Feasibility of container orchestration for adaptive performance isolation in multi-tenant SaaS applications

Eddy Truyen, André Jacobs, Stef Verreydt, Emad Heydari Beni, Bert Lagaisse, Wouter Joosen
imec-DistriNet, KU Leuven
firstname.lastname@cs.kuleuven.be

Abstract

SaaS application instances typically serve multiple tenants to improve cost-efficiency. This results in the need for adaptive performance isolation between tenants in order to guarantee custom service level objectives (SLOs) about request latency or throughput. Current solutions, which are based on request scheduling algorithms, suffer from *SLO instability* under globally varying workloads. This means that the configuration for an SLO has to be recalibrated when total workload patterns change such as an increase or decrease in the number of subscribed tenants, or the application becomes co-located with other types of resource-intensive applications. Lately container technology such as Docker and container orchestration frameworks like Kubernetes have been used to increase cost-efficiency, multi-tenancy and elasticity. This paper investigates if the problem of adaptive performance isolation can be mapped to resource management concepts of Kubernetes through a series of experiments. These experiments show that Kubernetes provides good support for QoS differentiation and adaptive resource allocation by grouping tenants according to their SLO class (e.g gold vs bronze) in different containers. Moreover, SLO instability does not occur when co-locating these containers with other container-based applications provided that a few interferences between CPU-, memory- and disk-io intensive applications are taken into account. However SLO instability does occur when the number of subscribed tenants changes. This latter problem is not caused by the replication and auto-scaling concepts of Kubernetes, but by a non-linear resource scaling phenomenon that is inherent when the goal is to meet multiple custom SLOs in a cost-optimal way.

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

Keywords Performance isolation, Multi-tenant SaaS, Container orchestration frameworks

ACM Reference Format:
Eddy Truyen, André Jacobs, Stef Verreydt, Emad Heydari Beni, Bert Lagaisse, Wouter Joosen. 2020. Feasibility of container orchestration for adaptive performance isolation in multi-tenant SaaS applications. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3341105.3374034

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SAC '20, March 30-April 3, 2020, Brno, Czech Republic
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3374034

1 Introduction

Application-level multi-tenancy is an architectural design principle for Software-as-a-Service (SaaS) applications to enable the hosting of tenants by a single application instance in order to reduce development and operational costs for the SaaS provider [21]. Typically, a service-level agreement (SLA), agreed upon between tenant and SaaS provider, consists among others of service-level objectives (SLOs) for performance and availability. While application-level multi-tenancy is the most cost-efficient approach for implementing multi-tenant architectures, it is also the most complex approach to implement performance isolation between tenants with respect to performance SLOs. Algorithms for scheduling requests from different priority queues need to be designed and calibrated for the application at hand. *Adaptive* performance isolation, in particular, is especially hard to design because three different requirements must be supported: (i) QoS differentiation between different SLA classes of tenants, (ii) admission control of aggressive tenants and (iii) dynamically adapting scheduling priorities among tenants to utilize provisioned cloud resources as efficiently as possible [12].

Current solutions for this problem are usually based on request scheduling with a tenant-aware control loop [17, 25]. The problem with these solutions is that they are not stable because the configuration for the same SLO needs to be adapted when there are variations in the total workload context of the SaaS application [25]. These variations include subtle changes in the deployment context of the local node with respect to other co-located resource-intensive applications and slowly evolving changes in the total workload volume when the number of subscribed tenants evolves over time. Thus, this affects the ability of SaaS applications to continuously guarantee a performance SLO to their tenants, even if these tenants respect their part of the SLO, i.e. keep their request rate under a maximum allowed throughput as stipulated in the SLO.

Recently, there has been a strong industry adoption of Docker containers due to their lower memory footprint and sufficient performance isolation [27]. Container orchestration (CO) frameworks such as Kubernetes [14] have also arisen that provide support for automated container deployment, scaling and management.

Kubernetes is a popular open source framework for managing containerized applications in a distributed environment, providing basic mechanisms for deployment, maintenance and scaling of applications [14]. It offers several useful features for resource management. A Pod in Kubernetes is the smallest deployable unit of computing which can be created and managed [5]. Containers belonging to the same application are grouped together in a Pod. The resources used by a Pod can be limited, e.g., by setting its so-called `requests` and `limits`. The `request` is the amount of resources which it is guaranteed to get; the `limit` is the maximal amount of resources it can obtain [7]. Another way of adjusting the resources available to an application is to scale it. Kubernetes offers default horizontal and vertical autoscalers, called the Horizontal

Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA) respectively. However, a well-known disadvantage of the VPA is that it currently requires to reschedule Pods when dynamically adjusting requests or limits [4].

In this paper we study to which extent the above mentioned features of Kubernetes can support adaptive performance isolation so that the complexities of using a request scheduler approach can be reduced and the desired property of SLO stability can be improved. We study this problem for a synthetic SaaS application benchmark that can be configured to be CPU-, memory and disk-intensive.

The remainder of this paper is structured as follows. Section 2 presents the requirements related to application-level multi-tenancy and adaptive performance isolation in particular. Then, Section 3 presents how the resource management concepts of Kubernetes can be used to simplify the architecture of multi-tenant SaaS applications. Thereafter, Section 4 presents the performance evaluation experiments to better understand the feasibility of Kubernetes to support adaptive performance isolation with good support for SLO stability. Section 5 discusses related work. Finally, Section 6 presents our conclusions.

2 Multi-tenancy and adaptive performance isolation

This section presents the main requirements of application-level multi-tenancy and adaptive performance isolation [11, 21, 25].

A SaaS provider offers an application service (e.g. a document archival service) to various organizations (e.g. a bank, a telecom provider), which are called tenants of the SaaS provider. A tenant and SaaS provider operate according to a service level agreement (SLA), which defines a contract with specific service level objectives (SLOs) about performance and availability. A performance SLO is typically expressed as a contract with mutual rights and obligations: if the tenant keeps below a *maximum allowed request rate*, the SaaS provider is able to guarantee a *minimum response latency* expressed in terms of percentiles between the range of 95-99th.

A multi-tenant SaaS application serves multiple tenants by a set of shared application instances that run on a shared cluster of computing and storage nodes, which are provisioned dynamically by a cloud provider. The architectural principle of processing requests of different tenants by the same application instance is called multi-tenancy [21, 25]. Application-level multi-tenancy with support for performance isolation maps then to the following requirements:

1. Customization: SaaS application instances are run-time customizable to the preferences of different tenants. The management of such tenant-specific preferences is automated as much as possible with tenant self-service dashboards.

2. Operational cost-efficiency: The [service level/price cost] ratio determines the competitiveness of the SaaS offering. Application-level multi-tenancy is clearly the most cost-efficient approach because a single application instance can be shared by multiple tenants, which is not the case when running each tenant in a separate virtual machine.

3. Adaptive performance isolation: To implement adaptive performance isolation in the application-level multi-tenancy approach, the following requirements must be satisfied:

3.1: Admission control: Aggressive tenants who violate the terms of the service level agreement with the SaaS provider cannot impact the service level delivered to abiding tenants. Admission control can be implemented by means of a request scheduler based on blacklisting [12].

3.2: QoS differentiation: Performance requirements can be customized to the needs of individual tenants by managing custom SLAs. QoS differentiation can be implemented by means of round robin scheduling of tenant requests with different priorities [12].

3.3 Adaptive resource allocation for improved server consolidation: Some SaaS providers additionally aim to dynamically minimize the number of provisioned nodes in accordance with the actual resource usage of tenants instead of the theoretical required node capacity for guaranteeing all tenants' SLOs.

3 Container-based architecture for SaaS

In this section, we introduce our vision how Kubernetes or container orchestration frameworks can be used to support adaptive performance isolation of multi-tenant web applications with a user-facing, latency-sensitive workload. As web applications are constructed as a multi-tier architecture, we present a tier-based view of the SaaS architecture and describe how the use of Kubernetes impacts each tier. Figure 1 gives an overview of this impact and also presents in green font how existing performance isolation middleware can be simplified. Thereafter we describe how Kubernetes can be used so that SaaS applications can provide support for all three requirements of adaptive performance isolation simultaneously.

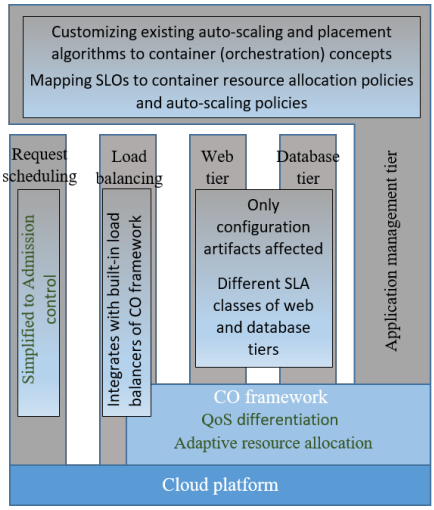

Figure 1. Impact on existing multi-tenant SaaS architecture when using Kubernetes or another container orchestration framework

3.1 Database and web tier

At the database tier, application and configuration data is stored according to a multi-tenant database scheme [9] where data of different tenants is stored in the same database process or even database entity (e.g., table, document, collection).

Configuration data of a tenant includes the desired features that must be activated for that tenant, access control policies about the tenant organization and the tenant's SLA class (golden, silver, bronze). These configurations can be managed by the tenant administrator by means of a self-service tenant dashboard provided by the Application management tier.

The most common and preferred approach in the literature on SLOs for multi-tenant database is to group tenants according to their SLA class in separate database processes [15]. Moreover, in order to implement performance isolation, request throttling is preferred over running database processes in separate virtual machines, that has been considered by the database community as not performing well enough for high-end performance SLOs such as 1000 tps [15]. A relevant question is whether container technology, which is considered as light-weight virtualization, is a possible alternative.

The web tier of every SaaS application is nowadays built on a multi-tenancy middleware layer that transparently adds the tenant ID to each user's session and relies on a dependency injection (D.I.) framework, such as Google's Guice, to support tenant-specific customizations on a per request basis [12, 21, 26]. Thus, each application instance is able to process requests from any tenant and the D.I framework uses the tenant ID to retrieve the configuration data of that tenant from the database.

It is possible deploy the existing code of data and web tiers "as-is" into containers using Kubernetes. Database and web service network endpoints can be exposed via a stable service IP address and a stable DNS names by relying on the Service concept of Kubernetes. The web tier needs to be configured with the correct DNS name(s) for retrieving the database network endpoints. These configurations can be dynamically injected into the web tier container by means the ConfigMap concept.

3.2 Application management tier

The Application Management tier is responsible for fault detection, fault recovery and auto-scaling of web and and database Pods depending on the number of active tenants. To implement this functionality, the application management service depends upon the services of the underlying cloud platform for automated creation or removal of virtual machines, as well as for monitoring various related metrics. To support performance isolation and adaptive resource allocation, the application management tier also includes a distributed monitoring subsystem and central tenant profiler component to categorize tenants in aggressive, abiding and passive tenants [11, 25]. Finally, this tier is responsible for mapping expected and actual workloads to optimal placement of web and database instances across virtual machines so that SLOs are met. This involves determining placement constraints about deployment network topology and VM sizes as well as employing appropriate placement algorithms for meeting SLOs ánd obtaining a good overall server consolidation. Each time when staging a new release of a SaaS application in the production environment, the Application management tier also uses resource-optimization tools such as DBSeer[1] to optimize VM sizes and the number of replicated VMs.

When using Kubernetes, existing application management code needs to be adapted for the APIs of Kubernetes. The placement decision also becomes more complex when using Kubernetes as it additionally involves determining appropriate Pod resource allocation policies, replication levels of Pods and placement constraints about co-location of Pods on VMs. On the positive side, the central profiling component can be simplified because it only needs to detect aggressive tenants as a result of a simplified request scheduling tier (see next section).

[1] https://dbseer.org

3.3 Load balancing tier

The load balancer tier of a traditional SaaS application relies on an off-the shelve load balancer software that can be configured to ensure that tenants's requests are appropriately forwarded to web application instances using a specific load balancing strategy such as as client-IP based load balancing.

When using a container orchestration framework, it is in principle possible to either use or bypass the built-in load balancers of Kubernetes. Services use by default the L4 loadbalancer. To expose the web tier service to end users outside the cluster, services must be exposed via a node port and an external loadbalancer of the underlying cloud provider forwards to the node port; alternatively, a declarative specification for an L7 loadbalancer can be created.

3.4 Request scheduling tier

Krebs et al. [11] presents three architectural approaches to implement admission control of aggressive tenants by means of request scheduling: (i) delaying agressive tenants, (ii) blacklisting agressive tenants, (iii) round robin scheduling with different queues for agressive and abiding tenants. To ensure that the network is spared from the agressive tenants, request schedulers are placed before or within the load balancing tier.

No request scheduling architecture exists to our knowledge that supports all three requirements of adaptive performance isolation simultaneously. It is possible to extend the round-robin architecture with support for QoS differentiation by dispatching abiding tenants in separate queues, one for each SLO class. Alternatively, Walraven et al. [25] extends the round-robin architecture with support for adaptive resource allocation by dividing abiding tenants into queues for normal and passive tenants. However, the configuration of the priority/frequency to schedule requests from the passive and normal queues must be continuously re-tuned when the ratio of passive and normal tenants changes over time.

Container orchestration frameworks have the potential to simplify the overall performance isolation architecture by taking care of the QoS differentiation and adaptive resource allocation requirements, while request schedulers are simplified as they only need to take care of admission control.

3.4.1 QoS differentiation

In line with the approach for multi-tenant databases [15] we propose to group tenants according to their SLA class in separate Services for the web and data base tier. Thus, golden and bronze Services with their own pool of replicated Pods are created for the web tier and golden and bronze database clusters are setup for the database tier. We also define separate user groups for each SLA class. In Kubernetes, this concept of user group is called a Namespace and it is possible to define a total of resource quota across the cluster of nodes as well as default resource allocation policies for the Pods in that Namespace. Golden Pods can also be assigned CPUs that are co-located in the same motherboard socket to ensure stable memory-IO and prevent cache interference.

Note that tenants from the highest SLO class, who wish security isolation and direct monitoring of resource usage, can also be assigned their own Service and Namespace. The advantage of such tenant-specific service is that tenant administrators may also be allowed administrative access to all API objects and resources of that Namespace as well as access to the resource usage metrics

API of Kubernetes for monitoring their Pods. QoS differentiation is then again considered as a placement decision problem, i.e finding the most cost-efficient placement of golden and bronze Pods on a cluster of nodes while still meeting SLOs of tenants. Lang et al. [15] shows that the placement decision problem for databases is a non-linear programming problem (i.e resources do not scale linearly with the numbers of tenants).

3.4.2 Adaptive resource allocation

SLO violation events of response latency can be mapped to appropriate resource usage thresholds of auto-scalers. The default auto-scaler in Kubernetes, named Horizontal Pod Autoscaler (HPA) [6], can be configured to keep average resource usage of a Service's Pods around a certain pivotal point. This pivotal point `p` is specified as the ratio of a Pod's `request` where `p` can also be higher than 1. In any case, the limit of the Pod must be set high enough so that the performance of the Pod is not constrained by this limit when its resource consumption level is at the pivotal point p.

Auto-scaling is the state-of-practice mechanism to support adaptive resource allocation of the web tier. Auto-scaling of databases to meet SLOs in the presence of fluctuating workloads, however, is a challenging problem [28]. Therefore vertical scaling concepts of Kubernetes are also relevant. More specifically, resource over-subscription concepts and resource contention prioritization schemes of Kubernetes can be leveraged to simulate vertical scaling without needing to restart Pods. Over-subscription entails that the `limits` of a Pod can be set higher than its `requests` in order to allow for bursty workloads. In this way, however, a node can be over-subscribed when the sum of the `limits` of the co-located Pods is higher than the node allocatable resources. When the node is actually about to run out of CPU or memory resources, CPU throttling or Pod eviction mechanisms are respectively triggered. For deciding which Pods are to be throttled/evicted first, Kubernetes uses different prioritization schemes for CPU and memory. For CPU, the ratio between the requests of the co-located Pods is used for dividing the available resources [24], whereas for memory a hierarchical priority scheme is used [13].

4 Performance evaluation

This section investigates if the problem of adaptive performance isolation can be comprehensively resolved by resource management concepts of Kubernetes. Firstly, we present the evaluation methodology. Secondly, we investigate if QoS differentiation and admission control (in case containers are reserved for a single tenant) can be supported by Pods that are configured with container allocation policies. Thereafter, we assess the SLO stability offered by the above concepts when increasing/decreasing the number of tenants and applying a linear scaling of containers accordingly. As a reminder, SLO stability refers to the ability to cost-effectively meet SLOs in the presence of global changes of workload without requiring manual recalibration. Finally we investigate if there is a causal relation between the adaptive resource management concepts of Kubernetes for horizontal and vertical scaling of Pods on the other hand and problems with SLO stability on the other hand.

4.1 Evaluation methodology

It is well know that for a workload with a linearly increasing request rate and a fixed set of resources, the average request latency remains below a constant until the request rate crosses a certain critical point of *RR* requests per sec. We name this point the *cut-off point*. After this cut-off point, the latency will increase asymptotically [22]. This common knowledge is formally supported by the universal law of scalability [8], which defines a model of relative capacity. Moreover, the law of scalability can be combined with the law of Little [18] to arrive at a quadratic expression for request latency in terms of the request rate *RR*:

$$Latency(RR) = \frac{1+\alpha(RR-1)+\beta RR(RR-1)}{\lambda}$$

with λ the number of Pod replicas, α the queuing factor (which represent that part of the computation that cannot be executed in parallel by multiple Pods and β the coherency penalty (for example, the time needed for the system to become stable after an increase or decrease of request rate).

The goal of the following experiments is to investigate to which extent the cut-off point of a Service for a set of tenants T remains the same when changing the global workload context in which the Pods of the Service runs. We see three types of changes in global workload context:
1. A Pod runs first alone on a node and then additional Pods are co-located on that node.
2. Primary stressed resources of co-located pods can be the same or different. The former deployment context is named a homogeneous pod deployment whereas the latter is named a heterogeneous Pod deployment in the remainder of this paper.
3. The replication level of Pods changes because the number of tenants changes.

If the cut-off point remains the same across different workload contexts, then the desired property of *SLO stability* can be guaranteed provided the SLO between a tenant and a SaaS application consists of the following clauses:

- for any request rate *RR* <= RR_{cutoff}, the application is able to offer an average response latency L that is equal or higher than the average of $Latency(RR)$. For example, an average latency of 5ms is acceptable for a request-oriented SaaS-application[2].
- the SaaS application commits to a latency target LT that is specified as a percentile o(e.g. 95th percentile of the requests must meet L).
- each of the T tenants commits to a request rate that is lower than RR_{cutoff}/T.

The latency target LT is thus guaranteed as long as the total request rate from all tenants stays under the cut-off point RR_{cutoff}.

In order to experimentally measure the cut-off point in different experiments, we consider a Golden and a Bronze SLO and associated Pod, Namespace and Service configurations in Kubernetes. In all experiments requests are sent in parallel to the Golden and Bronze Service and the request rate *RR* is gradually increased. Requests are sent using a multi-threaded python script. For a request rate *RR* and a number of tenants T, this script will spin up T threads and each thread sends a request every $1/RR$ seconds. For every request rate iteration 1600 requests are sent. The actual response latency of every request will be measured. The cut-off point corresponds with the request rate *RR* after which the target latency LT is violated. Note that the experiments differ in the workload context in which the Golden and Bronze Pods run. In some experiments we have configured the Golden and Bronze Pods with different resource stress parameters to evaluate homogeneous and heterogeneous

Pod deployments. In other experiments, we will gradually change the replication level of the the Golden and Bronze service.

4.1.1 Synthetic SaaS application

We assume our experimental SaaS application does not exhibit performance dependencies towards a shared component so we can measure as accurately as possible the behavior of Kubernetes. Therefore, we have not used a real SaaS application or existing benchmark for this experiment. Instead we have designed a synthetic SaaS application that consists of a single container can be configured as being CPU-bound, memory-bound, or I/O-bound.

The design of this synthetic application is based on the COMITRE approach [21]. The application is written in C++ and offers a REST API (`SaaS_API`) to which users can send request. Every user belongs to a tenant and therefore each request has a `tenantId` field so that the application can retrieve the tenant-specific configuration parameters. Among others, these parameters specify for three resource types (CPU, memory and disk I/O) a natural number that corresponds with the stress intensity. Then for each resource type, three specific stress functions are sequentially invoked as follows. First, a quantity of memory is allocated. Then, a computation is performed and the results of this computation are written to file. Finally allocated memory is released. When the stress parameter for a resource is set to 0, the corresponding stress function will not execute its code. As such it is possible to configure whether the application is CPU, memory- or disk I/O-bound so we can simulate both data-driven workloads as well as user-facing web applications.

The implementation of the different stress functions is widely inspired by the benchmark proposed by Matthews et al. [19]. This work has developed this benchmark to evaluate performance isolation between virtual machines. We haven chosen this benchmark because most other benchmarks such as lmbench[20], sysbench and stress-ng[10] only allow generating the highest possible stress load, while the benchmark by Matthews et al. allows us to configure the stress functions so that different computational complexities can be configured so that different average response latencies of the SaaS application can be simulated.

4.1.2 Experimental setup

The testbed for running all the experiments of this paper is an isolated part of a private OpenStack cloud, version Liberty. The OpenStack cloud consists of a master-slave architecture with two controller machines and droplets, on which VMs can be scheduled. The droplets have Intel(R) Xeon(R) CPU `E5-2650 2.00GHz` processors and 64GB DIMM DDR3 memory with Ubuntu xenial, while the master controller is an Intel(R) Xeon(R) CPU `E5-2430 2.20GHz` machine with Ubuntu xenial. Each droplet has two 10Gbit network interfaces. The droplets have 16 CPU cores of which 2 are reserved for the operation of the Openstack cloud. The storage infrastructure of the OpenStack private cloud is a two-tiered Ceph-based service consisting of replicated SSD storage across the droplets which serves as a cache for 30 SAS drives. Infrequently used data is eventually flushed to the SAS drives. The Kubernetes cluster that has been deployed on the Openstack cloud used version is installed with the kubeadm tool while using Kubernetes version `1.14.9`. The cluster consists of one master node and one or more worker nodes depending on the experiment. Worker nodes are deployed in VMs with 2 CPUs and 4GB RAM that all run in the same physical droplet. That way we can eliminate variations in network delay as an external variable. Moreover, the virtual CPU cores of each VM are exclusively pinned to physical cores that belong to the the same motherboard socket of the droplet. This latter configuration has a surprising impact on SLO stability for CPU (see Section 4.2.3).

4.2 QoS differentiation

4.2.1 Terminology

We assume that a Pod with a larger resource allocation policy can offer a **better** SLO. We define a Pod as comprising:

- an **Application** A that stresses one **Resource type** $R \in \{CPU, IO, MEM\}$. The application A is developed/configured to meet an **acceptable response latency** L on node architecture $ARCH$.
- **Resource allocation policy** $Q = cpu, mem$, with cpu and mem defined as a `<request,limit>` pair. To avoid the Pod being evicted due to resource contention we set $request == limit$.

If we can achieve QoS differentiation, then we can define a Golden Pod G and a Bronze Pod B with $G.Q > B.Q$ and $G^{RR_{cut-off}} > B^{RR_{cut-off}}$.

4.2.2 Configuration of the SaaS application

To achieve an average response latency of L of 5ms for the synthethic SaaS application on our testbed, the following stress parameters had to be set:

Cpu	150
Memory	12000
IO	100

Table 1. Stress parameters for the synthetic SaaS applications so that latency L $5ms$

We have defined a Golden and Bronze Pod for the synthetic SaaS application with $request == limits$ as follows. We have not set any limits on disk usage.

```
BRONZE:
    cpu: 237m
    memory: 875Mi
GOLD:
    cpu: 950m
    memory: 1900Mi
```

4.2.3 Results

Tables 2, 3 and 4 contain the observed cut-off points for the Bronze and Golden service, respectively, in a single Pod placement (where no other Pods are placed on the same node), in a homogeneous Pod placement (where a Golden and a Bronze Pod on the same node stress the same resource), and in a heterogeneous Pod placement (where Pods stress different resources). Not surprisingly, we can observe that QoS differentiation can be achieved as the bronze pods have a cut-off point that is smaller than that of Golden pods.

Finding 1: Comparing the cut-off points of single Pod deployments, homogeneous and heterogeneous Pod deployments shows that SLO stability of golden and bronze Pods is automatically attainable in various deployment scenarios with the exception of three cases: (i) co-locating CPU-intensive golden and bronze Pods on the

same node enables both Pods to sustain a higher cut-off point (due to shared CPU caches, cfr. Section 4.1.2); similarly, (ii) co-locating a memory-intensive golden Pod with a disk-io intensive bronze Pod can enable the former Pod to sustain a higher cut-off point (because the memory paging systems of the underlying VM performs better with an already warmed-up disk-io connection,cfr. Section 4.1.2); (iii) co-locating disk-io intensive golden and bronze Pods causes a drop of the cut-off point for both Pods (due to the lack of performance isolation for disk/io). In the former two scenarios, although very specific to the properties of the experimental testbed, SLOs can be enforced in a more cost-optimal fashion but SLO instability may occur if the specific co-location pattern of that scenario changes. The third scenario is clearly better to be avoided at all.

Tenant gold SLO	Cut-off point	Tenant bronze SLO	Cut-off point
Cpu intensive	230	Cpu intensive	45
Memory intensive	205	Memory intensive	45
IO intensive	135	IO intensive	40

Table 2. Cut-off points for single Pod placements

Tenant gold SLA	Cut-off point	Tenant bronze SLA	Cut-offpunt
Cpu intensive	255	Cpu intensive	50
Memory intensive	205	Memory intensive	40
IO intensive	110	IO intensive	40

Table 3. Cut-off points for homogeneous Pod placements

Tenant gold SLA	Cut-off point	Tenant bronze SLA	Cut-offpunt
Memory intensive	205	Cpu intensive	55
IO intensive	135	Cpu intensive	50
CPU intensive	230	Memory intensive	45
IO intensive	130	Memory intensive	45
CPU intensive	235	IO intensive	45
Memory intensive	220	IO intensive	45

Table 4. Cut-off points for heterogeneous Pod placements

4.3 Admission control

Although admission control is the main responsibility of an external request scheduler (see Section 3.5), container-level admission control is relevant for Pods that are reserved for a separate tenant. Figure 2 shows a request latency graph for homogeneous Pod placements when the bronzen tenant send requests at a rate higher than its observed cut-off point. As such it is considered an agressive tenant. The question is whether the golden tenant is affected by this aggressive tenant.

Finding 2: In case of single-tenant Pods it is possible to complementary enforce admission control between co-located Pods, except for homogeneous deployments of i/o-intensive Pods. This is because Kubernetes currently lacks performance isolation for disk-i/o.

4.4 Stability in the presence of global workload changes

This section presents the results of an experiment where the golden and bronze Service are composed of multiple Pods that are replicated across nodes. We perform an experiment where we start with 1 bronze tenant and 4 golden tenants, measure the cut-off points, and than replace 1 golden tenant with 1 bronze tenant. In the previous experiments we have calibrated Pods for 1 tenant. We assume

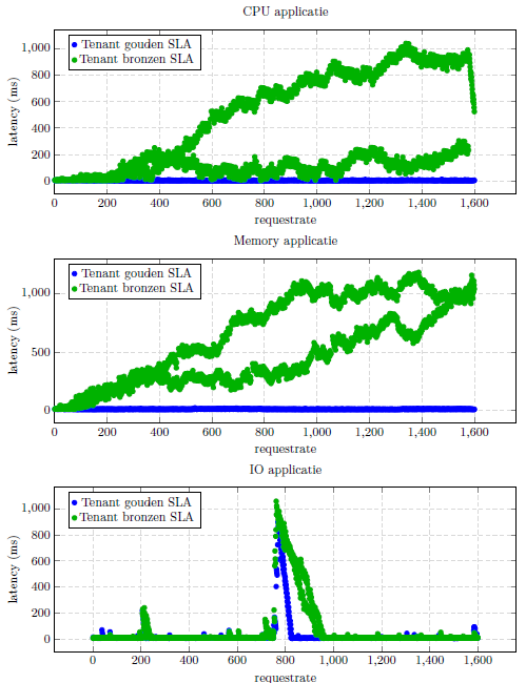

Figure 2. Request latency graph for homogeneous Pod placements when bronzen tenant sends requests at a rate higher than its observed cut-off point

that by linear scaling the Pods we can obtain appropriate resource allocation policies for multiple tenants. As such, the experiment changes the ratio of golden and bronze pods from 4:1 to 1:4. Figure 3 presents the results. It shows that as a golden tenant is replaced by a bronze tenant, the average cut-off point of the bronze tenants increases. Vice versa, when a bronze tenant is replaced by a golden tenant, the average cut-off point of the bronze tenants decreases.

Finding 3: Linear scaling of Pods in response to addition or removal of tenants does not lead to a stable cut-off point for the bronze tenants. As such either too much or too few resources are allocated when relying on linear scaling. This is due to an inherent non-linear scaling phenomenon [15].

4.5 Adaptive resource allocation

The goal of this experiment is to determine whether the above observed stability problems are due to the replication and auto-scaling concepts of Kubernetes. Therefore the goal is to validate whether these problems also appear when auto-scaling a single-tenant version of the SaaS application that is configured with only one SLO and a heavier CPU-intensive parameter of `500`. A linearly increasing workload is again applied. The latency SLO posed for the SaaS application is set to 75 ms.

Three separate tests are run. The first one subjects the application to a linearly increasing workload to determine the cut-off point for one replica. The application's `requests` and `limits` for CPU and memory are set the same as the golden SaaS application (i.e. CPU: `950m`, memory: `1900Mi`). For the second test, the HPA is added to the cluster and the same linearly increasing workload is again applied to the application. 80% of the `request` is selected as the

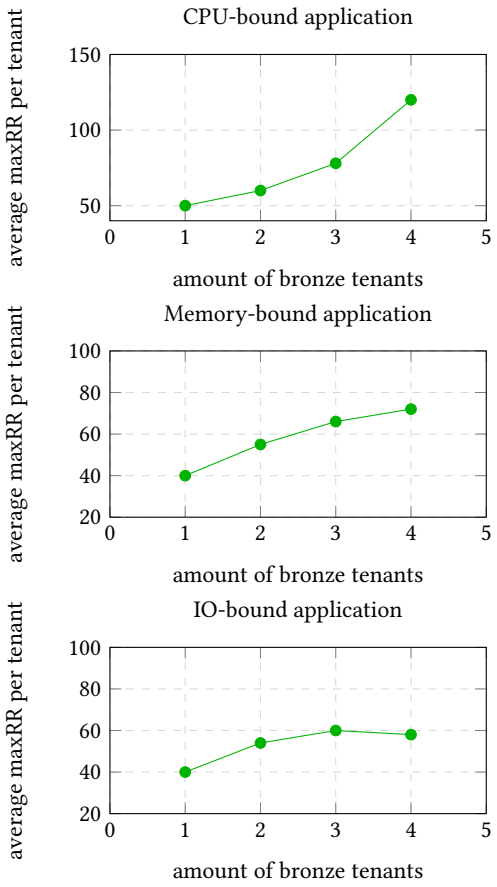

Figure 3. Changes in the cut-off point of the bronze service in the presence of ratio changes

pivotal point p of the HPA. For the third test, a simulation of vertical scaling is applied by setting the `limit` of the application for CPU with `150` millicores higher than its `request` and by co-locating a low-priority pod.

Results. The blue and red graphs in Figure 4 show the 95th percentile of the latencies without and with the HPA respectively. The application is able to process 60 requests per second without violating the SLO. With the HPA, the application is able to process only 110 requests per second without violating the SLO, but at 120 requests per second there is only a slight violation of 77.5 ms. As such, auto-scaling and replication of Pods only causes a small decrease in performance for CPU-intensive, single-tenant applications. The third test tries to rectify the measured SLO violations of the second test by applying the simulation of vertical scaling. The results are shown by the gray graph in Figure 4. Although we only increased the CPU limit with 150 ms, over-provisioning of resources can be observed as the cut-off points arises at 150 requests per second, which is substantially larger than the double of the cut-off-point of a single replica. As such, finding the optimal resource allocation policies of Pods in order to perfectly meet SLOs in a cost-optimal way boils again to a resource configuration tuning problem.

Finding 4: The default horizontal autoscaler approximates the desired SLO stability property for a CPU-intensive application that

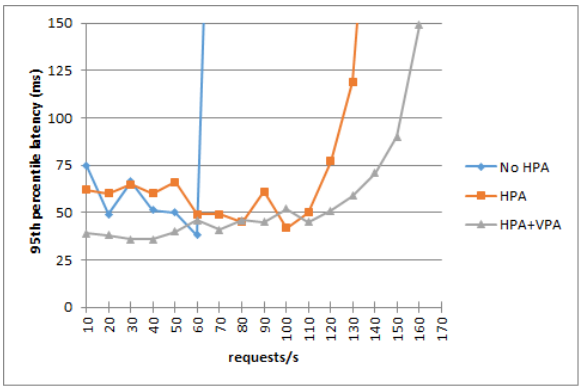

Figure 4. 95th percentile latencies of a single-tenant version of the SaaS application when exposed to a linearly increasing request rate.

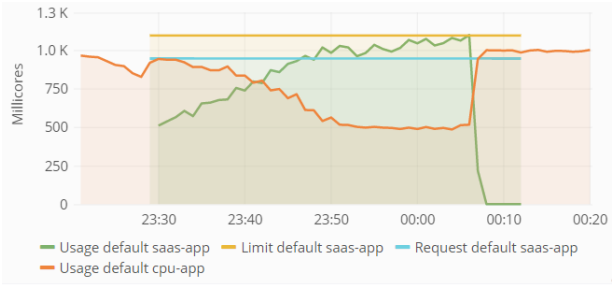

Figure 5. CPU usage during the third test. A new replica is added to a second worker node when the scaling threshold is breached and resources are re-claimed from the low priority Pod.

is exposed to a linearly increasing workload. As such, the main cause of the SLO stability problems of Finding 3 lies with the non-linear resource scaling phenomenon that appears when the goal is to meet multiple custom SLOs in a cost-optimal [15].

Figure 5 shows that the simulation of vertical scaling by co-locating a low-priority pod comes with the benefit of a higher resource utilization. The low priority pod is able to use the excess of resources on the node during low workloads. As the workload rises, the low priority pod is given access to less CPU cycles.

Finding 5: For single-tenant applications, the resource management concepts of Kubernetes – in particular concepts for over-subscription of resources and throttling of lower-priority Pods – allow to simulate vertical scaling of CPU resources of high-priority applications and still provide a good overall resource utilization.

5 Related work

Existing research [3, 27] has focused most attention on comparing the performance of a single container against a single virtual machine, both running directly in Linux on top of a bare-metal machine. In terms of performance isolation, containers do not yet provide complete isolation of resources as virtual machines do [27].

Literature on design and evaluation of container orchestration frameworks originates mostly from Google [1, 23]. The Borg system, and its predecessor Omega, are used for running all Google services. For example, the GCE IaaS uses Borg to schedule VMs

inside containers. Verma et al. [23] shows that the Borg system, a predecessor of Kubernetes, supports improved resource utilization in terms of number of machines needed for fitting a certain workload on.

Verma et al.[23] also report that the implementation of the cgroups mechanism requires substantial tuning of the standard Linux CPU scheduler in order to achieve both high resource utilization and low latency. Leverich et al. [16] also propose an improved Linux CPU scheduler. An evaluation of this scheduler shows that the 95th-percentile latency is negatively affected by co-located workloads but this decrease does not devolve to asymptotic delays.

6 Conclusion

This paper has investigated how commonly supported resource management concepts of container orchestration frameworks can simplify performance isolation middleware for multi-tenant SaaS applications and improve it with the desired property of SLO stability.

Experiments with co-locating CPU, memory and disk-io intensive workloads, which are based on a synthetic multi-tenant SaaS application benchmark, show that Kubernetes can support QoS differentiation, adaptive resource allocation and admission control for single-tenant Pods with certain guarantees for SLO stability. Findings 1 and 2 show that in order to prevent SLO instability and weak performance isolation of aggressive single-tenant Pods, co-location of disk-io intensive Pods must be avoided at all costs. Finding 1 further shows that SLO stability can be attained across a wide range of single Pod deployments, homogeneous and heterogeneous Pod deployments with two exceptions that are mainly due to the experimental testbed. Secondly, Finding 3 implies that linear scaling of Pods, when tenants subscribe or describe, leads to SLO instability. Thirdly, Finding 4 demonstrates this SLO instability is not dominantly caused by the horizontal auto-scaling concepts of Kubernetes, but by a non-linear resource scaling phenomenon. Fourthly, Finding 5 shows that the simulation of vertical scaling is a simple yet effective technique for supporting adaptive resource allocation.

We conclude that request scheduling architectures can be simplified to admission control, whereas container orchestration frameworks can take care of QoS differentiation and adaptive resource allocation. However, when tenant demand grows and the required set of replicated Pods must be scaled up, non-linear scaling functions are clearly a better fit for achieving SLO stability. How to integrate such non-linear scaling functions in state-of-the-art container orchestration frameworks like Kubernetes is an open research challenge.

References

[1] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (2016), 50–57.

[2] Jeff Dean. 2017. Latency Numbers Every Programmer Should Know. https://gist.github.com/jboner/2841832, last checked on 2017-06-08.

[3] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An updated performance comparison of virtual machines and Linux containers. *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2015). http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7095802

[4] Cloud Native Computing Foundation. [n.d.]. In-place Update of Pod Resources. https://github.com/kubernetes/enhancements/blob/29a22b61241b35bb280de83edc0aee40d1bd87bf/keps/sig-autoscaling/20181106-in-place-update-of-pod-resources.md. Accessed: 2019-09-09.

[5] Cloud Native Computing Foundation. [n.d.]. What is a Pod? https://kubernetes.io/docs/concepts/workloads/pods/pod/. Accessed: 2019-03-08.

[6] The Linux Foundation. [n.d.]. Horizontal Pod Autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed: 2019-07-29.

[7] Google.com. [n.d.]. Kubernetes best practices: Resource requests and limits. https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-resource-requests-and-limits. Accessed: 2019-03-08.

[8] Neil J. Gunther. 2008. A general theory of computational scalability based on rational functions. *arXiv preprint arXiv:0808.1431* (2008).

[9] Dean Jacobs and Stefan Aulbach. 2007. Ruminations on Multi-Tenant Databases. *BTW Proceedings* 103 (2007).

[10] Alexey Kopytov. 2017. sysbench github. https://github.com/akopytov/sysbench, last checked on 2017-04-11.

[11] Rouven Krebs, Christof Momm, and Samuel Kounev. 2012. Architectural Concerns in Multi-tenant SaaS Applications. In *The 2nd International Conference on Cloud Computing and Services Science (CLOSER 2012)*. ScitePress, 426–431.

[12] Rouven Krebs, Christof Momm, and Samuel Kounev. 2014. Metrics and techniques for quantifying performance isolation in cloud environments. *Science of Computer Programming* 90 (2014), 116–134. https://doi.org/10.1016/j.scico.2013.08.003

[13] Kubernetes. 2018. Evicting end-user pods. URL: https://github.com/kubernetes/website/blob/release-1.11/content/en/docs/tasks/administer-cluster/out-of-resource.md#evicting-end-user-pods, accessed 2018-09-18.

[14] Kubernetes. 2018. Production-Grade Container Orchestration. URL: https://kubernetes.io/, accessed 2018-01-23.

[15] Willis Lang, Srinath Shankar, Jignesh M. Patel, and Ajay Kalhan. 2014. Towards multi-tenant performance SLOs. *IEEE Transactions on Knowledge and Data Engineering* 26, 6 (2014), 1447–1463.

[16] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM.

[17] Hailue Lin, Kai Sun, Shuan Zhao, and Yanbo Han. 2009. Feedback-control-based performance regulation for multi-tenant applications. In *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*. https://doi.org/10.1109/ICPADS.2009.22

[18] John DC Little and Stephen C Graves. 2008. Little's law. In *Building intuition*. Springer, 81–100.

[19] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. 2007. Quantifying the Performance Isolation Properties of Virtualization Systems. *Proceedings of the 2007 Workshop on Experimental Computer Science (ExpCS 2007)* (2007), 6. https://doi.org/10.1145/1281700.1281706

[20] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference (ATEC '96)*. USENIX Association, Berkeley, CA, USA, 23–23. http://dl.acm.org/citation.cfm?id=1268299.1268322

[21] Laud Charles Ochei, Julian M. Bass, and Andrei Petrovski. 2015. Evaluating Degrees of Multitenancy Isolation: A Case Study of Cloud-Hosted GSD Tools. *Proceedings - 2015 International Conference on Cloud and Autonomic Computing, ICCAC 2015* (2015).

[22] M.M. Teixeira, M.J. Santana, and R.H.C. Santana. 2004. Using adaptive priority scheduling for service differentiation QoS-aware Web servers. *Performance, Computing, and Communications, 2004 IEEE International Conference on* (2004), 279–285. http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=1395003

[23] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. *Eurosys* (2015).

[24] Ananya Kumar Vishnu Kannan. 2019. How Pods with resource requests are scheduled. https://github.com/kubernetes/website/blob/release-1.14/content/en/docs/concepts/configuration/manage-compute-resources-container.md#how-pods-with-resource-limits-are-run. Accessed: 2019-06-13.

[25] Stefan Walraven, Tanguy Monheim, Eddy Truyen, and Wouter Joosen. 2012. Towards performance isolation in multi-tenant SaaS applications. *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing - MW4NG '12* (2012), 1–6. https://doi.org/10.1145/2405178.2405184

[26] Stefan Walraven, Eddy Truyen, and Wouter Joosen. 2011. A Middleware Layer for Flexible and Cost-Efficient Multi-tenant Applications. *Middleware 2011* 7049, i (2011), 370–389.

[27] Miguel Gomes Xavier, Marcelo Veiga Neves, and Cesar Augusto Fonticielha De Rose. 2014. A Performance Comparison of Container-Based Virtualization Systems for MapReduce Clusters. *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2014), 299–306. https://doi.org/10.1109/PDP.2014.78

[28] L. Zhao, S. Sakr, and A. Liu. 2015. A Framework for Consumer-Centric SLA Management of Cloud-Hosted Databases. *IEEE Transactions on Services Computing* 8, 4 (July 2015).