TU WIEN Informatics

Optimizing Container Elasticity for Microservices in Hybrid Clouds

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Alexander Wais, BSc
Matrikelnummer 01127899

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dr.-Ing. Stefan Schulte

Wien, 15. April 2021

Alexander Wais Stefan Schulte

Informatics

Optimizing Container Elasticity for Microservices in Hybrid Clouds

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Alexander Wais, BSc
Registration Number 01127899

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dr.-Ing. Stefan Schulte

Vienna, 15th April, 2021

Alexander Wais | Stefan Schulte

Erklärung zur Verfassung der Arbeit

Alexander Wais, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. April 2021

Alexander Wais

Acknowledgements

This thesis represents the finale of my master studies at TU Wien, therefore I would like to thank the ones who made this accomplishment possible.

First of all, I want to sincerely thank my supervisor, Stefan Schulte, for his permanent guidance, invaluable advice, and for always being incredibly responsive, all of which contributed significantly to the success of this work during a year of unusual challenges.

I also wish to express my deepest gratitude to my parents and family, who made my studies possible and encouraged my interest in everything technical since I was a child. Last but not least, a big thanks goes to Tamara for her encouragement and support during the last weeks and months. Thank you.

Kurzfassung

Containerisierte Microservices stellen den aktuellen Stand der Technik für das Deployment von komplexen Software-Applikationen in der Cloud dar. Durch die Nutzung von Platform-as-a-Service- oder Container-as-a-Service-Angeboten (PaaS bzw. CaaS) können die für die Bereitstellung von Containern benötigten Rechenkapazitäten rasch entsprechend den Bedürfnissen einer Organisation skaliert werden. Bestehende Ansätze setzen zur Optimierung der Elastizität solcher Systeme fortgeschrittene Skalierungsmethoden ein, um Container und/oder virtuelle Maschinen (VMs) vertikal oder horizontal zu skalieren – dabei sind Kosteneffizienz und Dienstgüteanforderungen meist die primären Treiber. Das Lösen derartiger Optimierungsprobleme erweist sich jedoch oft als schwierig, da sich je nach Optimierungsansatz zu lange, nicht praktikable Laufzeiten ergeben können. Des Weiteren sind Aspekte wie die Berücksichtigung von Netzwerklatenzen zwischen kommunizierenden Microservices, insbesondere zusammen mit hybriden Cloud-Deployments und in Kombination mit fortgeschrittenen Skalierungsmethoden, Gegenstand offener Forschungsfragen.

In dieser Arbeit greifen wir diese Aspekte in einem vereinheitlichten Ansatz für ein CaaS-Modell auf, welches (i) die elastische Skalierung von Containern und VMs, (ii) mehrere Datencenter in hybriden Clouds, und (iii) Co-Location von Containern im Hinblick auf deren Interaktionsaffinität und Netzwerklatenzen berücksichtigt. Das zugrundeliegende Optimierungsmodell wird mittels ganzzahliger linearer Optimierung (ILP) formuliert. Wir implementieren *Cooper*, eine auf der MAPE-K-Rückkopplungsschleife basierende Middleware, die zwei verschiedene Optimierungsansätze ermöglicht: Eine exakte Optimierung basierend auf dem ILP-Modell mittels IBM CPLEX, sowie einen heuristischen Ansatz, der einen genetischen Algorithmus verwendet um kürzere Laufzeiten für große Probleminstanzen bei gleichzeitig guten Ergebnissen zu erzielen.

Abschließend evaluieren wird die vorgestellten Optimierungsansätze mittels simulierter Szenarien und stellen einen Vergleich von Laufzeiten, Kostenersparnissen und Latenzreduktionen an. Als Vergleichsbasis dient dabei ein einfacher heuristischer Ansatz, welcher einen First-Fit-Algorithmus einsetzt. Die Ergebnisse für den genetischen Algorithmus weisen eine Latenzreduktion von 30 % und Kostenersparnisse von 8 % in einem mittelgroßen Szenario aus. Die ILP-basierte Optimierung erweist sich aufgrund des übermäßigen Ressourcenbedarfs als nicht praktikabel, mit Ausnahme von sehr kleinen Probleminstanzen. Das Weglassen des Co-Location-Aspekts in der ILP-Optimierung ermöglicht jedoch die Anwendung in größeren Szenarien.

Abstract

Containerized microservices have become the state-of-the-art deployment style for large software applications in the cloud. Using the benefits of Platform as a Service (PaaS) or Container as a Service (CaaS) offerings, the underlying computational resources required to provision containers can be quickly scaled in an automated manner according to an organization's needs. Several approaches exist to improve elasticity of such systems by applying smart scaling decisions to resources, such as vertical or horizontal scaling of containers, Virtual Machines (VMs), or both, with the primary goals being optimization for cost efficiency and Quality of Service. However, solving such optimization problems is a tricky task in existing approaches, as runtimes can become infeasible for large systems depending on the applied optimization technique. Further aspects, such as considering request latencies between communication-affine microservices in particular when deployed in a hybrid cloud environment, combined with advanced elastic scaling methods, are subject to open research challenges.

In this thesis, we propose an approach to address these challenges in a unified CaaS system model which considers (i) elastic scaling of containers and VMs, (ii) multiple data centers in hybrid clouds, and (iii) co-location of containers based on interaction affinity and network latency. The underlying optimization problem is formulated as an Integer Linear Programming (ILP) model. We implement *Cooper*, a middleware based on the well-known MAPE-K feedback loop, which provides two different optimization techniques: An exact approach based on the ILP model using IBM's CPLEX optimizer, and a heuristic approach based on a Genetic Algorithm that aims for better runtime performance of large problem instances while still achieving good results.

Finally, we evaluate the proposed approaches based on simulated scenarios, and provide a comparison with regard to runtime, cost savings and latency reduction. As a baseline, we employ a simple heuristic approach based on a First-Fit algorithm. Results show that the Genetic Algorithm achieves a latency reduction of 30 % and cost savings of 8 % in a medium-sized scenario. The ILP-based approach shows to be not feasible due to the performance limitations caused by its excessive resource demands, except for very small problem instances. However, dropping the latency-aware co-location aspect in the ILP optimization enables it to accomplish larger scenarios.

Contents

CHAPTER 1

Introduction

Cloud computing provides computational resources in extent to the users' requirements, as well as the economic benefit that resources only need to be leased, and thus paid, when needed. Different layers of cloud service models are available: The first and main pillar of cloud computing is Infrastructure as a Service (IaaS), which provides (mostly virtual) resources such as Virtual Machines (VMs) that can be scaled by adding and removing resources on user request. Platform as a Service (PaaS) builds upon IaaS and provides a higher level of abstraction, allowing users (i.e., software developers) to focus on application development with regard to functionality, while the cloud provider manages the underlying infrastructure. For a PaaS to operate in a sustainable and economically efficient way, resources need to be scaled up and down in an automated manner that avoids under-provisioning (i.e., causing performance degradation) and over-provisioning (often causing avoidable cost increases) of leased resources [15].

Elasticity can be characterized as the ability of such services to achieve scalability in a smart way, by (i) maintaining user requirements such as availability and fault tolerance; (ii) fulfilling provisioning, allocation and auto-scaling of resources while adapting to workload and capacity utilization; and (iii) applying these reconfigurations in a timely manner [3, 13, 43].

With the increased adoption of modern software engineering principles, current methodologies and technologies such as *microservices* and *containers* play an important role in today's scalable architectures and accommodation of developers' needs in line with DevOps practices [50]. Microservices are an approach to split up a system's capabilities into distinguished applications that can be independently developed and deployed, with containers being among the most common deployment approaches [41]. Container technologies such as Docker provide a virtual execution environment for applications similar to VMs, but are more resource-efficient by sharing common resources such as the operating system and its libraries. Similar to PaaS, the Container as a Service (CaaS) model provides an abstraction layer for the provisioning of containers [15].

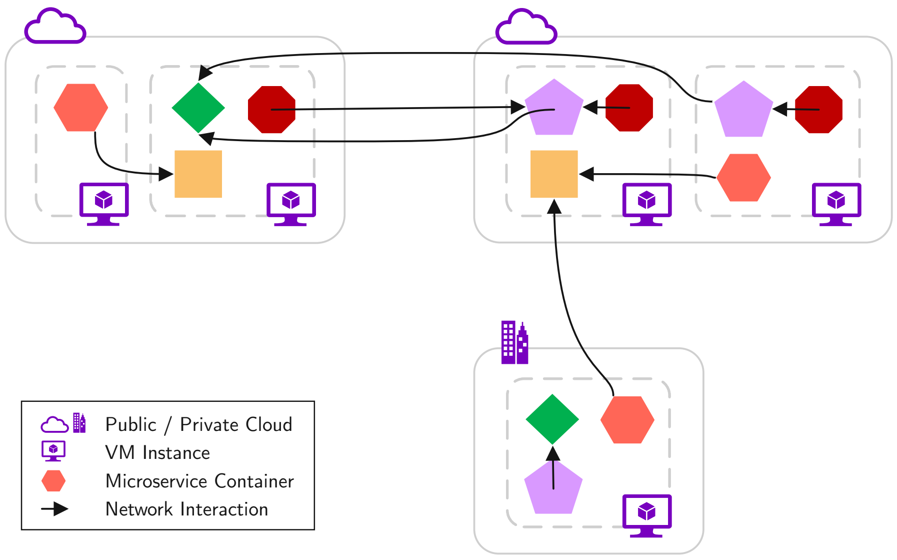


Figure 1.1: Motivational scenario

Basic concepts of elasticity have been around in PaaS and are well understood, such as scaling resources either horizontally (i.e., spawning or terminating VM instances) or vertically (increasing or decreasing the computational resources of VM instances), based on thresholds for CPU or memory utilization. However, more sophisticated approaches to optimize resource configurations, especially with regard to the placement of containers on underlying resources, are still subject to ongoing research [15]. With containers, an additional allocation dimension has to be taken into account: While common auto-scaling has been practiced for VMs before, containerization adds the aspect of scaling the container instances and distributing them over VMs. This adds a wide area of research potential to the topic, e.g., regarding decisions to scale containers vs. VMs, applying vertical or horizontal scaling, and where to place new containers (i.e., on which VM).

Further aspects of elasticity and resource allocation come into play with regard to microservices. Since individual services often interact with each other to perform system-wide transactions, they need to exchange messages over the network at runtime. Hence, the network distance in terms of network latency between microservice containers contributes to the system's overall performance and could be optimized for reduced request latencies. This can be achieved by placing containers of affine microservices more closely together (*co-location*), e.g., on the same VM instance, or within the same data center in case multiple clouds are utilized. These *hybrid clouds* may comprise both public data centers of different cloud providers as well as on-premise infrastructure (*private clouds*).

A motivational scenario is shown in Figure 1.1, depicting several containers belonging to different microservices. Microservices are distinguished by shape and color, and communicate with each other over the network (interaction between affine services is represented by arrow lines). The containers are distributed across VM instances, which for their part are provisioned in multiple data centers (i.e., public and private clouds). Looking at the arrangement of resources, we can state several questions that carry motivation for the topic of this thesis:

- How many containers need to be deployed for each microservice at a given point in time in order to handle incurred workload?
- Which VMs should be leased in each data center, considering that overall cost should be kept low?
- On which VM should each container be placed in order to utilize leased resources efficiently?
- How to place containers in order to minimize the latencies of network requests between them?

1.1 Aim of the Work

The main goal of this thesis is to extend the state-of-the-art in the field of container elasticity. The focus lies on gaining a better understanding of optimized container elasticity with regard to hybrid clouds, and in particular with respect to co-location of containers in a microservices setting. Therefore, an approach should be developed which constitutes the foundation for a CaaS middleware that schedules microservice applications on (hybrid) cloud infrastructures. At its core, a suitable optimization approach is to be modeled and implemented. Finally, as an important outcome of the work, a quantitative evaluation shall reveal the potential of our proposed approach.

Regarding the optimization scope and user requirements, the following aspects should be explored.

Hybrid cloud exploitation. The proposed solution should have a notion of different cloud data centers which provide the underlying computational infrastructure. Furthermore, in-house infrastructure of an organization (private clouds) should be considered in order to achieve a holistic approach to elasticity in public–private hybrid clouds, including the aspect of cloud bursting. Hence, heterogeneity and properties such as cost and leasing policies, as well as distances between data centers are to be considered.

Elasticity of VMs and containers. Both VMs and containers should be subject to optimization, for which the developed approach should facilitate both horizontal and vertical scaling. This has been proposed as "Four-Fold" scaling in existing research [44].

Optimizing container co-location. Given that distributed applications deployed in a microservices fashion typically show dependencies in terms of network interaction, it stands to explore container placement optimization for the following aspects: (a) Favoring co-location of interacting containers on the same VM, which reduces network traffic leaving the VM; (b) reducing the need for communication between containers located in different data centers, in order to minimize induced network hops and latency between data centers.

1.2 Methodological Approach

The methodology applied in this thesis consists of literature and conceptual work, followed by an implementation part and a concluding evaluation. Split into four major parts, it is summarized in the following.

Literature review and requirements exploration. As a first step, a literature review is conducted to find and analyze existing approaches and the covered use cases; which, if applicable, should be incorporated or extended in our work. Based on the conducted research, new elasticity requirements are to be identified, which are subject to open research questions. The specific requirements with regard to hybrid clouds and container co-location should be elaborated as the outcome.

Modeling the system and optimization problem. To find a suitable abstraction of the system, the mechanics of related cloud computing concepts and hybrid cloud infrastructures have to be analyzed with regard to the defined requirements and further incorporated into our solution design. A mode of operation is to be defined which describes how (reactive) monitoring, adaption to configuration changes and allocation of resources should take place. The underlying optimization problem is formulated as an Integer Linear Programming (ILP) model.

Implementation of the optimization algorithm and middleware. Representing the core of our proposed approach, suitable techniques for its optimization capabilities are developed. Besides an implementation of the formulated ILP model, proper heuristics should be explored which are capable of handling the NP-hard optimization problem efficiently. Therefore, we provide two separate optimization algorithms:

(a) Exact optimization based on the proposed ILP model

(b) Heuristic optimization utilizing a Genetic Algorithm

Furthermore, we implement the middleware required to leverage the optimization algorithm and carry out the resulting resource allocation, i.e., in order to lease the right VMs in different data centers and place containers accordingly. The middleware is also responsible for monitoring the system and determining any configuration changes which feed into the optimization and potentially trigger a reconfiguration.

To avoid dependencies on cloud providers, the middleware will simulate all cloud resources instead of providing actual integrations. However, suitable interfaces in its architecture allow for further extensions.

Testbed and evaluation. The previously implemented solution will be evaluated using different scenarios and assumptions. A quantitative evaluation will be conducted by means of monitoring and recording the evaluation results. For this purpose, a realistic testbed is to be set up which relies on the middleware's simulation capabilities. Multiple interconnected microservices will form the application landscape for a representative test setting. As the baseline for our evaluation, we employ a simplistic First-Fit optimization strategy. With different patterns of incoming network traffic applied, our implementation will be assessed with regard to cost, latency, and runtime aspects. The obtained results will reveal whether the proposed optimization approaches achieve significant improvements over the baseline with regard to efficiency and efficacy.

1.3 Structure of the Thesis

The remaining chapters of this work are organized as follows.

In Chapter 2, we provide an overview of the fundamental concepts of cloud computing and elasticity and introduce the related terminology and definitions.

In Chapter 3, we discuss existing approaches proposed in related work as identified during our literature review. Based on that, we state the currently open research challenges relevant to our thesis.

The conceptual part of our work is described in Chapter 4. After having the requirements of our proposed approach defined, we design the middleware's system model and provide a formulation of the underlying optimization problem.

Chapter 5 describes the implementation of the optimization algorithm in detail and provides further implementation details on the developed middleware.

In Chapter 6, we conduct a quantitative evaluation of our proposed approach. After presenting the evaluation scenarios and testbed, we will discuss the gathered results and provide a comparison to a baseline.

Finally, in Chapter 7 we conclude this thesis with a summary of the contributions of our work, and give an outlook for potential extensions in future work.

CHAPTER 2

Background

In this chapter, we present the background of state-of-the-art technologies and approaches that form the basis for the work conducted in this thesis. Along with discussing relevant concepts, we will introduce common terminology that will be used throughout this thesis.

Following an introduction to fundamental cloud computing paradigms, we outline the related concepts of virtualization, elasticity and engineering methodologies. Finally, we will give a brief introduction to relevant optimization techniques.

2.1 Cloud Computing

Cloud computing has been widely adopted and become a well-known term during the last decade, while continuously evolving. The "Cloud" is omnipresent nowadays, and beyond the more technological aspects of underlying cloud computing infrastructure and concepts, it has also become generally known to end consumers. As a nebulously defined term, it often acts as an umbrella term for an abstract, virtual "space" hosting various online services. Cloud-based platforms for file storage, media sharing and consuming such as iCloud[1], Dropbox[2], YouTube[3] and Netflix[4] are used by millions, just to name a few.

The ongoing global expansion of Internet-related infrastructure leads to daily growing consumption of Internet services and increasing throughput. Consequently, data centers need to fulfill ever-increasing requirements for computational power, storage capacity, reliability and speed to accommodate the growing demand. Expensive and maintenance-heavy hardware and software must be added or renewed on a regular basis to operate sufficiently and effective. Many organizations struggle with the rising needs and cannot

[1]https://www.icloud.com
[2]https://www.dropbox.com
[3]https://www.youtube.com
[4]https://www.netflix.com

maintain the necessary facilities, or do not want to obtain and operate the hardware infrastructure required for adequate in-house data centers due to economic reasons.

In consequence, cloud-based infrastructure and services emerged, which facilitate the on-demand utilization of existing shared resources in extent to individual organizations' needs. Prominent examples are Amazon Web Services (AWS)[5], Microsoft Azure[6] and Google Cloud[7]. In recent years, many companies moved away or are transitioning from running their own, dedicated on-premise IT infrastructure, to provisioning their infrastructure in the cloud, and even consuming fully-managed, off-the-shelf software solutions such as email and collaboration applications[8] in favor of previously self-hosted applications [15, 16].

By leveraging cloud computing technology, organizations benefit from multiple notable aspects, by

- only having to pay for actually used resources according to the organization's demand,
- saving fixed cost for on-premise infrastructure in favor of variable cost incurred by cloud resources,
- leaving the duty of operations to *Cloud Service Providers* (CSPs), which manage the setup, maintenance and adaption of infrastructure,
- being able to easily access and scale services on-demand.

2.1.1 Definition

Many definitions of cloud computing are around. Buyya et al. define the essence of "what clouds are" as follows [16]:

> A Cloud is a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on service-level agreements established through negotiation between the service provider and consumers.

Adhering to the National Institute of Standards and Technology (NIST), which published an extensive description of the concepts of cloud computing in 2011, the NIST definition of cloud computing states as follows [60]:

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g.,

[5]`https://aws.amazon.com`
[6]`https://azure.microsoft.com`
[7]`https://cloud.google.com`
[8]Prominent examples are Microsoft 365 (`https://www.microsoft.com/microsoft-365`) and Google Workspace (`https://workspace.google.com`).

> networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

Accordingly, CSPs offer their capabilities to *consumers*, which can be organizations of any kind or their business units. *Cloud infrastructure* is the foundation for any provided *service models* (which we discuss in the following section) and comprises the collection of hardware and software that can be viewed as containing both a physical layer and an abstraction layer. The former includes physical server, storage and network components that are necessary for the cloud services being provided, while the abstraction layer consists of the software which is deployed on the physical layer, and facilitates the essential characteristics [60].

The five essential characteristics of cloud computing are defined by the NIST as [60]:

On-Demand Self-Service Cloud consumers are able to automatically provision and change computing capabilities (e.g., server time and network storage) depending on their needs, without requiring human interaction with the CSP.

Broad Network Access Resources are available via standard network protocols and accessible from heterogeneous client devices.

Resource Pooling Computational resources are pooled by the CSP to serve multiple consumers in a multi-tenancy fashion.

Rapid Elasticity Consumers' demands can be met accordingly by (automatically) provisioning and releasing capabilities. Resources can be requested at any time and appear in unlimited quantity to the consumer.

Measured Service Appropriate to the type of service, resources are automatically controlled and optimized. Resource usage can be monitored, controlled and reported, providing transparency for both providers and consumers.

2.1.2 Service Models

Cloud computing comes in diverse fashion, offering different capabilities suitable for different use cases and consumers. Up to now, three layered main service models have been around which are offered by CSPs: *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) and *Software as a Service* (SaaS), each providing an increased level of abstraction [15].

In the following, we discuss the three cloud computing service models according to the NIST definition. Figure 2.1 shows the increasing scope of providers' responsibilities and the level of abstraction in each service model.

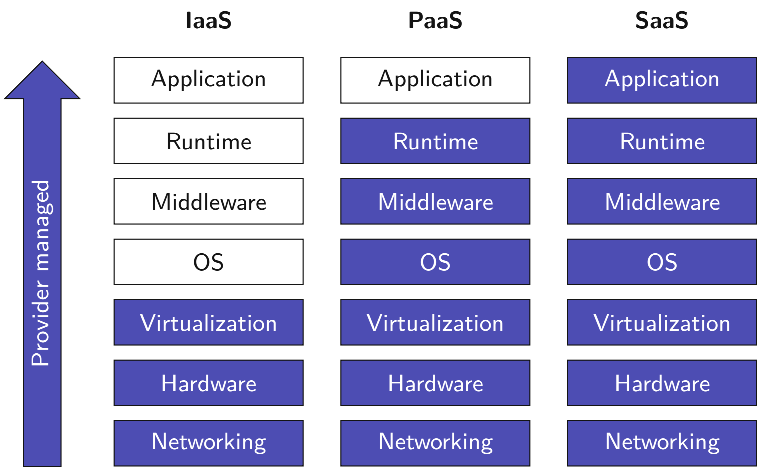


Figure 2.1: Cloud computing service models

Infrastructure as a Service

This service model layer is regarded as "the pillar of cloud computing" [15], offering access to fundamental computing resources (e.g., processing, storage, network), which enables consumers to deploy arbitrary software and operating systems (OS). The underlying "bare-metal" infrastructure is completely managed by the CSP, however, leaving control over the OS and provisioning of (virtual) resources to the consumer. The service provider therefore provides the necessary abstraction layer and software, allowing the consumer to control resources in a self-service and on-demand fashion.

Usually, IaaS involves virtualization as an abstraction layer when accessing capabilities, including the provisioning of computational resources and storage space. Based on its (virtual) capabilities and abstractions, IaaS also acts as the foundation for the following service models [15].

Platform as a Service

PaaS leverages the provider's cloud infrastructure together with supported programming languages, libraries, runtime environments and other tools, to provide a fully-managed framework for consumers to deploy and operate their custom applications. As a consequence and main benefit, the consumer does not need to manage any underlying cloud infrastructure such as (virtual) servers, storage, and OS, since the CSP takes care of managing, maintaining and scaling these capabilities. Consumers remain in control over deploying their applications and are responsible for their configuration.

Software as a Service

The SaaS model offers scalable, ready-to-use software, targeting end-users who access the software using various client devices (e.g., via Web browsers). Service providers take care of the whole application stack and its data, while consumers only get to configure software at the application level as to their requirements. Additionally to managing any underlying infrastructure, the CSP instruments software deployments and updates, and keeps both infrastructure and applications operable with regard to availability and scalability, often tied to Service Level Agreements (SLAs).

2.1.3 Private and Hybrid Clouds

The NIST defines four distinct cloud deployment models, *public*, *private*, *community* and *hybrid* clouds, as follows [60]:

- **Public clouds** are operated by CSPs who make their cloud infrastructure available to the general public. Multiple organizations of different kind use and share the provisioned capabilities.
- **Private and community clouds**: In a private cloud, the infrastructure is provisioned for the exclusive use by a single organization. Organizations may operate it on-premise, however, it may also be managed and operated by third parties, as well as located off-premise. Community clouds are similar, but shared by a specific community of organizations with common concerns.
- **Hybrid clouds** compose multiple distinct cloud infrastructures (with possibly different deployment models) which are bound together, to facilitate portability of data and applications between them. This is enabled by standardized or proprietary technology, while each of the clouds remains a unique entity.

Those different cloud models come with different use cases and benefits. While public clouds can scale virtually unlimited to consumers' demands, and usually bring a cost advantage to their users by offering fine-grained pay-per-use payment models on a utility computing basis, specific use cases for private and hybrid clouds are outlined in the following.

Private Clouds

As security and governance challenges may lead to the need for placing workloads on premises, organizations benefit from using private clouds by achieving high levels of security, since consumer organizations retain full control over their data. Likewise, private clouds allow organizations to meet up with certain regulations and stay compliant, e.g., in health care and financial environments [56].

Hybrid Clouds

Purpose and motivation for employing hybrid clouds are manifold. Distributed hybrid clouds allow to operate in geographically redundant modes, achieving availability and reliability, as failover mechanisms enable to quickly recover from outages and downtime incidents of a particular cloud [56].

Hybrid clouds also enable supplementing private cloud infrastructure with capabilities from a public CSP. These “public–private” hybrid clouds support *cloud bursting*: Businesses do not need to obtain excess hardware infrastructure for their on-premise data center, but are able to offload workload by (automatically) adding public cloud resources on demand. Thus, business opportunities are not lost due to lack of capacity, when the need arises to handle massive amounts of data processing, such as during workload spikes. Furthermore, cost often leads the decision to go down the hybrid path. A hybrid cloud computing strategy is often less expensive from an operational standpoint: Combining fixed cost of a private cloud that covers constant baseload, with variable pay-per-use cost of public cloud resources for accommodating peak load, can bring economic benefits [77, 56].

2.2 Virtualization and Container Technologies

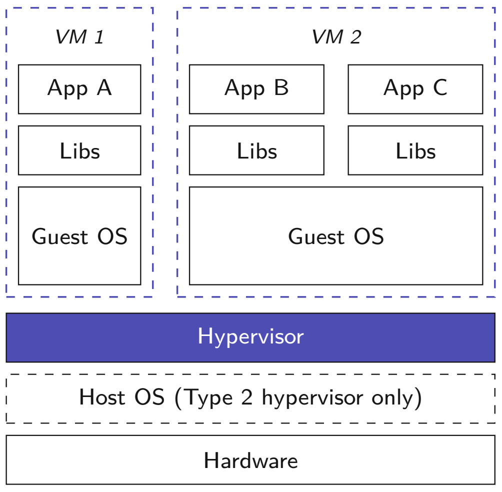
Since resource pooling, as a main characteristic of cloud computing, requires that multiple consumers share a common hardware infrastructure, suitable abstraction layers and technologies are needed which allow to split up and dynamically provision computational resources. Two commonly used concepts are available to facilitate these tasks: *Virtual Machines* (VMs) managed by *hypervisors*, and *containers*. Both offer the necessary mechanisms required for a suitable abstraction layer which provides multi-tenancy and isolation in a cloud environment. In the following sections, we discuss the main features of both concepts, their differences, as well as different and complementary areas of application.

2.2.1 Hypervisor-Based Virtualization

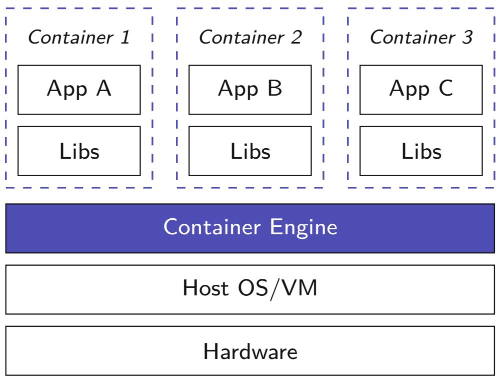
VMs provide the necessary virtualization capabilities for partitioning resources (such as storage and computational resources) on the infrastructure layer of a cloud computing architecture. *Hypervisors*, also referred to as *VM monitors*, are the underlying technology used to create pools of VMs on top of “bare-metal” hardware, i.e., Physical Machines (PMs). As an abstraction layer, the hypervisor provides multi-tenancy and isolation, and manages multiple VMs running on a single host machine. Each VM provides an emulated computer system which runs a *guest OS* by using the processing power, memory, network and disk resources of its host, partitioned and taken care of by the hypervisor, as depicted in Figure 2.2a.

Generally, two types of hypervisors are available to manage virtualization and decouple a guest OS from the underlying hardware [70, 61]:

- **Type 1 hypervisors**, or bare-metal hypervisors, run directly on the physical hardware of the underlying host PM. Usually, the hypervisor is a light-weight software installed on the host, without the need for an OS to run. Multiple hypervisors can be controlled by a management application (*control plane*), which installs and manages VM instances on multiple PMs, allowing for consolidated



(a) Hypervisor-based virtualization



(b) Container-based virtualization

Figure 2.2: Architecture of VMs vs. containers

management and migration of VMs between physical servers. Furthermore, the control plane software can consider requirements such as fault tolerance, optimal server allocation, energy consumption, and overallocation (i.e., allocating more resources to VMs than physically available on a host, assuming that instances often use fewer resources than actually claimed).

- **Type 2 hypervisors** run on a *host OS*, allowing them to run alongside other applications. The guest OS constitutes a process running on the host OS. Type 2 hypervisors are often used by end-users to use different OS on their workstations. Examples include Oracle's VirtualBox[9], VMware Fusion and Workstation[10], and Parallels Desktop[11].

Notably, there exist important hypervisors such as Linux's Kernel-based Virtual Machine (KVM)[12], which cannot be clearly classified as type 1 or type 2, but can be regarded as turning the kernel of the host OS into a type 1 hypervisor.

For the realization of public clouds, CSPs are known to use type 1 hypervisors including KVM. At AWS, for example, the Xen hypervisor[13] is used, while Microsoft uses its proprietary Hyper-V[14] platform for its cloud offerings.

[9]https://www.virtualbox.org
[10]https://www.vmware.com/at/products/desktop-hypervisor.html
[11]https://www.parallels.com/de/products/desktop
[12]https://www.linux-kvm.org
[13]https://xenproject.org
[14]https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v

2.2.2 Container Technologies

Opposed to VMs, container-based virtualization is a more “light-weight” virtualization technology. Containers effectively provide a virtualized OS using the services, networking and filesystem of a shared, underlying host OS, while keeping the resources of concurrent containers isolated (cp. Figure 2.2b).

Container-based virtualization shares the kernel of the host OS between all container instances (OS-level virtualization), while hypervisor-based virtualization emulates the hardware and brings up a completely new OS for each VM instance (hardware-level virtualization). This leads to less resource usage and performance benefits over hypervisor-based virtualization. Especially notable, containers benefit from significantly reduced startup times compared to VMs, as there is no need to boot a new OS each time a new container is started. However, due to the shared OS kernel, containers are limited to using operating systems that are based on the same kernel as the common host OS. The *container engine* (the equivalent to the hypervisor in VM-based virtualization) manages the resources of the whole container stack, and operates directly on top of the host OS [61]. As typically seen in cloud environments, containers most commonly run inside a (type 1 hypervisor) VM, rather than on bare-metal PMs. In this case, the guest OS of a leased VM resembles the containers’ host OS.

Today, distributed applications and cloud infrastructure are moving away from being VM-centric to being container-centric, due to smaller performance footprints and eased portability of containers [20]. *Containerization* refers to applications being packed and executed inside a container. This makes containers a “commodity”: Containerized applications can be run on any platform regardless of the packed applications and their runtimes, ultimately facilitating a standardized way for distribution, and are therefore most popular in the context of cloud computing.

Docker[15] dominates the field of container technologies and is the leading technology for container-based software development, thus being adopted by most CSPs [50]. Docker containers are run off *images*, which are built to the developer’s purpose. When building a Docker image, one usually starts from a *base image*, which contains the required runtime binaries and libraries, with the executable application being added on top. For persistent data storage, *volumes* need to be mounted to a container, which allows data to be persisted, e.g., on the host filesystem. Otherwise, any content inside a container is lost after it is removed.

Container as a Service

Besides the three main service models of cloud computing, new ones emerged with the progress of used technologies. The improved utilization of physical resources and further abstraction provided by containers lead to a novel service model which treats containers as first-class citizens, known as *Container as a Service* (CaaS) [15]. CaaS typically

[15] `https://www.docker.com`

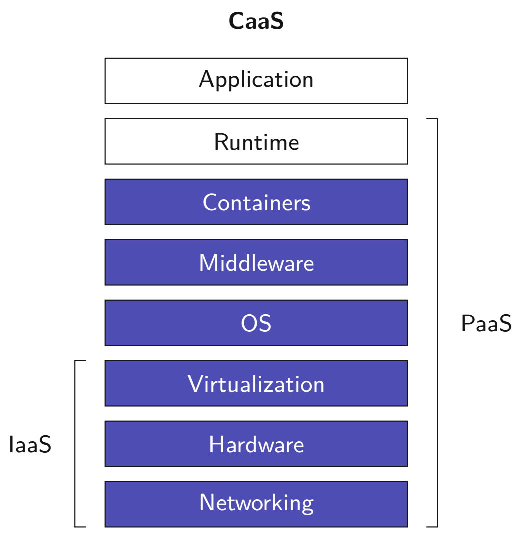


Figure 2.3: The CaaS service model (adapted from [82])

operates on top of the IaaS layer. While VMs provide the virtualized compute resources, CaaS provides the isolated environments for the deployment of containerized workloads. In contrast to PaaS, CaaS provides no preconfigured application runtime out of the box, however, it allows to run any specific made-to-measure application environments inside the consumer-provided containers. As depicted in Figure 2.3, CaaS offers one level less of abstraction compared to PaaS. Consequently, when operating a PaaS, CaaS may be used as a foundation instead of plain VM-based IaaS.

Container Orchestration

With the rise of containers, solutions for container orchestration which aid management of containerized applications have also gained wide-spread traction. When deployed in the cloud, container orchestration platforms are used to schedule containers on VM instances of an attached *VM cluster*. Additional to performing container scheduling, container orchestration platforms cope with cross-cutting concerns such as scaling, load balancing, service discovery and monitoring, and thus are particularly used when deploying modularized applications in a microservice setting (see Section 2.4) [17].

Among others, Kubernetes[16] is a popular open-source platform for orchestration of Docker containers, originally designed by Google as their in-house container management. Most CSPs have platforms supporting Kubernetes in their offering, which provide Kubernetes clusters in a CaaS fashion.

[16]`https://kubernetes.io`

2.3 Elasticity

Elasticity can be outlined as the ability of a system to automatically adjust to workload demands. In this section we discuss the definition and characteristics of elasticity, differentiate it from scalability, and provide an overview of the main concepts applied to achieve both scalability and elasticity.

2.3.1 Definition

As has already been referred to in Section 2.1.1, the NIST defines "Rapid Elasticity" being a main characteristic of cloud computing. According to the NIST, elasticity links scalability to resource demand: "Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand" [60].

However, multiple other definitions are around. Some characterize elasticity being similar to scalability with added automation, as Schouten [76]: "Elasticity is basically a 'rename' of scalability [...]" and "removes any manual labor needed to increase or reduce capacity".

Cohen, being closer to the NIST definition, defines elasticity as "the quantifiable ability to manage, measure, predict and adapt responsiveness of an application based on real time demands placed on an infrastructure using a combination of local and remote computing resources." [25]

While these definitions vary in scope and perspective, we follow a refined definition from Herbst et al. [43], which tries to capture the core aspects of elasticity more thoroughly:

> Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible.

Al-Dhuraibi et al. [3] summarize elasticity using the following equation:

$$Elasticity = scalability + automation + optimization$$

2.3.2 (Auto-)Scaling

As a main ability of – and prerequisite for – elasticity, the term *scalability* is often used interchangeably. In this thesis, we will follow a more narrow definition for scalability as distinguished from elasticity, to clearly differentiate between these related terms.

Based on Herbst et al. [43], we define scalability as follows:

> Scalability is the ability of a system to sustain increasing workloads by making use of additional resources. It does not consider temporal aspects of how fast, how often, and at what granularity scaling actions can be performed. In

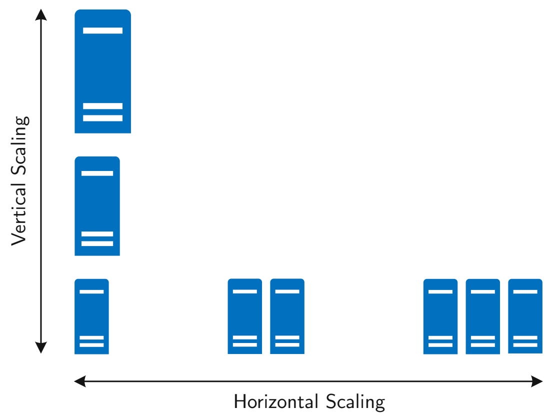


Figure 2.4: Vertical and horizontal scaling

contrast to elasticity, it is not directly related to how well the actual resource demands are matched by the provisioned resources at any point in time.

When scaling actions are executed automatically without manual intervention, this is referred to as *auto-scaling*.

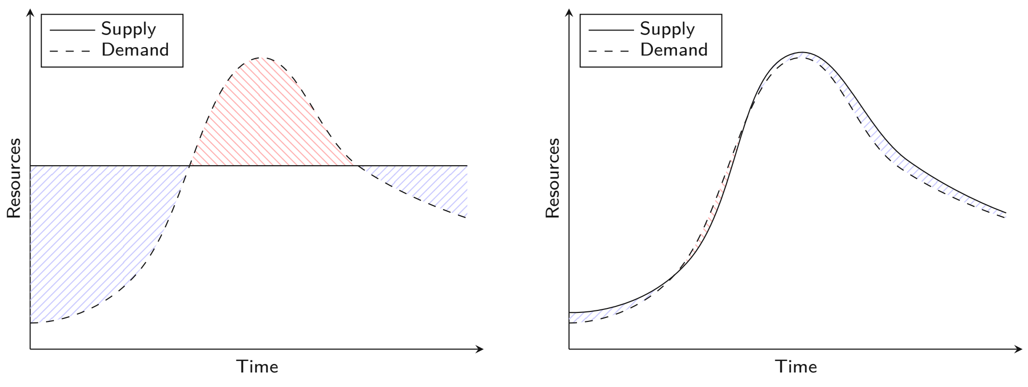
2.3.3 Scaling Dimensions

When resources are scaled either manually or automatically (e.g., in the context of an elastic system), scaling actions can take place in two dimensions as illustrated in Figure 2.4:

- **Vertical scaling:** Considering single components, vertical scaling adjusts their individual resource capacity. I.e., VMs or containers are scaled *up* by adding, and scaled *down* by reducing the assigned resources of an instance. Likewise, when vertically scaling bare-metal PMs, additional hardware needs to be installed, e.g., by adding more physical memory.

- **Horizontal scaling:** In this case, the individual components themselves remain untouched. Instead, the number of provisioned instances is changed by *scaling out* (adding instances) and *scaling in* (removing instances) the pool of available instances, in order to increase or decrease the system's overall resource capacity. Horizontal scaling is the most widely used method provided by CSPs [3].

Hypervisor-based virtualization is in particular suitable for horizontal scaling of VMs, while vertical scaling of VMs at runtime poses various challenges, as reassigning resources often requires a reboot of the VM. Container engines, besides facilitating horizontal scaling, also ease vertical scaling at runtime, since allocated resources can easily be adapted on-the-fly. Moreover, light-weight containers also provide benefits to horizontal scaling, as they typically offer shorter startup times.



(a) Over-/under-provisioning (b) Efficient provisioning

Figure 2.5: Over- and under-provisioning vs. efficient resource provisioning

2.3.4 Efficient Provisioning

As stated in the definition of elasticity in Section 2.3.1, resource efficiency plays an important role in elastic cloud computing. Since elasticity aims to adapt resource provisioning as quickly and as close to the optimum as possible by taking the right scaling actions, efficiency acts as one of the main purposes to do so. While various purposes for optimization objectives are around, the most common goals found in elastic approaches are to minimize cost and maintaining *Quality of Service* (QoS), i.e., ensuring that a system remains operable and fulfills given performance constraints [3].

To optimize for both, these two opposed goals need to be efficiently balanced. Simply provisioning a system with a maximum of resources certainly ensures QoS; however, the constant over-provisioning would incur unnecessary economic waste. An elastic system therefore strives to utilize cloud resources efficiently, and reduce any *over-* and *under-provisioning* as illustrated in Figure 2.5, in order to achieve a balanced state [3]:

- **Under-provisioning** refers to a state where supplied resource capacity does not meet resource demand. As the consumer needs more computational resources than provided, this leads to performance degradation: I.e., the workload may be too high to be processed, or suffer from slowdown, which leads to violated QoS requirements. Though no additional cost can be measured in this scenario directly, outage of critical systems and dissatisfied customers impact a business in the long run.
- **Over-provisioning** is entered when a system's supplied resources exceed demand, i.e., when provisioned resources are greater than the actually required capacity. While QoS requirements can be fulfilled, extra and unnecessary cost is incurred. This can be avoided by releasing resources, ideally minimizing them to meet the optimum in a balanced state as close as possible.

To determine the degree of efficiency of an elastic system, there are no common metrics and methodologies around. Besides various performance- or cost-related metrics, consumers could determine the frequency of over- and under-provisioning states being entered, the amount of absent/excess resource capacity, or the delay it takes to provision and release those resources, in order to derive quantitative or temporal metrics for efficiency [3].

2.4 Microservice Architectures

Microservices are an architectural style of developing and deploying complex software applications, aiming to improve a software's development lifecycle, delivery and deployment processes [55, 42].

As opposed to microservices, applications following a traditional monolithic architecture are built as a single unit (the *monolith*), which serves business logic, data processing, and the (client-side) user interface of a system. When requirements change or new ones emerge, code changes need to be integrated with any other ongoing development, followed by a full build of the whole application, as a prerequisite to releasing and deploying a new version of the monolith. Even if changes only affect a small part of the application, a complete rebuild and deployment is required, as the change and deployment cycles of a monolith's software modules are tied together. Replacing traditional monolithic applications and Service-Oriented Architectures (SOAs)[17], microservices have become a best practice and often the default style for creating enterprise applications at scale [55].

2.4.1 Characteristics

Microservice architectures pursue specific goals and have certain characteristics, differentiating them from SOA and monolithic architectures [55, 74]:

Specialization Microservices are organized around business capabilities and solve specific problems. Large systems are therefore decomposed into multiple, small services representing sub-domains of the system. The services' domain boundaries are typically determined following Domain-Driven Design (DDD) [32] to decompose a larger, complex domain into several bounded sub-domains. Consequently, one microservice often implements a single use case, such as invoicing.

Isolation and Independence Microservices are self-contained components which have no shared codebase and no shared dependencies. This makes microservices independently upgradeable, replaceable, and in particular independently deployable. Allowing for eased technological changes, the platform and technology stack for each service can be flexibly chosen depending on developer preference or business requirements specifics. At runtime, microservices run in separate processes, and may communicate with each other via Remote Procedure Calls (RPCs), Web-based protocols or asynchronous messaging (see Section 2.4.2). Well-defined remote interfaces encourage loose coupling and cohesiveness of services.

[17]SOA usually refers to a style where an Enterprise Service Bus (ESB) is used to integrate monolithic applications, or ESBs even apply business logic and abstract away complexity. Sometimes microservices are considered a fine-grained variant of SOA [42].

Owned by Small Teams Microservices are tied to small, autonomous teams which have full ownership of a particular microservice over its lifetime. I.e., one cross-functional team is responsible for (product) development, delivery and operations of a service. Regarding team size, Amazon shaped the notion of "two pizza teams", meaning that the size of a microservice should correspond to a team of not more than 12 developers.

Per-Service Data Stores Persistent data is privately owned by each microservice. Data stores (such as databases or BLOB storage) are attached to and managed by individual single services. This results in many logical data stores corresponding to the microservices' sub-domains, versus a large, shared database in traditional monoliths.

Resiliency The independence of distributed services encourages resiliency, i.e., a design for failure, which allows the overall system to be more resistant to individual service failures [55]. Typically, a failing microservice results in a degraded state of the system with some functionality becoming unavailable, while a failing monolith would cause the whole application to crash.

While these characteristics provide many benefits, especially to organizational culture (as discussed in Section 2.4.4), microservices also come with some drawbacks:

- Compared to monoliths, microservice architectures come with increased complexity as distributed systems bear additional technical effort.
- Remote inter-service calls affect performance due to network latency.
- Trade-offs must be made with regard to transaction consistency: Distributed transactions across multiple services lack atomicity, as typically found in database transactions that run within the process of a monolith, and therefore only provide eventual consistency [33].

2.4.2 Communication Paradigms

When accessing microservices or performing inter-service communication, two common paradigms are applied [55]:

Synchronous Request–Response Communication This type of inter-process communication is necessary when requesting data as an (external) client, or in case a synchronous downstream service call is unavoidable within a transactional process. A client initiates a call to a remote service and waits until receiving a response. Most commonly, HTTP-based RESTful APIs[18] expose endpoints to operate on a service's domain resources. Relying on well-established WWW standards and protocols such as HTTP eases interoperability, documentation for client developers, and leverages built-in capabilities for, e.g., caching and authentication.

[18]Application Programming Interfaces (APIs) that follow the Representational State Transfer (REST) paradigm

Asynchronous (Publish–Subscribe) Messaging Provided by lightweight message brokers, exchanging messages over message channels (with a persistent message queue) facilitates temporal decoupling between delivering and receiving a message. As a popular messaging pattern, *publish–subscribe* allows publishing a message to multiple subscribers. Inter-service communication in such asynchronous manner, where senders and receivers are not required to be available at the same time for the duration of a request, is a key enabler for achieving resiliency.

Importantly for microservices, in both cases the underlying communication infrastructure is “dumb”, i.e., Web protocols and message brokers have no notion of the system’s domain specifics or business logic (as opposed to, e.g., ESBs in SOAs) [55].

2.4.3 Common Patterns

When building microservice architectures, various patterns have emerged for certain requirements, such as discoverability. The following are an excerpt of the most important patterns applied for microservices [73]:

Server-Side Service Discovery and Load Balancing Each service instance registers and deregisters itself with a central *service registry*, or directly with a *load balancer*. Load balancers are often part of IaaS offerings. In case of horizontal scaling, the load balancer directs traffic between multiple instances of the same service. External clients or other microservices make requests via the load balancer using its DNS name, which subsequently forwards a request to one of the microservice instances looked up from the service registry.

Client-Side Service Discovery Different from server-side discovery, available service instances are looked up by the requesting microservice (the client) itself from a central service registry, eliminating the need to route requests through a load balancer. This pattern is also referred to as *client-side load balancing*. However, external clients in most cases still require a load balancer, which is reachable through a public domain name, as an entry point to access the services.

API Gateways Microservices are often accompanied by a dedicated service which acts as a single entry point for API requests of (external) clients. In its simplest form, an API gateway acts as a proxy routing requests to downstream services. Additionally, API gateways may solve cross-cutting concerns such as authentication and API versioning. As load balancers sometimes come with similar built-in capabilities, they may also take on the responsibilities of API gateways to a limited extent.

Backends for Frontends (BFFs)[19] Comparable to a specialized API gateway, a BFF provides APIs specifically for a particular client or class of clients (e.g., Web frontends or mobile apps). For this purpose, BFFs aggregate data from downstream

[19] `https://samnewman.io/patterns/architectural/bff`

services (using synchronous HTTP calls) and represent it exactly to the client's needs.

Micro Frontends[20] Applying the concept of microservices to the user-facing frontend components of a system, micro frontends have emerged. Instead of a single frontend application that acts as the user interface and connects to multiple backend microservices, multiple smaller frontend services take responsibility of bounded use cases.

2.4.4 DevOps Methodology

DevOps refers to a set of practices and tools for integrating traditional software development *(Dev)* with IT operations *(Ops)* and breaking down their organizational silos to aid faster software delivery cycles [71, 87].

An important role in DevOps play streamlined processes and automation techniques with regard to delivery. *Continuous Integration* (CI) and *Continuous Delivery/Deployment* (CD) are practices of frequently integrating, building, testing, and in the latter, delivering an application when changes to its codebase occur. The process of (semi-)automatically promoting code changes in such consecutive stages is referred to as *delivery pipeline*, which is typically carried out by a central CI/CD server[21]. Noteworthy, if the process fails at any stage, this results in the whole pipeline being aborted, forcing the team to fix any problem that broke the pipeline. In the case of CD, the pipeline includes distributing a production-ready artifact, such as an executable or container image. Subsequently, the application may be deployed upon manual approval (continuous *delivery*), or is being deployed to a staging or production environment fully automatically (continuous *deployment*).

Microservices effectively help adopting a DevOps culture by promoting small cross-functional teams and easing related organizational challenges [11]. As part of the delivery pipeline, microservices often rely on containers for deploying modularized services [50]. At each successfully built pipeline of a microservice, a new Docker image is being packaged, distributed, and potentially deployed on suitable infrastructure (e.g., using CaaS or container orchestration solutions, as discussed in Section 2.2.2). A high degree of automation for the whole pipeline improves speed and leads to the desired shorter *cycle times* for individual microservices (i.e., the time between developing a new feature and shipping it to production).

By combining microservices with DevOps methodologies, organizations benefit from autonomous services being developed and frequently delivered in parallel, ultimately resulting in increased agility and faster time-to-market.

[20] `https://micro-frontends.org`

[21] Popular open-source tools to accomplish CI/CD are Jenkins (`https://www.jenkins.io`) and Gitlab (`https://about.gitlab.com`), among others.

2.5 Optimization Techniques

Optimization plays a central role in our work. The term *optimization problem* refers to decision problems which aim at finding the best (i.e., optimal) out of multiple possible solutions. In the following sections, we discuss basic concepts of optimization and the fundamental techniques used to design and implement the approach of this work.

2.5.1 Complexity

An important factor for solving optimization problems is complexity. To characterize the hardness of problems, complexity classes describe the amount of time required to compute a solution for a given problem. Two important classes are P and NP. Decision problems in class P can be solved in a polynomial amount of computation time, and are therefore regarded as being efficiently solvable. In contrast, problems in NP are to be solved in non-deterministic polynomial time (i.e., there is no known algorithm to solve any given instance of the problem in polynomial time), and may become practically unsolvable with growing problem instances. *NP-complete* problems belong to the hardest problems of class NP, whose algorithms are capable of solving any other problem in NP [31].

2.5.2 Heuristics

An often encountered problem with exact optimization algorithms lies in the lack of sufficient efficiency, as exact solvers for NP problems may not achieve a result within reasonable time for large problem instances. In order to overcome such limitations, we can apply heuristic algorithms which find fast approximate solutions, and therefore are often more useful in terms of effectiveness and performance. While they are not guaranteed to find the exact optimum, heuristics still lead to good solutions, and provide a trade-off between resolution time and accuracy [66].

2.5.3 Linear Programming

An important optimization model, which is also employed in this work, is *Linear Programming* (LP), in which equations and constraints form a linear relationship. Linear optimization problems are formulated by defining an objective function along with linear constraints (equations or inequations). In its canonical form, LP models maximize the objective function, however, applying linear programming for minimization problems works analogously [66].

Many applications for optimization problems do not work with continuous variables, but require some variables to be integers. I.e., units of produced goods typically do not make sense as real numbers. Therefore, *Integer Linear Programming* (ILP) is a special variant of linear programming, which restricts variables to take integer values. Likewise, *Mixed Integer Linear Programming* (MILP) allows for both discrete and continuous decision variables. (M)ILP optimization problems are known to be NP-hard. Several

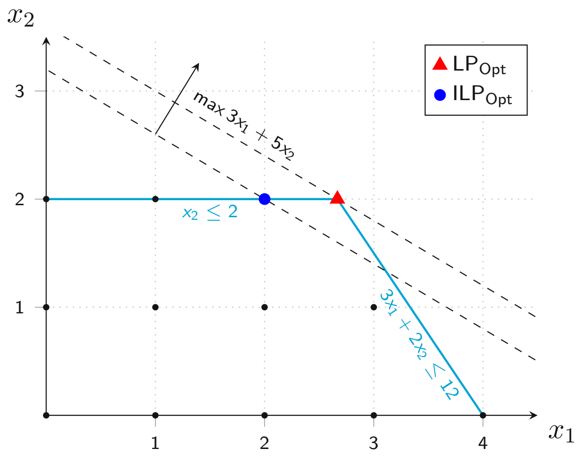


Figure 2.6: Plot of an LP optimization problem and its optimal solution

exact algorithms for solving LP-based problems exist, such as the cutting plane method, branch and bound method and branch and cut method [66].

The following example comprises a simple LP optimization problem, illustrated by a geometric plot in Figure 2.6:

$$\max\ 3x_1 + 5x_2 \tag{2.1}$$
$$x_2 \leq 2 \tag{2.2}$$
$$3x_1 + 2x_2 \leq 12 \tag{2.3}$$
$$x_1, x_2 \in \mathbb{R}_{\geq 0} \tag{2.4}$$

The objective function of our problem is defined in Equation 2.1 with x_1, x_2 being the decision variables to maximize. Inequations 2.2–2.4 formulate the constraints which restrict the variables as required. Both LP and ILP optima of the problem are shown in the geometric solution.

2.5.4 Genetic Algorithms

As a type of evolutionary algorithms, Genetic Algorithms are metaheuristics that incorporate mechanisms inspired by biological evolution. Using the principles of *natural selection* based on Charles Darwin's theory of evolution, Genetic Algorithms can provide high-quality solutions to optimization problems. By repeatedly adapting and reproducing good potential solution candidates while rejecting bad ones, the solutions evolve towards better results over successive generations [66].

The basic structure of a Genetic Algorithm starts by generating an initial *population* containing possible solution candidates (*individuals*). For that matter, individuals are

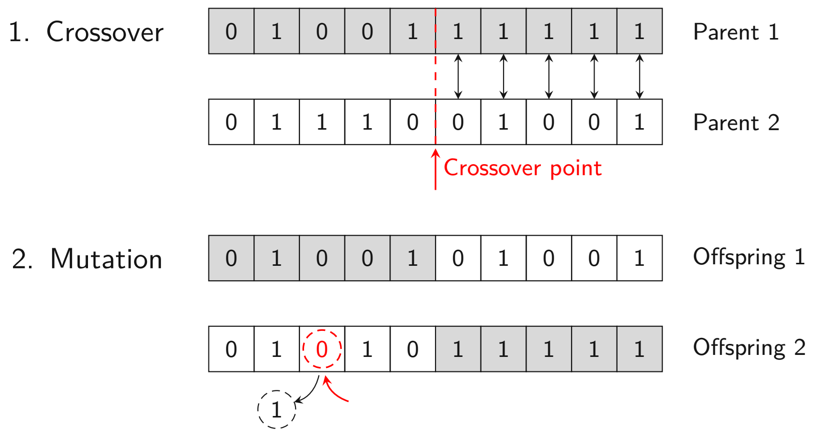


Figure 2.7: Crossover and mutation steps of a Genetic Algorithm

generated randomly or based on a seed; the generated population's size depends on the nature of the problem domain. Each individual of the population shows a *fitness* with regard to the optimization goal, which is used to distinguish good from bad solutions and to finally choose the best-fitting individual as the designated optimum [85].

In order to calculate a fitness score for each individual, a *fitness function* needs to be formulated which is feasible for the problem domain and covers the optimization requirements. When using a Genetic Algorithm as an approximation model, the fitness function typically follows the objective function of the original optimization problem (e.g., the objective function of an ILP model). Additionally, the fitness function typically includes a penalty function that evaluates to 0 for feasible solutions, and to a positive value for infeasible ones (e.g., if constraints are violated) [66].

To ensure evolution of a population, the following three repetitive steps are performed at each generation [85]:

- **Selection:** The fitness for each individual in the population is evaluated according to the fitness function. The fittest individuals are selected for reproduction ("survival of the fittest").

- **Crossover:** From the pool of selected individuals, pairs of parents are destined for breeding. In the simplest form of crossover, genes of the parents are exchanged until a crossover point is reached (see Figure 2.7). The resulting offspring contain a mixture of their parents' genes.

- **Mutation:** Some of the resulting chromosomes can be mutated, i.e., randomly altered as illustrated in Figure 2.7, which stimulates diversity within the population. The mutation rate needs to be chosen carefully, to achieve a beneficial degree of genetic drift without losing good individuals.

After each reproduction, the generated children are combined to form a new population. The algorithm is repeated until a specific termination condition is met, e.g., if an individual is found which achieves a certain level of fitness, if a fixed number of generations has been reached, or if no significant improvements could be made over a certain number of generations.

CHAPTER 3

Related Work

Approaches for basic auto-scaling strategies are well-known since the rise of cloud computing, and also have been applied with regard to container technologies, e.g., Docker. The development of more sophisticated algorithms and strategies towards optimized container elasticity for various user requirements is yet subject to ongoing research, as various studies show [15, 20, 3, 50, 19].

In this chapter, we will therefore explore relevant approaches that are related to the topic of our work. First, we provide an overview of existing auto-scaling mechanisms of state-of-the-art container technologies and service offerings by public cloud providers. Subsequently, we look in detail at current research approaches that deal with resource provisioning, container placement and optimization problems to cope with elasticity of computational resources and containers in the cloud, while considering different purposes and user requirements. We then give a comparison which summarizes the most relevant discussed approaches. Finally, we state the open research areas and challenges with regard to the aim of this thesis.

3.1 Auto-Scaling in Established Technologies and Cloud Services

Leading cloud providers such as AWS, Microsoft Azure and Google Cloud provide computing infrastructure (IaaS) featuring auto-scaling capabilities, as well as managed platforms for container deployment (CaaS). Furthermore, several (open-source) solutions for container orchestration are around, the most popular being Kubernetes [84]. We will outline these state-of-the-art approaches and their auto-scaling capabilities in the following.

3.1.1 Auto-Scaling in Commercial IaaS

Commercial CSPs offer at least basic auto-scaling capabilities for their computational resources (i.e., VMs). These are represented by Amazon EC2 Auto Scaling [5], Azure Autoscale [62] and Google's Autoscaling Groups [34].

The offerings are characterized by providing horizontal scaling for VM clusters, also called *fleets*, in terms of adding and releasing computational resources. Scaling is triggered based on various metrics, such as CPU usage or request count, and thereon based scaling policies (i.e., thresholds and rules). Predictive scaling strategies, such as provided by Amazon EC2 which therefore employs machine learning [12], are available but not disclosed in detail.

3.1.2 Managed Container Orchestration and CaaS

With the rise of containers, CSPs have extended their offering in recent years, providing managed platforms for container deployment on top of their IaaS offering.

Amazon Elastic Container Service (ECS) [7] is a managed service for orchestration of Docker containers that builds on Amazon's EC2 infrastructure. It provides automatic horizontal scaling of container instances, while additionally applying horizontal auto-scaling to the underlying VM cluster. With AWS Fargate [10], AWS offers to run ECS clusters on serverless infrastructure, i.e., hiding the presence of any computational resources and VM provisioning from the user. Azure's Container Instances [64] is a similar serverless platform for deploying isolated container workloads. However, the internals of referred serverless CaaS offerings are not disclosed with regard to details about infrastructure provisioning, scaling and container allocation.

Another cloud provider, Jelastic[1], offers container hosting and PaaS featuring auto-scaling capabilities. Noteworthy, they claim to be the only CSP that can automatically scale applications vertically [45].

All three leading CSPs offer managed Kubernetes clusters in a CaaS fashion. Amazon Elastic Kubernetes Service (EKS) [8], Azure Kubernetes Service (AKS) [63] and Google Kubernetes Engine (GKE) [35] each feature basic (horizontal) auto-scaling, respectively the auto-scaling possibilities of Kubernetes as described in the following.

3.1.3 Container Orchestration Frameworks

Most container orchestration frameworks feature built-in auto-scaling capabilities.

The open-source framework Kubernetes allows for horizontal auto-scaling of both VM clusters and *pods*, which embody container instances. The Cluster Autoscaler [51] adjusts cluster size as needed in order to run additional containers, and comes with built-in support for the infrastructure of the leading CSPs, among others. Using the Horizontal

[1]`https://jelastic.com`

Pod Autoscaler [52], the number of replicated pods can be automatically scaled based on metrics like CPU utilization. A beta version of the Vertical Pod Autoscaler [53] is available, which accounts for resource consumption or any custom metrics in order to resize containers vertically. However, this beta comes with several limitations, e.g., it is not designed for use together with the Horizontal Pod Autoscaler. Kubernetes Event Driven Autoscaling (KEDA) [23] complements auto-scaling of pods based on various event sources for metrics.

Other solutions for container orchestration provide similar capabilities. Docker Swarm[2] allows users to horizontally scale containers in and out [27]. Though automatic scaling is not available in Swarm by default (i.e., scaling commands have to be applied manually), instrumenting it to do so can be achieved relatively easily by employing custom scripts to facilitate the adaption process. Red Hat OpenShift[3] is a Kubernetes-based platform supporting horizontal cluster auto-scaling using its ClusterAutoscaler [72]. The Autoscaler of HashiCorp Nomad[4] supports horizontal application auto-scaling [40]. For the open-source system Apache Mesos[5], auto-scaling is available using Clusterman [89].

3.2 Advanced Approaches Towards Container Elasticity

Besides the naïve, mostly horizontal scaling techniques referred to in the previous section, exploring more sophisticated approaches for elasticity is of particular interest especially when it comes to containerized applications and microservices.

Existing efforts and current research approaches are of manifold shape. E.g., some of them might favor vertical or horizontal scaling, while others follow “hybrid” strategies for combined horizontal and vertical scaling. Systematically, existing approaches can be grouped by various aspects, as done by Al-Dhuraibi et al. The most relevant scopes of elasticity for the topic of this work are *Infrastructure* (i.e., VMs or containers), *Purpose* (e.g., performance, cost, availability), *Mode* (reactive vs. proactive/forecasting), *Method* (i.e., horizontal or vertical scaling) and *Provider* (single vs. multiple). The most common goals of observed research approaches are optimization for performance and/or cost [3].

3.2.1 Scheduling and Scaling of Containers

Some works build on improving or extending the auto-scaling capabilities of existing container orchestration tools, which we have outlined in Section 3.1. For example, Kaewkasi and Chuenmuneewong [46] propose an algorithm based on Ant Colony Optimization (ACO), which improves scheduling of containers inside Docker Swarm with regard to resource utilization and performance. Compared to the default, greedy round-robin algorithm of Docker Swarm, their metaheuristic algorithm achieves improvements in

[2]`https://docs.docker.com/engine/swarm`
[3]`https://www.openshift.com`
[4]`https://www.nomadproject.io`
[5]`http://mesos.apache.org`

application performance by approximately 15 %. However, no actual auto-scaling is taking place, i.e., this approach only addresses the problem of optimally scheduling a set of given containers. Casalicchio and Perciballi [21] propose an algorithm for improved horizontal auto-scaling with Kubernetes based on absolute metrics (KHPA-A). With regard to response time, a reduction of up to 66 % compared to Kubernetes' built-in Horizontal Pod Autoscaler is achieved.

Nardelli et al. propose an approach for Elastic Provisioning of VMs for Container Deployment (EVCD) [68] using an ILP optimization model. VMs are leased and released on demand and containers are relocated at runtime to optimize for multiple QoS metrics, while explicitly taking into account heterogeneous container requirements and characteristics (e.g., reusable container image layers). Replication and scaling of containers, however, is not considered. Deployment times could be reduced significantly compared to traditional greedy heuristics, which averaged 40 % higher than the optimal results.

A notable example for vertical container elasticity is ElasticDocker by Al-Dhuraibi et al. [2]: The proposed system powers autonomous vertical elasticity for Docker containers with live migration of containers between VMs. The system employs principles of the MAPE-K (*Monitor–Analyze–Plan–Execute over a shared Knowledge*) feedback loop for monitoring and controlling elasticity. Aiming for optimal performance, ElasticDocker scales the CPU and memory resources assigned to each container up and down based on workload. When resizing is no longer possible on the same host machine, a live migration of containers to another host is carried out. Outperformance against Kubernetes' Horizontal Pod Autoscaler is shown to be up to 38 %.

Khazaei et al. propose a general purpose platform for auto-scaling of microservices. Elascale [48] aims at extendability, and due to its agnostic design, different cloud infrastructure and container technologies can be supported with plug-ins. It leverages the open-source ELK[6] stack for the collection, storage and analysis of performance metrics. In the presented work, Docker Swarm is used as engine for container provisioning. Elascale ships with a reactive, threshold-based default algorithm for horizontal auto-scaling of microservices, which, additionally to resource (CPU, memory, network) utilization, takes into account a replication factor as dependency among services. This allows to incorporate requirements such as: "For each instance of a service X, there must be two running instances of service Y".

Zhang et al. propose a Genetic Algorithm-based approach for energy-efficient container placement in a CaaS scenario [90]. Therefore, containers are placed on a variable number of VMs, which for their part are allocated on PMs. Similarly, Tan et al. address the same optimization goal with a Genetic Algorithm that uses indirect problem representation in a dual-chromosome approach [81], and later propose an improved Group Genetic Algorithm [80] that outperforms rule-based approaches and previous Genetic Algorithms.

[6]Elasticsearch, Logstash, Kibana

3.2.2 Hybrid Scaling

Combined horizontal and vertical scaling of resources allows for more holistic approaches towards elasticity and a higher degree of optimization. We refer to this as “hybrid” scaling, according to Kwan et al.

With HyScale [54], Kwan et al. introduce an approach for fine-grained, combined vertical and horizontal container auto-scaling with regard to microservices. It aims for high performance (response times and availability) and resource utilization. In addition to scaling containers horizontally by calculating the number of required replicas per microservice, the hybrid approach focuses on a deterministic calculation of each microservice’s needs. The scaling algorithm takes into account CPU and memory utilization. To achieve fine-grained adjustments, containers are vertically scaled on the same machine, granted that enough resources are available. If there are insufficient free resources to meet a container’s scaling demand, horizontal scaling on another VM is performed. However, scaling of VMs is not addressed. The approach achieves up to 49 % faster response times compared to Kubernetes’ Horizontal Pod Autoscaler.

Earlier approaches focus solely on hybrid scaling of VMs, for instance SmartScale by Dutta et al. [28]. Their proactive approach aims to optimize VM scaling with respect to incurred cost, while keeping performance impact at a minimum. Another work by Yang et al. presents a cost-aware approach using workload prediction, which combines vertical and horizontal aspects in their VM scaling algorithm [88]. Han et al. present an approach for elasticity of multi-tier applications, which scales Web servers, application servers, and databases by adding, removing, or modifying (i.e., vertically scaling) VM instances with regard to optimization for cost-aware criteria [39].

3.2.3 Four-Fold Scaling of Containers and VMs

Hoenisch et al. [44] were the first to consider elasticity as a holistic optimization problem for both containers and VMs. In their approach, hybrid scaling is applied in four dimensions, by scaling both containers as well VMs horizontally and vertically. This is referred to as “Four-Fold Auto-Scaling”. The allocation decision is formulated as a multi-objective optimization problem using ILP. Vertical scaling of containers follows a coarse-grained model in their approach: When allocating containers for a given application, a set of discrete container configurations is available to choose from, each with predefined CPU and memory requirements and a maximum amount of acceptable requests. The optimization model allocates containers in their optimal configuration on a set of VMs, which are of optimal size with regard to cost-efficiency, while taking into account QoS constraints and startup times of VMs and containers. Compared to naïve scaling strategies, their approach achieves a cost reduction of 20 % to 28 %.

Likewise, Nardelli et al. use ILP in their approach for Adaptive Container Deployment (ACD) [69]. The proposed solution exploits horizontal and vertical elasticity of containers and VMs, while taking into account geo-distributed infrastructures, i.e., locations of data centers. Their model allows to require maximum network delays between microservices.

The formulated multi-objective ILP model saves up to 44 % of deployment cost compared to popular greedy bin-packing heuristics, but this NP-hard approach takes an average resolution time of 17 s. The faster heuristics, however, are not network-aware, hence cannot respect applications' network delay constraints.

3.2.4 Towards Co-Location and Affinity

Few efforts have been made to exploit advantages of co-location, in particular with regard to network latency.

Sampaio et al. propose REMaP [75], a novel approach for runtime placement adaption of microservices, which aims at performance optimization based on microservices' communication affinity (i.e., the number and size of messages exchanged over time). Performance improvements are achieved by co-locating high-affinity microservice instances on the same host. REMaP performs placement and live migration of microservices between hosts based on runtime and historical data. The affinity among microservices, as well as their resource utilization (e.g., CPU and memory), is determined dynamically and incorporated when adaptions to the placement of microservices are made. Results show that a performance improvement of up to 62 % can be achieved using this approach, as well as saving up to 40 % of hosts used. Furthermore, they evaluate two different optimization techniques: A heuristic algorithm based on First-Fit approximation, and solving the optimization problem using a SAT solver, which is guaranteed to find an optimal placement. The latter is unable to work with scenarios composed of more than 20 microservices.

An earlier approach by Chen et al. shows that co-locating VMs based on network communication dependencies achieves performance gains of 16.3 % to 87.6 %, depending on observed performance metric [22].

Mann explores the interplay of VM placement (i.e., the decision of an IaaS provider where to place VMs) and VM selection (on which VM to place application components by users) [57]. The problem is formulated from the IaaS provider's view and aims for minimized energy consumption. It maps applications to VMs, which in turn are mapped to PMs, and respects not only PMs' capacity constraints and energy consumption, but also applications' co-location constraints, hardware affinities, and license cost. The NP-hard problem is solved with an approximation using simple packing heuristics. The combined optimization of VM selection and placement proposed by that approach shows significantly improved results over considering the two problems in isolation.

Guerrero et al. propose to optimize the deployment of microservices using a Genetic Algorithm [38]. The presented approach aims at optimizing cost, availability (i.e., by recovering from VM failures), and network latency among microservices by taking into account the distance between VMs and data centers in a multi-cloud setup. Therefore, scalable numbers and sizes of VMs, and variable numbers of containers are considered in their implementation based on a Non-Dominated Sorting Genetic Algorithm II (NSGA-II). The approach shows significant improvements compared to a greedy First-Fit algorithm.

However, specific communication affinities, as well as differently sized containers for microservices are not considered.

3.2.5 Further Approaches

Further fields for optimization have been examined. For instance, Mastelic and Brandic [59] surveyed approaches to reduce energy consumption in cloud data centers, e.g., by exploiting hardware sleep modes and reducing overhead of shared services such as monitoring and virtualization using energy-aware software layers in place. Stankovski et al. [78] present an approach for a Distributed Adaptive Container Architecture based on Kubernetes, which aims at elasticity for time-critical applications (e.g. gaming, video conferencing). It applies edge computing concepts and optimizes resource allocations to maintain desired QoS requirements with regard to geographic locations. Van Den Bossche et al. [83] address the resource allocation of deadline-constrained bag-of-tasks applications in hybrid clouds. The proposed approach decides whether VMs can be scheduled on an organization's private infrastructure or need to be offloaded to public cloud providers, by taking into account task execution deadlines as well as network bandwidth constraints and cost. A later approach proposed by Mao et al. [58] employs a Genetic Algorithm for solving bag-of-tasks scheduling problems.

Most proposals for optimization of elasticity rely primary on reactive techniques. Notably, an early proactive approach for auto-scaling cloud infrastructure with regard to QoS has been made by Fernandez et al. [29] by profiling heterogeneous infrastructure resources, and predicting network traffic using time series analysis like regression techniques. An approach to scale containers based on network traffic prediction is studied by Kim et al. [49].

A framework for Application Oriented Docker Container resource allocation (AODC) by Guan et al. [37] aims at replacing VMs by containers from a cloud provider's perspective: Optimized for energy consumption and cost, it allocates Docker containers over PMs in data centers. They conclude that their approach for container placement significantly outperforms the placement of hypervisor-based VMs on PMs.

Cardellini et al. investigate the placement of operators in distributed Data Stream Processing (DSP) applications. In their approach, scheduling of operators (i.e., nodes that process elements of a data stream) is optimized for QoS in a decentralized scenario, by taking advantage of the increasing presence of edge/fog computing resources. Therefore, they come up with a general ILP formulation for the Optimal DSP Placement problem (ODP), and apply it in a scheduler for the Apache Storm[7] stream processing framework [18]. Availability, processing speed and resource utilization are taken into account, as well as data flows between operators, and network delays between computing nodes. In their further work, Nardelli et al. propose several model-based and model-free heuristics as an extension to their ODP approach, to overcome the limitations of the NP-hard ILP formulation [67].

[7] `https://storm.apache.org`

Table 3.1: Comparison of related work approaches

Approach	Scaling	Scope	Affinity	Distance	Optimization
ElasticDocker [2]	Vertical	Container	No	No	Rule-based
HyScale [54]	Hybrid	Container	No	No	Rule-based
EVCD [68]	(Horizontal)	Container, VM	No	No	ILP
Four-Fold [44]	Hybrid	Container, VM	No	No	ILP
ACD [69]	Hybrid	Container, VM	No	Yes	ILP
REMaP [75]	–	Container	Yes	No	SMT/First-Fit
NSGA-II [38]	(Hybrid)	Container, VM	No	Yes	Genetic Algorithm
ODP [18, 67]	–	–	Yes	Yes	ILP

3.3 Comparison

After presenting different approaches to elasticity, container placement and optimization problems in the previous sections, we now compare the discussed approaches. Considering the approaches which are most relevant to our work and seem most promising regarding support for a specific feature, Table 3.1 provides an overview of supported features by each approach.

We examine related approaches along the following aspects:

- **Scaling** indicates in which dimensions auto-scaling is applied, i.e., horizontal, vertical, or hybrid scaling.

- **Scope** states whether containers, VMs, or both are subject to placement and scaling decisions.

- **Affinity** denotes whether the degree of network communication between interconnected applications is taken into account for co-location, by means of placing affine resources together as close as possible (e.g., in case of microservices that shape a graph of synchronous network communication between them).

- **Distance** in terms of latency or hops between communicating nodes in a network graph. I.e., approaches that consider interaction between applications which are distributed over geo-distributed computational resources. This is particularly related to distributed applications in hybrid clouds.

- **Optimization** denotes the optimization technique applied to solve the placement or optimization problem. This can be exact optimization techniques like ILP or Satisfiability Modulo Theory (SMT), or heuristics such as First-Fit, rule-based algorithms or Genetic Algorithms.

Hybrid scaling for both containers and VMs is only applied by the "Four-Fold" approach [44] and ACD [69]. HyScale [54] features hybrid scaling for containers, but does not consider scaling of VMs. Other approaches employ horizontal or vertical scaling exclusively, or optimize the initial placement decision for container instances only (thus, not accounting for adaptations at runtime to accommodate changing workload requirements). To place a set of given containers, EVCD [68] applies horizontal scaling for VMs only.

ACD [69] and NSGA-II [38] consider connectivity graphs, i.e., whether components depend on each other, while the aspect of specific degrees of communication affinity among microservices is only addressed by REMaP [75]. However, container co-location in REMaP is limited to the decision of placing affine containers on the same VM, and does not cope with network distance between VMs or data centers. REMaP and NSGA-II do not apply elastic scaling actions to account for workload changes at runtime. ACD is the only approach to consider network distance combined with actual elastic scaling of containers, but does not consider specific communication affinities. The affinity of network communication between nodes together with their distance is only regarded by ODP [18], which is however only applicable for stream operator placement and does not cope with microservices.

The compared approaches apply both heuristic algorithms to solve their scheduling and optimization problems, as well as exact optimization approaches. In case of the latter, ILP is the most commonly used method in related works, though these approaches show major drawbacks when it comes to solving large problem instances, i.e., when dealing with larger numbers of microservices, VM instances or containers. Purposes of optimization in the observed works are manifold, however, most approaches focus on resource utilization, cost and/or performance in their optimization.

3.4 Research Challenges

In light of the increased adoption of microservices and evolving methodologies for container deployment in the cloud, elasticity faces a remaining open challenge in the combination of sophisticated auto-scaling techniques and optimized co-location for affinity among microservices. To successfully deliver cloud services within and across data centers, it is vital that network communication between service components occurs in an efficient and scalable manner [15].

So far, no exhaustive attempts have been made to incorporate co-location aspects into approaches for elasticity. While some approaches implicitly employ co-location principles with regard to resource utilization, i.e., when respecting CPU and memory consumption for container placement [44], exploiting co-location for other modes of affinity has not been regarded to a large extent. To the best of our knowledge, no existing approach exploits co-location of containers with regard to affinity and latency, combined with elastic auto-scaling of provisioned resources.

With regard to container elasticity, current research mostly focuses on the scaling of container instances, leaving scaling on the VM level out of consideration. Very few works follow an approach for hybrid scaling of both containers and VMs, which is considered a relevant factor when scaling many granular, containerized applications, as it is typically seen in a microservices setting.

Furthermore, the role of hybrid clouds with respect to elasticity of microservices has not been covered in depth so far. Most elasticity proposals support only single cloud deployments, and most container orchestration frameworks, as well as managed container services in the cloud, do not allow to operate across multiple clouds and data centers [3]. This poses open challenges, for example, with regard to cloud bursting, where computational resources are scaled out to a public cloud when an on-premise private cloud is used to capacity. It also stands to further explore co-location principles in a larger scope: As containerized applications in hybrid clouds are distributed over multiple data centers, they are supposed to benefit from strategies that favor placement of interacting containers within the same data center. Of the relevant existing works, only NSGA-II [38] considers the network distance between data centers and VMs for the provisioning of microservices.

A common challenge of NP-hard scheduling and optimization techniques applied in existing approaches remains in potentially long resolution times, causing scalability issues. Therefore, efficient heuristics need to be developed, which overcome these limitations for large problem instances. Existing works mostly propose exact formulated, NP-hard problems or simple greedy algorithms to achieve hybrid scaling. Therefore, the use of Genetic Algorithms requires further research, and represents a viable alternative for our work.

CHAPTER 4

Design

In this chapter, we define the general requirements of our approach, describe the design and system model for the proposed scheduling middleware, and provide the formulation of the optimization problem at its core.

4.1 Preliminaries

This work aims at developing an approach for the elastic deployment of microservice architectures, while specifically taking into account the aspect of container co-location based on microservices' network interaction and the accrued request latencies in hybrid cloud infrastructures. In doing so, we will partially incorporate concepts from existing approaches discussed in the previous chapter, and adapt, combine or enhance them. In this regard, the "Four-Fold" approach [44], ACD [69], REMaP [75] and the NSGA-II approach [38] are most relevant to our work. The general preliminaries and combined requirements for our approach are stated in the following sections.

4.1.1 System Requirements

The general requirements for our middleware are outlined in Table 4.1, with all defined requirements having direct implications on the optimization model at the core of our approach.

Requirements 1.1 and 1.2 reflect the properties of Four-Fold scaling, i.e., by employing hybrid scaling for both containers and VMs as a fundamental principle. By adopting a holistic approach to resource allocation decisions, we aim to facilitate improved efficiency in auto-scaling of elastic microservice architectures. Relevant previous work on that topic has been conducted in [44] and [54].

Requirements 2.1 and 2.2 reflect our goal to support multiple data centers in a hybrid cloud scenario, including private (on-premise) data centers. This allows us to consider

Table 4.1: System requirements

Four-Fold scaling

RQ 1.1	Both horizontal and vertical scaling actions are considered for provisioning and adapting VM resources.
RQ 1.2	Both horizontal and vertical scaling are considered for placing and adapting container workload.

Hybrid clouds

RQ 2.1	The system considers multiple data centers and the specifics of their resource capabilities, which constitute the pool of available VM resources.
RQ 2.2	The system has a notion of on-premise data centers (i.e., private clouds).

Efficient resource allocation

RQ 3.1	The overall cost of leased VM resources should be minimized.
RQ 3.2	It must be ensured that the load incurred for each microservice can be handled and QoS constraints are fulfilled.
RQ 3.3	When placing containers, the aspect of already cached container images should be considered to gain QoS benefits.

Co-location of interaction-affine microservices

RQ 4.1	Interaction-affine containers should be co-located to reduce network round-trips across VMs or data centers.
RQ 4.2	The distance (in terms of network latency) between data centers and VMs should be considered to reduce latency in inter-service communication.

public–private hybrid clouds and the associated cost structure in our approach, together with the aspect of cloud bursting (cp. Section 2.1.3). In existing approaches, hybrid clouds have not been regarded to a large extent in the context of microservice elasticity.

Requirements 3.1 and 3.2 reflect the common objective of elastic provisioning which is to optimize for cost efficiency while sustaining QoS constraints, i.e., by determining an equilibrium state between over- and under-provisioning (cp. Section 2.3.4). As cost is incurred by running VMs, leased resources should be kept at a reasonable minimum. However, there must be enough computational resources available at any time to meet the microservices' demand. Additionally, we want to incorporate the aspect of container image caching in Requirement 3.3: Placing containers on VM instances with their image already cached provides benefits in a way that container startup time is reduced, which allows contributing to QoS more quickly since the download of the container image can be skipped.

Requirement 4.1 accounts for a main goal of this thesis, which is to exploit the co-location of microservice instances. While in traditional approaches to auto-scaling, microservice instance are usually placed according to resource-centric information such as available VM resource and their cost, our approach additionally aims at improved performance of interaction-affine microservices. To conquer this challenge, the proposed solution needs to find a resource configuration which places affine microservices on the same, or a nearby VM instance (e.g., within the same data center), resulting in reduced latency of calls to a downstream service. For this purpose, the distance in terms of network latency between VM resources should be considered (Requirement 4.2).

Affinity between microservices arises from internal network interaction which is related to microservice patterns such as API gateways (cp. Section 2.4.3), or due to plain inter-service calls by dependent microservices. The distance between two endpoints is defined as the network latency which may depend on factors such as geographical distance and bandwidth. Related approaches covering similar requirements have been proposed in [75] and [38], and with the limitation to maximum delay constraints in [69], respectively.

4.1.2 Cost and Billing

CSPs charge varying cost for leased VMs depending on their type and capacity, with “larger” VMs (i.e., those with higher CPU and memory capacity) typically being more expensive in total, however their relative cost per resource unit may be decreasing compared to smaller ones. We can gather representative cost of current VM prices from the major CSPs’ pricing details, as published by AWS[1], Microsoft Azure[2] and Google Cloud[3].

The most relevant existing approaches discussed in Section 3.3, such as [44], account for fixed billing time units (BTUs) of VM resources. In this case, VMs have to be fully paid, e.g., for the duration of one hour, even if resources are released before the end of such BTU. This was common for most CSPs until a few years ago, when billing policies were relaxed and such obligations have been dropped. For instance, AWS moved from per-hour billing to per-second billing for their Linux-based VMs in 2017 [9]. Accordingly, in our approach billing is assumed to be exact on a per-second basis, respecting irregular periods of runtime by measuring the elapsed time between the launch and termination of a VM instance. This billing policy falls into place with the common on-demand VM pricing policies of major CSPs. I.e., AWS (applying per-second billing for Linux-based VMs only) [6], Google Cloud [36], and Microsoft Azure (billing only full minutes, not charging for extra seconds) [65] currently apply a widely similar billing model.

[1]`https://aws.amazon.com/ec2/pricing/on-demand`
[2]`https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux`
[3]`https://cloud.google.com/compute/vm-instance-pricing`

However, some CSPs apply a minimum charge for used VM instances: E.g., AWS and Google Cloud always bill the full first minute. This minimum billing period is considered negligible as to the grace period (which will be introduced in Section 4.2.6) of our approach being greater than one minute, and thus a VM would never be released prior to reaching the minimum charge.

4.1.3 Network Communication

The focus of the approach conducted in this thesis with regard to Requirement 4.1 lies on improving communication performance of microservices at runtime. This implies a notion for network requests of different type and origin, i.e.:

(a) Requests to the system which are received from external clients (originating from sources outside the microservice architecture).

(b) Requests resulting from other microservices within the system, i.e., inter-service communication of interacting services ("downstream service calls").

In both cases, we limit the scope of our approach to synchronous request–response communication (e.g., HTTP-based Web requests and API calls). While asynchronous messaging (e.g., via message queues) is a viable option in many use cases, particularly for inter-service communication, we do not take it into account as a factor for service interaction and, ultimately, co-location. The rationale behind this decision lies in asynchronous messaging delays bearing significantly less impact on performance, since temporal decoupling between senders and receivers is often intentionally accepted and goes hand in hand with the main advantages of this pattern (cp. Section 2.4.2). However, if performance of asynchronous messaging should be considered, this could be taken into account by treating message brokers as another microservice, which exchanges messages with other publishing and subscribing microservices. The aspect of synchronous and asynchronous communication in our approach is represented likewise by REMaP [75].

Furthermore, any applications deployed as part of the microservice architecture are assumed to be stateless, in the sense that clients do not require "sticky" sessions where subsequent requests need to be handled by the very same service instance.

4.1.4 Load Balancing and Service Discovery

As stated in the previous section, there are two kinds of network traffic, which need to be routed to their destinations accordingly (i.e., to container instances hosting a destination service). In order to take advantage of the benefits of co-location, proper load balancing strategies (see Section 2.4.3) must be applied. We assume the following load balancing infrastructure to be deployed with the microservice architecture:

- For inter-service requests, client-side service discovery is leveraged. The client service looks up available target service instances from a registry and subsequently issues the request directly to the most suitable (i.e., the closest instance capable of handling additional load). Thus, any round-trips via dedicated load balancers are eliminated which enables exploitation of co-location. Multiple widely-used solutions provide the means to deploy a microservice architecture using such a "service mesh", e.g., Istio[4] and HashiCorp Consul[5].

- External requests are first handled by a dedicated load balancer infrastructure, which forwards them to a suitable destination container instance according to the targeted service. This may be any instance capable of serving additional requests, and might be determined by the load balancer in a round robin fashion. Since external clients are not part of the service mesh, they rely on a known entry point, i.e., using the load balancer's DNS name.

4.1.5 Out of Scope

Regarding the capabilities of VMs, our focus lies on computational resources, i.e., CPU and memory. Network bandwidth is supposed to be sufficiently available on any VM type and thus not regarded for the means of our approach. The same premise applies to storage on VMs as well (i.e., volume types and size of disk storage), which is considered negligible due to standardized OS images and compact microservices with Docker containers showing a small footprint of downloaded container images. Since our approach is based on the virtual resources of the IaaS layer, we do not account for any physical hosts (i.e., PMs).

Our approach does not facilitate any kind of runtime migration, i.e., moving running container instances between VMs. The main reason for this is that current cloud services and container technologies have no (built-in) support for live migration techniques. Furthermore, an approach specifically covering the aspect of runtime migration is proposed by ElasticDocker [2].

Regarding networking, asynchronous communication patterns are not explicitly taken into account, as lined out in Section 4.1.3. Moreover, the supporting infrastructure for load balancing and service discovery as described in Section 4.1.4 is considered an abstract concept, which is neither part of the microservice architecture nor reflected in our system model.

[4]`https://istio.io`
[5]`https://www.consul.io`

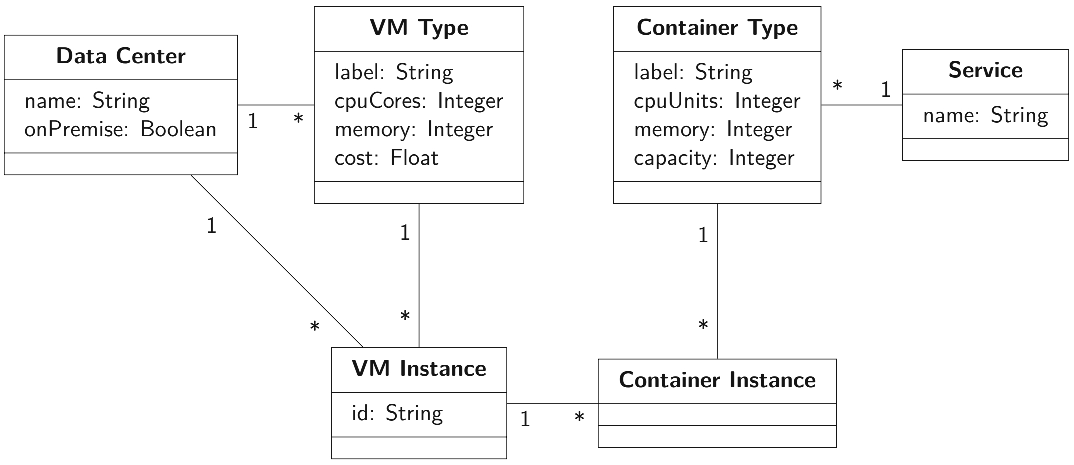


Figure 4.1: Domain model

4.2 System Model

The following sections describe the abstractions and concepts used to constitute the system model of our approach.

4.2.1 Domain Model

As shown in Figure 4.1, the system's domain model has a notion of multiple entities which account for the available cloud infrastructure and the configuration of microservices which are supposed to be elastically deployed by our middleware.

First of all, *data centers* provide the means for provisioning VM resources, which constitute the infrastructure (IaaS) layer. A data center is a physical and logical unit providing computational resources, and may be associated with different service providers (i.e., public CSPs or private clouds). Each data center offers an arbitrary number of *VM types* of different characteristics, and thereof arbitrary numbers of *VM instances* may be provisioned. VM types feature specific properties with regard to resource capacity (i.e., computational and memory capacity) which constrain the workload placed on a VM instance. Each VM type is specified with a number of CPU cores available (e.g., single- or dual-core equipped machines). Memory is specified in megabytes available to use for container workload. Furthermore, each VM type shows individual cost, which is defined as the hourly rate which is billed for each leased VM instance. While in the real world a VM instance is newly created when it gets leased, and destroyed on its termination, we assume in our abstraction that a fixed, arbitrary large pool of VM instances is available in each data center, in order to resemble a finite set which is feasible for optimization. In case of on-premise data centers (i.e., in a private cloud), the number of VM instances is limited according to the actually available resources.

From an application's view, each microservice is regarded as a logical entity which we refer to as *service*. For each service, we assume a set of predefined container types to be known. A *container type* is a specific configuration for Docker containers of a certain service with fixed parameters for resource consumption and capacity, with attributes for CPU usage (in terms of units as defined in the subsequent section), memory consumption in megabytes, and the number of network requests which can be processed, representing the container's capacity with regard to QoS.

Docker containers are the deployment unit for all services. A *container instance* resembles an actual container at runtime, hosting a single microservice instance. Multiple container instances of the same type can be allocated on different VM instances. Likewise, a VM instance may host several container instances of various types.

CPU Units

We treat CPU cores as the common unit for computational resources across VM types, with equal processing power each. The CPU requirements of container types are denoted in absolute *CPU units*, which are defined in terms of shares per CPU core, regardless of the VM's size or the number of running containers.[6] The processing power of a single CPU is defined to resemble a contingent of 1024 units. For instance, a container type which is specified to require 512 CPU units, means that the corresponding container instance would use 25 % of the processing power of a dual-core VM. Such concept of CPU units is similarly employed by AWS ECS [30].

Discrete Container Types

Types of containers and VMs are discretized in our model in a way that it involves a set of distinct, enumerated resource types with fixed predefined resource capacity each. While this is common for the latter in real CSP offerings (i.e., choosing VMs from a set of distinct types), containers offer much more granularity, as CPU and memory capacity can be freely specified. However, we aim to simplify the search space for our optimization problem, and thus follow the approach from [44] by employing discrete container types, which provide a trade-off between granularity and the optimization's complexity and accuracy. Thus, the configuration of deployed container instances is to be selected from the set of defined container types.

Furthermore, the notion of distinct container types in addition allows us to recognize a container's capacity as an abstract, fixed quantity, such that the load each container type is capable to handle with respect to QoS can be empirically predetermined. E.g., the amount of requests a given container type is able to handle, is determined during load tests. Again, although a non-predetermined approach would be more concise, as done in [54], we accept this trade-off in return for simplicity.

[6] As opposed to the concept of relative CPU shares in Docker, which act as a weighting factor for all containers running on a host [26].

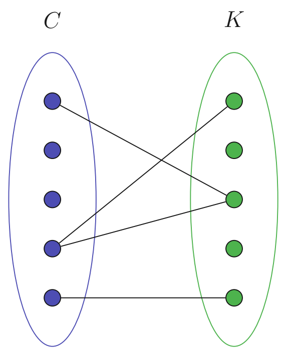


Figure 4.2: Resource allocation graph

4.2.2 Resource Allocation

In our model, the state of container placement and resource provisioning is represented in a many-to-many mapping relation between container types and VM instances. As depicted in Figure 4.2, this mapping resembles a bipartite graph $G = (U, V, E)$, where vertices U are the set of container types $c \in C$ and vertices V are the set of VM instances $k \in K$. Each edge $\{c, k\} \in E$ constitutes a container instance of type c which is placed on VM instance k. Furthermore, the graph is not weighted, and thus only accounts for placing one instance of each container type on the same VM. VM instances without edges are not used for container placement and thus considered unused (and consequently, not leased, given a VM is not located in an on-premise data center). We refer to such container–VM mapping configuration as *resource allocation*.

After all, this model achieves Four-Fold scaling as to Requirements 1.1 and 1.2, by employing hybrid scaling for both containers and VMs. Vertical scaling is accomplished by selecting different types of containers/VMs for resource allocation, while horizontal scaling is employed when adding or removing container/VM instances.

4.2.3 Distance

As stated by Requirement 4.2, the system needs to be aware of the distance (in terms of network latency) between VM instances in order to optimize for co-location of containers. We abstract the distance between VMs, so that VM instances are clustered according to the data centers in which they are located (Requirement 2.1), resulting in an undirected graph with vertices representing data centers, and edge weights representing latency in milliseconds. To account for latencies within each data center, the graph permits loops. The example in Figure 4.3 shows a distance graph between two data centers (outer vertices), along with provisioned VM instances (inner vertices). The distance between any pair of VM instances can easily be determined by looking up their data center's distance from an adjacency matrix D. I.e., a further graph for the distances between VM instances $\{k, l\} \in K$ can be derived. The diagonal of D contains the latency which

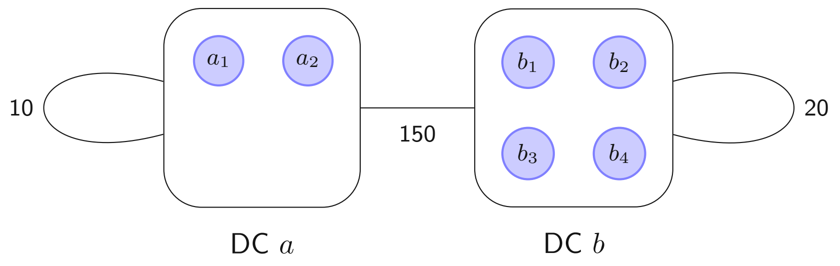


Figure 4.3: Distance graph of network latency between data centers

arises for network calls within a data center, hence determines the distance between all VM instances hosted in the same data center.

4.2.4 Load and Capacity

Each microservice in the system faces load in terms of requests, e.g., requests per second (RPS), which need to be handled. The *total load* each service s needs to handle is composed of requests that arrive from external clients (*external load*, L_s^E) and requests originating from other services within the microservice architecture, the *internal load* (L_s^I), as shown in Equation 4.1:

$$L_s = L_s^E + L_s^I \tag{4.1}$$

Let S be the set of services in the microservice architecture. The *total system load* L is defined in Equation 4.2 as the sum of the total load accrued for each service:

$$L = \sum_{s \in S} L_s \tag{4.2}$$

Service capacity refers to the load a service can handle with regard to QoS, according to the system's resource allocation state. It is determined by the sum of capacity supplied by each allocated container instance of a service. Service capacity and load determine the current state of the system with respect to over- and under-provisioning. Let Q_s be the calculated service capacity of a service s, then $Q_s < L_s$ indicates that the service is under-provisioned.

4.2.5 Service Interaction and Affinity

Interaction Graph

Our model represents the aspect of network interaction (i.e., "hops") between services in a directed, acyclic graph. Its nodes correspond to the services in the system, while edges represent the number of invoked downstream service calls from one service to another. Figure 4.4 shows an example for an interaction graph consisting of three services, i.e.,

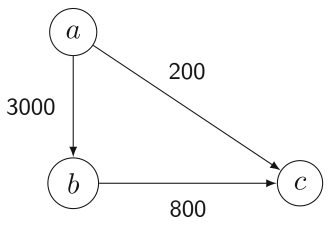


Figure 4.4: Service interaction graph

$S = \{a, b, c\}$. The interaction graph corresponds to an adjacency matrix I, referred to as *interaction matrix*, which holds the number of issued requests $i_{s,t}$ for each pair of services $s, t \in S$. As services are supposed to never call themselves using synchronous network requests, the diagonal of the matrix always resembles zero. The following interaction matrix corresponds to our example graph:

$$I = \begin{pmatrix} 0 & 3000 & 200 \\ 0 & 0 & 1000 \\ 0 & 0 & 0 \end{pmatrix}$$

Interaction Multiplication

In a simulated scenario, we want to use assumptions as to how likely a service is to make calls to another downstream service. For this purpose, we introduce a further adjacency matrix I^M which holds multiplication factors $i^M_{s,t}$ for each edge $\{s, t\}$ in a graph of services. Given I^M and the (simulated) external load L^E_s arriving at each service, we can infer both I and L^I_s. By traversing all paths of the graph, values of the interaction matrix I as well as the internal load for each service, L^I_s, can be accumulated while taking into account I^M.

Figure 4.5 shows the interaction graph including multiplication factors and the determined values labeled in blue. Each factor denotes the number of induced downstream service calls based on a service's total load L_s. I.e., given $i^M_{s,t} = 0.5$, every second request arriving at service s is assumed to trigger a downstream service call to t. The corresponding adjacency matrix I^M of interaction factors is represented as follows:

$$I^M = \begin{pmatrix} 0 & 1.5 & 0.1 \\ 0 & 0 & 0.25 \\ 0 & 0 & 0 \end{pmatrix}$$

Cyclic Dependencies

In return for abstraction, we do not differentiate each service request's nature. I.e., the system has no notion of varying behavior for the individual requests arriving at a service.

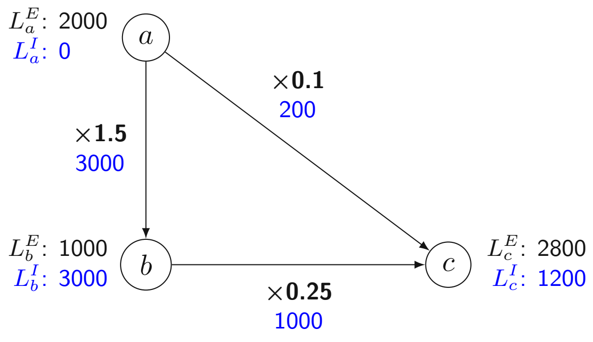


Figure 4.5: State of service interaction

Hence, cyclic dependencies of communication between services are invalid in our model of service interaction. Otherwise, cyclic interaction would cause an infinite loop of calls between two services, due to our concept of interaction multiplication, which issues an equal fraction of downstream calls for any incoming calls to a service. Yet in a real-world microservice architecture, bidirectional interaction would be feasible under the premise that a sequence of exchanged calls itself contains no cyclic dependencies.

Affinity

Similar to the approach in [75], we regard affinity as a bidirectional relationship between two services, which is represented in an undirected weighted graph. The corresponding adjacency matrix A contains the affinity $a_{s,t}$ for any pair of services, which is calculated in Equation 4.3 as follows:

$$a_{s,t} = \frac{i_{s,t} + i_{t,s}}{L} \tag{4.3}$$

where $i_{s,t} + i_{t,s}$ is the number of exchanged messages regardless of direction between services s and t according to the interaction matrix I, and L is the total load facing the system (cp. Section 4.2.4).

Given the example interaction state depicted in Figure 4.5, where $L = 10\,000$, $i_{a,b} = 3\,000$, $i_{a,c} = 200$, and $i_{b,c} = 1\,000$, the affinity matrix A corresponds to:

$$A = \begin{pmatrix} 0 & 0.3 & 0.02 \\ 0.3 & 0 & 0.1 \\ 0.02 & 0.1 & 0 \end{pmatrix}$$

4.2.6 Grace Period and Startup Times

To prevent temporary under-provisioning, and thus degradations in QoS or impaired availability, reconfigurations of resource allocation must be carried out gracefully. Specifically, when transitioning to a new resource allocation, previously running but revoked resources (i.e., terminated container instances and VMs hosting those containers) need to continue operation until the newly provisioned resources are ready to accept requests (and thus contribute to the service capacity).

The issue of not implementing such a mechanism is illustrated in the following scenario: Given two container instances of a service s are deployed on two VM instances. According to a newly optimized resource allocation, one of those containers will be deprovisioned. Instead, another container instance will be placed on a VM instance which is not leased yet. In a mode of immediate reconfiguration, where the revoked container instance would be terminated at the same time as initiating the launch of the newly provisioned VM and container instance, the service s would suffer from under-provisioning. As the new VM and container instance are not immediately available, but take some time to start up, the service capacity of s is possibly compromised (cp. Section 4.2.4).

We call the timespan during which any resources need to be kept running, until being deprovisioned according to the prospective resource allocation, the *grace period*. As defined in Equation 4.4, the grace period G is composed of three constants:

$$G = T^K_{start} + T^C_{start} + T^C_{stop} \tag{4.4}$$

A central factor is the time it takes from launching a new VM instance until it is ready for use. This *VM startup time* (T^K_{start}) varies between CSPs, VM instance types, operating systems and other factors such as data center location and time of day. Abrita et al. conducted a study showing that the startup times of VMs across AWS, Microsoft Azure and Google Cloud is on average 18 to 37 seconds, depending on the instance type [1]. Therefore, we rely on the conservative assumption that $T^K_{start} = 60\,\mathrm{s}$.

The *container startup time* (T^C_{start}) represents the duration it takes from initiating the start of a container instance until the hosted service reaches a steady state in which it is able to handle incoming requests. In our assumption, microservice containers show a small foot-print in terms of container image download size. While studies show that microservice containers boot within a few seconds including image loading [4], plus – depending on the underlying tech-stack – a startup latency of less than 10 seconds for applications to become responsive, as seen for simple JVM-based applications [79], we assume a conservative startup time of 30 seconds after which microservice instances are ready to accept requests.

After VMs and containers launched successfully (i.e., $T^K_{start}+T^C_{start}$ elapsed), the remaining revoked resources are no longer required to serve requests. However, services require

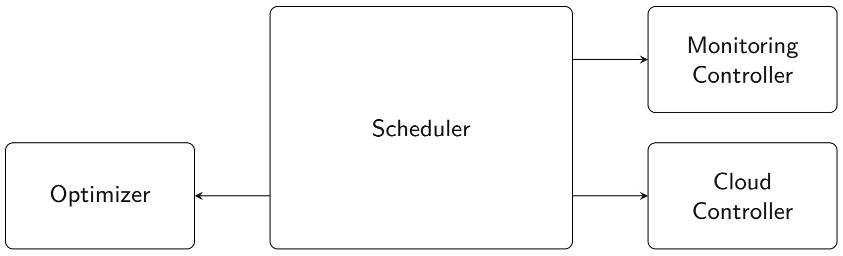


Figure 4.6: Architecture overview

some buffer to complete any ongoing workload of recent requests. For this purpose, we assume another 30 seconds for the *container draining period* (T^C_{stop}), in order to shut down container instances gracefully. Importantly, no new requests will be routed to a container instance during that period. Afterwards, revoked containers are stopped and related VMs can be released.

Summarizing, based on the assumptions made, we define a two-minute grace period:

$$G = 60\,\mathrm{s} + 30\,\mathrm{s} + 30\,\mathrm{s} = 120\,\mathrm{s} \tag{4.5}$$

4.3 Scheduling Middleware

A central middleware is responsible for carrying out the optimization and all related tasks in order to achieve our approach of elasticity for microservice architectures deployed in the cloud. This section provides an overview of its proposed architecture and mode of operation.

4.3.1 Architecture

The middleware's architecture is designed for coordination of the necessary tasks between multiple components and cloud providers. At its core, a scheduling loop continuously monitors the system state and eventually performs reconfigurations based on an optimized resource allocation. As illustrated in Figure 4.6, the overall architecture is composed of four components fulfilling different responsibilities:

Cloud Controller This component implements the necessary means to provision and deprovision VMs in cloud data centers, and to place and manage Docker containers. Multiple controllers following a common interface may be implemented specificially for targeting different CSPs.

Monitoring Controller A component collecting and aggregating data at runtime which is necessary to determine the system state with respect to load, service interaction and resource allocation. For this purpose, the Monitoring Controller may need to communicate with both CSPs (e.g., polling VM statuses, retrieving

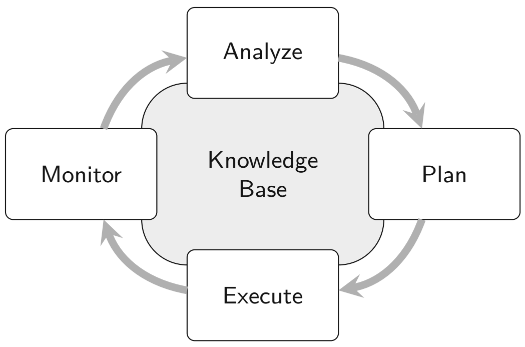


Figure 4.7: MAPE-K loop

request metrics from load balancers) and microservice applications or service mesh capabilities directly (i.e., for collecting insights about inter-service communication). As stated in Section 4.1.4, the latter may be provided by well-established solutions for client-side load balancing and service meshes.

Scheduler The Scheduler operates in a loop to continuously assess and adapt the system state for an optimal resource allocation according to our requirements. Therefore, it is initialized with the cloud providers' infrastructure and the microservice architecture at hand, according to our system model. Subsequently, it interacts with the Cloud and Monitoring Controller, and eventually calls the Optimizer to run an optimization.

Optimizer The Optimizer performs the computation to find a new resource allocation optimum. To employ different optimization methods, multiple implementations of the Optimizer could be used interchangeably.

4.3.2 Scheduling Loop

In this section we describe the Scheduler component, which carries out the central scheduling loop, in detail.

As an essential design principle, it follows the typical *MAPE-K* loop as depicted in Figure 4.7, which is commonly employed by self-adaptive systems [47]. While autonomously executing the steps of ***m**onitoring*, ***a**nalyzing*, ***p**lanning* and ***e**xecution*, the system relies on a central, shared ***k**nowledge base*, which maintains all necessary information about the system state. In our approach, the knowledge base comprises the system model with respect to cloud infrastructure and microservice architecture, the state of the system with regard to load and interaction, and the determined current and target resource allocations.

Scheduling cycles are to be executed in reasonable intervals. Based on the grace period and resource startup times defined in Section 4.2.6, we choose a cycle run every 30 seconds as a sensible frequency.

The components representing the steps of our MAPE-K-based scheduling loop are described in the following sections.

Monitor

In this step, the middleware collects all the data which is needed to calculate the current system state, and stores it in the knowledge base. For this purpose, the current load the system is facing gets measured, i.e., the Monitor records the number of requests incurred for each service regarding both external and internal load. In a real-world scenario, L_s^E would be polled from the load balancer(s) routing traffic from external clients. The interaction matrix I, and consequently the internal load L_s^I of each service, would be gathered from the network logs of the service mesh, which keeps track of all network communication between any containers.

In the approach of our thesis, those measures will be simulated. Hence, we continuously provide the Monitor with the external load L_s^E for each loop cycle, as well as a given interaction multiplication matrix I^M which reflects our assumptions of the microservices' interaction behaviour. Consequently, I is calculated based on L_s^E and I^M (cp. Section 4.2.5).

Additionally, the Monitor retrieves the current deployment state from the cloud providers, i.e., the list of leased VM instances and running containers, as well as their statuses. Having detected the currently running, healthy container instances, the service capacity Q_s for each service can be determined.

Analyzer

Derived from the gathered measures by the Monitor, the Analyzer determines the aggregated load per service, L_s, and the affinity matrix A, which is calculated based on the interaction matrix I and the total load L. Additionally, the Analyzer compares the currently supplied capacity of each service to its load, to determine the under-provisioning state for each service, which is indicated by $Q_s < L_s$.

Furthermore, the Analyzer inspects the validity of the current resource allocation in place. For this, it first compares the current system state to the latest optimization's underlying system state, in order to detect significant changes with regard to load or capacity. Second, the optimization objective function is evaluated against the current system state (determining the current "fitness") and compared to the results of the latest optimization's resource allocation. The resulting drift provides an indication whether the current resource allocation is still valid with regard to optimization, or if it would be worthwhile to perform a new optimization. Moreover, the current system state is validated for any constraint violations such as under-provisioning.

Finally, the gathered KPIs contributing to the system state and planning decisions are stored in the knowledge base.

Planner

The Planner is required to follow a different set of actions depending on the system's state in the current scheduling cycle. I.e., consecutive optimizations require at least an elapsed period of 120 seconds in between, according to our definition of the grace period (cp. Section 4.2.6). Accordingly, the Planner operates in one of the following two modes.

Performing an optimization. As a precondition, a new optimization is only considered when the system is not awaiting the grace period of an ongoing reconfiguration.

That said, the Planner decides in addition whether to entirely skip an optimization based on the drift of the system state since the previous optimization, as assessed by the Analyzer. Therefore, we define two parameters with regard to drift tolerances that allow for skipped optimizations. The first is the *load drift tolerance*, T_L, which defines the acceptable drift in service load (evaluated per service) since the last optimization. If the analyzed drift exceeds T_L, a new optimization is forced. Furthermore, the Planner considers a *fitness drift tolerance* (T_F): If the system's current deviation from the last optimization's fitness lies within this range, a new optimization may be omitted. Any violations of constraints (e.g., in case of degraded QoS due to under-provisioning) require a new optimization, however.

If indicated, a new optimization is performed based on the current system state, and the resulting optimized resource allocation is stored in the knowledge base. Subsequently, the Planner determines which resources of the current resource allocation are to be abandoned (i.e., VMs and container instances scheduled for termination) by comparing it with the optimized resource allocation. For any abandoned resources, the Planner schedules the respective VMs and container instances for future termination in consideration of the grace period, and derives an adjusted target allocation from the optimized resource allocation in order to keep those resources allocated for now. Finally, the resulting target allocation effective for the current cycle is made available to the knowledge base.

Awaiting grace period. If a scheduling cycle is performed while the system is still in the process of completing an ongoing reconfiguration to establish the most recent optimized resource allocation, the system is in a transitioning state where running a new optimization is not feasible. This applies if any resources are waiting to cease grace period, i.e., VMs or containers yet to be terminated which have been scheduled for termination in a previous optimization cycle.

In this case, the Planner decides for each instance individually whether it can be finally released or still needs to pass the remaining time span of the grace period. If any resources have been found to be terminated in the current cycle, the Planner creates a new target allocation off the optimized resource allocation, reflecting the changes in remaining resources scheduled for termination.

Executor

The Executor takes the necessary actions to establish the target resource allocation specified during planning. For this purpose, it compares the target allocation to the current resource allocation in order to determine the delta in terms of resources that need to be additionally provisioned, and those which need to be released (i.e., VM and container instances to launch/terminate).

Subsequently, the following actions are carried out according to the computed delta:

1. Launching newly required VM instances.
2. Newly required container instances are started, given the corresponding VM instance is up and running (i.e., in case of a previously launched VM, if the startup time of 60 s is completed). Otherwise, the launch of the container instance is deferred to a future scheduling cycle.
3. Running container instances, which are to be deprovisioned, are stopped.
4. Deprovisioned VM instances are terminated according to the target allocation.

4.4 ILP Optimization Model

In this section, we formulate the optimization problem to find a resource allocation optimum using an ILP optimization approach. For this, we will write out the definitions, objective function and constraints of the proposed approach in detail.

4.4.1 Variables

Table 4.2 defines the variables representing the overall model of the system used throughout the ILP formulation, including the system's domain entities, assumptions, and weighting factors of different optimization aspects.

Table 4.2: System variables

Variable	**Description**
$s \in S = \{1, \dots, s^{\#}\}$	The set of services in the system is represented by S, while s refers to a specific service thereof.
$c \in C = \{1, \dots, c^{\#}\}$	The set of container types C, with a specific container type represented by c.
$c_s \in C_s \subseteq C$	The set of container types eligible to run a service s is represented by C_s, which is a subset of C, while c_s refers to a specific container type of service s.

Continued on next page $\longrightarrow$

Continuation of Table 4.2

$v \in V = \{1, \dots, v^{\#}\}$	The set of VM types is represented by V, while v denotes a specific VM type.
$k \in K = \{1, \dots, k^{\#}\}$	The set of leasable VM instances in the system is represented by K, while k indicates a specific VM instance.
$k_v \in K_v \subseteq K$	The set of leasable VM instances of type v is represented by K_v, which is a subset of K, with k_v denoting a specific VM instance of type v.
$D = [d_{k,l}]$	The distance adjacency matrix of VM instances is represented by D, with $d_{k,l}$ denoting the distance in terms of network latency between VM instances k and l.
Q_c	The maximum capacity with regard to QoS supplied by a container type c in terms of network invocations.
U_c^{CPU}, U_c^{MEM}	Resource usage of a container type c in terms of CPU and memory, respectively.
R_v^{CPU}, R_v^{MEM}	Resource supplies of a VM type v in terms of CPU and memory, respectively.
P_v	Cost which accrues when leasing a VM instance of type v.
G	Cost premium for VMs entering the grace period.
M	The upper bound of placeable containers on any VM instance.
ω_C, ω_G, ω_L, ω_Q, ω_I	The weights specifying the importance of each term in the objective function as defined in Section 4.6.

As the optimization model requires information about the system state, we define the necessary related variables in Table 4.3. The values of these state variables are determined by the Monitor and Analyzer components.

Table 4.3: State variables

Variable	**Description**
L_s	The accrued load of a service s in terms of network invocations.
$A = [a_{s,t}]$	The adjacency matrix for service affinity is denoted by A, with $a_{s,t}$ representing the specific affinity between services s and t.

Continued on next page ⟶

Continuation of Table 4.3

$Z = [z_{c,k}]$	The adjacency matrix Z represents the previous resource allocation in place, with $z_{c,k}$ indicating that a container of type c has been previously allocated on VM instance k.
β_k	Boolean variable indicating whether a VM instance k is already running. I.e., k has been leased according to the previous resource allocation, or the VM instance is located in an on-premise data center and thus treated equivalently.
$I_{s,k}$	Boolean variable indicating whether the container image of service s is cached on a VM instance k.

As fundamental part of the ILP model, a decision variable for the resource allocation mapping is required. Additionally, further decision variables act as helper variables in our model, as defined in Table 4.4. Since all decision variables are integers, the formulated approach represents an ILP optimization problem, rather than MILP.

Table 4.4: Decision and helper variables

Variable	**Description**
$x_{(c,k)}$	Boolean decision variable representing the prospective resource allocation, with $x_{(c,k)} = 1$ indicating that a container instance of type c is to be placed on VM instance k.
$y_{(k)}$	Boolean helper variable indicating whether VM instance k is leased in the prospective resource allocation (regardless of its state in the previous resource allocation).
$g_{(k)}$	Boolean helper variable indicating that VM instance k will enter grace period for the prospective resource allocation (i.e., the VM instance has been running before, but will be released).
$\lambda_{(c_s,k,c_t,l)}$	Boolean helper variable representing the simultaneous placement of a container instance of type c_s on VM instance k, and a container instance of type c_t on VM instance l. I.e., this variable takes the value 1 if both $x_{(c_s,k)} = 1$ and $x_{(c_t,l)} = 1$.
$\mu^{CPU}_{(s,k)}$, $\mu^{MEM}_{(s,k)}$	Helper variables representing the maximum amount of accumulated resource requirements during the grace period for container instances of a service s running on a VM instance k, with regard to CPU and memory consumption, respectively.

4.4.2 Objective Function

The objective function of our ILP model is defined in Equation 4.6. It consists of five terms, which are subject to be minimized in order to find an optimum.

$$\begin{aligned}
\min \Bigg[& \omega_C \cdot \sum_{v \in V} \sum_{k_v \in K_v} (P_v \cdot y_{(k_v)}) \\
& + \omega_G \cdot \sum_{v \in V} \sum_{k_v \in K_v} \Big((P_v + G) \cdot g_{(k_v)} \Big) \\
& + \omega_L \cdot \sum_{s,t \in S} \sum_{c_s \in C_s} \sum_{c_t \in C_t} \sum_{k,l \in K} (\lambda_{(c_s,k,c_t,l)} \cdot a_{s,t} \cdot d_{k,l}) \\
& + \omega_Q \cdot \sum_{s \in S} \Big(\sum_{c_s \in C_s} \sum_{k \in K} (Q_{c_s} \cdot x_{(c_s,k)}) - L_s \Big) \\
& + \omega_I \cdot \sum_{s \in S} \sum_{c_s \in C_s} \sum_{k \in K} \Big((1 - I_{s,k}) \cdot x_{(c_s,k)} \Big) \Bigg]
\end{aligned} \tag{4.6}$$

The individual terms of the objective function aim to achieve the optimization goals as to our requirements (see Section 4.1.1) by addressing the following aspects.

Term 1: Minimize cost. The first term, i.e., $\sum_{v \in V} \sum_{k_v \in K_v} (P_v \cdot y_{(k_v)})$, ensures cost reduction for the prospective resource allocation as claimed by Requirement 3.1, by computing the total cost. Therefore, the cost P_v of a VM type v is summed up for each leased VM instance k_v, which is indicated by the helper variable $y_{(k_v)}$ taking the value 1. In addition, this term implicitly aims at resource efficiency, by avoiding cost of unused resources. To control the value of this term within the objective function, we use the weight ω_C.

Term 2: Minimize waste in grace period. The purpose of the second term is two-fold: It adds to the goal of cost minimization by reducing the cost which accrues solely for the purpose of keeping abandoned VM instances running during the grace period (thus, VMs not contributing to future resource allocation), while rewarding the reuse of already leased VM instances. I.e., the volatility in movement of container instances to newly provisioned VMs should be reduced. Therefore, we sum up the cost for already leased VM instances k_v that enter the grace period, which is indicated by $g_{(k_v)} = 1$. A constant "premium", G, is added to each abandoned VM's cost P_v, representing the minimum charged even if $P_v = 0$. Thereby, random movements of containers between VMs in private clouds are prevented, in which case the whole term would be ineffective given $P_v = 0$. We choose $G = 10^{-10}$ as a reasonable small cost premium. The term is weighted within the objective function using the weight ω_C.

Term 3: Minimize latency by co-location. This term contributes directly to Requirements 4.1–4.2, by favoring the co-location of affine services on close VM instances (e.g., on the same VM instances or VMs within the same data center). The concept of distance and affinity has been introduced in Sections 4.2.3 and 4.2.5. By calculating the affinity-weighted distance $d_{k,l} \cdot a_{s,t}$, we obtain the "cost" for interaction between each pair of services s and t deployed on VM instances k and l. This value is summed up for all pairs of services, VM instances, and container instances c_s and c_t where actual container instances are deployed. However, using a representation for this condition of the form $x_{(c_s,k)} \cdot x_{(c_t,l)}$ is not feasible in our ILP formulation by definition, since a multiplication of decision variables defies a linear term. For that reason, we introduce the boolean helper variable $\lambda_{(c_s,k,c_t,l)}$, which is defined by Constraint 4.18. Consequently, we only add to the sum where $\lambda_{(c_s,k,c_t,l)} = 1$. The importance of co-location to minimize latency within the optimization model is balanced using the variable ω_L.

Notably, containers entering the grace period are considered a minor factor for the co-location optimum of the prospective resource allocation, and are thus omitted.

Term 4: Minimize over-provisioning. To keep provisioning of excess resources at a minimum, this term subtracts the expected load L_s for a service s from the sum of actually supplied capacity for the service s on any VM instance k (i.e., $\sum_{c_s \in C_s} \sum_{k \in K} (Q_c \cdot x_{(c_s,k)})$). The difference is summed up for all services $s \in S$. While over-provisioning on VM level is already implicitly considered in terms of cost minimization by the first term, this term ensures that over-provisioning of containers placed on VM instances, which have free unused resources left, is avoided. Although the negative implications of more resources provisioned at the same cost might not be obvious in the first place, this term takes care of the important aspect to keep free VM resources at disposition for future reallocation and scaling actions. To control its weight, we employ the variable ω_Q.

Term 5: Use cached container images. The last term aims at gaining QoS benefits by exploiting accelerated startup times of containers, according to Requirement 3.3. This is achieved by employing caching container images on VM instances. For the deployment of any container of type c_s belonging to a service s, the container image for s must first be initially downloaded onto the respective VM instance, in order to launch a service's container instance for the first time. We assume that once an image is downloaded, it will be stored in a local cache of the VM until its termination, and will be available for future deployments of containers of that service s, facilitating faster startup times to contribute to the service's capacity more quickly. A cached image of a service s on a VM instance k is indicated by $I_{s,k} = 1$. Therefore, we sum up the amount of missing image caches (i.e., $1 - I_{s,k}$) for all allocated containers of the prospective resource allocation (i.e., if $x_{(c_s,k)} = 1$). The weight ω_I is used to balance this aspect's value within the objective function.

4.4.3 Constraints

In order to obtain a valid result, the optimization must adhere to multiple constraints, as defined in the following.

QoS Requirements

According to Requirement 3.2, we need to ensure that QoS requirements with regard to supplied capacity are fulfilled for each service. Therefore, in Constraint 4.7, we calculate the sum of provided capacity for each service on the left side of the inequation. Q_{c_s} represents the maximum capacity of a container of type c_s in terms of possible invocations. It is added to summation for each allocated container instance on a VM instance k, i.e., if $x_{(c_s,k)} = 1$. The right-hand side of the constraint requires the resulting capacity to be greater than or equal to the total load incurred for the respective service s, which is represented by L_s. Consequently, this constraint prevents under-provisioning based on the current knowledge about the system state.

$$\forall s \in S\colon \quad \sum_{c_s \in C_s} \sum_{k \in K} (x_{(c_s,k)} \cdot Q_{c_s}) \geq L_s \tag{4.7}$$

Resource Allocation Constraints

Overallocation of VMs must be prevented in a way that deployed containers can not exceed VM resources at any time. Constraints 4.8 and 4.9 restrict the sum of allocated container resources on a VM instance k_v to its supplied resource capacity. On the left-hand side of the inequations, we sum up the allocated resources on a VM instance k_v for each service s. The helper variables $\mu^{CPU}_{(s,k_v)}$ and $\mu^{MEM}_{(s,k_v)}$ reflect the maximum required container resources to reserve for a service s during the grace period. The resulting sum of resource demands for each VM instance are limited by the supplied CPU and memory resources of the respective VM type v on the right side of the constraints (i.e., R^{CPU}_v and R^{MEM}_v), given that the VM instance k_v is being allocated (i.e., $y_{(k_v)} = 1$). Otherwise, the allocated resources are limited to zero.

$$\forall v \in V,\ \forall k_v \in K_v\colon \quad \sum_{s \in S} \mu^{CPU}_{(s,k_v)} \leq R^{CPU}_v \cdot y_{(k_v)} \tag{4.8}$$

$$\forall v \in V,\ \forall k_v \in K_v\colon \quad \sum_{s \in S} \mu^{MEM}_{(s,k_v)} \leq R^{MEM}_v \cdot y_{(k_v)} \tag{4.9}$$

To facilitate proper reconfiguration of services within a VM with regard to the grace period, we restrict reallocations of containers on VM level to vertical scaling of single container instances. This is ensured by Constraint 4.10, which limits the number of containers belonging to a service s running on a specific VM instance k to 1. On the left side of the inequation, the number of allocated containers is counted by summing up the values $x_{(c_s,k)} = 1$. As a result, scaling out to multiple container instances within a VM is prohibited. Rather, resizing container instances vertically is encouraged. I.e., an increase or decrease in supplied capacity for a service s on a VM instance is achieved by reconfiguring an already allocated container instance to take on a smaller or larger type of possible container types $c_s \in C_s$.

$$\forall s \in S,\ \forall k \in K\colon \quad \sum_{c_s \in C_s} x_{(c_s,k)} \leq 1 \tag{4.10}$$

To determine the maximum resource requirements on a VM, we need to take into account the deployed container resources on VMs per individual service, in order to reflect reconfigurations during the grace period. Since only one container per service may be deployed at a time on a VM instance, vertical reconfiguration can be performed instantly, without requiring multiple container instances to run concurrently during the grace period. Hence, it is sufficient to reserve enough resources to fulfill the higher demand of the already running and the prospective container instance.

The helper variables $\mu^{CPU}_{(s,k_v)}$ and $\mu^{MEM}_{(s,k_v)}$ are defined by Constraints 4.11–4.12 and 4.13–4.14, respectively, and represent the maximum resources of a VM instance k allocated to containers of a service s during grace period. We therefore require the left-hand side of the inequations to be greater than or equal to the summed up resource usage of each container c_s in terms of CPU and memory (i.e., $U^{CPU}_{c_s}$ and $U^{MEM}_{c_s}$, respectively), in both cases of (a) container instances being deployed in the prospective resource allocation (i.e., $x_{(c_s,k)} = 1$), and (b) already running container instances ($z_{c_s,k} = 1$). The term $(1 - g_{(k)})$ ensures that Constraints 4.12 and 4.14 are omitted if the given VM instance is being abandoned (i.e., if $g_{(k)} = 1$), so that the left-hand side can take the value 0. This is feasible since no further containers are allocated on abandoned VM instances, eliminating the need to consider any resource constraints. Otherwise, positive values of $\mu^{CPU}_{(s,k_v)}$ and $\mu^{MEM}_{(s,k_v)}$ would interfere with Constraints 4.8–4.9, resulting in an undesired restriction of $y_{(k_v)}$ to take the value 1.

$$\forall s \in S,\ \forall k \in K\colon \quad \mu^{CPU}_{(s,k)} \geq \sum_{c_s \in C_s} (x_{(c_s,k)} \cdot U^{CPU}_{c_s}) \tag{4.11}$$

$$\forall s \in S,\ \forall k \in K\colon \quad \mu^{CPU}_{(s,k)} \geq \sum_{c_s \in C_s} (z_{c_s,k} \cdot U^{CPU}_{c_s}) \cdot (1 - g_{(k)}) \tag{4.12}$$

$$\forall s \in S,\ \forall k \in K\colon \quad \mu^{MEM}_{(s,k)} \geq \sum_{c_s \in C_s} (x_{(c_s,k)} \cdot U^{MEM}_{c_s}) \tag{4.13}$$

$$\forall s \in S,\ \forall k \in K\colon \quad \mu^{MEM}_{(s,k)} \geq \sum_{c_s \in C_s} (z_{c_s,k} \cdot U^{MEM}_{c_s}) \cdot (1 - g_{(k)}) \tag{4.14}$$

VM Allocation

A container instance deployed on a VM (i.e., if $x_{(c,k)} = 1$) requires the respective VM instance k to be leased. Corresponding to [44], Constraint 4.15 ensures that the helper variable $y_{(k)}$, which indicates a leased VM instance k, takes the value 1 if any containers are allocated on k. Since several containers may be placed on one particular VM instance, the left-hand side of the inequation may exceed the value 1. Thus, to satisfy this constraint, the right side is multiplied with an arbitrary large number, i.e., $y_{(k)} \cdot M$, where M represents the upper bound for the number of containers hosted on a VM. We choose $M = 1\,000$, which is most likely sufficient for any practical use case.

Additionally, Constraint 4.16 restricts $y_{(k)}$ to take the value 1 only if k has any containers allocated to it, and the value 0 otherwise (i.e., if $\sum_{c \in C} x_{(c,k)} = 0$). This is necessary as $y_{(k)}$ is not minimized by the objective function in case of on-premise VM instances with cost $P_v = 0$.

$$\forall k \in K\colon \quad \sum_{c \in C} x_{(c,k)} \leq y_{(k)} \cdot M \tag{4.15}$$

$$\forall k \in K\colon \quad \sum_{c \in C} (x_{(c,k)}) - y_{(k)} \geq 0 \tag{4.16}$$

In Constraint 4.17, we require the helper variable $g_{(k)}$ to reflect a VM's leasing status from the previous period, i.e., β_k, in case the VM instance k is not used in the prospective resource allocation (i.e., if $y_{(k)} = 0$). If k is used in the prospective resource allocation, $g_{(k)}$ takes the value 0.

$$\forall k \in K\colon \quad g_{(k)} = \beta_k \cdot (1 - y_{(k)}) \tag{4.17}$$

Co-Location

The helper variable $\lambda_{(c_s,k,c_t,l)}$ represents the boolean expression $x_{(c_s,k)} \wedge x_{(c_t,l)}$ in our ILP model, which is needed to determine whether any two given container instances are allocated simultaneously. This is achieved by Constraint 4.18, which provides a linear formulation of the aforementioned conjunction [14]:

$$\begin{gathered} \forall s,t \in S,\ \forall c_s \in C_s,\ \forall c_t \in C_t,\ \forall k,l \in K\colon \\ \lambda_{(c_s,k,c_t,l)} \geq x_{(c_s,k)} + x_{(c_t,l)} - 1 \end{gathered} \tag{4.18}$$

No further constraints are needed in order to restrict $\lambda_{(c_s,k,c_t,l)}$ to 0 in case the expression does not hold, since the objective function optimizes for a minimum value of $\lambda_{(c_s,k,c_t,l)}$.

Variable Restrictions

Finally we define in Constraint 4.19 that the used decision variables, including helper variables, are restricted to the appropriate number ranges of booleans and non-negative integers:

$$\begin{aligned} x_{(c,k)} &\in \{0,1\} & & \\ y_{(k)} &\in \{0,1\} & \mu^{CPU}_{(s,k)} &\in \mathbb{Z}_{\geq 0} \\ g_{(k)} &\in \{0,1\} & \mu^{MEM}_{(s,k)} &\in \mathbb{Z}_{\geq 0} \\ \lambda_{(c_s,k,c_t,l)} &\in \{0,1\} & & \end{aligned} \tag{4.19}$$

CHAPTER 5

Implementation

In this chapter, we outline the implementation details of the proposed scheduling middleware from Chapter 4. Moreover, we implement and discuss two different approaches for solving the optimization problem: Besides using an exact ILP solver, we provide a heuristic implementation of the optimizer which employs a Genetic Algorithm.

Therefore, in the course of this thesis a reference implementation called Cooper (***Co**-location **op**timized **e**lastic **r**unner for microservices*) is provided, which in its current state is built for simulation and evaluation purposes. Thus noteworthy, our implementation is narrowed down to the fundamental aspects of the MAPE-K-based scheduling loop and the optimization approach, while actual cloud dependencies and infrastructure are substituted by (simulated) stub components.

The source code repository of Cooper is publicly available on GitHub[1].

5.1 General Implementation Details

5.1.1 Technology Stack

Cooper is a command-line based application written in the object-oriented programming language Java 11[2]. As application framework and for cross-cutting concerns, Spring Boot[3] is leveraged which provides ready-to-use means for eased configuration and dependency injection.

To avoid boilerplate code and to automatically enrich data objects with getters and setters, Project Lombok[4] is used. Additionally, Cooper makes use of the utility libraries

[1]`https://github.com/alexwais/cooper`
[2]`https://docs.oracle.com/en/java/javase/11`
[3]`https://spring.io/projects/spring-boot`
[4]`https://projectlombok.org`

JGraphT[5], which provides graph data structures, and Jackson[6] for parsing and generating CSV files. Apache Maven[7] is utilized as build tool and for dependency management.

The underlying system model (e.g., services, data centers, VM types, distance graph and interaction multiplication graph) is to be configured via YAML files, which can easily be swapped out depending on the desired evaluation scenario. In Appendix A, we list example YAML files showing the structure of Cooper's configuration parameters. Spring Boot provides support for such configuration files out of the box, with the ability to switch between different run configurations accordingly.

5.1.2 Simulated Components

As mentioned, connecting to actual third-party dependencies is out of scope for this implementation of our scheduling middleware. Therefore, the Cloud Controller and the Monitoring Controller (see Section 4.3.1) are implemented as stubs useful for simulation purposes. The related classes are located in the `at.ac.tuwien.dsg.cooper.simulated` package.

As a substitute for (multiple) Cloud Controllers operating different CSPs and private clouds, our implementation provides only one no-op implementation for passing instructions to potential CSPs in the `SimulatedCloud` class.

The `SimulatedMonitoringController` provides the Scheduler with static system metrics (i.e., load measures) which would normally be obtained via integrations with CSPs and monitoring infrastructure. A CSV file containing time series fixture representing service load is used as input, which can be configured depending on desired load arrival scenarios.

5.1.3 Scheduler and Optimization

At the core of the MAPE-K-based Scheduler component, a loop operates at an interval of 30 seconds. This is sufficient due to our design decision: We rely on a two-minute grace period, with in-between scheduling actions taking place every 30 seconds at most (cp. Section 4.2.6). When the supply of load measures from the Monitoring Controller stops, the loop finishes and causes the application to exit.

A central role within the middleware plays the Optimizer component, which is invoked during the planning phase. Different optimization approaches, which are described in detail in the subsequent sections of this chapter, are provided by implementing the `Optimizer` interface shown in Listing 5.1.

[5]`https://jgrapht.org`
[6]`https://github.com/FasterXML/jackson`
[7]`https://maven.apache.org`

Listing 5.1: Optimizer interface

```java
public interface Optimizer {

    OptResult optimize(Allocation currentAllocation,
                       SystemMeasures systemMeasures,
                       Map<VmInstance, Set<Service>> cachedImages);

}
```

To determine a new optimized allocation, Optimizers are provided with the current allocation state, system measures (as determined by the Monitoring Controller), and the list of cached service images per VM instance.

5.1.4 Interaction Simulation

Since we do not employ actual service discovery/load balancing facilities in our implementation, the flow of inter-service requests in the system needs to be simulated as well within Cooper. Therefore, we introduce a mechanism to simulate the distribution of internal service load among containers, to ultimately determine the latency of inter-service requests. This measure will be used in subsequent parts of our implementation and evaluation. The algorithm is implemented in `InteractionSimulation` and related classes of the package `at.ac.tuwien.dsg.cooper.interaction`, and operates as follows.

As the initialization step, the external load of each service, L_s^E, is distributed pro-rata among the allocated containers according to each container's load capacity. For the simulation, the algorithm then recursively iterates over the nodes of a rooted tree, which comprises three levels of a network of data centers, VMs and container instances. At the lowest level in that tree, the container instances induce internal service load targeted at the related services, based on their assigned external load and the load multiplication factors $i_{s,t}^M$. The induced load is then attributed to the parent node (i.e., the containers' hosting VM instance). The VM node first tries to assign the induced to load to its child nodes, the containers, which might induce further internal load. If all child nodes are used to capacity, the remaining load "spills over" to the next parent node, (i.e., data centers), which follow the same procedure to place the induced load among their children. This is repeated until all requests and induced internal load are processed. Noteworthy, this recursive simulation approach is only possible due to our assumption of an acyclic service interaction graph.

Furthermore, the simulation algorithm records the source and target containers for each induced inter-service request. Using that information, the request latency is derived from the distance graph of data centers (see Section 4.2.3). If source and target container reside on the same VM instance, a latency of 0 ms is assumed.

5.2 Basic Optimization Using a First-Fit Algorithm

As a first approach towards feasible solutions for our optimization problem, we implement a naïve algorithm operating in a "First-Fit" fashion, as commonly seen for the class of bin-packing problems [24]. We rely on this baseline approach in other parts of this work, particularly for the evaluation (see Chapter 6) and as part of the Genetic Algorithm (see Section 5.4).

The implemented approach performs simple scaling actions to fulfill basic QoS and resource requirements according to the constraints defined in Section 4.4.3. Beyond that, it also aims to avoid unnecessary over-provisioning. Therefore, the algorithm implemented in the `FirstFitOptimizer` class performs the following steps for determining a new resource allocation:

Remove excess containers In case of over-provisioned services, the algorithm iterates over the related, currently allocated containers. If a container's load capacity is less or equal than the remaining over-provisioned capacity of the service, the container is marked for termination.

Determine additionally needed containers For each under-provisioned service, the containers to provision are selected in a "First-Fit Decreasing" manner: First, the available container types are sorted from large to small in terms of load capacity. Then we iterate over the container types, and repeatedly add a container type while its successor's capacity is below the remaining under-provisioned capacity, or there is no smaller container type left. Otherwise, the algorithm moves on to the next container type. This is repeated until under-provisioned service capacity is compensated, i.e., Constraint 4.7 is met.

Place containers on existing VMs We sort the containers to place descending by resource demand and use the container types' memory demand as a proper proxy for that purpose. The algorithm then places each container in a First-Fit manner: Iterating over the set of currently leased VMs, it places the container on the first encountered VM instance that has sufficient resource capacity available. This is ensured by checking whether the VM instance bears enough free CPU and memory resources, considering any already allocated containers as well as containers marked for removal (cp. Constraints 4.8–4.14).

"Open" new VMs for remaining containers In case of excess containers left, additional VM instances are provisioned. Therefore, the available unused VMs are sorted by cost. Starting with the cheapest, the algorithm places as many of the remaining containers as possible, without overallocating the VM instance, and considering the limitation of our optimization model to place at most one container of the same service. This is repeated until all containers are placed.

5.3 Exact Optimization Using an ILP Solver

For the implementation of an exact optimizer according to our ILP model, this work uses the Java library of the commercial ILP solver IBM CPLEX[8].

The ILP-based optimization is implemented in the `IlpOptimizer` class, which initializes a new ILP problem instance for every optimization run. Therefore, the class `IlpProblem` assembles the CPLEX decision variables, objective function and constraints based on the optimization input data and the system model, according to our ILP formulation defined in Section 4.4. By invoking CPLEX and mapping the results of the decision variables, the problem is solved and the ILP optimizer returns its result to the Scheduler.

CPLEX is employed using its default parameters, i.e., no specific parameters are set e.g. for tolerances of result quality.

5.4 Heuristic Optimization Using a Genetic Algorithm

For the heuristic implementation of the optimizer, we leverage a Genetic Algorithm which employs the Jenetics[9] Java library. The related classes are located inside the package `at.ac.tuwien.dsg.cooper.genetic`.

In Jenetics' terminology, *individuals* of a *population* are specified by the *genotype* which acts as a blueprint for the individuals' structure and features. Each feature is represented by a *gene* within the genotype, which can take mutable values (referred to as *alleles*) of a certain value type. Associated genes can be grouped in *chromosomes*, which share the same constraints for all genes. Similar to the decision variables in the ILP model, the alleles are the moving parts of the optimization problem, and determine the actual shape and fitness (together called the *phenotype*) of each specific individual. Finally, each individual is a candidate result to our optimization problem, which is approximated by choosing the fittest of individuals [86].

In the following sections we provide details about the specific implementation of the Genetic Algorithm and how it is applied to our domain's optimization problem.

5.4.1 Evolution Engine

Listing 5.2 shows the employed evolution configuration using Jenetics' `Engine` builder, located in the `GeneticAlgorithmOptimizer` class.

In the initial builder method in Line (2), the engine is configured with our domain specific codec and fitness function, which is discussed in more detail in the oncoming sections. Line (3) adds a custom constraint and repairing mechanism, which is applied during the initialization of individuals, as discussed in detail in Section 5.4.3. Line (4) instructs the engine to optimize for minimal fitness.

[8]`https://www.ibm.com/analytics/cplex-optimizer`
[9]`https://jenetics.io`

Listing 5.2: Structure of the Genetic Algorithm's evolution engine

```
1   var bestPhenotype = Engine
2       .builder(fitnessFunction, codec)
3       .constraint(repairingConstraint)
4       .minimizing()
5       .populationSize(25)
6       .survivorsFraction(0.5)
7       .maximalPhenotypeAge(60)
8       .survivorsSelector(
9           new EliteSelector<>(1,
10              new TournamentSelector<DistributedIntegerGene, Float>(3)
11          )
12      )
13      .offspringSelector(
14          new RouletteWheelSelector<>()
15      )
16      .alterers(
17          new UniformCrossover<>(0.05, 0.05),
18          new SwapMutator<>(0.05),
19          new Mutator<>(0.05)
20      )
21      .build()
22      .stream()
23      .limit(120)
24      .collect(EvolutionResult.toBestPhenotype());
```

Line (5) defines that the Genetic Algorithm's population shall comprise 25 individuals. In Line (6), the fraction of surviving individuals of each generation is set to 50 % of the population (i.e., those will live on to the next generation unaltered), with an age limit set to 60 generations in Line (7). The defined values showed a reasonable balance between survival of fit individuals and diversity, leading to practical results in preliminary tests.

Lines (8)–(12) parameterize the selection of survivors. For that matter, tournament selection in combination with *elitism* is used. The `EliteSelector` ensures that the fittest individual will always be selected to survive, while the remaining individuals for the surviving 50 % are determined by the `TournamentSelector`, which repeatedly selects the fittest individual from random samples of 3 individuals (comprising the "tournament").

For selecting the offspring population for the next generation, a `RouletteWheelSelector` is defined in Lines (13)–(15). This selection strategy randomly selects individuals with a probability proportional to their fitness. The selected individuals are then altered by the related *alterers* in Lines (16)–(20), which perform recombination and mutation for the offspring population. These mechanisms are discussed in detail in Section 5.4.6.

The evolution engine stream is limited to perform 120 generations in Line (23). Finally, the last line extracts the best phenotype from all generated individuals.

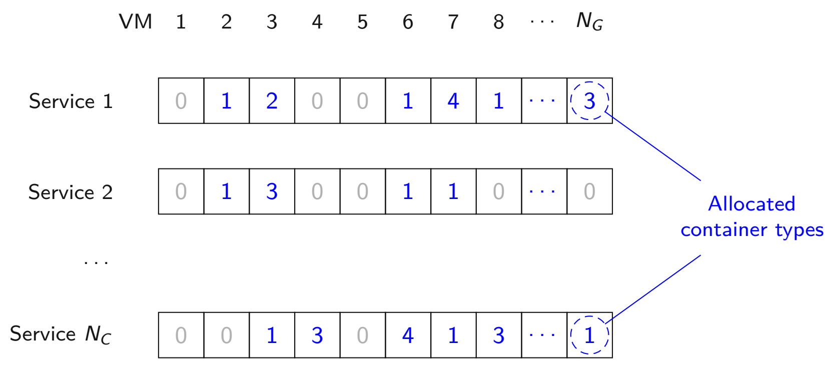


Figure 5.1: Chromosome encoding of resource allocations

5.4.2 Encoding

Choosing the right encoding of the problem domain is crucial for designing a Genetic Algorithm. It is responsible for mapping the domain-specific features to chromosomes and genes, which can then be digested by the Genetic Algorithm. For the purpose of this thesis, the encoding must reflect the allocation of containers among VMs, i.e., a mapping matrix of containers to VMs. Therefore, the genotype in our implementation always follows a two-dimensional matrix representation, which consists of N_C chromosomes (rows), each containing the same number of N_G genes (columns).

During implementation, encoding has been approached using the following different strategies of matrix representations:

- **Containers by VMs, boolean genes:** Each chromosome represents a VM instance $k \in K$, while their genes represent a specific container type $c \in C$ belonging to any service. The alleles hold booleans, representing whether a given container type is actually placed on a VM instance. This results in $N_C = k^{\#}$ chromosomes with $N_G = c^{\#}$ genes each.
- **VMs by services, container types as genes:** Each chromosome represents a service $s \in S$, containing genes for each VM instance $k \in K$. The alleles represent the types of placed containers on each VM instance, which is encoded by an integer value in the range $\{0, \ldots, c_s^{\#}\}$ depending on the chromosome's associated service s. I.e., the range of alleles corresponds to the number of available container types for a given service, with non-allocated VMs being indicated by zero values. This results in $N_C = s^{\#}$ chromosomes, each containing $N_G = k^{\#}$ genes.

As preliminary tests showed better outcome and performance of the latter strategy, we proceed with the *VMs by service* encoding, which is illustrated in Figure 5.1. Its main advantage originates from a more compact problem size, with a multitude lower number of genes compared to the boolean mapping encoding. Additionally, this encoding strategy also enforces that each service is limited to having one container instance deployed per VM at the same time, according to Constraint 4.10 of the ILP model.

The required encoding facilities, including methods for mapping between our domain model allocations and Jenetics' genotypes, are implemented in the `AllocationCodec` class.

5.4.3 Initialization

When creating the initial population, it is desired to equally distribute container placements over the problem space for its best exploration – while preventing any constraint violations like under-provisioning and matching load demand as closely as possible. Jenetics provides implementations for several types of genes (e.g., boolean, numeric), which, besides defining the type of the alleles and their constraints, act as a factory to randomly generate the alleles of newly created individuals.

In order to achieve a suitable initialization in our problem domain, we rely on the custom implementations `DistributedIntegerGene` and `DistributedIntegerChromosome`, which consider an important domain-specific aspect. They specifically aim at placing containers according to a certain probability, which represents the share of required load capacity on the globally available capacity. This comes down to genes following a probability distribution when initializing an allele with a certain container type vs. no container placement.

Let P_s be the probability that any container of service $s \in S$ is placed on any VM instance. For each chromosome (i.e., service), its genes show the probability P_s for initializing the allele with a value > 0, and a probability of $1 - P_s$ for initializing it with 0 (representing that no container is placed on a certain VM).

P_s is determined in Equation 5.1 as follows:

$$P_s = \frac{L_s}{\sum_{c_s \in C_s} (Q_{c_s} \cdot \frac{k^\#}{c_s^\#})} \tag{5.1}$$

Within the equation, we compute a hypothetical overall capacity available to service s in the denominator, i.e., $\sum_{c_s \in C_s} (Q_{c_s} \cdot \frac{k^\#}{c_s^\#})$. Therefore, it is assumed that all container types are distributed evenly among all VM instances, i.e., each container type would be placed on $k^\# / c_s^\#$ VM instances.

Furthermore, the probability P_s is distributed evenly among the available container types of a service, in order to retrieve the probability for a specific container placement, which is represented by P_{c_s}. I.e., for a service s having $c_s^\#$ container types defined, the probability for each container to be placed on any VM is $P_{c_s} = P_s / c_s^\#$.

5.4.4 Repairing

The initial container placement distribution accounts for approximated balance of demand and supply, but does not enforce any constraints. Pre-tests have shown that considering constraints only as part of the fitness function is not sufficient in order to guarantee that the Genetic Algorithm terminates with a valid individual.

For this purpose, we leverage the constraint mechanism provided by Jenetics by implementing the custom `RepairingConstraint` class. If an individual is tested invalid with regard to VM overallocation (cp. Constraints 4.8–4.9), they are repaired to fully comply with our domain specific requirements. However, we only consider 20 % of individuals to test for their validity; in the remaining cases, the check is omitted with the intention to maintain diversity within the population by keeping the majority of random individuals.

Of the individuals tested invalid, we repair

(a) 20 % by applying the First-Fit algorithm introduced in Section 5.2, which replaces the individual by a new valid allocation based on the system's previous target allocation,

(b) 80 % by executing a dedicated algorithm to repair the existing, randomly initialized individual.

The latter repairing algorithm (b) operates as follows:

1. Iterate over all VM instances and remove excess containers from any overallocated VMs.
2. Determine the missing capacity of any under-provisioned services.
3. For each under-provisioned service, randomly place containers on VMs with spare resources available until QoS requirements are met (i.e., missing capacity is compensated).

5.4.5 Fitness Function

The fitness function is the Genetic Algorithm's equivalent to the objective function of our ILP formulation. It is evaluated for each individual and used as selection criterion, with lower values indicating fitter individuals. To determine a fitness value, the implemented `FitnessFunction` class performs a calculation comprised of six terms, each containing a term weight similar to the ILP objective function, as outlined in Equation 5.2:

$$\begin{aligned} fitness \;=\; & P \cdot \omega_C \;+\; P_G \cdot \omega_G \;+\; L \cdot \omega_L \\ & +\; Q \cdot \omega_Q \;+\; I \cdot \omega_I \;+\; V \cdot \omega_V \end{aligned} \tag{5.2}$$

The terms covering the aspects of cost (P) and cost accrued for overhead during the grace period (P_G) follow the first two terms of the ILP objective function. Likewise, the terms regarding over-provisioning (Q) and container image caching (I) are calculated based on the current system state and the underlying system model similar to the corresponding terms in the ILP objective function.

For minimizing interaction latency, we follow a different approach than in our ILP formulation. Since we are not limited to linear terms, we leverage the interaction simulation introduced in Section 5.1.4. This intends advantages in accuracy, since we value the actual individual load between containers (according to our simulation), in contrast to the approach of the ILP objective function, which relies on the more general global affinity between services. The resulting average latency is represented by L in the fitness function.

Furthermore, the Genetic Algorithm introduces a sixth term, which accounts for fundamental constraints by adding violation penalties (V) to the fitness result. Constraint violations are counted by summing up the following occurrences:

- **Overallocated VMs:** Any VM instances where placed containers exceed the actual resource capacity (i.e., the VM's CPU or memory supply), including abandoned containers during grace period. This corresponds to Constraints 4.8–4.9 in the ILP formulation.

- **Under-provisioning (QoS):** We count the amount of unhandled requests, i.e., the number of requests exceeding a service's capacity, reflecting the impact of violated QoS constraints (cp. Constraint 4.7 of the ILP model).

Noteworthy, no further helper constraints need to be considered besides the above fundamental constraints related to actual requirements, as opposed to the ILP model.

5.4.6 Recombination and Mutation

If only selection was applied, populations would tend to converge to a local "fittest" individual without sufficient exploration of the search space. To maintain diversity while exercising "survival of the fittest" over the course of evolution, we perform recombination and mutation as a central principle. The related code snippet defining the alterers of our evolution engine is shown in Listing 5.3.

For recombination, offspring are created from pairs of existing parent individuals, aiming to obtain novel individuals showing combinations of the parents' traits. Therefore, the `UniformCrossover` alterer in Line (2) swaps single genes at the same index between two chromosomes. The probability for a given individual to be selected for recombination is defined to be 0.05 in the first argument; the same probability is also applied for swapping two given genes by the second argument. In fact, the uniform crossover swaps out container types of services on a VM instance, including the presence of a service at all, which stimulates exploration of the search space with regard to co-location of affine services.

The `SwapMutator` in Line (3) moves container placements between VMs, by swapping the order of genes within a chromosome with a probability of 5 %. As a benefit of this

Listing 5.3: Alterers of the evolution engine

```
1  .alterers(
2      new UniformCrossover<>(0.05, 0.05),
3      new SwapMutator<>(0.05),
4      new Mutator<>(0.05)
5  )
```

type of gene swapping, the overall service capacity stays the same when moving around containers, hence QoS constraints are maintained.

Finally, the `Mutator` in Line (4) randomly alters any gene with a probability of 5 %. To obtain the mutated allele, a new value is generated according to the chromosome's probability distribution of container types, as defined in Section 5.4.3.

Notably, if either recombination or mutation leads to invalid individuals, the custom `RepairingConstraint` ensures that any constraint violations get fixed.

Lastly it should be mentioned, that we chose the defined parameters as they have shown to lead the Genetic Algorithm to suitable results during preliminary tests. While ensuring a certain degree of diversity is essential, choosing greater probabilities for recombination and mutation leads to increased undirected fluctuation in the population, and showed rather counterproductive.

CHAPTER 6

Evaluation

After providing the implementation for the proposed approach, we conduct a quantitative evaluation of our work in this chapter. Therefore, we first describe the evaluation setting in detail and introduce the simulated scenarios used to evaluate Cooper. As the outcome of the evaluation, we then present the observed results in detail and show how the implemented optimization approaches perform in comparison.

6.1 Prerequisites

6.1.1 Data Center Infrastructure

In our evaluation scenarios we assume a hybrid cloud landscape which comprises three data centers that supposedly reside at different geographic locations. There are two public cloud data centers, *DC-1* and *DC-2*, and a private data center, *DC-Private*, which is closer located to DC-1. Figure 6.1 shows the graph of data center distances, which are represented by the supposed network latency for inter-service requests between data centers in milliseconds. Self-loops represent the internal latency which arises for network calls between VM instances within a data center.

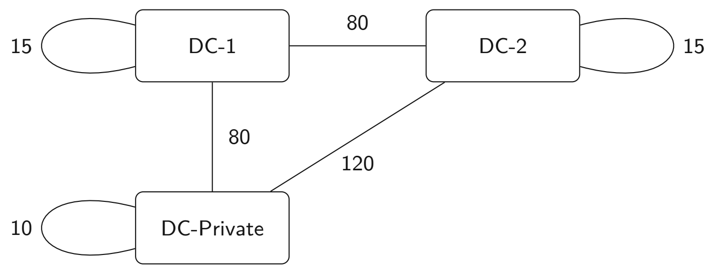


Figure 6.1: Network latency between data centers

Table 6.1: Resource configuration and cost of VM types

	Resources		Cost		
VM Type	CPUs	Memory	DC-1	DC-2	DC-Private
2.small	2	2 GB	0.027	0.033	–
2.medium	2	4 GB	0.052	0.056	–
4.large	4	8 GB	0.100	0.099	0
4.xlarge	4	16 GB	0.190	0.182	–

For the provided computational resources, we define four differently sized types of VMs that are available in the public cloud data centers. Each VM type varies in supply of CPU cores and memory as defined in Table 6.1. Furthermore, the VM types differ in leasing cost depending on the data center. DC-Private offers a fixed contingent of two *4.large* instances at zero leasing cost.

The specified cost follows roughly the common price ranges of public CSPs. As an additional characteristic, in our pricing assumption the cost per unit (with regard to memory supply) decreases for larger VM types. Furthermore, DC-2 charges higher cost for the smaller VM types, while it offers price advantages for the larger ones compared to DC-1.

6.1.2 Microservice Archetypes

For the microservice architecture deployed in our evaluation scenarios, we define the following three archetypes of services. The services' topology is represented by the interaction multiplication graph shown in Figure 6.2.

- **Gateway service *gw*:** This service represents an aggregating service node that handles a high throughput of external load (such as an API gateway or BFF, see Section 2.4.3), and performs downstream requests to backend services for a significant share of incoming load. We assume 50 % of incoming requests trigger a call to service *a*, in 10 % of handled requests a call to service *b* is induced. In the remaining cases, the gateway relies on cached responses or processes preflight requests that do not require a downstream request.

- **Backend service *a*:** This service exclusively acts as an internal backend service for the upstream *gw* service and does not receive any load from external clients. We assume that its business logic requires a call to service *b* in 50 % of handled requests.

- **Backend service *b*:** This service is a dependency for services *gw* and *a*. In addition, it serves external requests.

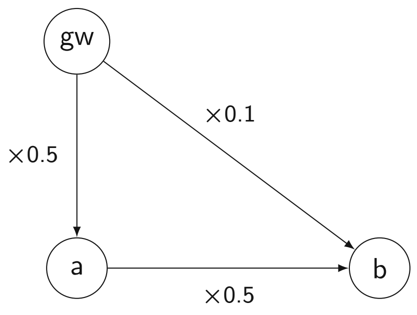


Figure 6.2: Interaction topology of service archetypes

For each service, we define four available container types as shown in Table 6.2. The container types differ between the gateway service and the backend services, which share the same configurations. Each container type can handle a predefined amount of arriving load (specified as hypothetical requests per second, RPS), corresponding to a certain resource demand (i.e., CPU units and memory in megabytes). The container types of the gateway service show a proportional exponential increase in both load capacity and memory demand, while CPU resource consumption develops linearly. On the other hand, the container types for the backend services a and b show doubled resource usage for each 50 % increase in load capacity. Notably, since the gateway service is assumed to mainly operate in a proxy-like manner for downstream requests with little resource consuming business logic carried out by itself, it can handle more load per resource unit compared to the backend services.

Table 6.2: Service container types

(a) Gateway service gw

Load Capacity	Memory	CPU units
1000 RPS	512 MB	1024
2000 RPS	1024 MB	1536
4000 RPS	2048 MB	2048
8000 RPS	4096 MB	2560

(b) Backend services a, b

Load Capacity	Memory	CPU units
600 RPS	1024 MB	512
900 RPS	2048 MB	1024
1350 RPS	4096 MB	2048
2025 RPS	8192 MB	4096

6.1.3 Evaluation Scenarios

First, we define two base scenarios for the microservice architecture deployed during our evaluation. Both are based on the previously introduced service archetypes and topology, but feature a different amount of services:

- **Scenario A (3 services):** Comprises the three services $S = \{gw, a, b\}$ which correspond exactly to the aforementioned service archetypes and topology.

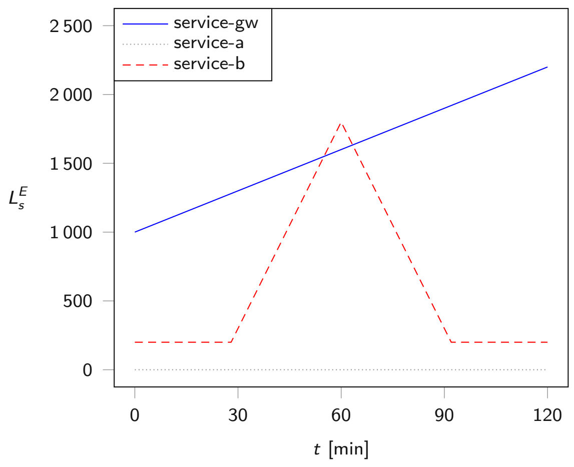


Figure 6.3: External load arrival patterns

- **Scenario B (30 services):** To pose a larger microservices scenario, we replicate the same archetypes by a factor of 10. Thus, we obtain a set of 30 services, $S = \{gw_0, a_0, b_0, \ldots, gw_9, a_9, b_9\}$. The interaction multiplication graph remains unaltered for each of the ten subsets of related services, $S_i = \{gw_i, a_i, b_i\} \in S$.

Furthermore, we consider different scenarios of load arriving over the evaluation time period of 120 minutes. In the following, we define a baseline scenario of incurred external load per service.

As previously defined for the service archetypes, external load incurs only for the gateway service and service b, while service a only receives internal downstream service calls induced by the gateway service. To account for diverse behaviour in load change over time, we assume the gateway service gw and service b feature two contrasting arrival patterns of external load, as shown in Figure 6.3:

- **Linear pattern:** The gateway service gw features a constant increase in external load, starting at 1000 RPS at $t = 0$ min and increasing by additional 10 RPS every minute, i.e., following the linear function $L_{gw}^E(t) = 10t + 1000$.

- **Pyramid pattern:** Service b shows a stable base load of 200 RPS. From $t = 30$ min to $t = 90$ min, the service experiences a “pyramid” style increase and decline, showing a linear increase of 50 RPS every minute until reaching the maximum of 1800 RPS at $t = 60$ min, followed by the inverted decline back to the base load.

On top of that, we provide three differently sized scenarios by replicating this base scenario using 1x, 10x, and 100x factors for their load.

Finally, we use combinations of the base service and load scenarios, resulting in the following five evaluation scenarios: *A@1x*, *A@10x*, *A@100x*, *B@1x* and *B@10x*.

Table 6.3: Available VM instances by scenario

Scenario	F^S	F^L	Public VM types' instance count ($k_v^\#$)	Total instance count ($k^\#$)
A@1x	1.5	1	2	18
A@10x	1.5	10	15	122
A@100x	1.5	100	150	1202
B@1x	15	1	15	122
B@10x	15	10	150	1202

6.1.4 Instance Count

Although the amount of available resources in the cloud is often referred to as virtually "unlimited", our optimization model relies on a finite set of provisionable VM instances. Moreover, it is of our interest to keep the search space (and thus, the available resources) at a reasonably small scale in order to maintain practical runtimes. The following sensible count of available VM instances showed beneficial during preliminary tests, while no noticeable compromise in result quality was observed.

We obtain the number of available VM instances per type, $k_v^\#$, by multiplying a scenario multiplication factor (F^S) with a load multiplication factor (F^L), as defined in Equation 6.1:

$$k_v^\# = \left\lceil F^S \cdot F^L \right\rceil \tag{6.1}$$

Furthermore, the total number of VM instances in the search space, $k^\#$, is determined by the number of VM instances available in the private cloud, and the number of VM types in each public cloud data center. Since we assume having two VMs present in the on-premise data center, and four possible VM types available in each of the two public cloud data centers, we obtain the total count of VM instances as defined in Equation 6.2:

$$k^\# = 8 \cdot k_v^\# + 2 \tag{6.2}$$

In Table 6.3, we summarize the configuration of scenario and load factors, and the resulting count of VM instances in the search space for each evaluation scenario.

6.1.5 Optimization Modes

For our evaluation, we assess five optimization modes and compare how they perform against each other for each of the evaluation scenarios. We employ the implemented exact ILP approach and the heuristic Genetic Algorithm approach, along with modified variants for each of them: To evaluate the benefits and drawbacks of the latency-reducing co-location aspect, we introduce two **-NC* (**N**o **C**o-location) modes, which do not consider that part within the optimization. Additionally, the simple First-Fit heuristic acts as the baseline for our evaluation.

The optimization modes are denoted and summarized as follows:

- **FF:** Baseline mode applying the implemented simple First-Fit approach (see Section 5.2).
- **ILP:** Exact optimization approach following our ILP model as designed in Chapter 4.
- **ILP-NC:** Same as ILP, but omitting the optimization aspect of minimized latency by exploiting co-location (i.e., omitting term 3 of the objective function as defined in Section 4.4.2).
- **GA:** Heuristic optimization mode using the Genetic Algorithm implemented in Section 5.4.
- **GA-NC:** Same as GA, but modified to not take into account the latency/co-location aspect. Therefore, the related term of the fitness function is omitted. Notably, this also renders the interaction simulation unnecessary, which is employed in the original GA mode in order to determine inter-service request latencies.

6.1.6 Parameterization

First, we define the general tolerances allowing the Planner to skip an optimization (cp. Section 4.3.2). For the load drift tolerance, we set $T_L = 2\,\%$; the fitness drift tolerance is set to $T_F = 5\,\%$. These parameters are effective for all optimization modes equally.

For the ILP- and GA-based approaches, we have to choose sensible parameters for the term weights of the objective function, and fitness function, respectively, as their calibration is crucial for the outcome of the optimization. The following parameters used in our evaluation have shown to produce viable results during preliminary tests.

The terms of the ILP objective function are weighted as defined in Table 6.4. Notably, ω_L is not applicable for the ILP-NC optimization mode, as the latency/co-location aspect is omitted (i.e., ω_L resembles zero).

The term weights of the Genetic Algorithm's fitness function are defined in Table 6.5 and differ from the ILP objective function, since these adaptions showed beneficial during preliminary tests. As a peculiarity of the fitness function, we have to consider the extra term which adds penalties for constraint violations. For that matter, we use a term weight with a reasonable large value of $\omega_V = 1\,000$. Furthermore, the parameterization of the co-location related term deserves closer attention due to the different nature of cost and interaction latency within the fitness function. While the cost factors (i.e., the terms weighted by ω_C and ω_G) are subject to scale with the number and types of leased VMs, the average latency of internal requests is generally independent from the system's resource demand. Therefore, we set ω_L specifically for each evaluation scenario,

Table 6.4: Term weights of the ILP objective function

Term	Variable	Value
Minimize cost	ω_C	1
Minimize waste in grace period	ω_G	0.2
Minimize latency by co-location	ω_L	0.001
Minimize over-provisioning	ω_Q	0.000001
Use cached container images	ω_I	0.0001

Table 6.5: Term weights of the Genetic Algorithm fitness function

(a) General term weights

Term	Variable	Value
Minimize cost	ω_C	1
Minimize waste in grace period	ω_G	0.4
Minimize over-provisioning	ω_Q	0.0001
Use cached container images	ω_I	0.0001
Constraint violation penalty	ω_V	1 000

(b) Parameterization of ω_L

Scenario	ω_L
A@1x	0.0001
A@10x	0.001
A@100x	0.01
B@1x	0.005
B@10x	0.1

in order to achieve a suitable balance between cost and latency minimization for different magnitudes of load and resource usage. The values of ω_L used in GA mode are defined in Table 6.5b.

6.1.7 Measures

To collect the results from our evaluation, we record the relevant measures of each conducted evaluation run. Therefore, we apply a sampling rate of two-minute periods, such that we obtain measures for each period $t \in \{0, 2, 4, \ldots, 118\}$, which corresponds to the defined grace period (see Section 4.3.2). Specifically, at the end of each grace period cycle (when the Scheduler component has completed any ongoing reallocation and the final target resource allocation is established), the relevant measures of the current system state are captured and recorded.

The following measures are determined and recorded by Cooper:

Runtime Each time the Planner invokes the Optimizer component, it measures the time until its completion, and captures the duration of each optimization run in milliseconds. Thus, this also includes any preparations prior to the actual optimization itself, e.g., the initialization of the decision variables and constraints in case of the ILP-based approaches. If no optimization is invoked within a scheduling cycle (i.e., if drift of the system state is within the tolerance as defined in

Section 6.1.6 and no constraints are violated), we record a separate flag indicating whether an optimization has occurred in a given period.

Cost To determine accrued cost, every 30 seconds we evaluate the cost of currently leased VMs (proportional to the hourly rates defined in Section 6.1.1). This is the interval at which leased VM resources may change, according to our design of the scheduling loop and the assumptions around the grace period. These samples of VM cost are further accumulated to obtain the accrued cost in each two-minute period of simulation time.

Inter-service request latency At the end of each two-minute sampling period, we evaluate the average latency of inter-service requests based on the current resource allocation. Therefore, we run our interaction simulation as described in Section 5.1.4.

Upon completion of an evaluation run, the collected measures are stored in a CSV file containing the records for each two-minute period. Within Cooper, the `EvaluationService` class is responsible for gathering all the related data and compiling the CSV output.

6.1.8 System Setup

The machine used to execute our evaluation runs with Cooper is an Apple MacBook Pro (13-inch, 2018), which is equipped with a 2.3 GHz Intel Core i5 quad-core CPU, 16 GB of DDR3 RAM and a solid state disk. The installed operating system is macOS Mojave (version 10.14.4).

6.2 Results

In the following, we present the results of our quantitative evaluation. Notably, we conducted five evaluation runs for each combination of optimization mode and evaluation scenario, hence the following results represent the mean of $n = 5$ samples.

6.2.1 Runtime

The average runtime of a single optimization varies considerably depending on the problem size (thus, evaluation scenario), and among optimization modes. Table 6.6 shows the arithmetic mean of average optimization runtimes of the $n = 5$ samples in seconds, including the standard deviation. The results cover only the actual optimization occurrences, i.e., single omitted optimizations during scheduling cycles are not treated as zero values.

In general, we find the FF mode clearly best performing, being close to 0 s of runtime in all scenarios. Notably, we witness the ILP-based approach being subject to considerable performance bottlenecks, in a way that Cooper is not able to perform a full evaluation run for scenarios A@10x upwards in case of ILP, and scenario B@10x in case of the ILP-NC mode. The reason for abortions was either the evaluation machine running

Table 6.6: Average optimization runtimes

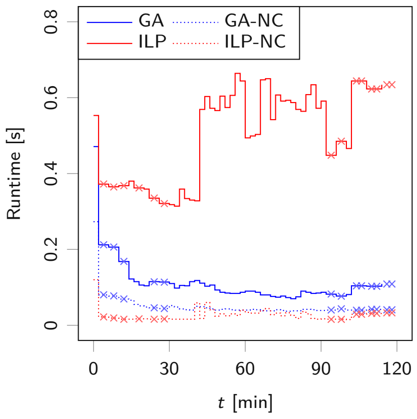
	A@1x	A@10x	A@100x	B@1x	B@10x
FF	0.00 ($\sigma = 0.00$)	0.00 ($\sigma = 0.00$)	0.00 ($\sigma = 0.00$)	0.00 ($\sigma = 0.00$)	0.02 ($\sigma = 0.00$)
GA	0.11 ($\sigma = 0.00$)	1.37 ($\sigma = 0.17$)	51.04 ($\sigma = 2.06$)	4.88 ($\sigma = 0.27$)	139.62 ($\sigma = 7.91$)
GA-NC	0.05 ($\sigma = 0.00$)	0.15 ($\sigma = 0.00$)	1.67 ($\sigma = 0.05$)	0.64 ($\sigma = 0.01$)	5.57 ($\sigma = 0.15$)
ILP	0.51 ($\sigma = 0.00$)	–	–	–	–
ILP-NC	0.03 ($\sigma = 0.00$)	0.45 ($\sigma = 0.02$)	15.33 ($\sigma = 0.08$)	1.44 ($\sigma = 0.00$)	–

out of memory, or the ILP solver being stuck, i.e., we aborted the evaluation when no solution could be found within an hour for a single optimization.

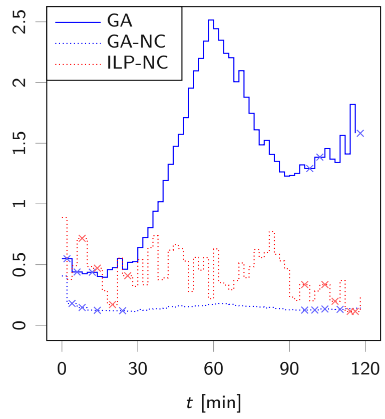
Looking at the individual optimization runtimes over the course of 120 minutes we can observe some characteristics depending on the optimization mode and evaluation scenario. Figure 6.4 shows the average runtime of $n = 5$ samples at each point in time (note that the scale of the y-axis differs for each plotted scenario). Initially, the first optimization run at $t = 0$ min shows spikes in runtime especially for the GA-based modes in scenario A@1x (Figure 6.4a) and for ILP-NC in A@100x (Figure 6.4c). A possible explanation could be that it is harder to find an optimum without an existing previous resource allocation, which narrows the search space due to its effect on the grace period cost. The GA approach's runtime seems to be linked to the incurred load of the system, or the amount of allocated resource, respectively. Especially for scenarios A@10x, A@100x and B@10x, we can observe a noticeable similarity with the combined linear and pyramid style load arrival patterns previously introduced in Section 6.1.3. In all, the ILP-NC mode seems to show arbitrary runtimes independent from incurred load, however showing a considerable spike in scenario A@100x (Figure 6.4c) starting at $t = 30$ min, i.e., when the load increase of service b kicks in. Moreover, it can be observed in general that the ILP-NC mode is more volatile over time t compared to the GA-based approaches.

Additionally, we plot the occasions of skipped optimizations using X marks in Figure 6.4. As we can observe, the defined tolerances to skip an optimization are met only outside the pyramid load arrival of service b, i.e., when $t < 30$ min and $t > 90$ min. In scenario A@100x, conditions to skip an optimization are never met.

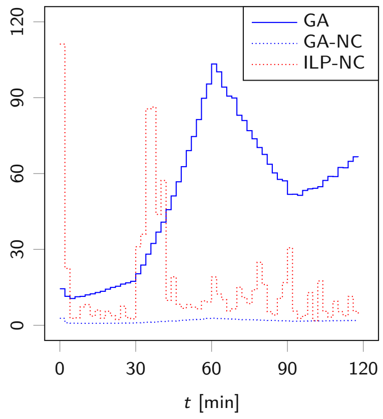
For a performance comparison of different optimization modes, we plot the average optimization runtime on a logarithmic scale in Figure 6.5. As seen for the smallest scenario A@1x, the ILP approach is slowest, followed by GA, GA-NC and ILP-NC. In the larger scenarios, GA shows the highest runtimes, since the ILP approach fails to complete the evaluation within feasible time or given hardware resources. Yet ILP-NC performs fastest for the A@1x scenario, it shows an almost proportional increase in runtime for each 10x multiple of incurred load for the scenarios A@1x to A@100x, and is therefore in the larger scenarios outrun by GA-NC, which achieves a more moderate slope in increased runtime. In all scenarios, GA is outperformed by the simplified modes GA-NC and ILP-NC, which do not consider the latency aspect.



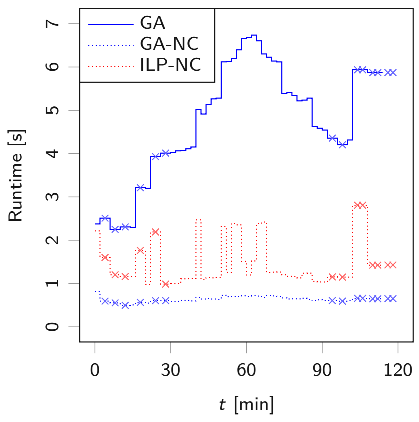
(a) Scenario A@1x



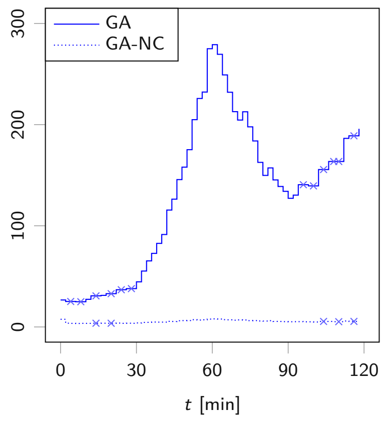
(b) Scenario A@10x



(c) Scenario A@100x



(d) Scenario B@1x



(e) Scenario B@10x

Figure 6.4: Optimization runtimes over time

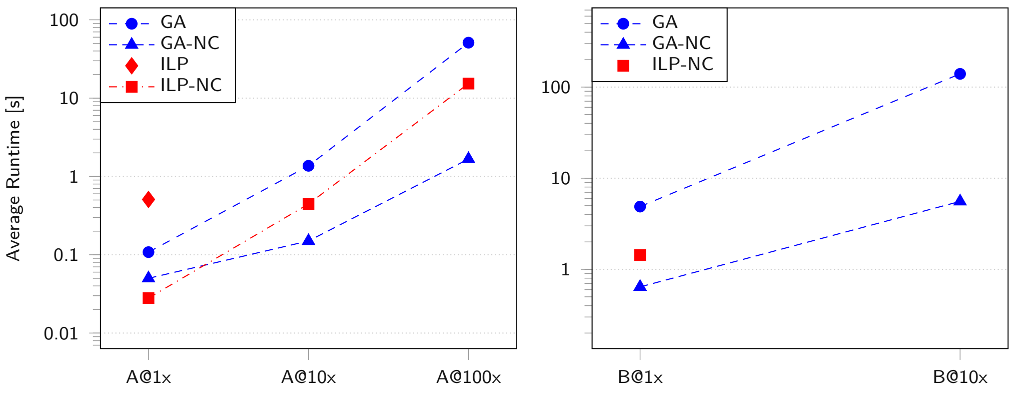


Figure 6.5: Comparison of average optimization runtimes

Table 6.7: Total cost

	A@1x	A@10x	A@100x	B@1x	B@10x
FF	0.05 ($\sigma = 0.00$)	1.71 ($\sigma = 0.01$)	20.91 ($\sigma = 0.03$)	1.46 ($\sigma = 0.01$)	19.73 ($\sigma = 0.02$)
GA	0.01 ($\sigma = 0.01$)	1.57 ($\sigma = 0.04$)	20.92 ($\sigma = 0.23$)	1.54 ($\sigma = 0.05$)	19.73 ($\sigma = 0.04$)
GA-NC	0.01 ($\sigma = 0.00$)	1.52 ($\sigma = 0.03$)	21.06 ($\sigma = 0.15$)	1.49 ($\sigma = 0.04$)	19.53 ($\sigma = 0.18$)
ILP	0.00 ($\sigma = 0.00$)	–	–	–	–
ILP-NC	0.00 ($\sigma = 0.00$)	0.95 ($\sigma = 0.00$)	12.17 ($\sigma = 0.00$)	1.05 ($\sigma = 0.00$)	–

6.2.2 Cost

During our evaluation, we measured the total cost reflected in Table 6.7, which shows the mean of the five evaluation samples including the standard deviation. The cumulated total cost over time is depicted for each evaluation scenario in Figure 6.6. Additional plots of the cost accrued in each sample period can be found in Appendix B.

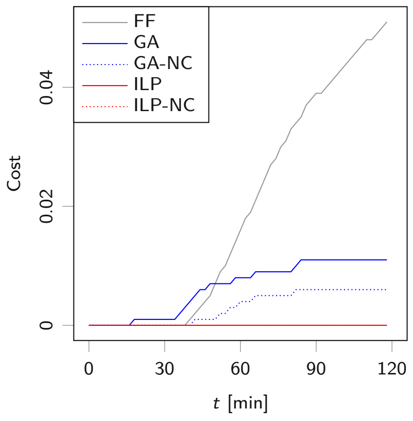
We can observe the most noticeable differences between optimization modes in scenario A@1x, where the ILP-based approaches maintain the optimal minimum of zero cost, while the GA-based approaches show a considerable achievement in cost reduction compared to the FF baseline. This can be explained by the advanced modes' accomplishment to primarily use the free, private cloud VM instances. While the GA-based modes occasionally lease some additional VMs during the load increase of service b (from 30 min to 90 min), the ILP-based approaches avoid scaling out to any public cloud VMs.

In Figure 6.7, we compare the percentage of cost savings based on the mean FF baseline; error bars indicate the standard deviation among the $n = 5$ evaluation run samples. While the GA-based modes achieve cost savings in the A@10x scenario (GA: 8.2 %, GA-NC: 11.3 %), they struggle to show improvements over FF in the larger scenarios. We can find an explanation for this in the fact that the Genetic Algorithm employs the First-Fit algorithm for a certain share of repaired individuals, which gains importance for larger problem sizes. In scenario B@1x, GA and GA-NC even fall behind the FF baseline by -5.8 % and -2.0 %, respectively. The ILP-NC approach, however, achieves significantly lower total cost in every scenario, which does not come unexpected due to its exact optimization for cost minimization. For GA-NC, we can observe improved cost reduction compared to the latency-aware GA approach in most scenarios.

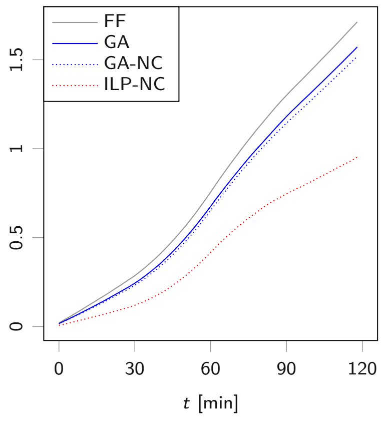
6.2.3 Latency

The total average of inter-service request latency in milliseconds is shown in Table 6.8, including the standard deviation among our $n = 5$ evaluation samples. Figure 6.8 shows how the cumulative average request latency develops over time. In addition, plots of the average request latency measured in each sample period can be found in Appendix B.

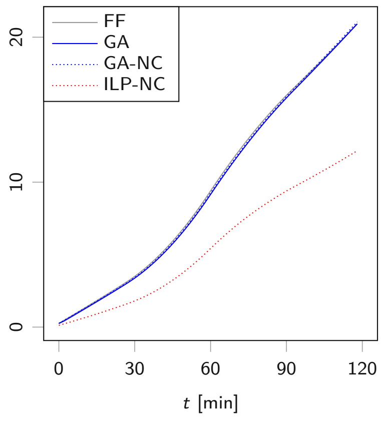
In the smallest scenario, A@1x (Figure 6.8a), the ILP approach shows the lowest latency over the whole duration of the evaluation, followed by ILP-NC, GA and GA-NC, all



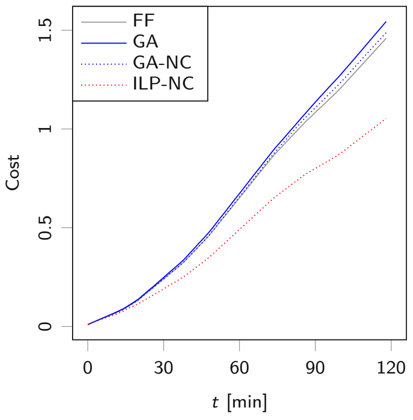
(a) Scenario A@1x



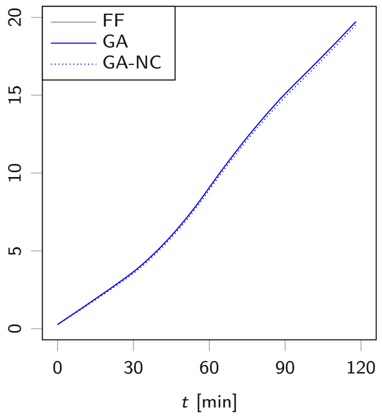
(b) Scenario A@10x



(c) Scenario A@100x



(d) Scenario B@1x



(e) Scenario B@10x

Figure 6.6: Total cost over time

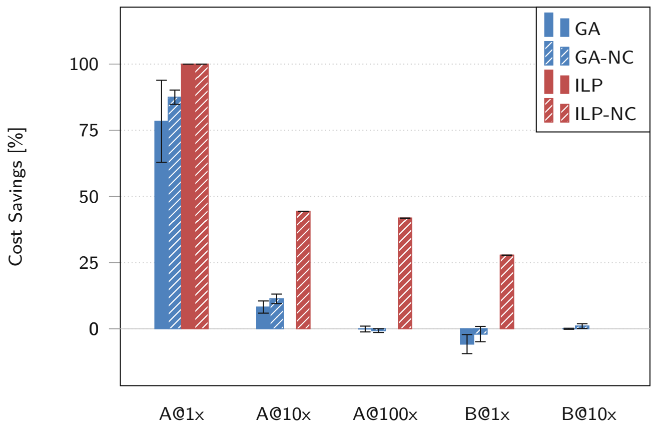


Figure 6.7: Comparison of cost savings

Table 6.8: Average request latencies

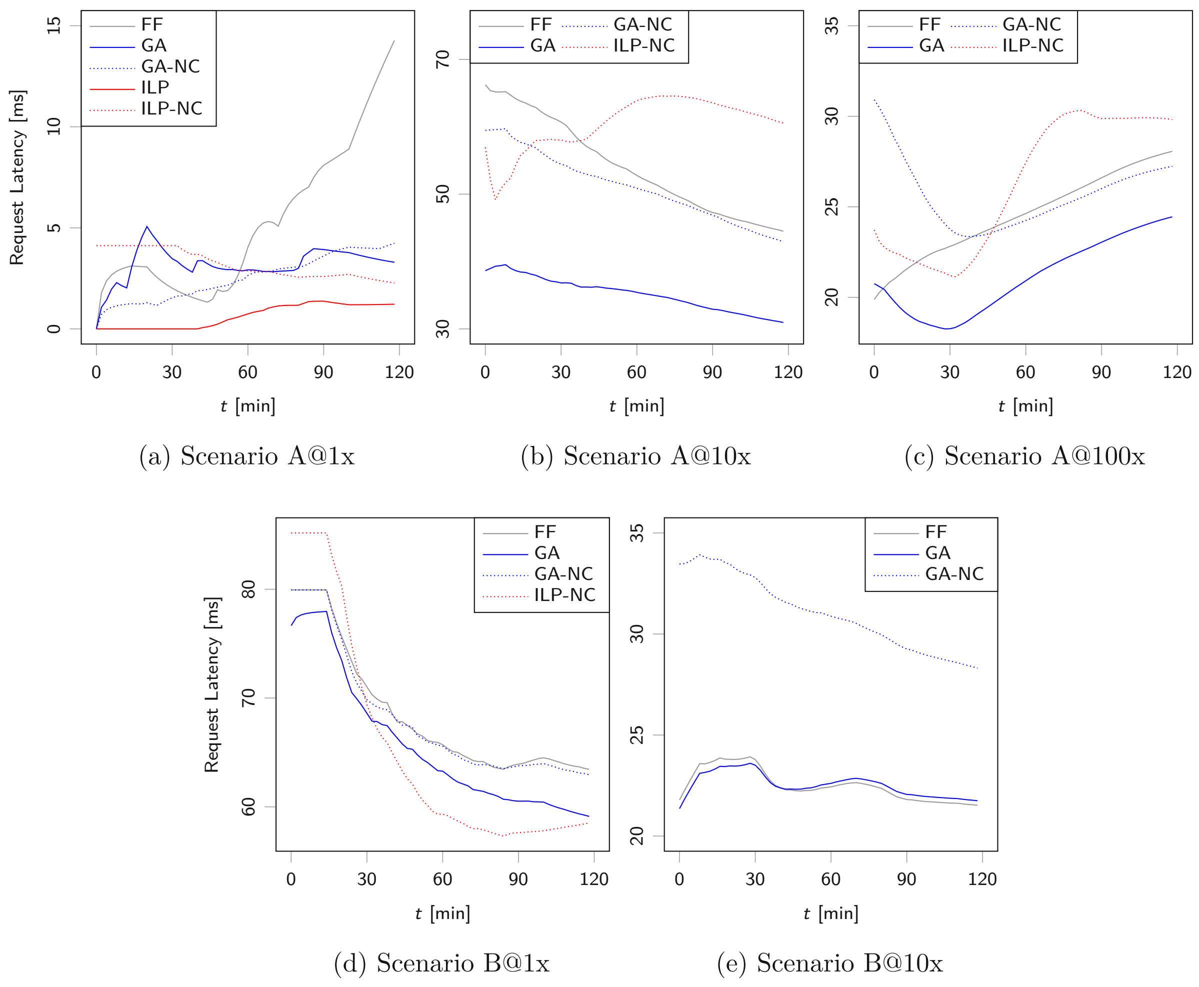
	A@1x	A@10x	A@100x	B@1x	B@10x
FF	14.27 ($\sigma = 1.32$)	44.49 ($\sigma = 0.92$)	28.06 ($\sigma = 0.12$)	63.45 ($\sigma = 0.72$)	21.53 ($\sigma = 0.37$)
GA	3.29 ($\sigma = 1.25$)	30.95 ($\sigma = 10.01$)	24.45 ($\sigma = 2.21$)	59.12 ($\sigma = 5.92$)	21.75 ($\sigma = 0.44$)
GA-NC	4.24 ($\sigma = 0.99$)	42.93 ($\sigma = 7.69$)	27.24 ($\sigma = 2.99$)	62.96 ($\sigma = 1.32$)	28.31 ($\sigma = 1.47$)
ILP	1.22 ($\sigma = 0.00$)	–	–	–	–
ILP-NC	2.27 ($\sigma = 0.00$)	60.57 ($\sigma = 0.02$)	29.82 ($\sigma = 0.01$)	58.51 ($\sigma = 0.13$)	–

of which achieve total averages below 5 ms. Notably, the ILP mode achieves to place containers in a way that all inter-service requests can be handled within the same VM instance until $t = 40$ min, hence showing 0 ms of latency since we attribute zero latency to requests in this case (cp. Section 5.1.4). FF clearly shows the highest resulting latency, with a total average of 14.27 ms.

In scenario A@10x (Figure 6.8b), the GA mode outperforms the other (latency unaware) approaches, keeping the average latency below 40 ms. Notably, FF, GA-NC and GA achieve to almost continuously reduce latency over time. In A@100x (Figure 6.8c), the system experiences considerably increased latency with the ILP-NC approach during the pyramid-shaped load increase of service b, starting at $t = 30$ min.

In scenario B@1x (Figure 6.8d), all approaches manage to reduce average latency, with GA and ILP-NC taking the lead. The latter even achieves to outperform GA as a byproduct of the cost minimization. In scenario B@10x (Figure 6.8e), the GA approach fails to achieve an outperformance over FF. Though the result is ambiguous: At first, GA shows lower latencies than FF. Starting with $t = 30$ min, however, GA accrues increased latencies, losing its advantage and showing higher total average latency.

When comparing the achieved latency reductions of different optimization modes relative to the FF baseline, we obtain from Figure 6.9 that in A@1x all approaches achieve to reduce latency considerably, by up to 91.5 % in case of ILP. Furthermore, GA shows improvements over FF in A@10x (30.4 % reduction), A@100x (12.9 % reduction) and B@1x (6.8 % reduction). In B@10x, however, the GA approach struggles to beat the FF baseline. Notably, both GA-based approaches show a considerable standard deviation in their results, as represented by the error bars. Looking at scenarios A@10x, A@100x and B@1x, we can observe that GA-NC shows a tendency for reduced latencies compared to FF. The ILP-NC approach shows a significant shortcoming in scenario A@10x (-36.1 %), as does the GA-NC approach in B@10x (-31.5 %).



(a) Scenario A@1x (b) Scenario A@10x (c) Scenario A@100x

(d) Scenario B@1x (e) Scenario B@10x

Figure 6.8: Cumulative average of request latency over time

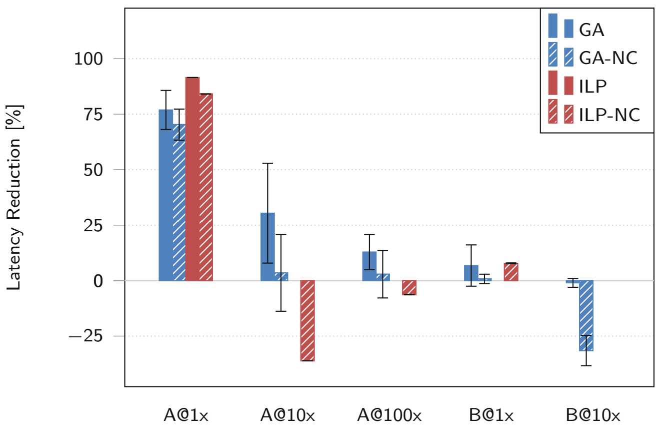


Figure 6.9: Comparison of request latency reductions

6.2.4 Summary

Based on the observed results, we can state in general that the evaluated approaches show expected behaviour in many aspects. Each approach demonstrated its ability to achieve improvements compared to the baseline; however, depending on scenario and problem size, results vary significantly and are ambiguous especially for the larger ones.

With regard to optimization runtime, the presented approaches follow the expected trend. As shown in Section 6.2.1, with increasing problem size the exact optimization approaches ILP and ILP-NC require considerable more time and resources than their heuristic counterparts. Most notably, the fully-fledged ILP mode is only feasible for the smallest scenario and refuses the evaluation in the other scenarios. A single optimization using ILP-NC takes on average 15.3 s in the A@100x scenario, which is 9x longer compared to 1.7 s using GA-NC. Notably, though not directly comparable, ILP-NC still performs better than the GA mode, which bears the highest absolute runtime in our evaluation with an average of 140 s in scenario B@10x.

In terms of cost minimization (Section 6.2.2), the ILP(-NC) approach clearly emerges as the winner. While saving 100 % of leasing cost in the A@1x scenario by fully exploiting private cloud resources, ILP-NC still achieves cost savings between 28 % and 44 % in the larger scenarios. The GA-based approaches, however, show deficits in achieving cost benefits reliably. While GA and GA-NC both save at least 78 % of cost in A@1x, and still manage to save 8 % (GA) and 11 % (GA-NC) in A@10x, they fail to outperform the baseline in the larger scenarios.

In Section 6.2.3, our evaluation showed the GA approach to be most reliable with regard to latency reduction. In the mid-range scenarios, we observe GA outperforming the baseline by 7 % to 30 %, though failing to achieve improvements in the largest scenario B@10x. Notably, ILP-NC and GA-NC show significant underperformance with regard to interaction latency in some scenarios.

Noteworthy, with increasing problem size the GA-based approaches lose advantage over the simple FF mode, regarding both cost and latency aspects. This can be explained as the First-Fit algorithm is employed to a certain extent when repairing invalid individuals. With larger problem sizes, the Genetic Algorithm seems less likely to find valid solutions based on random initialization and alteration, and therefore converges towards the valid individuals generated by the First-Fit algorithm.

In conclusion, it depends on the use case which approach to choose as the most promising. If latency reduction is a main driver for optimization, running Cooper in GA mode is the most favorable option, which besides shows viable results with regard to cost. In a medium-sized scenario (A@10x), GA shows a latency reduction of 30 % and cost savings of 8 %. On the other hand, if cost minimization is the predominant motivation, ILP-NC is the first choice, saving 44 % of cost in a mid-sized scenario. If runtime is critical or if we have to deal with problems that are too large for ILP-NC, GA-NC might be a viable alternative, though showing limited improvements over the baseline.

CHAPTER 7

Conclusion

In this final chapter, we conclude the outcome of our work and summarize the main parts of this thesis. First, we outline the contributions of our proposed approach, followed by an outlook and possibilities to extend our work in the future.

7.1 Contributions

In this work, we filled an open gap for elasticity of microservices with a focus on interaction latency, especially in conjunction with sophisticated, cost-aware auto-scaling of containers and VMs in the setting of hybrid clouds. Although considerable research has been made in related work with regard to elastic container placement techniques, elasticity in hybrid clouds and optimization for certain aspects such as QoS and cost efficiency, these capabilities have not been combined with latency-aware aspects in a unified elastic approach before.

After the open research challenges have been identified, we first stated the fundamental requirements for an elastic system that is concerned with hybrid clouds, cost efficiency and latency minimization. Based on that, we proposed a solid design to address those requirements in an integrated system model which lays the groundwork for our implementation. Besides the design of our MAPE-K-based middleware, we introduced an ILP formulation to express the underlying optimization problem.

With Cooper, we provided a reference implementation and evaluation testbed for our approach, that facilitates running experimental scenarios without cloud infrastructure dependencies. Moreover, as significant contribution of our work, we implemented two different optimization approaches: An exact ILP optimizer and a heuristic approach based on a Genetic Algorithm, that can be used and studied in Cooper. The outcome of our work is a CaaS middleware that fulfills hybrid scaling decisions for VMs and containers and optimizes the elastic system for the main objectives of cost minimization and latency reduction between communication-affine microservices. Moreover, it considers the aspects of hybrid cloud infrastructures including cloud bursting.

Finally, we conducted a quantitative evaluation of our proposed approach. Therefore we defined multiple evaluation scenarios covering different problem sizes and characteristics, based on which we compared the results of the competing optimization approaches with regard to efficacy and efficiency. Moreover, we identified the benefits and shortcomings of the implemented optimization approaches in terms of result quality, runtime and feasibility depending on the problem size, as well as possible trade-offs regarding these aspects. In conclusion, the gathered results carry motivation for further research in future work.

7.2 Future Work

Based on the designed approach and its implementation, our work can be used as a solid foundation for further extensions and research, as outlined in the following sections.

Middleware Design

Static assumptions in the current design of Cooper could be superseded in a way, that the actual system measures are monitored and considered. This applies, e.g., for the services' grace period and the startup time of containers. Likewise, the configuration parameters employed in our system model with regard to data centers, services and containers can be determined at runtime by monitoring the actual infrastructure and services. I.e., network latencies could be measured periodically, and the affinity between services could be determined based on actual recorded inter-service requests. Concerning the leasing cost of VMs, the current static parameterization can be extended to consider different pricing models, such as feeding the middleware with current prices of bid-based pricing models.

Generally, the proposed design of our middleware follows a reactive approach as to the analysis of monitored data. This could be enhanced by a proactive mode, which employs prediction models that anticipate future changes of, e.g., workload or prices, to achieve improved behaviour of our elastic system.

Regarding containers, the assumed maximum load capacity with regard to QoS could be derived by monitoring and analyzing the relevant metrics of differently sized containers, such as their response times, resource utilization and processed load. Furthermore, the current model of predefined, discrete container types could be replaced by continuously parameterized container instances. This would also add importance and room for maneuver to the vertical scaling aspect for containers.

The design of our proposed approach currently negates the temporal aspect of optimization in the course of scheduling cycles. To become a serviceable CaaS middleware, the duration of performed optimizations should be considered in a way that the necessary steps are taken when the system state changes significantly in the meantime, or a long-running optimization overlaps with the subsequent scheduling cycle, for example.

In future extensions of the middleware, its general design could be adapted to account for additional factors with regard to cost and infrastructure usage, e.g., by exploring the aspect of data transfer cost between data centers, network bandwidth, or other related (provider-specific) aspects in the context of hybrid clouds.

Implementation and Integration

To become fully operational for running actual applications in the cloud, Cooper needs to be extended to interact with cloud infrastructures, e.g., by providing integrations with IaaS offerings of public CSPs.

For container orchestration and service discovery, other existing solutions can be leveraged in order to achieve the tasks of deployment, configuration and the execution of scaling actions, as well as client-side load balancing and monitoring based on well-known existing frameworks. Possible candidates for integrations may be the open-source software projects Kubernetes, Istio or HashiCorp Nomad/Consul. Their existing capabilities (e.g., auto-scaling components in Kubernetes) could be reused and extended with custom implementations where necessary.

Regarding service discovery and load balancing, the decision-making for the actual routing of requests to target containers is crucial when it comes to latency minimization. Therefore, the right strategies towards latency-aware load balancing need to be applied and implemented to be carried out by the middleware.

Finally, the simulation of service interaction employed in this work could be completely replaced by monitoring of the actual exchanged messages in the network, based on data gathered from container orchestration infrastructure and client-side load balancing.

Optimization

In general, the calibration of parameters for the contributed optimization approaches could be further improved. For example, parameterization regarding the weights of the objective/fitness function or parameters used in the recombination and mutation steps of the Genetic Algorithm should be further explored. Particularly, more sophisticated fine-tuning of parameters depending on the problem size may be promising, and could be applied for more parameters than in our current approach. For this purpose, strategies to derive suitable values for parameters based on the problem size or related metrics could be developed.

With increasing problem size, the Genetic Algorithm of the heuristic approach converges towards the results of the First-Fit strategy. To counter this flaw, the Genetic Algorithm deserves further development allowing it to reduce the stake of the First-Fit algorithm during the reparation of individuals, or to dump it completely. Therefore, future work could adapt the algorithm's crucial parts such as the fitness function, initialization and repairing algorithms. We can also think of making fundamental changes to the encoding strategy.

The exact optimization approach is generally expected to raise performance bottlenecks when tasked with larger scenarios. However, it may be worthwhile to further develop the ILP model, or even employ different optimization solving techniques such as Quadratic Programming that may be more suitable for the co-location aspect, in order to achieve faster runtimes. Thereby, an exact approach considering latency-awareness may become feasible for larger scenarios than we witnessed in our evaluation. Besides, further improvements of the ILP model's equations could be addressed in general, as well as tuning the ILP solver and parameterizing it with regard to trade-offs towards performance.

Additional promising potential could be incorporated by adding further heuristics to the currently provided approaches. Specifically, we suggest that an "incremental" optimization resolution could be restricted to a subset of the problem space, i.e., spanning only a limited number of entities at the time of optimization. Comparable to the *baseload* of a system, the larger part of the problem space would be considered stable, and thus untouched, while the actual optimization covers only the volatile "edge" of the problem space. A challenge of this approach would be to determine a proper strategy for drawing such border of the optimization problem. Effectively, such heuristic could lead to significantly reduced runtimes in the ILP-based approach, and improve efficacy of the Genetic Algorithm, which showed better results for smaller problem sizes.

APPENDIX A

Configuration Files

The following listings show exemplary YAML configuration files used for the parameterization of Cooper, based on actual parameters from our evaluation.

Listing A.1: General application configuration parameters

```yaml
cooper:
  scenario: scenario-a
  loadMultiplicator: 10
  scenarioMultiplicator: 1.5
  loadFixture: load-fixture/scenario-a@10x.csv
  evaluationOutput: evaluation/scenario-a@10x_GA.csv
  optimization:
    strategy: GA
    gaLatencyWeight: 0.001
```

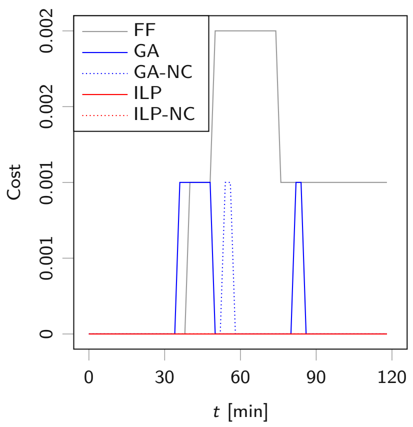
Listing A.2: Scenario configuration parameters (based on evaluation scenario A)

```yaml
dataCenters:
  DC-Private:
    onPremise: true
    instanceTypes:
      - label: 4.large
        cpuCores: 4
        memory: 8GB
        count: 2
  DC-1:
    instanceTypes:
      - label: 2.small
        cpuCores: 2
        memory: 2GB
        cost: 0.033
      - label: 2.medium
        cpuCores: 2
        memory: 4GB
        cost: 0.056
      - label: 4.large
        cpuCores: 4
        memory: 8GB
        cost: 0.099
      - label: 4.xlarge
        cpuCores: 4
        memory: 16GB
        cost: 0.182
  DC-2:
    instanceTypes:
      - label: 2.small
        cpuCores: 2
        memory: 2GB
        cost: 0.027
      - label: 2.medium
        cpuCores: 2
        memory: 4GB
        cost: 0.052
      - label: 4.large
        cpuCores: 4
        memory: 8GB
        cost: 0.1
      - label: 4.xlarge
        cpuCores: 4
        memory: 16GB
        cost: 0.19

```

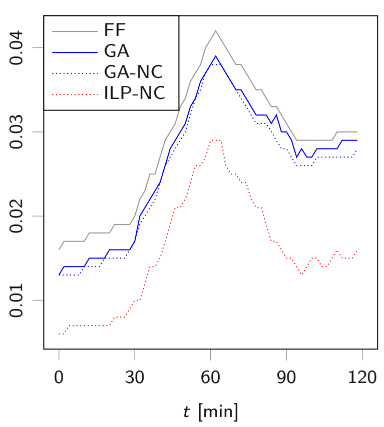
```yaml
distance:
  - a: DC-Private
    b: DC-Private
    latency: 10
  - a: DC-1
    b: DC-1
    latency: 15
  - a: DC-2
    b: DC-2
    latency: 15
  - a: DC-1
    b: DC-2
    latency: 80
  - a: DC-1
    b: DC-Private
    latency: 80
  - a: DC-2
    b: DC-Private
    latency: 120

services:
  service-gw:
    downstreamServices:
      service-a: 0.5
      service-b: 0.1
    containerConfigurations:
      - label: 1000@512
        rpmCapacity: 1000
        cpuUnits: 1024
        memory: 512MB
      - label: 2000@1024
        rpmCapacity: 2000
        cpuUnits: 1536
        memory: 1024MB
      - label: 4000@2048
        rpmCapacity: 4000
        cpuUnits: 2048
        memory: 2048MB
      - label: 8000@4096
        rpmCapacity: 8000
        cpuUnits: 2560
        memory: 4096MB
  service-a:
    downstreamServices:
      service-b: 0.5
    containerConfigurations:
      - label: 600@1024
        rpmCapacity: 600
        cpuUnits: 512
        memory: 1024MB
      - label: 900@2048
        rpmCapacity: 900
        cpuUnits: 1024
        memory: 2048MB
      - label: 1350@4096
        rpmCapacity: 1350
        cpuUnits: 1536
        memory: 4096MB
      - label: 2025@8192
        rpmCapacity: 2025
        cpuUnits: 2048
        memory: 8192MB
  service-b:
    containerConfigurations:
      - label: 600@1024
        rpmCapacity: 600
        cpuUnits: 512
        memory: 1024MB
      - label: 900@2048
        rpmCapacity: 900
        cpuUnits: 1024
        memory: 2048MB
      - label: 1350@4096
        rpmCapacity: 1350
        cpuUnits: 1536
        memory: 4096MB
      - label: 2025@8192
        rpmCapacity: 2025
        cpuUnits: 2048
        memory: 8192MB
```

APPENDIX B

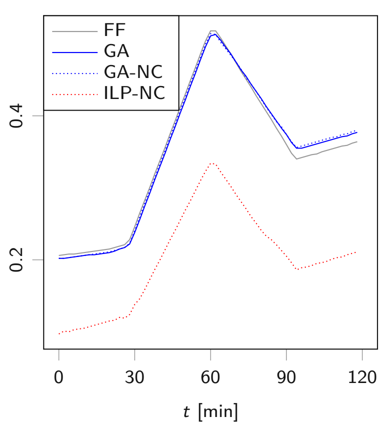
Additional Result Plots



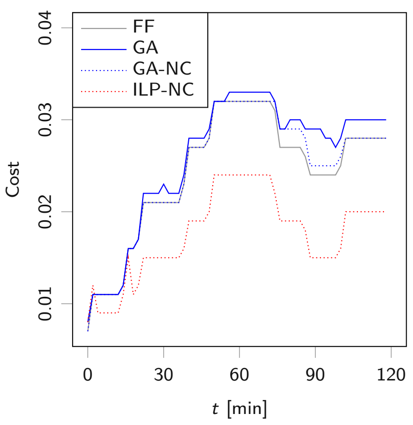
(a) Scenario A@1x



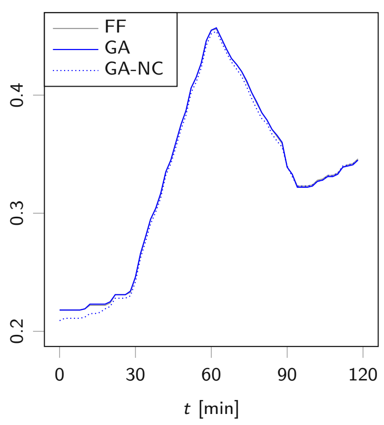
(b) Scenario A@10x



(c) Scenario A@100x

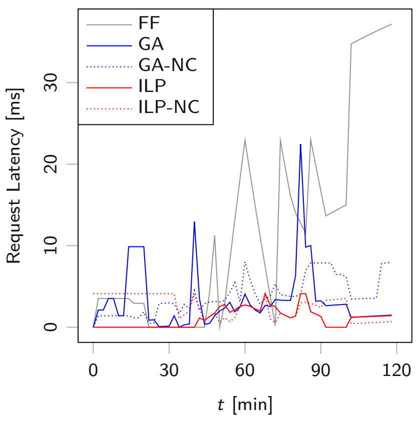


(d) Scenario B@1x

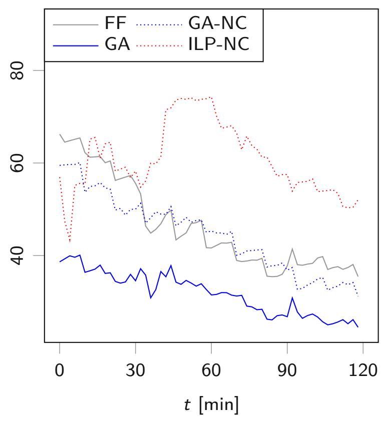


(e) Scenario B@10x

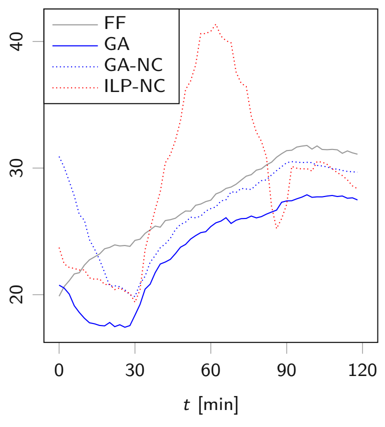
Figure B.1: Incurred cost per sample period



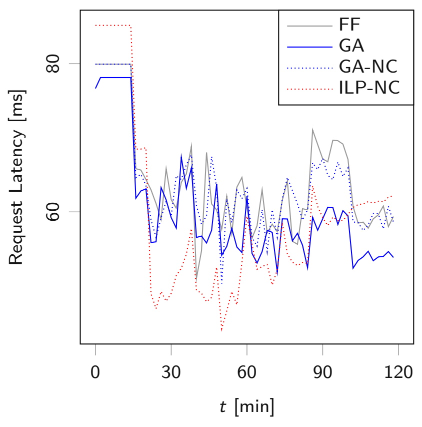
(a) Scenario A@1x



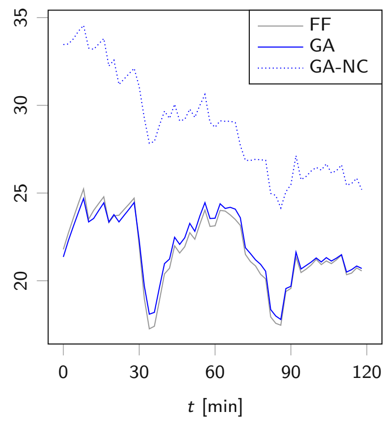
(b) Scenario A@10x



(c) Scenario A@100x



(d) Scenario B@1x



(e) Scenario B@10x

Figure B.2: Average request latency per sample period

List of Figures

List of Tables

List of Listings

Acronyms

MILP Mixed Integer Linear Programming. 23, 55

NIST National Institute of Standards and Technology. 8, 9, 11, 16

OS Operating System. 10, 12–14, 41

PaaS Platform as a Service. 1, 2, 9, 10, 15, 28

PM Physical Machine. 12, 14, 17, 30, 32, 33, 41

QoS Quality of Service. 18, 30, 31, 33, 38, 39, 43, 45, 48, 52, 54, 57, 58, 66, 71–73, 91, 92

REST Representational State Transfer. 20

RPC Remote Procedure Call. 19

SLA Service Level Agreement. 11

SOA Service-Oriented Architecture. 19, 21

VM Virtual Machine. 1–4, 12–15, 17, 28–45, 48, 49, 51–60, 64–66, 69–72, 75, 76, 79, 80, 82, 85, 87, 91, 92, 103, 105

WWW World Wide Web. 20

YAML YAML Ain't Markup Language (Yet Another Markup Language). 64, 95

Bibliography

[1] Samiha Islam Abrita, Moumita Sarker, Faheem Abrar, and Muhammad Abdullah Adnan. „Benchmarking VM Startup Time in the Cloud". In: *First BenchCouncil International Symposium on Benchmarking, Measuring, and Optimizing (Bench '18), Revised Selected Papers*. Springer International Publishing, October 2019, pp. 53–64. ISBN: 978-3-030-32813-9. DOI: `10.1007/978-3-030-32813-9_6`.

[2] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. „Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER". In: *Proceedings of the IEEE 10th International Conference on Cloud Computing (CLOUD '17)*. June 2017, pp. 472–479. DOI: `10.1109/CLOUD.2017.67`.

[3] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. „Elasticity in Cloud Computing: State of the Art and Research Challenges". In: *IEEE Transactions on Services Computing* 11.2 (March 2018), pp. 430–447. ISSN: 2372-0204. DOI: `10.1109/TSC.2017.2711009`.

[4] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. „Performance Evaluation of Microservices Architectures Using Containers". In: *Proceedings of the IEEE 14th International Symposium on Network Computing and Applications (NCA '15)*. September 2015, pp. 27–34. DOI: `10.1109/NCA.2015.49`.

[5] Amazon Web Services. *Amazon EC2 Auto Scaling*. URL: `https://aws.amazon.com/ec2/autoscaling` (Accessed: 15 May 2020).

[6] Amazon Web Services. *Amazon EC2 On-Demand Pricing*. URL: `https://aws.amazon.com/about-aws/whats-new/2017/10/announcing-amazon-ec2-per-second-billing` (Accessed: 30 August 2020).

[7] Amazon Web Services. *Amazon Elastic Container Service*. URL: `https://aws.amazon.com/ecs` (Accessed: 15 May 2020).

[8] Amazon Web Services. *Amazon Elastic Kubernetes Service*. URL: `https://aws.amazon.com/eks` (Accessed: 15 May 2020).

[9] Amazon Web Services. *Announcing Amazon EC2 per second billing*. 2017. URL: `https://aws.amazon.com/about-aws/whats-new/2017/10/announcing-amazon-ec2-per-second-billing` (Accessed: 30 August 2020).

[10] Amazon Web Services. *AWS Fargate*. URL: https://aws.amazon.com/fargate (Accessed: 15 May 2020).

[11] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. „Microservices Architecture Enables DevOps: an Experience Report on Migration to a Cloud-Native Architecture“. In: *IEEE Software* 33.3 (May 2016), pp. 42–52. DOI: 10.1109/MS.2016.64.

[12] Jeff Barr. *New – Predictive Scaling for EC2, Powered by Machine Learning*. 2018. URL: https://aws.amazon.com/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning (Accessed: 15 May 2020).

[13] Hayet Brabra, Achraf Mtibaa, Walid Gaaloul, and Boualem Benatallah. „Model-Driven Elasticity for Cloud Resources“. In: *Proceedings of the 30th International Conference on Advanced Information Systems Engineering (CAiSE '18)*. Springer International Publishing, June 2018, pp. 187–202. ISBN: 978-3-319-91563-0. DOI: 10.1007/978-3-319-91563-0_12.

[14] Gerald Brown and Robert Dell. „Formulating Integer Linear Programs: A Rogues' Gallery“. In: *INFORMS Transactions on Education* 7.2 (January 2007), pp. 153–159. DOI: 10.1287/ited.7.2.153.

[15] Rajkumar Buyya, Satish Narayana Srirama, Giuliano Casale, Rodrigo Calheiros, Yogesh Simmhan, Blesson Varghese, Erol Gelenbe, Bahman Javadi, Luis Miguel Vaquero, Marco A. S. Netto, et al. „A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade“. In: *ACM Computing Surveys* 51.5 (November 2018). ISSN: 0360-0300. DOI: 10.1145/3241737.

[16] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. „Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility“. In: *Future Generation Computer Systems* 25.6 (June 2009), 599–616. ISSN: 0167-739X. DOI: 10.1016/j.future.2008.12.001.

[17] Rajkumar Buyya, Maria Alejandra Rodriguez, Adel Nadjaran Toosi, and Jaeman Park. „Cost-Efficient Orchestration of Containers in Clouds: A Vision, Architectural Elements, and Future Directions“. In: *Journal of Physics: Conference Series* 1108 (November 2018). DOI: 10.1088/1742-6596/1108/1/012001.

[18] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. „Optimal Operator Placement for Distributed Stream Processing Applications“. In: *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems (DEBS '16)*. Association for Computing Machinery, June 2016, 69–80. ISBN: 978-1-45034021-2. DOI: 10.1145/2933267.2933312.

[19] Emiliano Casalicchio. „Container Orchestration: A Survey“. In: *Systems Modeling: Methodologies and Tools*. Ed. by Antonio Puliafito and Kishor S. Trivedi. Springer International Publishing, 2019, pp. 221–235. ISBN: 978-3-319-92378-9. DOI: 10.1007/978-3-319-92378-9_14.

[20] Emiliano Casalicchio and Stefano Iannucci. „The state-of-the-art in container technologies: Application, orchestration and security“. In: *Concurrency and Computation: Practice and Experience* 32.2 (January 2020). DOI: 10.1002/cpe.5668.

[21] Emiliano Casalicchio and Vanessa Perciballi. „Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics“. In: *Proceedings of the IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W '17)*. September 2017, pp. 207–214. DOI: 10.1109/FAS-W.2017.149.

[22] Jianhai Chen, Kevin Chiew, Deshi Ye, Liangwei Zhu, and Wenzhi Chen. „AAGA: Affinity-Aware Grouping for Allocation of Virtual Machines“. In: *Proceedings of the IEEE 27th International Conference on Advanced Information Networking and Applications (AINA '13)*. March 2013, pp. 235–242. ISBN: 978-1-4673-5550-6. DOI: 10.1109/AINA.2013.22.

[23] Cloud Native Computing Foundation. *Kubernetes-based Event Driven Autoscaling*. URL: https://github.com/kedacore/keda (Accessed: 15 May 2020).

[24] Edward G. Coffman, Michael R. Garey, and David S. Johnson. „Approximation Algorithms for Bin Packing: A Survey“. In: *Approximation Algorithms for NP-Hard Problems*. Ed. by Dorit S. Hochbaum. PWS Publishing Co., 1996, 46–93. ISBN: 978-0-534-94968-6.

[25] Reuven Cohen. *Defining Elastic Computing*. 2009. URL: http://www.elasticvapor.com/2009/09/defining-elastic-computing.html (Accessed: 28 June 2020).

[26] Inc. Docker. *Runtime options with Memory, CPUs, and GPUs*. URL: https://docs.docker.com/config/containers/resource_constraints (Accessed: 13 April 2021).

[27] Docker, Inc. *Swarm mode overview*. URL: https://docs.docker.com/engine/swarm (Accessed: 15 May 2020).

[28] Sourav Dutta, Sankalp Gera, Akshat Verma, and Balaji Viswanathan. „SmartScale: Automatic Application Scaling in Enterprise Clouds“. In: *Proceedings of the IEEE 5th International Conference on Cloud Computing (CLOUD '12)*. June 2012, pp. 221–228. ISBN: 978-1-4673-2892-0. DOI: 10.1109/CLOUD.2012.12.

[29] Hector Fernandez, Guillaume Pierre, and Thilo Kielmann. „Autoscaling Web Applications in Heterogeneous Cloud Infrastructures“. In: *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E '14)*. March 2014, pp. 195–204. DOI: 10.1109/IC2E.2014.25.

[30] Massimo Re Ferre and Samuel Karp. *How Amazon ECS manages CPU and memory resources*. 2019. URL: https://aws.amazon.com/de/blogs/containers/how-amazon-ecs-manages-cpu-and-memory-resources (Accessed: 13 April 2021).

[31] Lance Fortnow and Steve Homer. „A Short History of Computational Complexity“. In: *Bulletin of the EATCS* 80 (January 2003), pp. 95–133.

[32] Martin Fowler. *Domain-Driven Design*. 2020. URL: https://martinfowler.com/bliki/DomainDrivenDesign.html (Accessed: 3 August 2020).

[33] Martin Fowler. *Microservice Trade-Offs*. 2015. URL: https://martinfowler.com/articles/microservice-trade-offs.html (Accessed: 20 June 2020).

[34] Google Cloud. *Autoscaling groups of instances*. URL: https://cloud.google.com/compute/docs/autoscaler (Accessed: 15 May 2020).

[35] Google Cloud. *Google Kubernetes Engine*. URL: https://cloud.google.com/kubernetes-engine (Accessed: 15 May 2020).

[36] Google Cloud. *VM instances pricing*. URL: https://cloud.google.com/compute/vm-instance-pricing (Accessed: 30 August 2020).

[37] Xinjie Guan, Xili Wan, Baek-Young Choi, Sejun Song, and Jiafeng Zhu. „Application Oriented Dynamic Resource Allocation for Data Centers Using Docker Containers“. In: *IEEE Communications Letters* 21.3 (March 2017), pp. 504–507. ISSN: 2373-7891. DOI: 10.1109/LCOMM.2016.2644658.

[38] Carlos Guerrero, Isaac Lera, and Carlos Juiz. „Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications“. In: *The Journal of Supercomputing* 74 (July 2018), pp. 2956–2983. DOI: 10.1007/s11227-018-2345-2.

[39] Rui Han, Moustafa M. Ghanem, Li Guo, Yike Guo, and Michelle Osmond. „Enabling cost-aware and adaptive elasticity of multi-tier cloud applications“. In: *Future Generation Computer Systems* 32 (March 2014), pp. 82–98. ISSN: 0167-739X. DOI: 10.1016/j.future.2012.05.018.

[40] HashiCorp. *Nomad Autoscaler*. URL: https://github.com/hashicorp/nomad-autoscaler (Accessed: 15 May 2020).

[41] Sara Hassan, Rami Bahsoon, and Rick Kazman. „Microservice transition and its granularity problem: A systematic mapping study“. In: *Software: Practice and Experience* 50.9 (June 2020), pp. 1651–1681. DOI: 10.1002/spe.2869.

[42] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. „Performance Engineering for Microservices: Research Challenges and Directions“. In: *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering Companion (ICPE '17)*. Association for Computing Machinery, April 2017, 223–226. ISBN: 978-1-45034899-7. DOI: 10.1145/3053600.3053653.

[43] Nikolas Herbst, Samuel Kounev, and Ralf Reussner. „Elasticity in Cloud Computing: What it is, and What it is Not“. In: *Proceedings of the 10th International Conference on Autonomic Computing (ICAC '13)*. June 2013, pp. 23–27. ISBN: 978-1-931971-02-7.

[44] Philipp Hoenisch, Ingo Weber, Stefan Schulte, Liming Zhu, and Alan Fekete. „Four-Fold Auto-Scaling on a Contemporary Deployment Platform Using Docker Containers“. In: *Proceedings of the 13th International Conference on Service Oriented Computing (ICSOC '15)*. November 2015, pp. 316–323. DOI: 10.1007/978-3-662-48616-0_20.

[45] Jelastic, Inc. *Automatic Vertical Scaling.* URL: `https://docs.jelastic.com/automatic-vertical-scaling` (Accessed: 15 May 2020).

[46] Chanwit Kaewkasi and Kornrathak Chuenmuneewong. „Improvement of container scheduling for Docker using Ant Colony Optimization“. In: *Proceedings of the 9th International Conference on Knowledge and Smart Technology (KST '17).* February 2017, pp. 254–259. DOI: `10.1109/KST.2017.7886112`.

[47] Jeffrey O. Kephart and David M. Chess. „The Vision of Autonomic Computing“. In: *Computer* 36.1 (January 2003), 41–50. ISSN: 0018-9162. DOI: `10.1109/MC.2003.1160055`.

[48] Hamzeh Khazaei, Rajsimman Ravichandiran, Byungchul Park, Hadi Bannazadeh, Ali Tizghadam, and Alberto Leon-Garcia. „Elascale: Autoscaling and Monitoring as a Service“. In: *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering (CASCON '17).* November 2017, pp. 234–240.

[49] Won-Yong Kim, Jin-Seop Lee, and Eui-Nam Huh. „Study on Proactive Auto Scaling for Instance through the Prediction of Network Traffic on the Container Environment“. In: *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication (IMCOM '17).* Association for Computing Machinery, January 2017, pp. 1–8. ISBN: 978-1-45034888-1. DOI: `10.1145/3022227.3022243`.

[50] Mikael Koskinen, Tommi Mikkonen, and Pekka Abrahamsson. „Containers in Software Development: A Systematic Mapping Study“. In: *Proceedings of the 20th International Conference on Product-Focused Software Process Improvement (PROFES '19).* Springer International Publishing, November 2019, pp. 176–191. ISBN: 978-3-030-35333-9. DOI: `10.1007/978-3-030-35333-9_13`.

[51] Kubernetes. *Cluster Autoscaler.* URL: `https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler` (Accessed: 15 May 2020).

[52] Kubernetes. *Horizontal Pod Autoscaler.* URL: `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale` (Accessed: 15 May 2020).

[53] Kubernetes. *Vertical Pod Autoscaler.* URL: `https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler` (Accessed: 15 May 2020).

[54] Anthony Kwan, Jonathon Wong, Hans-Arno Jacobsen, and Vinod Muthusamy. „HyScale: Hybrid and Network Scaling of Dockerized Microservices in Cloud Data Centres“. In: *Proceedings of the IEEE 39th International Conference on Distributed Computing Systems (ICDCS '19).* July 2019, pp. 80–90. DOI: `10.1109/ICDCS.2019.00017`.

[55] James Lewis and Martin Fowler. *Microservices.* 2014. URL: `https://martinfowler.com/articles/microservices.html` (Accessed: 20 June 2020).

[56] David S. Linthicum. „Emerging Hybrid Cloud Patterns“. In: *IEEE Cloud Computing* 3.1 (February 2016), pp. 88–91. DOI: `10.1109/MCC.2016.22`.

[57] Zoltan Á. Mann. „Resource Optimization Across the Cloud Stack“. In: *IEEE Transactions on Parallel and Distributed Systems* 29.1 (August 2017), pp. 169–182. DOI: 10.1109/TPDS.2017.2744627.

[58] Jingjing Mao, Lulu Sun, Yi Zhang, and Jin Sun. „An Improved Genetic Algorithm for Solving Bag-of-tasks Scheduling Problems with Deadline Constraints on Hybrid Clouds“. In: *Proceedings of the IEEE International Conference on Progress in Informatics and Computing (PIC '18)*. December 2018, pp. 305–310. DOI: 10.1109/PIC.2018.8706139.

[59] Toni Mastelic and Ivona Brandic. „Recent Trends in Energy-Efficient Cloud Computing“. In: *IEEE Cloud Computing* 2.1 (April 2015), pp. 40–47. ISSN: 2325-6095. DOI: 10.1109/MCC.2015.15.

[60] Peter M. Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. Special Publication 800-145. National Institute of Standards and Technology, September 2011. DOI: 10.6028/NIST.SP.800-145.

[61] Dirk Merkel. „Docker: Lightweight Linux Containers for Consistent Development and Deployment“. In: *Linux Journal* 2014.239 (March 2014). ISSN: 1075-3583.

[62] Microsoft. *Overview of autoscale in Microsoft Azure*. 2018. URL: https://docs.microsoft.com/en-us/azure/azure-monitor/platform/autoscale-overview (Accessed: 15 May 2020).

[63] Microsoft Azure. *Azure Kubernetes Service (AKS)*. URL: https://azure.microsoft.com/en-us/services/kubernetes-service (Accessed: 15 May 2020).

[64] Microsoft Azure. *Container Instances*. URL: https://azure.microsoft.com/en-us/services/container-instances (Accessed: 15 May 2020).

[65] Microsoft Azure. *Linux Virtual Machines Pricing*. URL: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux (Accessed: 30 August 2020).

[66] Katta G. Murty. *Optimization Models For Decision Making*. Vol. 1. 2003.

[67] Matteo Nardelli, Valeria Cardellini, Vincenzo Grassi, and Francesco Lo Presti. „Efficient Operator Placement for Distributed Data Stream Processing Applications“. In: *IEEE Transactions on Parallel and Distributed Systems* 30 (2019), pp. 1753–1767.

[68] Matteo Nardelli, Christoph Hochreiner, and Stefan Schulte. „Elastic Provisioning of Virtual Machines for Container Deployment“. In: *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering Companion (ICPE '17)*. Association for Computing Machinery, April 2017, pp. 5–10. ISBN: 978-1-45034899-7. DOI: 10.1145/3053600.3053602.

[69] Matteo Nardelli, Valeria Cardellini, and Emiliano Casalicchio. „Multi-Level Elastic Deployment of Containerized Applications in Geo-Distributed Environments“. In: *Proceedings of the IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud '18)*. August 2018, pp. 1–8. DOI: 10.1109/FiCloud.2018.00009.

[70] Gerald J. Popek and Robert P. Goldberg. „Formal Requirements for Virtualizable Third Generation Architectures“. In: *Communications of the ACM* 17.7 (July 1974), 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073.

[71] M Rajkumar, Anil Pole, Vittalraya Adige, and Prabal Mahanta. „DevOps culture and its impact on cloud delivery and software development“. In: *Proceedings of the International Conference on Advances in Computing, Communication, & Automation (ICACCA Spring '16)*. April 2016, pp. 1–6. DOI: 10.1109/ICACCA.2016.7578902.

[72] Red Hat, Inc. *Applying autoscaling to an OpenShift Container Platform cluster.* URL: https://docs.openshift.com/container-platform/4.4/machine_management/applying-autoscaling.html (Accessed: 15 May 2020).

[73] Chris Richardson. *API Gateway / Backends for Frontends.* URL: https://microservices.io/patterns/apigateway.html (Accessed: 21 June 2020).

[74] Chris Richardson. *What are microservices?* URL: https://microservices.io (Accessed: 21 June 2020).

[75] Adalberto R. Sampaio, Julia Rubin, Ivan Beschastnikh, and Nelson Souto Rosa. „Improving microservice-based applications with runtime placement adaptation“. In: *Journal of Internet Services and Applications* 10 (February 2019), pp. 1–30. DOI: 10.1186/s13174-019-0104-0.

[76] Edwin Schouten. *Rapid elasticity and the cloud.* 2012. URL: https://www.ibm.com/blogs/cloud-computing/2012/09/12/rapid-elasticity-and-the-cloud (Accessed: 28 June 2020).

[77] Borja Sotomayor, Rubén Montero, Ignacio Llorente, and Ian Foster. „Virtual Infrastructure Management in Private and Hybrid Clouds“. In: *IEEE Internet Computing* 13.5 (September 2009), pp. 14–22. DOI: 10.1109/MIC.2009.119.

[78] Vlado Stankovski, Jernej Trnkoczy, Salman Taherizadeh, and Matej Cigale. „Implementing Time-Critical Functionalities with a Distributed Adaptive Container Architecture“. In: *Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services (iiWAS '16)*. Association for Computing Machinery, November 2016, pp. 453–457. ISBN: 978-1-45034807-2. DOI: 10.1145/3011141.3011202.

[79] Vijay Sundaresan, Debasish Banerjee, and Timothy Pickett. *Modernize and optimize Spring Boot applications.* 2019. URL: https://developer.ibm.com/technologies/java/articles/modernize-and-optimize-spring-boot-applications (Accessed: 8 October 2020).

[80] Boxiong Tan, Hui Ma, and Yi Mei. „A Group Genetic Algorithm for Resource Allocation in Container-Based Clouds“. In: *Proceedings of the 20th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP '20)*. Springer International Publishing, April 2020, pp. 180–196. ISBN: 978-3-030-43680-3. DOI: 10.1007/978-3-030-43680-3_12.

[81] Boxiong Tan, Hui Ma, and Yi Mei. „Novel Genetic Algorithm with Dual Chromosome Representation for Resource Allocation in Container-based Clouds“. In: *Proceedings of the IEEE 12th International Conference on Cloud Computing (CLOUD '19)*. July 2019, pp. 452–456. ISBN: 978-1-7281-2705-7. DOI: 10.1109/CLOUD.2019.00078.

[82] University of Melbourne. *ContainerCloudSim: An Environment for Modeling and Simulation of Containers in Cloud Data Centers*. URL: http://www.cloudbus.org/cloudsim/container.html (Accessed: 4 April 2021).

[83] Ruben Van Den Bossche, Kurt Vanmechelen, and Jan Broeckhove. „Online Cost-Efficient Scheduling of Deadline-Constrained Workloads on Hybrid Clouds“. In: *Future Generation Computer Systems* 29.4 (June 2013), 973–985. ISSN: 0167-739X. DOI: 10.1016/j.future.2012.12.012.

[84] VMware, Inc. *The State of Kubernetes 2020*. 2020. URL: https://tanzu.s3.us-east-2.amazonaws.com/campaigns/pdfs/VMware_State_Of_Kubernetes_2020_eBook.pdf (Accessed: 15 May 2020).

[85] Darrell Whitley. „A Genetic Algorithm Tutorial“. In: *Statistics and Computing* 4 (June 1994), pp. 65–85. DOI: 10.1007/BF00175354.

[86] Franz Wilhelmstötter. *Jenetics Library User's Manual 6.1*. 2020. URL: https://jenetics.io/manual/manual-6.1.0.pdf (Accessed: 26 December 2020).

[87] Rouan Wilsenach. *DevOpsCulture*. 2015. URL: https://martinfowler.com/bliki/DevOpsCulture.html (Accessed: 20 June 2020).

[88] Jingqi Yang, Chuanchang Liu, Yanlei Shang, Bo Cheng, Zexiang Mao, Liu Chunhong, Lisha Niu, and Chen Junliang. „A cost-aware auto-scaling approach using the workload prediction in service clouds“. In: *Information Systems Frontiers* 16 (March 2014), pp. 7–18. DOI: 10.1007/s10796-013-9459-0.

[89] Yelp. *Clusterman*. URL: https://github.com/Yelp/clusterman (Accessed: 15 May 2020).

[90] Rong Zhang, Yaxing Chen, Bo Dong, Feng Tian, and Qinghua Zheng. „A Genetic Algorithm-Based Energy-Efficient Container Placement Strategy in CaaS“. In: *IEEE Access* 7 (August 2019), pp. 121360–121373. DOI: 10.1109/ACCESS.2019.2937553.