KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Multi-tenant performance SLOs for container orchestrated batch workloads

Matthijs Kaminski

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Gedistribueerde systemen

Promotoren:

Prof. dr. ir. Wouter Joosen
Dr. Eddy Truyen

Assessoren:

Prof. dr. Adalberto Simeone
Emad Heydari Beni

Begeleider:

Emad Heydari Beni

Academiejaar 2017 – 2018

Acknowledgements

This thesis has been quite a long journey. Luckily, I had a lot of wonderful people helping and supporting me throughout the ups and downs. With this acknowledgement, I would like to thank each and everyone of them.

I would like to thank my promotor, Eddy Truyen for the opportunity to work on such an interesting research topic. Your guidance and feedback led me to create this end result of which I am proud.

Many thanks to my promotor, Wouter Joosen and my coach, Emad Heydari Beni for their feedback and insights during our meetings.

Last but definitely not least, I would like to thank my parents, Sien and my friends for supporting me during these past years.

Matthijs Kaminski

Contents

Abstract

The cloud computing paradigm has become more popular over the past recent years. Cloud providers offer various on-demand products ranging from hardware infrastructure to software services. Providers of the latter face the constant challenge to maximize the use of their rented infrastructure in order to offer their software service at the most competitive price. Often a multi-tenant architecture is employed in order to maximize resource sharing between tenants. This requires techniques to differentiate the service between different classes of tenants. The quality of the service provided is specified in an Service Level Agreement (SLA) between tenant and provider. Recent work states that the resource management concepts of container technology such as Docker and container orchestration platforms such as Kubernetes can be used to achieve multi-tenancy with quality of service differentiation while offering fine-grained resource allocation. In order to use these container orchestration concepts, SaaS providers need to translate Service Level Objectives (SLOs), which are part of the SLA, to low-level resource allocations. This difficult task typically requires extensive domain expertise.

Existing state-of-the-art approaches to this problem rely on a model of the application and therefore again requires extensive domain expertise in order to create an accurate model of the application. In opposition to this approach, we propose a generic approach that does not require any model of the application or another type of domain expertise. Instead, we determine mappings between SLOs and resource allocations by adapting a performance tuning technique that relies on continuous experimentation of the application in the production environment. We believe such continuous experimentation is feasible in contemporary DevOps environments where a new version of the application is tested in the production environment before exposing it to clients.

In this master thesis this approach is implemented as part of an automated tool, k8-resource-optimizer, capable of translating SLOs for a given application and workload to Kubernetes resource concepts. The tool allows the Kubernetes application to be declaratively specified as a typical Helm chart and allows declarative configuration of SLOs, resource search space parameters and workload profiles. The latter are intended to stress the application with the maximal allowed workload in order to determine whether the SLO is violated or not.

Experimental evaluation of k8-resource-optimizer with a simple batch processing application has shown that this approach is practically able to produce an optimal or near-optimal resource allocation in various deployment scenarios with one or more SLOs for one or more tenant organizations. Therefore, k8-resource-optimizer shows potential as a useful DevOps tool.

Nederlandstalige samenvatting

Het cloud computing paradigma is gedurende de laatste jaren een belangrijk onderdeel geworden van IT-infrastructuren. Het is een overkoepelende term die gebruikt wordt voor zowel de aangeboden hardware in datacenters, alsook voor de verschillende beschikbare diensten die gebruik maken van deze hardware. Computerkracht wordt aangeboden als een nutsvoorziening via het netwerk. De aangeboden modellen variëren van laag niveau *Infrastructure as a Service* (IaaS) tot hoog niveau *Software as a Service* (SaaS). Aanbieders van SaaS applicaties maken zelf meestal gebruik van een gehuurde IaaS. Deze op aanvraag beschikbare infrastructuur maakt de nood aan een dure privè infrastructuur van SaaS aanbieders overbodig.

Een *Service Level Agreement* (SLA) is een formele overeenkomst tussen een klant en een dienstleverancier. Dit contract bevat een beschrijving van de dienst, aangeboden garanties en gebruiksvoorwaarden. Een SLA bevat typisch ook enkele *Service Level Objectives* (SLOs). Een SLO beschrijft een meetbare eigenschap van de aangeboden dienst zoals beschikbaarheid, wachttijd of doorvoer.

SaaS aanbieders streven ernaar om hun dienst aan te bieden met een hoge kwaliteit voor een competitieve prijs. Hiervoor hanteren ze een *multi-tenant* architectuur waarbij er gestreefd wordt naar een maximale benutting van de capaciteit door middel van het delen van hardware of software tussen klanten. Deze aanpak vereist echter wel technieken die het mogelijk maken om verschillende kwaliteitsniveaus van de dienst aan te kunnen bieden. Hiernaar wordt verwezen als *Quality of Service* (QoS) differentiatie.

Recent is er een enorme groei aan de populariteit van containertechnologieën zoals Docker. Deze virtualisatietechnologie biedt een gelijkaardige performantie als traditionele virtuele machines op een meer kost-efficiënte manier. Containers laten bovendien een meer flexibele en verfijnde allocatie van *compute resources* zoals CPU en geheugen toe. Voor het beheren van een groot aantal containers op een cluster van machines zijn er containerorchestratie systemen, zoals Kubernetes ontwikkeld. Recent academisch werk suggereert het gebruik van*resource management* concepten van Kubernetes voor het bereiken van QoS-differentiatie in een multi-tenant architectuur bovenop container orchestratie. SaaS aanbieders kunnen gebruikmakend van deze concepten de capaciteit van hun gehuurde infrastructuur beperken tot de capaciteit nodig voor het bereiken van afgesproken SLOs. Om dit mogelijk te maken, moet

er een kost-efficiënte vertaling van SLOs naar *resource* allocatie strategie gemaakt worden voor de applicatie van de SaaS aanbieder. Voor deze taak wordt de term *SLA-decomposition* gebruikt. Het uitvoeren van deze taak blijkt een moeilijke opdracht voor de meeste ontwikkelaars en vereist veel ervaring en domeinkennis.

Bestaande oplossingen voor dit probleem gebruiken veelal een theoretisch model van de applicatie. Het opstellen van dergelijke modellen vereist echter ook enige expertise. In deze thesis stellen we een methode voor die geen applicatie modellen of domeinkennis vereist. In deze methode wordt de vertaling tussen SLOs en resource allocaties gerealiseerd door een aangepaste *performance tuning* techniek die de applicatie onderwerpt aan continue experimentatie in de productie omgeving.

In deze thesis wordt deze methode geïmplementeerd als een geautomatiseerde tool genaamd k8-resource-optimizer. Deze tool is in staat om SLOs van een gegeven applicatie en gegeven werkbezetting te vertalen naar Kubernetes *resource* concepten. Het ontwerp van de tool zorgt voor een gemakkelijke integratie met Kubernetes en Helm een populaire *package manager* voor Kubernetes.

Uit experimentele evaluatie van k8-resource-optimizer met een simpele *batch processing* applicatie is gebleken dat deze tool is staat is om een optimale of bijna optimale *resource* allocatie te produceren in scenario's met een of meer SLOs van verschillende klantorganisaties. Hoewel er verdere experimentatie nodig is, lijkt k8-resource-optimizer een handige DevOps applicatie die gebruikt kan worden wanneer een nieuwe versie van een applicatie getest wordt in de productieomgeving, voordat deze toegankelijk is voor klanten.

List of Figures and Tables

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

In recent years, the cloud computing paradigm has become a key component of IT infrastructures. It is an umbrella term as it refers to both the hardware available in data centers and the various provided services utilizing that hardware [3]. The paradigm enables on-demand network access to a rapidly available pool of computing resources [45]. Computing is offered as a utility. The models offered by vendors range from the lower-level infrastructure as a service (IaaS) to higher level software as a service (SaaS). It is evident that SaaS providers themselves often make use of IaaS. The seemingly endless amount of on-demand resources offers SaaS providers the opportunity to minimize or completely abandon their on-premise infrastructure, thus minimizing capital expenses while dynamically managing operational expenses (i.e., CapEx to OpEx [3]).

In a business context, a Service Level Agreement (SLA) between a customer and a service provider is a formal agreement, encapsulating, but not limited to, service behavior guarantees, customer usage terms and possible penalties for violating these guarantees/terms. An SLA typically also contains several Service Level Objectives (SLOs). An SLO represents a specific measurable characteristic of the service, typically Quality of Service (QoS) metrics such as availability, latency or throughput.

SaaS providers face the constant challenge of offering their products at the best service level for the most competitive price. To further minimize their costs, SaaS providers employ the architectural design principle of multi-tenancy. Multi-tenancy attempts maximizing the sharing of resources among multiple customers organizations, referred to as tenants.

The sharing of resources between tenants, however, poses a few challenges: (i) dealing with aggressive tenants (admission control), (ii) offering quality of service differentiation (QoS-differentiation) for different SLA-classes and (iii) achieving efficient resource utilization. The research community has proposed several strategies for

achieving multi-tenancy.

In [65] these challenges are tackled at the application-level by introducing a middleware layer. The middleware allows prioritization of tenant tasks within a submission queue. This prioritization is achieved by scheduling tasks based on information retrieved by control loops. Assignment of both the task of dealing with aggressive tenants as the task of achieving QoS-differentiation to the scheduling algorithm increases its complexity. As a result, these solutions appear unstable under changing workloads and thus often require reconfiguration of algorithm parameters.

More recent work [63] proposes container technology and container orchestration as a possible solution for multi-tenancy. Containers are a more lightweight version of virtualization compared to traditional virtual machines (VMs). By sharing the OS kernel, containers can offer similar performance as VMs with less overhead. Container orchestration platforms enable the operation and organization of containers on a cluster of machines. [63] suggest the use of resource allocation concepts (e.g., CPU and memory) of Kubernetes [39], a container orchestration platform, to achieve QoS-differentiation between SLA-classes. In addition, two strategies for sharing resources between multiple tenants of possible different SLA-classes are given.

1.2 Problem statement

The goal of the thesis is to build upon the proposals of [63], using container orchestration resource allocation policies to achieve multi-tenancy for SaaS providers. The assignment of compute resources to achieve QoS-differentiation introduces the problem of SLA-decomposition [9]. The problem can be formulated as follows. Given an application, a workload and set of SLOs find a resource allocation that satisfies the SLOs. In the context of SaaS providers, the resource allocation is preferable as cost-efficient as possible. A user study conducted by [31] states 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. This indicates that resource allocation requires extensive expertise and knowledge of the application. Therefore, SLA-decomposition is proven to be an even more difficult task. Moreover, when new tenants are subscribed, simple linear resource scaling will either over-provision or under-provision resources as has been shown in previous work [29, 40].

Existing state-of-the-art has applied non-linear programming to the problem of multi-tenant SLOs for databases [40]. Here, an optimization problem is solved where the constraints of the objective function are non-linear. However, the quality of the solution succeeds or breaks depending on how well the model of the SaaS application corresponds with real application data.

1.3 Approach

The ultimate goal of the thesis is the design and implementation of a tool capable of automatically deriving a cost-efficient SLA-decomposition for a SaaS application deployed via a container orchestration platform, Kubernetes in particular. In other words, given an application, a workload and SLOs find a cost-efficient resource allocation that satisfies the SLOs. Furthermore, the tool should support the deployment strategies for multi-tenancy suggested by [63]. The correctness and feasibility of the tool should be evaluated in the context of a multi-tenant batch processing application.

Achieving this goal requires the completion of several smaller goals:

- **Selection of a technique to perform a SLA-decomposition.** The creation of an automated tool requires an algorithmic solution to the problem of SLA-decomposition. The purpose of the tool is to simplify the decomposition task for system administrators. The proposed solution should not require extensive knowledge of the application nor of the decomposition technique employed by the automated tool.

- **Design and implementation of proof of concept automated SLA-decomposition tool.** The selected SLA-decomposition technique should be adapted for the Kubernetes container orchestration environment. The adaption has to be implemented in a proof of concept automated tool. The design of the tool should also allow for compatibility with different Kubernetes applications.

- **Evaluation of the proof of concept in a controlled environment.** The correctness and feasibility of the proposed solution have to be evaluated in a controlled environment. A controlled environment allows for easier validation. For that reason, a simple and understandable batch processing application should be designed and implemented.

1.4 Contribution of thesis

The thesis provides insights into several SLO-decomposition capable techniques proposed in research. From these techniques, the performance optimization algorithm of BestConfig [69], a search-based configuration auto tuner, was selected for its understandability and adaptivity. The algorithm was adapted for the container orchestration setting and implemented in an automated tool named k8-resource-optimizer. k8-resource-optimizer searches for an optimal resource allocation based on real performance evaluation, thus eliminating uncertainties present in model-based approaches. To our knowledge, these techniques have not yet been applied to the problem of multi-tenant SLOs. Building on top of well-known Kubernetes tools, k8-resource-optimizer is able to perform a resource efficient SLO-decomposition for a multi-tenant Kubernetes application starting from an input configuration describing an application template, a deployment strategy and the required SLOs.

1.5 Overview of text

The remainder of the thesis is structured as follows. Chapter 2 contains the necessary background information for the thesis. It presents an overview of cloud computing, multi-tenancy, container-technology and container orchestration. Chapter 3 discusses relevant related work and gives insights into the several techniques that can be used for SLA-decomposition. The chapter concludes with the selection of the most appropriate technique used in the proposed solution. Chapter 4 presents the application built to evaluate the solution presented by the thesis. Chapter 5 discusses the architecture and implementation of the proposed solution and associated prototype, called k8-resource-optimizer. Next, in Chapter 6 the accuracy and performance of the k8-resource-optimizer tool are evaluated in various scenarios. Finally, our conclusion and future work are presented in Chapter 7.

Chapter 2

Background

This chapter discusses relevant background material for the thesis. The concepts, explored in this chapter, further clarify the relevance of the thesis and serve as building blocks for the remaining chapters. Starting from the adoption of the cloud computing paradigm, the chapter elaborates on how concepts such as multi-tenancy and virtualization via containerization push the limits of efficient resource utilization. Next, an in-depth overview of the open-source container orchestration platform Kubernetes is provided. Lastly, insights are provided into the universal scalability law and the applicability of Little's law for performance testing.

2.1 Cloud computing

Companies try to minimize the total cost of ownership by moving to cloud computing or utility computing. The cloud computing paradigm enables on-demand access to a shared pool of configurable compute resources (software applications, system software and hardware infrastructure) [66]. A service provider can provision the required resources from the cloud with minimal effort. Within this paradigm, services are offered in real time over the internet in three different models [44, 52].

- *Infrastructure as a Service (IaaS)* in which access is offered to virtual machines, network, data storage and other fundamental computing resources. The customer is able to deploy arbitrary software using these resources.

  Examples: Digital Ocean, Microsoft Azure and Amazon Elastic Compute Cloud (EC2).

- *Platform as a service (PaaS)* offers customers a platform enabling easy and efficient deployment of applications (at scale) while abstracting the complexity of managing the underlying infrastructure. PaaS allows customers to solely focus on the development of their application.

  Examples: Google app engine, Microsoft Azure and Heroku.

- *Software as a service (SaaS)* allows customers to make use of a specific application developed by a SaaS provider. Compared to the traditional use of software, the management and deployment is the task of the SaaS provider. In this strategy, common resources and a single instance of both the application and underlying database are used to support multiple customers simultaneously.

  Examples: Google apps and Salesforce.

2.2 Multi-tenancy

The software as a service (SaaS) model offers applications to customers as an on-demand service. Providers of these applications try to leverage economies of scale by employing a multi-tenancy architectural design principle. The goal of multi-tenancy is to minimize the total cost of ownership for the provider by maximizing the sharing of resources among multiple customers organizations, referred to as tenants. [65]

2.2.1 Service Level Agreements

By moving their core business functions to an entrusted cloud provider, cloud customers give up control of the underlying compute resources. It is vital for these tenants to obtain certain guarantees on the service delivered by the provider. A Service Level Agreement (SLA) is a formal contract between the SaaS provider and the tenant specifying both properties of the provided service and the expected behavior of the tenant. An SLA typically also contains a set of Service Level Objectives (SLOs). An SLO is a measurable characteristic of the service. An SLO is typically related to performance constraints (latency, throughput and deadlines) or availability (uptime percentage) of the service. Within the specification of an SLA a trade-off must often be made between expressiveness and usability. The SLA must cover all the expectations of a customer while remaining simple to weight, verify, evaluated and enforce [13]. Some typical examples of SLOs are shown below:

- The application has a monthly uptime percentage of 99.95%.
- If the arrival rate of the tenant workload $<$ X requests/s then a throughput T is guaranteed.

2.2.2 Challenges

While the goal of multi-tenancy is promising, sharing a cluster of dynamically provisioned nodes between multiple tenants imposes a number of challenges and requirements [63].

- Performance isolation: the activities of one tenant should not be able to influence the service level delivered to other tenants. This requirement should be achieved during normal system load and when an aggressive tenant violates the terms of the SLA.

- QoS differentiation: the performance guarantees specified by an SLA can be individually customized for each tenant. A SaaS provider should be able to offer different subscriptions.

- Flexible resource allocation for improved server consolidation: a SaaS provider employs a multi-tenant architecture to achieve a lower operational cost. This is partially achieved by planning the required node capacity based on the actual resource usage of tenants instead of the theoretical required capacity. This can be further aided by the use of request schedulers allowing to distinguish between normal, passive and aggressive tenants.

2.2.3 Strategies

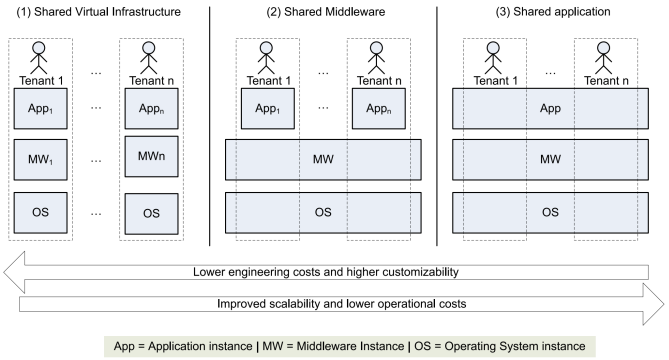
To achieve multi-tenancy different strategies, each offering different trade-offs concerning operational costs and upfront application engineering costs, can be employed by the SaaS-provider [67]. The strategies are illustrated in Figure 2.1.

- Multi-tenancy can be achieved at the level of the *operating system*. In this strategy, a virtualization technology can be used to partition compute resources among multiple virtual machines. Each tenant is assigned an application instance running on a dedicated virtual machine. This approach offers both a higher level of performance isolation and lower upfront engineering costs but suffers from an inefficient utilization of resources.

- In *middleware-level multi-tenancy* a middleware platform is used to enable sharing compute resources between multiple tenants at the level of the operating system. An application instance is deployed on top of the middleware platform for each tenant. By not replicating the operating system for each tenant a higher level of cost efficiency can be achieved but an increased complexity in managing resources and performance isolation is introduced. By tackling these problems at the level of the middleware a part of the engineering complexity is shifted to this level.

- The most efficient resource utilization can be achieved by sharing application instances between multiple tenants. In *Application-level* multi-tenancy achieving performance isolation is done by the application itself thereby increasing the engineering complexity and costs.

Figure 2.1: Different strategies to achieve multi-tenancy [67].



2.3 Containerization

Cloud providers rely on virtualization technology for the achievement of large-scale resource scaling. For more than a decade, virtual machines (VMs) have been the backbone of a provider's infrastructure offering hardware independence, availability, isolation and security [68]. More recently, containers a more lightweight virtualization technology have made advances in their multi-tenant capabilities and have seen an increase in adoption by providers [50]. Both are discussed in this section.

2.3.1 Virtualization technology

In cloud environments, virtualization technology is used for flexible and dynamic allocation of physical resources to virtualized applications and to achieve multi-tenancy by sharing a physical server among multiple applications. In data centers virtualization is commonly used at the *hardware level* and *operating system level* to deploy and manage virtual machines and applications at scale [55].

Hardware level virtualization uses a hypervisor on a server to create virtual machines. Each virtual machine provides an abstraction of a physical machine and runs an independent operating system with applications. The hypervisor is responsible for resource allocation and performance isolation.

Operating system level virtualization allows resources to be shared at the level of the OS. Virtual machines running at the OS level are referred to as containers. Isolation, abstraction and resource allocation of containers is performed by the OS kernel of the host OS. By sharing the OS kernel among multiple containers, containers are regarded as a more lightweight virtualization technology. Containers only contain the application and its dependencies.

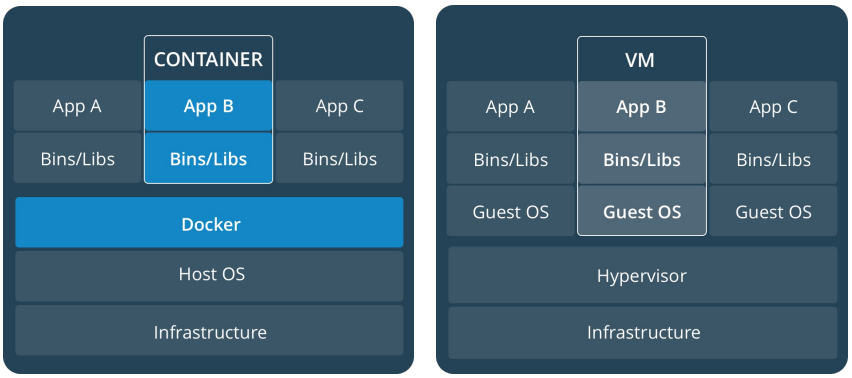
Linux containers (LXC) [7] employ different mechanisms of the Linux kernel to achieve resource isolation, namely control groups and namespaces.

- **Cgroups** [51] allow for fine-grained control of the allocation of system resources (CPU time, system memory, network bandwidth) among processes and process groups. For example, it is possible to limit or prioritize memory, CPU or I/O usage of different containers.
- **Namespaces** [49] allow for isolation of kernel resources among processes. A namespace makes a resource appear to be private and isolated for the container. The Linux kernel provides the following namespaces: process ids, inter-process communication (IPC) mechanisms, network stack and mount points.

Linux containers (LXC) offers a lightweight implementation which performs at native speed and provides good isolation. However, while sharing a kernel between containers minimizes overhead, there are limitations in terms of the security environment. [16]

By offering virtualization at different levels, containers and VMs offer different trade-offs concerning performance isolation and performance. Research [55] concludes that while containers offer closer to bare metal performance compared to VM, they offer worse performance isolation in multi-tenant environments. In addition, containers offer soft resource limits compared to the hard resource limits of virtual machines which allow for better server consolidation in over-commitment scenarios.

Figure 2.2: Container vs virtual machine. [15]



Docker

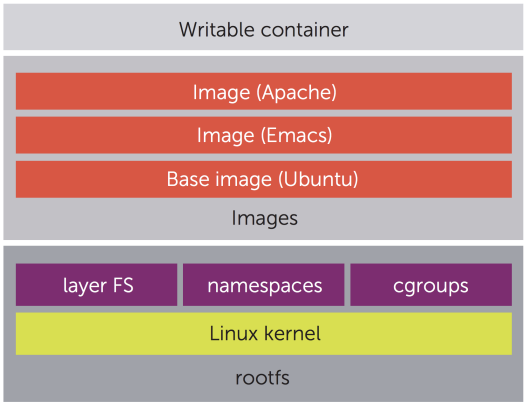
Recently, thanks to Docker [14], containers have gained popularity and have been adopted into the software development process. However, Docker is not a new container technology, at its core it employs the kernel-level mechanisms of Linux containers (LXC), for which it defines a unified API [46]. In addition, the open-source Docker project offers a commandline-interface and daemon that offers easy packaging of applications in containers and the deployment of these containers.

To achieve this Docker introduces the concept of images. A container is represented by a lightweight image. A container image is a lightweight, executable package

containing an application and its dependencies (runtime, libraries, environment variables, and config files). Virtual machines (VMs) can be seen as full, monolithic images. In particular, Docker image consists of file-system layers stacked upon each other, as illustrated in Figure 2.3. Only the top layer, the container itself, is writable, therefore it is state-full and executable. A container is thus composed out of layers of individual images built on top of a base image, allowing for easy extensibility. [50]

A Docker image is specified by and build from a DockerFile. Images can be made easily accessible through Docker registries such as Dockerhub.
These images allow for easy and fast deployment of Docker containers across different operating systems and cloud provider stacks. Resulting in a game-changing technology for DevOps, system administrators and developers.

Figure 2.3: Architecture of a container image. Images can be stacked upon each other using the cgroups and namespace extensions of a Linux kernel. The top container image is writable. [50]



2.4 Container Orchestration

Container technologies such as Docker offer advanced features to use containers in production. These solutions often offer deployment of applications limited to single machines. However, applications offered by SaaS providers may include multiple different containers working together running on a cluster of nodes. Container orchestration (CO) systems allow deployment and management of containers at scale. Popular orchestration engines are Kubernetes, Docker Swarm and Mesos Marathon.

2.4.1 Key capabilities of a container orchestration platform

The task of a container orchestration (CO) platform is not limited to the initial deployment but the entire life-cycle of multiple containers. The goal of CO is to simplify cluster management while ensuring fault tolerance, availability, scalability and reliability. Allowing users to benefit from the complete potential of containers.

In order to meet this goal, CO platform must at least offer the following key capabilities. [33]

Cluster state management and scheduling A cluster can be composed out of multiple virtualized or physical instances. Running containers on top of these instances and recover from failures requires a stable cluster. State management encapsulates various tasks such as flexible scheduling, re-partitioning of resources and data and propagating of dependent system changes. [33]

High availability and fault tolerance A container orchestration platform must ensure high availability and fault tolerance in order to be useful for application developers. Most platforms therefor employ design principles of reliability engineering e.g., single point of failure elimination, failure detection or load balancing. [33]

Security Containers executing on top of the platform essentially form untrusted entities, with potentially malicious intentions, a high security standard is required. The standard should include: container images sanity check policies, access control for both containers and users, and techniques to minimize the container attack surface. [33]

Simplified networking Containers must be able to communicate across nodes in an efficient and secure manner. This requires the mapping of allocated host ports to containers. The overhead corresponding to this mapping intensifies at scale. The CO platform should thus provide a flexible, secure and scalable solution for networking. [33]

Service discovery The large number of services present in a cluster need to be able to communicate with each other. In traditional clusters services were handled as pets (i.e, having static names, IP addresses), however, in dynamic environments such as container orchestration platforms, they are regarded as cattle. The CO platform must provide mechanisms for addressing/labeling/grouping and service discovery. [33]

Monitoring and governance The container orchestration platform needs to support traditional monitoring techniques such as logging, resource usage and network trace-routes. Monitoring must be possible at both the level of the underlying infrastructure and the containers themselves. [33]

Integrating for continuous integration and delivery Software development teams should be able to integrate the CO platform within their employed continuous integration and delivery (CI/CD) pipeline. The CO platform should contain mechanisms for rolling updates, rollbacks, etc. [33]

2.4.2 Kubernetes

Kubernetes [39], commonly referred to as K8, is one of the most popular and adopted orchestration systems. It is an open-source project led by Google. Kubernetes is Google's solution for the growing demand of container deployments by external developers in its public business cloud and is based upon its predecessors Borg [64] and Omega [54] that have been used to schedule the internal Google workload. Its main design goal is formulated as: *"to make it easy to deploy and manage complex distributed systems, while still benefiting from the improved utilization that containers enable [6]"*.

To achieve the above stated goal Kubernetes introduces a number of concepts for both containers and cluster resources. It implements the infrastructure as code model by provides an abstraction layer on top of the physical infrastructure [25]. It allows to setup and manage container infrastructure by the configuration of these introduced concepts via a REST API or declarative YAML configuration files. Below the most relevant concepts are introduced.

Kubernetes concepts

Pods. A pod is the smallest unit of deployment within Kubernetes. It is a group of one or more containers that logically belong together. A pod and thus its containers run on the same node. They share the same network, storage and context (Linux namespaces, cgroups). A pod gets assigned a unique IP address. Pods are not self-healing meaning when an error occurs within the pod or during scheduling. It will be deleted. Due to this short life cycle using a single pod resource and its assigned IP address for applications is impractical. Kubernetes handles this by employing controllers and services. [36]

Deployments. A Deployment controller manages a pod or 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). A deployment, in addition, allows for declarative updates. These updates can be used to change the number of container replicas of a ReplicaSet or to update a specific container image within the pod. The Deployment controller changes the actual state to the desired state described by the update. [34]

Services. Services offer a solution for the short life cycle of pods (and their IP addresses). Services within Kubernetes offer a manner to expose a ReplicaSet of Pods via a unique name, stable IP address, network policy and ports. To determine which set of pods is targeted by a service, Kubernetes employs a Label selector. Labels are the core grouping primitive of Kubernetes and unlike names and UIDs do not offer uniqueness.
A Service can be exposed outside the cluster by using an external load balancer or by specifying a NodePort. When using a Nodeport each node within the cluster will expose the port and forward request into the service. [38]

Namespaces. Namespaces allow to partition resources of a physical cluster among multiple user 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. [35]

Resource limits. Kubernetes allows the allocation of compute resources of both containers and Namespaces by the means of resource limits. These limits can be soft (limit) and hard (request) limits. A Request specifies the quantity of resource that is guaranteed to the container. A limit specifies the maximum quantity allocated to the container. When the requested resource quantities of a container are less than its limit, the container may be allocated additional resources if there are unallocated resources available. The current supported compute resources are CPU, memory and storage within the root partition of the local node. Within a Pod it is possible to specify both request and limit for each container. When request and limits are specified, the scheduler will use them for both scheduling and eviction (when node capacity is reached) decisions. [37]

Kubernetes architecture

The basic architecture of Kubernetes is illustrated in Figure 2.4. A client-server architecture is employed in which master and node setups are deployed on different machines. Below the different components that build up the architecture are briefly explained.

API Server. The API server is responsible for the configuration and validation of API objects (pods, deployments, services,...). It offers communication on cluster state via a REST interface for both components and administrators. [58]

Controller manager. The controller manager is responsible for managing several core control loops part of Kubernetes. A control loop uses the API server to observe the shared state of the cluster and attempts to move to the desired state via configuration changes. [59]

Scheduler. The task of assigning pods to nodes is the responsibility of the scheduler component. The scheduler attempts to do a reasonable placement based on resource and quality of service requirements (e.g., not place a pod on a node with insufficient resources). It is possible for users to control the placement of pods via NodeSelector tags or affinity and anti-affinity constraints in the configuration file. [56, 61]

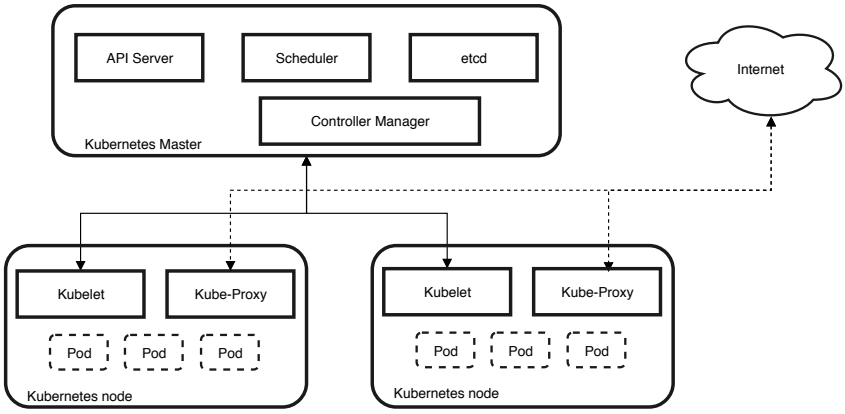
In addition it is possible to assign Quality of Service (QoS) classes to pods. These are used by Kubernetes in the decision process about scheduling and eviction of pods. Currently, Kubernetes supports three types of classes: guaranteed, burstable and best-effort. In decreasing order of priority (i.e., most likely to be killed in the case of

resource shortages). The QoS classes are assigned to pods based on the presence of request and limit specifications of resources in their configuration files. [57]

Kubelet. The kubelet component is an agent running on each node. It is responsible for running and maintaining pods on its residing node. The set of maintained pods is described in the form of PodSpecs (mainly received via the API-server). [62]

Kube-proxy. The kube-proxy daemon provides a simple network proxy for the services on each node. It enables forwarding of requests to the correct containers and can provide primitive load balancing. [60]

Figure 2.4: Kubernetes architecture



2.5 Performance evaluation

As discussed in previous sections, SaaS providers employ multi-tenancy to improve cost efficiency and offer several performance guarantees to their customers in the form of SLOs. An unavoidable consequence of multi-tenancy is the need to support a growing number of users in a single system. Providers need to have a clear insight into the *scalability* of their systems. However, scalability is a difficult thing to define, let alone quantify. Citing the words of Dr. Neil J. Gunther: *"if you can't quantify it, you can't guarantee it"*. [23]

2.5.1 The Universal Scalability Law

Dr. Neil J. Gunther provides a formal definition of scalability: *"scalability can be defined as a mathematical function, a relationship between independent and dependent variables (input and output)" [23].* The Universal Scalability Law (USL) by Dr. Neil J. Gunther is presented in Equation 2.1. It computes the relative capacity $C(N)$ at

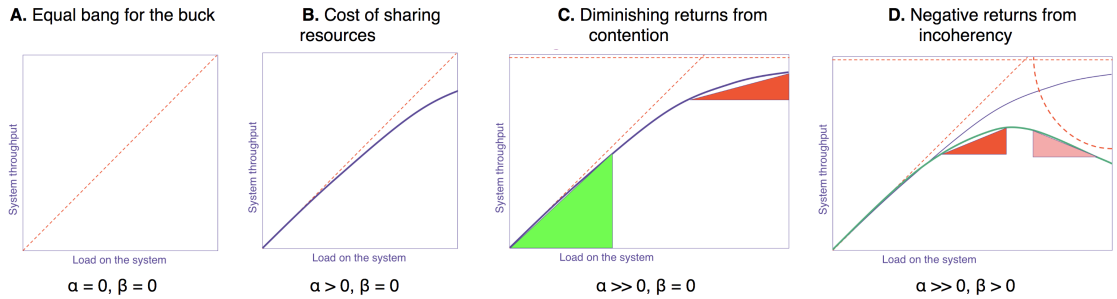


Figure 2.5: The impact of different USL coefficients on scalability. [23]

a load of N users. Relative capacity is the normalized throughput.

$$C(N) = \frac{\gamma\ N}{1 + \alpha\ (N-1) + \beta\ N\ (N-1)} \tag{2.1}$$

The Universal Scalability Law incorporates factors that contribute to the sublinearly scalability of most systems. Namely, **concurrency** (γ), **contention** (α) and **coherency** (β) as explained below. Their impact is visualized in Figure 2.5.

- **Concurrency**(γ): γ defines the slope if the system was linearly scaling i.e., $C(N) = \gamma N$. It has been referred to as the *coefficient of performance* by [53].
- **Contention** (α) : When scaling most systems parallelism while be limited at some point by contention (i.e., waiting or queuing for shared resources). The maximum speedup by parallelism is limited by the serialized portion (α) of the work [53].
- **Coherency** (β): created by crosstalk between components. Because crosstalk is possible between each pair of components in the system, the penalty grows quadratic $N(N-1)$. [53]

USL in practice

USL can provide insights into a system's scalability pains via values of the concurrency, contention and coherency coefficients. These can be obtained by collecting a dataset of measurements of the system, system load N and corresponding throughput, and using a statistical technique such as nonlinear least square regression which fits the USL to the dataset. [53, 27]

2.5.2 Little's law

The Universal Law of Scalability serves as an alternative for the often less intuitive queuing models frequently used for modeling scalability. It omits the need to know the service time for every queue in the performance model in order to predict the

response time or latency [23]. Nevertheless, queuing theorem and its lemmas such as Little's Law can provide useful insights into performance modeling.

John D.C Little's Law [41] states the following for stable systems:

> *"The average number of items in a queuing system equals the average rate at which items arrive multiplied by the average time that an item spends in the system.[41]"*

$$L = \lambda \, W \tag{2.2}$$

$$N = X \; Rt \tag{2.3}$$

Equation 2.2 shows Little's law, below its terms and its applicability to web services (Equation 2.3) are explained:

- L: Average number of items in the system. For a web service this is represented by the average number of concurrent users in the system N.
- λ: Long-term average arrival rate of items in the system per time unit. Little's law assumes a stable system for which the arrival rate and exit rate are identical. In a web service this is represented by the throughput X.
- W: Average waiting time of an item in the system, queuing time and service time combined. This is represented by the response time Rt or latency of a request in a web service. When dealing with a system involving think time (e.g., after the response from a web service, a user needs time to think about his/her next request), $(Rt + Zt)$ is used.

Workload generator validation

Developers employ software tools (JMeter [30], Locust [42], etc.) to simulate a workload and test the performance of a system. A workload is a set of actions that represent the behavior of a client in the system. It is part of the test plan stating the number of concurrent users N executing the workload for a specified period of time.

The results of a performance test are typically expressed in throughput X and response time Rt. Using Little's law it is possible to validate these results. For example, a test plan of 1000 concurrent users N results in a throughput of 50 requests per second and an average response time of 15 seconds. Following Little's law, a concurrence of only 750 users was reached, instead of the specified 1000. Little's law can thus be used to check if a workload generator works as specified.

Alternatively, if the average throughput and response time for a production system are known, Little's law can be used to correctly draw up a test plan.

Response time or latency?

Performance can either be measured in throughput or latency, both are correct but offer a different point of view. System performance is typically expressed in throughput: "The system can handle a million operations per second". However, users care more about their personal experience with a system which is influenced by the average latency of their request. [53]

Figure 2.6 shows the relation between the average latency and throughput of a database under increasing load N. In this case, latency is kept stable by increasing the throughput with the increasing load up to a certain point. A bottleneck caps the throughput of the systems and by Little's law for an increasing load and constant throughput, the latency must increase. [28]

However in other systems, as discussed in Section 2.5.1, an increasing load might induce a diminishing result on the system's throughput. Following Little's law a decreasing throughput results in a higher latency under the same load.

Thus, for stable systems by Little's law, the USL can be reformulated in terms of latency instead of throughput. Equation 2.4 shows this reformulation. [53]

$$Latency(N) = \frac{1 + \alpha\ (N-1) + \beta\ N\ (N-1)}{\gamma} \tag{2.4}$$

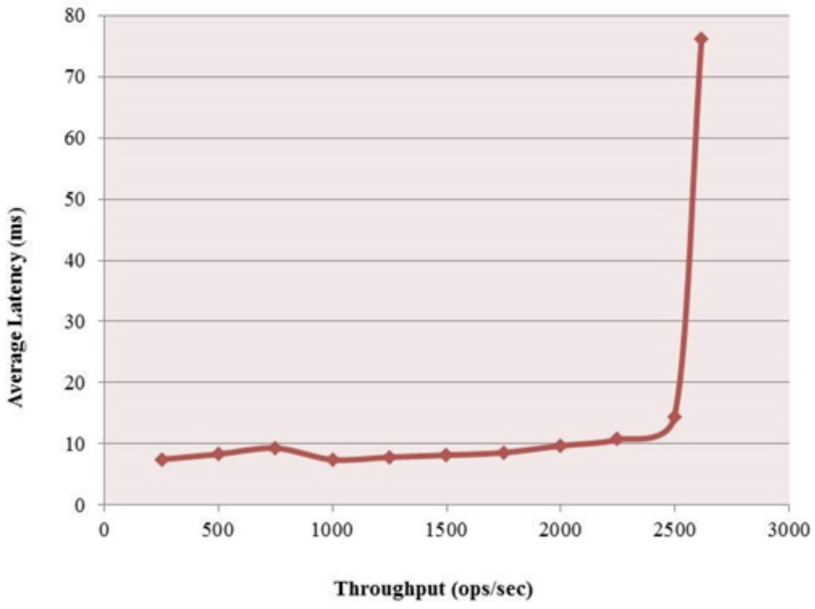


Figure 2.6: Relation average latency vs. throughput for database benchmark with increasing load. [28]

Chapter 3

Related work

This chapter discusses relevant related work that serves as inspiration for the work of this thesis. In Section 3.1, two different approaches for achieving multi-tenancy on top of Kubernetes including the consequences of tenant aggregation are discussed. The remaining sections discuss several existing state-of-art approaches related to the SLA-decomposition problem tackled in this thesis. Section 3.3 argues which approach will be the basis of the work presented by this thesis.

3.1 Multi-tenancy via container orchestration

In the following sections discuss proposals made by members of both the Kubernetes working group and research community to adapt or leverage Kubernetes concepts to achieve multi-tenancy at different levels.

3.1.1 Hard multi-tenancy in Kubernetes: K8 is the new kernel

Kubernetes, initially released in June 2014, is still under rapid development. Enabling support for multi-tenancy is one of the aspects the community's working group is focusing on. This section discusses some recent architectural proposals of the community to achieve hard multi-tenancy in Kubernetes. Information in this section is retrieved from a blog post [1] of Jessie Frazelle (*Microsoft*) [19], a respected former docker employee and Kubernetes working group member, and documents [20] of the K8's multi-tenancy working group.

Hard multi-tenancy

Multi-tenancy, as discussed before, can be achieved on multiple levels in the technology stack. All focusing primarily on the improvement of resource utilization. Another differentiation, based on trust assumption between the tenants, can be made between to classify approaches. When tenants of a K8 cluster are assumed to be non-malicious and security isolation measures merely serve to prevent accidents, the term *soft*

[1] https://blog.jessfraz.com/post/hard-multi-tenancy-in-kubernetes/

multi-tenancy is used. Such as cluster sharing within one organization. Soft isolation often allows tenants to use unused resources that were allocated to other tenants, but enforce quotas when necessary [48]. On the other hand, *hard multi-tenancy* refers to the sharing of cluster resources among (potentially malicious) tenants from multiple organizations which do not fully trust each other. The remainder of the section focuses on achieving hard multi-tenancy in Kubernetes [20].

From a security point of view, Kubernetes has two potential attack surfaces: unauthorized use of Kubernetes API and remote code execution in containers of running services. Different tenants should thus not be able to manipulate each other's clusters and their containers should be isolated. An exploit leading to supervisor privileges for a single tenant could have a devastating result. [19]

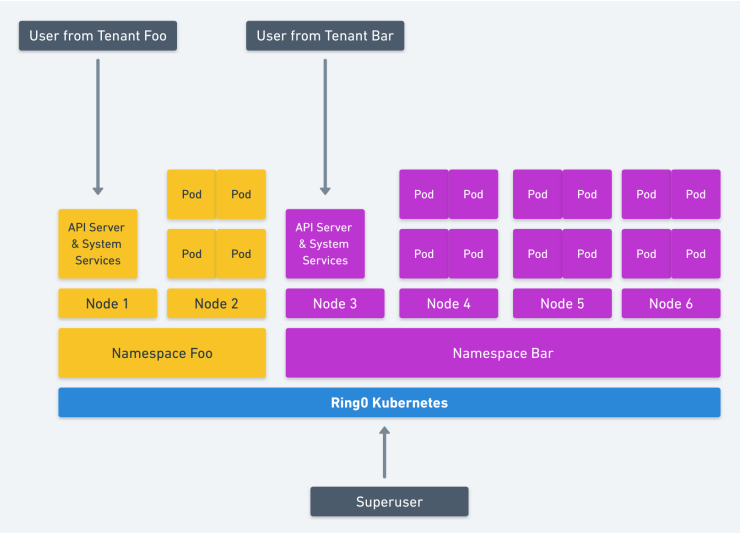
Proposals

For effective security, a system should be composed out of multiple layers. The failure of a single layer should not compromise the entire system. The following paragraphs discuss how such a multi-layered approach can be achieved within Kubernetes via the proposals of [19].

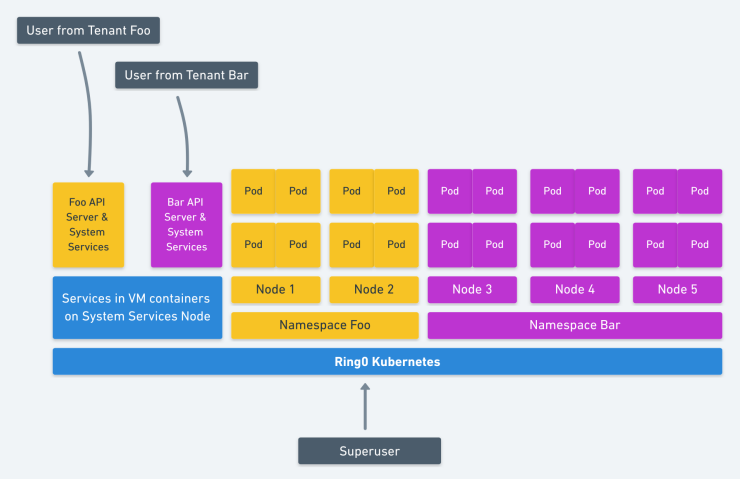
K8 in K8 Namespaces Kubernetes Namespaces are seen as a boundary for separating multiple users in the cluster. Kubernetes system services such as the API-server, however, are shared between these users. An exploit of a single namespace could possibly affect all the other namespaces. A proposal to eliminate this attack surface, is for each namespace to have its own Kubernetes system services (running in pods within that namespace). An exploit of these services would not grant access to the root Kubernetes services but be contained to the namespace. This idea borrows from Nested Kernel: Intra-Kernel Privilege Separation [10]. Here, a small kernel nested inside the monolithic kernel allows for privilege isolation. By employing this nested structure, users can strictly see the resources assigned to them in the namespace via their own system services. This matches the purpose of namespaces in Linux: what a process can see, is controlled by the namespace. [19]

Nodes assigned to Namespaces Kubernetes namespaces currently employ quota (CPU and memory) to limit resources utilization of the services designated to them. In order to have even stricter isolation between services of different namespaces, a proposal is a stricter assignment of what is exactly controlled by namespaces. One possibility is to assign nodes at the machine level to a particular namespace. All services (including system services) would be constrainted to run on these machines. Services of different tenants would never run on the same machine. And namespaces would have no notion of other nodes. This strict assignment combined with the separate API from the previous paragraph is illustrated in Figure 3.1(a). [19]

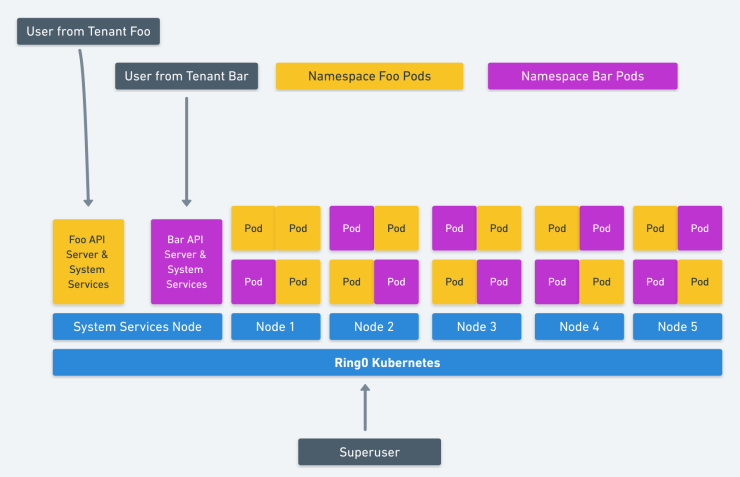
In order to achieve a better resource utilization, Kubernetes system services could be run in isolated VM containers (such as the Kata lightweight VM containers [18])



(a) Namespaces on separate nodes.



(b) Namespace system services on shared node.



(c) Containerized nodes to share physical nodes.

Figure 3.1: Proposal architecture for hard multi-tenancy in Kubernetes. [19]

on a designated services node. This is shown in Figure 3.1(b). [19]
A third option is to completely virtualize the system. Here a node assigned to a namespace and on which containers are executed is itself a VM container. This enables the sharing of physical nodes between namespaces. Figure 3.1(c) illustrates this option. [19]

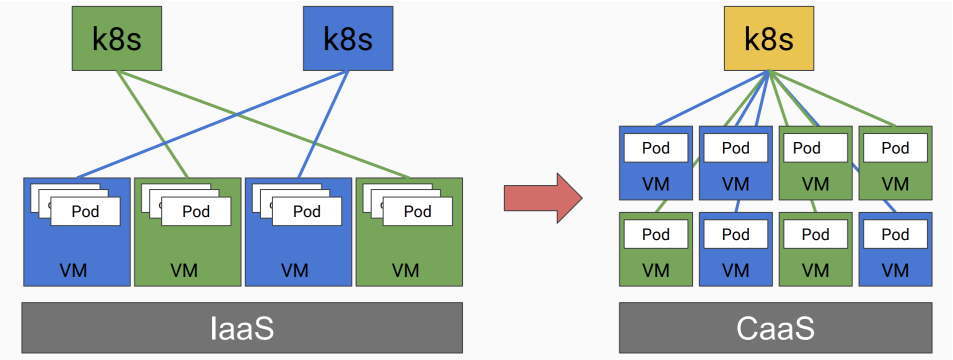


Figure 3.2: Evolution from IaaS to CaaS using containerized lightweight virtual machines. [11]

Retrospect

The proposals presented above diminish the attack surface for logical vulnerabilities of Kubernetes and the Kubernetes API by assigning each namespace its own separate system services. Additionally, containers from different namespaces are isolated by full virtualization. This is achieved by dedicated nodes or by containers running lightweight VMs. Combined these techniques deliver a strong solution for the threat model of hard multi-tenancy.

From a critical viewpoint, the question of the benefits of this model compared to running multiple separate Kubernetes clusters comes to mind. In the latter, resource sharing between clusters on a physical node happens at the level of virtual machines, which may be less efficient. An added benefit of the approach above is the less cumbersome administration of the cluster through its centralized nature. In essence, these approaches resemble the transition from Infrastructure as a service (Iaas) to Containers as a service (CaaS).

3.1.2 Towards a container-based architecture for multi-tenant SaaS applications.

The paper "*Towards a container-based architecture for multi-tenant SaaS applications*" [63] presents a container based middleware platform architecture and three possible deployment strategies to achieve multi-tenancy SaaS applications within a container orchestration environment. This work is briefly discussed in following paragraphs.

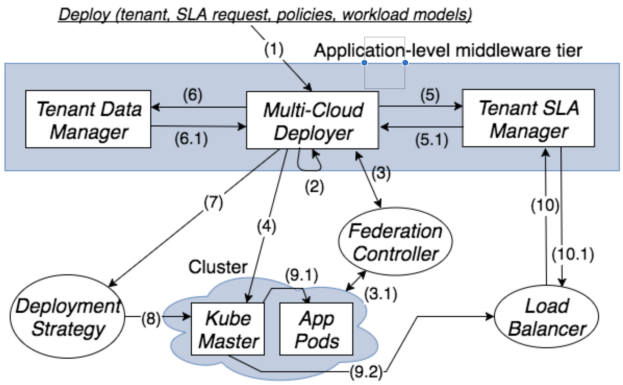


Figure 3.3: Middleware architecture for multi-tenant SaaS applications. [63]

Architecture of middleware

A possible high-level architecture presented by the paper is shown in Figure 3.3. It is composed out of three services, each implementing a policy-based architecture that can be extended with a control loop in which decisions are made based on application-centric monitoring. [63]

The *Multi-Cloud Deployer* is responsible for deploying SaaS application across multiple container orchestration clusters from different cloud providers. In this specific example architecture, it interacts with a Kubernetes cluster. It deploys namespaces and applications based on specifications delivered by the *Tenant SLA Manager* and *Tenant Data Manager*. [63]

The *Tenant SLA Manager* manages tenant SLAs and provides performance isolation by specifying the employed *Requests and Limits* of the application containers in each namespace. It also manages admission control for each tenant via the front-end loadbalancer (assumed present for each application). Lastly, the *Tenant Data Manager* provides adaptive management of multiple database technologies. [63]

Deployment strategies

The paper [63] describes three possible deployment strategies for SaaS applications which employ container or container orchestration concepts. These strategies differ

in the chosen trade-off between cost-efficiency and security isolation.

Shared container Each application instance of an existing multi-tenant application is allocated one container which is part of one pod. Each pod is run on a separate node. Resulting in each node in the cluster running one single container capable of handling client requests. The application instance must handle multi-tenancy itself (i.e., performance isolation and enabling tenant-specific features for all classes of tenants are handled at run-time). This approach shares resources at the highest level and does not leverage any security or performance isolation capabilities of the underlying container orchestration platform.

In terms of a Kubernetes specific deployment, clients connect to a single service (with a stable IP) which distribute requests among the pods. The pods are controlled by a single specified deployment configuration (next-generation replica controller). [63]

Namespace per tenant Each tenant is assigned to its own separate, dedicated namespace. This allows to specify the exact resource requests and limits of pods needed to satisfy the SLA requirements of the tenant. Furthermore, it allows the container images to be tailored to solely load the tenant-specific features. In contrast to the previous strategy, here the advantages of the underlying platform such as security isolation between tenants are being used. However, this approach of using container orchestration as middleware for multi-tenancy is less cost-efficient than the previous.

This strategy relies on the Kubernetes namespace concepts, allowing the division of cluster resources based on quota. For each namespace, a service and deployment have to be specified. Tenant forward request to their corresponding services. [63]

Namespace per SLA class The last proposed deployment strategy combines the previous two. Here tenants are grouped in distinguishing SLA classes. A separate, dedicated namespace with the corresponding quota is created for each class. The advantages of the previous approaches remain present to some extent. Sharing a container among tenants of the same class allows for a feature specific container and higher resource utilization but requires application-level admission control. It also allows for security between SLA-classes but not between tenants. [63]

3.1.3 Conclusion multi-tenancy on K8

The previous section discussed proposals showing Kubernetes applicability for hard multi-tenancy achieving Containers as a Service (CaaS) or as an underlying platform for multi-tenant SaaS applications. In both scenarios, a trade-off is to be made between security isolation and cost-efficiency. Despite this, the proposals aim to maximize cost-efficiency while minimizing the complexity of cluster management or the complexity of additionally needed admission control. Kubernetes and container

orchestration in general has the potential to be a building block for multi-tenant scenarios.

3.1.4 Consequences of tenant aggregation

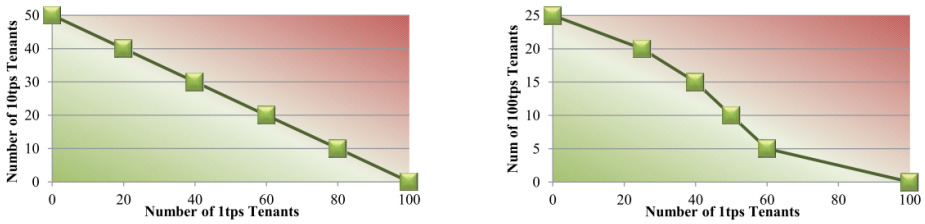


Figure 3.4: Mix of different SLOs scheduled at a single SKU. Points above the frontier line represent scheduling policies failing to meet tenant SLOs. Point on the frontier line meet all tenant SLOs. Scheduling policies in the area below the frontier line satisfy tenant SLOs but potentially waste resources. [40]

The work of [40] specifies the challenges of balancing multi-tenancy performance and operation costs in the context of performance-based Service-Level-Objectives SLOs for a Database-as-a-Service (DaaS) provider. The challenge is the specification of a hardware provisioning policy and a tenant scheduling. The former refers to the selection of machines (possibly different SKUs) needed for a given set of tenants. The latter is the mapping between tenants and provisioned SKUs that minimizes costs while meeting the SLO's performance goals. Figure 3.4 shows the alteration between homogeneous and heterogeneous tenant mixes of two SLOs on a single SKU. The frontier line shows the mix of tenant SLOs that satisfy all tenant SLOs without the waste of resources. When the frontier is linear (left), a provider can easily define a scheduling policy. However this is not always the case, as different parts of the system can become a bottleneck. The allocation problem becomes even more complex with a mix of SKUs. [40]

The authors propose a three-step plan for tackling this problem. Starting by determining the upper boundary for homogeneous tenants mixes of every SLO s_i on all available SKUs (i.e., how many tenants of class s_i can be scheduled on SKU without violating the SLOs). Secondly, a characterization function $\hat{f}(\vec{b})$ for each SKU is evaluated where b_i is the number of tenants of class s_i. The function returns either the average performance for each tenant class or $false$. A systematic search is employed to determine solely the values for which $\hat{f}$ is true, essentially constructing the frontier. The search starts from a homogeneous mix of high-performance tenants (output step 1) and iterative substituting a fixed small number of highest-performance tenants with low-performance tenants until SLOs are violated. When dealing with more than two SLOs all combinations need to be benchmarked. Using the characterization functions from the first two steps an optimization problem is formulated. A brute-force solver is used that explores the feasible regions. [40]

3.2 Dealing with Service Level Objectives

3.2.1 Morpheus: Towards automated SLOs for enterprise clusters

Morpheus [31] is a state-of-art cluster scheduler for recurring batch jobs. It presents interesting features such as the programmatic derivation of SLOs and job resources enabling a *tuning-free* experience for its users. Starting from a user study showing that users of a shared cluster are 120x more likely to complain about the (un)predictable completion time of their job than about the fairness of sharing. The study results also showed a wide range of under/over-allocation of job resources confirming the difficulty of manual tuning. The data-driven solution, briefly discussed below, is optimized for end-user satisfaction while being able to reduce the required cluster footprint.

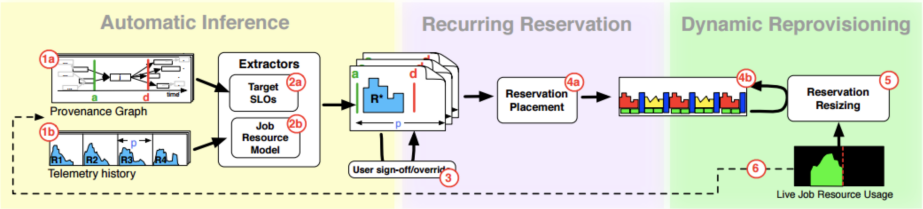


Figure 3.5: Life-cycle of a recurring job in Morpheus. [31].

Extraction of target SLOs SLOs in Morpheus are focused on job completion by deadline, the one observable metric its users care about. The extractor, responsible for the derivation of these deadlines, utilizes a provenance graph (PG). The PG is a graph representation of key properties of job and system information built from application, file-system and front-end logs. Individual jobs are grouped into periodic jobs by means of templating (based on job names, code signatures and inter-arrival times). By the use of statistics on the history of grouped jobs, a deadline can be offered.

Extraction of job resources For the inference of job resources, the extractor relies on the input of detailed telemetry information. Based on time-series of resource utilization from multiple job runs, Morpheus solves a linear programming optimization problem (including terms for over- and under-allocation penalties) to find the best fit for all historic patterns. It thus produces a resource allocation that approximates the actual requirements of the periodic job. [31]

User sign-off or override The inferred SLO and resource model is presented to the user in the form of a recurring reservation request. He/she has the ability to either sign of on the proposal, add further knowledge to the proposal by overriding parameters or reject the proposal. In the last case the job is executed using the standard fair-queuing semantics of the cluster. [31]

Reservation placement Upon approval, the recurring reservation request is a reservation placement mechanism. Morpheus extends YARN's standard reservation system with a novel online packing algorithm specialized for periodic jobs, called LowCost. For the details, the consultation of the paper is recommended. The use of reservations enables to prevent unpredictability introduced by a shared environment. Recurring jobs are scheduled in a manner that isolates them from each others noise (e.g., non-interfering workloads, which do stress the same resource type, will not be allocated on the same node). [31]

Dynamic reprovisioning While LowCost eliminates sharing-induced unpredictability, Morpheus has to deal with the challenge of inherent unpredictability. Examples of factors that introduce performance variance include but are not limited to, changes in SKUs (different machine configurations - Stock Keeping Units), code base changes or skew in input data. These issues are mitigated by a dynamic re-provisioning algorithm which continuously monitors a job's resource consumption. For a slower-than-expected job, the resources are enlarged. The resource consumption is constantly fed back to back to the provenance graph and telemetry information for future job runs. [31]

3.2.2 SLA Decomposition: Translating Service Level Objectives to System Level Thresholds

In SLA Decomposition [9] Yuan Chen *et al.* present an approach for translating SLOs to lower-level resource quotas for each component of a web service. Their solution combines *performance modeling* with *performance profiling* to create models. Traditionally, these complex translations are made by domain-experts relying on past experience with specific applications deployed in a particular environment. Design decisions about operating systems, middleware, infrastructure and the inherent dynamism of systems make the automated derivation of system thresholds from high-level goals (SLAs) a difficult challenge. [9]
The definition of the SLA-decomposition and approach from the paper are discussed in this section.

SLA decomposition task

In the paper, the SLA decomposition task is defined as finding a mapping between service level objectives (e.g., response time and throughput) and the state of each individual component in the system (e.g., resource requirements and configuration). An example from the paper is given the SLOs (R, T), expressing objects on *Request latency* and *Request throughput*, and a typical 3-tier application composed out of n HTTP server, an application server and a database, the decomposition is finding the following mapping:

$$(R, T) \rightarrow (\theta_{http-cpu}, \theta_{http-mem}, \theta_{app-cpu}, \theta_{app-mem}, \theta_{db-cpu}, \theta_{db-mem}) \tag{3.1}$$

The problem of SLA decomposition can be viewed as the inverse of the performance modeling. In performance modeling, a model based on resource utilization and configuration settings is used to predict the overall system's performance. [9]

SLA decomposition approach

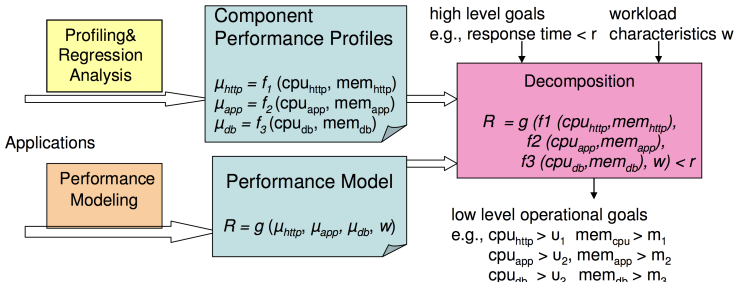


Figure 3.6: Conceptual architecture of approach for SLA decomposition by [9].

Figure 3.6 illustrates the presented approach of [9] to tackle the SLA decomposition problem. It builds components performance profiles and a performance model and combines them to solve the decomposition problem. Each step is briefly explained below.

Component profiling: A component profile characterizes a component's performance as a function of resource allocation (e.g., memory, CPU) and configuration. This function is derived using a regression-based approach on performance measurements of the component such as mean service rate μ. The measurements are obtained from benchmarks with a variety of workloads in which resource allocations of the component are altered in each iteration. The result of this step is a function of the following form for each component:

$$\mu = f(CPU, MEM) \tag{3.2}$$

Performance modeling: A performance model is used to define the relation between every single component and the overall system performance. Given a particular workload w and set of components' performance characteristics, the model predicts the performance of the combined components. For the 3-tiered web service the model g for response time is proposed :

$$R = g(f_{http}(CPU, MEM), f_{app}(CPU, MEM), f_{db}(CPU, MEM), w) \tag{3.3}$$

To obtain such a model, the paper proposes to model the multi-tier web service as a novel queuing network model. Each tier is represented in the network as a multi-station queuing center (i.e., G/G/K queue), illustrated in Figure 3.7. This reflects the commonly used multi-threaded architecture of servers. The performance of the queuing network is evaluated using Mean-value analysis (MVA). Using this

approach, the authors claim that the performance of a multi-tier application can be accurately predicted using single tier's performance and workload characteristics. The details of the model and analysis are omitted in this thesis. [9]

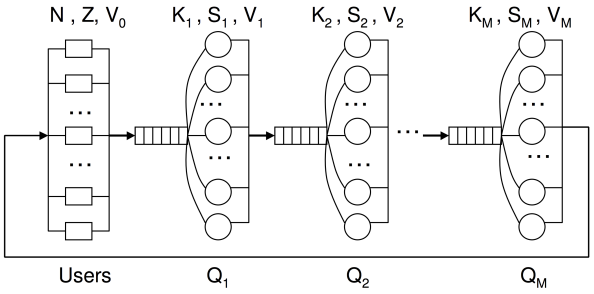


Figure 3.7: Multi-tier web application represented as a closed multi-station queuing network. It models N concurrent users sessions. A user session consists of a succession of requests with a think time Z between each request. Each tier i is represented by queue Q_i with K_i worker threads. S_i and V_i respectively represent the mean service rate and mean request rate of tier i. V_0 denotes the average request rate issued by the users. [9]

Decomposition: Given both performance profiles and performance models for performance characteristics used in the SLOs, using the corresponding equations the decomposition problem can be defined as a constraint satisfaction problem. An example of such a problem with constraints maximum response time r and minimum throughput X is given below:

$$g(f_{http}(CPU_{http}), f_{app}(CPU_{http}), f_{db}(CPU_{db}), w) < r$$
$$g(f_{http}(CPU_{http}), f_{app}(CPU_{http}), f_{db}(CPU_{db}), w) > x$$

For solving constraint problems various methods exist such as linear programming or optimization techniques. However, the solution space is often large and the solution is non-deterministic. As a remedy, the authors suggest the employment of a coarser granularity of parameters (e.g., 5% CPU slices) to minimize the search space or the use of heuristics (e.g., response time minimizes with an increase of CPU) for a more efficient search. The search may halt as soon as a feasible solution is found. [9]

In [5], a similar approach is used. A three-layered queuing network of resources is used as a performance model for containerized cloud applications. The layers respectively correspond to applications, containers and virtual machines. In contrast to component profiling, the paper implements Kalman filters [32] in a MAPE-K [12] based architecture in order to estimate the model parameters. Values gathered by the *Monitoring* and *Analysis* modules are evaluated towards specific thresholds (depending on the service level objectives). The *planning* module uses the performance model to determine resource scaling actions based upon the violation of one of the thresholds. In order to deal with the irregularities of web traffic, a *Scaling Heat algorithm* is proposed to avoid unnecessary scaling actions.

3.2.3 Automated configuration tuning: Best Config

The paper *BestConfig: Tapping the Performance Potential of Systems via Automatic configuration tuning* [69] presents an automatic configuration tuning system for general systems in the cloud that can optimize performance goals by adjusting configuration parameters. It recommends the best configuration found within a limit number of allowed tests. This is achieved by combining an effective sampling method called *divide-and-diverge sampling* and *recursive-bound-and-search*, a search-based optimization algorithm. Both are discussed in the remainder of this section.

Parameter tuning

A good configuration setting can lead to a strong improvement of a system's performance, especially for systems with a recurring workload. Performance tuning refers to the process of searching for the configuration with the best performance. In a manual setting, the process consists of finding a heuristic approximating the performance model of the system, adjusting configuration setting based on the model, running benchmarks and observing the corresponding performance. If the desired performance is not achieved, the heuristics need to be adjusted and the process is repeated. Resulting in a time-consuming and labors process. In addition, the growing number of configuration parameters of recent systems (e.g., 180 in Hadoop) can lead to complex configuration issues and complicates the search for heuristics. [69]

BestConfig tries to address the challenges of automatic configuration tuning simultaneously. By offering a decoupled architecture, it is possible to *support a variety of systems, workloads and deployment environments*. Easy usage within the deployment environment, is necessary because the deployment environment can heavily influence a system's performance model. Furthermore, the high number of parameters in recent systems led to a *high-dimensional parameter space*. The high cost of sampling makes it impossible to get a complete image of the mapping between all configurations and their performance. Since building a performance simulator (e.g., Starfish [26] for Hadoop) for every system is unfeasible, automatic configuration tuning should work with a *limited amount of samples*. In addition, it should allow the result to be improved when the number of samples increases. Lastly, performance is represented in different metrics for different systems. For example, for a data analytic cluster the goal might be to reduce the total run time, while for a web service the throughput needs to be maximized. Some systems might even have multiple performance goals. [69]

Divide & diverge sampling (DDS)

In order to get a wide coverage of the high-dimensional parameter space, in BestConfig it is *divided* into subspaces. Given n parameters, each parameter's range is divided into k intervals. Combining a sample from each interval results in a sample set with a granularity exponential to the number of parameter dimensions (i.e., k^n combinations). This is illustrated in Figure 3.8(a) by all the dots. Resulting in a high

cost for sampling. The authors suggest, to reduce the granularity but keep a wide coverage, the sample set can be *diverged* the most by representing each interval of each parameter exactly once. They observed that the combination of all parameter intervals is nonessential since: *"the impact of an influential parameter's value on the performance can be demonstrated through the comparisons of performances, disregard of other parameters' value [69]"*. In Figure 3.8(a), the green dots are thus the selected samples to be tested. With a larger resource limit (i.e., number of tests), the number of intervals k can be increased, resulting in a better image of the parameters space. Resulting in a scalable sampling method. [69]

Utility function

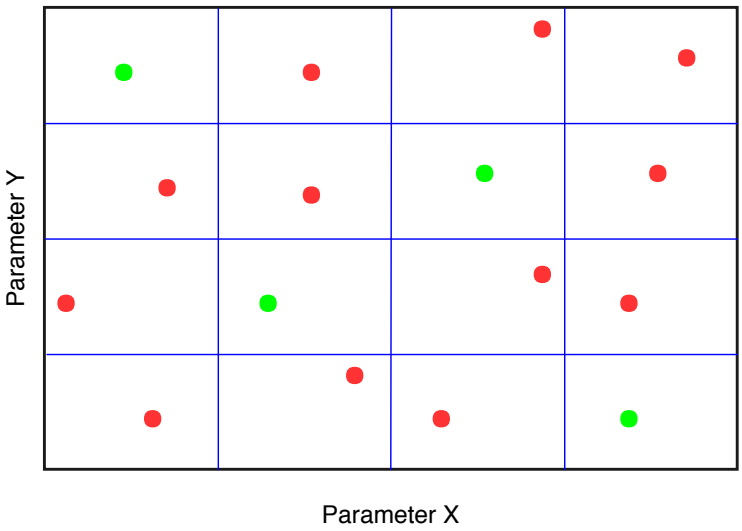
For simplicity, RBS (explained in the next paragraph) optimizes towards a single, scalar performance metric. This metric is retrieved from a *utility function* which has user-concerned performance goals as inputs. In other words, it translates the benchmark result of a sample configuration into a single score. This score is either minimized or maximized by BestConfig. This function is defined by the user. An example given in the paper of a utility function f, where a user wants to increase throughput x and decrease latency l is:

$$f(x_t, x_l) = \frac{x_t}{x_l} \tag{3.4}$$

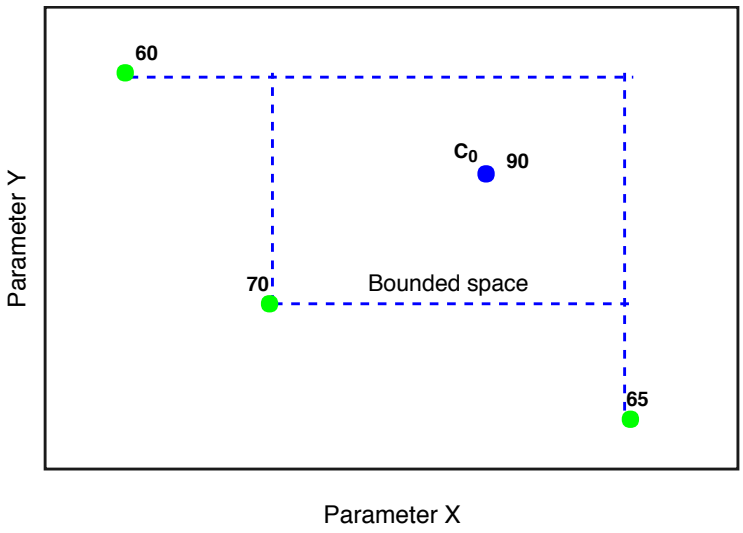
Recursive Bound & search (RBS)

Figure 3.9 shows the complete performance surface of a tomcat server. It shows the performance metric presented by the utility function, namely throughput. The goal of a performance optimization algorithm (PO) is to find the best configuration with a limited view of the performance surface. This view is the result of benchmarks on the sample set chosen by DDS. RBS is a search-based PO that builds on the following assumption of continuous performance surfaces: *Given a continuous surface, there is a high possibility of finding other points with similar or better performance around the point with the best performance in the sample set [69].* RBS thus searches the area near the sample C_0 awarded the highest score by the utility function. This area is referred to as the *bounded space*. The bounds of this space are chosen in the following manner for each parameter p_i. The new lower-bound for a parameter p_i^{min} is the largest value of p_i in the sample set that is smaller p_i of sample C_0. The upper bound for a parameter p_i^{max} is the smallest value p_i in the sample set that is larger than p_i of sample C_0. This is illustrated in Figure 3.8(b). If the resource limit (number of iterations) has not been exceeded, DDS is repeated with the new parameter ranges, as defined by the bounded space. Otherwise, C_0 is returned as the best found configuration. [69]

However, the PO might get stuck in local sub-optimum and consequently possibly not find the global optimum, even when given sufficient resources. To mitigate this case, BestConfig added the possibility of a recursive step. If no sample with



(a) 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.



(b) Illustration of BestConfig's RBS on a 2D parameter space. The blue dot C_0 indicates the parameter combination with the highest score on the utility function. The bounded space represents the search space for the next iteration.

Figure 3.8: Illustration of BestConfig's DDS & RBS.

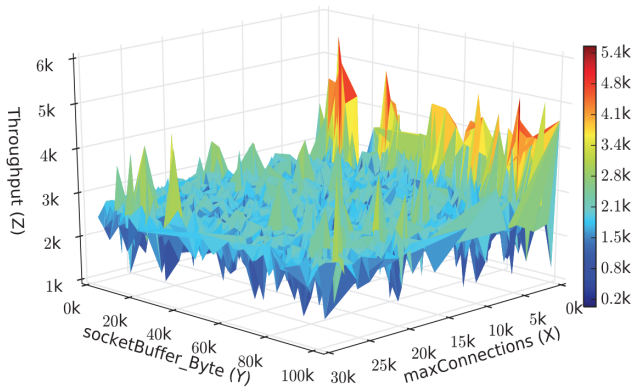


Figure 3.9: The performance surface of a tomcat server. The employed utility function reflects the throughput of a sample configuration. [69]

better performance is found in a new round, RBS simply restarts from the beginning with the complete parameter space. The paper gives a detailed explanation on why the combination of DDS and RBS works. This discussion is omitted in this thesis. [69]

Model-based method for performance tuning

Besides search-based approaches, it is also possible to use a model-based performance optimization algorithm. The authors of BestConfig argue that these methods require a larger number of samples given a high-dimensional parameter space and rely heavily on the knowledge of the developer. For example, the developer needs to know if the model is quadratic or linear but with a large number of components in deployments predicting this has shown to be a difficult task. Lastly, models often require the configuration of hyper-parameters, which is a tuning problem itself. To prove the infeasibility of model-based approaches, BestConfig was compared to both Co-training Model Tree (COMT) [8] and Gaussian Process Regression (GPR) [17]. The results show that the models make bad assumptions about the system resulting in wrong predictions, cannot output a competitive configuration compared to BestConfig and tend to over-fit when given more resources. [69]

3.2.4 Bayesian optimization as performance model for search minimization

Big data analytics face a similar configuration problem due to the rapid growth of available techniques and frameworks such as MapReduce, Spark and others. Finding a better configuration can result in large cost and time savings. CherryPick [2] is an optimization tuner for various applications that exploits the characteristics of Bayesian optimization to obtain accurate enough performance models in order minimize the search for an optimal or nearly-optimal configuration. The authors point out similar restrictions of both poorly adaptive model-based and costly exhaustive search approaches as mentioned in previous sections.

The use of Bayesian optimization allows to achieve several key characteristics wanted of an auto tuner. Adaptivity is possible since the approach treats every application as a black-box. The obtained models are solely accurate enough to rank configurations. By adapting the model every run, an interactive search can be conducted, resulting in low overhead. [2]

A run of the interactive search consists of running a job with a configuration, building and updating a black box model (relation between settings and cost/performance constraints), ranking configurations based on model and choosing the next configuration to run. The steps are repeated until the accuracy is high enough. Bayesian optimization has two components: a prior function and an acquisition function. The former models the cost and performance of the jobs by giving a confidence interval of possible fitting functions. The later is used for ranking the (remaining) configurations, it calculates the expected improvement compared to the current best configuration. This allows to emit areas in the space search with less promising configurations. An additional advantage of Bayesian optimization is its capability to incorporate additive noise into the confidence interval. This noise can be observed from historical data. A naive solution would be to run tests multiple times, resulting in a larger overhead. [2]

3.3 Conclusion

The related work discussed in the first section of this chapter shows that search for approaches to achieve multi-tenancy on top of container orchestration, specifically Kubernetes, is an active and interesting research topic. The fine-grained control of compute resources and flexible deployment offered by container orchestration can potentially be leveraged by SaaS providers to maximize their cost-efficiency while respecting their customer's SLAs. However, finding the best configuration for each application has proven to be a difficult task for developers and system administrators in general.

Approaches, discussed in this chapter, to deal with the assignment of resource thresholds in the presence of SLAs can be distinguished by whether or not performance models are used to search the best possible configuration. Model-based approaches suffer from several drawbacks such as high complexity for administrators, poorly adaptive to different applications and the requirement of extensive knowledge on the inner workings of an application. In addition, models do not guarantee the requirement of fewer configuration tests and even tend to overfit when offered more samples.

The optimization approaches mentioned in Sections 3.2.3 and 3.2.4 treat the application as a black box and do not require the construction of a model of the application. The use of general optimization techniques offers better adaptivity to a

variety of applications. In terms of usability, BestConfig's utility function is slightly more intuitive than CherryPick's Bayesian optimization. BestConfig's evaluation of configurations through continuous experimentation, eliminates uncertainty on the quality of the results present in predictions by model-based approaches. Therefore BestConfig's performance optimization algorithm is selected as the basis of the SLA-decomposition approach presented in this thesis. In the next chapter, it is adapted to fit the SLA-decomposition problem and implemented as part of a proof-of-concept.

Chapter 4

Simple batch processing application

This chapter describes a simple batch processing application that is used to demonstrate and validate the capabilities of the work presented in this thesis. The implementation of a simple application was chosen over an existing application for the ability to evaluate the work presented in this thesis in a controlled setting. The application described in this chapter is specifically designed to stress the CPU of a container. The quality of service described in SLOs of this application can thus be differentiated by regulating the allocated CPU.

4.1 Design

The design of the application is inspired by the architecture of the document processing application described in [65] and Pull Queues [22] available in Google App Engine. The application consists of two major components: the Queue and Workers. The design is shown in Figure 4.1.

4.1.1 Queue

The queue is used by tenants to submit jobs to the application via a publicly available interface. A job can consist of any amount of tasks of a specific type. Upon receiving a job, a unique reference for the job is created and the required tasks are added to the queue.

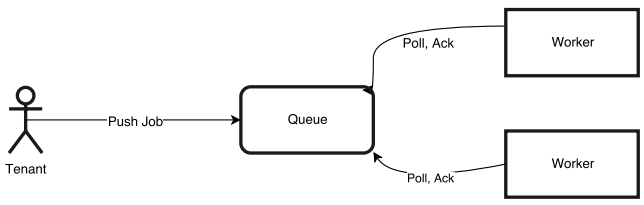


Figure 4.1: Design simple batch processing application.

Tasks within the queue have a unique identifier and a reference to their corresponding job. The unique identifier is used by the workers to acknowledge a completed task. When a task was pulled by a worker but has not been acknowledged after a certain time interval, the task is re-queued. This simple mechanism allows to cope with the possible failure of workers. When all the tasks of a job are completed, the job is marked as completed.

4.1.2 Worker

The worker component is responsible for the actual execution of a task. As by the pull design the worker will periodically poll the queue for work. Once a task is completed, the worker will poll the queue again. The result and identifier of the previous task are piggybacked by this poll. The type of work done by the worker is determined by the type of task.

4.2 Implementation

The components of the application are implemented as restful web services in the Java Spring Framework.

4.2.1 Queue

As a web service the queue offers the following REST API to the tenants and the workers.

- `\pushJob\amount`: will push a job with a certain amount of tasks onto the queue as defined by the path parameter.
- `\pollTask\acknowledge-task-uuid`: will poll the queue for a task. If a task is present within the queue, it will be returned as a JSON object. When a task identifier is specified as a path variable, the corresponding task will be successfully acknowledged.
- `\status`: returns a report about the current state of the queue.

The tool described in Chapter 5 requires the application to be testable by a load testing tool such as JMeter [30], Scalar [27] or Locust [42]. The majority of load testing tools perform synchronous HTTP-request to evaluate the performance of a web application. The synchronous nature of the HTTP-request implies that TCP connection is kept alive until a response is delivered to the tool. In the Spring framework, each HTTP-request is processed on a separate thread. However the batch-oriented application described in this chapter works asynchronous (i.e, jobs placed in the queue are asynchronously processed by the workers) and the completion of jobs might take a considerable amount of time. The connection and thus the thread must be kept alive during the entire execution of the job. This results in a build-up of active threads on the Queue component in the presence of multiple tenant requests.

To combine the synchronous nature of load testing tools with the asynchronous computation of the application, the concept of Java `Future` [4] is utilized. When a job is submitted it is associated with a `Future`, blocking the thread responsible for the HTTP-request. A `Future` allows the thread to be easily resumed (yielded) when the job has been marked completed by the queue.

4.2.2 Worker

The worker is implemented as a scheduled task from the Spring library. It will poll for a task after completing a task and each fixed time interval (1 second). A poll is executed by a thread. A poll thread is stopped when no new tasks can be retrieved from the queue. However, additional threads can be spawned each time interval. The currently implemented task execution for the worker stresses the CPU.

The implementation of the CPU stress test is inspired by the work of Matthews *et al.* [43] on quantifying the performance isolation of virtualization systems. The authors suggest to utilize a tight loop of integer arithmetic operations to individually stress the CPU. The implemented stress test shown in Listing 4.1 performs a loop of calculations in which the factorial of 30 is recursively calculated. The number of loops can be configured.

Listing 4.1: CPU stress test implementation.

```
public int run(){
    if(this.stressSize != 0){
        result = 0;
        for(int i =0 ; i < 100 * this.stressSize; i++){
            result = fac(30);
        }
    }
    return result;
}

private int fac(int n){
    if(n==1) {
        return 1;
    }
    else{
        int r =  fac(n-1);
        return r * n;
    }
}
```

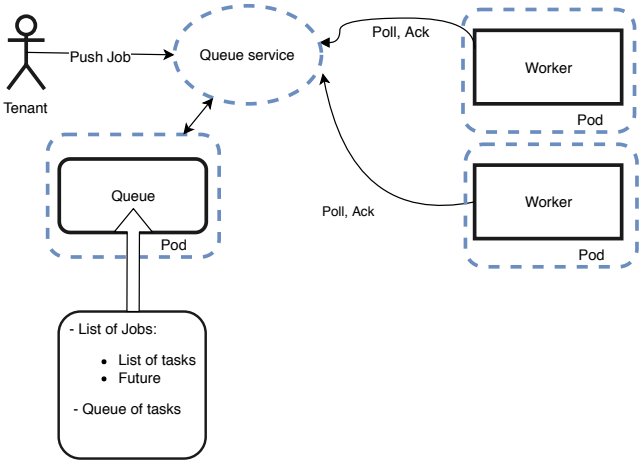


Figure 4.2: Deployment of the simple batch processing application within a Kubernetes namespace.

4.3 Deployment in orchestration platform

This section describes how the components are deployed within the container orchestration platform, Kubernetes. The deployment is shown in Figure 4.2.

4.3.1 Queue

The queue is deployed as a pod within a given namespace. The queue's REST API can be discovered/accessed via a Kubernetes service.

4.3.2 Worker

The workers are deployed as pods part of a deployment within the same namespace as the queue. The workers use the Kubernetes DNS to discover the IP of the service coupled to the queue. The number of workers can be changed by the replica parameter of the deployment configuration.

Chapter 5

k8-resource-optimizer

This chapter presents the architecture and implementation of k8-resource-optimizer, an automated SLA-decomposition tool for Kubernetes applications. The k8-resource-optimizer allows to create a SLO-decomposition for a particular application given a workload and tenant configuration. The first section of this chapter briefly revisits the problem statement of SLA-decomposition. Next, an overview of the essential requirements for the architecture of the tool is given. The remaining sections discuss the adaption of BestConfig's optimization algorithm for SLA-decomposition and the overall architecture of the tool.

5.1 Motivation

Today's industry-level SaaS providers face the constant challenge of offering their products at the best service level for the most competitive price. SaaS providers employ trends such as multi-tenancy and container orchestration in order to meet these agreed-upon SLAs while minimizing resource utilization, thus reducing operating cost. For the simple batch processing application, introduced in Chapter 4, the following SLOs are composed:

- Bronze tenant:
  - Throughput: 0.5 jobs per second
  - Job size: 250 tasks
- Silver tenant:
  - Throughput: 0.5 jobs per second
  - Job size: 500 tasks

As discussed in Section 3.1.2 resource provisioning policies of container orchestration platforms offer a manner to achieve Quality of Service differentiation. Given a set of SLAs and a service, a provider must translate these SLOs to component level resource requirements. This process is referred to as SLA-decomposition by [9]. As

an example, for the simple batch processing application and an SLO with throughput T and latency R, the SLA decomposition task could consist of finding the following mapping:

$$(T, R) \rightarrow (\theta_{cpu-queue}, \theta_{mem-queue}, \theta_{cpu-worker}, \theta_{mem-worker}) \tag{5.1}$$

However, enterprise SaaS applications have complex architectures consisting of numerous interacting components, making SLA-decomposition a difficult task. Simply finding a configuration that satisfies the SLO does not allow the provider to be competitive, constraints need to be introduced. A common goal is to prohibit *under-provisioning* and to minimize *over-provisioning*. A system is in an over-provisioned state if the amount of allocated resources if greater than the resources required to meet the SLO for the customer. In contrast, a system is under-provisioned when the allocated resources do not suffice to meet the SLOs, resulting in SLA-violation. Under-provisioning a component might potentially result in an unstable system causing high-level characteristics such as availability to become unpredictable. [1]

The task of creating a decomposition mapping is often performed by an expert or system administrator relying on their domain knowledge and past experience. A user study on job submissions to a production cluster reported in [31] shows that 70% of jobs were over-provisioned and 20% of them allocated 10x more resources than necessary. This indicates that finding the optimal resource allocations is a hard problem for the average developer.

In this work, we set out to design a tool for automatically tuning resource allocations for a given SLO and given workload. The next section describes the requirements for such a tool.

5.2 Requirements

The automatic derivation of a resource allocation for a given application and SLO is a complex task which is further complicated by design decisions such as architecture, underlying technology and deployment. For example, the performance evaluation of a job-oriented service differs from the methodology used for a long-running request-oriented web-service. An application can have a monolithic or microservice architecture. The underlying platform and the deployment strategy also influence the performance characteristics of an application. In sum, the tool should try to fulfill the following requirements:

- **R1:** Minimal overhead: SaaS applications compromised out of multiple components lead to a multi-dimensional search space for the optimal resource allocations that meet the given SLOs. A configuration should be found with a minimal amount of sampling. In addition, changes or extensions to commercial applications or the environment in which they are deployed are part of the daily development process. Thus, the overhead added to this process by the tool should be kept to a minimum.

- **R2:** Versatile: The tool should be able to be configured for a wide variety of architectures and workloads. The architecture could exist of any number of components, each having different components for tuning. Both long-lived workloads and batch workloads should be usable to evaluate the performance of a particular application.

- **R3:** Extensible: The architecture of the tool should be extensible. There should be the option to extend to other orchestration platforms, use a different performance evaluation method or employ an alternative for workload simulation.

- **R4:** Understandable: Most development teams do not have seasoned researchers at their disposal to build complex performance model for their applications. Or the scope of their infrastructure exceeds the scope of systems for which performance models were previously built, such as Hadoop. The tool should employ a performance evaluation method that is understandable, trustful and configurable.

- **R5:** Minimal invasive: The adaptions needed to be made to the applications in order to easily change the configuration need to be kept to a minimum. In the ideal situation, a developer should not alter her/his normal workflow.

5.3 Architecture

5.3.1 Overview

The k8-tool is inspired by the BestConfig application presented in [69]. As already shortly motivated in the conclusion (Section 3.3 of related work, the BestConfig approach is selected over model-based optimization approaches because of its adaptivity towards different applications and understandablity. The focus of the tool, as derivable from the name, is on Kubernetes applications and incorporates the notion of a *Service Level Objective* that needs to be met utilizing as few resources as possible. It consists of two major components, *SLA-decomposer* and *Bench*. It's architecture is shown in Figure 5.1.

The *SLA-decomposer* is responsible for the translation of Service Level Objects to resource requirements of the various components that make up the Kubernetes application. Its goal is to find the most optimal configuration of resources given an SLO.

The *Bench*-component actually tests the performance of a given configuration of the Kubernetes application by using a workload simulator, such as a load testing framework.

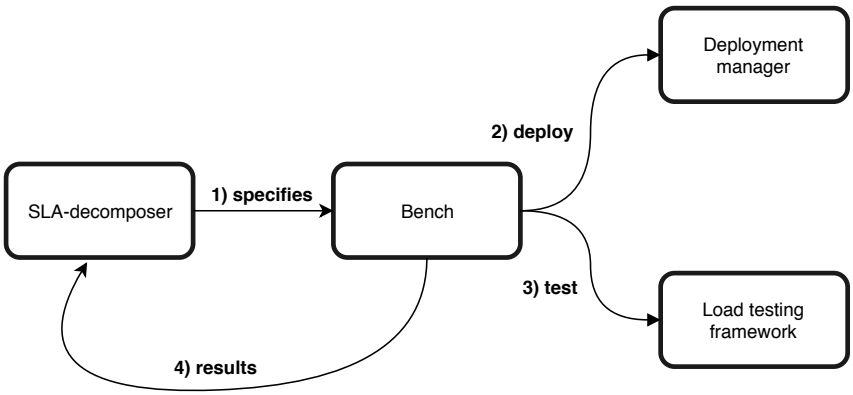


Figure 5.1: The architecture of the k8-resource-optimizer tool.

5.3.2 SLA-decomposer

The SLA-decomposer employs an adapted version of the performance optimization algorithm used by BestConfig [69]. The complete algorithm is explained in Section 3.2.3. The algorithm consists of three major components: **Recursive Bound & Search** (RBS), **Divide and Diverge sampling** (DDS) and a **Utility function**. The combination of RBS and DDS allows to find the best configuration in a high dimensional parameter space in a limited amount of samples and iterations. Allowing **R1** to be achievable. A configuration is considered as the best option when it receives the highest score by the utility function.

Input configuration

Listing 5.1 shows an example of an input configuration for the SLA-decomposer. A configuration consists out of multiple SLAs. Each SLA specifies the following options: Service Level Objectives to be met, amount of tenants and the parameters (resources) to be tuned. For each parameter a search space is defined which restricts the upper and lower bounds of the resource. It is also possible to specify a necessary prefix or suffix for the parameter (e.g., suffix m for 500m CPU). Lastly, a multi-tenant deployment strategy is configured. This is either *Namespace per SLA* or *Namespace per tenant* as described in Section 3.1.2. Also, the number of iterations and number of samples in each iteration for the PO algorithm can be specified.

Given a configuration the SLA-decomposer will complete the following steps in each iteration:

1. Take x samples of each parameter in their current search spaces using DDS.
2. Create a *Bench* configuration according to the specified deployment strategy, parameter samples and evaluation test(s).
3. Execute the *Bench* configuration.
4. Process the results of the evaluation method (specified in *Bench* configuration) to obtain a utility function score for the chosen parameter combinations.
5. Select the current best parameter combination and perform RBS to obtain the next search spaces for each parameter.

Utility function

In BestConfig, the utility function is a scalar function. It translates multiple user-concerned performance goals to a single value. In the case of SLA-composition in which SLOs have to be met with a minimal set of resources we propose the following utility function:

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

This utility function dismisses configurations for which the SLA is violated. An SLA is violated when the configuration has failed to meet one of the SLOs. For the configurations that do satisfy the SLOs the utility function creates a scalar value in terms of resource consumption. In this function, a lower resource consumption leads to a higher score. Each parameter (resource) can be assigned a weight according to the preference of the user.

When using a different application with possible different performance goals, a

user only has to specify an altered utility function. Other part of the performance optimization algorithm remain unmodified. The tool offers a understandable (**R4**) and reusable approach.

Listing 5.1: SLA-decomposer configuration file.

```
---
samples: 4
iterations: 3
# application configuration files
charts:
  - name: my-app
    chartdir: /exp/conf/helm/mychart
# SLA specifications and amount of tenants
slas:
  - name : gold
    chart: my-app
    slo :
        jobsize: 600
        throughput: 8
    amount: 1
    parameters:
      - name: workerCPU
        searchspace:
          min: 200
          max: 500
        prefix:
        suffix: m
      - name: workerMem
        searchspace:
          min: 100
          max: 300
  - name: silver ## namespace similar to gold

# Employed deployment strategy
strategy: NSPSLA # or NSPT

---
```

5.3.3 Bench

The bench component offers a manner of specifying and performing a benchmark for evaluating different configurations of a Kubernetes cluster. It allows the employment of different applications with various parameters deployed in multiple namespaces. Listing 5.3 shows an example of a benchmark plan.

A benchmark plan, expressed in YAML, consists of the following components: number of iterations, applications used throughout the cluster specification, namespaces that make up the cluster and the experiments used for evaluation. An iteration here is different from an iteration of the SLA-decomposer. Iterations allow multiple configurations to be specified in one file. Each iteration a parameter value is changed. This is discussed below.

Applications

The Kubernetes applications being evaluated during the benchmark are specified as Helm charts. The choice of Helm follows from its popularity in the Kubernetes community and the centralized method it offers to alter configurations of applications. Relying on Helms format of applications allows to achieve requirements **R2** and **R5**. Since most complex applications can be specified as a Helm chart and no changes need to be made to the application in order to easily make use of the tool.

Helm [24] is a package manager for complex Kubernetes applications consisting out of numerous deployments and services. It offers a structured manner of specifying these applications as a package, referred to as a chart, consisting out of multiple templates (deployments, services). A command line tool (helm) allows to install, upgrade and delete charts by communicating with a server (tiller) running inside the cluster. Additionally, Helm offers a manner to parameterize a chart through a built-in values object. Inside templates, references can be made to the fields of the values object. A user specifies the values object for a chart in a centralized values YAML file.

Listing 5.2 shows the directory structure for the simple batch processing application from Chapter 4. It contains deployment files for each component and specification files for both the namespace and the service.

Listing 5.2: Directory structure of simple batch processing application helm chart.

```
mychart
|-- Chart.yaml
|-- templates
|    |-- consumer-deployment.yaml
|    |-- queue-deployment.yaml
|    |-- queue-service.yaml
|    |-- namespace.yaml
|-- values.yaml
```

Namespaces

The namespaces section of the benchmark plan specifies the namespaces that make up the cluster. A namespace consists of a unique name, an application which is deployed in the namespace and the parameters for the application. Each parameter

is identified with a name which is a reference to the field name of the values object, as explained in the previous paragraph. Secondly, the values of the parameter during different iterations of the benchmark are specified. This is either a constant value or a list of values. At the moment only parameters represented by double values are supported. However, it is possible to specify a prefix and suffix for a parameter.

Experiments

The last section of the benchmark plan specifies which experiments are to be run in during the different iterations. Currently experiments using Locust [42] and Scalar [27] (partially) are supported. In the future other workload generation/performance measurement tools such as JMeter [30] might be added. One can schedule an experiment to be run during every iteration or specify a specific iteration.

Listing 5.3: Benchmark plan configuration file.

```
---
iterations: 4
charts:
  - name: my-app
    chartdir: /path/chartdir
namespaces:
  - name: tenant-g
    chart: my-app
    values:
      - name : workerCPU
        type: incremental
        prefix:
        suffix: m
        list: [1,1,5]
        constant: 1
experiments:
  always:
    - iteration: 0
      type: scalar
      scalar:
        parameters:
          name: experiment-name
          url: service-url
          jobsize:
            start: 100
            end: 4000
            inc: 200
  iterations:

---
```

5.4 Implementation

The k8-resource-optimizer is implemented using the Go programming language. The choice for the Go language follows from the fact that it is the language in which Kubernetes is implemented and the easy manner of performing system tasks (e.g., execution of other programs such as a load testing framework).

5.4.1 Helm as deployment manager

To utilize the potentials of Helm, a Go wrapper was implemented for k8-resource-optimizer. It allows the correct installation and deletion of Helm charts via a simple interface.

5.4.2 Locust as load testing framework

When evaluating the performance of a configuration setting for an application it is important to use a load testing method that can simulate user behavior close to real-world demands. In order to performance test applications within k8-resource-optimizer, a wrapper for Locust [42] is implemented in the Go programming language. Locust is an open source load testing tool. It employs Python code to define user behavior, making it easily extensible. Both HTTP-based and custom clients (e.g., RPC clients) can be implemented. In addition, it has the capability to distribute load tests over multiple machines making it ideal for testing larger cloud-based applications. Various articles by industry leaders, such as [21] by Google cloud platform, demonstrate the combination of Locust and containers in Kubernetes to achieve scalable, distributed load testing.

Chapter 4 describes an application used for the evaluation of the k8-resource-optimizer tool. This multi-tenant application may serve a any number of tenants on different end-points concurrently. The k8-resource-optimizer tool will generate a Locust python class representing a user-type for each SLA class described in Section 5.3.2. Combined the classes from the test plan for Locust. A Locust execution requires the specification of the total amount of users to spawn. By setting a weight attribute to each user class it is possible to control the distribution of user-types. The template to generate a user class is shown in Listing 5.4. Following the execution the Locust wrapper will transform the CSV-formatted results to a JSON format that can be used by the k8-resource-optimizer.

Listing 5.4: Locust user class template.

```
class Taskset#TENANTID(TaskSet):
    @task
    def pushJob(self):
    with self.client.get("/",name="#NAME", catch_response=True)
        as resp:
            if resp.content != "completed all tasks":
                resp.failure("Got wrong response")
```

```
class Tenant#TENANTID(HttpLocust):
    weight = #WEIGHT
    host = "#URL"
    min_wait = #MIN
    max_wait = #MAX
    task_set = Taskset#TENANTID
```

Chapter 6

Evaluation

In this chapter, experiments are conducted to evaluate the correctness and performance of k8-resource-optimizer in Chapter 6. The first experiment validates the correctness of the employed workload generator. The second experiment validates the CPU dependency of the application. The remainder of the experiments validate the SLA-decomposition capabilities of k8-resource-optimizer in various settings.

6.1 Evaluation environment

The experiments in this chapter are conducted on a single-node Kubernetes cluster (version 1.8.0) inside a VM using Minikube [47] (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.

6.2 Experiment 1: Workload generator validation

The k8-resource-optimizer tool (Section 5) uses Locust (Section 5.4.2) as a workload generator to evaluate the performance of an application. Section 2.5.2 showed that Little's law can be used to evaluate the correctness of such a load testing tool. Since the benchmark is a key component for the tool, an experiment is conducted to check its correctness (i.e., it generates the correct amount of concurrent users).

Experiment setup For the experiment two Kubernetes namespaces, large and small, are created on the cluster. In both namespaces an instance of the simple batch processing application is deployed. The CPU quota for the worker pod in the large and small namespace are respectively set to 500 Millicores and 250 Millicores. This refers to both limit and request settings. The size of jobs submitted to the queues by tenants of big and small is 1000 tasks. In three consecutive runs different amounts of each user-type are created.

Results The results generated by Locust are shown in Table 6.1. Little's law states that $\textit{average response time} * \textit{throughput} = \textit{concurrent amount of users}$. This

is shown in the last column. The first column shows the amount of specified users. The results show that concurrency for each user-type generated by Locust is close to the specified amount. As, such it can be concluded that Locust is a valid workload generation tool.

Table 6.1: Experiment 1: Results Little's law based evaluation of workload generation tool. The amount of concurrency is close to the requested amount of users.

# users	Name	# requests	Average RT	Requests/s	Concurrent users
1	large	128	2930	0.34	1.02
1	small	73	5163	0.19	0.98097
2	total	201	3741	0.54	2.02014

# users	Name	# requests	Average RT	Requests/s	Concurrent users
2	large	275	4806	0.41	1.97046
2	small	128	10360	0.19	1.9684
4	total	403	6570	0.61	4.0077

# users	Name	# requests	Average RT	Requests/s	Concurrent users
3	large	279	7127	0.42	2.99334
1	small	124	5340	0.19	1.0146
4	total	403	6577	0.61	4.01197

6.3 Experiment 2: Effect of job size and tenant concurrency on throughput

Chapter 4 describes the application used to evaluate the k8-resource-optimizer tool. The application accepts jobs containing a variable amount of tasks. The maximum size of a job is an integral part of a tenant's SLO specification. The purpose of the following experiment is to validate that a job size influences the performance of an application. In addition, the experiment inspects the impact of multiple concurrent tenants on the throughput.

Experiment setup In the experiment a single Kubernetes namespace is created on the cluster, in which an instance of the application is deployed. In other words, there a single queue and worker are deployed. The CPU request and limit for the worker pod is set to 1000 Millicores. The complete experiment consists out of two sub-experiments. In each sub-experiment multiple iterations of Locust experiments are executed. During the iterations the job size is increased from 100 to 2900 tasks in steps of 100 tasks. In the first sub-experiment, a single tenant submits 50 successive jobs to the application each iteration. In the second sub-experiment, a total of 100 jobs are submitted in each iteration by two concurrent tenants. Separately, the sub-experiments provide insights into the effect of the job size on the performance

of the application. Combined the experiments show how the performance of the application varies in the presence of concurrent tenants.

Results The results of both sub-experiments are shown in Figure 6.1. As expected an increasing job size results in a lower throughput. The Locust experiments with two concurrent tenants achieve an overall higher throughput compared to those of a single tenant. This a consequence of the inner workings of the tested application. By utilizing a pull queue, a worker must periodically poll the queue for tasks. Each poll starts a new thread. When a thread is finished executing a task, it will acknowledge that task and immediately check for a new task. Hence, multiple threads can be started. With a single tenant, the queue is empty between jobs. This results in worker threads to be stopped. This is not the case in the presence of multiple tenants. As a consequence, Figure 6.2 shows that two tenants utilize more CPU compared to a single tenant. It also shows that the worker pod is only capable of using 750 of 1000 Millicores. This indicates that vertical scaling of the worker beyond 750 Millicores probably has no positive effect on its performance.

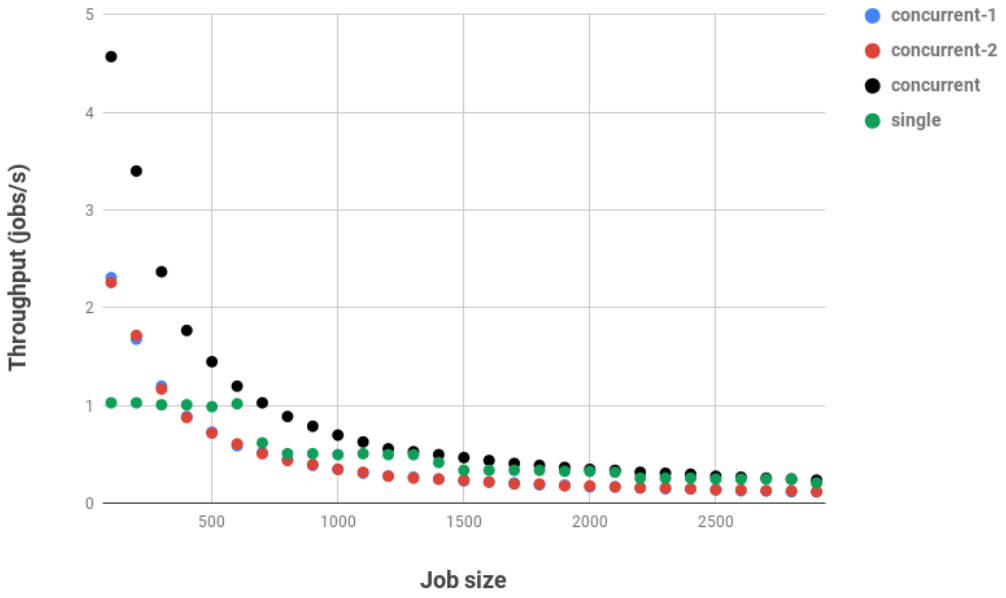
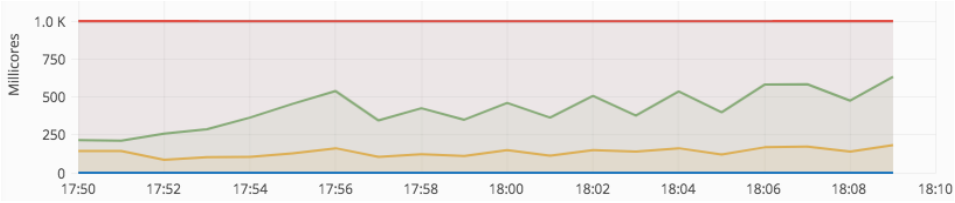
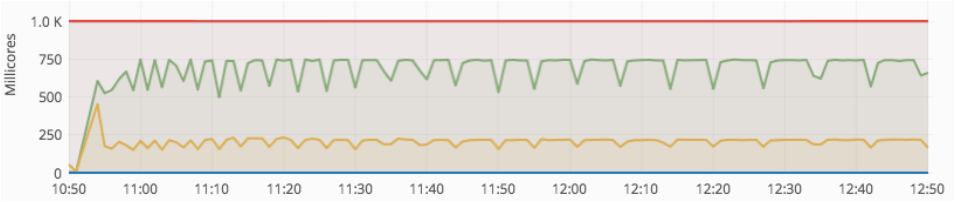


Figure 6.1: Experiment 2: Influence of job size and user concurrency on throughput.



(a) Single tenant.



(b) Two tenants.

Figure 6.2: Experiment 2: Grafana dashboard CPU utilization.

6.4 Experiment 3: Single SLA single parameter

The goal of this experiment is to test the auto-tune capability of k8-resource-optimizer for a single configuration parameter of a single tenant. The parameter under tune is the CPU resource limit and request for the worker component of the application. This parameter will be referred to as `workerCPU` for the remainder of this chapter. The objective is to find a minimum configuration setting that satisfies the SLA.

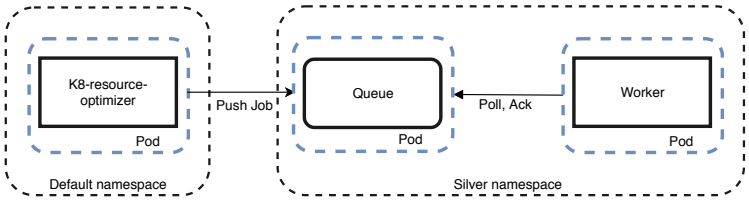


Figure 6.3: Experiment 3: Deployment single SLA single parameter.

Experiment setup The k8-resource-optimizer input configuration for the experiment is shown in Listing 6.1. k8-resource-optimizer will optimize the application for a single tenant of the silver SLA-class. This class requires a minimum throughput of 0.5 jobs per second where each job contains up to 500 tasks. To test whether a configuration meets the SLO, each configuration is load tested with three Locust experiments. One experiment for each of the following job sizes: 300, 400 or 500 tasks. Every experiment performs 10 warm-up request and 50 measured requests. This is to cope with the presence of a cold start. The search space for parameter `workerCPU` is between 200 Millicores and 1000 Millicores. The optimization will perform 4 iterations of *DDS* and *RBS* (Section 3.2.3). Each iteration tests 4 configuration samples. Resulting in 16 configurations to be tested in total. Figure 6.3 illustrates

the deployment of the experiment in the cluster. The tool itself is also deployment inside a pod in the cluster.

Listing 6.1: Experiment 3: Input configuration.

```
---
iterations: 4
samples: 4
charts:
  - name: my-app
    chartdir: /exp/conf/helm/mychart
slas:
  - name : silver
    chart: my-app
    jobsize: 500
    throughput: 0.5
    amount: 1
    parameters:
      - name: workerCPU
        searchspace:
          min: 200
          max: 1000
        prefix:
        suffix: m
strategy: NSPSLA

---
```

Results The results of the experiment produced by k8-resource-optimizer are shown in Table 6.2. These results show that samples picked by the tool during the first iteration provide a wide coverage of the search space. Configuration samples assigned a zero score do not meet the required SLO. The throughput shown is for the largest experiment with job size 500. During each next iteration, a narrower search space is selected by *RBS* as expected. This is visible in Figure 6.5 as the density of samples is the highest between 345 and 400. A CPU resource and limit setting of 360 is the most optimal found by the tool in 4 iterations. Table 6.3 shows the increase of the best score after each iteration. This shows that the second iteration provides a much larger increase compared to the later iterations.

The total execution of the experiment took approximately 2 hours and 10 minutes as shown in Figure 6.4. The majority of the time is spent executing the load tests. Depending on the impact of a 3% (in this case) more cost-efficient setup in a production environment the later iterations could be discarded.

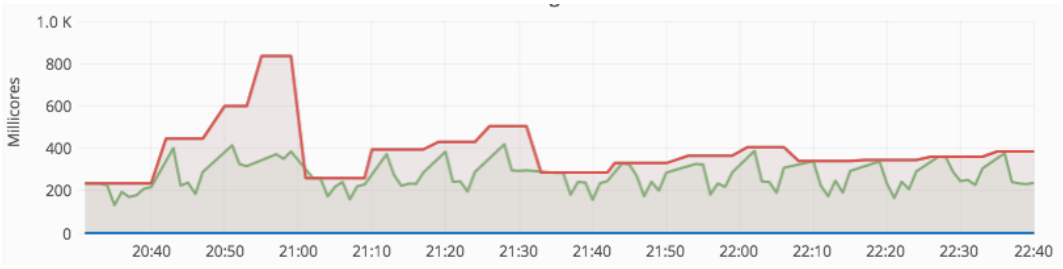


Figure 6.4: Experiment 3: Metrics Grafana dashboard.

Table 6.2: Experiment 3: Results of k8-resource-optimizer on 16 samples in 4 iterations.

ITERATION: 1

workerCPU	**Score**	**Throughput on 500 tasks (Jobs/s)**
235	0	0.34
445	2.25	0.52
600	1.67	0.95
835	1.2	1.01

ITERATION: 2

workerCPU	**Score**	**Throughput on 500 tasks (Jobs/s)**
260	0	0.37
380	2.63	0.51
450	2.22	0.52
560	1.79	0.54

ITERATION: 3

workerCPU	**Score**	**Throughput on 500 tasks (Jobs/s)**
275	0	0.39
345	0	0.48
370	2.7	0.51
395	2.53	0.53

ITERATION: 4

workerCPU	**Score**	**Throughput on 500 tasks (Jobs/s)**
350	0	0.48
360	2.78	0.5
365	2.74	0.5
375	2.67	0.52

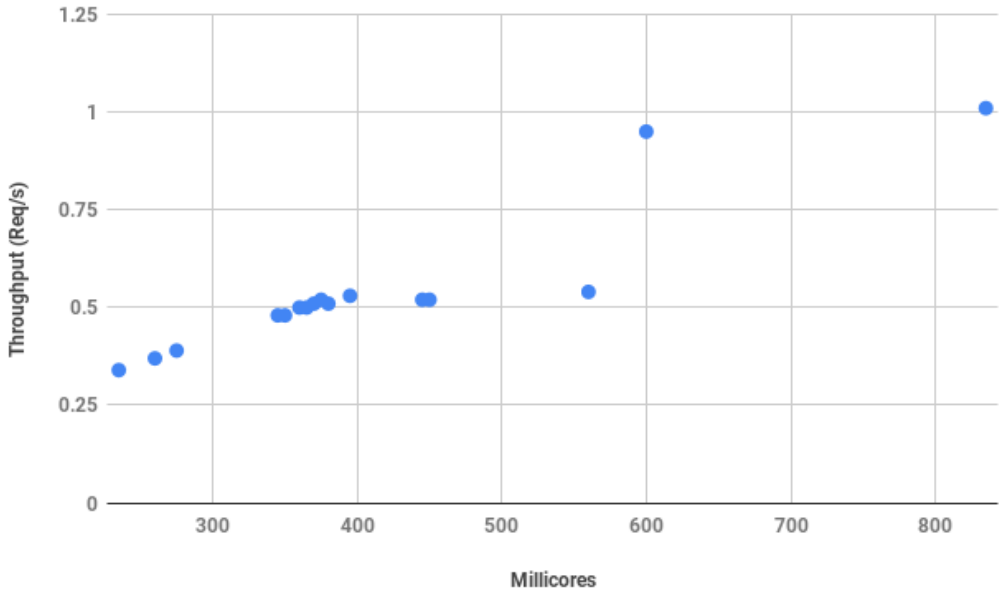


Figure 6.5: Experiment 2: Throughput vs. CPU sample results

Table 6.3: Experiment 3: Utility function score increase per iteration in percentage.

Iteration	**Best score**	**Increase %**
0	2.25	0
1	2.63	17
2	2.7	3
3	2.78	3

6.5 Experiment 4: Set of tenants with homogeneous SLA classes

The following experiment is conducted in order to evaluate the capability of k8-resource-optimizer to perform a SLA-decomposition for multiple tenants of the same SLA class. The parameter under tune is `workerCPU`. The experiment utilizes the shared namespace per SLA class strategy to achieve multi-tenancy within Kubernetes as discussed in Section 3.1.2.

Experiment setup For this experiment a SLA-decomposition is conducted for two silver tenants. The SLO of the silver SLA class is the same as in Experiment 3. In the shared namespace per SLA class strategy, both tenants send requests to the same application instance running in a single Kubernetes namespace. Similar to Experiment 3, three separate Locust experiments are conducted. Since two concurrent tenants are present the Locust experiments perform a total of 100 measured requests. In each experiment, Locust will spawn two users. The SLO must be met for each user separately. The search space for the parameter `workerCPU` is between 200 Millicores and 800 Millicores. The optimization will perform 3 iterations of *DDS* and *RBS* (Section 3.2.3). Each iteration tests 4 configuration samples. Resulting in 12 configurations to be tested in total. The deployment of the namespace per SLA class strategy is the same as illustrated in Figure 6.3.

Results The results of the experiment produced by k8-resource-optimizer are shown in Table 6.5. The best CPU setting for the worker pod found to support two tenants is 560 Millicores. Compared with 360 Millicores for a single tenant in the previous experiment, it shows that the application performance does not scale linearly with the CPU setting. Table 6.4 shows the best score increase in percentage for each iteration. In the last iteration, no better setting is found. If a fourth iteration was executed a backtrack step to the second search space would be performed. The execution of the experiment took approximately 1 hour and 30 minutes. The overall results show that k-bench has the capability to auto-tune for multiple tenants.

Table 6.4: Experiment 4: Namespace per SLA utility function score increase per iteration in percentage.

Iteration	Best score	Increase %
0	1.39	0
1	1.43	3
2	1.38	-3

Table 6.5: Experiment 4: Namespace per SLA results of k8-resource-optimizer on 12 samples in 3 iterations. Throughput is for job size of 500 tasks.

ITERATION: 1

workerCPU	**Score**	**Tenant 1 (Jobs/s)**	**Tenant 2 (Jobs/s)**
335	0	0.28	0.27
545	0	0.49	0.48
575	1.39	0.53	0.52
735	1.09	0.64	0.65

ITERATION: 2

workerCPU	**Score**	**Tenant 1 (Jobs/s)**	**Tenant 2 (Jobs/s)**
560	1.43	0.52	0.51
630	1.27	0.59	0.6
655	1.22	0.61	0.62
680	1.18	0.63	0.64

ITERATION: 3

workerCPU	**Score**	**Tenant 1 (Jobs/s)**	**Tenant 2 (Jobs/s)**
555	0	0.49	0.49
580	1.38	0.51	0.5
595	1.34	0.56	0.55
615	1.3	0.59	0.57

6.6 Experiment 5: Mix of tenants with heterogeneous SLA classes

The goal of the following experiment is the auto-tune capability of k8-resource-optimizer for a mix of tenants with different SLA classes. The parameter under tune is `workerCPU`. The namespace per SLA class strategy is used.

Experiment setup For this experiment, a SLA-decomposition is performed for a single silver and a single bronze tenant. The SLO specifications for the silver SLA class are the same as in Experiment 2. The bronze SLA class requires a minimum throughput of 0.5 jobs per second where each job contains up to 250 tasks.
During the experiment, two instance of the application are deployed in two separate namespaces. The worker CPU parameter for both instances is tuned simultaneously. The search space the `workerCPU` for the silver and bronze namespace are respectively `200 - 800` and `100 - 700`. The optimization will perform 3 iterations of *DDS* and *RBS*. Each iteration tests 4 configurations samples (pair of silver and bronze setting). Resulting in 12 configurations to be tested in total.
To test whether the configurations meet their required SLOs, three Locust experiments are conducted on each sample . In each experiment, two Locust users are concurrently

spawned and executed. One for each type of tenant. During the three Locust experiments the job size is increased in steps of 100 tasks, starting from 300 tasks for silver and 50 tasks for bronze. In total each experiment performs 150 requests.

Results The results of the experiment produced by k8-resource-optimizer are shown in Table 6.6. The best `WorkerCPU` settings found for bronze and silver are respectively 225 Millicores and 360 Millicores. The increase in best score for each SLA is shown Table 6.7. For the silver namespace no better is found within search space `290 - 600` during the second iteration. Therefore, a backtrack to the original namespace is performed during the last iteration. However, the setting with the best score found during the last iteration is within that second search space. This is an example in which backtracking might be performed prematurely. The execution of the experiment took approximately 1 hour and 30 minutes. The overall results indicate that k8-resource-optimizer is capable of tuning multiple namespaces simultaneously.

Table 6.6: Experiment 5: Results of k8-resource-optimizer on 12 samples in 3 iterations for mix of bronze and silver tenants. Throughput is for job size of 250 tasks for bronze and 500 tasks for silver.

Bronze

ITERATION: 1

workerCPU	Score	(Jobs/s)
205	0	0.48
655	1.07	1
480	1.46	1.01
270	2.59	0.53

ITERATION: 2

workerCPU	Score	(Jobs/s)
370	1.89	0.93
420	1.67	1
245	2.86	0.51
325	2.15	0.56

ITERATION: 3

workerCPU	Score	(Jobs/s)
260	2.69	0.52
225	3.11	0.51
265	2.64	0.52
305	2.3	0.52

Silver

ITERATION: 1

workerCPU	Score	(Jobs/s)
290	0	0.35
390	2.05	0.51
600	1.33	0.63
655	1.22	0.95

ITERATION: 2

workerCPU	Score	(Jobs/s)
435	1.84	0.5
325	0	0.39
515	1.55	0.52
440	1.82	0.52

ITERATION: 3

workerCPU	Score	(Jobs/s)
750	1.07	0.96
595	1.34	0.75
360	2.22	0.5
225	0	0.3

Table 6.7: Experiment 5: Utility function score increase per iteration in percentage for bronze and silver.

Bronze

Iteration	Best score	Increase %
0	2.59	0
1	2.86	10
2	3.11	9

Silver

Iteration	Best score	Increase %
0	2.05	0
1	1.84	-10
2	2.22	8

6.7 Experiment 6: Single SLA multiple parameters

The optimization algorithm implemented in k8-resource-optimizer (inspired by the work of [69]) can be used to find a configuration in a multiple dimensional parameter search space as described in 3.2.3. Figure 6.2(b) shows that a worker instance can be virtually scaled up to 750 Millicores. However, the design of the application allows the amount of worker to be horizontally scaled. The goal of the following experiments is to test if k8-resource-optimizer can be used to tune two parameters simultaneously: `workerCPU` and `workerReplicas`.

Experiment setup For this experiment, a SLA-decomposition is performed for three silver tenants. The namespace per SLA class strategy for multi-tenancy is employed. Therefore, all tenants submit requests to the same application instance. The SLOs and the method of evaluation are the same as in Experiment 4, but extended to 3 tenants. The search space for the `workerCPU` parameter is `200 - 750` and `2 - 4` for the `workerReplicas` parameter. To avoid configurations that lead to large CPU allocation, a constraint is added to the sampling method. The constraint is $300 < \texttt{workerCPU} \times \texttt{workerReplicas} < 2000$. This is added to speed up the performance. With a granularity for CPU of 5 Millicores, there are 254 combinations of the two parameters.
The optimization will perform 5 iterations of *DDS* and *RBS*. 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 6.8 and visualized in Figure 6.6. Figure 6.6 shows how for each iteration the search space becomes more narrow as samples are located closer together. The execution of the experiment took approximately 2 hours. In order to validate the results of the k8-resource-optimizer, two extra decompositions were run in which the amount of replicas is fixed. Table 6.9 shows the results for two replicas with search space `400 - 600` and three replicas with search space `300 - 500`. These results validate the finding from the first decomposition. The results indicate that k8-resource-optimizer is capable of performing a SLA-decomposition for two parameters.

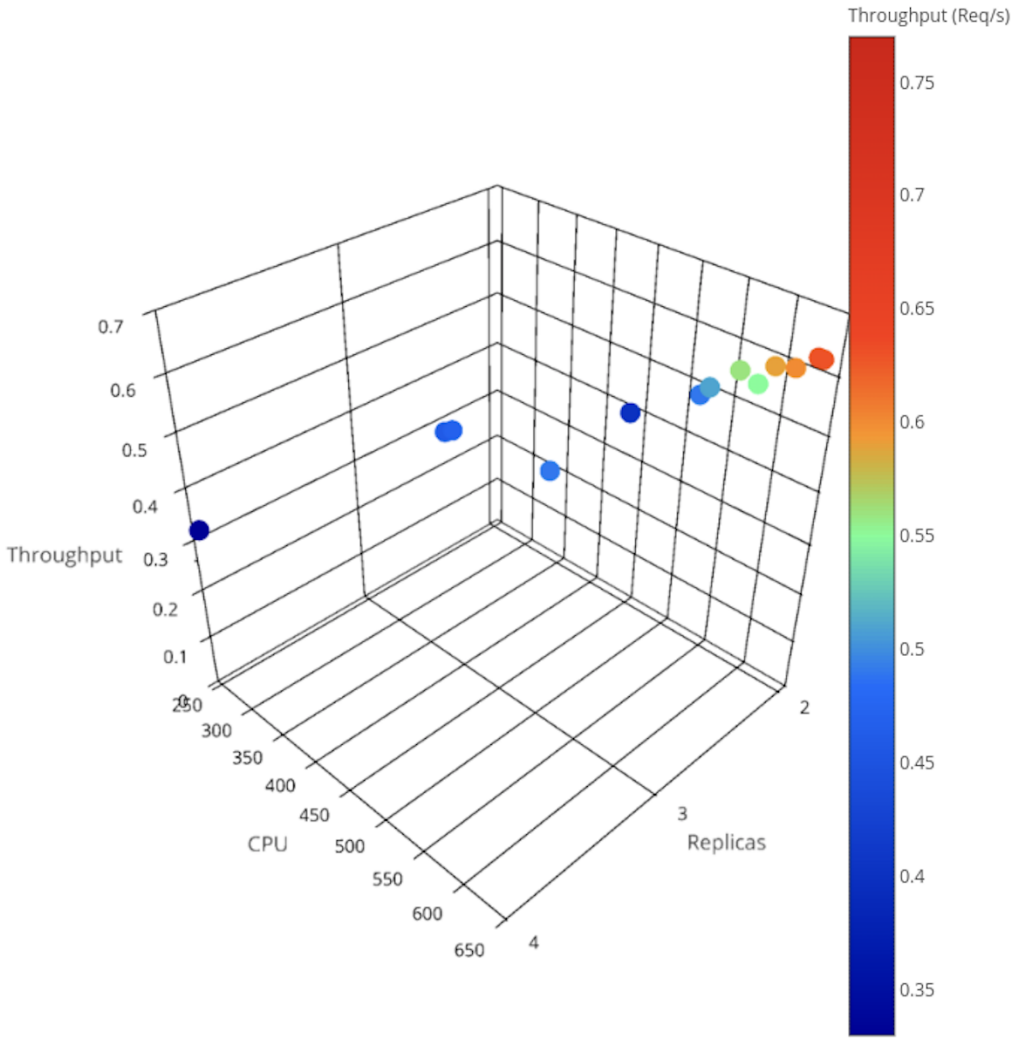


Figure 6.6: Experiment 6: Throughput vs. CPU and replica configuration.

Table 6.8: Experiment 6: Results of k8-resource-optimizer on 15 samples in 5 iterations Throughput is for job size of 500 tasks.

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

Table 6.9: Experiment 6: Validation decompositions with fixed amount of worker replicas.

ITERATION: 1

workerReplicas	**workerCPU**	**Score**	**Throughput (Jobs/s)**
2	410	0	0.39
2	455	0	0.43
2	505	0	0.48
2	575	2.04	0.54

ITERATION: 2

workerReplicas	**workerCPU**	**Score**	**Throughput (Jobs/s)**
2	560	2.07	0.54
2	570	2.05	0.55
2	535	2.12	0.51
2	520	0	0.48

ITERATION: 1

workerReplicas	**workerCPU**	**Score**	**Throughput (Jobs/s)**
3	310	0	0.36
3	355	0	0.42
3	405	0	0.49
3	475	2.05	0.58

ITERATION: 2

workerReplicas	**workerCPU**	**Score**	**Throughput (Jobs/s)**
3	460	2.09	0.57
3	470	2.06	0.58
3	435	2.15	0.52
3	420	2.19	0.51

6.8 Summary of experiment results

In this chapter, an evaluation was presented of the various capabilities of k8-resource-optimizer. Experiment 1 validated the correctness of the employed workload generator. In the remaining experiments, k8-resource-optimizer was evaluated in the following scenarios: single parameter tuning of a single tenant, a mix of homogeneous tenants, a mix of heterogeneous tenants and multiple parameter tuning. In all scenarios, k8-resource-optimizer was capable to produce an acceptable decomposition within the given constraints. Overall, the results are promising for the applicability of auto-tuning for SLA-decomposition in a multi-tenant Kubernetes environment.

Due to timing constraints, k8-resource-optimizer was not evaluated in the context of tuning multiple components. Results similar to those of multiple parameters are expected in this scenario. This proposed as future work. In addition, future work could include exploration to eliminate unnecessary preliminary backtracking of BestConfig's optimization algorithm and to reduce the runtime of Locust experiments.

Chapter 7

Conclusion

7.1 Revisiting the problem statement

The on-demand available compute resources offered by cloud providers combined with the fine-grained resource allocation of container technology allows SaaS providers to minimize their operating costs. In addition, SaaS providers often employ a multi-tenant architecture in which resources are shared among multiple tenants. The work of [63] shows how the resource allocation policies of Kubernetes (e.g., CPU request and limits) can be used to offer a different quality of service to different tenants. SaaS providers wish to restrict the capacity of their infrastructure to the amount required to meet the SLOs agreed upon with their customers in signed SLAs. The task of mapping SLOs to a cost-efficient resource allocation is a problem, referred to as SLA-decomposition. SLA-decomposition is difficult especially for multi-tenant SaaS applications with different performance SLOs for different classes of tenants.

Existing approaches for SLA-decomposition of multi-tenant SaaS applications require extensive domain expertise and struggle with the conceptual gap between the model of the application and the true performance characteristics of the application. Moreover, the required amount of resources for different sizes of the demanded workload volume is inherently a non-linear resource scaling problem. A model accurate for five tenants could be completely wrong for ten tenants.

7.2 Summary of contributions

This thesis has proposed an approach that does not require any domain expertise and that can be applied for different workload sizes in order to estimate the required set of resources for any desired combinations of tenants from various SLA classes. In summary, the thesis makes the following contributions.

An extensive analysis of the problems surrounding the SLA-decomposition task is presented. The increasing complexity of software systems and their deployment environments results in complex performance characteristics making it impossible for

system administrators to correctly estimate required resources. Both related research and conducted experiments show that most systems do not scale linearly and the use of simple scaling policies are thus not economically beneficial.

A survey of existing state-of-art approaches related to the problem is provided. Auto-tuning approaches based on continuous experimentation are selected as the most suitable over other model-based approaches. The black box approach of these methods offers more adaptivity for different types of applications and does not demand extensive knowledge of the application. By evaluation through continuous experimentation in the production environment, uncertainty on the quality of the resulting configuration is eliminated.

A specific auto-tuning approach, BestConfig [69], is applied and evaluated in the context of a simple batch processing application. Adapted for the SLA-decomposition problem it has shown to be capable of finding a cost-efficient resource allocation even in the presence of multiple search parameters.

A prototype has been developed as an easy-to-use tool, named k8-resource optimizer, that integrates well with Kubernetes, the de-facto standard for container orchestration platforms, and a popular package manager for Kubernetes named Helm. Its operational effort for system administrators is limited to the declarative specification of SLOs, resource search space parameters and workload profiles.

7.3 Lessons learned

The experimental findings have shown that k8-resource-optimizer is capable of producing optimal or near-optimal SLA-decompositions for a Kubernetes application in deployment scenarios with one or more SLOs for one or more tenant organizations. Whilst further evaluation with different applications is needed, k8-resource-optimizer seems to be an ideal match for contemporary DevOps applications. K8-resource-optimizer can be ideally performed in the production environment when a new version of the application is being dark launched, canary or A/B tested. In addition, the decompositions produced by k8-resource-optimizer could be used in a capacity management policy of a SLA-manager responsible for reactive provisioning and deprovisioning containers whenever a tenant submits a new batch or whenever a batch is finished.

7.4 Limitations of k8-resource-optimizer

The chosen solutions present in the design of k8-resource-optimizer impose several limitations on the capability of the tool. A limitation is defined as an inherent limitation that cannot be envisioned to be solved in future work. The tool is known to have the following limitations.

While capable of estimating the required resources of a periodic job, the approach is not capable of dealing with SLOs containing deadlines. This task requires the presence of scheduling component such as the online packing algorithm proposed by [31] or the task re-prioritization service presented in [65].

The k8-resource-optimizer's resource regulation capabilities are restricted to resource policies offered by Kubernetes. Currently, Kubernetes resource management concepts to do not support network or disk I/O bandwidth. As a result, k8-resource-optimizer only works out of the box for CPU and memory allocations of the state-less web-tier.

The regulation of network bandwidth for Kubernetes applications requires an external rate limiter at the boundary of the network. The rate limiter limits the presence of network congestion thereby allowing the assumption that the technical specified bandwidth of network adapter is always available. Network traffic control plugins are available for the Mesos orchestration platform [1]. Parameters of these plugins can be auto-tuned.

In contrast to disk space, disk I/O bandwidth is not controllable per container in Kubernetes. This problem only aggravates in cloud-based environments due to the distributed nature of the storage system underlying persistent services. However, for multi-tenant SaaS this does not matter. For reasons of simplicity, this argument is defended for traditional web applications where a typical SaaS application is stored as a stateless web tier backed up by a multi-tenant database. In this scenario, the database tier requires a separate SLO management method. Currently, a request scheduler is still the best approach.

7.5 Future work

A first extension is the further evaluation of the tool with different applications. These applications could require tuning of different Kubernetes configuration parameters such as memory request/limits or the tuning of both CPU and memory allocation simultaneously. Other evaluation could include experimentation where multiple SLO types must be met simultaneously (e.g., throughput and job completion deadlines).

A second extension could include optimizations to reduce the runtime of the tool. The majority of the runtime is spent during the load tests evaluating the performance of a configuration. The runtime of a single load test is decided by the performance of the configuration and the required number of requests to be completed. Weak performing configuration will thus increase the total runtime of the tool. A possible extension could include the specification of a runtime for a single load test instead of the number of requests. Resulting in a fixed specifiable total runtime.

[1] https://github.com/apache/mesos/blob/1.5.x/docs/isolators/cgroups-net-cls.md

A third extension could include the implementation of an SLA-decomposition technique similar to the Bayesian optimization approach used by CherryPick [2] discussed in Section 3.2.4. A comparison of the performance of both approaches could be conducted.

Appendices

Appendix A

Short tutorial

This short tutorial describes how to build container images for the simple batch processing application and k8-resource-optimizer. Next it describes how to deploy the tool and perform a SLA-decomposition for the simple batch processing application.

A.1 Prerequisites

The tutorial assumes that the following tools are installed on your development machine. The provided scripts are written in for a Bash compatible shell.

- Apache Maven - https://maven.apache.org
- Docker - https://www.docker.com
- Minikube - https://github.com/kubernetes/minikube
- Helm - https://github.com/helm/helm
- Golang - https://golang.org

The tutorial also requires a Docker Hub account with (empty) repositories named: k8-resource-optimizer, consumer and demo.

A.2 Simple batch processing application

The source code of the components of the simple batch application can be found in the directory named application. A component can be build and pushed to your dedicated Docker Hub repository by running the provided bash script named `build.sh` located in the application directory inside the component's directory. Use the variable `DOCKERACCOUNT` part of the script to specify your Docker Hub account. The script expects the component name as a command-line argument. This is either `consumer` or `demo` (queue). The script uses Maven to build the application. In the `pom.xml`, a docker-maven plugin by Spotify is used for the creation of the docker container image.

```
cd application/consumer
./../build.sh consumer
```

The image names in the templates of the Helm chart belonging to the application have to be changed to those of your personal Docker Hub account. The helm chart is located in the `conf` directory. The provided `changeAccountName.sh` bash script can do this for you. This script keeps the original files with a `.bak` extension.

```
./changeAccountName.sh YOUR_ACCOUNT
```

A.3 k8-resource-optimizer

The k8-resource-optimizer can be build and pushed to your dedicated Docker Hub repository by running the provided bash script name `build.sh` inside the main directory. This script first builds a Go binary for a Linux environment and stores it in the bin directory. It proceeds with building a Docker container capable of executing the tool. The dockerfile assures that the binary and configuration files are part of the Docker image. The script requires the specification of your Docker Hub account as the variable `DOCKERACCOUNT`.

```
./build.sh
```

The k8-resource-optimizer can be deployed via the provided `tool-deployment.yaml` file in the main directory. The image specification in this file should be altered to your personal Docker account. The `changeAccountName.sh` script mentioned in the previous sections takes care of this. The following command actually deploys the tool within your Kubernetes cluster.

```
kubectl create -f tool-deployment.yaml
```

When the pod containing the tool is deployed, you are able to start an interactive shell on the pod . The `connect.sh` script does this automatically.

```
./connect.sh
```

Once connected with the pod, you are able to run the tool on an given decomposition configuration. An example is given in `conf/decompose.yaml`. The configuration is passed as a command-line argument. Once the tool is finished a report is created under the name `test_report.txt`.

```
cd exp
./k8-resource-optimizer conf/sladecompose.yaml
```

Since it is cumbersome task to rebuild the container image of the tool for every change during development, a script is provided to build and copy the binary of the tool to a running pod (containing an older version of the tool) from your development machine. The `buildandcp.sh` script can be found in the main directory of the project.

Bibliography

[1] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle. Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on Services Computing*, 11(2):430–447, 2018.

[2] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, volume 2, pages 4–2, 2017.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[4] Baeldung. Guide to completablefuture in java spring, 2018. http://www.baeldung.com/java-completablefuture. [Online; Accessed July 16, 2018].

[5] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu. Delivering elastic containerized cloud applications to enable devops. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 65–75. IEEE Press, 2017.

[6] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, Apr. 2016.

[7] Canonical Ltd. Lxc: linux container, 2018. https://linuxcontainers.org/lxc/. [Online; Accessed June 25, 2018].

[8] T. Chen, Y. Chen, Q. Guo, Z.-H. Zhou, L. Li, and Z. Xu. Effective and efficient microprocessor design space exploration using unlabeled design configurations. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 5(1):20, 2013.

[9] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai. Sla decomposition: Translating service level objectives to system level thresholds. In *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*, pages 3–3. IEEE, 2007.

[10] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *ACM SIGPLAN Notices*, volume 50, pages 191–206. ACM, 2015.

[11] David Oppenheimer. Multitenancy deep dive, 2018. https://schd.ws/hosted_files/kccncna17/a9/kubecon-multitenancy.pdf. [Online; Accessed June 27, 2018].

[12] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[13] T. Dillon, C. Wu, and E. Chang. Cloud computing: issues and challenges. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 27–33. Ieee, 2010.

[14] Docker. Docker, 2018. https://www.docker.com/.[Online; Accessed June 25, 2018].

[15] Docker Inc. Comparing containers and virtual machines, 2018. https://www.docker.com/what-container#/virtual_machines. [Online; Accessed June 27, 2018].

[16] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614. IEEE, 2014.

[17] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[18] T. O. Foundation. Kata containers: The speed of containers, the security of vms, 2018. https://katacontainers.io/.[Online; Accessed June 4, 2018].

[19] J. Frazelle. Hard multi-tenancy in kubernetes, 2018. https://blog.jessfraz.com/post/hard-multi-tenancy-in-kubernetes/.[Online; Accessed June 4, 2018].

[20] J. Frazelle. Kubernetes - multi-tenancy design scratch space, 2018. https://docs.google.com/document/d/1PjlsBmZw6Jb3XZeVyZ0781m6PV7-nSUvQrwObkvz7jg.[Online; Accessed June 4, 2018].

[21] Google Cloud. Distributed load testing using kubernetes, 2018. https://cloud.google.com/solutions/distributed-load-testing-using-kubernetes. [Online; Accessed July 16, 2018].

[22] Google Cloud. Google app engine: Task queue overview, 2018. https://cloud.google.com/appengine/docs/standard/java/taskqueue/. [Online; Accessed July 16, 2018].

[23] N. J. Gunther. How to quantify scalability - the universal scalability law (usl), 2018. http://www.perfdynamics.com/Manifesto/USLscalability.html.[Online; Accessed May 18, 2018].

[24] Helm. Helm - the kubernetes package manager, 2018. https://helm.sh/ /.[Online; Accessed May 17, 2018].

[25] J. Hermanns and A. Steffens. The current state of infrastructure as code and how it changes the software development process. *Full-scale Software Engineering*, 19, 2015.

[26] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: a self-tuning system for big data analytics. In *Cidr*, volume 11, pages 261–272, 2011.

[27] T. Heyman, D. Preuveneers, and W. Joosen. Scalability analysis of the openam access control system with the universal scalability law. In *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on*, pages 505–512. IEEE, 2014.

[28] A. Hussain. Little's law- an insight on the relation between latency and throughput, 2018. http://blog.flux7.com/blogs/benchmarks/littles-law.[Online; Accessed May 18, 2018].

[29] A. Jacobs. Haalbaarheidsstudie van container orchestratie voor performantie-isolatie in multi-tenant saas-applicaties. unpublished master thesis, 2017.

[30] JMeter. Apache jmeter, 2018. https://jmeter.apache.org/.[Online; Accessed May 17, 2018].

[31] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, pages 117–134, 2016.

[32] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

[33] A. Khan. Key characteristics of a container orchestration platform to enable a modern application. *IEEE Cloud Computing*, (5):42–48, 2017.

[34] T. L. F. Kubernetes. Kubernetes deployments, 2018. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.[Online; Accessed May 18, 2018].

[35] T. L. F. Kubernetes. Kubernetes namespaces, 2018. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.[Online; Accessed May 18, 2018].

[36] T. L. F. Kubernetes. Kubernetes pods, 2018. https://kubernetes.io/docs/concepts/workloads/pods/pod/.[Online; Accessed May 18, 2018].

[37] T. L. F. Kubernetes. Managing compute resources for containers, 2018. https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/.[Online; Accessed May 18, 2018].

[38] T. L. F. Kubernetes. Services, 2018. https://kubernetes.io/docs/concepts/services-networking/service/.[Online; Accessed May 18, 2018].

[39] T. L. F. Kubernetes. What is kubernetes?, 2018. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.[Online; Accessed May 18, 2018].

[40] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan. Towards multi-tenant performance slos. *IEEE Transactions on Knowledge and Data Engineering*, 26(6):1447–1463, 2014.

[41] J. D. Little and S. C. Graves. Little's law. In *Building intuition*, pages 81–100. Springer, 2008.

[42] Locust.io. Locust: An open source load testing tool, 2018. https://locust.io/. [Online; Accessed July 16, 2018].

[43] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 workshop on Experimental computer science*, page 6. ACM, 2007.

[44] P. Mell and T. Grance. The nist definition of cloud computing. *Association for Computing Machinery. Communications of the ACM*, 53(6), June 2010.

[45] P. Mell, T. Grance, et al. The nist definition of cloud computing. 2011.

[46] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[47] Minikube developers. Minikube: running kubernetes cluster locally, 2018. https://github.com/kubernetes/minikube. [Online; Accessed July 16, 2018].

[48] B. Mozafari, C. Curino, and S. Madden. Dbseer: Resource and performance prediction for building a next generation database cloud. In *CIDR*, 2013.

[49] M. page Michael Kerrisk Eric W. Biederman. Linux programmer's manual - namespaces, 2018. http://man7.org/linux/man-pages/man7/namespaces.7.html.[Online; Accessed May 18, 2018].

[50] C. Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[51] Redhat. Introduction to control groups (cgroups), 2018. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01.[Online; Accessed May 18, 2018].

[52] B. P. Rimal, E. Choi, and I. Lumb. A taxonomy and survey of cloud computing systems. In *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*, pages 44–51. Ieee, 2009.

[53] B. Schwarz. Practical scalability analysis with the universal scalability law, 2015.

[54] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[55] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 1–13. ACM, November 2016.

[56] The Linux Foundation. Assigning pods to nodes, 2018. https://kubernetes.io/docs/concepts/configuration/assign-pod-node/. [Online; Accessed June 27, 2018].

[57] The Linux Foundation. Configure quality of service for pods, 2018. https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/. [Online; Accessed June 27, 2018].

[58] The Linux Foundation. Kube api server, 2018. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/. [Online; Accessed June 27, 2018].

[59] The Linux Foundation. Kube controller manager, 2018. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/. [Online; Accessed June 27, 2018].

[60] The Linux Foundation. Kube-proxy, 2018. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/. [Online; Accessed June 27, 2018].

[61] The Linux Foundation. Kube scheduler, 2018. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/. [Online; Accessed June 27, 2018].

[62] The Linux Foundation. Kubelet, 2018. https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/. [Online; Accessed June 27, 2018].

[63] E. Truyen, D. Van Landuyt, V. Reniers, A. Rafique, B. Lagaisse, and W. Joosen. Towards a container-based architecture for multi-tenant saas applications. In *Proceedings of the 15th International Workshop on adaptive and reflective middleware*, ARM 2016, pages 1–6. ACM, December 2016.

[64] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[65] S. Walraven, W. D. Borger, B. Vanbrabant, B. Lagaisse, D. V. Landuyt, and W. Joosen. Adaptive Performance Isolation Middleware for Multi-tenant SaaS. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, 2015.

[66] S. Walraven, D. Landuyt, A. Rafique, B. Lagaisse, and W. Joosen. Paashopper: Policy-driven middleware for multi-paas environments. *Journal of Internet Services and Applications*, 6(1):1–14, December 2015.

[67] S. Walraven, E. Truyen, and W. Joosen. A middleware layer for flexible and cost-efficient multi-tenant applications. volume 7049, pages 370–389, 2011.

[68] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240. IEEE, 2013.

[69] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338–350. ACM, 2017.

Fiche masterproef

Student: Matthijs Kaminski

Titel: Multi-tenant performance SLOs for container orchestrated batch workloads

UDC: 681.3

Korte inhoud:

SaaS Providers face the constant challenge to maximize the use of their rented infrastructure in order to offer their software service at the most competitive price. Often a multi-tenant architecture is employed in order to maximize resource sharing between tenants. This requires techniques to differentiate the service between different classes of tenants. The quality of the service provided is specified in an Service Level Agreement (SLA) between tenant and provider. Recent work states that the resource management concepts of container technology such as Docker and container orchestration platforms such as Kubernetes can be used to achieve multi-tenancy with quality of service differentiation while offering fine-grained resource allocation. In order to use these container orchestration concepts, SaaS providers need to translate Service Level Objectives (SLOs), which are part of the SLA, to low-level resource allocations. This difficult task typically requires extensive domain expertise.Existing state-of-the-art approaches to this problem rely on a model of the application and therefore again requires extensive domain expertise in order to create an accurate model of the application. In opposition to this approach, we propose a generic approach that does not require any model of the application or another type of domain expertise. Instead, we determine mappings between SLOs and resource allocations by adapting a performance tuning technique that relies on continuous experimentation of the application in the production environment. We believe such continuous experimentation is feasible in contemporary DevOps environments where a new version of the application is tested in the production environment before exposing it to clients.In this master thesis this approach is implemented as part of an automated tool, k8-resource-optimizer, capable of translating SLOs for a given application and workload to Kubernetes resource concepts. Experimental evaluation of k8-resource-optimizer with a simple batch processing application has shown that this approach is practically able to produce an optimal or near-optimal resource allocation in various deployment scenarios with one or more SLOs for one or more tenant organizations. Therefore, k8-resource-optimizer shows potential as a useful DevOps tool.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Gedistribueerde systemen

Promotoren: Prof. dr. ir. Wouter Joosen
Dr. Eddy Truyen

Assessoren: Prof. dr. Adalberto Simeone
Emad Heydari Beni

Begeleider: Emad Heydari Beni