A framework for black-box SLO tuning of multi-tenant applications in Kubernetes

Matthijs Kaminski, Eddy Truyen, Emad Heydari Beni, Bert Lagaisse, Wouter Joosen
imec-DistriNet, KU Leuven
firstname.lastname@cs.kuleuven.be

Abstract

Resource management concepts of container orchestration platforms such as Kubernetes can be used to achieve multi-tenancy with quality of service differentiation between tenants. However, to support cost-effective enforcement of Service Level Objectives (SLOs) about response time or throughput, an automated resource optimization approach is needed for mapping custom SLOs of different tenants to cost-efficient resource allocation policies. We propose a versatile tool for cost-effective SLO tuning, named k8-resource-optimizer, that relies on black-box performance tuning algorithms. We illustrate and validate the tool for optimizing different resource configuration properties of a simple job processing application. Our experiments showed that k8-resource-optimizer can find near-optimal configurations for different multi-tenant deployment settings and different types of resource parameters. However an open research challenge is that, when the number of parameters increases, the total tuning cost may also increase beyond what is acceptable for contemporary cloud-native applications. We shortly discuss three possible complementary solutions to tackle this challenge.

CCS Concepts • **Software and its engineering → Software performance**; *Cloud computing*; *Software as a service orchestration system.*

Keywords Black-box optimization, Container orchestration platforms, Multi-tenancy

ACM Reference Format:
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 *5th International Workshop on Container Technologies and Container Clouds (WOC '19), December 9–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3366615.3368352

1 Introduction

Cloud-native platform and application providers face the constant challenge of offering their products at the best service level for the most competitive price. To further minimize their costs, the architectural design principle of multi-tenancy is employed. Multi-tenancy attempts maximizing the sharing of resources among multiple customers organizations, referred to as tenants [20].

The cost-effective management of custom service-level agreements (SLAs) for different tenants typically requires dealing with two challenges: (i) offering custom SLAs to different tenants (QoS differentiation) and (ii) achieving improved resource utilization of servers (cost-efficiency) [10].

Recently, container technology and container orchestration frameworks have been positioned as a possible solution for supporting QoS differentiation with improved cost-efficiency by means of finding the minimal cluster size to fit tenants with heterogeneous workloads [19]. However, this approach assumes that application managers must specify optimal resource allocation policies for the application components so that SLOs can be met in a cost-effective manner. A user study conducted by Microsoft [9] found that 70% of jobs submitted to a production cluster were over-provisioned. And for 20% of the jobs 10x more resources than necessary were allocated.

The problem of finding a cost-effective assignment of resource allocation policies so that an SLA can be met, has been referred to as the SLA-decomposition problem [2]. However, existing SLA-decomposition techniques are not easy to reuse across different deployment settings of the same application due to the fact that a performance model of the deployed application must be created in advance based on proven theories such as queuing theory; moreover this model might be wrong and thus will lead to incorrect resource allocations. For example, the developer needs to know if the model is quadratic or linear but predicting this has shown to be a difficult task with a large number of application components in deployments. Lastly, the performance models often require the configuration of hyper-parameters, which is a tuning problem itself [24].

In this paper we explore how the SLA-decomposition problem can be solved for container-based applications by applying *black-box performance optimization algorithms*. Such algorithms test a limited number of container resource configurations of a running application in order to find a near optimal resource configuration. For this purpose we have designed and implemented a resource optimization framework, named k8-resource-optimizer, that is capable of automatically deriving a cost-efficient SLA-decomposition using existing black-box optimization algorithms such as the BestConfig algorithm [24] and bayesian optimization [1]. We have implemented k8-resource-optimizer on top of Kubernetes, the leading platform in container orchestration and we have evaluated the feasibility of the approach in the context of a simple job processing application.

Of course, black-box performance optimization is not a new idea. It has been researched extensively for performance tuning of middleware and databases [21, 24] and bayesian optimization has been shown effective in the selection of VM-instances for data-analytics [1, 8]. However, deploying and running black-box performance optimization techniques is not trivial and actually requires quite some expertise from the user. Therefore, the main contribution of k8-resource-optimizer is that it offers a certain level of

WOC '19, December 9–13, 2019, Davis, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7033-2/19/12...$15.00
https://doi.org/10.1145/3366615.3368352

abstraction and automation to facilitate the process of employing these techniques for multi-tenant container deployments.

The remainder of this paper is structured as follows. Section 2 presents the necessary background information for the paper. Section 3 discusses relevant related work and gives insights into the several black-box optimization techniques that can be used for SLA-decomposition. Section 4 presents the design and implementation of the k8-resource-optimizer framework. Next, in Section 5 the performance of the k8-resource-optimizer framework is evaluated in the context of a multi-tenant SaaS application for job processing. Subsequently, Section 6 presents possible solutions for reducing the total tuning cost. Finally, our conclusion and future work are presented in Section 7. A demo version of k8-resource-optimizer is available as Docker image `decomads/k8-resource-optimizer` at Docker Hub.

2 Background

2.1 Kubernetes

Kubernetes [6] is one of the most popular and adopted orchestration platforms. Kubernetes introduces a number of concepts for both containers and cluster resources [16]. It allows to setup and manage container-based applications via declarative configuration files. Below are the most relevant configuration concepts introduced. These concepts are also supported by other container orchestration platforms such Docker Swarm and Mesos [16].

Pods. A pod is the smallest unit of deployment within Kubernetes. It is a group of containers that logically belong together and are always executed on the same node. A pod is also a unit of failure.

Deployments. A Deployment controller manages a ReplicaSet of pods. A ReplicaSet allows pods to be replicated across multiple nodes. A Deployment object is used to specify the desired state of pods (e.g. number of replica's) [3].

Namespaces. Namespaces allow to partition resources of a physical cluster among multiple tenant organizations. Each namespace gets a share of the resources of the cluster, via resource quotas. Resource quotas are supported for CPU, memory and persistent volumes [4].

Resource requests and limits. Kubernetes allows the allocation of compute resources to both containers and the enclosing Pods by means of guaranteed *resource requests* and *maximum resource limits*. The currently supported compute resources are CPU, memory and storage within the root partition of the local node [5].

There are possible multi-tenant deployment strategies to employ the above container orchestration concepts. These strategies differ in the chosen trade-off between cost-efficiency and security isolation [17]:

Namespace per tenant. Each tenant is assigned to its own separate, dedicated namespace. This allows specifying the exact resource requests and limits of pods needed to satisfy the SLOs of the tenant. The advantage of this approach is improved security and performance isolation between tenants, but this approach is less cost-efficient. Typically this strategy is most suitable for Platform-as-a-Service.

Namespace per SLA class. Tenants are grouped in different SLA classes (e.g. gold, bronze). A separate, dedicated namespace with the corresponding quota is created for all the tenants in each SLA class. The advantage of the approach is that the sharing of a container among tenants of the same SLA class allows for a higher resource utilization, but there is no container security and performance isolation between different tenants that belong to the same SLA class. This strategy is most suitable for Software-as-a-Service applications with a front-end rate limiter for admission control of aggressive tenants, but some tenants of the SaaS application may still prefer the isolation of a separate container [17].

The SLA-decomposition problem for container orchestration platforms can then be defined as follows: *Given (1) different Deployments that belong to different Namespaces, (2) a specific workload for each of these Deployments, and (3) a custom SLO for each Deployment, find the minimum resource requests and limits for the Pods in each Deployment, so that the SLOs of the Deployments are met.*

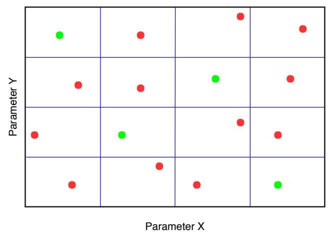

Figure 1. Illustration of BestConfig's DDS on a 2D parameter space. Dots represent all combination of sampled parameter spaces. Blue lines indicate sampling intervals. Green dots are the selected dots after random divergence.

2.2 BestConfig

In this paper we use a variant of hypercube-sampling, named BestConfig [24], for solving the above SLA-decomposition problem. This technique treats the application as a black box and does not require the upfront construction of a performance model of the application. Its algorithm optimizes towards a single, scalar performance metric. This metric is calculated by an *utility function* that has user-concerned performance goals such as latency and resource cost as input parameters.

BestConfig consists of an effective sampling method called *divide-and-diverge sampling* (DDS) and *recursive-bound-and-search* (RBS), a search-based optimization algorithm.

2.2.1 Divide-and-diverge

BestConfig can support a *high-dimensional parameter space* in order to handle systems with a lot of performance-sensitive parameters. To limit the number of required samples k, the high-dimensional parameter space is divided into subspaces. Given n parameters, each parameter's range is divided into k intervals, resulting into k^n possible parameter combinations that can be sampled. The authors show that a good coverage of these parameter combinations can be attained by *divergence* of the sample set, i.e. representing each interval of each parameter exactly once in the sample set. Figure 1 illustrates divergence: the green dots are the k selected samples to be tested.

2.2.2 Recursive Bound & search

RBS is a search-based performance optimization algorithm that defines the area of the parameter space near the sample $C_i \in C_{0..k-1}$ that has been awarded the highest score by the utility function. This area is referred to as the *bounded space* for the next iteration of

DSS. When DDS cannot find a better optimum than C_i, BestConfig will backtrack to the bounded space of the previous iteration.

3 Related work

Black-box tuning for best VM instance selection and container resource scaling The closest work to k8-resource-optimizer are black-box auto-tuning approaches for the selection of VM instances under performance guarantees while minimizing costs. *Ernest* [18] 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. *CherryPick* [1] utilizes Naive bayesian optimization, a technique for optimizing black-box functions, to find optimal or near-optimal VM instances for recurring big data analytics jobs. *Arrow* [8] introduces Augmented Bayesian Optimization which modifies off-the-shelf bayesian optimization by integrating low-level performance information (e.g., CPU utilization or work memory allocation) to faster find near-optimal solutions. Recent works in the space of container-based cluster computing extend a static performance optimization phase with adaptive resource reconfiguration at run-time. *MIRAS* [23] employs model-based reinforcement learning to reduce the number of performance tests. *MEER* [22] even dismisses the feasibility of performance optimization based on utility functions. Instead, it only requires two static performance tests to determine statistical confidence intervals for the optimal memory resource limit of containers in Apache Spark; these intervals are further fine-tuned at run-time. For multi-tenant container-based SaaS applications [17], however, static performance optimization remains an important tool for SLO management provided that performance SLOs enforce a maximum request rate upon tenants.

Black-box tuning of program configuration parameters towards a specific performance objective is an active research domain. *BestConfig* [24](see also Section 2.2) tunes general systems with high-dimensional parameter spaces using a recursive search with stratified sampling. Similarly, Latin Hypercube Sampling (LHS) is able to tune a large-number of orthogonal parameter spaces [21]. These techniques are relevant as container-based applications also have a larger multi-dimensional search space (multiple types of resources, multiple types of Pod deployments) and the numerical ranges for CPU and memory are also more fine-grained. 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 ànd resource cost.

4 K8-resource-optimizer

4.1 Architecture and implementation

This section presents the architecture of k8-resource-optimizer (see Figure 2). The k8-resource-optimizer framework consists of two major components, *SLA-decomposer* and *Bench*. The *SLA-decomposer* is responsible for the translation of SLOs to resource allocation policies for a container-based multi-tenant application. It takes a user-specified input configuration file and transforms it to a benchmark plan. The *Bench*-component uses this benchmark plan to search for the optimal resource configuration using a sequence of performance tests. The Bench component can simultaneously test different resource configurations in multiple namespaces. It further depends on the deployment manager for the process of shutting down and restarting containers with a new resource configuration while maintaining loose coupling with the underlying container orchestration framework. Finally it depends on a load testing framework for testing the performance of a resource configuration.

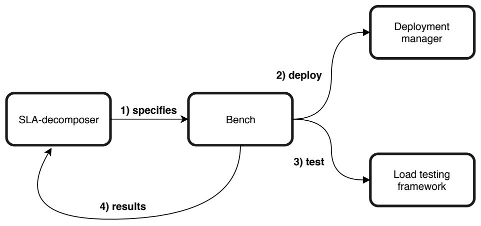

Figure 2. The architecture of k8-resource-optimizer.

Listing 1. Input for the SLA-decomposer.

```
---
charts:
  - name: my-app
    chartdir: /exp/conf/helm/mychart
slas:
  - name : silver
    chart: my-app
    jobsize: 500
    throughput: 0.5
    nbOfTenants: 3
    parameters:
      - name: workerCPU
        searchspace:
          min: 200
          max: 750
          granularity: 5
        prefix:
        suffix: m
      - name: workerReplicas
        searchspace:
          min: 2
          max: 4
          granularity: 1
  - name : bronze
    chart: my-app
    jobsize: 250
    throughput: 0.5
    nbOfTenants: 2
    parameters:
      - name: workerCPU
        ...
      - name: workerReplicas
        ...
namespaceStrategy: NSPSLA
optimizer: bestconfig
nbOfIterations: 2
nbOfSamplesPerIteration: 2
utilFunc: resourceBased
outputDir: datasets/my-app/bestconfig
```

This architecture and its Kubernetes-specific instantiation have been implemented in the Go programming language. Existing load testing frameworks can be integrated into the Bench component. In this paper we use the Locust framework, which is a distributed user load testing framework intended for load-testing different types of systems [11]. The implementation of the deployment manager component relies on Helm [7].

4.2 SLA-decomposition

To illustrate the concepts of SLA decomposition, Listing 1 shows an example of an input configuration for the SLA-decomposer component. A configuration consists of **an application specification** that is to be deployed in one or more namespaces. Kubernetes applications are specified as Helm charts [7].

Thereafter **one or more SLAs** are specified. Each SLA specifies the following options: Service Level Objectives to be met and the amount of tenants. The service level objectives such as throughput and job size are application-specific and are therefore supported by an application-specific workload generator within the load testing framework.

Then **the resource parameters** to be tuned are specified. For each parameter, a search space is defined which restricts the upper and lower bounds of the parameter, and a granularity for iterating over the values in this space. It is also possible to specify a necessary prefix or suffix for the parameter (e.g., suffix m for 500m CPU). The Kubernetes-specific extension of the Deployment manager can non-intrusively set the values of any numerical parameters in any Kubernetes configuration files by relying on the *chart template* feature of Helm, which expects that all tunable parameters are specified in a separate `Values` file. For example, the `workerCPU` parameter is nested inside a regular Kubernetes YAML file for a Deployment as follows:

```
replicas: {{.Values.workerReplicas}}
...
  resources:
    requests:
      cpu: {{.Values.workerCPU}}
      memory: 500Mi
    limits:
      cpu: {{.Values.workerCPU}}
```

Then, **the multi-tenant deployment strategy** is configured. This is either *Namespace per SLA* or *Namespace per tenant* as described in Section 2.

The next part of the input configuration concerns **the selection of the optimization algorithm** and the *number of iterations and samples* that the optimization algorithm must take. The bench component currently supports the following algorithms. Hereby the notion of iterations and samples is generic enough to subsume the different algorithmic structures:

- BestConfig: during each iteration, divide-and-diverge sampling first takes the given number samples in the given parameter search space. Then, recursive-bound-and-search is applied to narrow down the search space around the best sample or backtrack to the previous search space.
- BayesianFmfn: for the implementation of bayesian optimization we rely on a popular open-source Python library. During each iteration, one sample configuration is selected by means of the *expected improvement acquisition function* [15].
- Random (incremental) search: iterations map to the amount of samples tested. The samples argument has no effect. Incremental search means that values of the same parameter across successive iterations are always incremented, but which parameter to increment is randomized.
- Exhaustive search: samples are calculated based on the total possible combinations of parameters to be tuned.

Finally, the **utility function** is a scalar function. It translates multiple user-concerned performance goals to a single value. In the case of SLA-decomposition where SLOs have to be met with a minimal set of resources, the utility function first dismisses configurations for which the SLA is violated. For the configurations that do satisfy the SLA, the utility function creates a scalar value in terms of resource consumption. In this function, a lower resource consumption leads to a better score and each resource parameter can be assigned a weight according to the preference of the user:

$$f([]SLOmetrics, []resoureLimits) = \begin{cases} 0 \text{ if SLA is violated} \\ \sum_{r=1}^{len(resourceLimits)} (1 - \frac{current(r_i)}{upperbound(r_i)}) \times weight(r_i) \\ \text{if SLO is met} \end{cases}$$

5 Evaluation

We have evaluated the correctness and performance of k8-resource-optimizer in for multiple resource parameters, multiple tenants and multiple specific SLAs. Due to space limitations, the evaluation is restricted to the scenario where multiple parameters for multiple tenants with the same SLA are optimized[1]. All experiments are conducted with the BestConfig algorithm (see Section 2.2) on a single-node Kubernetes cluster (version 1.8.0) inside a VM using Minikube [13] (version 0.24.1). The VM is allocated 4 cores and 8192 MB memory. The underlying hardware utilizes a 2.6GHz hyper-threading quad-core processor.

5.1 Application scenario

We have implemented a simple job processing application to evaluate the feasibility of k8-resource-optimizer. The application consists of two major components: the Queue and Workers. The design is shown in Figure 3.

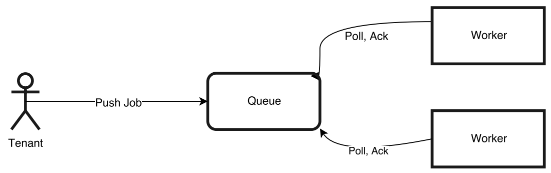

Figure 3. Design simple job processing application.

The components of the application are implemented as restful web services in the Java Spring Boot Framework. The queue offers a REST API for pushing jobs on the queue and pulling tasks from the queue: a job consists of a certain amount of tasks. The worker will poll in fixed time interval (1 second) or after completing the current task.

We assume that all tasks are the same. This assumption is valid for traditional cluster scheduling frameworks such as Hadoop

[1]The thesis text, which presents all validated scenarios, is available at https://github.com/k8-scalar/k8-resource-optimizer/

but not for contemporary frameworks such as Spark [22]. Moreover we assume tasks are CPU-intensive. As such, workers implement a CPU-stressing computation that is inspired by the work of Matthews et al. [12] on quantifying the performance isolation of virtualization systems. The authors suggest to utilize a tight loop of integer arithmetic operations; more specifically the factorial of 30 is recursively calculated in the loop. The number of loops can be configured.

Note, k8-resource-optimizer does not only focus on applications of the job type, but also targets user-facing web applications where the SLO is specified in terms of response latency.

5.2 Multiple parameters, multiple tenants with same SLA

The goal of the following experiment is to test if k8-resource-optimizer can be used to tune two different types of resource parameters simultaneously: `workerCPU` and `workerReplicas`. For CPU, Kubernetes `requests` are set equal to `limits` to ensure proper CPU isolation between containers when the number of co-located containers on a node evolves.

Experiment setup For this experiment, an SLA-decomposition is performed for three silver tenants but no bronze tenants. The used performance SLO and multi-tenancy strategy are shown in Listing 1). The namespace per SLA class strategy is employed as multi-tenancy strategy. Therefore, all tenants submit requests to the same application instance. The search space for the `workerCPU` parameter is `200 - 750` and `2 - 4` for the `workerReplicas` parameter. This bounds for this search space have been chosen by means of trial and error. To speed-up the optimization, a constraint must be added to exclude wasteful resource configurations from the parameter surface. The constraint is $300 < \texttt{workerCPU} \times \texttt{workerReplicas} < 2000$. This constraint is currently to be implemented directly into the Bench component. With a granularity for CPU of 5 Millicores, there remain 254 combinations of the two parameters. The optimization will perform 5 iterations. Each iteration tests 3 configurations samples (pair of silver and bronze setting). Resulting in 15 configurations to be tested in total.

Results The results produced by k8-resource-optimizer are shown in Table 1. The increase in best score is given. During each iteration the bounded search space determined by DDS becomes more narrow as samples are located more closely together. The best score is found by RDS during the fifth iteration. The execution of the experiment took approximately 2 hours.

Table 1. Experiment 2: Results of k8-resource-optimizer on 15 samples in 5 iterations. Job size is 500 tasks and the targeted throughput is 0.5 jobs per second.

ITERATION: 1

workerReplicas	workerCPU	Score	Throughput (Jobs/s)	Total CPU
2	635	3.18	0.63	1270
4	255	0	0.33	1020
3	380	0	0.46	1140

ITERATION: 2

workerReplicas	workerCPU	Score	Throughput (Jobs/s)	Total CPU
2	615	3.22	0.6	1230
2	640	3.17	0.63	1280
3	390	0	0.47	1170

ITERATION: 3

workerReplicas	workerCPU	Score	Throughput (Jobs/s)	Total CPU
3	510	0	0.49	1530
2	595	3.26	0.59	1190
2	440	0	0.4	880

ITERATION: 4

workerReplicas	workerCPU	Score	Throughput (Jobs/s)	Total CPU
2	580	3.29	0.55	1160
2	520	0	0.49	1040
3	605	2.57	0.75	1815

ITERATION: 5

workerReplicas	workerCPU	Score	Throughput (Jobs/s)	Total CPU
2	560	3.34	0.56	1120
2	**530**	**3.42**	**0.51**	**1060**
3	590	2.6	0.77	1770

6 Towards fast optimization in continuous delivery

The experimental findings with a job processing application in multiple Kubernetes deployment scenarios have shown that k8-resource-optimizer is capable of finding cost-effective SLA-decompositions for two parameters in a limited number of samples.

Whilst further evaluation with different applications is needed, k8-resource-optimizer seems to have potential in the testing phase of a continuous delivery pipeline that is typically used in mature DevOps organizations. To make k8-resource optimizer itself cost-effective in this DevOps context, however, the total auto-tuning time when targeting high-dimensional parameter spaces must be significantly reduced. As demonstrated in section 3, a vast amount of work has already proposed faster optimization algorithms for selecting optimal VMs. Still, we believe it is worthwhile to explore the following three complementary improvements that can be applied together to tackle this open challenge.

The first possible solution is to apply **sensitivity analysis algorithms** to distinguish performance-sensitive parameters from those parameters that are not sensitive to trade-offs between resource cost and performance, and only apply optimization to the former. For example, we ran k8-resource-optimizer with the well-known elementary effects algorithm [14] on an extended version of the job processing application. It took approximately 70 performance tests of 8 minutes in order to obtain significant measures for 6 resource parameters (cpu / memory of 3 Pods). Thus, existing sensitivity analysis algorithm are also too costly in finding good samples. Surprisingly, there is little current research on improving the performance of sensitivity analysis.

A second solution starts from the hypothesis that which optimization algorithm is the fastest in finding a *good enough* resource configuration, depends on the specific application. Therefore, it is plausible to perform a once-off effort to **statistically compare different algorithms** in terms of their trade-off between the number of required samples and the distance to the optimal resource configuration. With this end in view, k8-resource optimizer can be executed in off-line mode, i.e. samples are taken from a database with exhaustively collected performance tests within a reduced search space that is determined by the aforementioned sensitivity analysis algorithm. For this comparison, we recommend an utility function that also allows to distinguish resource configurations that only slightly violate the SLO. For example, the function can assign a score $1 + d$ where d is equal to the distance between the 99th percentile of the measured performance and the SLO. Configurations that do satisfy the SLO should be assigned a score between $[0, 1]$, where the lower resource allocation cost receives a lower score. Figure 4 shows for all algorithms a statistical summary of the 10 worst results out of 1000 off-line runs for the extended version of the job processing application using this utility function. The Y-axis shows the 99th percentile distance with the optimal configuration

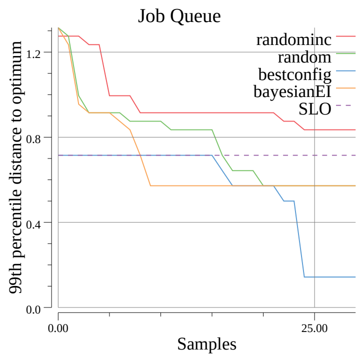

Figure 4. Comparing the worst-case efficacy of optimization methods in finding the near-optimal configurations in the reduced search space. The Y-axis shows the 99th percentile of distance to the optimum solution (99% of the runs find similar of better configurations).

as found by exhaustive search. This distance is calculated as the difference in respective utility scores. The area below the purple-dashed horizontal line corresponds with all resource configurations for which the SLO is satisfied. BestConfig is shown to be the fastest in finding configurations that are close to the cost-optimal one. Moreover bayesian optimization is fast in finding configurations that meet the SLO but is not able to find cost-optimal configurations at all. This is because bayesian optimization has no automatic backtracking as BestConfig does (see Section 2.2). Therefore, runs with bad seeds may get stuck in a local optimum. As Figure 4 plots the 99th percentile distance, the performance of these bad runs is essentially shown.

The third possible solution is to develop **application-specific deployment tactics** so it becomes possible to apply the optimization to coarse-grained deployment units of the application instead of fine-grained resource parameters. For example, specifically related to job processing applications, jobs could be executed by a small set of heterogenous Worker deployments that are ranked in terms of CPU and memory size from small sized Pods to large-sized pods. The optimization boils then down to finding the optimal amounts of Pod replicas for each of the deployments. Generic support for application-specific customizations to optimization algorithms (such as the additional constraint in the second experiment) also falls under the scope of this solution.

7 Conclusion

Existing approaches for SLA-decomposition require extensive domain expertise and struggle with the conceptual gap between the performance model of the application and the true performance characteristics of the application. As a result, black-box optimization algorithm that do not require a model of the application have gained increasing attention in recent research. This paper has proposed the k8-resource-optimizer framework that is specifically designed for SLA-decomposition in multi-tenant container-based cloud environments. K8-resource-optimizer works with multiple black-box optimization algorithms and has been implemented and evaluated in Kubernetes. Moreover, the k8-resource-optimizer framework can also be customized for other container orchestration platforms and it can be integrated with multiple performance testing tools. An open research challenge involves managing the trade-off between reducing the total tuning cost and finding near-optimal resource configurations. In ongoing work we explore multiple complementary roads to this problem.

Acknowledgments

We thank the reviewers for their helpful comments. This research is partially funded by the Research Fund KU Leuven and the IWT-DeCoMAds project.

References

[1] Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., and Zhang, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI* (2017).
[2] Chen, Y., Iyer, S., Liu, X., Milojicic, D., and Sahai, A. SLA decomposition: Translating service level objectives to system level thresholds. In *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on* (2007), IEEE.
[3] Foundation, C. N. L. Kubernetes deployments, 2018. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.[Online; Accessed May 18, 2018].
[4] Foundation, C. N. L. Kubernetes namespaces, 2018. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.[Online; Accessed May 18, 2018].
[5] Foundation, C. N. L. Managing compute resources for containers, 2018. https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/.[Online; Accessed May 18, 2018].
[6] Foundation, C. N. L. What is kubernetes?, 2018. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.[Online; Accessed May 18, 2018].
[7] Helm. Helm - the kubernetes package manager, 2018. https://helm.sh/.[Online; Accessed May 17, 2018].
[8] Hsu, C.-J., Nair, V., Freeh, V. W., and Menzies, T. Arrow: Low-level augmented bayesian optimization for finding the best cloud vm. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)* (2018), IEEE.
[9] Jyothi, S. A., Curino, C., Menache, I., Narayanamurthy, S. M., Tumanov, A., Yaniv, J., Mavlyutov, R., Goiri, I., Krishnan, S., Kulkarni, J., et al. Morpheus: Towards automated slos for enterprise clusters. In *OSDI* (2016).
[10] Krebs, R., Momm, C., and Kounev, S. Metrics and techniques for quantifying performance isolation in cloud environments. *Science of Computer Programming 90* (2014).
[11] Locust.io. Locust: An open source load testing tool, 2018. https://locust.io/. [Online; Accessed July 16, 2018].
[12] Matthews, J. N., Hu, W., Hapuarachchi, M., Deshane, T., Dimatos, D., Hamilton, G., McCabe, M., and Owens, J. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 workshop on Experimental computer science* (2007), ACM.
[13] Minikube developers. Minikube: running kubernetes cluster locally, 2018. https://github.com/kubernetes/minikube. [Online; Accessed July 16, 2018].
[14] Morris, M. D. Factorial sampling plans for preliminary computational experiments. *Technometrics 33*, 2 (1991).
[15] Noguiera, F. Bayesian optimization. https://github.com/fmfn/BayesianOptimization, 2019.
[16] Truyen, E., Van Landuyt, D., Preuveneers, D., Lagaisse, B., and Joosen, W. A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks. *Applied Sciences 9*, 5: 931 (2019).
[17] Truyen, E., Van Landuyt, D., Reniers, V., Rafique, A., Lagaisse, B., and Joosen, W. Towards a container-based architecture for multi-tenant saas applications. In *Proceedings of the 15th International Workshop on adaptive and reflective middleware* (December 2016), ARM 2016, ACM.
[18] Venkataraman, S., Yang, Z., Franklin, M. J., Recht, B., and Stoica, I. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI* (2016).
[19] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM.
[20] Walraven, S., Truyen, E., and Joosen, W. A middleware layer for flexible and cost-efficient multi-tenant applications. *Middleware 2011 LNCS 7049* (2011).
[21] Xi, B., Liu, Z., Raghavachari, M., Xia, C. H., and Zhang, L. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th international conference on World Wide Web* (2004), ACM.
[22] Xu, G., and Xu, C. MEER : Online Estimation of Optimal Memory Reservations for Long Lived Containers in In-Memory Cluster Computing. In *ICDCS 2019* (2019).
[23] Yang, Z., Nguyen, P., Jin, H., and Nahrstedt, K. MIRAS : Model-based Reinforcement Learning for Microservice Resource Allocation over Scientific Workflows. In *ICDCS 2019* (2019).
[24] Zhu, Y., Liu, J., Guo, M., Bao, Y., Ma, W., Liu, Z., Song, K., and Yang, Y. BestConfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM.