Optimal configuration of API resources in cloud native computing

Eddy Truyen Wouter Joosen
DistriNet, KU Leuven, Leuven, Belgium
Eddy.Truyen@kuleuven.be Wouter.Joosen@kuleuven.be

This paper presents how an existing framework for offline performance optimization can be applied to microservice applications during the Release phase of the DevOps life cycle. Optimization of resource allocation during the Release phase remains a largely unexplored problem as most research has focused on intelligent scheduling and autoscaling of microservices in the production environment. Yet horizontal auto-scaling of containers, based on CPU usage for instance, may still leave these containers with an inappropriately configured amount of memory, if no upfront fine-tuning of both resources is applied. We evaluate the framework using the TeaStore microservice application and statistically compare different optimization algorithms, supporting informed decisions about their trade-offs between sampling cost and distance to the most optimal configuration. This shows that upfront screening for reducing the search space is helpful when the goal is to find the optimal resource configuration as found in exhaustive search. When the goal is to statistically compare different algorithms with respect to the optimal configuration, screening must also be applied to make data collection of all data points in the search space feasible. If the goal is to find a near-optimal configuration within a limited sampling budget, it is better to run bayesian optimization without screening.

1 Introduction

Modern software is increasingly developed with cloud-native deployment in mind, driving a shift from monolithic architectures toward distributed systems composed of containerized microservices. These services are typically deployed on elastic platforms such as container orchestration (CO) systems [20], with Kubernetes being the most prominent example. Containerization supports DevOps practices and continuous integration/deployment (CI/CD), enabling rapid release cycles and agile iteration [2]. One key advantage of CO platforms is their ability to define fine-grained resource allocations—such as CPU and memory—on a per-container basis [30]. These resource decisions have a direct impact on application performance metrics (e.g., tail latency or throughput), which are typically governed by strict service-level objectives (SLOs) [3, 33].

Accurately translating high-level SLOs into efficient container resource configurations has been a hot topic of research in the past decade. Most CO platforms require manual translation, often leading to suboptimal outcomes like over-provisioning and underutilization of resources, which increases costs [30]. Case studies show significant inefficiencies, such as T-Mobile's containers having only 5.15% memory utilization [22], and most jobs in production clusters being over-provisioned [16]. The complexity of modern microservice stacks makes manual tuning slow and error-prone, often taking weeks [23], which doesn't fit with fast release cycles. Non surprisingly, a mass of work has been done on the use of machine learning and optimization to improve the performance efficiency of containers [40].

However, most of this research has tackled the operational phases of DevOps lifecyle, thus neglecting the development phases [26]. The operational stage includes deploying a application in a production

Submitted to:
WACA 2025

environment and scheduling its containers to the most appropriate computing nodes, monitoring their resource usage and SLO compliance of the application, and if needed to horizontally or vertically scale some of their containers. It is also important to choose the right initial amounts of resources for containers even when auto-scaling is applied. While horizontal or vertical auto-scaling tactics do improve the overall utilization of the provisioned containers, these solutions do not entail a complete solution for the cost-optimal management of SLOs. Horizontal auto-scaling based on CPU thresholds may still lead to containers being over- or under-provisioned in terms of memory, i.e., there should be a balance between memory and CPU resource allocation. This problem also remains when auto-scaling is based on multiple metrics such as the memory usage or CPU usage as scaling is still performed when either one of these metrics crosses a threshold. Similarly, vertical auto-scaling [31], while adjusting CPU and memory resources for each container, can cause container restarts, which may result in service disruptions and increased latency. Finally, auto-scalers are applied per microservice and therefore do not take dependencies between microservices into account. Off-line optimization can configure microservice applications as a whole by employing end-to-end user scenarios for performance testing.

Therefore, we have been working in the past on a optimizer framework, named k8-resource-optimizer, that targets the Release phase of the DevOps cycle, before the application is deployed in a production environment [17, 25]. In particular, k8-resource-optimizer is an existing framework for offline performance optimization of containerized multi-tenant applications deployed on Kubernetes (K8s), the de-facto standard for container orchestration. The framework can also be extended with new types of workloads and optimization goals for these workloads as well as different optimization algorithms. Once these plugin components are developed, the user needs to provide minimal input: the workload intensity, the SLO to be met, and the set of tunable resource parameters. Last but not least, framework can be used to fine-tune any resource parameter that is configurable via the K8s API, thus supporting also performance optimization of string-based application configuration parameters of microservices themselves.

In this paper, we make the following contributions:

- We apply k8-resource-optimizer to offline resource optimization of a microservice-based application deployed on container orchestration platforms.
- We demonstrate that k8-resource-optimizer can statistically compare different optimization algorithms to understand their trade-off between sampling cost and distance to most optimal resource configuration.
- We investigate the relevance of upfront screening for reducing the search space so that optimization algorithms have a higher probability to find the most optimal configuration or to find a near-optimal configuration with a limited sampling budget.

Overview of the paper. The paper is organized as follows. Section 2 presents the background to understand the remainder of this paper. Subsequently, Section 3 describes the architecture and implementation of k8-resource-optimizer. Then, Section 4 evaluates the framework in the context of the TeaStore microservice application. Next, Section 5 discusses the lessons learned and the usefulness of k8-resource-optimizer for optimization of application configuration parameters of microservices. Thereafter, Section 6 describes the related work on off-line performance optimization in cloud-native computing. Finally, Section 7 presents our conclusions and directions of future work.

2 Background

This section presents the overall approach behind performance optimization and introduces screening and specific optimization algorithms that are evaluated in the paper.

2.1 Performance optimization

In performance optimization, the goal is to find an optimum of an utility function $f : X \to \mathbb{R}$ by iteratively sampling values of $f(x)$ in order to find a global optimum of $x_{opt} \in X$. If evaluation of $f(x)$ is expensive, the search for the global optimum should be done efficiently in terms of the number of evaluations. Several optimization algorithms have been suggested to guide this search such as Bayesian optimization, simulated annealing, and genetic algorithms [4]. These algorithms typically make trade-offs between the *exploration* of the search space and *exploitation* of insights in already sampled regions to guide the selection of the next sample.

In what follows, we formalize the performance optimization problem for a given containerized microservice application as: For a workload W, find the optimal or a near-optimal resource configuration $C^*(\vec{x})$ that satisfies the *SLOs* of given *SLA s* and minimizes the operational cost $P(\vec{x})$. We use $\vec{x}$ to denote the selected parameter settings of a configuration C, where $\vec{x}$ includes resources settings (CPU, RAM, disk I/O, network bandwidth) for each of microservices part of the applications. Let $SLI(\vec{x})$, Service Level Indicator [3], be the actual measurements of the performance test for settings $\vec{x}$. Since, the definition for an optimal configuration depends heavily on the application context, we allow for a user-defined *utility function* with configuration settings $\vec{x}$ and $SLI(\vec{x})$ as input parameters. Our goal is two-fold: Firstly, to find the configuration C^* with utility score u^* which has the smallest possible distance to the most optimal configuration C_W for the given workload. The distance between C_W and C^* is expressed as the absolute value the difference in utility $|u^* - u_W|$. Secondly, the time to find an optimal configuration is limited. Therefore, the search should be performed within a limited amount of performance test samples.

2.2 Factor screening

The size of the search space is determined by the search ranges for each parameter $[min, max]$. However, as developers might be clueless on resource requirements of their services, this search-space may contain regions that are incapable of even remotely satisfying the SLO, and as such exploration of these regions would be pointless. Therefore it makes sense to reduce the full parameter space to a sub-area that has a high probability to contain the optimal resource configuration and that has a smooth performance surface. We assume that this allows off-the-shelf optimization algorithms to perform much better. With this end in view, it is possible to run k8-resource-optimizer with a factor screening/sensitivity analysis algorithm that relies on calculating Elementary Effects (EE) as implemented by Morris One-At-a-Time (MOAT) [27]. The MOAT algorithm discovers for a given function which inputs have a substantial influence on the outputs. When applied to performance optimization, the function maps the resource parameters of the search space to attained Service Level Indicators (SLIs) such as latency and throughput, and the algorithm works as follows. The k-dimensional search-space (for k resource parameters), is partitioned in p uniform levels for each parameter, resulting in a p^k grid of possible settings. The method iteratively evaluates configurations in the search-space, starting from a random point in the grid. During each iteration, one resource parameter x_i is altered at a time in a discrete parameter space while fixing the other parameters. As such, a *trajectory* is created in the search-space of $k+1$ evaluations. Altering parameter x_i in consecutive evaluations allows to calculate the elementary effect (EE) of that parameter, $EE_i = \frac{y(x_1,\ldots,x_i+\Delta,\ldots,x_k)-y(x_1,\ldots,x_i,\ldots,x_k)}{\Delta}$, where $y(x)$ is based on the Service Level Indicators (SLIs) obtained from the evaluations. We use $\Delta = \frac{p}{2(p-1)}$, leading to steps slightly larger than half the range of the normalized parameter range between $[0,1]$, as recommended by [32]. Repeating the process for r *trajectories* with r being between 5 to 15 (i.e., a total of $r*(k+1)$ evaluations), allows for the calculation of the mean μ_i, modified mean μ_i^* (which are the absolute values of EE_i) and standard deviation σ_i of each parameter

x_i. The mean and modified mean express the overall influence of a configuration parameter on the SLIs, and standard deviation indicates dependence on other parameter settings [32]. Based upon the results of the screening algorithm, the parameter search ranges are adapted as follows: (i) the *MIN* setting is set to the minimum setting tested capable of satisfying a more relaxed SLI (e.g., a tail-latency that is 40% slower than the targeted SLI), (ii) the size of the search-space is set in accordance to the impact of (μ_i^*, σ_i) of the parameter compared to other parameters.

2.3 Optimization algorithms

Bayesian optimization (BO) [34] has become a popular technique to solve optimization problems. It sequentially models the utility function as a stochastic process with configuration settings as input parameters. First, the *configuration-vs-utility* space is modeled by regressing a set of prior sample points (configurations already evaluated). This allows to create an estimate of the entire search-space, BO calculates a confidence interval of the utility function. This model is then sequentially refined by the addition of samples via Bayesian posterior updating. The next sample point is selected using a predefined *acquisition function*. This function aims to balance exploitation (i.e., regions with a high probability of containing an optimum) and exploration (i.e., regions with high uncertainty). We selected the popular Expected Improvement (EI) function for this task, as it has shown promising results in similar contexts [1, 23] and it does not require tuning of its own parameters [35]. BO's sequential approach is suitable for scenarios where the evaluation of a single data point is a time-consuming task [23], hence it is a suitable approach for our use-case.
BestConfig [41] explores the search-space by iterations of their proposed divide-and-diverge (DDS) sampling method and recursive bound-and-search (RBS) algorithm. DDS is a form of stratified sampling to achieve a wide coverage of the search-space (exploration). Based on the results of samples proposed by DDS, RBS either narrows the search-space near high-achieving samples (exploitation) or backtracks to avoid local optimums.

3 Framework

In this section, we present the command line interface, architecture and implementation of the k8-resource-optimizer framework.

3.1 Command line interface

To optimize the deployment of a containerized microservice-based application on K8s, k8-resource-optimizer relies on two key inputs: (1) a helm chart that defines the base deployment including all K8s manifests; more specifically a `values.yaml` file inside the helm chart defines all the customizable K8s resource parameters and default values for these parameters, and (2) an optimizer configuration file that specifies the service-level objectives (SLOs), and the parameter search space for tunable parameters of the Helm chart. In particular the optimizer configuration specifies one or more SLAs where each SLA presents a specific helm chart with a specific application. When more SLAs are specified, their corresponding applications are expected to be co-located on the same K8s cluster. Each SLA defines the following items:

- The name of the helm chart that should be used for deploying the application.
- One or multiple SLOs such as throughput or latency.

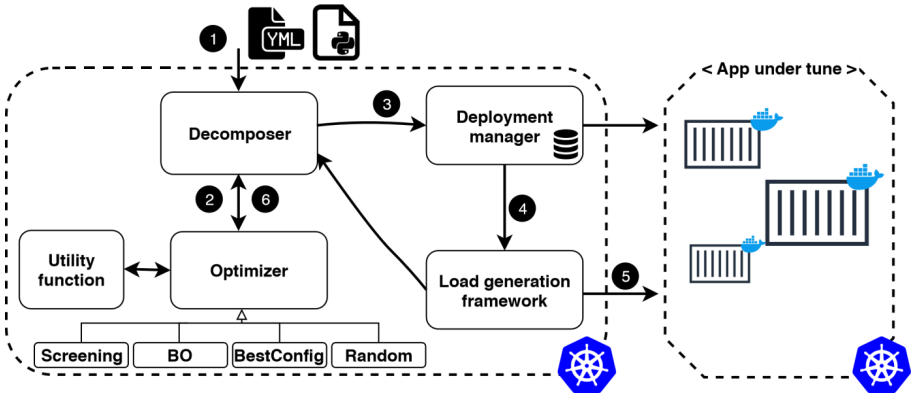

Figure 1: The architecture of k8-resource-optimizer.

- The expected workload intensity in terms of the number of concurrently running tenants and the expected throughput per tenant.
- A set of tunable parameters for each microservice that have to be defined upfront in the `values.yaml` file of the referenced helm chart. For each tunable parameter, a search range is defined (`min`, `max`, and `granularity`) and unit suffixes (e.g., `m` for millicores, `Mi` for mebibytes).

Next an optimization strategy is included which consists of the following items:

- `nbOfIterations` and `nbOfSamplesPerIteration`, which control the search budget,
- `namespaceStrategy` (e.g., `NSPSLA` or `NSPT`) to determine if the application is multi-tenant or needs to be launched separately for each tenant.
- `optimizer` (e.g., `MOAT`, `bestconfig`) which selects the underlying optimization algorithm,
- `utilFunc`, a utility function used to evaluate the performance efficiency of each configuration.

An example `values.yaml` file and optimizer configuration file for the TeaStore application are presented in Appendix A and B, respectively. K8-resource optimizer can also be used for optimization of any other resource parameter in the K8s API. For example, configuration parameters of the microservices can be optimized by making each parameter templatable in the Helm chart. This is a matter of including a variable for it in the `values.yaml` file. String based parameters should be converted in the helm chart to an Integer range with a granularity of 1. An example of the latter is shown in Appendix C.

Finally, a benchmark scenario is to be implemented when running k8-resource-optimizer by instantiating the `Experiment` interface. For microservice applications, we target an end-to-end user scenario where a user sequentially invokes multiple operations of the microservice application to simulate a full user journey. Such end-to-end user scenario is preferred as the basis of performance testing during optimization as this implicitly accounts for the dependencies between different microservices and optimizes the application as a whole instead of optimizing each microservice individually.

3.2 Architecture

The architecture of k8-resource-optimizer is illustrated in Figure 1 and is composed of loosely coupled components interacting via common interfaces. The data-flow between these components during optimization is given in Algorithm 1.

⇒ ❶ load optimizer configuration and performance test;
for *iteration* **do**
  ❷ ask Optimizer for next configuration $C(\vec{x})$ to sample;
  ❸ deploy application with resource parameter settings of $\vec{x}$;
  ❹ wait for application to be marked ready;
  ❺ obtain $\text{SLI}_1(\vec{(x)})$, $\text{SLI}_2\vec{(x)})$, ..., $\text{SLI}_n(\vec{(x)})$ from performance test;
  ❻ feed results to Optimizer;
end
⇐ (near)-optimal configuration

Algorithm 1: Iteration loop of k8-resource-optimizer

Optimizer The optimizer provides a reusable interface suitable for most optimization algorithms based upon exploration and exploitation. The predefined *utility function* interface, described in Section 2.1, allows the combination of any optimization algorithm and user-defined utility function.

Deployment Manager Responsible for the deployment of a particular helm chart and values.yaml file inside a K8s cluster. K8-resource-optimizer leverages K8s' readiness probes, to ensure that the application is fully deployed before running a performance test and retries a failed installation/removal of deployments.

Load generation In order to support a variety of load-generation frameworks (e.g., Locust, JMeter, Vegeta), k8-resource-optimizer provides an interface for experiments to be run as a performance test. Currently, the popular load-generation framework Locust [11] is integrated.

3.3 Implementation

The core framework without plug-in components has been implemented using the Go programming language in 1.5K lines of code. To extend the k8-resource optimizer with a benchmark scenario and utility function for the TeaStore application, we need to implement an `Experiment` class and an `UtilityFunction` that together require 302 lines of code. We have implemented 11 different optimization algorithms including bayesian optimization, BestConfig, MOAT, as well as random search and an exhaustive search algorithm in 2.9K lines of code. Finally, we also developed 1K lines of code for static analysis of the performance of different optimizations techniques to understand the optimality versus the search cost ratio of the these techniques.

3.4 Setting workload intensity and resource parameter conversion

As stated in Section 3.1, k8-resource-optimizer requires manual specification of the workload intensity as well as the parameters to be optimized and their conversion to K8s API resources.

The workload intensity is specified in terms of the number of concurrent tenants and a request or job arrival rate per tenant. How to set the workload intensity depends on the expected workload fluctuation pattern, i.e., bursty, monotonically increasing, seasonal and stable [28, 24]. For stable and bursty workloads, the workload intensity should be set slightly larger than the constant workload volume of the

stable workload and set equal to the highest expected peak for the bursty workload. For monotonic and seasonal workloads, one should set the highest expected workload intensity.

The conversion of parameters to K8s API resources relies heavily on the scripting facilities of helm charts. The `values.yaml` file of a helm chart typically specifies for each container one parameter per resource type and a parameter for the number of container replicas (cfr. Appendix A). The K8s `yaml` files in the helm chart then convert these parameters to K8s API resources as follows.

Resource parameters in the `values.yaml` file must be converted to the K8s API `request` and `limit` entries. The `request` entry specifies the amount of node resources that should be available to the container, while the `limit` field sets the maximally allowed resource usage of the container. Both parameters are enforced by the operating system of the node [6].

In our experience, the resource parameters should be directly copied into the `limit` entry. The `request` entry's values are set equal to or a certain percentage lower than those of the `limit` entry [6, 38]. For stable workloads, this percentage will be smaller than for bursty workloads; if the application concerns a high-priority workload, however, requests must be equal to limits.

Moreover, in stable or bursty workloads, the number of container replicas is fixed and typically set higher than 1 to ensure availability of the microservices. For monotonic and seasonal workloads, the number of microservice replicas is to be included as a tunable parameter in the optimizer configuration file of k8-resource-optimizer, and resource parameters are to be converted as in bursty workloads. The optimal number of replicas found should then be set as the maximum number of replicas in auto-scaling solutions.

4 Evaluation

In this section, we illustrate the workflow of k8-resource-optimizer for TeaStore [19], a popular reference microservice application for performance evaluation of research prototypes. Thereafter we evaluate the usefulness of factor screening and compare the cost-effectiveness of the different optimization algorithms.

First, the experimental settings are explained in Section 4.1. Then factor screening is illustrated in Section 4.2, providing insights into the obtained reduced search-spaces for TeaStore v1.2.0. Next, Section 4.3 shows the results of the exhaustive data collection process after factor screening, while Section 4.5 evaluates to which extent the different optimization algorithms can find the optimal configuration as found by the exhaustive search, and Section 4.4 compares the cost-effectiveness of different algorithms with respect to the trade-off between search cost and distance of a found near-optimal configuration to the optimal one. Subsequently, Section 4.6 evaluates the usefulness of factor screening. Finally Section 4.7 evaluates the effective resource utilization of TeaStore as achieved by k8-resource-optimizer, hereby pointing to a few oversights in the overall workflow.

4.1 Experimental environment

This section provides a detailed description of the experimental environment and the configuration benchmark scenario used when running k8-resource-optimizer for TeaStore.

Infrastructure. Our testbed consisted of a private OpenStack cloud. The physical machines of this cloud come with 2.60 GHz Intel Xeon E5-2660 processors and 128GB DDR3 memory. The Kubernetes (v1.14) cluster consists of two nodes, master and worker, running on top of Ubuntu 16.04 VMs. The

master has 4 CPU cores and 8GB of RAM and the worker node has 8 CPU cores and 16GB of RAM. These nodes are actually VMs that run on the same physical machine but CPU pinning is enabled to minimize performance interference. As such, physical CPU cores are exclusively reserved for a single VM and all virtual CPUs of a VM map to CPU cores that belong to the same motherboard socket.

Microservice application. TeaStore [19] is a microservice application developed as a reference application for performance testing. It offers deployment specifications for multiple CO platforms where each microservice is deployed in its own container.

Benchmark scenario and workload intensity For TeaStore, a number of requests are sequentially invoked as follows: login → get categories → view products in a category → add product to cart → view profile → logout. The full benchmark scenario is presented in Appendix D.

As workload intensity we let 10 concurrently running tenants execute the benchmark scenario. We assume a stable workload without bursts and therefore set the K8s resource requests equal to K8s resource limits. The tested SLO in the evaluation concerns the 99th percentile latency of separate requests, which is set at 1000 ms.

Utility function In our experiments we use the following utility function,

$$u(\vec{x}) = \begin{cases} 1 + (SLI(\vec{x}) - SLO) & \text{if } SLI(\vec{x}) > SLO \\ normP(\vec{x}) & \text{otherwise} \end{cases}$$

where $normP(\vec{x})$ equals to the normalized resource allocation cost of $\vec{x}$. This utility function assigns a configuration that violates the SLO a score larger than 1 depending on the distance between the 99th percentile of the SLI and the SLO. It assigns configurations that satisfy the SLO a score between $[0,1)$, but discards their observed SLI in preference for resource allocation. The SLO-satisfying configuration with the lower resource allocation cost receives a lower score. The goal is to minimize this score.

Optimization algorithms For the implementation of Bayesian optimization (BO) we rely on a popular open-source Python library [29]. Here, the corresponding BO plugin maintains and communicates with a running process of the library. The library's specific implementation treats the search space as continuous. In our use-case, however, parameters have a certain granularity (e.g., 125 Mil), resulting in a discrete search space. Our optimization algorithm therefore, translates the suggested samples by the BO to the nearest discrete settings. For BestConfig's optimization algorithm and the factor screening algorithm MOAT, we implemented the algorithms as described in their respective papers. Other baseline optimization algorithms are random search and incremental random search (RandomInc). It is an optimizer that shuffles the order of parameters randomly, and then systematically iterates over all possible configurations built incrementally based on that shuffled parameter order. In practice, it generates all configurations but only tests a subset, depending on sampling budget, meaning it is not truly exhaustive unless allowed to complete.

4.2 Factor screening

The first step of k8-resource-optimizer's methodology uses MOAT (cfr. Section 2.2) to determine reasonable parameter ranges and hereby reducing the search space before employing the optimization algorithms. The initial bounds for the parameter ranges for CPU and memory are set the same for all the

	TeaStore
SLO - 99th prctl latency relaxed/target/strict (ms)	1250/1000/750
SLO - workload intensity (# concurrent tenants)	10
Initial CPU bounds (millicore)	[500, 1125]
Initial Mem bounds (Mi)	[512, 1152]
Initial search space (configs)	6^{14}
Reduced search space (configs)	2048
Sample time (min)	10

Table 1: Initial search spaces, reduced search spaces and sample time of applications.

Parameter	σ	μ^*	**min**	**max**
TeaStore				
webuiCpu	919	1755	625	750
registryCpu	352	243	500	625
dbMemory	278	230	512	640
imageCpu	278	175	500	500
persistenceMemory	354	242	512	512
dbCpu	436	302	500	625
authCpu	257	225	500	625
authMemory	410	250	512	512
persistenceCpu	498	360	500	625
recommenderCpu	577	345	500	625
recommenderMemory	234	128	512	512
registryMemory	457	313	512	512
imageMemory	354	203	512	640
webuiMemory	291	148	512	512

Table 2: Results of the factor screening for TeaStore include the modified mean μ^* and the deviation σ of 99th percentile latency. [min, max] are the newly selected parameter search spaces after factor screening.

TeaStore microservices. A minimum bound has been set by starting from the maximum bound and increasingly decreasing it with a granularity of 125 millicores for CPU resources and with a granularity of 128 Mi for memory resources, until the application does not start-up successfully. This results in a search space of 6^{14} possible configurations for TeaStore's 14 parameters, with each parameter having 6 possible settings. We run MOAT with 10 *trajectories* resulting in 150 configurations to be evaluated for TeaStore v1.2.0. The results of the factor screening algorithm are listed in Table 2 and the 99th percentile latency of the samples is plotted in Figure 2. It is clear from Table 2 that some parameters have a larger impact on the 99th percentile latency, i.e., those with a higher μ^* and σ values. Figure 2 shows the clear impact of configuration settings on the observed latency. However, subsequent points are also often clustered indicating that some parameter changes have a minimal effect on the observed latency. As explained in Section 2.2, the new min_i is set to the lowest parameter value of x_i achieving a less strict SLO (see Table 2). The max_i is based on the maximum parameter value of x_i that achieves a stricter SLO and the relative impact of (μ^*, σ). These new settings are also shown in Table 2. For TeaStore, the search space is reduced to 2048 configurations, pointing to a enormous reduction factor of 10^7.

4.3 Exhaustive data collection

The methodology for comparing the different optimization algorithms entails to exhaustively evaluate all configurations of the reduced search space that resulted from factor screening, and to store the evaluations in a persistent dataset to be leveraged for effective comparison of the optimization algorithms. A similar approach for evaluation of optimization algorithms has been used for optimization of storage solutions [4]. Figure 3 shows the 99th percentile latency CDF among all configurations for TeaStore. The measured 99th percentile latency values vary across a wide range. Some configurations experience twice the desired SLO. For TeaStore, merely 11% of the 2048 configurations is capable of satisfying the SLO. This shows that even after factor screening, the search space is still large enough to be explored by the optimization algorithms.

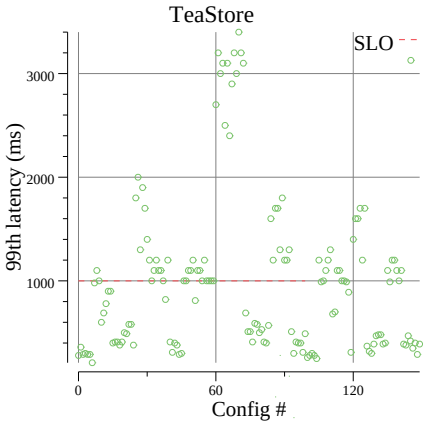

Figure 2: 99th percentile latency for all configurations evaluated during factor screening.

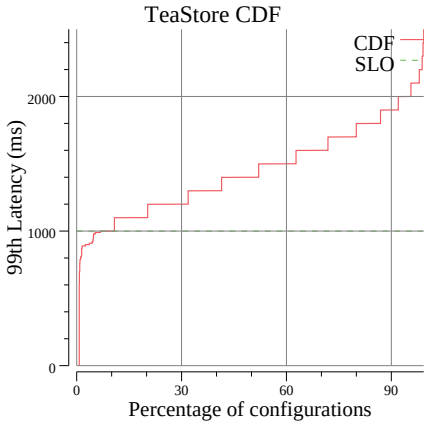

Figure 3: 99th percentile latency CDF of exhaustive search space.

4.4 Reaching the most optimal

As described in Section 2.3, we employ optimization techniques to guide our search in a cost-effective manner (i.e, the number of required performance tests). Here, we focus particularly on 1) Bayesian-optimization (BO) with expected improvement (EI) 2) BestConfig's optimization algorithm 3) Random Search, which performs random selection 4) Random search with increment (randomInc) which iterates over possible configurations with a randomly selected sequence of parameters (see Section 4.1 for more information on their implementation).

The selected optimization algorithms rely on random exploration to get good coverage of the search space. To account for this randomness in the evaluation, we collected 1,000 runs for each algorithm. Figure 4 shows the results of 1,000 runs on the dataset of each algorithm for TeaStore. The Y-axis indicates the percentage of runs that have found the most-optimal configuration (this value is known from the exhaustive search). The number of samples (or iterations of the algorithm) is limited to 100 for TeaStore. As such, the algorithms can effectively sample ≈5% of the search space. For all algorithms, it is clearly shown that the more samples are collected, the more runs find the optimal configuration. However, they differ significantly in their efficacy and speed. For both applications, BO with EI performs the best with more than 80% of the runs finding the most optimal configuration within a limited number of samples. BestConfig and Random search rarely succeed in finding the most optimal solution.

4.5 Reaching the near-optimal the most quickly

The previous section evaluated the capacity of the optimization algorithms to find the optimal configuration as found by exhaustive search. In practice, however, near-optimal configurations that satisfy the SLO with reasonable cost reduction are equally desirable. The goal is to offer guidance to select the most effective algorithm given an available sampling cost. In addition, because all algorithms have a random element, it is better to compare the worst performing run of the algorithms. Figure 5 shows the 99th percentile distance to the most-optimal configuration for the 1,000 simulated runs for each algorithm. The purple dashes indicate the distance between the utility score (1.0) at which the SLO is satisfied (as expressed by the employed utility function) and the most optimal configuration as found by exhaustive

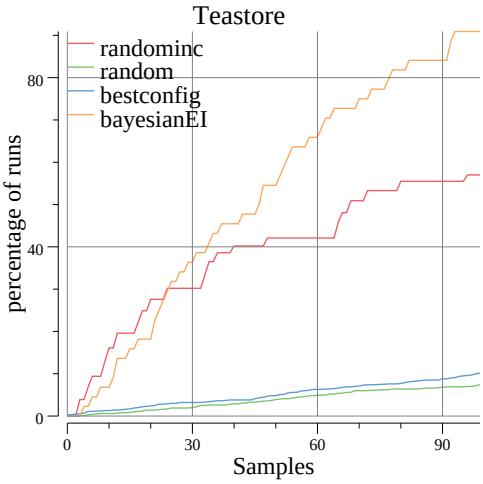

Figure 4: Comparing the efficacy of implemented optimization algorithms in finding the most optimal configuration in the reduced search space. The Y-axis shows the percentage of 1,000 runs that found the most optimal configuration within a certain amount of a samples (X-axis).

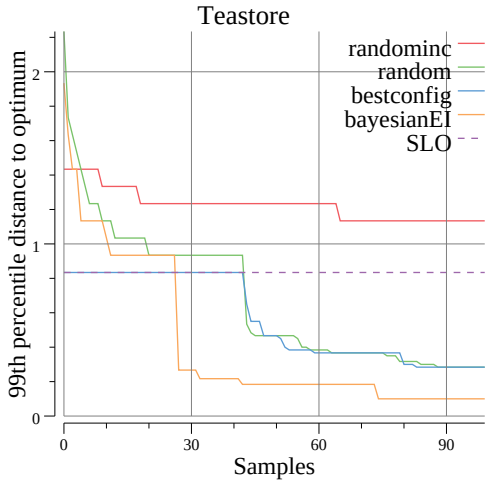

Figure 5: Comparing the worst-case efficacy of optimization methods in finding the near-optimal configurations in the reduced search space. The Y-axis shows the 99th percentile (99% of the runs find similar or better configurations) of the distance to the optimal configuration.

search.

The results show that BO with EI is the most cost-effective in finding a near-optimal configuration, but BestConfig and random search are also still cost-effective in finding such configurations. RandomInc does not always find a configuration that meets the SLO because by keeping the order of parameters after shuffling fixed, it may not find such a solution after 100 samples.

4.6 Evaluation of factor screening

Factor creening is set at 150 samples. This is an significant cost. The question arises whether applying the algorithms on the full parameter space without screening would have resulted in finding the optimal configuration faster or even finding a more optimal configuration than found by the exhaustive search within the bounds set of the screening algorithm. To answer this question, we ran the Bayesian Optimization (BO) algorithm at 150 samples on the complete search space for TeaStore before screening (cfr. the initial bounds set in Table 1). Figure 6 shows the results. The red box represents the search space selected by the screening method, with the most expensive configuration in the top red corner. The red dot shows the best result found when combining factor screening and BO, the blue dots indicate the best-found configurations by running BO on the complete search space without factor screening. This shows that standalone BO can find near-optimal solutions as well using the same cost as the screening cost. Thus it can find near-optimal solutions faster than when combining BO and screening, which supports more efficient performance optimization during the Release phase of the DevOps lifecycle, where rapid and accurate resource tuning is critical.

However, combining factor screening and BO hugely increases the probability to find the optimal

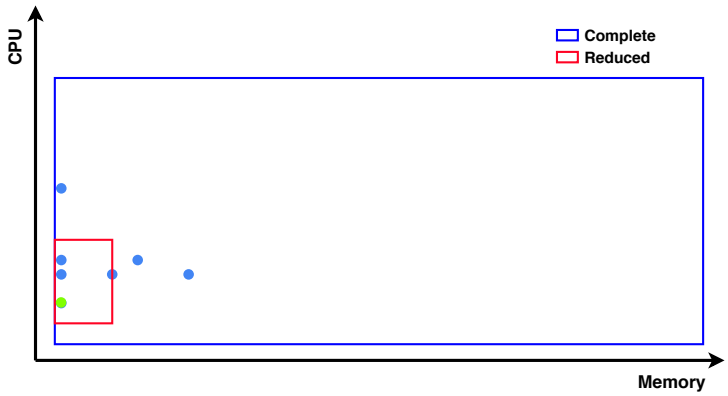

Figure 6: Results of stand-alone BO compared to combining factor screening and BO for TeaStore. The red box represents the reduced search space determined by the factor screening method. Green dot indicates the best result when combining screening and BO. Blue dots are the best found configurations by BO.

configuration, while this is not the case when running BO stand-alone. As such we conclude that factor screening remain advisable for finding the optimal configuration without caring for the sampling cost.

Of course, factor screening is also required for reducing the effort to statistically compare multiple optimization algorithms (cfr. Section 4.4). This enables fast, automated comparison of optimization strategies, which aligns with the need for efficient tooling in DevOps. Indeed, factor screening drastically reduces the search space, making it feasible to collect samples for all data points in limited amount of time, which is valuable during the Release phase where time constraints often limit the feasibility of full search strategies. Thereafter each algorithm can be evaluated very quickly by running k8-resource-optimizer from the collected dataset multiple times. Repeating 1000 runs takes about 6 minutes on average. Still, sequentially collecting all 2048 samples took more than 14 days. We also applied factor screening for TeaStore version 1.3.0 and this reduced the search space to 128 samples, requiring a full day (see Table 3 for a break-down of the durations).

	Sample	**Screening (150 samples)**	**Exhaustive**	**1000 optimization runs**
TeaStore v1.2	10 min	25 hours	14.2 days	7 min
TeaStore v1.3	10 min	25 hours	21.5 hours	5 min

Table 3: Break down duration cost.

4.7 Resource utilization

Figure 7 illustrates the resource utilization of the resource configurations that have been found by the worst-case run of the most cost-effective algorithm after the maximum amount of samples. This resource utilization has been measured using K8s' Heapster monitoring service [7]. In the case of TeaStore, a mix of high and low utilization of resources between the different services is observed. The reasons for this is twofold: firstly, the application's bottlenecks are located at the heavily utilized Authentication and WebUI service. Secondly, other services such as the Recommendation, Registry and Database services experience a low degree of utilization in terms of CPU despite although the configuration selected the

lowest possible CPU setting for these settings. This is because a higher amount of CPU is needed for starting up the Java-based services, but once the microservices are started up, the CPU usage tends to be lower. Unfortunately, at the time of the research, in K8s v1.14, it is not possible to resize the resources of containers without restarting these containers, which limits flexibility during the Release phase, when rapid adjustments with minimal disruption are crucial. But this feature is now in alpha development [36]. K8s-resource optimizer could therefore be extended so that two CPU parameters are optimized, one for starting the containers and when one warmed up and processing workload.

Setting minimal bounds is a tricky problem as a too high minimal bound may exclude the optimal configuration. This is illustrated by the low memory usage of the database component. We made the wrong assumption that the database service also ran on Tomcat but it does not. Therefore, instead of setting the same minimal bound for all microservices (cfr. Table 1), we should have tested the minimal bound for each microservice separately.

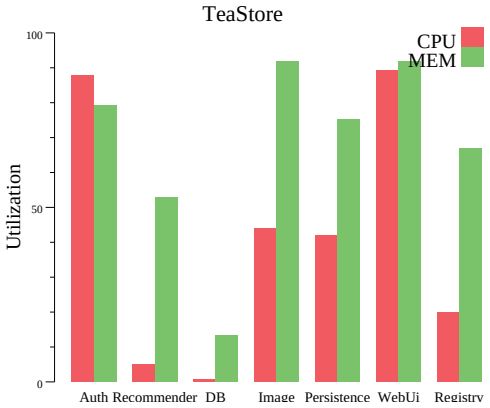

Figure 7: Worst case utilization of separate microservices after optimization.

5 Towards optimization of application-specific configurations

The evaluation results in Sections 4.6 and 4.7 demonstrate the practicality of using k8-resource-optimizer during the Release phase of the DevOps lifecycle to discover cost-efficient resource configurations for attaining performance SLOs of microservices. A key takeaway is that standalone Bayesian Optimization (BO) can be highly effective in identifying near-optimal configurations with the same sampling cost as the screening algorithm. However, when the objective shifts to finding the optimal configuration with higher certainty, or enabling statistical comparisons across optimizers, factor screening remains essential despite its upfront cost.

As discussed in Section 1, k8-resource-optimizer is not restricted to container-level resource parameters but any parameter that is configurable via the K8s API. Section 3.1 further explained how such application-level parameters can be included in the optimizer's search space via Helm chart templating. By representing them as variables in the `values.yaml` file, developers can explore a much broader configuration space. This enables optimization not just of infrastructure-level settings, but also of functional choices that may significantly affect cost-performance trade-offs.

Application-specific configurations of microservices can thus also be included in the search space. Examples of such application-specific configurations include feature toggles such as switching from high power mode to low power mode in case of resource contention. K8-resource optimizer can detect

performance invariants for such feature toggles during the release phase. For example, the following performance invariant can be directly obtained from k8-resource optimizer: *high power mode of a microservice only attains the desired SLO when CPU allocation is higher than 750 millicores*. This invariant can then be enforced at deployment time by either ensuring at least 750 millicores during deployment or by starting the microservice in low power mode.

K8-resource optimizer can also be used to better understand the performance efficiency of different alternative features. However, this increased flexibility poses challenges. The inclusion of application-specific parameters increases the dimensionality of the search space, making techniques like factor screening more valuable for pruning irrelevant parameters. Moreover, interactions between resource settings and application configurations may create complex dependencies that require joint consideration during optimization.

6 Related work

This section presents existing work in the area of performance optimization during the development stage of the DevOps life cycle. We refer the reader for an overview of performance optimization during the operational stage to existing surveys [40, 26].

Performance modeling. Previous work has proposed several methods based on performance modeling of microservice applications [18, 15]. The performance model estimates the resources required for an SLO or the maximum request rate without violating the SLO. Performance models can either be constructed using queuing theory [5] or via various supervised machine learning techniques such as neural networks or state vector machines [21]. The quality of the queuing model or machine learning model relies heavily on the expertise of the application development team. In opposition, extending k8-resource optimizer for another microservice application requires implementing an end-to-end benchmark scenario, an implementation of an appropriate utility function, and the definition of an optimizer configuration file that is based on the `values.yaml` file of the helm chart of the application. The sampling cost of 150 performance tests for factor screening or optimization is in line with the profiling/training cost of the performance modeling approaches. K8-resource optimizer also allows comparing different optimization algorithms with respect to finding the most optimal configuration or reaching a near-optimal configuration fast enough.

Performance optimization for best VM instance selection. The closed work to ours are studies on the selection of VM instances to achieve performance SLOs while minimizing costs. *Ernest* [37] can select VM sizes within a given instance family for various machine learning applications by training a common performance model with a small number of samples. The internal performance model exploits known patterns in ML applications making it poorly adaptive for other types of applications. *CherryPick* [1] utilizes Naive Bayesian optimization to find optimal or near-optimal VM instances for recurring big data analytics jobs. *Arrow* [12] introduces Augmented Bayesian Optimization which modifies off-the-shelf BO by integrating low-level performance information (e.g., CPU utilization or work memory allocation) and design choices, allowing for more informed decision making. *Scout* [14] leverages historical data from the optimization process and low-level performance metrics resulting in a more efficient search. *Micky* [13] bundles several of the above techniques in a collective optimizer to further reduce measurement costs.

There are clear analogies between k8-resource-optimizer and selecting the cost-optimal VM-instance for an application. Our work deals however with a more complex configuration search space. Moreover k8-resource-optimizer is also more reusable as it can work with, and compare different optimization

techniques. In opposition, the above works have only focused on Bayesian optimization or very similar search strategies.

Performance optimization of program configuration parameters is an active research domain. *BestConfig* [41] tunes general systems with high-dimensional parameter spaces using a recursive search with stratified sampling. Similarly, Latin Hypercube Sampling (LHS) and smart hill climbing have been used to tune an application server [39]. BOAT [8] utilizes structured Bayesian optimization while leveraging contextual information to automatically tune application performance. The recent work of *Metis* [23] enhances Bayesian optimization and demonstrates its capability to tuning the tail latency of the Bing Ads key-value store. *AdaptiveConfig* [10] combines an efficient search algorithm and a business rule engine to search at run-time for the most optimal job scheduler configuration. Einziger et al. [9] compares hill climbing and an indicator-based approach for optimization of adaptive cache management. This work discusses the configuration of the optimization algorithms such as the distance between different samples and the frequency of taking samples. Their findings show that these particular optimization algorithms do not perform differently from each other.

The main difference between k8-resource-optimizer and the above works is that the latter mainly focuses on optimizing performance, whereas we focus on optimizing performance and resource cost.

7 Conclusion

This paper has presented how to apply k8-resource-optimizer to performance optimization of containerized, microservice applications during the Release phase of the DevOps lifecyle. When the goal is to find a near-optimal configuration using a limited sampling budget, our findings show that is better to run k8-resource optimizer with bayesian optimization without the use of factor screening/sensitivy analysis. When the goal is the find a configuration that is close to the optimal solution, upfront screening to reduce the search space is advisable. Screening is also required for the statistical comparison of different optimization algorithms so that the collection of a relevant dataset of samples becomes feasible.

The primary limitation of k8-resource-optimizer is that the benchmark scenario used for performance testing is fixed. For example, consider a benchmark involving two `Experiment` classes that each trigger different API operations of the TeaStore application, with an initial workload distribution of 50% each. One `Experiment` class is primarily memory-intensive, while the other is predominantly CPU-intensive. If the overall workload intensity remains constant but the workload distribution shifts to 90%–10%, the previously determined resource configuration becomes significantly misaligned with actual demands — resulting in severe under-provisioning for one resource type and over-provisioning for the other. A promising direction for future work is therefore to develop efficient methods for evaluating a mix of different `Experiment` classes with a range of different workload distributions, thereby accounting for fluctuations in user behavior and enabling interpolation of resource allocations for untested workload distributions.

Another direction of future work is accounting for multiple versions of the software of the microservices themselves. Consider the deployment of a new version of the TeaStore microservice using a new container image, or a configuration change in the software. Currently, each software version would require a new run of k8-resource optimizer from scratch.

References

[1] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu & Ming Zhang (2017): *CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics.* In: *NSDI*, 2, pp. 4–2.

[2] Armin Balalaie, Abbas Heydarnoori & Pooyan Jamshidi (2016): *Microservices architecture enables devops: Migration to a cloud-native architecture*. *IEEE Software* 33(3), pp. 42–52.

[3] Betsy Beyer, Niall Richard Murphy, David K Rensin, Kent Kawahara & Stephen Thorne (2018): *The Site Reliability Workbook: Practical Ways to Implement SRE*. " O'Reilly Media, Inc.".

[4] Zhen Cao, Sachin Tiwari, Erez Zadok & Vasily Tarasov (2018): *Towards Better Understanding of Black-box Auto-Tuning: A Comparative Analysis for Storage Systems*. In: *USENIX Annual Technical Conference*, pp. 893–907. Available at `www.usenix.org/conference/atc18/presentation/cao`.

[5] Yuan Chen, Subu Iyer, Xue Liu, Dejan Milojicic & Akhil Sahai (2007): *SLA decomposition: Translating service level objectives to system level thresholds*. In: *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*, IEEE, pp. 3–3.

[6] Cloud Native Computing Foundation (2019): *Configure Quality of Service for Pods - Kubernetes*. Available at `https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/`.

[7] Cloud Native Computing Foundation (2019): *Tools for Monitoring Compute, Storage, and Network Resources - Kubernetes*. Available at `https://v1-10.docs.kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/`.

[8] Valentin Dalibard, Michael Schaarschmidt & Eiko Yoneki (2017): *BOAT: Building auto-tuners with structured Bayesian optimization*. In: *Proceedings of the 26th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, pp. 479–488.

[9] Gil Einziger, Ohad Eytan, Roy Friedman & Ben Manes (2018): *Adaptive Software Cache Management*. In: *Proceedings of the 19th International Middleware Conference*, Middleware '18, ACM, New York, NY, USA, pp. 94–106, doi:10.1145/3274808.3274816. Available at `http://doi.acm.org/10.1145/3274808.3274816`.

[10] R. Han, Z. Zong, L. Y. Chen, S. Wang & J. Zhan (2018): *AdaptiveConfig: Run-Time Configuration of Cluster Schedulers for Cloud Short-Running Jobs*. In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1519–1526, doi:10.1109/ICDCS.2018.00158.

[11] Jonatan Heyman, Carl Byström, Joakim Hamrén & Hugo Heyman (2019): *Locust - A modern load testing framework*. Available at `https://locust.io/`.

[12] Chin-Jung Hsu, Vivek Nair, Vincent W Freeh & Tim Menzies (2018): *Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM*. In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, pp. 660–670.

[13] Chin-Jung Hsu, Vivek Nair, Tim Menzies & Vincent Freeh (2018): *Micky: A Cheaper Alternative for Selecting Cloud Instances*. *arXiv preprint arXiv:1803.05587*.

[14] Chin-Jung Hsu, Vivek Nair, Tim Menzies & Vincent W Freeh (2018): *Scout: An Experienced Guide to Find the Best Cloud Configuration*. *arXiv preprint arXiv:1803.01296*.

[15] Anshul Jindal, Vladimir Podolskiy & Michael Gerndt (2019): *Performance Modeling for Cloud Microservice Applications*. In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, Association for Computing Machinery, p. 25–32.

[16] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni et al. (2016): *Morpheus: Towards Automated SLOs for Enterprise Clusters.* In: *OSDI*, pp. 117–134.

[17] Matthijs Kaminski, Eddy Truyen, Emad Heydari Beni, Bert Lagaisse & Wouter Joosen (2019): *A framework for black-box SLO tuning of multi-tenant applications in Kubernetes*. In: *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds*, WOC '19, Association for Com-

puting Machinery, New York, NY, USA, p. 7–12, doi:10.1145/3366615.3368352. Available at https://doi.org/10.1145/3366615.3368352.

[18] Hamzeh Khazaei, Nima Mahmoudi, Cornel Barna & Marin Litoiu (2022): *Performance Modeling of Microservice Platforms*. *IEEE Transactions on Cloud Computing* 10(4), pp. 2848–2862, doi:10.1109/TCC.2020.3029092.

[19] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann & Samuel Kounev (2018): *TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research*. In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, IEEE, pp. 223–236.

[20] Nane Kratzke & Peter-Christian Quint (2017): *Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study*. *Journal of Systems and Software* 126, pp. 1–16.

[21] Sajib Kundu, Raju Rangaswami, Ajay Gulati, Ming Zhao & Kaushik Dutta (2012): *Modeling virtualized applications using machine learning techniques*. In: *ACM Sigplan Notices*, 47, ACM, pp. 3–14.

[22] Garrett Lahmann, Thom McCann & Wes Lloyd (2018): *Container Memory Allocation Discrepancies: An Investigation on Memory Utilization Gaps for Container-Based Application Deployments*. In: *Cloud Engineering (IC2E), 2018 IEEE International Conference on*, IEEE, pp. 404–405.

[23] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang & Guangzhong Sun (2018): *Metis: Robustly Tuning Tail Latencies of Cloud Systems*. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 981–992.

[24] Tania Lorido-Botran, Jose Miguel-Alonso & Jose A Lozano (2014): *A review of auto-scaling techniques for elastic applications in cloud environments*. *Journal of grid computing* 12(4), pp. 559–592.

[25] Eddy Truyen Matthijs Kaminski (2024): *k8-scalar/k8-resource-optimizer*. https://github.com/k8-scalar/k8-resource-optimizer/. [Accessed: 2024-01-25].

[26] Sergio Moreschini, Shahrzad Pour, Ivan Lanese, Daniel Balouek-Thomert, Justus Bogner, Xiaozhou Li, Fabiano Pecorelli, Jacopo Soldani, Eddy Truyen & Davide Taibi (2025): *AI Techniques in the Microservices Life-Cycle: A Systematic Mapping Study*. *Computing* 107(100), doi:10.1007/s00607-025-01432-z. Available at https://doi.org/10.1007/s00607-025-01432-z.

[27] Max D Morris (1991): *Factorial sampling plans for preliminary computational experiments*. *Technometrics* 33(2), pp. 161–174.

[28] Ali Yadavar Nikravesh, Samuel A Ajila & Chung-Horng Lung (2015): *Towards an autonomic auto-scaling prediction system for cloud resource provisioning*. In: *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, IEEE Press, pp. 35–45.

[29] Fernando Noguiera (2019): *Bayesian Optimization*. https://github.com/fmfn/BayesianOptimization.

[30] Maria A Rodriguez & Rajkumar Buyya (2018): *Container-based cluster orchestration systems: A taxonomy and future directions*. *Software: Practice and Experience*.

[31] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand & John Wilkes (2020): *Autopilot: workload autoscaling at Google*. EuroSys '20, Association for Computing Machinery, doi:10.1145/3342195.3387524.

[32] Andrea Saltelli, Marco Ratto, Terry Andres, Francesca Campolongo, Jessica Cariboni, Debora Gatelli, Michaela Saisana & Stefano Tarantola (2008): *Global sensitivity analysis: the primer*. John Wiley & Sons.

[33] Eric Schurman & Jake Brutlag (2009): *The user and business impact of server delays, additional bytes, and HTTP chunking in web search*. In: *Velocity Web Performance and Operations Conference*.

[34] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams & Nando De Freitas (2016): *Taking the human out of the loop: A review of bayesian optimization*. *Proceedings of the IEEE* 104(1), pp. 148–175.

[35] Jasper Snoek, Hugo Larochelle & Ryan P Adams (2012): *Practical bayesian optimization of machine learning algorithms*. In: *Advances in neural information processing systems*, pp. 2951–2959.

[36] Eddy Truyen, Bert Lagaisse, Wouter Joosen, Arnout Hoebreckx & Cédric De Dycker (2021): *Flexible Migration in Blue-Green Deployments within a Fixed Cost*. In: *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*, WOC'20, Association for Computing Machinery, New York, NY, USA, p. 13–18, doi:10.1145/3429885.3429963. Available at `https://doi.org/10.1145/3429885.3429963`.

[37] Shivaram Venkataraman, Zongheng Yang, Michael J Franklin, Benjamin Recht & Ion Stoica (2016): *Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics*. In: *NSDI*, pp. 363–378.

[38] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune & John Wilkes (2015): *Large-scale cluster management at Google with Borg*. In: *Proceedings of the Tenth European Conference on Computer Systems*, ACM, p. 18.

[39] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H Xia & Li Zhang (2004): *A smart hill-climbing algorithm for application server configuration*. In: *Proceedings of the 13th international conference on World Wide Web*, ACM, pp. 287–296.

[40] Zhiheng Zhong, Minxian Xu, Maria Alejandra Rodriguez, Chengzhong Xu & Rajkumar Buyya (2022): *Machine Learning-based Orchestration of Containers: A Taxonomy and Future Directions* 54(10s). doi:10.1145/3510415.

[41] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song & Yingchun Yang (2017): *Bestconfig: tapping the performance potential of systems via automatic configuration tuning*. In: *Proceedings of the 2017 Symposium on Cloud Computing*, ACM, pp. 338–350.

A `Values.yaml` file for TeaStore Deployment

Listing 1: Helm chart values for deploying TeaStore in K8s.

```
namespace: teastore
version: 1.2.0 #TeaStore version=1.2.0

persistenceReplicas: 1
persistenceCpu: 1000m
persistenceMemory: 1000Mi

webuiReplicas: 1
webuiCpu: 1000m
webuiMemory: 1000Mi

recommenderReplicas: 1
recommenderCpu: 1000m
recommenderMemory: 1000Mi

registryReplicas: 1
registryCpu: 500m
registryMemory: 1000Mi

dbReplicas: 1
dbCpu: 1000m
dbMemory: 1000Mi
```

```
authReplicas: 1
authCpu: 1000m
authMemory: 1000Mi

imageReplicas: 1
imageCpu: 1000m
imageMemory: 1000Mi
```

B Optimizer Configuration for TeaStore

Listing 2: Configuration for k8-resource-optimizer targeting the TeaStore deployment.

```
nbOfIterations: 10
nbOfSamplesPerIteration: 6
charts:
  - name: teastore
    chartdir: charts/teastore-helm
slas:
  - name: silver
    chartName: teastore
    slos:
      throughput: 0.5
      99th: 1000.0
    nbOfTenants: 10
    parameters:
      - name: persistenceCpu
        searchspace:
          min: 500
          max: 1125
          granularity: 125
        suffix: m
      - name: persistenceMemory
        searchspace:
          min: 512
          max: 1152
          granularity: 128
        suffix: Mi
      - name: webuiCpu
        searchspace:
          min: 500
          max: 1125
          granularity: 125
        suffix: m
      - name: webuiMemory
        searchspace:
          min: 512
          max: 1152
          granularity: 128
        suffix: Mi
      - name: recommenderCpu
```

```
      searchspace:
        min: 500
        max: 1125
        granularity: 125
      suffix: m
    - name: recommenderMemory
      searchspace:
        min: 512
        max: 1152
        granularity: 128
      suffix: Mi
    - name: registryCpu
      searchspace:
        min: 500
        max: 1125
        granularity: 125
      suffix: m
    - name: registryMemory
      searchspace:
        min: 512
        max: 1152
        granularity: 128
      suffix: Mi
    - name: dbCpu
      searchspace:
        min: 500
        max: 1125
        granularity: 125
      suffix: m
    - name: dbMemory
      searchspace:
        min: 512
        max: 1152
        granularity: 128
      suffix: Mi
    - name: authCpu
      searchspace:
        min: 500
        max: 1125
        granularity: 125
      suffix: m
    - name: authMemory
      searchspace:
        min: 512
        max: 1152
        granularity: 128
      suffix: Mi
    - name: imageCpu
      searchspace:
        min: 500
        max: 1125
        granularity: 125
```

```
      suffix: m
    - name: imageMemory
      searchspace:
        min: 512
        max: 1152
        granularity: 128
      suffix: Mi

namespaceStrategy: NSPSLA
optimizer: bestconfig
utilFunc: teastore
outputDir: teastore-sensitivy
```

C Template a String based parameter in an helm chart for K8-resource optimizer

Listing 3: `values.yaml`

```
mySettingChoice: 1
```

Listing 4: `values.schema.yaml`

```
properties:
  mySettingChoice:
    type: integer
    enum:
      - 1
      - 2
    description: |
      1 = SimpleSetting
      2 = ComplexSetting
```

Listing 5: `deployment.yaml` file containing mySettingChoice parameter

```
# templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
```

```
      - name: myapp
        image: myapp:latest
        env:
          - name: MY_SETTING
            value: |
              {{ index (dict 1 SimpleSetting 2 ComplexSetting) .Values.mySettingChoice
                   }}
```

D Implemented benchmark scenario

Step	Task	What happens
0	user	Picks a random user postfix number (e.g., user37, user58, etc.)
1	index	Visits the homepage (GET /)
2	loginPage	Visits the login page (GET /login)
3	loginAction	Logs in with a username/password. Parses the response to find category links
4	seeCategoryPage	Visits a random category page parsed from the login response
5	seeProductPage	From the category page, picks a random product page
6	addToCart	Adds the product to the cart (GET /cartAction?addToCart&productid=X)
7	seeCategoryPage2	Visits another category page (repeat)
8	seeProductPage2	Visits another product page (repeat)
9	addToCart2	Adds another product to the cart
10	getProfile	Visits the user's profile page (GET /profile)
11	startAndLogout	Repeats homepage visit, then logs out (GET /loginAction?logout=) and clears cookies

Table 4: Simulated User Flow: Step, Task, and Description. A waiting time of 0ms is set between tasks.