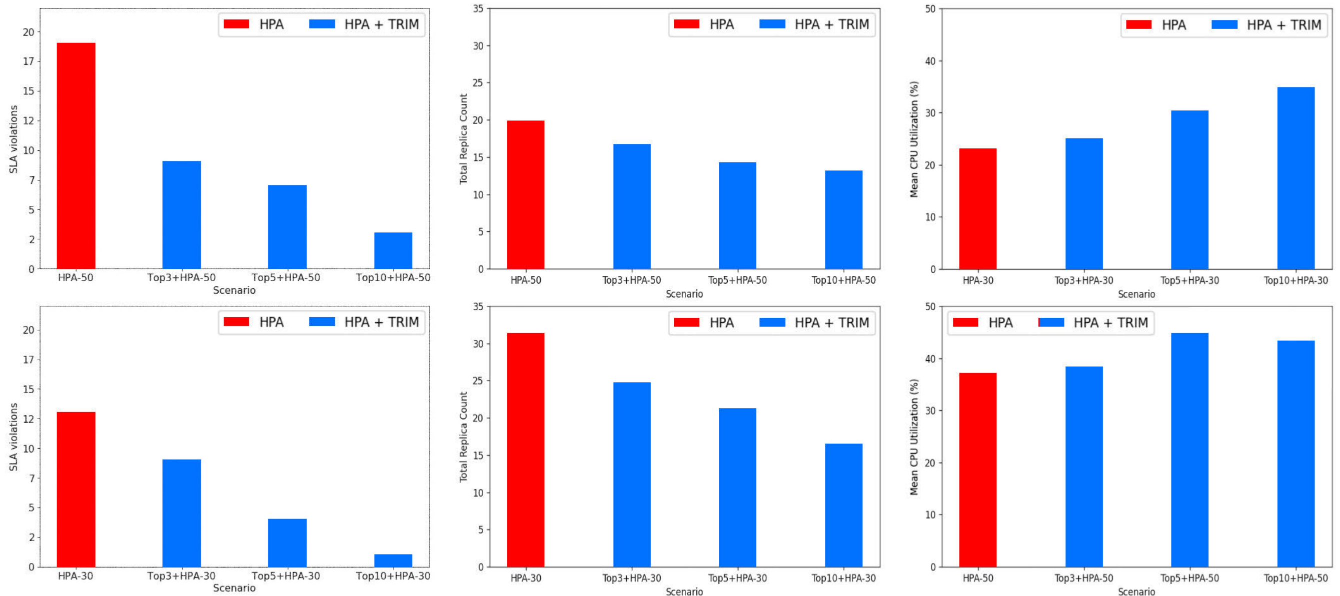

FIGURE 8. HPA vs. TRIM.

reduce costs and SLA violations by up to 34% and 92%, respectively.

We now provide a more detailed analysis of the results by comparing HPA-50 with and without TRIM. In most of the intervals where TRIM finds a match, one can observe that the allocated resources are fewer with TRIM due to the optimization performed by MOAT. However, there are some regions where TRIM allocates more resources than HPA-50. These represents regions where MOAT ensures that there are enough resources to prevent SLA violations but HPA-50 does not. Overall, TRIM uses fewer resources than using HPA-50 in isolation. While using TRIM, all of the observed SLA violations are found in a subset of the regions where TRIM did not find a match and HPA-50 takes over. These SLA violations occur due to under-provisioning of resources. This further confirms that SLA violations can be further reduced by merely expanding the number of popular ranges considered by TRIM.

Finally, we analyze the small number of cases where the system experiences an SLA violation with TRIM. As mentioned previously, the violations happen when TRIM falls back on HPA. Depending on the experiment, the extent of violations under TRIM are comparable or lower than that of HPA. For example, HPA-30 has a mean violation of 2% over the response time target. While $\textbf{Top}^3$ + HPA-30 resulted in an identical 2% violation over the target, $\textbf{Top}^5$ + HPA-30 resulted in a slightly higher violation of 3%. $\textbf{Top}^{10}$ + HPA-30's violations are measured to be under 1%. Similar results are observed for HPA-50. As noted previously, TRIM results in significantly lower SLA violations than merely using HPA. Thus, combined with its comparable or lower extent of SLA violations, TRIM comprehensively outperforms HPA.

VIII. CONCLUSION AND FUTURE WORK

Microservice applications often encounter time-varying workloads and hence can benefit from auto-scaling. In addition to being SLA-aware and efficient, a microservice auto-scaler should be CD-compatible, i.e., obviate the need for modeling as well as the need to characterize system performance under a large number of workload patterns and resource allocations. Currently there is a lack of auto-scaling solutions that address the three key requirements of SLA awareness, efficiency, and CD compatibility simultaneously. We develop an auto-scaler called TRIM that addresses this gap.

TRIM exploits a novel insight we derived from analyzing the workloads of more than 24, 000 real-world microservice applications. Over time these workloads are dominated by a small number of popular intensity ranges and these intensity ranges continue to be popular over the subsequent period. Using a novel resource allocation module called MOAT that we developed, TRIM quickly pre-computes SLA aware and efficient resource allocations for these popular workload ranges and uses these at runtime to perform auto-scaling.

Extensive evaluation using analytical, in-house, and public cloud systems demonstrates the effectiveness of our approach. Specifically, using the analytical systems, we show that the resource allocations obtained by MOAT are very close to theoretical minimum allocations obtained analytically. Furthermore, MOAT outperforms a BO technique in identifying efficient resource allocations and reducing profiling effort. We also show that TRIM in consort with MOAT can significantly improve the performance of the industry-standard HPA auto-scaler. Specifically, by expending very little profiling effort our approach is able to reduce costs and SLA violations by up to 34% and 92%, respectively.

TABLE 8. Tuning cores and memory.

Iteration	Service	Cores	Memory (MB)	Mean CPU PSI	Mean Memory PSI	Bottleneck type	Notes
1	auth	0.5	512	17.27	52.02	Memory	
	books	0.5	512	12.16	60.80	Memory	Two request classes (get-book & edit-book) violate response time targets. Hence, the books service is updated with $2 \times \Delta_{mem}$.
	gateway	0.5	512	26.93	23.11	CPU	
2	auth	0.5	2560	59.01	0.00	CPU	Addition of memory shifts bottleneck to CPU.
	books	0.5	4608	70.57	0.00	CPU	
	gateway	0.5	512	0.88	33.09	Memory	CPU bottleneck shifts to Memory.
3	auth	2.5	2560	54.10	0.00	CPU	CPU bottleneck is being resolved over the next iterations.
	books	4.5	4608	39.19	0.00	CPU	
	gateway	0.5	512	4.18	21.87	Memory	
4	auth	4.5	2560	13.99	0.00	CPU	
	books	8.5	4608	13.70	0.00	CPU	
	gateway	0.5	512	8.08	33.92	Memory	All request classes violated targets so the gateway service is updated with $3 \times \Delta_{mem}$.
5	auth	4.5	2560	0.02	0.00	–	
	books	8.5	4608	0.02	0.00	–	
	gateway	0.5	6656	34.29	0.00	CPU	
6	auth	4.5	2560	1.02	0.00	–	
	books	8.5	4608	0.00	0.00	–	
	gateway	6.5	6656	0.74	0.00	–	

The TRIM technique is agnostic to the specific traces used to drive the resource allocation since the basic idea is to use data observed over a current time period to infer resource allocations for the next period. As we show in the paper, this approach seems very promising. We will study other traces as they become available to confirm whether our findings regarding popularity of workload intensity ranges generalize to other microservice systems.

TRIM assumes that the sandbox used to estimate resource allocations during the profiling is similar to the deployment environment. It is possible for this assumption to be violated. For example, a cloud subscriber may change resource allocation policies, e.g., choose to over-subscribe a resource among multiple applications more aggressively. In such a scenario, the pre-computed scaling strategies obtained by MOAT might become sub-optimal. A thorough empirical examination of the sensitivity of our approach to such phenomena is deferred to future work. We also defer to future work validation with larger scale microservice applications.

Focusing on our existing TRIM implementation, we will explore improved techniques to handle scenarios where the measured intensity does not match any of the popular intensities characterized by TRIM. Furthermore, we will explore the use of the MOAT algorithms within evolutionary approaches.

Future work will look at adapting ML-based techniques to handle the agile nature of microservice deployments. This will require effort on two fronts. First, reference implementations of ML-based techniques will need to be developed to foster repeatable research in this area. Second, experimental studies need to be designed using these reference implementations. For example, such studies can explore the robustness of a Q-learning function learnt by RL to application changes, e.g., updates to microservices and addition of new microservices and request classes.

APPENDIX A
TUNING MULTIPLE RESOURCES

We now demonstrate how MOAT can be used to tune multiple resources simultaneously. Specifically, we extend the enhanced algorithm to consider both the cores and memory allocated to a microservice. To better delineate CPU and memory bottlenecks, we use the Pressure Stall Information (PSI) tool [52] supported by Linux (kernel 5.2+, cgroup v2). The PSI of a resource is the percentage of time a task is blocked while waiting on the resource over a time window. PSI values are supported for CPU, memory, and I/O resources [52]. For example, a measured CPU PSI of 50% indicates that half of the tasks were blocked waiting on CPU over a time window. Since Docker and, by association, Kubernetes rely on cgroups to manage resources, we can measure the PSI metrics of a container or pod resource over a window of time. PSI can offer better insights on bottlenecks than relying on resource utilization alone. For example, high memory utilizations may not always indicate a memory bottleneck. In contrast, a high PSI value for memory indicates significant contention, which can adversely impact service performance.

MOAT incorporates PSI values as follows. As detailed in Sec. V, it identifies the service resources used by a request class violating its response time target. It then selects the resource with the highest PSI value. This resource is increased to alleviate the bottleneck. In the pruning phase, the service resource with the smallest PSI value is selected for pruning.

We present an experiment using the Bookstore application on our on-premise system to show how MOAT drives resource allocations when the bottleneck type shifts during the tuning process. We sample PSI values over 10 second intervals. For this experiment, we consider vertical scaling with each service hosted on its own dedicated replica. For any given service's container, MOAT computes the number of cores and memory to achieve response time targets for request classes. We configure the enhanced algorithm with $\Delta_{cpu} = 2$ and $\Delta_{mem} = 2$ GB. All other settings are identical to the ones used in Sec. VI.

Table 8 shows the iterations of MOAT's enhanced algorithm for this experiment. It takes 6 iterations for MOAT to converge on a solution that satisfies response time targets for all request classes. The table depicts how MOAT uses PSI values to identify candidate services and resource type, i.e., core or memory, for additional allocations. For example, after iteration 1, 2 request classes violate their response time targets. Furthermore, the memory of the "books" service has the highest PSI value and both the violating request classes use this service. Accordingly, MOAT allocates $2 \times \Delta_{mem}$ additional memory to that service. This additional memory alleviates the bottleneck, as observed by the very low memory PSI value for the books service in iteration 2. In iteration 2, the CPU PSI value of books is the highest and hence MOAT targets additional core allocations to that service. Eventually, all resources of all services have very low PSI values (indicated by a '-' in the bottleneck type column of the table) and all response time targets are satisfied in iteration 6. We note that MOAT never enters the pruning phase in this experiment since all the request class response times are very close to their corresponding targets.

REFERENCES

[1] *Production-Grade Container Orchestration*. Accessed: Aug. 8, 2021. [Online]. Available: https://kubernetes.io/
[2] M. Shahin, M. A. Babar, M. Zahedi, and L. Zhu, "Beyond continuous delivery: An empirical investigation of continuous deployment challenges," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Nov. 2017, pp. 111–120.
[3] M. Hajjat, P. N. Shankaranarayanan, D. Maltz, S. Rao, and K. Sripanidkulchai, "Dealer: Application-aware request splitting for interactive cloud applications," in *Proc. 8th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2012, pp. 157–168.
[4] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "The evolution of distributed systems towards microservices architecture," in *Proc. 11th Int. Conf. Internet Technol. Secured Trans. (ICITST)*, Dec. 2016, pp. 318–325.
[5] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst. (NIPS)*, vol. 2. Cambridge, MA, USA: MIT Press, 2015, pp. 2503–2511.
[6] A. Tsymbal, "The problem of concept drift: Definitions and related work," *Comput. Sci. Dept., Trinity College Dublin*, vol. 106, no. 2, p. 58, 2004.
[7] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices," in *Proc. 14th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2020, pp. 805–825.
[8] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: Workload autoscaling at Google," in *Proc. 15th Eur. Conf. Comput. Syst.*, Apr. 2020, pp. 1–16.
[9] A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 1994–2004.
[10] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, Jul. 2019, pp. 68–75.
[11] I. Prachitmutita, W. Aittinonmongkol, N. Pojjanasuksakul, M. Supattatham, and P. Padungweang, "Auto-scaling microservices on IaaS under SLA with cost-effective framework," in *Proc. 10th Int. Conf. Adv. Comput. Intell. (ICACI)*, Mar. 2018, pp. 583–588.
[12] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2020, pp. 205–218.
[13] *HPA*. Accessed: May 1, 2021. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[14] V. M. Mostofi, D. Krishnamurthy, and M. Arlitt, "Fast and efficient performance tuning of microservices," in *Proc. IEEE 14th Int. Conf. Cloud Comput. (CLOUD)*, Sep. 2021, pp. 1–6.
[15] V. M. Mostofi. (2021). *Auto-Scaling Containerized Microservice Applications*. [Online]. Available: https://prism.ucalgary.ca/handle/1880/113888
[16] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 1–33, Jul. 2018, doi: 10.1145/3148149.
[17] P. Singh, P. Gupta, K. Jyoti, and A. Nayyar, "Research on auto-scaling of web applications in cloud: Survey, trends and future directions," *Scalable Comput., Pract. Exp.*, vol. 20, no. 2, pp. 399–432, May 2019.
[18] *Amazon Auto Scaling Service*. Accessed: Aug. 13, 2021. [Online]. Available: http://aws.amazon.com/autoscaling/
[19] S. A. Javadi and A. Gandhi, "User-centric interference-aware load balancing for cloud-deployed applications," *IEEE Trans. Cloud Comput.*, vol. 10, no. 1, pp. 736–748, Jan. 2022.
[20] J. Mukherjee and D. Krishnamurthy, "PRIMA: Subscriber-driven interference mitigation for cloud services," *IEEE Trans. Netw. Service Manage.*, vol. 17, no. 2, pp. 958–971, Jun. 2020.
[21] S. Taherizadeh and V. Stankovski, "Dynamic multi-level auto-scaling rules for containerized applications," *Comput. J.*, vol. 62, no. 2, pp. 174–197, Feb. 2019.
[22] W.-H. Liao, S.-C. Kuai, and Y.-R. Leau, "Auto-scaling strategy for Amazon web services in cloud computing," in *Proc. IEEE Int. Conf. Smart City/SocialCom/SustainCom (SmartCity)*, Dec. 2015, pp. 1059–1064.
[23] S. Frey, C. Lüthje, C. Reich, and N. Clarke, "Cloud QoS scaling by fuzzy logic," in *Proc. IEEE Int. Conf. Cloud Eng.*, Mar. 2014, pp. 343–348.
[24] P. Jamshidi, C. Pahl, and N. C. Mendonça, "Managing uncertainty in autonomic cloud elasticity controllers," *IEEE Cloud Comput.*, vol. 3, no. 3, pp. 50–60, May 2016.
[25] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *Proc. IEEE Int. Conf. Cloud Eng.*, Mar. 2014, pp. 195–204.
[26] C. Qu, R. N. Calheiros, and R. Buyya, "A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances," *J. Netw. Comput. Appl.*, vol. 65, pp. 167–180, Apr. 2016.
[27] N. Grozev and R. Buyya, "Multi-cloud provisioning and load distribution for three-tier applications," *ACM Trans. Auto. Adapt. Syst.*, vol. 9, no. 3, pp. 1–21, Oct. 2014.
[28] M. R. Hossen, M. A. Islam, and K. Ahmed, "Practical efficient microservice autoscaling with QoS assurance," in *Proc. 31st Int. Symp. High-Perform. Parallel Distrib. Comput.* New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 240–252, doi: 10.1145/3502181.3531460.
[29] F. Lombardi, A. Muti, L. Aniello, R. Baldoni, S. Bonomi, and L. Querzoni, "Pascal: An architecture for proactive auto-scaling of distributed services," *Future Gener. Comput. Syst.*, vol. 98, pp. 342–361, Sep. 2019.
[30] A. Y. Nikravesh, S. A. Ajila, and C.-H. Lung, "Towards an autonomic auto-scaling prediction system for cloud resource provisioning," in *Proc. IEEE/ACM 10th Int. Symp. Softw. Eng. Adapt. Self-Manage. Syst.*, May 2015, pp. 35–45.
[31] T. Vondra and J. Šedivý, "Cloud autoscaling simulation based on queueing network model," *Simul. Model. Pract. Theory*, vol. 70, pp. 83–100, Jan. 2017.

[32] R. Han, M. M. Ghanem, L. Guo, Y. Guo, and M. Osmond, "Enabling cost-aware and adaptive elasticity of multi-tier cloud applications," *Future Gener. Comput. Syst.*, vol. 32, pp. 82–98, Mar. 2014.
[33] F. Rossi, V. Cardellini, and F. L. Presti, "Hierarchical scaling of microservices in kubernetes," in *Proc. IEEE Int. Conf. Autonomic Comput. Self-Organizing Syst. (ACSOS)*, Aug. 2020, pp. 28–37.
[34] A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, and S. Kounev, "Chamulteon: Coordinated auto-scaling of micro-services," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 2015–2025.
[35] A. A. Khaleq and I. Ra, "Intelligent autoscaling of microservices in the cloud for real-time applications," *IEEE Access*, vol. 9, pp. 35464–35476, 2021.
[36] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, "Energy-aware server provisioning and load dispatching for connection-intensive internet services," in *Proc. NSDI*, vol. 8, 2008, pp. 337–350.
[37] M. Abdullah, W. Iqbal, A. Erradi, and F. Bukhari, "Learning predictive autoscaling policies for cloud-hosted microservices using trace-driven modeling," in *Proc. CloudCom*, Dec. 2019, pp. 119–126.
[38] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach," *IEEE Trans. Cloud Comput.*, vol. 10, no. 2, pp. 1100–1116, Apr. 2022.
[39] *SWITCH Project*. Accessed: Aug. 2, 2021. [Online]. Available: http://www.switchproject.eu/
[40] *TRIM, MOAT*. Accessed: Aug. 10, 2021. [Online]. Available: https://github.com/vahidmostofi/acfg
[41] B. Liu, R. Buyya, and A. N. Toosi, "A fuzzy-based auto-scaler for web applications in cloud computing environments," in *Proc. Int. Conf. Service-Oriented Comput.* Springer, 2018, pp. 797–811.
[42] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, "Elascale: Autoscaling and monitoring as a service," 2017, *arXiv:1711.03204*.
[43] M. Abdullah, W. Iqbal, J. L. Berral, J. Polo, and D. Carrera, "Burst-aware predictive autoscaling for containerized microservices," *IEEE Trans. Serv. Comput.*, vol. 15, no. 3, pp. 1448–1460, May 2022.
[44] D. Freedman and P. Diaconis, "On the histogram as a density estimator: L_2 theory," *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, vol. 57, no. 4, pp. 453–476, 1981.
[45] M. D. Zeiler, "ADADELTA: An adaptive learning rate method," 2012, *arXiv:1212.5701*.
[46] P. P. Galuzio, E. H. de Vasconcelos Segundo, L. D. S. Coelho, and V. C. Mariani, "MOBOpt—Multi-objective Bayesian optimization," *SoftwareX*, vol. 12, Jul. 2020, Art. no. 100520. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352711020300911
[47] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Upper Saddle River, NJ, USA: Prentice-Hall, 1984.
[48] R. Fourer, D. M. Gay, and B. W. Kernighan, "AMPL: A modeling language for mathematical programming," Tech. Rep., 2003.
[49] A. Drud, "CONOPT: A GRG code for large sparse dynamic nonlinear optimization problems," *Math. Program.*, vol. 31, no. 2, pp. 153–191, Jun. 1985.
[50] A. S. Drud, "CONOPT—A large-scale GRG code," *ORSA J. Comput.*, vol. 6, no. 2, pp. 207–216, May 1994.
[51] *K6 Load Generator*. Accessed: Dec. 11, 2020. [Online]. Available: https://k6.io/
[52] J. Weiner. (Apr. 2018). *PSI—Pressure Stall Information—The Linux Kernel Documentation*. [Online]. Available: https://www.kernel.org/doc/html/latest/accounting/psi.html

VAHID MIRZAEBRAHIM MOSTOFI received the M.Sc. degree from the University of Calgary. His research interest includes performance evaluation, with an emphasis on microservices-based architectures.

EVAN KRUL received the B.Sc. degree (Hons.) in software engineering from the University of Calgary, in 2022. He is currently pursuing the M.Phil. degree in computer science with the University of New South Wales, Sydney, Australia. He received an internship from the University of Calgary.

DIWAKAR KRISHNAMURTHY (Member, IEEE) is currently a Professor with the University of Calgary. He is involved in research projects related to cloud computing, big data analytics, and extended reality. His research interest includes the performance evaluation of software systems.

MARTIN ARLITT (Senior Member, IEEE) is currently a Principal Research Scientist and a Research Team Manager with OpenText. He is an Adjunct Assistant Professor with the University of Calgary. His 100 research papers have been cited more than 13,700 times (according to Google Scholar; H-index = 45). He has 49 granted patents and more pending. His general research interests include workload characterization of computer servers, performance evaluation of distributed computer systems, and analyzing network traffic to improve IT security. He is an ACM Distinguished Scientist.