FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Optimized Auto Scaling of Elastic Processes in the Cloud using Docker Containers

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Gerta Sheganaku
Matrikelnummer 1027018

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ass.-Prof. Dr.-Ing. Stefan Schulte
Mitwirkung: Dr. Ingo Weber

Wien, 10.10.2017

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Optimized Auto Scaling of Elastic Processes in the Cloud using Docker Containers

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Business Informatics

by

Gerta Sheganaku
Registration Number 1027018

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ass.-Prof. Dr.-Ing. Stefan Schulte
Assistance: Dr. Ingo Weber

Vienna, 10.10.2017

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Gerta Sheganaku
Westbahnstraße 1B / 17
1070 Wien
Austria

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.

New York, 5. Oktober 2017

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgements

This thesis represents the final project that completes a versatile education in computer science and business. Throughout the past years, I was granted a variety of opportunities and a lot of flexibility that helped me in pursuing versatile goals and interests. I am especially grateful for the opportunity and privilege to develop and write this thesis at Data61 in Sydney. During my time there, I often ran past a poster installed in front of the Allianz Stadium, reading "Behind every story of success is an even greater story of support", feeling very grateful for the many people that have accompanied and supported me on my journey in various ways and without whom this work would not have been possible.

First of all, I want to sincerely thank my advisor, Stefan Schulte, for having me at the Distributed Systems Group, for providing great guidance and valuable inputs throughout the thesis, and for always being very responsive to all my inquiries. I am also most grateful to my co-advisor, Ingo Weber, for providing a pleasant and productive work environment at the Architecture & Analytics Platforms Team at Data61 and being available for detailed and insightful discussions. A very special thanks goes to all my colleagues at Data61, who immediately integrated me into their team structure, showed interest in my research topic, and gave me the opportunity to also participate in other exciting research areas. I also want to thank Prof. Hilda Tellioglu and the International Office at Vienna University of Technology, for granting a mobility scholarship and co-funding part of my stay in Sydney.

I am very grateful for all the wonderful people I have met throughout my studies and the friendships that have evolved from following common goals, as well as encouraging and motivating each other. A particular thanks goes to my dear friend Gerhard Schreihans, who has not only always been a highly reliable and motivated colleague, but also a master in taking any situation with a great sense of humor. Furthermore, I want to thank my previous employer and colleagues, who were highly flexible, very understanding of any time constraints, and gave me countless opportunities to gain great experiences, valuable to both my studies and future career.

Last but foremost, I want to express my deepest and sincerest gratitude to my family, especially my loving parents for their sacrifices, encouragements, and unconditional support throughout my entire life, to always provide me an environment in which I could realize my own potentials. I particularly thank my wonderful partner, Waldemar Hummer, for always being by my side, with endless emotional support, patience, and encouragement, and for empowering me to strive for my best every day. A big thank you also goes to all my friends from Vienna and Sydney, for all the memorable moments that have added a lot of joy and balance to my life.

Abstract

With software services becoming ever more ubiquitous, organizations are increasingly relying on interconnected business processes that act as a clockwork to orchestrate the interplay of individual services. Due to the volatility of business process landscapes, the amount of data or the number of process instances which need to be handled concurrently may vary to a large extent. Therefore, it is difficult to estimate the ever-changing demand of computational resources, especially for highly volatile domains. With the advent of cloud computing and the virtually unlimited availability of computational resources in a utility-like fashion, organizations of any size and across industries are able to adjust their cloud-based infrastructure rapidly and on-demand. Besides applications, also complete business processes can be hosted on leased virtual resources, which enables the realization of so-called elastic processes, i.e., processes which are executed on scalable cloud infrastructure. Elastic processes can have various areas of application, e.g., manufacturing, banking, or healthcare, where flexible and scalable process support is of uttermost importance for being able to model, instantiate, execute, and monitor cross-organizational processes. Despite the manifold benefits of elastic processes, there is still a lack of solutions supporting them. The allocation of resources for elastic processes is done on a rather high level, i.e., services are allocated to Virtual Machines (VMs) - usually in a 1-to-1 fashion, which leads to a waste of computational resources. Since elastic processes are part of potentially huge process landscapes, optimal resource allocation and process scheduling are complex tasks, usually solved by defining an optimization model. This poses a difficult computational problem, since optimization of process tasks under given constraints is a NP-hard problem.

In order to support process owners and cloud operators, this work addresses different challenges of cost-optimized elastic process enactment, by extending existing methodologies and algorithms from the fields of elastic processes, operations research, and cloud computing. This thesis presents a novel solution that enables a fine-grained allocation of services to lightweight containers instead of complete VMs, leading to a higher level of control over leased computational resources and thereby to a better resource utilization. Optimization models and heuristics that consider resource, quality, and cost elasticity, while taking the fine-grained control into account are provided and realized as part of a newly designed container-based middleware. Extensive evaluations of the main presented approach for optimized enactments of given process landscapes under given Quality of Service (QoS) constraints show substantial cost savings and a better resource utilization while maintaining Service Level Agreements (SLAs), when compared to recently proposed VM-based approaches.

Kurzfassung

Mit der zunehmenden Verbreitung von Software-Services setzen Unternehmen vermehrt auf vernetzte Geschäftsprozesse, um das Zusammenspiel einzelner Services zu orchestrieren. Aufgrund der Volatilität von Prozess-Landschaften kann die Datenmenge sowie Anzahl an parallelen Prozessinstanzen stark variieren. Daher ist es schwierig, den sich ständig ändernden Bedarf an Rechenressourcen, insbesondere für hoch-dynamische Domänen, im Voraus abzuschätzen. Mit dem Aufkommen von Cloud-Computing und der praktisch unbegrenzten Verfügbarkeit von Rechenressourcen können Unternehmen ihre cloudbasierte Infrastruktur schnell und bedarfsgerecht anpassen. Neben Anwendungen können auch komplette Geschäftsprozesse auf geleasten virtuellen Ressourcen gehostet werden, was die Realisierung sogenannter elastischer Prozesse in einer skalierbaren Cloud-Umgebung ermöglicht. Elastische Prozesse können beispielsweise in der Produktion, im Bankwesen oder im Gesundheitswesen zum Einsatz kommen, wo flexible und skalierbare Prozessunterstützung von äußerster Wichtigkeit ist, um organisationsübergreifende Prozesse modellieren, instanziieren, ausführen und überwachen zu können. Trotz der vielfältigen Vorteile elastischer Prozesse fehlt es noch an Lösungen zu deren Unterstützung. Die Zuweisung von Ressourcen für elastische Prozesse erfolgt mit relativ grober Granularität, d. h. Dienste werden Virtuellen Maschinen (VMs) zugewiesen - gewöhnlich in einem direkten Verhältnis, was zu einer Unterauslastung von Rechenressourcen führt. Da elastische Prozesse oft ein Teil potenziell riesiger Prozesslandschaften sind, sind eine optimale Ressourcenallokation und Prozessplanung komplexe Aufgaben, die in der Regel mittels eines Optimierungsmodells gelöst werden. Dies führt zu einem schwierigen Berechnungsproblem, da die optimierte Ausführung von Prozessschritten unter gegebenen Bedingungen ein NP-hartes Problem darstellt.

Zur Unterstützung von Prozesseignern und Cloud-Operatoren befasst sich diese Arbeit mit der Erweiterung bestehender Methoden und Algorithmen aus den Bereichen elastische Prozesse, Operations Research und Cloud-Computing. Diese Arbeit stellt eine neuartige Lösung vor, die eine feingranulare Zuordnung von Diensten zu leichtgewichtigen Containern, anstelle kompletter VMs, ermöglicht. Dies führt zu einer höheren Kontrolle über gemietete Rechenressourcen und damit einhergehend zu einer besseren Ressourcenausnutzung. Optimierungsmodelle und Heuristiken, welche die Ressourcen-, Qualitäts- und Kostenelastizität, als auch die feingranulare Kontrolle berücksichtigen, werden präsentiert und als Teil einer neu entworfenen Container-basierten Middleware umgesetzt. Umfangreiche Bewertungen des vorgestellten Hauptansatzes für optimierte Inszenierungen bestimmter Prozesslandschaften unter vorgegebenen Dienstgüte-Einschränkungen zeigen erhebliche Kosteneinsparungen und eine bessere Ressourcennutzung bei gleichzeitiger Einhaltung von Service Level Agreements (SLAs) im Vergleich zu aktuell vorgeschlagenen VM-basierten Ansätzen.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

CHAPTER 1

Introduction

"Cloud is about how you do computing, not where you do computing."

– Paul Maritz, *VMware CEO*

"Cloud computing is empowering, as anyone in any part of the world with internet connection and a credit card can run and manage applications in the state of the art global data centers; companies leveraging cloud will be able to innovate cheaper and faster."

– Jamal Mazhar, *Kaavo Founder and CEO*

This chapter provides a general introduction into the topic of elastic process optimization, presents the main motivation, problem statement and aim for this work and outlines the contributions provided in this thesis.

1.1 Motivation

With the emergence of cloud computing and the theoretically nearly unlimited availability of computational resources in a utility-like fashion [24], companies and organizations across industries have the opportunity to rapidly access required resources, well adjusted to their individual and changing demand. During the past years this has revolutionized the way companies, from small start-ups to large corporations operate. Anyone has the possibility to make use of high-end IT infrastructure and increase or decrease the amount of used computational resources without the need of large upfront capital and time investments. This does not only constitute a driver for innovation, as it decreases the risks associated with new projects and businesses, but also fosters mobility and flexibility and allows for the adoption of more agile, resource efficient and cost-effective modes of operation, by facilitating fast adaptations in changing environments.

At the same time, both, wide intra- and inter-organizational cooperations in the business world, have led to the need of efficient BPM techniques tackling organizational, managerial and technical aspects of service compositions for effective business process enactment [69]. The Service-Oriented Architecture (SOA) paradigm decouples service providers from service consumers and offers dynamic binding. It can, for instance, be implemented through the use of Web Services (WSs) [45] [120] [24]. Due to ever-changing process landscapes, e.g., in terms of the number of arriving process requests or varying amount of data to be processed, the services needed to enact business processes often show a massively fluctuating demand for computational resources. Hence, the ability of cloud computing to provide computational resources in a flexible manner can provide multiple benefits for the enactment of business processes [138].

By deploying business processes in a virtual cloud infrastructure and executing them using elastic cloud resources, so-called elastic processes can be realized [44]. Elastic processes can be applied in very different domains like, e.g., banking or manufacturing, where flexible and scalable process support is of uttermost importance to be able to model, instantiate, execute and monitor cross-organizational processes. This leads to new requirements both with regard to the adaptation of the underlying process models and the runtime support of process instances by computational resources. One crucial aspect is the elasticity of IT-supported business processes: As the amount of data or the number of process instances which need to be handled concurrently may vary to a large extent, it is difficult to estimate the ever-changing demand for computational resources. This is especially the case in highly volatile domains [138].

To allow scaling of elastic processes, and meet both functional as well as non-functional requirements like given Service Level Agreements (SLAs), while considering stress factors like peak time usage, cloud resources need to be allocated and deallocated or reconfigured dynamically on demand. Scaling can be achieved either vertically or horizontally [106], while horizontal scaling describes a system's ability to adapt computing power by replicating or adding/removing system components, whereas vertical scaling refers to changing the computing power of individual system components. In general, horizontal scaling in a cloud computing context can be achieved by leasing and releasing VM instances. To decrease the time and resource overhead associated with leasing new VMs, recently a new layer of abstraction as an alternative to traditional operating system virtualization has emerged, with the introduction of lightweight container engines such a Docker[1] or CoreOS rkt[2]. Those containers allow isolating the execution of single processes, which provides benefits like resource allocation and network, memory or filesystem isolation without the additional overhead of deploying a full Operating System (OS) [12] [105]. Furthermore, next to additional possibilities for applying horizontal scaling techniques, containers also facilitate vertical scaling in virtualized environments, as the allocated resources of a container can easily be adapted during runtime. In contrast, dynamically adjusting the memory allocation of an OS that runs within a VM has traditionally been difficult, often requiring a full reboot of the OS [92].

[1] https://www.docker.com/

[2] https://coreos.com/rkt

1.2 Example Scenario

To illustrate the manifold application areas for elastic BPM, we describe a simplified example scenario from the automotive industry.

When first looking at a real world example like the BMW Group and analyzing their annual report from 2015 [54] we can see that the automative industry is currently making big efforts towards digitalization and connectivity of more traditional areas like production and logistics for manufacturing more flexibly and efficiently. Furthermore, automobile groups often operate in multifaceted business areas. Besides production and retail processes, the BMW Group also have to manage a huge number of financial processes, as they offer different financial services like leasing and credit contracts to retail customers. Leased products represent one of the largest assets on the balance sheet.

Figure 1.1 shows an international fictional car manufacturer with several worldwide distributed manufacturing sites, retail outlets, repair shops and customer centers in different areas of the world. Each area hosts their own data centers in form of a private cloud. As many of the manufacturers' business and production processes are similar all over the world, it also spans a broader private cloud over all worldwide data centers, providing services and business processes used in multiple locations. In peak times, the manufacturers' computational resources can reach a limit, therefore additional resources can be leased from a public cloud when needed.

The car manufacturer makes use of a wide range of business processes and services. Some examples of the services and processes that benefit from a cloud-based elastic process enactment are the following [51]:

- For the design of new vehicles, the car manufacturer often collaborates with worldwide offices and design processes often incorporate multiple users in multiple locations requesting similar services at the same time. During such processes, there is often the need for some compute-intensive tasks like image rendering. Services offering such tasks may be needed in multiple design processes and can often be reused.

- Before starting the manufacturing of a new car, a great number of compute-intensive crash-test simulations need to be performed. Such simulations can consist of a multitude of reusable sub-simulations.

- Assuming the car manufacturer operates modern factories and makes use of flexible production networks, there will be a sheer number of manufacturing processes that need cloud-based load balancing techniques. Again, during those manufacturing processes some process steps are very compute-intensive like image rendering or flexible production-based calculations.

- The car manufacturer furthermore makes use of both advanced real-time analytics processes as well as hourly, daily, and monthly data processing, reporting and analytics for the manufacturing processes and plant operations. The generated data is used in a multitude of employed automated quality assurance processes including automated feedback to production plant networks.

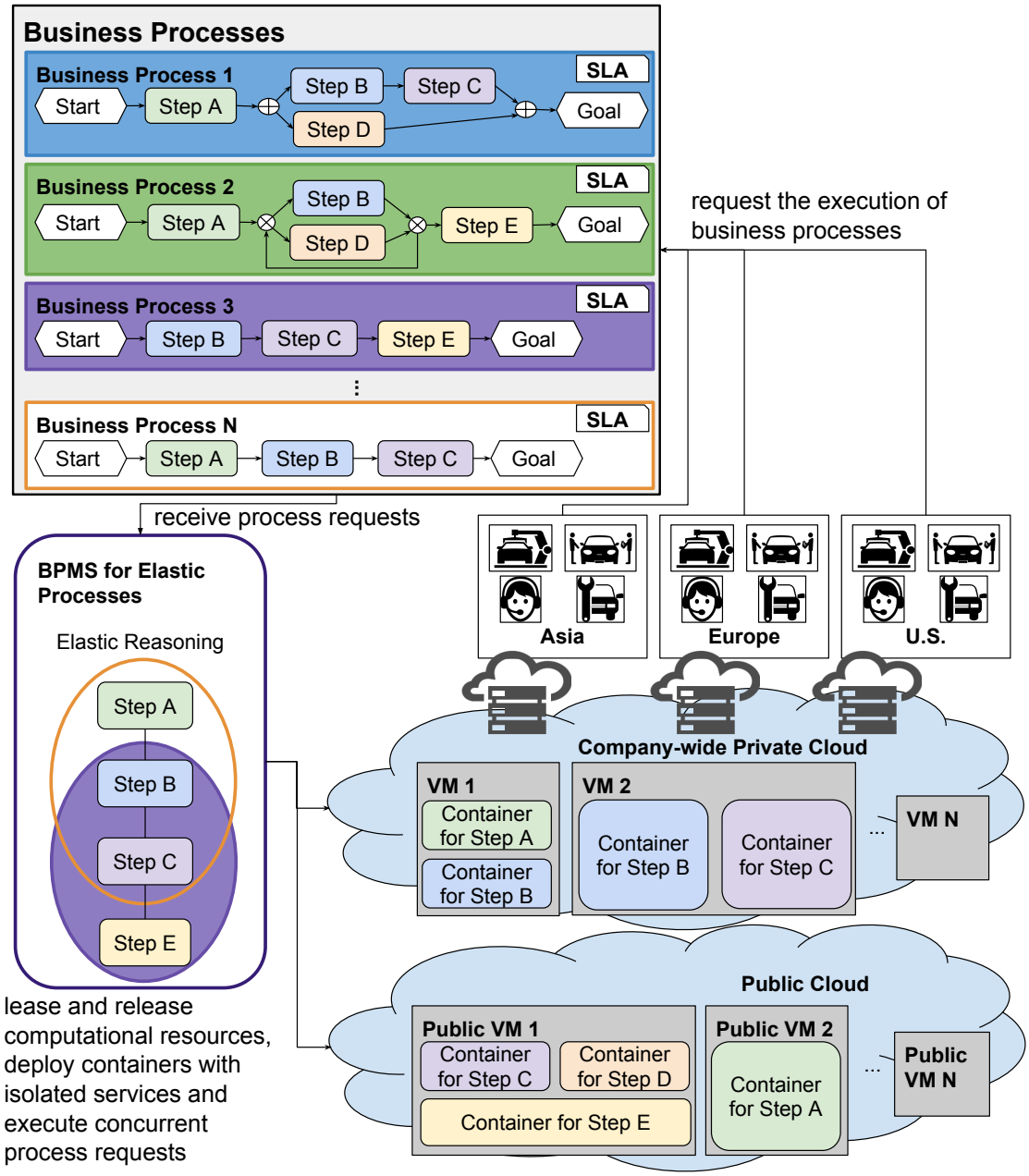


Figure 1.1: Example Scenario

- The manufacturer develops a prototype of a self-driving car. For the monitoring and operation processes of those cars, digital maps with the real-time vehicle and sensor data are used.

- Traditional cars are also equipped with Internet of Things (IoT) features. Digital maps are used in services that enhance customer experience, like automated recommender services, based on real-time monitored driver behavior, also connecting the real-time data to third party applications.

- To provide a greater customer satisfaction, the manufacturer deploys processes for automated vehicle servicing based on monitored sensor data as well as small to large automated software updates that are rolled out periodically.

- The car manufacturer also employs processes to support its financial services, as a large number of the vehicles are leased to retail customers. Important services that are often reused in customer related processes are general customer management services and credit approval and risk assessment services.

- Like in most other domains, automated backup processes are employed that collect and persist data from multiple data sources.

As can be seen from the example processes, the car manufacturer has a huge process landscape with short and long running processes of different priority, some real-time processes as well as processes that can be postponed into the future. During peak times many of the processes are instantiated and need to run concurrently, while some steps can be scheduled for a later execution. When executing the services also privacy issues as well as governmental regulations have to be taken into consideration. This is why certain data can only be processed in certain locations and may not be transferred to the cloud. The available services can be used in multiple processes and different context. The amount of processed data and the needed computing power are hard to predict. This is why the car manufacturer can make use of the benefits of elastic process enactment, i.e., minimizing the amount of used resources.

To satisfy the aforementioned requirements, the company needs a BPMS as depicted in Figure 1.1 that takes all the different cost, quality, and resource constraints into account and allows to allocate computational resources and schedule multiple concurrent business process requests on the leased resources in an efficient and effective way. A more detailed explanation of the most important elements of the example scenario as shown in Figure 1.1 can be found in Chapter 4, with a more detailed illustration in Section 4.3.

It should be noted that the automotive industry is only one of many possible application domains for elastic processes. Elastic business process enactment can be applied to any domain that has a large, volatile process landscape, with processes that are composed of a number of different services and a fluctuating number of process instances that need to be enacted in certain time frames [138].

Another very typical application domain can, for example, be found in the financial services sector [71] [87], e.g., banks are known to have huge process landscapes, while their IT costs amount to around 15% to 20% of operating expenses [95]. Other application areas comprise

e.g. the healthcare domain [6] [98], the insurance domain [72], or the manufacturing industry as a whole [136]. Most of these volatile domains can profit from the realization of elastic processes, as they allow for a more efficient use of the IT budget, by minimizing costs spent on computational resources.

1.3 Problem Statement

As elastic processes can make use of advanced techniques in virtualization, containerization, and resource allocation, they can offer a large number of benefits. While the concepts of SOA, cloud computing, and BPM have been trending and widely studied for over a decade, there is still a lack of solutions supporting cloud-based elasticity with regard to business process executions. The main reason for this is believed to be the fact that findings from cloud-based application provisioning cannot be directly mapped to the area of BPM without further adaptations and the necessary development of new techniques also considering the state of business processes, e.g., with regard to their data and control flows [138].

While automated resource allocation and service scheduling are important research problems for cloud computing [22], only very few approaches take elastic processes into account. Current approaches mostly focus on single services or Scientific Workflows (SWFs) [138]. Findings from those domains are valuable but can not be directly mapped to elastic processes. While approaches that focus on single services and web applications in the cloud [22] [65] [86] [162] do not account for the process perspective, approaches using cloud resources for Scientific Workflows [67] [78] [119] mostly only consider single workflows and are more data flow oriented, while business process scheduling is applied for complex landscapes of concurrent process requests that are very control flow driven and often share the same services [93].

In terms of elasticity measures, previous research mostly regarded resource elasticity and the adjustment of machine power in isolation, without also considering other important and very interdependent elasticity measures like cost and quality elasticity. Resource elasticity looks at the change of used resources, e.g., CPU and RAM, while cost elasticity accounts for the change of costs, e.g., regarding pricing models for on-demand instances or spot instances in the Amazon cloud[3]. Quality elasticity focuses on the change of the provided Quality of Service (QoS), e.g., availability, reliability or throughput [144]. When designing a system for enacting elastic processes those trade-offs should be taken into account [44].

In existing approaches for elastic processes [62] [63], the allocation of resources is done on a rather coarse-grained level, i.e., services are allocated to VMs – usually in a 1-to-1 fashion, which may lead to a waste of computational resources and also poses a challenge for the application of vertical scaling approaches. The fact that VMs are rather heavy-weight means of virtualization puts a limit on the ability to provide a highly dynamic execution environment with optimized resource utilization.

Regarding the more fine-grained control that can be achieved by applying light-weight containerized architectures, optimization models for finding optimal resource allocation solutions for single web applications using containers have been proposed [65], but, as already mentioned, such models do not take the process perspective into consideration.

[3] https://aws.amazon.com/en/ec2/pricing/

Since elastic processes are part of potentially huge process landscapes, optimal resource allocation and process scheduling is a complex task which usually requires to define and solve a formal optimization model [63]. The computational complexity of such optimization models is crucial for the time required by the solver to find an optimal solution. It has been shown that optimization for process tasks under given constraints is an NP-hard problem [138] which cannot be solved in polynomial time (if P $\neq$ NP).

1.4 Aim of the Work and Scientific Contribution

In order to support process owners and cloud operators, this thesis addresses different challenges, which build the foundation for cost-optimized elastic process enactment taking into consideration trade-offs with regard to quality and resource constraints. The focus is on extending existing methodologies and algorithms from the fields of elastic processes, operations research, and cloud computing.

We propose an optimization model and a runtime environment for the execution of elastic processes in a containerized architecture that optimizes the enactment of given process landscapes under given QoS constraints, in particular, process deadlines. An implementation and evaluation of the presented approach are provided. The goal is to offer a solution for cost-efficient resource allocation and process scheduling for the enactment of complex elastic processes.

For enacting process landscapes we use a research prototype able to launch and terminate VMs in a public cloud (Amazon Elastic Compute Cloud (EC2)) as well as in an OpenStack-based private cloud. The prototype was recently extended in order to deploy, launch and resize Docker containers on VMs. This allows the implementation of Docker-based service deployments, i.e., services, which are the basic building steps for processes, can be deployed as Docker containers and invoked concurrently in different process instances. State-of-the-art approaches, like those presented in [62] or [63], are based on a direct assignment of services to VMs, often leading to a waste of computational resources since VMs are not fully utilized. In comparison to the previous work, the optimization approach presented in this thesis makes use of fine-grained control over computational resources while using a containerized architecture.

Therefore, the primary goal of the thesis is the development and evaluation of an according cost-based optimization model that allows conducting a more dynamic resource allocation and scheduling for elastic processes. Compared to the current state-of-the-art, we make use of both horizontal as well as vertical scaling techniques. At the same time, a significant level of complexity is added. Where previously services were mapped to VMs, scheduling and resource allocation now need to be considered both between services and containers as well as between containers and VMs.

We model and solve the problem as a linear programming problem, and discuss other heuristic approaches.

1.5 Methodological Approach

The methodological approach for achieving a cost-optimized elastic process enactment by making use of fine-grained resource control comprises four main parts:

- First, it is necessary to conduct a survey of related work on models for cloud architecture and application topologies, with a focus on containerized architectures and lightweight container-based service placement. General approaches for placing applications in Docker containers will be examined. We illustrate the advantages and disadvantages of different resource allocation and scheduling techniques for services and business processes. Based on these theoretical concepts, a solution which allocates services for the realization of elastic processes to lightweight containers instead of complete VMs is designed.

- Second, an optimization model is conceptualized to allocate resources to services in a cost-efficient way. Optimization algorithms and heuristics for service and container placements on cloud infrastructure will be investigated. The optimization concepts will be ported to and extended for elastic processes.

  The optimization model will take into account:

  - QoS constraints, especially process deadlines and execution times
  - Vertical scaling of Docker containers
  - Horizontal scaling of VM resources and Docker containers
  - Different workflow patterns [153]:
    * sequences
    * parallel splits (AND-splits)
    * synchronization (AND-join)
    * exclusive choice (XOR-splits)
    * simple merge (XOR-join)
    * arbitrary cycles (loop)

- Third, for a proof of concept the developed optimization model and runtime environment for resource allocation and process scheduling of elastic processes is integrated into a corresponding middleware for managing VM and container instances in a public (Amazon EC2) as well as an OpenStack-based private cloud. Therefore, an extension of an existing Java-based middleware will be designed, planned and implemented using well-established software engineering techniques in order to create a stable and robust prototype and to guarantee timely responses to changing process landscapes.

- Fourth, the prototype will be evaluated in terms of cost analysis for the proposed scheduling approach, taking into account a realistic set of software services to be carried out. The approach will be compared to existing related work.

1.6 Structure of the Work

The remainder of this thesis is structured as follows:

- Chapter 2 gives an overview of the most important concepts that lay a foundation for the conducted work. Developments from the area of cloud computing with a focus on virtualization and containerization techniques are presented. Furthermore, the main concepts of BPM and the ideas behind elastic processes are revisited. Additionally, the fundamentals of optimization models and techniques like linear programming and heuristic problem-solving approaches are discussed.

- Chapter 3 presents relevant publications and related work regarding models for cloud architectures and containerized architectures. We discuss current resource allocation and scheduling approaches for the cloud, also considering work that focuses on container-based approaches. Approaches that solve scheduling and resource allocation problems by defining and solving optimization models are presented.

- In Chapter 4, we present our advanced resource allocation and scheduling approach for elastic processes in containerized architectures. First, we discuss a more fine-grained use case including example processes, mapping to the scenario depicted in Section 1.2, which will serve as a working example throughout the following chapters. Furthermore, we discuss the system requirements for the realization of elastic processes and provide the design of a container-based architecture for process scheduling and resource allocation. We discuss possible solution approaches for the realization of optimized process enactments and provide a detailed system model and an optimization model for elastic processes.

- In Chapter 5 we describe the realization of the designed runtime environment and optimization model for the cost-optimized execution of elastic processes. We also discuss other possible implementations using genetic algorithm-based heuristics.

- Chapter 6 presents a detailed evaluation and discussion of the designed and implemented optimization model and containerized runtime environment while using the fine-grained use case presented in Section 4.3 as a reference. Furthermore it presents an extensive numerical evaluation of the runtime performance of the designed system using example processes.

- Finally, Chapter 7 concludes this thesis with final remarks and gives an outlook of future work.

CHAPTER 2

State of the Art

"If you think you've seen this movie before, you are right. Cloud computing is based on the time-sharing model we leveraged years ago before we could afford our own computers. The idea is to share computing power among many companies and people, thereby reducing the cost of that computing power to those who leverage it. The value of time share and the core value of cloud computing are pretty much the same, only the resources these days are much better and cost effective."

— David Linthicum, author of *Cloud Computing and SOA Convergence in Your Enterprise*

This chapter presents the background about the most important concepts that form a basis for the conducted work and introduces the terminology that is used throughout the thesis. We discuss concepts from cloud computing, with a focus on virtualization and containerization techniques, revisit the ideas behind BPM and elastic processes, and give an introduction into optimization models.

2.1 Cloud Computing

Cloud computing [8] [24] [102] is a paradigm that has strongly evolved during the past decade and is nowadays omnipresent. It focuses on providing virtualized resources for computation, storage, and communication of applications. Infrastructure, platforms, applications, and storage space can be provided as a service. This allows users to utilize as much of an already existing infrastructure and technologies as they need at a certain point in time, without having to face expensive and time-consuming tasks with regard to setup, operation, and adaptation of an up to date computing infrastructure. By sharing a pool of available resources, users only have to pay for the resources they actually use depending on their individual demands.

The range of application areas for cloud computing is extremely wide and ranges from private use cases like picture and video sharing provided by Flickr[1] or YouTube[2] to document sharing, editing and backup services like those provided by Dropbox[3] or Google Docs[4], email services like Gmail[5] and commercial use in advertising [23], banking and financial services [71] [87], health care [6] [98] and even the insurance sector [72] or the manufacturing domain [136] to name just a few.

2.1.1 Definition and Evolution of Cloud Computing

The National Institute of Standards and Technology (NIST) defines cloud computing as *"[...] a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released [...]"* [102].

Buyya et al. have previously proposed a definition for cloud computing as follows: *"A Cloud is a type of parallel and distributed system consisting of a collection of inter-connected and virtualised computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers."* [23]

Furthermore, the essential characteristics of cloud computing as defined by the NIST describe that cloud computing is an *on-demand self-service*, meaning that consumers can easily lease needed computational resources like server time, network storage, memory, and bandwidth, on-demand from service providers without the need of human interaction. A *broad network access* makes the resources available over the network and they can be accessed through standard mechanisms. To serve multiple customers simultaneously, providers use a multitenant model and the technique of *resource pooling* by dynamically assigning and reassigning physical and virtual resources according to the varying demands of different customers. The concept of *rapid elasticity* allows to scale the required resources up and down by leasing and releasing resources depending on the actual demanded. This can happen automatically and the available resources can appear to be virtually unlimited to the user. To monitor, control and report the mostly automatically controlled and optimized resource usage of cloud services, they are offered as *measured services* [102].

Looking as far back as to the 1960s, we can observe that some of the main concepts of cloud computing already existed back then under the term time-sharing. Ziegler defined time-sharing as "[...] a technique that permits concurrent utilization of the same installation by two or more persons working at remote devices capable of direct, online access to the data processing equipment" [175].

Both cloud computing and time-sharing evolved to offer a similar solution, which is mainly a model to share computing power among different users and applications in a utility-like fashion [24] [33], but the motivation and background for the developments are very different. Simply

[1] https://www.flickr.com/
[2] https://www.youtube.com/
[3] https://www.dropbox.com/
[4] https://docs.google.com/
[5] https://www.google.com/gmail/

by regarding the two mentioned definitions, we can see that the definition of cloud computing emphasizes the problem of managing a dynamic infrastructure and system landscape, whereas the definition of time-sharing underlines the necessity of sharing scarce resources among users.

The evolution of similar solutions in a similar setting, while starting from very different preconditions is also a very common concept in biological science, where convergent evolution describes how two organisms that are not closely related, independently evolve similar traits while having to adapt to a similar environment. A prominent example in the literature is the similarity of the wings of birds and bats, the latter being mammals and not closely related to birds. Other examples include the very similar body shape of dolphins and sharks, one being a marine mammal and the other a fish or the similar eyes of the bottlenose dolphin and the octopus, which also have radically different evolutionary backgrounds [101].

Time-sharing models were developed when computing power was extremely expensive and scarce. Looking back to the sixties, in 1962 the IBM 7090 and its successor the IBM 7094 were some of the first main frame computers to be used for time-sharing systems [31] [33]. A typical IBM 7094 machine, which was designed for large-scale computing at that time, could perform up to 500 thousand logical decisions per second and was sold for over three million US Dollars or rented for more than 60 thousand US Dollars per month [70]. Considering the inflation rate, this can be compared to a selling price of over 23 million US Dollars and a rental price of around 500 thousand US Dollars per month today. Thus only very few large corporations could afford computing power at that time and even had to share it, because of the cost of the resources. The IBM 7094 was also used during the Gemini program and the first three unmanned Apollo missions by the National Aeronautics and Space Administration (NASA), but soon had to be replaced by the more powerful IBM 360/75 machines [149].

Still, this computing power is hardly comparable to nowadays standards. According to Google, even one simple Google search today, uses approximately the same computing power, which was in total available to the NASA, in-flight and on-ground, throughout the entire Apollo program, consisting of 17 missions, including the first manned moon landing [96]. Since the start of the computing area, we could observe a tremendous price-performance gain, both with regard to computing power as well as memory and storage performances, meaning that for the same price the performance has improved exponentially over time [99].

Today, computing power is so cheap that it can be found everywhere around us. Not only obvious items like laptops, smartphones, and smartwatches, but even simple household devices like toasters are increasingly being equipped with processors and memory chips. So today's necessity for cloud computing in most cases does not emerge from the pure cost of the computing power, but mostly from the cost of management and operation of an up to date and secure computing infrastructure, that allows to efficiently use latest technologies in a fast-paced time with frequent and often radical changes. Further, the use of cloud services helps to minimize the risk of failure for new ventures, as it reduces upfront investments and therefore can help to foster innovation. The use of cloud computing allows to decouple a company's business processes from the used hardware and software technologies, so the used technologies do not represent a limiting factor in business process design and enactment.

2.1.2 Current Trends and the Future of Cloud Computing

Like it is the case with other omnipresent technologies like electricity and running water, computation is becoming a utility, meaning that it is possible to pay for it only on demand, by using cloud computing technologies [8] [24] [44].

IoT

With the decreased cost of computation, we can also observe that over time there has been a shift from building chips mostly for servers and (personal) computers, to the ubiquitous operation of interconnected chips all around us, found in cars, clothes, walls, furniture or kitchen utensils [79]. This coincides with the IoT paradigm, which focuses on the interconnection of intelligent and self-configuring nodes (things) in a global network infrastructure. Already in 2011, the number of connected devices had exceeded the number of people on earth [55] and it is expected that by 2020 there will even be around 50 billion connected devices [5]. IoT operates to a large extent on Machine to Machine (M2M) communication, meaning that two machines interact with one another, often making use of wireless communication, embedded computing and cloud computing technologies [1] [17]. Ubiquitous and pervasive computing scenarios are based on the IoT, with small, distributed, real-world devices which have only limited computational and storage resources. In contrast to cloud computing, which can virtually offer unlimited resources, the resource-constrained "things" can often not guarantee important performance metrics and characteristics like availability, reliability, security or privacy on their own [17]. Thus the combination of more advanced cloud solutions and the field of IoT might resolve many of the limitations. It is expected that the new paradigm referred to as CloudIoT [17] or Cloud of Things (CoT) [1], which merges the fields of cloud computing and IoT, will widely change and revolutionize the future internet. Also, the newly evolving paradigm of fog computing, which evolved from cloud computing but focuses on the edge of the network, is expected to play an important role in supporting future IoT applications. Fog computing focuses on applications with latency constraints that need more mobility support, and is designed for a large number of smaller nodes with a wide geo-distribution and the need for real-time interactions rather than large batch processing [16] [17]. IoT is expected to have major impacts on everyday life, both for personal applications and in business scenarios. Use cases are wide and range from e-health, assisted living, building automation, smart cities, smart mobility, smart transportation and automotive like self-driving cars to logistics, industrial automation, transportation of goods as well as security [17].

New Areas of Application for Cloud Services

With all these developments, the trend of using cloud services has also arrived in more traditional sectors. As an example, in the healthcare industry medical diagnostics of patients can be accessed around the world and allow for international treatments. Patients can also be monitored remotely to improve their medical care [32]. Especially through the use of new wearable healthcare sensors but also numerous other healthcare services and applications like e.g., ambient assisted living, IoT and cloud computing in combination with advanced cloud processing and an-

alytics techniques play a crucial role in the development of pervasive healthcare and new healthcare provisioning systems that might soon revolutionize the healthcare sector [41] [59] [73]. Similar trends can also be observed in the manufacturing domain as well as increasingly also in the educational sector. Although many technologies for those sectors already exist, they often have not yet been adopted, because of different reasons ranging from currently existing and contradicting regulations, privacy and security reasons, to a lack of knowledge about the application and use of the new technologies among professionals in different industries [32] [73].

New Evolving Computing Paradigms based on the Cloud

Another recent computing paradigm highly related to cloud computing which is expected to further shape many future applications in a highly connected world is the blockchain [34] [146] [161]. A blockchain is a public ledger in the form of a distributed database, consisting of records of all executed transactions among participating parties which are verified by consensus of a majority of the participants. A popular application of the blockchain technology is Bitcoin, a decentralized peer-to-peer digital currency. Besides a wide range of applications in the financial world [151], like, for example, smart contracts, there are numerous other areas of applications for the blockchain technology, including jurisdiction, political and government applications as well as social and scientific domains, e.g., with solutions for decentralized storage, decentralized proof of documents, decentralized IoT applications and internet applications like decentralized DNS solutions and even applications in the music industry [34] [146]. The main focus is on developing democratic open and scalable trust systems instead of centralized authorities and thereby cutting out the middleman [161].

These attributes of blockchain can provide solutions to some obstacles of IoT applications, especially with regard to security issues. In general, the blockchain might offer solutions for many trusted computing problems in society, by giving users control over their personal and sensitive data without having to trust third parties and without compromising security, while also giving companies the possibility to make use of data without the need of implementing additional security measures. Legal and regulatory decisions about the collection, storage, and sharing of data could be made easier and in some cases also automatically enforced by using decentralized blockchain technologies [176]. Especially in the field of healthcare, also with regard to the evolving IoT healthcare solutions, sensitive and personal patient data needs to be shared frequently. Therefore proposed blockchain technologies that give control and ownership back to the patients could provide a new way to improve the intelligence of healthcare systems while taking privacy issues into account [166]. More recently, blockchain technologies have been proposed for the execution of business processes [49] [104] [130].

We can observe that many of the current and future trends including IoT [55], advancements in Artificial Intelligence (AI) [133] [114] including Machine Learning (ML) [107] and blockchain solutions [146] either evolved from cloud computing or are strongly dependent on cloud solutions. To a great extent, we can expect future technologies to be highly interconnected and to operate on cloud solutions. An advanced integrated application could be a smartwatch that interacts with smart city traffic sensors and therefore automatically pays for traffic and parking bills using blockchain-based smart contracts [146].

Quantum Computing as a Cloud Service

As already mentioned in the previous section, cloud computing allows to rapidly make the latest technological developments available on demand to anyone over the internet. This means that not only large corporations willing to set up and maintain large infrastructures and hire expensive IT staff are now able to make use of cutting edge technologies. Recent developments that reinforce this advantage of cloud computing can be found in the field of quantum computing. Although the developments are still in an exploratory beginning phase, IBM has, for example, already made quantum computing available on its IBM Bluemix [6] cloud platform. This allows everyone to access the new technology and therefore also help in advancing it, as it is still in an early stage. The ultimate goal is to create a universal quantum computer, capable of performing important computing tasks for science and business exponentially faster than classical computers [129]. This might represent the beginning of a new era in computing as well as change and extend current cloud computing concepts. Especially such developments as quantum computing make cloud solutions essential, not only with regard to cost savings because of cheaper, easily accessible and more efficient resources, but more particularly because it offers access to bleeding edge technologies that are not readily available on the market yet and have not even been used by large corporations before. When looking at quantum computing, for example, it requires trained physicists to run quantum computers in very particular surroundings [134]. This again brings us back to a very comparable precondition as we have already faced it back in the sixties when time-sharing models were developed and resources had to be shared as they were scarce, highly expensive and hard to maintain.

It should be noted that although computing power has grown exponentially, there is a limit to the possible growth for computers as they are built today. Until now, Moore's law, which, based on observations, predicted the growth of the number of transistors per computer chip, has been very accurate. In 1965, Moore predicted that the number of transistors would double every year for the next decade [108] and in 1975 he adjusted the prediction to a doubling for every two years [109]. Although also in the recent past the number of transistors per chip and with it the available processing power has been growing at an exponential rate as described by Gordan Moore [94], Moore's law still has a limiting factor, as it is physically impossible to shrink transistors even further once they have reached the size of atoms. It has been proposed that when shrinking transistors to the size of electrons, the quantum limit year, according to Moore's law and in consideration of Heisenberg uncertainty would already be reached in 2036 [123]. Additionally when following the trend of interpreting Moore's law with regard to the growth of computing power and processing speed instead of only the number of transistors on a chip, there is a considerable bottleneck between very fast CPUs and other components such as RAM and Hard Disc Drives (HDDs) which can lead to lower performances when Input/Output (I/O) speed is low for tasks with a high I/O bound. Advancements for minimizing the waiting time are therefore important and have been addressed by manufacturers by offering fast flash drives next to HDDs. In most cases, these new technologies are more expensive. Therefore it is often advantageous to share and access them via cloud-based data centers [11] [134]. The use of cloud services also facilitates keeping up to date in a fast paced environment and easily making use of

[6] https://www.ibm.com/cloud-computing/bluemix/

latest advancements where necessary. Whether or not we will hit a limit of Moore's law with regard to an increase in computing power, also regarding the potential of quantum computing and future developments, is under debate [123], but in any case further rapid changes in technology are to be expected also in the future, so that cloud models can offer high flexibility and fast adoption times.

The developments in quantum computing are from an economic perspective a very interesting combination of technology-push and market-pull, also known as demand-pull, strategies [40]. On the one hand, there are companies like IBM pushing the new technology into the market, but they are doing so at a very early stage, so future users and early adopters can have a determining influence on the development. On the other hand, by offering the early technologies in the cloud, users can easily experiment with the possibilities of the technology, state their demand and therefore dictate the further development of the technology, which corresponds to the market-pull paradigm.

The future applications for quantum computing can currently at most only be estimated, but because of the quantum superposition principle, quantum computers will potentially be able to solve certain problems exponentially faster than today's computers. Quantum computers might especially be strong in solving linear systems of equations, which are frequently needed in nearly all fields of science and engineering. Quantum computing is expected to provide very large advantages for massive computations as they are needed for optimization problems, for AI applications including ML solutions or also for simulations like those used in pharmaceutical research or agent-based models. Also, the development of new cryptographic applications will be a major research area, as quantum computing is expected to be offered in the cloud and therefore secure cloud quantum computing encryption methods will be necessary. Current quantum computing experiments already use the early stage quantum computing cloud services currently offered by IBM [26] [37] [68] [118]. Soon more advancements in the area of Quantum Computing as a Service (QCaaS) are to be expected [126].

The successful introduction of powerful quantum computing systems can lead to the need for adaptation and changes in current systems, especially with regard to security issues. When looking at the already discussed blockchain technology, which is based on the concept of a computationally tamper-proof ledger [176] that can not be corrupted by a single party, advancements in quantum computing that provide considerably higher computational resources for complex mathematical operations could bring such blockchain systems to their knees [34] and very new quantum encryption methods would need to be developed.

Future Implications

The discussed trends show that different important developments from varying fields are based on cloud computing services and principles and it can be expected that especially the interconnection of these different developments will play a major role for future cloud applications. As cloud solutions will form the basis for the success of many innovations, an optimal management of cloud resources will be a crucial factor for the effective realization and adoption of future applications and scenarios.

2.1.3 Cloud Service Models

Cloud computing providers offer very diverse services in a pay-per-use manner on different layers of abstraction and for different types of use cases. The architecture of cloud computing can be split into four layers building upon each other [170]:

- The *data center layer* consists of the actual hardware and offers physical resources like CPU, memory, disk and network links.

- the *infrastructure layer* (also known as the virtualization layer) builds the heart of cloud computing and offers the underlying resources from the data center layer in the form of a virtualized pool of resources. Virtualized computational resources are offered using VMs and storage space is provided in blocks. Therefore a dynamic resource assignment is possible, without being bound to the static properties of the physical resources. Amazon's EC2[7] is a prominent cloud hosting web service for renting VMs, while their Simple Storage Service (S3)[8] is a popular web service for storage space.

- The *platform layer* can provide OSs and application frameworks, e.g., for Java or .Net and thus facilitates the deployment of applications in VMs. An example is the Google App Engine[9] that provides API support for implementing web applications.

- The *application layer* offers actual cloud applications that can easily be accessed and scale automatically. Prominent examples include Google Apps[10], Youtube, or Facebook[11].

Based on the four-layered cloud computing architecture, three main categories of cloud services, depicted in Figure 2.1, are commonly distinguished. The three cloud service models all offer very different capabilities for different users and are [102]:

- **Infrastructure as a Service (IaaS)**: An IaaS provider offers a virtualized infrastructure to the customer. The provider manages networking, storage, and servers and offers the resources via virtualization techniques like VMs to the customers who can setup, deploy and run arbitrary OSs, middlewares, runtime environments and applications on the leased resources, without having to manage the underlying infrastructure. IaaS offers the most flexible cloud service model and gives developers more control over the infrastructure. Many IaaS users develop and deploy platforms on top of the provided IaaS and make these platforms available as Platform as a Service (PaaS) offerings. Users of IaaS of course also have the possibility to develop and deploy software on top of their own platform layer and offer Software as a Service (SaaS) solutions. Next to providing IaaS, like Amazon EC2 does, providers can also offer Data as a Service (DaaS), which is a service for storing and retrieving data in the cloud. A prominent example for DaaS is Amazon S3.

[7] https://aws.amazon.com/ec2/
[8] https://aws.amazon.com/s3/
[9] https://cloud.google.com/appengine/
[10] https://gsuite.google.com/
[11] https://www.facebook.com/

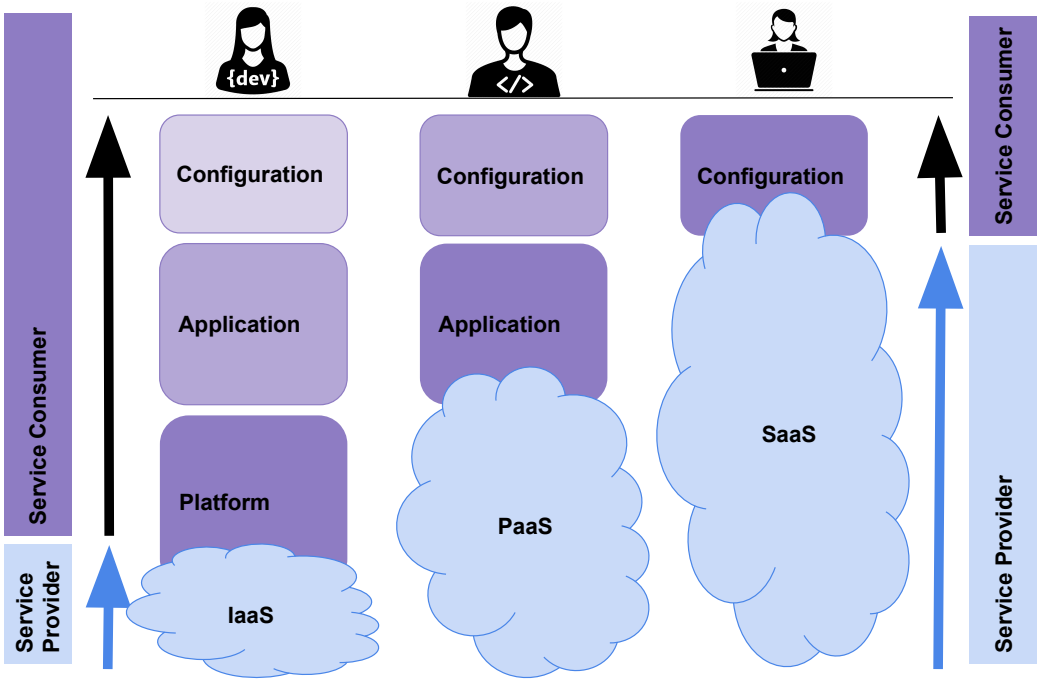


Figure 2.1: Cloud Service Models

- **PaaS**: provides IaaS and a platform layer with OS support and software development frameworks like, for example, certain middlewares and runtime environments. Typical users of PaaS offerings could be web developers, who lease a platform offering certain programming language execution environments, a database, and a web server. PaaS users can develop software on the given platform and make the software available in form of SaaS offerings.

- **SaaS**: targets end users of elastic and scalable cloud software. The user can use the software on demand and define configurations, but does not need to worry about any underlying technicalities. The SaaS provider takes care of the deployment and functionality of the application and its data.

When obtaining a packaged software, instead of using a cloud service, a user has to manage the application and its data, the runtime environment, middleware, and OS, as well as the virtualization layers including the optimization of the utilization of the underlying physical resources like servers, storage, and the network.

SLAs

Cloud service providers promise certain metrics and levels of performance for their services to customers by establishing contracts, which are referred to as SLAs. The non-functional QoS

guarantees specified in SLAs target important parameters like availability, throughput, downtime, bandwidth or response time. Service providers have to monitor the defined QoS guarantees and make sure that they deliver the services such that each defined non-functional requirement corresponds to the corresponding agreed Service Level Objective (SLO). Furthermore, SLAs state penalties for the violation of the defined SLOs [24] [81].

2.1.4 Cloud Deployment Models

There are four different cloud deployment models as defined by the NIST [102], differing in ownership, size, and accessibility. Depending on business requirements, security and privacy issues [128] [143], users can choose between:

- **Private Clouds**: A private cloud offers its infrastructure exclusively to one organization, often set up in an organization's own data center. Private clouds can also be offered by third parties and be located off premise. Multiple business units of an organization can access the private cloud and make use of the cloud's scalable resources. Well maintained private clouds can offer high levels of security as the cloud is only exposed to internal customers. Furthermore, a private cloud facilitates meeting regulatory and security compliances, as the cloud can be managed and controlled by one company.

- **Community Clouds**: A community cloud is similar to a private cloud with the difference that the cloud infrastructure is not exclusively utilized by only one organization, but a community often comprised of multiple organizations with shared concerns, e.g., with the same policies and compliances.

- **Public Clouds**: In a public cloud the cloud infrastructure is shared by the general public and operated by a cloud provider on their premises. A big advantage is that costs can be shared by the public cloud users on a very fine-grained utility computing basis, using pay-per-use models. Spikes in a user's demand for computational resources can be met without the risk of under-provisioning of the available cloud infrastructure. Public clouds are often thought to be less secure than private clouds, as it has to be ensured that the publically shared infrastructure is not maliciously attacked. Furthermore often differing policies and regulations of the different participants can not always be taken into account.

- **Hybrid Clouds**: A hybrid cloud is a composition of multiple distinct cloud infrastructures that are bound together but still remain unique entities. A hybrid cloud enables portability of data and applications between the different underlying infrastructures. A typical example of a hybrid cloud is a company maintaining a private cloud for critical applications and also using a public cloud for spikes in demand of computational resources, but also the combination of two or more different public cloud infrastructures can be a hybrid cloud. A hybrid cloud offers a possibility to combine the advantages of different cloud infrastructures and even out their limitations.

2.1.5 Virtualization and Containerization Techniques

When regarding the definition of cloud computing as discussed in Section 2.1.1 we can derive that for realizing cloud computing we need a technology that allows to split up and dynamically share computational resources while providing isolation and control of resources in a multitenant environment. There are two commonly used technologies for achieving these requirements, on the one hand, we have the well-known *hypervisors* that create and manage VMs, but on the other hand, also *containers* can provide isolation and multitenancy.

In the following sections we discuss the main features of virtualization and containerization techniques, compare both concepts with each other and argue how they can be used separately as well as complementary.

VMs and Hypervisor-based Virtualization

Virtualization builds the basis for the realization of cloud computing techniques and refers to the ability to abstract physical computing resources such that the resources of single physical machines, e.g., the available CPU, memory, disk, and bandwidth, can be split and function as individual logical units. Therefore, multiple VMs can host different isolated operating systems on virtual hardware on top of the same physical machine.

In the cloud computing architecture, virtualization is implemented in the infrastructure layer as introduced in Section 2.1.3, by partitioning storage and computational resources and therefore creating a pool of virtualized resources. Such pools are created by using virtualization technologies referred to as *hypervisors* or *VM monitors*. The OS that runs on virtualized resources is called the *guest OS* and is managed by a hypervisor, which takes care of the virtualization and resource isolation, manages VMs on the physical hardware and coordinates the flow of instructions between guest OSs and the hardware. By separating the guest OS from the actual hardware, virtualization offers a possibility to realize the key features of cloud computing, such as the dynamic assignment of computational resources while also masking the technological complexity from the users [24] [135] [170].

There are two main different types of hypervisors that allow to emulate hardware and bring up guest OSs on top of the virtual resources and therefore decouple the guest OS from the underlying hardware. The two types are depicted in Figure 2.2 and are [105] [122] [135]:

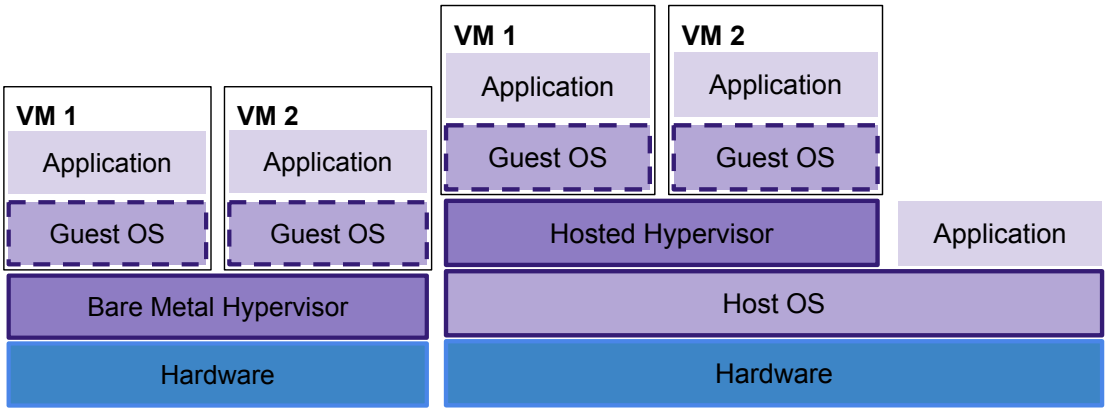
- *Type-1*, also known as *bare metal* or *native* hypervisors run directly on the underlying host's physical hardware, without the need of a host OS. The hypervisor is in most cases a relatively slim software installed on each host computer and controlled by a management software, which can control multiple hypervisors on multiple physical hosts. The management software can install and manage VM instances' on a hypervisor, consolidate physical servers and migrate VM instances from one server to another physical server, based on the instances needs while considering an optimal server utilization that allows to turn servers on and off as needed and therefore to minimize power consumption and electricity bills. Furthermore, the management software can also take fault tolerance issues into account and hide server crashes from the users. Most new generation type-1 hypervisors also allow overallocation, meaning that more total resources than actually available

on a host can be allocated to VM instances, while at any time all instances of a host will not use more than what is actually available on the physical hardware. As the instances often use fewer resources than they actually claim, this form of dynamic allocation allows for a better utilization of the physical resources while still being able to provide the allocated resources to VM instances when they need them. Examples for type-1 hypervisors include VMware ESXi[12] which is managed by vShpere[13], the Xen hypervisor[14] managed by the Citrix XenServer[15], Microsoft Hyper-V[16] or Oracle VM Server for x86[17] or for SPARC[18]. It should be noted that many hypervisors are based on the open source Xen hypervisor and companies, therefore, offer basic functionalities for their hypervisors for free but charge for additional features like the discussed overallocation, fault tolerance, and server consolidation.

- *Type-2* or *hosted* hypervisors can run on top of almost any host OS, e.g., Linux, MacOS or Windows, while in contrast to type-1 hypervisors also other applications can be installed and run alongside the hypervisor on the host OS. The hypervisor manages the guest OSs, which run as processes on the host. Usually, a virtualization application runs as a software layer in the guest OS and allows to control the virtualization parameters. Therefore no additional management software is needed, as it is the case with type-1 hypervisors. Special attention should be paid during resource allocation for type-2 hypervisors. In contrast to type-1 hypervisors, the resources allocated to guest OSs on hosted hypervisors are taken from the underlying host and allocated to the guest instance, no matter if the instance actually uses the resources or not. Therefore the host system might crash if there are not enough resources left for the basic functionality of the host OS itself and the other applications running next to the virtualization environment directly on the host OS. Examples for type-2 hypervisors include the Oracle VM VirtualBox[19], Microsoft's Windows Virtual PC[20], VMware Fusion[21] for Mac, VMware Workstation for Windows[22] and for Linux[23] or Parallels[24] Desktop for Mac.

Other important hypervisors such as the Linux Kernel-based Virtual Machine (KVM)[25] are not easily categorizable as type-1 or type-2 hypervisors. There have been many debates [121] on how to classify KVM, which is a Linux kernel module. The debates often focus on issues like performance, latency, security, scalability, and isolation with regard to the classification as

[12] http://www.vmware.com/products/vsphere-hypervisor.html
[13] http://www.vmware.com/products/vsphere.html
[14] https://xenproject.org/
[15] https://xenserver.org/
[16] https://www.microsoft.com/en-au/cloud-platform/server-virtualization
[17] https://www.oracle.com/virtualization/vm-server-for-x86/index.html
[18] https://www.oracle.com/virtualization/vm-server-for-sparc/index.html
[19] https://www.virtualbox.org/
[20] https://www.microsoft.com/en-us/download/details.aspx?id=3702
[21] http://www.vmware.com/products/fusion.html
[22] http://www.vmware.com/products/workstation.html
[23] http://www.vmware.com/products/workstation-for-linux.html
[24] https://www.parallels.com
[25] https://www.linux-kvm.org/



(a) Type-1 Hypervisor Architecture (b) Type-2 Hypervisor Architecture

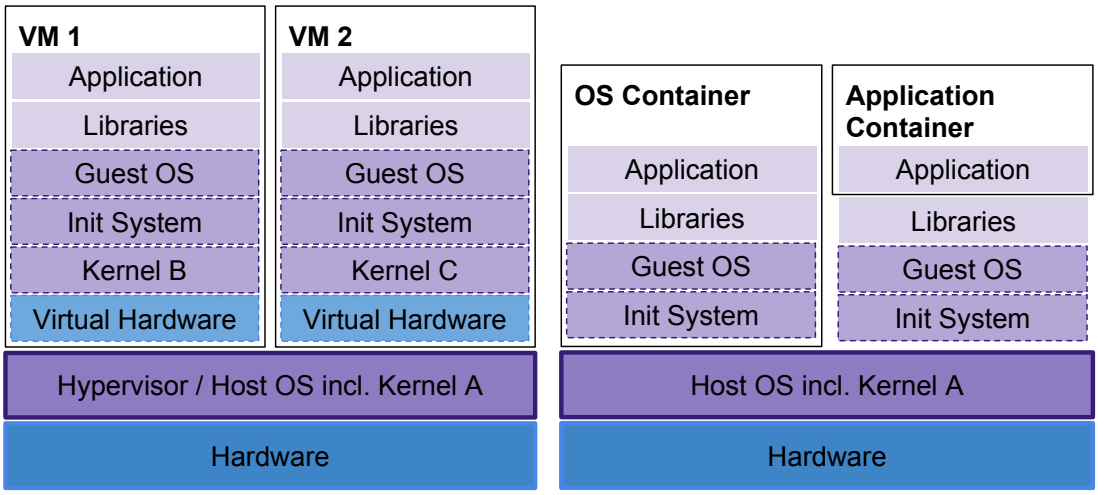
Figure 2.2: Virtualization Architectures (adapted from [135])

a pure bare metal or hosted hypervisor. As KVM builds upon a general purpose OS, it could be seen as a type-2 hypervisor. At the same time, if no other applications are installed directly on the Linux machine, KVM could also be classified as a type-1 hypervisor, as it converts the host OS into a VM monitor and only the Linux kernel is used to schedule CPU, memory and I/O, therefore only one translation between the physical hardware and the virtualized resources is needed for resource isolation, which is a characteristic of type-1 hypervisors [46] [52].

For the realization of their own private clouds, some enterprises often use VMware installations with an ESXi Hypervisor, but also some public cloud providers make use of ESXi [12]. When looking at well-known cloud providers, we can see that the Xen hypervisor is used by Amazon Web Services (AWS)[26] and Rackspace[27], while KVM, which was supported by Linux more recently as an open source alternative, is used by more lately constructed clouds like those from AT&T[28], HP[29] or Comcast[30]. OpenStack[31], and most OpenStack distributions like those from RedHat[32] or Cloudscaling[33] also make use of KVM. Microsoft, of course, makes use of its own Hyper-V for its cloud offerings like Microsoft Azure[34].

But the use of hypervisors is not the only possible way to build cloud computing offerings. Successful cloud platform providers that heavily rely on containerization technologies as introduced in the following section include Google and IBM [12].

[26] https://aws.amazon.com/
[27] https://www.rackspace.com/
[28] https://www.business.att.com/enterprise/Portfolio/cloud/
[29] https://www.hpe.com/us/en/solutions/cloud.html
[30] https://business.comcast.com/cloud
[31] https://www.openstack.org
[32] http://www.redhat.com/en/insights/openstack
[33] http://cloudscaling.com/
[34] https://azure.microsoft.com/



(a) Hypervisor Architecture

(b) Container Architecture

Figure 2.3: Containerized Architecture vs. Hypervisor-Based Architecture (adapted from [18])

Containerization Technologies and a Comparison to VMs

While hypervisor virtualization technologies are based on emulating hardware and bringing up another completely new OS on top of virtual hardware (hardware-level virtualization), containers are about virtualizing the host OS itself and providing the host OS' services, e.g., networking and filesystem, in a virtualized manner (OS-level virtualization). Although the underlying basics of container technologies are not new and the ideas behind container technologies date back to time-sharing systems, container technologies have only recently gained significant popularity, especially due to their advantages for service providers in the PaaS space and thanks to successful container technologies like Docker[35] [12] [18].

Figure 2.3 compares the architecture of containerized architectures to hypervisor-based architectures and illustrates that hypervisors allow to bring up any OS on top of any other OS. As also the hypervisor will at least have its own kernel or use the underlying host OS' kernel, there are always multiple kernels involved when it comes to hypervisor-based virtualization. In contrast, a container management service like Docker, OpenVZ[36] or LXC[37] is installed on the host OS and enables the host's kernel to manage the isolation between multiple applications and processes and to provide the OS' services in a virtualized way. This means that multiple isolated containers can run on a node sharing a single kernel instance. As there is exactly one kernel involved when running containers, it is only possible to bring up multiple copies of OSs that are based on the same kernel on a single node. This means that with containerization technologies it is for example not possible to bring up a Linux and a Windows OS container on the same node, as the Windows OS and the Linux OS do not share the same kernel, but it is possible to bring

[35] https://www.docker.com
[36] https://openvz.org
[37] https://linuxcontainers.org

Table 2.1: Comparison: VMs vs. Containers

	VMs	**Containers**
Virtualization	hardware-level virtualization, VMs run on emulated hardware	OS-level virtualization, containers run on virtualized resources of the host's OS
Kernel	multiple kernels, one for the host OS and/or hypervisor and one additional kernel for each guest OS in every VM is loaded into its own memory region	only one kernel is shared by all guest OSs and the kernel image is loaded into the physical memory
Boot process	started through standard boot process leading to a number of hypervisor processes on the host	started as application processes on the host directly or through a container-aware daemon like systemd
Startup Time	boot up in a few minutes	boot up in a few seconds
Isolation	sharing libraries or files between guests and between guests and host not possible	libraries and subdirectories can be mounted and shared
Storage	heavyweight, the whole OS incl. kernel and programs have to be installed and run	lightweight, as the base OS is shared and also libraries can be shared for application containers
Performance	overhead for translating machine instructions from guest OS to host OS and poor performance under resource pressure due to competing kernels	near-native performance compared to host OS and better resource management under high workloads since resources are managed by only one kernel

up multiple instances of the same OS or of different OSs that share the same kernel, like e.g., CentOS[38] and RHEL[39], on a single node [12] [18] [42] [90].

As containerization technologies are based on sharing only one single kernel, containers are less attractive when there is the need for heterogeneous environments, as it is sometimes the case in the enterprise space, but containers imply a number of advantages for service providers, who can easily set up larger homogenous groups of machines running the same OS kernel [18].

Comparison of VMs and Containers: The main differences between hypervisor-based systems using VMs and container-based systems are discussed in Table 2.1 [12] [18] [42] [90] [116].

[38] https://www.centos.org/

[39] https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux

Advantages of Containerized Architectures: One of the biggest advantages of containerized architectures is that they allow for a higher degree of elasticity on multiple levels. One reason for this is that the container stack is very light. While application container images mostly only reach the size of some Megabytes (MBs), VM images that pack the full OS and kernel often reach the order of Gigabytes (GBs). Therefore, containers are faster to start, stop or migrate and it is even easily possible to add and remove resources like CPU and memory without the need to restart or reboot the whole OS which facilitates horizontal scaling and even allows for instant vertical scaling [12] [18].

Furthermore, containers can provide a higher density on the same node and are able to deliver the required performance and operate more stable than hypervisor-based systems under high resource pressure. The reason for this is that there is only one kernel that is put under resource pressure and a single kernel can better handle limiting constraints and control the resources as it has full information about what is happening inside of the containers. Therefore, resource sharing and efficiency can be enhanced, especially as the page cache of the single kernel can easily be shared. In contrast, when using a hypervisor-based system it can often happen that the host OS claims pages of memory from the guest OSs and the guests try to reclaim it again and therefore the whole system can slow down. This is a problem as hypervisor-based systems were not created with limited resources in mind but to better utilize free resources on running machines [18] [46].

Another advantage that comes from only having one kernel is that patching problems often faced when updating VM kernels are easily solved and all container-based guests of one kernel directly benefit from patches applied to that kernel [18].

OS Containers and Application Containers: An *OS container* is a virtual environment that allows isolating processes while still sharing the kernel of the host OS. As the containment does not only apply to one process but also all child processes, an entire OS can be booted in an OS container, starting with the init process and providing the same level of isolation and security as a VM. OS containers can be created using container manipulation technologies like LXC or OpenVZ which are based on concepts like *CGroups* for controlling resources in the kernel (e.g., CPU, memory, devices, etc.) and *namespaces* for providing isolation inside the kernel (e.g., network namespaces, PID namespaces, etc.). An *application container* is very similar to an OS container, with the difference that application containers are designed to package and run only one single service instead of multiple processes inside of one container. This also means that application containers are not meant to pack, ship and deploy full guest OSs. Instead, only the application and all the required libraries are shipped in one container [18] [42]. To give an example for this, when deploying a simple Wordpress application, an OS container could include multiple services like an SSH daemon, the Wordpress theme with libraries, an Apache server, a mail server, etc. When using application containers there would be a separate container for each service, e.g., one container only for the Wordpress theme and the required libraries, another container for the Apache server and so on. These individual containers would then be linked to communicate with each other. One of the most prominent technologies for creating application containers is Docker.

Application Containers for DevOps and Microservices: One of the main advantages of application containers is that they facilitate the realization of DevOps [9] practices and goals. DevOps focuses on practices to improve the cooperation of "software development" and "IT operations" such that the time between committing a change to a system and the change being visible in the production environment is minimized. According to DevOps. software delivery processes should be automated as much as possible. As continuous deployment practices lead to a more frequent build, test and release cycle of software, a rapid and reliable deployment of software and changes has to be guaranteed.

One of the key advantages that application containers provide to developers is that they can easily pack their applications into standardized container formats and afterwards simply deploy and run the containers in any environment that supports the container format. For Docker, this would mean that an application in a Docker container will easily be deployed and behave the same whether it is shipped, e.g., to EC2 or a private laptop, as all dependencies and libraries are packaged into the container image.

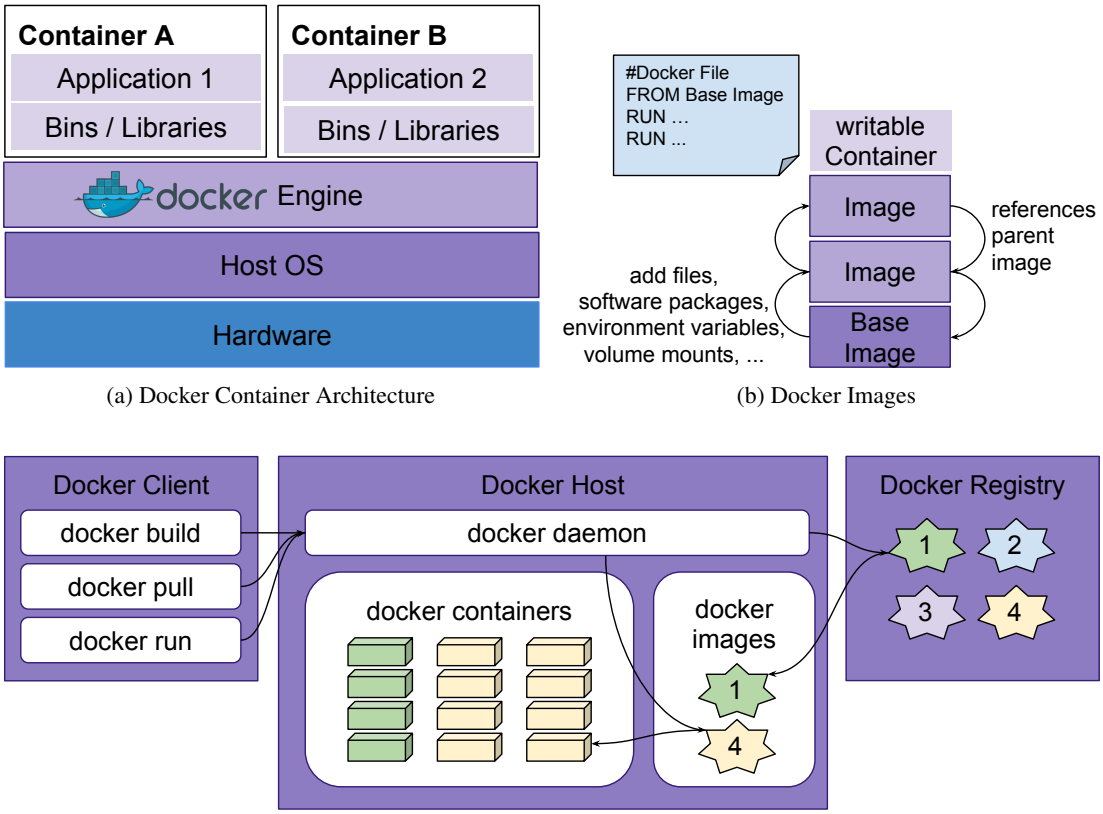
People working in operations benefit from the use of such containers, as they no longer have to manage all dependencies and write scripts to match the application's changing needs. Instead, they can simply offer the container engine, e.g., the Docker Engine and a basic infrastructure, so the developers can take care of anything else. This facilitates and speeds up the software shipping process and continuous deployment practices.

Application containers are lightweight and they are well-suited as a base technology for implementing microservice architectures. With microservices, larger application architectures are broken down into independently deployable services that are loosely coupled and can be mapped to different business processes. Therefore, single large applications can be represented as a suite of smaller, independently manageable services, communicating with each other [116].

Docker: The Docker project is described as *"an open platform for developers and sysadmins to build, ship, and run distributed applications, whether on laptops, data center VMs, or the cloud"*. The Docker Engine is the most prominent example for a container engine supporting the creation of application containers. Figure 2.4a illustrates the containerization architecture when using Docker containers, while Figure 2.4b shows the general structure of Docker Images and describes their creation process. Figure 2.4c gives an overview of the interplay of different Docker components in a Docker-based system architecture. The most important components of Docker as depicted in Figure 2.4 include:

- **Docker Objects**

  Docker Image: A Docker Image is an immutable read-only template for creating Docker containers. It allows to store the application code and configurations and can be compared to an executable. The Docker Image can include a snapshot of the application code and the entire environment in which the application was created, so the application will behave the same way also when deployed for example in production containers on other machines. A Docker Image typically consists of several layers. Figure 2.4b illustrates the creation of a new Docker image. A Docker Image is created by executing a *Dockerfile* that defines the steps to create and run an image. An image can be based on



(a) Docker Container Architecture

(b) Docker Images

(c) Overall Docker Architecture

Figure 2.4: Docker Platform

another image, therefore a *base image* can be defined in the Dockerfile and customizations to the base image can be added. An example would be starting with a CoreOS base image, installing the Apache web server and MongoDB, creating default data directories, exposing default ports and adding the new application including all necessary configurations on top of the base image. Each instruction in a Dockerfile (e.g., for adding files, software packages, environment variables, volume mounts, etc.) creates a new layer in the image, while each layer references its parent layer. This image architecture makes images relatively lightweight, as changing a Dockerfile and rebuilding an image will only rebuild the layers that have actually changed (and all child layers). Docker Images can be pushed to a public registry for general availability, or to a private registry with restricted access.

Docker Container: A Docker container can be described as a runnable instance of a Docker Image that can be created, started, stopped, migrated or deleted. While the image is immutable, a container is mutable and ephemeral, so when a container is removed the writable layer depicted in Figure 2.4b is generally also lost, unless the container's state is stored in a persistent storage. It is also possible to create new images based on the current state of a running container. Containers are isolated from other containers and the host

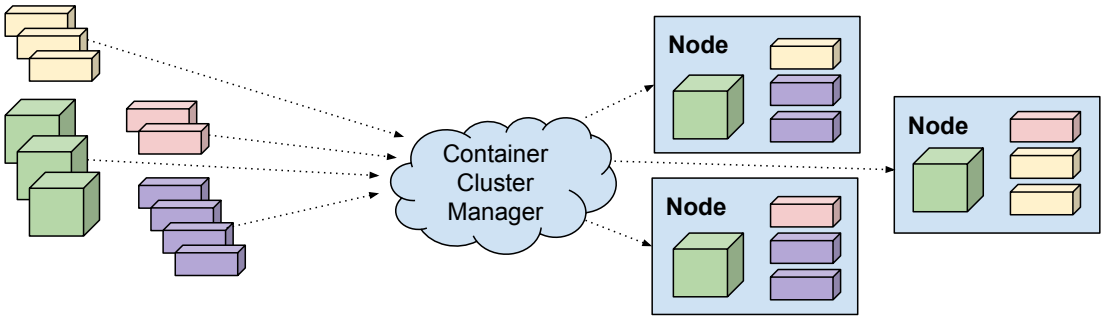


Figure 2.5: Container Cluster Management

machine, but the level of isolation is controllable.

- **Docker Registry:** Docker Images can be stored in a registry. There are public registries, e.g., on Docker Hub[40] or Docker Cloud[41] but it is also possible to use a private registry. When using the *docker pull* command, the Docker daemon pulls the required images from a registry and stores them on the Docker host to run containers based on that image. Multiple containers can be started from the same image, so a Docker Image has to be pulled only once for all corresponding containers on a Docker host. As the same image can be used for multiple container instances, the state of those containers will only differ in their writable layer, and regardless of the more lightweight containerized architecture, less storage space compared to virtualization with VMs is needed, as VMs typically require a full dedicated copy of the image in their filesystem.

- **Docker daemon:** The Docker daemon listens for Docker API requests from a Docker client and manages Docker objects like Docker Images, containers, networks and volumes on a Docker host.

- **Docker client:** A Docker client sends commands to one or multiple Docker daemons.

To build and manage a cluster of containers communicating with each other, and in order to scale containers across multiple nodes, technologies like Google's Kubernetes[42] or Docker Swarm[43] that also offer replica management and load balancing can be used. The main functionality of such container cluster managers is depicted in Figure 2.5 and consists of scheduling, packaging and connecting a wide number of user containers on nodes in a dynamic cluster. In other words, such container cluster management tools can bridge the gap between rather low-level virtualization technologies like those provided by the Docker Engine or LXC and high-level application architectures consisting of multiple interconnected and elastic services. Therefore, container cluster management tools help in building distributed scalable applications out of invokable, addressable and scalable services.

[40] https://hub.docker.com/
[41] https://cloud.docker.com/
[42] https://kubernetes.io/
[43] https://docs.docker.com/swarm/

(a) Vertical Scaling (b) Horizontal Scaling

Figure 2.6: Vertical vs. Horizontal Scaling

2.1.6 Scalability in Cloud Computing

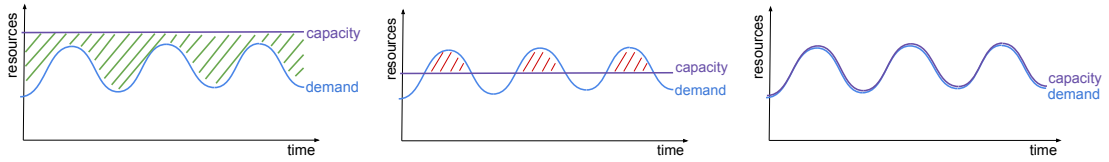
One of the main advantages of the cloud computing paradigm is the concept of rapid elasticity, which allows to allocate and deallocate or reconfigure cloud resources dynamically and on demand, following a set of predefined rules. Cloud application scalability can be achieved by either implementing vertical or horizontal scaling techniques [106] [155]. Figure 2.6 illustrates those two mechanisms.

- **Vertical scaling:** Vertical scaling allows to scale individual system components up and down by changing their assigned computational resources, i.e., by adding and removing virtualized hardware resources to a node. After applying vertical scaling techniques, the scaled component remains one logical unit. Changing the allocated resources of running instances, e.g., adding more physical CPU and RAM on-the-fly has been traditionally difficult when using hypervisor-based virtualization techniques, often requiring a full reboot of the OS [92] [155]. On-the-fly vertical scaling is facilitated by using lightweight container engines as described in the previous section, as allocated resources can easily be adapted during runtime.

- **Horizontal scaling:** Horizontal scaling refers to a system's ability to scale the available computational resources in and out by changing the number of available system components, i.e., by adding/removing server replicas and load balancers to distribute the load. Horizontal scaling can easily be achieved by using hypervisor-based virtualization techniques as well as more lightweight containerization techniques that can decrease time and resource overhead associated with leasing new VM instances. As horizontal scaling techniques change the number of logical or physical nodes of a system, the number of nodes that have to be provisioned by load balancers changes.

2.1.7 Elasticity in Cloud Computing

The scalability features provided by cloud computing environments have led to a wide range of elastic cloud services like, e.g., offerings for elastic computing by AWS's EC2. By making use



(a) Over-Provisioning (b) Under-Provisioning (c) Dynamic Provisioning

Figure 2.7: Provisioning under fixed resources and dynamic provisioning (adapted from [8])

of hosted instances instead of setting up own data centers, customers can not only take advantage of the provided infrastructure and expertise of cloud hosting providers but also benefit from the elasticity properties of cloud offerings. Virtualized resources in a cloud environment can be leased from multiple sources for only short time periods and added within minutes, while containers often even only need a few seconds to be up and running. Therefore the utilized resources can be matched to the actual workload as well as cost and QoS constraints. In contrast, when using data centers with static resource capacities users run a very high risk of over-provisioning or under-provisioning their services and applications which comes with high additional costs and business-related risks [8].

- **Over-provisioning** refers to a state where the statically leased or set up resource capacity is higher than the actual demand for computational resources. This situation is depicted in Figure 2.7a and can lead to a high amount of unused and therefore wasted computational resources. Over-provisioning is often implemented for applications to make sure that there are enough resources available also during request spikes and known or unknown resource usage peak times. The associated costs for over-provisioning are easy to calculate and equal to the additional cost that arises from leasing or maintaining the green shaded area in Figure 2.7a.

- **Under-provisioning** refers to a state where the statically leased or set up resource capacity is lower than the actual demand for computational resources. This situation is depicted in Figure 2.7b by the red shaded area. Under-provisioning leads to situations where the requested load can not be processed and user requests have to be rejected. The costs of under-provisioning are harder to calculate as rejected user requests might be serviced by other providers and therefore dissatisfied customers might be lost in the long run. This can lead to a less frequent occurrence of request spikes and a decreasing number of requests in general. Therefore, companies might face the situation of having an unplanned over-provisioning infrastructure for the decreasing number of requests.

- **Dynamic provisioning** is depicted in Figure 2.7c and eliminates the risks of over-provisioning and under-provisioning. Dynamic provisioning can be achieved by using elastic cloud services on demand so leased resource capacities can always correspond to the actual load. No computational resources are wasted and unexpected spikes can also be provisioned for the time of their occurrence. Therefore, cloud-based dynamic resource provisioning is ideal for large volatile process landscapes with a varying amount of user requests.

2.2 Business Process Management

BPM is an interdisciplinary field, combining various process related organizational and management disciplines with computer science and Information Technology (IT) related fields, to enable the flexible management and transformation of an organization's internal and external-facing interconnected business processes [131]. Research in the field of BPM has led to a wide variety of *"methods, techniques, and tools to support the design, enactment, management, and analysis of operational business processes"* [154]. Business processes are an essential part of every organization and consist of tasks, activities, and decisions that can be performed by human-provided or software-based services. Intelligently designed and well-performed processes have a great impact on the QoS delivered to customers. Therefore organizations with better designed, flexible and well-executed processes can often outperform other organizations that deliver similar products and services [43].

2.2.1 Business Process

Dumas et al. [43] define business processes as *"a collection of inter-related events, activities and decision points that involve a number of actors and objects, and that collectively lead to an outcome that is of value to at least one customer"*.

The *orchestration* of business processes can be specified using formal process models, that describe the individual process steps, tasks, and activities including relations between them and execution constraints. An automated sequence of tasks within a process is referred to as a *workflow* and a *process instance* refers to a concrete running or executed instance of an executable process model [138].

Processes are commonly distinguished into three categories [131]:

- *core processes* are business essential and focus on creating value for external customers. Such value-adding processes include for example highly structured transactional processes like order fulfillment and customer service, but also creative processes like product development.

- *enabling processes* are also referred to as support processes, as they support and enable the operation of core processes and create value for internal customers. Such processes include human resource-related processes, information systems development or financial reporting.

- *governing processes* are management processes that are necessary to lead the company, like strategic planning or risk management processes. BPM also falls into this category.

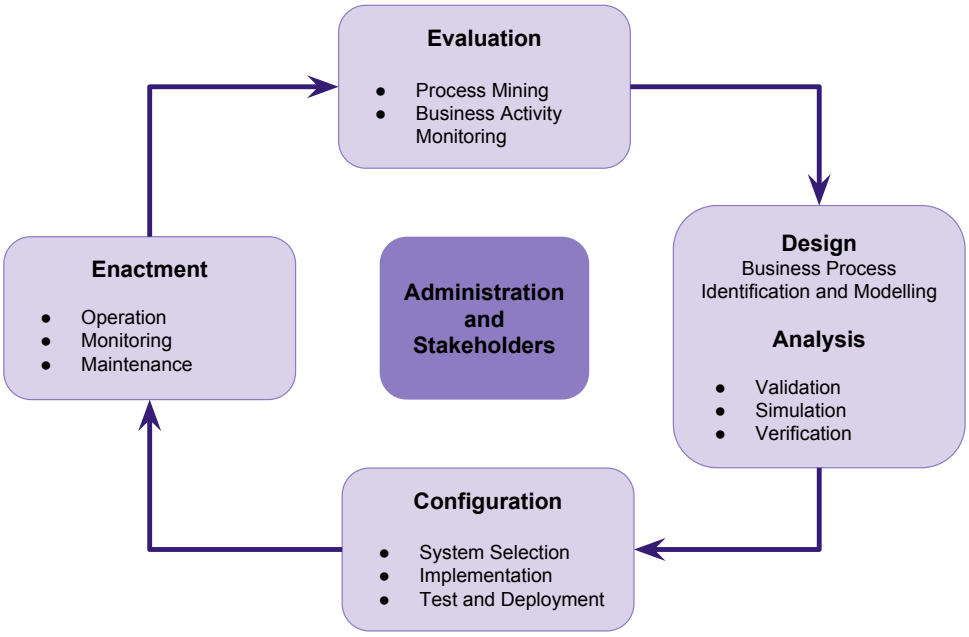


Figure 2.8: BPM Lifecycle (taken from [138])

2.2.2 BPM

Based on their definition for business processes Dumas et al. further define BPM as *"a body of methods, techniques and tools to discover, analyze, redesign, execute and monitor business processes"*.

BPM considers both intra- and inter-organizational processes and focuses on their dynamic interplay and improvement in a consistent, continuous and iterative way. The BPM lifecycle is depicted in Figure 2.8 and typically consists of the phases: design and analysis, configuration, enactment, and evaluation [138]. In the remainder of this thesis, we will focus on the enactment phase.

2.2.3 BPMS

To support BPM activities and to enact and manage business processes, software-based BPMSs are utilized. Van der Aalst et al. define BPMS as *"a generic software system that is driven by explicit process designs to enact and manage operational business processes"* [154]. A BPMS executes process instances according to the defined process models while considering execution and QoS constraints, like cost metrics and execution deadlines for individual processes or tasks. A BPMS also takes care of process *choreographies* to ensure process interoperability between different participants. The large number of concurrent process instances managed by BPMSs form an often very volatile *process landscape* that can also connect multiple organizations and in most cases is hard to predict [138].

2.3 Elastic Processes

An elastic process is a business process that is carried out using elastic cloud infrastructures and resources [138]. Elastic processes combine the concepts from BPM presented in Section 2.2 with developments from cloud computing as presented in Section 2.1, while especially taking advantage of the scalability and elasticity features enabled by the advanced resource allocation and virtualization capabilities of the cloud.

Another recently introduced term often used to describe the cloud-based enactment of elastic processes is Business Process as a Service (BPaaS). Based on the idea of cloud-based "Process as a Service" by Wang et al. [156], Accorsi defined BPaaS as *"a special SaaS provision model in which enterprise cloud offerors provide methods for the modelling, utilisation, customisation, and (distributed) execution of business processes"* [4]. Elastic processes and BPaaS can also give rise to the integration of shared services [10] in flexible process landscapes. Shared services may be provided by a centralized unit within a multi-site corporation or services can also be shared across different organizations. For the realization of elastic processes, business processes can be flexibly composed of a number of services, enacted in scalable cloud infrastructure, that can be provided using any of the cloud deployment models introduced in Section 2.1.4.

2.3.1 Elasticity Measures for Elastic Processes

The concept of elasticity is well-defined in multiple fields like economics (e.g., price elasticity [148]) or physics (e.g., material science [15]) and has more recently also been applied to the computing domain to describe on-demand cloud service provisioning [44]. All definitions of elasticity commonly describe the sensitivity of one variable in response to the change of some other variable(s) [61].

The main elasticity measures regarded for elastic processes are *resource elasticity* (change of used resources, e.g., CPU and RAM), *cost elasticity* (change of cost, e.g., regarding pricing models for on-demand instances or spot instances on Amazon) and *quality elasticity* (change of the provided QoS, e.g., availability, reliability or throughput [144]). In previous research, the focus has been on resource elasticity and the adjustment of machine power in response to some external stimuli like the number of user requests, but as the three elasticity measures are strongly interdependent, they should be regarded in combination, taking their trade-offs into account when designing a BPMS for enacting elastic processes [44]. In the following we present the three main dimensions that should be considered for elastic processes in more depth.

Resource elasticity

Resource elasticity describes the change in the amount of used resources based on the incoming demand. Both human and computational resources can provide services for business processes. Computational resources used for the realization of workflows include CPU, RAM or network bandwidth, while the demand for the resources changes with the amount of incoming service requests. By leasing and releasing the resources for the individual services, dynamic provisioning as introduced in Section 2.1.7 can be achieved [44].

Cost elasticity

Cost elasticity refers to the change of costs with the changing amount, duration, and type of provisioned resources, usually caused by a change in service requests, taking into account the pricing model of the cloud provider. Varying costs can arise in both private as well as public cloud-based deployment models. Costs incurred from so-called *utility computing* as it is offered in public clouds, e.g., by providers like AWS, are covered by pay-as-you-go pricing models [8]. These often dynamic pricing models include the costs for setting up, running and maintaining the provided infrastructure. Amazon, for example, offer their EC2 instances as *on-demand instances* with a simple pay-per-use model and no long-term commitments, and *spot instances* with fluctuating spot prices and long-term commitments. When leasing spot instances, users can define what they are willing to pay. Prices fluctuate based on the current demand, and instances of those users who bid less than the current spot price are terminated during phases of high demand. This allows Amazon to balance their resource usage and lower their maintenance costs while providing a cost-elastic incentive for users who can delay the execution of their services to off-peak times [44].

Quality elasticity

Quality elasticity describes the change in the delivered quality of a service in response to fluctuations in the current load of the system. QoS can be defined by regarding execution and response times, throughput, availability or reliability of a service while constraints may be negotiated between service providers and service consumers. With regard to elastic processes, the quality of the overall process and the quality of the enactment of multiple different processes have to be considered. As processes and workflows are composed of multiple services, the quality of elastic processes depends on the quality of the component services and their interplay. Quality elasticity for cloud applications often follows a monotonic trend such that the QoS increases with the number of utilized resources. A good example of this desired (yet not always fully achievable) property is the MapReduce framework, which offers elasticity features and offers a scalable execution speed with an increased number of processing nodes in a distributed file system. Quality measures are often strongly related to a cost function such that different customers can define different preferences for the cost and quality relation of their services [44]. This case is depicted on the customer side in Figure 2.9 which is further discussed in Section 2.3.2.

Trade-offs between elasticity measures

The three elasticity measures are highly interdependent and changes to one of the measures come with trade-offs for the other measures. The achievable quality of a service is often in direct relationship to the utilized resources, which again have an effect on the pricing. In other words, when regarding the defined elasticity measures for elastic processes as dimensions, the result space of possible configurations for optimized cloud service delivery forms a surface in the three-dimensional space defined by the three elasticity measures.

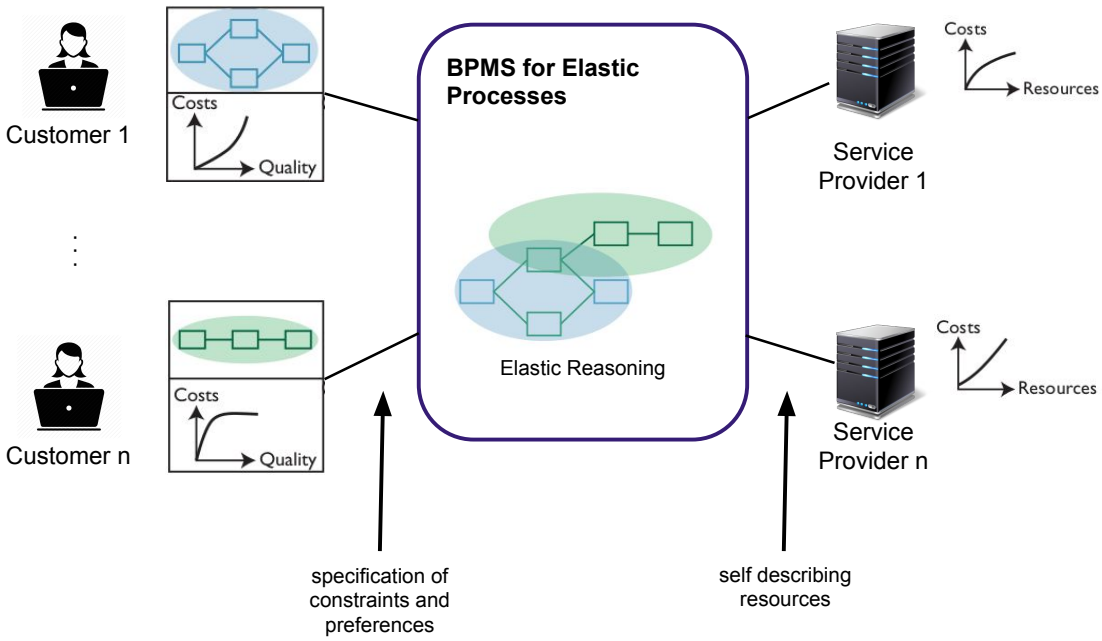


Figure 2.9: Elastic Process Environment (adapted from [44])

2.3.2 Conceptual Architecture for Elastic Processes

Figure 2.9 depicts a conceptual architecture for the execution environment of elastic processes. On the left handside, it shows multiple customers with individual cost-to-quality preference curves. Customers should be able to specify those QoS and cost preferences and pass them to a BPMS for the enactment of elastic processes. The system also takes service providers' resource descriptions including their pricing models for different resources into consideration. With the dynamic inputs from customers and service providers, an elastic BPMS should be able to calculate an optimal or fitting solution, considering all given constraints, by using elastic reasoning mechanisms. The BPMS can, for example, solve an optimization model to identify optimal service placements and also manage the optimized distribution of the incoming requests over the deployed service nodes.

It should be noted that it is not a trivial task to build a scalable and elastic system, and if not properly designed the behavior of the system can lead to unexpected costs or inefficient resource utilization [84]. Some of the most important parts that should build the basis for a reliable elastic system are continuously monitoring and re-planning components. Due to the high complexity and the large size of the search space, exact algorithms often cannot deliver a result in feasible time and have to be replaced by approximative approaches and heuristics only considering partial information [44].

2.4 Optimization Models

Optimization, also called mathematical optimization or mathematical programming is a field of applied mathematics that is commonly used in disciplines like computer science, operations research or finance. Optimization is concerned with finding algorithms for solving numerical problems to make effective decisions or predictions. Optimization problems are therefore decision problems that try to find a "best" solution and have to be solved by minimizing or maximizing a target function, possibly under certain constraints [27] [48] [169].

2.4.1 History of Optimization

The early roots of optimization can be found in the first known appearance of linear equation systems for solving practical problems 300 BC in China. Later optimization problems built the foundation for the development of theoretical mechanics and physics. Mid 1700 Euler formalized the *principle of least action* [47], which can be used to obtain the equations of motion of natural systems, while the motion can be described by solving a minimization problem. This principle also builds the basis for quantum mechanics and many other fields of modern physics. Lagrange also played a crucial role and shaped the notion of *duality*, which is a central concept in optimization [13]. In the 1800s, Gauss developed a method for solving least squares problems by solving normal equations and was also one of the first to actually apply his theoretical optimization method to a real-world scenario by predicting the trajectory of a planetoid. The real-world application of optimization theory during the first centuries of its development was the exception and not the norm, especially in areas other than physics. Only in the late 1940s, when computers became available, optimization theory could be applied to a wide variety of practical areas, with important contributors including, for example, Von Neumann, Householder and Givens [7] [48]. The use of computers pointed out a number of numerical difficulties with optimization concepts created in the past and therefore reshaped some areas in the field of optimization as it led to the development of newer and improved concepts with wide areas of application in mind. With the development of efficient FORTRAN packages for solving linear algebra problems like LINPACK [38] [39] and EISPACK [141] in the 70s and 80s and LAPACK [7] as a prominent successor that takes advantage of more modern computer architectures, in the 90s, the evolving interdependence between optimization algorithms and software became omnipresent. Scientific computing platforms like Matlab [44], R [45], Scilab [46], or Octave [47] finally facilitated the use of the FORTRAN packages by providing user-friendly interfaces and allowing to formulate problems in languages very similar to their actual mathematical notation. Linear algebra has also become a commodity, meaning that only a few experts develop algorithms and techniques that can be utilized via interfaces by users from different fields to calculate solutions without having to know details about the underlying procedures [27]. A particularly successful example of a linear algebra application that provides solutions to very large problems is the PageRank algorithm [115], which is used by Google Search to rank search results [20].

[44] https://www.mathworks.com/products/matlab.html
[45] https://www.r-project.org/
[46] http://www.scilab.org/
[47] https://www.gnu.org/software/octave/

2.4.2 Areas of Application

Optimization is an interdisciplinary field, combining knowledge from a variety of fields like mathematics, statistics, complexity theory and algorithms, to name a few. The areas of application are even wider, with a traditional focus on business and engineering and by now nearly every branch of science and technology. Optimization is often used for solving diverse planning, routing, scheduling and assignment problems. Prominent areas of application that commonly utilize optimization techniques include production planning, engineering design, economics, transportation, telecommunication, energy, finance, and management. Especially with the huge data sets that have to be processed in the field of data science, the use of optimized algorithms is a crucial factor to ensure success and progress when using ML and information retrieval techniques [7] [48].

2.4.3 Important Concepts and Basic Principles of Optimization

Optimization is a mathematical approach for solving decision problems. Decision processes generally involve the formulation of decision problems and finding (near-)optimal solutions. Decision models and their decision-making methods are diverse and include e.g., graph-based decision models like decision trees [125] or also grid analysis [53] that makes use of decision matrices. Optimization is another form of decision modeling in which one seeks to minimize or maximize a function by choosing allowed values for variables of a decision problem. The function to be minimized or maximized is referred to as the *objective function* and can be subject to some *constraints* representing limits for the decision actions [27] [169].

For modeling decision problems, the following three types of variables are used [169]:

- **Decision variables** can be chosen according to defined constraints and represent alternative courses of action and the actual decision that has to be made by a decision maker during the optimization process. An example of decision variables for a scheduling and resource allocation problem in cloud computing could be the number of leased resources of a certain type for a certain timeframe as well as the placement decision of services on leased resources.

- **Result variables** correspond to the output of the decision process and are represented by objective functions that have to be minimized (e.g., costs, the number of unused computational resources) or maximized (e.g., profit, the execution importance of critical services). Examples from the area of elastic processes would be the minimization of related costs (e.g., leasing costs and penalty costs), service deployment times considering the state of leased resources, or resource utilization measures. As can be seen, a decision problem can be complex and have multiple objectives, which have to be balanced against each other, while their individual importance can be prioritized by using weighting factors. Such weighting factors can be classified as uncontrollable variables as described in the next bullet point. The result variables are determined by the decision defined by the decision maker by choosing a specific set of values for the decision variables as well as factors that cannot be controlled by decision makers and the relation among the optimization variables.

- **Uncontrollable variables** can not be controlled by a decision maker, but have an effect on the result variables. They can be fixed parameters like the weighting factors mentioned in the previous bullet point, or they may vary and form constraints for the decision and result variables. In the area of elastic processes such constraints can be used, e.g., to limit the number of service requests sent to an instance for processing based on different considerations of the whole dynamic process landscape.

Standard notation for optimization problems

The previously discussed basic components of optimization models are commonly notated as follows [27]:

$$p^* = \min_x f_0(x) \tag{2.1}$$

subject to:

$$f_i(x) \leqslant 0 \qquad \text{, i = 1, . . . , m} \tag{2.2}$$

Equation 2.1 shows an objective function $f_0(x)$ with the vector $x \in \mathbb{R}^n$ as the corresponding decision variable. The optimal value of the objective function is denoted by p^*. The functions $f_i(x), i = 1, \ldots, m$ shown in Equation 2.2 represent the constraints that limit the objective function.

Notation of maximization problems as minimization problems

In Equation 2.1, we describe a minimization problem. Similarly, a maximization problem can be defined, but maximization problems can also easily be cast to minimization problems, when considering that for every g_0 the following Equation 2.3 holds [27] [48]:

$$p^* = \max_x g_0(x) = -\min_x -g_0(x) \tag{2.3}$$

Therefore any maximization problem can easily be transformed into a minimization form and written as follows:

$$-p^* = \min_x -g_0(x) \tag{2.4}$$

Feasibility of optimization models

An optimization model is said to be *feasible* if there exists at least one point x that satisfies all constraints. The *feasible set* of an optimization problem is defined as:

$$X = \{x \in \mathbb{R}^n | f_i(x) \leqslant 0, i = 1, \ldots, m\} \tag{2.5}$$

If the feasible set is empty or in other words, if there exists no x that satisfies all constraints, the optimization model is said to be *infeasible*.

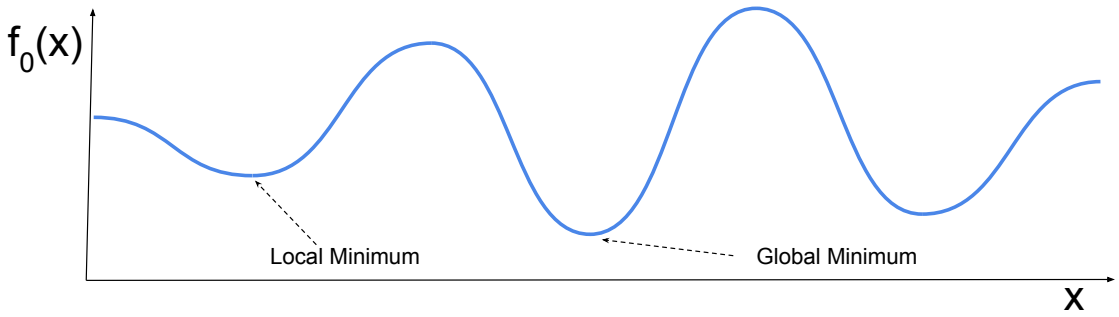


Figure 2.10: Local vs. Global Optima

Optimality of optimization models

An optimal solution of an optimization model refers to a value of the vector x of decision variables, which attains the optimal value p^* when used as an input for the objective function. Such an optimal solution is also called an *optimal point*. The set of all feasible points for which an objective function achieves the optimal value p^* is called the *optimal set* and defined as follows:

$$X_{opt} = \{x \in \mathbb{R}^n | f_0(x) = p^*, f_i(x) \leqslant 0, i = 1, \ldots, m\} \tag{2.6}$$

If the optimization problem is infeasible, the optimal set is empty.

Suboptimality: Points x are said to be ϵ-*suboptimal* if they are ϵ-close to the optimal value, which is defined as follows:

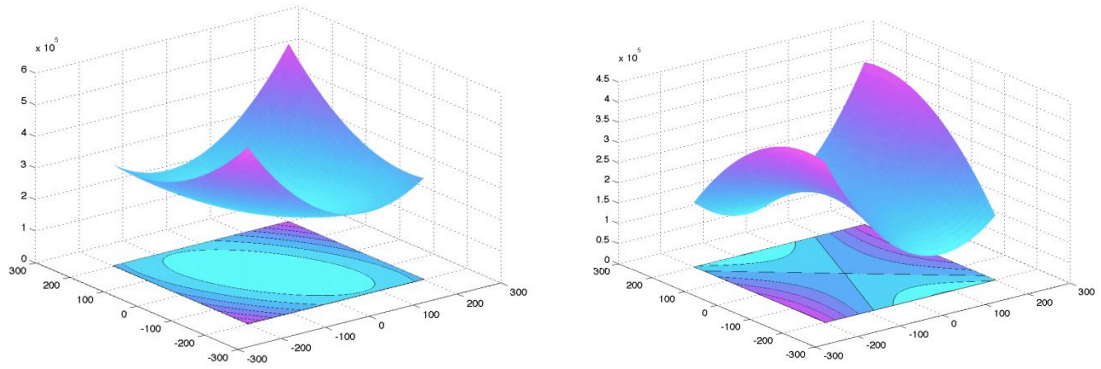
$$p^* \leqslant f_0(x) \leqslant p^* + \epsilon \tag{2.7}$$

For many optimization problems, numerical algorithms often are only able to compute suboptimal solutions and do not reach optimality [27].

Local vs. global optimality: Figure 2.10 shows the difference between a local and a global optimal solution. Local optimality only considers optimal solutions of the objective function $f_0(x)$ within a defined distance, by comparing nearby points. Local optimal solutions are not necessarily also the global optimal solution and often represent a challenge for many optimization algorithms, which can get trapped in local minima (or maxima) and therefore fail to find the actual global minimum (or maximum).

Convex optimization models

Figure 2.11a shows the graph of a convex optimization problem where the objective and constraint functions are convex. In particular, this means that each move on a downward slope brings one closer to the global minimum. In contrast, in a concave graph each step on an upward



(a) Convex Function

(b) Non-convex Function

Figure 2.11: Convex vs. Non-Convex Functions [48]

slope brings one closer to the global maximum of the function, which is relevant for maximization problems. This also means that for convex problems a local minimum is also the global minimum and general methods from convex optimization can be applied [27]. Most non-convex optimization models are hard to solve, as they can have local minima, but there are multiple methods for solving them, one of the most prominent being the *branch-and-bound* [30] [111] technique, where the optimization problem is divided into subclasses which are solved with convex or linear approximations. These approximations form a lower bound for the cost of the individual subclasses, which are then further subdivided. This process will lead to an actual optimal solution whose cost is equal to the best lower bound solution found from any of the subclasses. The optimal solution is not necessarily also unique and the process can be terminated in advance when finding a feasible solution that lies within defined tolerance limits [169].

Linear algebra and matrix theory represent some of the most important building blocks of convex optimization models [27].

Tractable vs. non-tractable optimization models

Optimization problems for which a solution can be found numerically, while the computational effort grows with the problem size (i.e., the number of decision variables and constraints) are called tractable problems. Typically, problems that can be formulated in the form of linear algebra or in a convex form are tractable while convex problems of a particular structure can be solved by using reliable numerical solvers [27].

Most optimization models with variables that are constrained to be integers are “hard” and exact solutions are often unaffordable. Such hard problems can often be defined such that approximate or relaxed solutions can be found [27] [76].

[48] adapted from the lecture slides of the EECS department at UC Berkeley for the course “Optimization Models” from spring 2015

2.4.4 Important Classes of Optimization Models and Heuristics

There is a wide range of optimization model classes and solution heuristics.

Optimization models range from convex optimization problems like *least squares* problems, *linear programs*, *convex quadratic programs*, or *geometric programs* to non-convex problems like it is mostly the case for *integer optimization* problems, *non-linear programming* problems, or *multi-objective optimization* problems [27].

Often, exact optimization algorithms are not efficient and heuristics that solve the problems faster are preferred. Such heuristics often lead to a good solution, which has the major drawback that it can not guarantee optimality or completeness [111]. Some well-known heuristical algorithms include *tabu search*, *simulated annealing*, *genetic algorithms*, *support vector machines*, *swarm intelligence* or *artificial neural networks*.

In the following, we will focus on briefly introducing some important classes of optimization models and heuristics that are especially relevant for the understanding of this thesis.

Linear programming

One of the most widely used and important optimization models is linear programming, in which the objective function as well as all constraints form linear relationships. This especially means that the effect of changes will remain in the order of the modified decision variables. Despite this restriction, many real-world business problems can be formulated as linear programs, so that fact-based decisions that produce measurable cost and performance improvements can be made. Often, this allows the implementation of automated and fast decision processes, especially for routine matters. Moderate sized linear programming problems can easily be modeled and solved using convex optimization solvers [27] [48] [169].

Similar to the general problem introduced in Equation 2.1 and Equation 2.2, a linear programming model has the following form [27]:

$$p^* = \min_x \sum_{j=1}^{n} c_j x_j \tag{2.8}$$

subject to:

$$\sum_{j=1}^{n} A_{ij} x_j \leqslant b_i \qquad \text{, i = 1, . . . , m} \tag{2.9}$$

where c_j, b_i, and A_{ij} are given real numbers.

The *simplex method* developed by Dantzig in 1947 [36] [111] [113] for solving linear programming problems involves linear equations and shows that optimization played an important role for the advancement of linear algebra. The simplex method is known to be very efficient and allows to determine optimal values of the objective function by systematically examining the vertices of feasible regions.

Linear programming furthermore builds one of the underlying key techniques for solving other classes of optimization problems and their subproblems.

Quadratic programming

Quadratic programming is a form of non-linear programming and represents an extension of linear programming, where the objective function contains a sum-of-squares.

Quadratic problems are e.g., of high relevance to the finance sector, where the linear term in the objective function can represent the expected Return on Investment (RoI) while the quadratic term can model the risk factors [169].

A quadratic programming problem has the following form:

$$p^* = \min_x \sum_{i=1}^{r} \left(\sum_{j=1}^{n} C_{ij} x_j \right)^2 + \sum_{j=1}^{n} c_j x_j \tag{2.10}$$

subject to:

$$\sum_{j=1}^{n} A_{ij} x_j \leqslant b_i \qquad \text{, i = 1, . . . , m} \tag{2.11}$$

where C_{ij}, c_j, b_i, and A_{ij} are given real numbers.

Quadratic programs arise naturally and often in the form of linearly constrained least squares problems and in this case form convex problems. In general, most convex quadratic-constrained quadratic programs are relatively easy to solve for medium-sized problems. On the contrary, quadratic problems can also have a non-convex objective function and non-convex quadratic-constrained quadratic problems are generally known to be a class of optimization problems that are very hard to solve [27].

Integer programming and mixed integer linear programming

In many cases, only expressing linear relationships is not enough and decisions involve discrete and boolean choices like the number of VMs of a certain type that have to be leased at a particular point in time and whether or not to place a service on a particular leased machine. *Integer programming* allows to express such situations, in which variables are constrained to integer values [111]. Integer programming problems are not convex and therefore considerably harder to solve than regular linear programs. If a linear programming problem includes both discrete and continuous decision variables, it is called a *MILP* problem. Mixed integer problems can also have convex quadratic objective functions and constraints [27] [48].

In Chapter 4 we will present a detailed MILP model for placing service instances in a virtualized execution environment making use of virtualization and containerization technologies. For solving our problem, we will use the CPLEX Optimizer[49], which makes use of a branch-and-cut search for solving MILP models. Branch-and-cut manages a search tree consisting of linear programming subproblems that have to be processed. The branch-and-cut algorithm includes a branch-and-bound algorithm as briefly introduced in Subsection 2.4.3 and solves the

[49] https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

subclasses using the simplex method for linear programming problems previously introduced in this subsection. Furthermore, the branch-and-cut algorithm adds a constraint to the model to cut the size of the solution domain and to reduce the number of branches necessary to solve the MILP problem.

It can often be meaningful to mix multiple optimization approaches so their individual strengths can be combined. Solutions for combining MILP with constraint programming, which both can solve similar problems in a similar way with their own strengths and weaknesses, have been proposed to solve problems that would be intractable by trying to solve them by using either of the individual approaches [76].

Multi-objective optimization problems

Multi-objective optimization programming refers to a process where two or more conflicting objective functions, subject to certain constraints, are optimized simultaneously [169]. This means that a solution will not minimize or maximize each objective to its fullest, but each objective will reach a point where further optimizations will have a negative effect on other objectives and overall a negative effect on the multi-objective problem. The need for considering multiple objectives at once is important in all fields that have to take trade-offs into consideration, before being able to make an optimal decision.

Typical examples include the minimization of weight and the maximization of the strength of a component, or the maximization of performance and simultaneous minimization of fuel consumption of a vehicle. An example for elastic processes would be the minimization of costs for leased resources while maximizing the offered QoS.

For solving multi-objective optimization problems, the weighting method can be used, in which the different objectives are assigned some weights and put together to form a single weighted objective function.

It is also possible to solve multi-objective optimization problems using heuristics, like genetic algorithms [83], which are described in the following subsection.

Heuristical approaches and genetic algorithms

Heuristics can be used to find fast approximate solutions when classical algorithms fail to find an exact solution [111]. In the past, multiple metaheuristics like simulated annealing, tabu search, swarm algorithms and genetic algorithms have been applied to a wide number of optimization problems including scheduling problems [66].

Multiple heuristics have been proposed for task-scheduling optimization in cloud computing, also considering multi-objective optimization models. Solutions using particle swarm optimization can be used for numerical optimization problems and are based on the idea of animal flocking behavior [127].

Genetic algorithms can be classified under *evolutionary algorithms* and, among multiple other areas of application, have been proposed for SaaS placement and resource optimization [167], as well as multi-objective scheduling algorithms for workflows in distributed cloud environments [77], or QoS-aware service composition [28], as they can provide a good trade-

off between runtime and solution quality of complex scheduling algorithms while considering several solutions in parallel.

Like other evolutionary algorithms, genetic algorithms try to adopt the principles of evolution to find a good solution for optimization problems and make a few assumptions about the relationships between decision variables as well as the objective function and the constraints. Genetic algorithms yield different solutions in different runs and are based on non-deterministic random samples. A population of candidate solutions is maintained and rated according to a fitness function. Members of the population are selected and mutated by performing different crossover mutation algorithms using information from current population members called the parent elements. Such crossover mutations simulate natural selection and reproduction phenomena found in nature, by matching parents according to their fitness functions while also using probabilities and by combining the parent's DNA and passing it to new members of the population. Random mutations can also be performed on subparts of the population, but in general, a "survival of the fittest" approach is followed. By using a population of solutions the evolutionary algorithm avoids becoming trapped at local optima. Fitness in constrained optimization problems can be measured by a solution's feasibility and its objective function value [111].

CHAPTER 3

Related Work

"It is not knowledge, but the act of learning, not possession but the act of getting there, which grants the greatest enjoyment."

— Johann Carl Friedrich Gauss

"With the computer and programming languages, mathematics has newly-acquired tools, and its notation should be reviewed in the light of them. The computer may, in effect, be used as a patient, precise, and knowledgeable 'native speaker' of mathematical notation."

— Kenneth Eugene Iverson

Elastic processes generally can make use of advanced virtualization and containerization technologies and optimized resource allocation and task-scheduling mechanisms. Under the terms of BPM, SOA, and cloud computing multiple relevant concepts for optimized scheduling and resource allocation have been proposed, but despite the many benefits that cloud computing can bring to the enactment of processes, only little work has been conducted on resource optimization and scheduling approaches for elastic processes [63].

In the following chapter, we will give an overview of relevant existing work and the most important approaches in the field of task-scheduling and resource allocation for single services and web applications as well as for workflows and processes in the cloud. We will introduce existing optimization approaches and heuristics, which make use of different cloud architectures, virtualization, and containerization technologies. We discuss a number of relevant papers closely related to the area of elastic processes. In the paper "Elastic Business Process Management: State of the Art and Open Challenges for BPM in the Cloud" Schulte et al. [138] give a comprehensive introduction to the field and discuss many approaches for task-scheduling and resource allocation, some of which we will also discuss in the following. Furthermore, we will especially address approaches considering containerized architectures.

It should be noted that optimization approaches for the cloud-based enactment of services and processes can focus on different stakeholders, ranging from infrastructure providers, to platform providers, application providers, and end users. Approaches especially also focus on different types of resources available to cloud providers. Cloud providers can be infrastructure, platform or service providers who can carry out the VM and container allocation using the available resources. Such resources can be physical machines that are owned by the cloud provider, in the case of which we talk about white-box resources, or resources can be leased from other cloud providers, in the case of which we talk about a black-box resource pool. In both cases, a single or multiple clouds can be involved in the process. When regarding white-box resources, the cloud provider is responsible for the optimization of the resources whereas cloud providers do not have any knowledge about the underlying physical infrastructure of black-box resources and are therefore not responsible for the optimization of the physical resources. Black-box resources can often be leased using predefined configurations as offered by Amazon EC2, but there are also providers like ICloud, who offer the definition of own VM configurations. These differences often require diverse optimization approaches and can lead to varying achievable efficiencies, especially when regarding the different available pricing schemes [97] [124]. In our conducted work, we developed a multi-objective optimization approach from the platform providers' perspective. We try to minimize the costs of leased black-box computational resources while satisfying SLAs to avoid penalty costs as far as possible and profitable for the provider.

Finding resource allocation and task-scheduling approaches for the cloud is not trivial, as it has been proven that deciding on the number of replicas and their placement is an NP-hard problem [80]. Therefore, we frequently need complex optimization techniques such as linear, non-linear, and constraint programming or complex heuristics, which can also be extended by predictive algorithms [91]. The existing literature has proposed primarily heuristic algorithms, rather than exact optimization approaches. In most cases, exact algorithms formulate problems using mathematical programming and solve them by utilizing existing solvers. Integer Linear Programming (ILP) is by far the most commonly used method for formulating such exact problems [97]. Furthermore, necessary monitoring, goal-comparison, and system adjustment procedures have to be incorporated [91].

3.1 Scheduling and Resource Allocation Approaches for Single Services and Web Applications in the Cloud

A multitude of approaches for scheduling and resource allocation have been proposed in the field of cloud computing for single services and applications [22] [65] [86] [97] [124] [152] [162]. These approaches focus on cost measures based on the actual utilization of resources as well as breaches of negotiated SLAs, but generally do not consider important process perspectives, such as the relations and dependencies among multiple process steps and the underlying services. Therefore, findings from resource allocation and scheduling solutions for single services can not be directly mapped to elastic processes, but they provide valuable insights and approaches that can be used as a basis for the optimization of cloud-based process enactments.

3.1.1 Optimization Approaches using Hypervisor-based Architectures

The majority of the work on QoS-constrained applications enacted on cloud infrastructure makes use of basic virtualization technologies using hypervisors.

Wu et al. [162] take the perspective of SaaS providers when proposing a scheduling and resource allocation policy to maximize profits and customer satisfaction, while considering QoS-based SLAs. Besides penalty rates for violated SLAs, their scheduling algorithm incorporates the heterogeneity of VMs in terms of price, startup times, and data transfer times when deciding which type of VM to assign to incoming requests.

Litoiu et al. [91] propose a cloud optimization architecture that corresponds to the cloud service model introduced in Subsection 2.1.3. For each layer of the cloud service model, they introduce an according optimization layer representing the interests and goals of the different stakeholders.

Van den Bossche et al. [152] propose algorithms that account for cost-efficient scheduling of deadline-constrained workloads while making use of public cloud providers as well as private infrastructures. This hybrid cloud support accounts for computational and data transfer costs as well as bandwidth constraints and also considers that application workloads consist of multiple parallel tasks.

Lampe et al. [86] propose a cost-effective resource allocation approach for software services, controlling the number of leased VMs for certain service types and the placement of software service instances on VM instances. They make use of a Knapsack-based heuristic to solve said assignment. Furthermore, they propose a binary integer programming-based optimization approach, which has the drawback of a high computational complexity compared to the heuristic approach. They further make an important observation that their optimization approach is designed such that it finds an optimal solution with minimal costs for individual optimization periods, but does not guarantee an optimal distribution strategy over multiple periods, as the optimal solution over multiple periods can only be computed ex-post with more available information.

In general, there are two main ways to calculate the resources to be allocated to single services. On the one side, it is possible and popular to simply focus on the incoming load and choose a reactive approach based on rules [138], on the other side, one can predict the future load based on the previously collected information. Li and Venugopal [89] focus on scaling techniques for IaaS providers, by using a reinforcement learning approach, which is based on ML techniques that assign points for different actions (e.g., migrating services, starting and stopping of VMs, etc.) if the performed actions lead to a better solution, determined by comparing the resulting QoS measures.

Other approaches that have been developed use genetic algorithms for resource management in the cloud. Yusoh et al. [167] [168] propose evolutionary-based genetic algorithms for resource optimization, considering SaaS providers that deliver applications in a composite form, meaning providers who offer a set of services that work together to deliver higher-level software applications. As multiple different software instances can share certain services, scaling decisions can be made on the level of the components instead of considering the load for the full application instances. Yusoh et al. formulated algorithms for the initial placement and the scaling of resources, that improve execution times and resource usage while minimizing costs and satisfying the constraints of the problem.

3.1.2 Optimization Approaches using Containerized Architectures

In this subsection, we continue presenting approaches for resource allocation of single services, but focus on some newer approaches that try to exploit advantages that can come with the use of containerized architectures as we have already briefly introduced in Section 2.1.5 of the previous chapter.

In the existing literature, hypervisor-based virtualization technologies are extensively used for the provisioning of elastic resources. However scaling decisions that are mainly based on optimizing the direct placement of services on VMs often come with additional costs that also have to be considered like, for example, the time and resources used for booting a full OS such that real resource isolation can be attained. In contrast, container-based virtualization technologies build upon the host's OS image and therefore do not run a completely new OS. Containers can also share binaries and libraries to remain as lightweight as possible. Therefore VM-based resource management approaches can be heavyweight and lead to higher costs for pay-per-use cloud offerings.

According to [165], the main benefits of container-based approaches for scheduling and resource allocation, when compared to virtualization technologies that are based on hypervisors are the following:

- Containers do not virtualize a full OS and are therefore lightweight. Containers can be created instantly and significantly quicker than VMs.
- Containers can be configured and reconfigured easily so that they correspond to the actual demand for computational resources. They do not need a full, time-consuming reboot, as it is the case with VMs. This also means that cloud providers who offer containerized services instead of offering full VMs do not need to provide a range of predefined sets of instance types, as it is the case with VM.
- Containers allow implementing an effective DevOps approach and abstract from inconsistencies of underlying platforms, by unifying the development, test and production environments.

Container level scalability in PaaS clouds is analyzed by Vaquero et al. [155], who show that the use of containers for deploying applications is highly relevant, as PaaS clouds mainly offer a ready to use execution environment for applications, which allows developers to focus on programming rather than on setting up the environment. The relevance of containers and container middlewares for PaaS clouds is also discussed in detail by Dua et al. [42] and by Pahl [116], especially pointing out the advantages of container-based isolation over pure VMs-based isolation as well as the possibilities for DevOps and the interoperable orchestration throughout container clusters. Pahl et al. [117] show that research on cloud container orchestration has only very recently gained attention, after the introduction of Docker containers in 2013, with a strong increase in research papers mid 2015. At the same time, Weerasiri et al. [157] have observed that while infrastructure-focused virtualization approaches tend to adopt OS-level hypervisors, approaches for resource virtualization in PaaS and SaaS environments are moving towards adopting environment-level container managers.

Xu et al. [165] propose a resource scheduling approach for virtualized cloud environments that solely builds upon virtualization technologies and does not use hypervisor-based virtualization. They apply a stable matching model, which is a game theoretic problem that aims to find a stable mapping between agents with certain preferences. A mapping from containers to physical machines which aim at reducing the response times for customer requests while improving the resource utilization of cloud providers is proposed.

Zhao et al. [171] discuss the challenges of scheduling approaches using lightweight container-based isolation and propose an optimization model for locality-aware scheduling for containerized cloud services. They show that round-robin approaches for resource allocation and scheduling are associated with a significant performance overhead for both disk I/O and network traffic, as they do not consider dependencies of containers. Resources shared by containers on the same node can become a bottleneck, for example, for HDDs a concurrent access leads to a decreased I/O throughput, also if the HDD is partitioned for disjoint access, as there only exists one single head for disk seek and data access. Zhao et al. propose their optimization model based on the following three observations:

- When scheduling multiple compute-intensive container-based applications on a single node, there is only a very minimal and neglectable overhead when compared to the execution of only a single compute-intensive application on the node. This shows the strength of container-based isolation of CPU resources.

- When scheduling multiple data-intensive applications on the same node, a very high overhead is created. In the case of two applications that share the same node, the execution time for both applications practically doubles, compared to the individual execution of the applications on the node.

- When deploying a compute-intensive and a data-intensive application on the same node, they only have minimal impact on each other and their individual execution times are nearly the same as when executing them separately on the same node.

Therefore it is important to find solutions that take points mentioned above into account when developing scheduling and resource allocation approaches in containerized architectures, instead of simply incorporating state of the art round-robin policies.

A multi-objective optimization problem for finding an optimal resource allocation and task scheduling approach of applications, which makes use of both, VMs as they are commonly offered by infrastructure providers, as well as Docker containers which run on the leased VMs, is proposed by Hönisch et al. [65]. The authors especially target PaaS providers who make use of IaaS cloud services like EC2 and need to place applications on the leased resources. The approach allows for a better and more flexible utilization of the leased VM resources by isolating applications in containers and placing multiple isolated applications on the leased resources. The aim is to scale services or web applications running inside of containers, while each application is deployed in the instance of a corresponding container type. A VM instance can host multiple container instances of different types and each container type can be deployed on multiple VMs instances. Different container instances of the same type can have different configurations that

define requirements in terms of needed CPU-shares and RAM, which have to be met by the underlying VM. For each application, a Dockerfile specifies how to start a new container instance of the type that corresponds to the application. The scaling approach is referred to as "Four-Fold Auto-Scaling" by the authors as it incorporates four dimensions, by performing both horizontal and vertical scaling for VMs as well as containers. Horizontal scaling refers to the change in the number of VM and container instances of the system, while vertical scaling refers to the change of computational resources available to the same number of VM or container instances in a system. The optimization problem decides on how many VMs of which types are needed and how many containers for different applications of which configurations have to be distributed in which way among the leased VMs. The computed leasing plan and container placement strategy aims at minimizing costs and meeting given SLAs.

As already mentioned in this section Xu et al. [165] pointed out the advantage of containerized architectures that allows cloud providers to offer containerized services in a flexible manner without having to provide a predefined set of instance types. As Hönisch et al. [65] work with containers that are assigned a configuration from a predefined set, there might be some room for improvement in the presented approach, by configuring containers in a more flexible manner and assigning continuous configuration variables instead of choosing from a rather small discrete set of configurations. The main considerations and structure of our work are comparable to this container-based solution developed by Hönisch et al., but instead of only regarding the execution of single applications, we will present a solution for the execution of elastic processes.

The problem of deploying application containers in the cloud, while using an elastic set of computational resources in the form of VMs, has recently also been addressed by Nardelli et al. [112]. The authors propose an ILP problem that considers heterogeneous container requirements and VM resources when calculating an optimal allocation as well as a runtime reallocation of containers to the elastic set of leased VMs. The optimization model considers deployment times and deployment costs for containers as well as multiple QoS metrics which can easily be reconfigured. The presented approach outperforms the greedy first-fit and round-robin heuristics, which are often used for deploying containers on VMs.

The current research on Docker containers mostly focuses on the deployment at a single cloud. Cluster managers like Kubernetes [1], Mesos [2], IBM Bluemix Container Service [3], and Amazon EC2 Container Service [4] mostly do not allow to run containers across multiple clouds and data centers. This comes with a number of limitations that go beyond the pricing-related disadvantages that come with a "provider lock-in" [2]: Firstly, it is not easily possible to perform cloud-bursting by, for example, scaling out to the public cloud when an on-premise private cloud is busy or unavailable. Secondly, single cloud approaches do not offer fault recovery mechanisms and do not provide resilience when it comes to zone or data center outages. Thirdly, privacy regulations that require keeping private data in-house while performing other services outside of the private cloud can not be enforced. Furthermore, it is not easily possible to optimize data transfer costs by keeping services near different data sources.

[1] https://kubernetes.io/

[2] http://mesos.apache.org/

[3] https://www.ibm.com/cloud-computing/bluemix/containers

[4] https://aws.amazon.com/ecs/

Only recently a shift to considering multi-cloud cluster management has been observed, with Kubernetes now providing a basic support for running a single cluster in multiple failure zones [5]. Abdelbaky et al. [2] propose a prototype framework that uses a constraint programming model for targeting the limitations mentioned above by allowing the deployment, migration, and management of containers across clusters in multiple clouds and data centers, while being able to use any container scheduler and also taking user and provider constraints into consideration. The model can find a feasible, but not optimal deployment solution.

All the mentioned approaches and strategies in this section are developed for single applications and do not account for BPM as they do not consider the process perspective with its different relations among services. Nevertheless, the findings are very relevant and can partly also be used for the development of our optimization approach for elastic processes.

3.2 Scheduling and Resource Allocation Approaches for Workflows and Business Processes in the Cloud

Scheduling and resource allocation approaches from cloud-based application provisioning cannot be directly mapped to the area of BPM without further adaptations. For the realization of elastic processes, it is especially necessary to develop techniques that also consider the state of business processes, which means that we also have to account for the data and control flows that come with the execution of processes [138].

In this section, we present existing approaches for the execution of workflows and business processes in the cloud and discuss their strengths, limitations, and applicability to the domain of elastic processes. Generally, there exist very few solutions supporting cloud-based elasticity with regard to business process executions. The existing approaches also hardly make any use of containerized architectures, that could allow for the realization of more fine-grained scheduling and resource allocation approaches.

3.2.1 Optimization Approaches using Hypervisor-based Architectures

A large number of approaches for the execution of SWFs in the cloud [67] [78] have been proposed. The approaches schedule the workflows on computational resources based on optimization algorithms and heuristics that consider cloud-based characteristics and pricing models [3] [25] as well as execution and data transfer costs [119]. A number of approaches also regard the runtime of the SWFs [67] [147] and account for QoS constraints [3]. The main limitation of approaches for SWFs is that they do not consider concurrent workflows, which are the norm in the area of BPM, but only schedule single workflows. Furthermore, SWF approaches are generally more data flow-oriented, while business process scheduling is used in complex landscapes, with concurrent process requests. Such process landscapes often share the same services and tend to be more control flow-driven [93].

There are only few scheduling and resource allocation approaches that allow for a concurrent execution of elastic processes in a cloud environment.

[5] https://kubernetes.io/docs/admin/multiple-zones/

Xu et al. [164] developed a solution for scheduling multiple workflows with multiple QoS constraints on the cloud. Their multiple QoS constrained scheduling strategy for multiple workflows accounts for the different quality requirements of different users with concurrent service requests. The proposed scheduling algorithm considers multiple factors that affect the total makespan and cost of workflows. It aims at improving the mean execution time and execution cost of all workflows as well as the scheduling success rate, meaning the number of executed workflows that could be finished and satisfied their QoS requirements. In our approach, we share some assumptions that have also been made by Xu et al., for example, that processes can share single services and concurrent executions of the same processes can occur with varying quality constraints.

A multi-objective genetic scheduling algorithm of BPEL workflows in distributed cloud environments was presented by Juhnke et al. [77]. Their approach considers data dependencies between BPEL workflow steps and uses workflow execution times, composed of computing times and data transmission times, as well as the costs for the needed cloud-based computational resources to compute a solution. By using weights, the Pareto-optimal solution of the multi-objective heuristic can be transformed into a unique solution.

A number of strategies for matching and scheduling that are based on similar assumptions have also been proposed by Bessai et al. [14]. The authors also aim at scheduling process steps of business processes using elastic cloud-based resources, while considering the features of cloud computing and optimizing for execution times and execution costs.

Both the approaches of Juhnke et al. and Bessai et al. provide solutions that allow for a parallel execution of processes, but in contrast to our work, their approaches do not account for SLAs like deadlines.

SLAs like deadlines for single tasks and processes have been considered for sequential processes by more recent approaches of Wei and Blake [158] [159]. They introduce an adaptive workflow configuration algorithm for dynamically managing VMs for service-based workflows in the cloud. Services, which are the building block of workflows, are assigned to VM instances which are aggregated on physical machines. The algorithms decisions are based on predictive results. In our approach, we develop an optimization model that also considers SLA for complex process models.

A scheduling and reasoning algorithm for elastic processes using cloud resources was proposed by Hönisch et al. [64]. They also consider SLAs, in particular, execution deadlines for computing an efficient scheduling plan. The demand of upcoming process steps of multiple workflows is calculated to find an appropriate cloud resource allocation plan. Their proposed scheduling and reasoning approach is executed in ViePEP [137], their BPMS which can execute workflows consisting of multiple service steps in the cloud. The initially presented algorithm is also limited to simple sequential processes and only supports one VM type.

Based on their previous work, Schulte et al. [139] have later formulated an optimization model for scheduling and resource allocation for elastic processes and designed a heuristic that is based on the proposed model. The heuristic aims at predicting the resource demand of process steps and the total cost for leasing different types of VMs. Also, this heuristic was integrated into ViePEP and is limited to sequential workflows.

In their later work, Hönisch et al. [63] extended ViePEP and their basic scheduling and resource allocation approach for elastic processes, by developing an MILP-based optimization approach. The cost-based optimization problem accounts for complex elastic processes and not only sequential workflows. It computes a scheduling and resource allocation plan based on costs and execution deadlines. Their service instance placement model considers a worst-case scenario of the known process structure and is also integrated into ViePEP. The authors subsequently released a similar approach that is also able to operate in hybrid cloud environments [62].

All the mentioned approaches in this subsection use hypervisor-based virtualization to enact workflows and processes in the cloud. To make use of advantages of container-based virtualization approaches, as discussed in Subsection 3.1.2, we propose a more fine-grained model for resource allocation and scheduling of elastic processes.

3.2.2 Optimization Approaches using Containerized Architectures

There exists hardly any related work that considers optimization approaches for scheduling and resource allocation for workflows executed on container-based cloud architectures. The very few existing approaches have been proposed in the context of SWFs [50] [172]. The reason for this is that container-based architectures can solve some of the biggest issues for SWF platforms, who often struggle with heterogeneous applications with different and incompatible external packages and dependencies during software development and execution. For example, different versions of Python or R can lead to problems in reproducing experimental results. Furthermore, developers face an increased difficulty when attempting to reuse methods [50] [150]. Developers have often tried to solve this problem by offering VM snapshots that capture the entire development environment [50]. But as hypervisor-based virtualization is associated with a considerable overhead on resource utilization and long startup times impose latency on the system [46], a far better solution would be the isolation of each application's runtime environment in standardized lightweight containers. But not only SWFs would benefit from such an approach. In fact, all workflow execution platforms, including BPMS, have to deal with the fact that each computing node has to install the execution environment of workflow steps that have to be executed on them before the execution of a step can be initialized [50]. Especially for workload-based dynamic allocation of computational resources, these properties become critical, as a solely hypervisor-based virtualization approach would include an overhead for resizing and rebooting VMs whereas lightweight containers can drastically cut costs of software deployment and allow for a more flexible execution of complex process landscapes [150].

Gerlach et al. [50] proposed a solution that uses containers to achieve software isolation for solving the software deployment problem described above. The authors aim is to find a general solution that can be applied to all SWF platforms, as they all face the software deployment problem.

Also, Zheng et al. [172] propose a solution with regard to some main design challenges for an SWF scheduler. They enable a two-level resource scheduling model, as to facilitate the sharing of resources among different frameworks. Zheng et al. have come to the conclusion that deploying and running high throughput SWFs on a container-based scheduling platform leads to a higher system efficiency and lower performance loss.

In our work, we exploit the advantages that can come with task-scheduling and resource allocation approaches using container-based architectures and propose an elastic BPM framework and optimization model that allocates resources on a fine-grained level, using containers and not full VMs for isolating applications.

3.3 Conclusion

In this chapter, we discussed the existing related work on task-scheduling and resource allocation approaches for cloud environments. We examined the current literature from four main perspectives: Firstly, we looked at task-scheduling approaches for single services and applications. Secondly, we discussed task-scheduling for workflows and processes. Thirdly, we examined hypervisor-based cloud architectures for realizing such elastic scheduling approaches. Lastly, we discussed literature that makes use of container-based cloud architectures for more efficient scheduling solutions.

When summing up the main findings for each of the four perspectives, we can observe the following:

- **Task-scheduling for single services**: has been widely addressed from multiple perspectives and with a wide variety of both simple and advanced solution approaches, which consider different cost, resource utilization and SLA-based quality measures. Many findings from such research studies are of importance and single aspects also have to be considered when optimizing for elastic processes, but generally scheduling approaches for single tasks and applications do not take the process perspective into account. This means that relations and dependencies which exist among process steps in different process models and process instances are not considered by single service scheduling approaches. However, as such relations form the basic building block of business processes, it is essential that a BPMS takes them into account.

- **Task-scheduling for workflows**: has been less widely addressed with most approaches being related to SWFs and only a few approaches addressing elastic business processes and concurrent workflows. SWFs only schedule single large scale workflows. Therefore, they do not account for concurrent workflows and business processes which can share elastic resources among different process instances. Some approaches account for concurrent workflows but do not consider SLAs. Only very few approaches also take QoS constraints and SLAs into account. Some of these solutions consider assumptions that we also address in our work, one of the most important beeing that multiple process instances of the same or of different processes can share single services concurrently and with varying QoS constraints. The few existing task-scheduling approaches for concurrent workflows and processes allocate resources on a rather coarse-grained level, as they allocate single services to full VMs and do not account for more flexible light-weight container-based architectures that can also provide the needed isolation in a fine-grained manner, while also allowing for more dynamic horizontal and especially vertical scaling realizations.

- **Hypervisor-based cloud architectures**: are proposed by most related work solutions for both single applications as well as workflows and processes. These solutions do not exploit the advantages that come with light-weight container technologies.

- **Container-based cloud architectures**: can deliver a great number of advantages, as they offer a more flexible and light-weight deployment and execution environment for services while allowing the realization of flexible resource allocation approaches. Research on container technologies is generally still very young and there is a lack of tools supporting container management and orchestration, especially in clustered clouds [117]. Findings from VM-based auto-scaling technologies can partially also be used for containers, however, next to having to consider more flexible solutions, container-based auto-scaling problems also entail a resource allocation problem, as they have to be scheduled on VMs or physical machines [124]. Container-based approaches for task-scheduling have only very recently been addressed by related work focusing on single applications. Except for a few first container-based task-scheduling approaches in the context of SWFs, container-based architectures have not been proposed for workflows and processes. To the best of our knowledge, there do not exist container-based task-scheduling approaches for concurrent workflows and elastic processes.

Generally, we can observe that for each of the four perspectives, both, heuristics, as well as optimization algorithms, have been proposed, but the number of presented heuristics far outnumbers the number of exact optimization approaches. Many proposed algorithms are rather simple approaches based on the well-known bin-packing problem for which simple heuristics exist. Other heuristic approaches make use of more complex algorithms, e.g., by utilizing Knapsack-based heuristics or advanced evolutionary based algorithms. The problem with heuristic algorithms is that they mostly do not offer any performance guarantees, which can lead to well-performing experiments in controlled environments, but still yield a rather poor performance in real settings, especially for larger and highly constrained problems. The proposed exact algorithms on the other hand generally formulate mathematical programming problems which are solved by appropriate existing commercial or open-source solvers. These exact problems come with the limitation that they are only applicable to smaller problem sizes, as they often have a high computational complexity and even mid-sized problem instances can take a long time to be solved. This lies in the nature that the task placement problem for elastic processes and applications is an NP-hard problem in the strong sense and therefore exact solutions with polynomial or pseudopolynomial time can not be formulated, but efficiently solvable special cases can be designed [97]. When using a solver for mathematical programs it will deliver optimal results, but the input-dependent worst-case runtime can become exponential and therefore large problems can often not be solved in a reasonable amount of time.

To make use of the possibilities that come with container-based cloud architectures, we developed a BPMS which makes use of a multidimensional optimization approach for light-weight and fine-grained resource allocation and task-scheduling and allows for the enactment of multiple concurrent elastic process instances in flexible cloud architectures. The approach considers process related quality measures like SLAs and also accounts for complex process models.

CHAPTER 4

Approach

"There's no way to approach anything in an objective way. We're completely subjective; our view of the world is completely controlled by who we are as human beings, as men or women, by our age, our history, our profession, by the state of the world."

– Charlie Kaufman

"You must change your approach in order to change your results."

– Jim Rohn

As already mentioned in the introduction and also illustrated in Section 1.2, BPM plays a central role for a wide variety of organizations across different industries. For being able to guarantee an efficient and effective execution of automated processes and workflows, a cloud-based BPMS has to be designed. Such a BPMS is responsible for scheduling and enacting all steps of many parallel incoming process requests on available computational resources in a suitable manner.

The number of required resources depends on the diverse processes and the services they are composed of. Services are executed concurrently and can be resource-intensive. Processes can generally be more data-intensive or more compute-intensive, while we focus on requirements for computational resources like CPU and RAM and do not consider data-transfers. Some processes or single process steps are urgent and have to be executed immediately, while others can be postponed to the future and have to be finished by a deadline. Additionally, a number of process requests might be reoccurring, so a BPMS can plan for them in advance, while many processes can have highly fluctuating and unpredictable request patterns. The varying number of process requests of different types and different priorities has a large impact on the type and amount of needed computational resources over time, with a large number of resources needed during peak times and fewer resources needed during off-peak times. To avoid the problems related

to situations of over-provisioning and under-provisioning as introduced in Subsection 2.1.7, the computational resources have to be leased, managed, and released by the BPMS on demand.

For a dynamic provisioning for elastic processes in a scalable cloud system, different horizontal and vertical scaling techniques as introduced in Subsection 2.1.6 can be used. The utilization of recent virtualization and containerization technologies as described in more detail in Subsection 2.1.5, enables the realization of those dynamic scaling techniques, while each technology offers different advantages and comes with different limitations. By combining various technologies, their individual strengths can be exploited for designing highly flexible systems with fine-grained resource allocation capabilities.

The needed computational resources can be deployed using different cloud service and cloud deployment models as introduced in the Subsections 2.1.3 and 2.1.4 respectively. In this thesis we take a service provider's perspective when designing a BPMS for the enactment of elastic processes. While cloud providers can generally manage both white-box resources, meaning resources owned by the provider, as well as black-box resources, meaning resource pools from which VMs can be leased without knowledge about the underlying physical infrastructure [97], in this thesis, we only focus on cloud service providers utilizing black-box resources. This means in case the service provider utilizes internally managed computational resources, like a companies' own private cloud infrastructure, we assume that this private cloud infrastructure is managed and internally charged by another functional area of the company. In particular, we examine ways service providers can make use of cloud resources offered using different cloud deployment models, but we do not consider scheduling and resource allocation on the level of physical machine resources. This thesis targets PaaS providers, who offer a platform for deploying and executing processes composed of different services while scheduling those services on leased cloud-based resources.

When looking at the related work discussed in Chapter 3, we can not observe any currently existing task-scheduling and resource allocation approaches targeting cloud-based concurrent workflows or elastic business processes that make use of containerization technologies. Because of the fine-grained resource allocation possibilities that can arise from utilizing container-based architectures and also due to the other advantages of container-based virtualization and task isolation that we have already discussed in more detail in Subsection 2.1.5 and Chapter 3, we propose a novel solution which combines both, hypervisor-based as well as container-based virtualization for task-scheduling and resource allocation of elastic business processes. Our solution implements fine-grained resource allocation techniques and considers concurrent process and service requests.

Existing scheduling and resource allocation approaches for elastic processes vary in the considered parameters and assumed preconditions. In the following, we discuss the preliminaries including necessary background information, assumptions, and parameters of our considered task-scheduling and resource allocation problem. We present the general requirements for the realization of elastic processes, the specific system requirements for their enactment in container-based architectures and the theoretical concepts behind our presented BPMS. We present a system model for our approach, discuss important concepts using a working example, and formulate our approach as a MILP optimization problem. Furthermore, we illustrate the execution of our designed working example using the presented approach.

4.1 Preliminaries

Based on our observations from the related work and state of the art we define some preliminaries for our task-scheduling and resource provisioning approach for elastic processes in containerized cloud infrastructures. The presented general preliminaries for elastic processes are similar to those used by the related work of Hoenisch et al. [63].

4.1.1 General Requirements of Elastic Processes

Elastic processes are often part of wide business process landscapes that consist of a large number of different business processes. In our approach, we solely refer to automated processes, which are also known as workflows [88] [93]. Such processes are comprised of multiple process steps that can be carried out as tasks by software services using machine-based computational resources without the need for human interaction. Single *services* are offered as software and form the building blocks of *process models*. A service can be used in multiple different process models. A process model can be requested, leading to an executable *process instance*. As soon as a service is deployed on a cloud-based infrastructure, we obtain a *service instance*. Service instances are hosted in the cloud in an isolated way. A *service invocation* describes the unique invocation and execution of a service instance to serve a requested process instance. A service invocation can only be executed on one service instance, but multiple service instances for the same software-service can be deployed in a cloud infrastructure. A service instance can serve multiple incoming service invocations simultaneously, as long as it has sufficient assigned resources. This means that multiple process instances of the same or different process models can share service instances for process steps that need the same software service. The shared service instances allow for a flexible use of the available cloud infrastructure and build one of the key properties of elastic processes.

Process models can be comprised of complex constructs. We do not only consider the execution of sequential processes but also allow the definition of arbitrary AND-blocks, XOR-blocks, and repeated loop sections [153]. We assume that the next step of a process instance is known as soon as the preceding step has been scheduled.

A client can request the execution of any defined process model and is the process owner of the so created process instances. The process owner can define QoS constraints in the form of SLAs for each requested process instance. SLAs can be comprised of multiple SLOs, with the deadline being one of the most important SLOs. While business-critical processes often need to be executed immediately, processes that are less critical can have loose deadlines and their execution may be postponed into the future. Therefore, we allow process owners to set a deadline by defining the maximum execution time for a requested process instance. In this thesis, we focus on execution times as SLOs, but we do not explicitly cover other possible QoS attributes of the processes and their services like availability, reliability, or throughput, which is in line with previous work [63].

4.1.2 System Requirements for the Enactment of Elastic Processes in Container-based Architectures

Due to the fact that multiple clients can request the execution of a varying number of process instances at any time, the business process landscape is highly dynamic. Such an ever-changing process landscape requires different amounts of computational resources during different time periods. Therefore, the leased cloud-based computational resources need to be optimized for obtaining a cost and resource efficient system landscape.

As already mentioned, processes are composed of individual process steps that are realized through software services. The currently existing related work has focused on isolating such software services by deploying them on individually leased VM instances. Most cloud providers only allow the leasing of VM instances according to a discrete set of predefined resource sizes and for predefined time frames. This can lead to a waste of leased computational resources, as VMs are bound to a certain software service also when only a few number of requests that need less than the available resources have to be processed. Therefore, the use of lightweight Docker containers for resource isolation and application deployment comes with the great advantage of a more fine-grained resource control. Multiple Docker containers can be deployed on different leased VM instances and starting new container instances only require a fraction of the time needed to set up running VM instances.

A system for enacting elastic processes that makes use of both containers and VM resources has to consider multiple preconditions. As discussed in Subsection 2.1.5 and in Chapter 3 different providers offer a multitude of possible leasing models. In the following, we focus on the basic attributes and requirements of the most frequently offered and used cloud models. We especially take the following technical considerations into account:

- Computational resources like CPU and RAM can be leased in the form of VMs from cloud providers according to their discrete resource leasing models. When leasing new VM instances we have to consider the startup time for instantiating a new VM before it can be utilized. Cloud providers usually offer VMs for fixed BTUs [1]. The leasing duration can be prolonged by leasing a certain VM for multiple BTUs in a row. The resources of running VM instances are considered as fixed and cannot be extended during a BTU. If a larger or smaller instance is required at the end of a BTU the old instance has to be released and a new instance with updated resource configurations can be leased. This will lead to an additional VM startup time that has to be considered before the new VM instance can be utilized. An arbitrary number of VM instances with arbitrary resource configurations can be leased at any time.

- Software services are not directly deployed on VMs but packed into Docker images. Multiple Docker containers that offer a specific software service can be created from the corresponding Docker image and deployed on different leased and running VM instances. Service instances in Docker containers can simultaneously be invoked by different requests. Docker containers for the same software service can be deployed on arbitrary many VM instances but each VM instance can only run Docker containers of different

[1] https://aws.amazon.com/ec2/pricing/

service types. Each Docker container can be assigned a varying amount of computational resources. The assigned resources for a container can be adapted on demand and are only limited by the actually available resources of the underlying VM.

- When running containers on VMs some important times have to be considered. When a Docker image is first pulled from a registry onto a leased and running VM instance, the time for downloading the image on to the VM instance has to be considered. As soon as an image has once been pulled, it remains on the VM instance for its entire lifetime, until the VM instance is shut down, or a new version of the image is published in the registry, which we do not explicitly consider in this work. This especially means that the image of Docker containers that were once running on a VM, will remain on the VM also after the associated containers are removed. Docker containers can only be instantiated and started from existing images on a VM. The actual startup time for such Docker containers is in the order of a few seconds.

The cloud-based computational resources can be acquired from both private and public cloud providers, while multiple pricing models can be considered.

4.1.3 BPMS for the Execution of Elastic Processes

To implement a cost optimized task-scheduling and enactment of elastic processes a BPMS that additionally offers functionalities of a cloud controller and a container cluster manager as introduced in Subsection 2.1.5 and in Figure 2.5 is required. The BPMS needs to manage cloud based computational resources, deploy the necessary amount and type of containers with invokable software services on the leased resources and schedule incoming process requests on the available service instances. Both the cost for leasing cloud-based computational resources as well as penalty costs for violated SLAs have to be considered by the BPMS when making task-scheduling and resource allocation decisions.

The most important responsibilities of such a BPMS and the main goals that have to be considered when designing a system for the enactment of elastic processes in a container-based cloud infrastructure can be summed up as follows:

- Concurrent incoming process requests from multiple clients need to be accepted and processed.

- A process scheduling plan has to be created and the single process steps of the concurrent process requests have to be scheduled and executed by corresponding software services. Therefore, the resource demand of the planned service invocations has to be computed.

- The needed software services have to be assigned a sufficient amount of computational resources. For allowing a flexible and fine-grained resource allocation, containers should be used to run the software services in an isolated mode.

- Cloud-based computational resources have to be managed, leased and released based on the task-scheduling and container deployment plan.

- SLAs, in our case deadline constraints, for process instances have to be considered and met, as a violation of such constraints leads to penalty costs. Process instances with a closer deadline have to be assigned a higher importance, so their execution can be accomplished in time.

- Costs that occur from leasing cloud-based computational resources and violation of deadline constraints have to be minimized. To compute the costs, an appropriate cost model has to be developed and a cost/performance analysis has to be performed when designing task-scheduling and resource allocation plans.

These listed tasks have to be performed repeatedly as new process requests arrive, when the process landscape changes, and whenever new process steps have to be scheduled. The careful computation of a task-scheduling and resource allocation plan is essential to avoid costs that can arise from leasing additional computational resources for scheduling inefficiently planned service invocations that can not be delayed any further without causing penalty costs for the corresponding process instance. Leasing and releasing of computational resources and the execution of service instances on them should be optimized and task-scheduling and resource allocation have to be replanned continuously. Therefore, it is necessary to implement a self-adaptive auto-scaling system that optimizes the business process landscape.

Self-adaptive Auto-scaling Systems

In this work, we will extend and reconstruct ViePEP [137], a research prototype that was designed as a BPMS testbed for the execution of elastic processes in public and private cloud environments. ViePEP currently only considers task-scheduling and resource allocation using solely hypervisor-based virtualization. The platform has recently been extended and offers the possibility to place and run Docker containers on leased computational resources, but currently there exists no optimization approach for making use of the more fine-grained resource allocation possibilities. In our approach, we extend ViePEP and allow for the self-optimization of process landscapes that make use of fine-grained resource allocation and software service isolation properties that come with the utilization of containerized architectures. We design a BPMS that automatically provisions a set of VM resources, places different types of isolated containers for software services on those resources and optimizes the process landscape according to fluctuating application workloads and resource demands. The system dynamically tunes the type and number of utilized VMs and containers while making sure to meet its performance goals, which are the minimization of costs for the computational resources and the satisfaction of SLAs, to minimize penalty costs. These design goals correspond to a classic automatic control problem which is commonly abstracted as the MAPE-K feedback control loop [82] that continuously repeats itself over time. MAPE-K refers to the four phases Monitoring, Analysis, Planning, and Execution that have to be performed over a Knowledge Base which is utilized during all phases. Feedback loops are the most important entities when designing self-adaptive systems, which can operate in continuously changing environments with emerging requirements that are often hard to predict at design time [19].

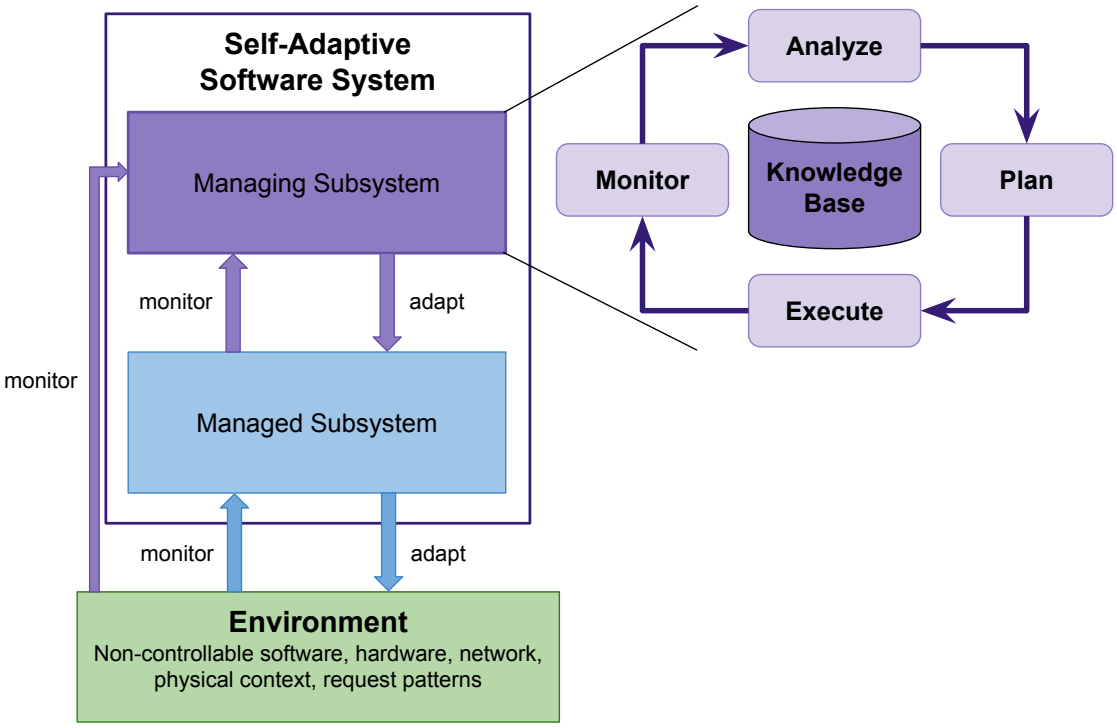


Figure 4.1: MAPE-K loops in self-adaptive software systems (adapted from [82] and [160])

We illustrate MAPE-K feedback control loops in Figure 4.1. Self-adaptation of a software system is based on the design principle *separation of concerns* [160]. A self-adaptive system generally consists of a managed subsystem which is adapted by a managing system. In our case, the managed subsystem can receive and execute process requests from multiple clients on different computational resources. The subsystem is able to deploy software containers on leased VMs and execute concurrent process steps. The managing system operates on top of the managed subsystem and is responsible for the optimization and adaptation logic. This means that the managing system decides on the actually needed number and type of VMs, the number and type of containers that should be placed on the computational resources and their allocated resource amount, as well as the order and timing of the execution of concurrent process steps to meet process deadlines, while minimizing costs for leasing computational resources. The managing system makes decisions based on available information about the managed subsystem and environment parameters outside of the system. For executing its application logic, the managed subsystem incorporates the feedback from the managing system and also makes use of data that it monitors from the system's environment. In our case, the managed subsystem can lease VMs, deploy Docker containers with software services and perform the necessary process executions according to the managing system's plan, while it also makes use of information of existing leasing contracts with cloud providers to decide if VMs need to be leased for further BTUs, so they correspond to the given leasing plan.

Auto-scalers face different challenges during each phase of the MAPE-K cycle that have to be considered during the design of such systems [124]. The most important challenges our approach addresses for each phase are the following:

- **Monitor**: The system needs to monitor its state, especially the running and scheduled process instances, the leased computational resources, and the resource usage in terms of CPU and RAM of deployed containers. Furthermore, the system needs to monitor the number and type of incoming process requests. The information about the internal system state and the changes from the external related environment are an essential part of the feedback loop and form the basis for analyzing and adapting the system.

- **Analyze**: The monitored data and the current system state are analyzed in order to detect possible states of over- or under-provisioning and to check if process instances are still within their defined SLA constraints or whether violations have to be expected.

- **Plan**: Based on the monitored and analyzed system state and environmental influences multiple planning steps have to be accomplished. First, the system needs to plan the sequence and timing of execution for single process steps in order to meet process deadlines and make the best possible use of the available infrastructure. Second, the amount and type of Docker containers that isolate single service instances for executing different service invocations have to be calculated. Third, the amount and type of VM instances that provide computational resources have to be planned. After the planning phase, the system has to calculate a task-scheduling and resource allocation plan that provides details about the needed amount of computational resources including necessary leasing times, as well as the decisions on which VM to place which Docker containers and on which container instances to schedule the single planned service invocations.

- **Execute**: The calculated task-scheduling and resource allocation plan has to be executed by the BPMS. This especially means that different types of VM instances have to be leased or released and Docker containers with corresponding software services have to be placed on the available resources according to the plan. Once the resources are set up, the scheduled tasks can be executed on them.

- **Knowledge Base**: The information about the systems configuration and current state is stored in a knowledge base and can be requested throughout the feedback cycle. Such information includes, for example, the running, queued, and finished process instances, the current state of execution for each process instance and its process steps, the containers on which the process instances are deployed, the currently running VM instances and their leasing time and state, the information which container instance is deployed on which VM instance, or which container types have ever been deployed on which individual VM instances.

4.2 Basic Approach and System Model

Our approach for the enactment of complex elastic processes aims at finding an optimized execution plan for task-scheduling, i.e. scheduling of service invocations for realizing elastic processes, and resource allocation, i.e. leasing and releasing VM instances and assigning resources to Docker containers that run software services for executing scheduled tasks.

4.2.1 Basic Approach

Similar to the related work of Hoenish et al. [65] on resource allocation of single software services, our system for enacting elastic processes follows a four-fold auto-scaling approach. This means that our system considers scaling over four dimensions when allocating resources. Both VMs and containers are adjusted horizontally and vertically.

- **Horizontal scaling**:
  - **VMs**: Horizontal scaling of VMs is accomplished by changing the number of leased VM instances of the system.
  - **Containers**: Horizontal scaling of containers is realized by changing the number of deployed container instances offering a certain software service.
- **Vertical scaling**:
  - **VMs**: Vertical scaling of VMs refers to the change of computational resources of a system, while the number of deployed VM instances remains unchanged. It has to be noted that vertical scaling in this context does not refer to single VM instances, but to the system landscape as a whole, as we do not address problems related to reassigning computational resources to running VM instances, but allow to exchange running VM instances by VM instances of different types offering different resources.
  - **Containers**: Vertical scaling of containers is realized by resizing deployed container instances so that they offer more or less computational resources of the underlying VM instance. The resizing is achieved by changing the assigned CPU-shares to container instances.

The main goal is to minimize the cost of the system's process enactments, by considering costs for leasing computational resources and penalty costs that occur when QoS constraints are violated. The system needs to optimize its main tasks related to the execution of process requests with respect to these goals. The system's main tasks are:

- leasing and releasing of VM instances.
- placement of Docker containers that run software services for executing process steps on the leased VM instances.
- dynamic assignment and reassignment of minimum guaranteed computational resources to Docker containers.
- scheduling of service invocations on the running containers.

4.2.2 System Model

In order to explain our approach in more detail, we define the used system model for the elastic process landscape.

Process Variables

Our approach considers multiple process models consisting of multiple different services, which can be shared across the process models, while each service invocation requires a specific amount of resources in terms of CPU and RAM. We allow the definition of complex process models, consisting of sequences, XOR-blocks, AND-blocks, and repeated loop blocks. Simple sequential processes generally have only one trivial next step. AND-constructs allow the parallel invocation of multiple next steps from multiple branches. XOR-constructs require the exclusive invocation of the next step of one branch, which our system will pick randomly. Repeat loop blocks can be executed repeatedly. Our system will pick the number of repetitions randomly assuming a maximum amount of possible repetitions. Our optimization approach will consider a worst-case analysis when considering future XOR-constructs and repeat loop blocks, meaning it will assume the longest path. Table A.1 summarizes the used variables for describing processes.

Table 4.1: System Model: Process Variables

Variable Name	Description
$p \in P = \{1, \ldots, p^{\#}\}$	The set of process models is represented by P, while p indicates a specific process model.
$i_p \in I_p = \left\{1, \ldots, i_p^{\#}\right\}$	The set of process instances of a specific process model p is represented by I_p, while i_p indicates a specific process instance of the process model p.
$j_{i_p} \in J_{i_p} = \left\{1, \ldots, j_{i_p}^{\#}\right\}$	The set of process steps not yet executed (and not yet started) to realize a specific process instance i_p is represented by J_{i_p}, while j_{i_p} refers to a specific process step of the process instance i_p.
$j_{i_p}^{*} \in J_{i_p}^{*} \subseteq J_{i_p}$	The set of next process steps that have to be scheduled for execution to realize a specific process instance i_p is represented by $J_{i_p}^{*}$, while $j_{i_p}^{*}$ refers to a specific next process step of the process instance i_p. Note: AND-Constructs can execute multiple paths in parallel and can, therefore, have multiple next process steps in the set $J_{i_p}^{*}$.
$j_{i_p}^{run} \in J_{i_p}^{run} \not\subset J_{i_p}$	The set of currently running and not finished process steps of a specific process instance i_p is represented by $J_{i_p}^{run}$, while $j_{i_p}^{run}$ refers to a specific currently running process step of the process instance i_p.

Continued on next page –>

Table 4.1: System Model: Process Variables

	Note: AND-Constructs can execute multiple paths in parallel and can, therefore, have multiple currently running process steps in the set $J_{i_p}^{run}$.
DL_{i_p}	The deadline for the enactment of the process instance i_p, is defined by DL_{i_p} and refers to a certain point in time.
$DL_{j_{i_p}^*}$	The deadline for starting the enactment of a next process step $j_{i_p}^*$ such that it still meets its process deadline DL_{i_p}, while performing a worst-case analysis considering all subsequent steps is defined by $DL_{j_{i_p}^*}$ and refers to a certain point in time.
ω_{DL}	The weighting factor for controlling the effect of deadline constraints is indicated by ω_{DL}.
$r_{j_{i_p}}^C, r_{j_{i_p}}^R$	The resource demands of a process step j of the process instance i_p in terms of CPU ($r_{j_{i_p}}^C$) and RAM ($r_{j_{i_p}}^R$) are indicated by the respective terms.

Optimization Time Variables

It is important to mention that the optimization approach refers to a certain time period τ_t and the parameter t is used to describe the point in time that marks the beginning of a time period. Table A.2 summarizes the used optimization time variables.

Table 4.2: System Model: Optimization Time Variables

Variable Name	**Description**
$t, t_{j_{i_p}}$	A specific point in time is specified by t. The point in time at which a specific service invocation j_{i_p} was scheduled is referred to as $t_{j_{i_p}}$
τ_t, τ_{t+1}	The current time period starting at a specific point in time t is indicated by τ_t. τ_{t+1} refers to the next time period starting at $t + 1$. At the beginning of each time period the optimization model is calculated based on all currently available information and its results are enacted.
ϵ	The minimum time period that is at least needed between two optimization runs is defined by ϵ in milliseconds.

Virtual Machines

Our approach accounts for different types of VMs from multiple cloud providers with varying resources, configurations, and pricing models. A VM may be leased for multiple BTUs while the length of a BTU may vary for each VM type. Table A.3 summarizes the used variables for describing VMs.

Table 4.3: System Model: Virtual Machines

Variable Name	Description
$v \in V = \{1, \ldots, v^{\#}\}$	The set of VM types is represented by V, while v indicates a specific VM type.
$k_v \in K_v = \left\{1, \ldots, k_v^{\#}\right\}$	The set of leasable VM instances of type v is represented by K_v, while k_v is the k^{th} VM instance of type v. Note: For a time period starting at t, we assume the number of leasable VMs of type v to be limited with $k_v^{\#}$.
BTU_v	The BTU of a VM instance k_v of VM type v is indicated by BTU_v. A BTU is a billing cycle and the minimum leasing duration for a VM instance k_v. A VM instance may be leased for multiple BTUs in a row, but releasing a VM before the end of a BTU leads to a waste of paid resources.
BTU^{max}	An arbitrary large upper bound of possible leasable BTUs for any VM instance k_v is expressed by the helper variable BTU^{max}.
c_v	This variable represents the leasing cost for VM instances of type v for one full BTU.
$\gamma_{(v,t)} \in \mathbb{N}_0^+$	This variable indicates the total number of BTUs to lease any VMs of type v in the time period τ_t.
$\beta_{(k_v,t)} \in \{0, 1\}$	This variable indicates if the VM instance k_v is/was already running at the beginning of the time period τ_t.
$g_{(k_v,t)} \in \{0, 1\}$	This variable indicates if the VM instance k_v was already running at the beginning of time period τ_t or is being leased during τ_t.
$d_{(k_v,t)}$	This variable indicates the remaining leasing duration of the VM instance k_v at the beginning of the time period τ_t.
ω_d	The weighting factor for controlling the effect of the remaining leasing duration on the optimization outcome is indicated by ω_d.
s_v^C, s_v^R	The resource supplies of a VM type v in terms of CPU (s_v^C) and RAM (s_v^R) are indicated by the respective terms.
$f_{k_v}^C, f_{k_v}^R$	The free resources in terms of CPU ($f_{k_v}^C$) and RAM ($f_{k_v}^R$) of a VM instance k_v are indicated by the respective terms.
ω_f^C, ω_f^R	Weighting factors for controlling the effect of free resources in terms of CPU (ω_f^C) and RAM (ω_f^R) on the optimization outcome are indicated by the respective terms.

Containers and Services

Our approach deploys single services that are part of processes into containers. Therefore, we consider multiple container types, each representing a single service. Each container instance running a specific software service can have different configurations in terms of guaranteed minimum resources. Table A.5 summarizes the used variables for describing containers.

Table 4.4: System Model: Containers and Services

Variable Name	Description
$st \in ST = \{1, \ldots, st^{\#}\}$	The set of all container types mapping all possible service types is represented by ST, while st indicates a specific container type for a certain service type.
st_j	The service type of a certain process step j is indicated by st_j.
$c_{st} \in C_{st} = \left\{1, \ldots, c_{st}^{\#}\right\}$	The set of container instances mapping specific service instances of type st is represented by C_{st}, while c_{st} is the c^{th} container instance of type st.
c_{st_j}	A container instance of a certain service type st that can run a process step j is indicated by c_{st_j}.
$z_{(st,k_v,t)} \in \{0,1\}$	This variable indicates if any containers of service type st, i.e., any $c_{st} \in C_{st}$ have already been deployed on a VM instance k_v (and hence the container image for starting new container instances of service type st has been downloaded on k_v) until the time period τ_t.
ω_z	The weighting factor for controlling the effect of existing images on VM instances for the optimization outcome is indicated by ω_z.
$s_{c_{st}}^C, s_{c_{st}}^R$	The guaranteed minimum resource supplies of a container instance c_{st} in terms of CPU ($s_{c_{st}}^C$) and RAM ($s_{c_{st}}^R$) are indicated by the respective terms.
ω_s^C, ω_s^R	Weighting factors for controlling the effect of resource supply of container instances in terms of CPU (ω_s^C) and RAM (ω_s^R) on the optimization outcome are indicated by the respective terms.
M	An arbitrary large upper bound of possible container deployments for any VM instance k_v is expressed by the helper variable M.
N	An arbitrary large upper bound of possible service invocations for any container instance c_{st} is expressed by the helper variable N.

Startup, Deployment, Execution, and Penalty Times

Our approach considers the execution times of single process steps for calculating execution times for process instances. Furthermore, the startup times for VMs, the time for pulling new Docker images onto running VM instances, and the deployment times for starting Docker containers from Docker images are considered. Our approach is based on a worst-case assumption, meaning for calculating remaining execution times and scheduling plans the system will assume the worst-case of having to lease a new VM instance, pull the needed image onto the instance and deploy a new container for each remaining step. Table A.6 summarizes the used variables for describing execution times.

Table 4.5: System Model: Startup, Deployment, Execution, and Penalty Times

Variable Name	Description
Δ_v, Δ	The time in milliseconds to start a new VM of type v is indicated by Δ_v. The maximum VM-startup time of any type, i.e. $max_{v \in V}(\Delta_v)$ is expressed by Δ.
Δ_{st}	The time for pulling a container image for a service type st on a VM instance for the first time is expressed by Δ_{st}.
$\Delta_{c_{st}}$	The time for starting a container instance from an existing image for a service type st on a VM instance is expressed by $\Delta_{c_{st}}$.
e_{i_p}	The remaining execution duration of a process instance i_p is expressed by e_{i_p}. It does not include the execution and deployment time of the process steps that will be scheduled for the current optimization period τ_t, nor the remaining execution time of the currently already running steps.
$l \in L = \{1 \dots l^{\#}\}$,	The set of all paths is represented by L, while l indicates a specific path within a process.
$L_a \cup L_x \cup RL \subseteq L$	The set of possible paths for AND-blocks L_a, XOR-blocks L_x and repeat loop constructs RL are expressed by the respective terms.
re	The maximum repetitions of a repeat loop are indicated by re.
$e_{i_p}^{seq}, e_{i_p}^{L_a}, e_{i_p}^{L_x}, e_{i_p}^{RL}$	The remaining execution duration of sequences ($e_{i_p}^{seq}$), AND-blocks ($e_{i_p}^{L_a}$), XOR-blocks ($e_{i_p}^{L_x}$) and repeat loop constructs ($e_{i_p}^{RL}$) of the process instance i_p are expressed by the respective terms. The terms do not include the remaining execution time of the currently running steps nor the execution and deployment times for process steps that will be scheduled for the current optimization period τ_t.

Continued on next page –>

Table 4.5: System Model: Startup, Deployment, Execution, and Penalty Times

$\hat{e}^s_{i_p}, \hat{e}^l_{i_p}$	The remaining combined service execution duration, time for pulling needed container images, container deployment and VM startup time for all not yet scheduled service invocations of sequences or repeat loop blocks ($\hat{e}^s_{i_p}$) and paths in AND- or XOR-blocks ($\hat{e}^l_{i_p}$) that still need to be invoked to finalize the enactment of the process instance i_p are expressed by the respective terms. Those values do not include the remaining execution time of already running process steps $j^{run}_{i_p}$ or process steps that are being scheduled in the current optimization period τ_t.
$ex_{j^*_{i_p}}$	The combined execution duration, container deployment and VM-startup time of the next to be scheduled process step $j^*_{i_p}$ is expressed by $ex_{j^*_{i_p}}$.
$ex_{j^{run}_{i_p}}$	The remaining execution time of process steps already scheduled in previous periods and not finished until the start of τ_t is expressed by $ex_{j^{run}_{i_p}}$.
$e^p_{i_p}$	The penalty time units of a process instance i_p, i.e., the execution duration of i_p that occurs beyond the deadline DL_{i_p} of i_p, is expressed by $e^p_{i_p}$.
$c^p_{i_p}$	The penalty cost per time unit of delay of the process instance i_p is represented by $c^p_{i_p}$.

Important Decision Variables

The designed system should calculate a solution that answers the questions of how many VM instances of what VM types to lease, for how many BTUs to lease the VM instances, what Docker containers of which type and with which configuration to deploy on each leased VM instance, and what process steps to invoke on which deployed container. Table A.4 summarizes the used decision variables.

Table 4.6: System Model: Decision Variables

Variable Name	**Description**
$y_{(k_v,t)} \in \{\mathbb{N}^+_0\}$	This variable indicates how often (i.e., for how many BTUs) the VM instance k_v should be leased in the time period τ_t.
$a_{(c_{st},k_v,t)} \in \{0,1\}$	This variable indicates if the container c_{st} with the service type st should be deployed on VM k_v in the time period τ_t.
$x_{(j_{i_p},c_{st},t)} \in \{0,1\}$	This variable indicates if process step j of process instance i_p should be invoked on container c_{st} in the time period τ_t.
$x_{(j_{i_p},k_v,t)} \in \{0,1\}$	This variable indicates if process step j of process instance i_p should be invoked on VM k_v in the time period τ_t.

4.3 Working Example

To illustrate the most important features explained in the preliminaries and to aid the explanation of our presented solution approaches, we introduce an example scenario that presents the components that have to be considered when optimizing container-based system landscapes for elastic processes in more detail.

This working example is based on the example scenario presented in Section 1.2, but only considers a very simplified system and process landscape for illustration purposes. Looking at Figure 1.1, we can see the general structure of a container-based elastic process landscape. Our international car manufacturer can request the execution of different business processes from multiple sites and locations simultaneously. The process requests can have different SLAs and are sent to a BPMS that handles the execution of elastic processes. The BPMS can execute the incoming process requests on a container-based cloud infrastructure. Therefore, the BPMS has to calculate the resource demands, lease and release computational resources, allocate containers and schedule the process steps considering process deadlines while minimizing the costs of the leased resources.

In the following, we discuss the main functions and responsibilities of the BPMS in more detail by using the simplified business processes, service and container types, available computational resources in the form of VMs, and incoming request patterns as described in the subsections below. In the following Section 4.4, we will explain our solution approach for solving the working example and in Section 4.5 we will illustrate the application of our presented approach on the working example discussed in this section.

4.3.1 Considered Business Processes

Of the various business processes that the car manufacturer has employed across the multiple sites and locations, in the following we consider the simplified and generalized business process models $p = 1, 2, 3$, and N ($\{1, 2, 3, N\} \in P$) as depicted in Figure 4.2, which correspond to the business processes already used for our illustration in Section 1.2.

Each business process can be requested by different users simultaneously, leading to different process instances ($i_p \in I_p$), while each user may define different SLAs for each business process instance. The SLO we consider is the process deadline (DL_{i_p}) which indicates the time until the execution of an incoming process instance (i_p) has to be finished. The BPMS must schedule all incoming process requests based on their importance. As can be seen in Figure 4.2, multiple process models share process steps of the same service types ($st \in ST$). Therefore the BPMS can schedule the individual process steps (j_{i_p}) of different process instances based on different process models on the same corresponding isolated software services (c_{st}) in the container-based elastic cloud infrastructure. When calculating an optimized scheduling, the BPMS has to differentiate between long-running and short-running processes, and processes with strict SLAs that have to be scheduled as soon as possible and processes with more loose SLA requirements that can also be postponed into the future if there currently are not enough computational resources available and an immediate execution would lead to preventable additional leasing costs for the service provider.

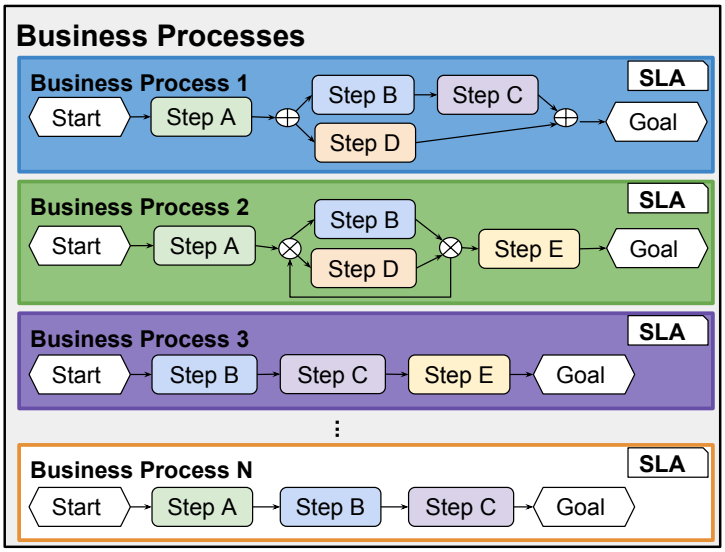


Figure 4.2: Simplified Business Processes of the Car Manufacturer

4.3.2 Service Types and Container Types

The depicted process models in Figure 4.2 are composed of individual loosely coupled process steps of certain service types ($\{A, B, C, D, E\} \in ST$) following the principles of SOA. Steps of the same service type st but of different process models and different process instances can share the same software service instances c_{st}. The functionalities of the process steps are implemented by the following software services of corresponding service types st as illustrated in Table 4.7.

Table 4.7: Process Steps of the Example System

Service Type	CPU Requirement	RAM Requirement	Makespan
Service Type A	50 CPU points	270 MB	30 sec.
Service Type B	30 CPU points	100 MB	120 sec.
Service Type C	120 CPU points	900 MB	60 sec.
Service Type D	70 CPU points	1024 MB	90 sec.
Service Type E	80 CPU points	630 MB	150 sec.

The instances of each service type have certain CPU requirements ($r^C_{j_{ip}}$) which we calculate as CPU points. CPU points can be interpreted as the percentage of one single CPU core that one service invocation of a service instance of the service type requires. This means that service invocations of service type A require half of the resources of one CPU core while service invocations of service type D require 1.2 CPU cores for a successful execution. Each service type also has different RAM requirements ($r^R_{j_{ip}}$) which we consider in our approach but will neglect in this example for simplifying the problem. Furthermore, we have to consider the makespan of each service invocation which indicates the time required to execute a corresponding process step.

The BPMS has to schedule the incoming requests on software containers c_{st} that are placed on VM instances k_v which offer the required computational resources (s_v^C, s_v^R). Each container c_{st} includes the program code of a certain software service st, therefore, a container can only execute service invocations of one specific service type. The process steps j_{i_p} of multiple process instances can share one container instance c_{st} and its software service as long as the container offers sufficient resources $(s_{c_{st}}^C, s_{c_{st}}^R)$. While multiple containers for the same service type c_{st} can be deployed on different VM instances k_v, and every VM instance may host multiple containers, one VM instance can only run containers of distinct service types st.

4.3.3 VM Types

The containers c_{st} that offer computational resources for service invocations have to be placed on VM instances k_v. In our ongoing example, our system can lease and release VM instances of the following VM types v as shown in Table 4.8.

Table 4.8: VM Types of the Example System

VM Type	CPU	RAM	Leased from	Price per BTU	BTU
VM Type 1	single core	1024 MB	public cloud A	8	5 min
VM Type 2	dual core	2048 MB	public cloud A	13	5 min
VM Type 3	quad core	4096 MB	public cloud A	18	5 min
VM Type 4	single core	1024 MB	private cloud B	5	5 min
VM Type 5	dual core	2048 MB	private cloud B	10	5 min
VM Type 6	quad core	4096 MB	private cloud B	15	5 min

The service provider can choose from a set of public and private cloud resources when leasing new VM instances k_v. Each VM type offers different types of computational resources in terms of CPU (s_v^C), RAM (s_v^R) and location. In our example, we assume only one private cloud and only one public cloud provider but generally, a multitude of cloud providers with different pricing models could be utilized by the service provider. The BTU (btu_v) indicates the amount of time for which a VM instance k_v of a certain VM type v can be leased for the given price. A service provider may lease a VM for multiple BTUs in a row or extend the lease of a VM instance while it is leased for additional BTUs. When releasing a VM instance after the last leased BTU, the VM is shut down and its state is discarded. To simplify the example we assume a general BTU of five minutes for each leasable VM instance but generally infrastructure providers could easily also define different BTUs for their resources.

Although virtually unlimited, we assume the number of leasable VM instances k_v of a certain VM type v to be limited in a certain time period. For this simple example, we assume that the service provider is allowed to lease only one VM instance of each specified private cloud VM type ($v \in \{4, 5, 6\}$) and up to two VM instances of the specified public cloud VM types ($v \in \{1, 2, 3\}$). The two public cloud VM instances of VM type 1 are denoted in the following as VM1.1 and VM1.2 while the VM instance from the private cloud of Type 4 is referenced to as VM4.1. We use the same naming convention for the remaining leasable VM instances.

4.3.4 Incoming Process Requests

Process requests i_p can be sent anytime and concurrently from multiple users to the BPMS. In this working example, we consider the following process requests as depicted in Table 4.9.

Table 4.9: Incoming Process Requests

Process Instance Name	Business Process	Time Received	Deadline	Penalty
P1.1	1	t = 0	t = 11	2
P3.1	3	t = 0	t = 100	2
PN.1	N	t = 0	t = 11	2
P3.2	3	t = 5.5	t = 18.5	2
PN.2	N	t = 5.5	t = 16.5	2

In the following, we reference the single service invocations of each process instance as [Process Instance Name].[Service Type Name], so the first step of process instance P1.1 is for example referred to as P1.1.A.

The indicated timestamps represent minutes. As an example, between the timestamps $t = 0$ and $t = 1$ one minute has passed, while between the timestamp $t = 0$ and $t = 1.5$ 90 seconds have passed.

Furthermore, the BPMS needs to consider penalty costs for every time unit that the execution of process requests exceeds the defined deadline. Penalty costs can be individually defined for each process instance. In this simplified example, we assume linear penalty costs of 2 units for each time unit that the execution exceeds the defined deadline of any process instance. In general, it is easily possilbe to define different penalty costs for each process instance.

Our approach aims at finding an optimized resource allocation and scheduling plan for the execution of the incoming process requests. Due to increased requests over time, the auto scaling system will have to increase the number of provisioned resources for certain applications and also deprovision resources when the number of requests decreases. In the following Section 4.4 we present our solution approach for solving the problem outlined in this section. Afterward, we will demonstrate the execution of this ongoing example using the presented solution approach in Section 4.5. Later, in Section 6.1 we evaluate the working example, in Section 6.2 we discuss some important observations and in Section 6.3 we discuss the main advantages compared to previous work using only VMs.

4.4 Multi-Objective Problem Formulation using MILP

In this section, we formulate our task-scheduling and resource allocation problem as a MILP optimization problem. The multi objective optimization problem represents the scheduling problem for one optimization round starting at time t . An optimization round is triggered whenever new process steps can be scheduled, for example, whenever new process requests arrive or when previously running steps of process instances are finished. Furthermore the optimization is triggered when SLAs violations can be foreseen.

Our optimization problem takes complex process patterns, penalty costs, container-based metrics, different pricing models, and BTUs for VMs into consideration. The optimization problem focuses on minimizing the costs that arise from enacting elastic process landscapes on container-based cloud infrastructures. Solutions for the problem describe when and on which containers and service instances to schedule service invocations, on which VMs to place containers and service instances, and which VMs to lease or release.

4.4.1 Four Fold Service Instance Placement Problem

As discussed in Subsection 4.2.1 we design a system that facilitates the scaling over four dimensions and name it the Four Fold Service Instance Placement Problem (FFSIPP).

Objective Function

In 4.1 we present the objective function which is subject to minimization under the subsequently following constraints.

$$
\begin{aligned}
min & \sum_{v \in V} \left(c_v * \gamma_{(v,t)} \right) \\
& + \sum_{p \in P} \sum_{i_p \in I_p} \left(c^p_{i_p} * e^p_{i_p} \right) \\
& + \sum_{st \in ST} \sum_{c_{st} \in C_{st}} \sum_{v \in V} \sum_{k_v \in K_v} \left(\omega_z * (1 - z_{(st,k_v,t)}) * a_{(c_{st},k_v,t)} \right) \\
& + \sum_{st \in ST} \sum_{c_{st} \in C_{st}} \sum_{v \in V} \sum_{k_v \in K_v} \left(\omega_d * d_{(k_v,t)} * a_{(c_{st},k_v,t)} \right) \\
& + \sum_{v \in V} \sum_{k_v \in K_v} \left(\omega^C_f * f^C_{k_v} + \omega^R_f * f^R_{k_v} \right) \\
& + \sum_{st \in ST} \sum_{c_{st} \in C_{st}} \sum_{v \in V} \sum_{k_v \in K_v} \left((\omega^C_s * s^C_{c_{st}} + \omega^R_s * s^R_{c_{st}}) * a_{(c_{st},k_v,t)} \right) \\
& + \sum_{p \in P} \sum_{i_p \in I_p} \sum_{j^*_{i_p} \in J_{i_p}} \sum_{c_{st_j} \in C_{st}} \left(\omega_{DL} * (DL_{j^*_{i_p}} - \tau_t) * x_{(j_{i_p},c_{st_j},t)} \right)
\end{aligned}
\tag{4.1}
$$

The objective function considers seven terms for calculating a scheduling plan, indicated by the decision variable $x_{(j_{i_p},c_{st_j},t)}$, which decides on which container instance to schedule a service invocation and the decision variable $a_{(c_{st},k_v,t)}$, which decides on which VM instance to deploy a container instance. In the following we explain each term:

- **Term 1** - *minimize total VM leasing costs:* The first term, i.e., $\sum_{v \in V} \left(c_v * \gamma_{(v,t)}\right)$, calculates the total costs for leasing VM instances of each VM type v at time t, by considering the leasing costs c_v for the VM types and the total number of leased BTUs $\gamma_{(v,t)}$ that VM instances of the corresponding type v are leased at t. The aim is to minimize the total costs for leasing VMs.

- **Term 2** - *minimize penalty costs:* The second term, i.e., $\sum_{p \in P} \sum_{i_p \in I_p} \left(c^p_{i_p} * e^p_{i_p}\right)$, calculates the total penalty costs that arise from scheduling decisions that lead to execution times for process instances beyond their defined deadlines in time t or in the future. The penalty time $e^p_{i_p}$ of a process instance i_p is calculated according to a worst-case analysis. The aim is to minimize such penalty costs and therefore make sure that incoming process requests are executed in time, as defined in their SLAs.

- **Term 3** - *minimize container deployment time:* The third term minimizes the sum of time needed for deploying all newly scheduled containers on VM instances k_v. This is achieved by giving a preference to the deployment of container instances on VM instances on which the image for the container type is cached locally. The term $(1 - z_{(st,k_v,t)})$ makes sure that the weighting factor ω_z is only considered when the cache for container deployment does not exist on the VM instance k_v.

- **Term 4** - *maximize future resource supply of already leased VMs:* The fourth term gives a preference to deploying containers for service invocations on VMs that have a shorter remaining leasing duration $d_{(k_v,t)}$ compared to other VMs with already existing longer leasing durations. Although at time t this preference does not lead to any cost-based advantages, it assures that the already leased resources offer more long-running resources for future optimization periods. The term calculates the remaining leasing duration of all VMs using a weighting factor. As we want to give a preference to deploying containers on VM instances with a small leasing duration, we minimize the term to maximize the future remaining leasing durations of available resources.

- **Term 5** - *minimize sum of unused present VM resources:* The fifth term minimizes the sum of all leased but unused computational resource capacities in terms of CPU ($f^C_{k_v}$) and RAM ($f^R_{k_v}$) of all leased VM instances. Due to this term, our approach makes sure to not only schedule all service invocations that can not be delayed any further without leading to penalty costs but to also pre-schedule service invocations that may already be executed but have a distant deadline. We consider this term using weighting factors for CPU and RAM resources, such that we give more importance on the first two terms of our objective function.

- **Term 6** - *minimize sum of unused container resources:* The sixth term makes sure that resources in terms of CPU ($s^C_{c_{st}}$) and RAM ($s^R_{c_{st}}$) supplied by containers for scheduling service invocations are minimized, such that containers do not reserve more resources than they actually need for guaranteeing the successful execution of scheduled service instances. Therefore, the term reduces the risk of overprovisioning by demanding to lease the smallest resources to containers that still fulfill the scheduling demand. Similar to term 5, we use weighting factors to limit the influence of this term to the overall objective function.

- **Term 7** - *maximize the importance of scheduled service invocations:* The last term calculates the difference between the last possible scheduling deadline of the next schedulable steps when performing a worst-case analysis based on the current time. The term considers all process instances and aims at giving a scheduling preference to process steps with closer deadlines. The term is responsible for defining the importance of schedulable process steps that do not necessarily have a pressing process deadline on remaining free resources. Note that in contrast to the related work [63], we do consider the actual enactment deadline for the next schedulable steps based on the worst-case analysis instead of simply comparing the overall process deadlines. We also define the term such that past process deadlines can be considered and steps that are past their deadline receive a higher scheduling importance. We consider a weighting factor ω_{DL} to assign a lower weight to the term.

Constraints

The objective function is subject to minimization under the following constraints:

Constraint 4.2 makes sure that deadlines are considered for each process instance i_p. The constraint does not guarantee that deadlines are not violated, but it defines the penalty times $e^p_{i_p}$ of process instances which are subject to minimization in the presented term 2 of the objective function. The constraint demands that the current time plus the remaining worst-case execution time of process instances is smaller or equal to the defined deadline plus the potential penalty time which occurs when process instances finish after their deadline. This constraint considers the remaining execution time including the time of process steps that are being scheduled during the current time period and process steps that have already been scheduled in previous periods and are still running.

$$\tau_t + ex^{run}_{j_{i_p}} + ex^{*}_{j_{i_p}} + e_{i_p} \leqslant DL_{i_p} + e^p_{i_p} \tag{4.2}$$

Constraint 4.3 helps in defining the start of the next optimization period τ_{t+1}, by demanding that the start of the next optimization period τ_{t+1} plus the remaining worst-case execution time of process instances is smaller or equal to the respectively defined process deadlines plus the possible penalty times. It should be noted that in contrast to related work [63] our approach only considers the worst-case execution time of all process steps that have not been scheduled yet

until τ_t and have to be scheduled in a later optimization period τ_{t+1} or later for the calculation of the start time of the next optimization period τ_{t+1}. At this stage, the approach especially does not consider the execution time of currently running process steps or process steps that are scheduled for execution in τ_t, as the earliest possible time when next steps of a process instance can be scheduled is when the previous steps, already running or scheduled in τ_t, are finished. Adding the remaining time of the currently running or scheduled steps to the left side of Equation 4.3 can potentially lead to triggering premature optimization rounds at a $t+1$ where predecessor steps of the next schedulable steps are not finished yet (and hence no new steps can be scheduled).

Constraint 4.4 makes sure that the next optimization time period τ_{t+1} is defined to be in the future. We use a value $\epsilon > 0$ to avoid optimization deadlocks arising from a too small value for τ_{t+1}.

The start of the next optimization period is directly calculated as a result of the previous optimization rounds. Nevertheless, as this optimization model only refers to one optimization round and only operates based on the current knowledge, the optimization can also be triggered by other events, like newly incoming requests or finalized process steps and run earlier than the here calculated time $t + 1$.

$$\tau_{t+1} + e_{i_p} \leqslant DL_{i_p} + e^p_{i_p} \tag{4.3}$$

$$\tau_{t+1} \geqslant \tau_t + \epsilon \tag{4.4}$$

Equation 4.5 calculates the remaining execution duration for all process instances i_p. The remaining execution duration e_{i_p} does not include the execution and deployment time of the process steps that will be scheduled for the current optimization period τ_t, nor the remaining execution time of the currently already running steps.

$$e_{i_p} = e^{seq}_{i_p} + e^{L_a}_{i_p} + e^{L_x}_{i_p} + e^{RL}_{i_p} \tag{4.5}$$

The overall remaining execution duration e_{i_p} of a process instance i_p is the sum of the execution times of the different process patterns that the process instance is composed of. We aggregate the upper bound execution times of the different process patterns, following the approach by Jäger et al. [75]. The worst-case execution times for sequences are defined in Equation 4.6, for AND-blocks in Equation 4.7, for XOR-blocks in Equation 4.8 and for Repeat Loop blocks in Equation 4.9. For AND-blocks the longest remaining execution time has to be considered naturally. As we are using a worst-case analysis, we also consider the longest path for the calculation of the remaining execution time of XOR-blocks. For Repeat Loop blocks the sub path execution time is multiplied by the maximal possible amount of repetitions re.

Equations 4.6 - 4.9 consider the overall remaining execution time of all not yet running process steps for each process instance i_p as defined by the two helper variables $\hat{e}^s_{i_p}$ for sequences

and $\hat{e}^{l}_{i_p}$ for XOR- and AND-blocks in Equations 4.11 and 4.12. Since we perform a worst-case analysis, both equations consider the worst-case VM startup time, the time for pulling the respective Docker image onto the running VM instance and the time for starting a new container instance for each remaining process step j_{i_p}. If a process step $j^{*}_{i_p}$ is being scheduled in the current time period τ_t, Equations 4.6 - 4.9 subtract the worst-case execution time of the scheduled steps as defined in Equation 4.10 from the overall remaining execution time.

$$e^{seq}_{i_p} = \begin{cases} \hat{e}^{s}_{i_p} - ex_{j^{*}_{i_p}} & \text{, if } x_{(j^{*}_{i_p}, c_{st_j}, t)} = 1 \\ \hat{e}^{s}_{i_p} & \text{, otherwise} \end{cases} \tag{4.6}$$

$$e^{L_a}_{i_p} = \begin{cases} \max\limits_{l \in La}(\hat{e}^{l}_{i_p} - ex_{j^{*}_{i_p}}) & \text{, if } x_{(j^{*}_{i_p}, c_{st_j}, t)} = 1 \\ \max\limits_{l \in La}(\hat{e}^{l}_{i_p}) & \text{, otherwise} \end{cases} \tag{4.7}$$

$$e^{L_x}_{i_p} = \begin{cases} \max\limits_{l \in Lx}(\hat{e}^{l}_{i_p} - ex_{j^{*}_{i_p}}) & \text{, if } x_{(j^{*}_{i_p}, c_{st_j}, t)} = 1 \\ \max\limits_{l \in Lx}(\hat{e}^{l}_{i_p}) & \text{, otherwise} \end{cases} \tag{4.8}$$

$$e^{RL}_{i_p} = \begin{cases} re * \hat{e}^{s}_{i_p} - ex_{j^{*}_{i_p}} & \text{, if } x_{(j^{*}_{i_p}, c_{st_j}, t)} = 1 \\ re * \hat{e}^{s}_{i_p} & \text{, otherwise} \end{cases} \tag{4.9}$$

$$ex_{j^{*}_{i_p}} = \sum_{c_{st_j} \in C_{st_j}} \left((e_{j^{*}_{i_p}} + \Delta_{c_{st_j}} + \Delta_{st_j} + \Delta) * x_{(j^{*}_{i_p}, c_{st_j}, t)} \right) \tag{4.10}$$

$$\hat{e}^{s}_{i_p} = \sum_{j_{i_p} \in J^{seq}_{i_p}} (e_{j_{i_p}} + \Delta_{c_{st_j}} + \Delta_{st_j} + \Delta) \tag{4.11}$$

$$\hat{e}^{l}_{i_p} = \sum_{j_{i_p} \in J^{l}_{i_p}} (e_{j_{i_p}} + \Delta_{c_{st_j}} + \Delta_{st_j} + \Delta) \tag{4.12}$$

Constraints 4.13 and 4.14 make sure that for all $st \in ST, c_{st} \in C_{st}$, the sum of CPU and RAM resources required by all the service invocations that either already run or are scheduled to run on the container instance c_{st} in τ_t, do not exceed the containers' guaranteed minimum resource supply (in terms of CPU and RAM).

$$\sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in (J^{*}_{i_p} \cup J^{run}_{i_p})} (r^{C}_{j_{i_p}} * x_{(j_{i_p}, c_{st_j}, t)}) \leqslant s^{C}_{c_{st_j}} \tag{4.13}$$

$$\sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in (J^*_{i_p} \cup J^{run}_{i_p})} (r^R_{j_{i_p}} * x_{(j_{i_p}, c_{st_j}, t)}) \leqslant s^R_{c_{st_j}} \tag{4.14}$$

Constraints 4.15 and 4.16 make sure that for all $v \in V, k_v \in K_v$, the sum of CPU and RAM resources required by all containers that run or have to be deployed on a VM instance k_v in τ_t, do not exceed the capacity of a VM of type v.

$$\sum_{st \in ST} \sum_{c_{st} \in C_{st}} (s^C_{c_{st}} * a_{(c_{st}, k_v, t)}) \leqslant s^C_v \tag{4.15}$$

$$\sum_{st \in ST} \sum_{c_{st} \in C_{st}} (s^R_{c_{st}} * a_{(c_{st}, k_v, t)}) \leqslant s^R_v \tag{4.16}$$

Constraints 4.17 and 4.18 demand that for all service types st the sum of all supplied resources by all container instances c_{st} of a certain service type st is greater or equal to the sum of the total resource demand (in terms of CPU and RAM) of all service invocations of the specific service type st.

$$\sum_{v \in V} \sum_{k_v \in K_v} (s^C_{c_{st}} * a_{(c_{st}, k_v, t)}) \geqslant \sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in (J^*_{i_p} \cup J^{run}_{i_p})} (r^C_{j_{i_p}} * x_{(j_{i_p}, c_{st_j}, t)}) \tag{4.17}$$

$$\sum_{v \in V} \sum_{k_v \in K_v} (s^R_{c_{st}} * a_{(c_{st}, k_v, t)}) \geqslant \sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in (J^*_{i_p} \cup J^{run}_{i_p})} (r^R_{j_{i_p}} * x_{(j_{i_p}, c_{st_j}, t)}) \tag{4.18}$$

Constraints 4.19 and 4.20 define for each VM instance $v \in V, k_v \in K_v$ the remaining free capacities in terms of CPU ($f^C_{k_v}$) and RAM ($f^R_{k_v}$) that are not reserved and allocated to containers on the VM instance. As expressed by the variable $g_{(k_v,t)} \in \{0, 1\}$, which is further defined in Constraints 4.21 and 4.22, we only consider VM instances that are either already running or are being leased in τ_t.

$$g_{(k_v,t)} * s^C_v - \sum_{st \in ST} \sum_{c_{st} \in C_{st}} (s^C_{c_{st}} * a_{(c_{st}, k_v, t)}) \leqslant f^C_{k_v} \tag{4.19}$$

$$g_{(k_v,t)} * s^R_v - \sum_{st \in ST} \sum_{c_{st} \in C_{st}} (s^R_{c_{st}} * a_{(c_{st}, k_v, t)}) \leqslant f^R_{k_v} \tag{4.20}$$

Constraints 4.21 and 4.22 define the value of the helper variable $g_{(k_v,t)} \in \{0,1\}$ which takes the value of 1 if a VM instance is either already running ($\beta_{k_v} = 1$) or being leased for at least one BTU ($y_{(k_v,t)} \geq 1$) in τ_t. $g_{(k_v,t)}$ is restricted to be a boolean variable in Constraint 4.45.

$$g_{(k_v,t)} \leqslant \beta_{(k_v,t)} + y_{(k_v,t)} \tag{4.21}$$

$$\beta_{(k_v,t)} + y_{(k_v,t)} \leqslant BTU^{max} * g_{(k_v,t)} \tag{4.22}$$

Constraint 4.23 makes sure that for all VM instances $v \in V, k_v \in K_v$ a VM instance k_v will be leased or is running if containers are to be placed on it. Therefore, we calculate the number of containers that are scheduled or already running on k_v and make use of our helper variable $g_{(k_v,t)}$ as defined in Constraints 4.21 and 4.22, as well as a helper variable M that presents an arbitrarily large number, e.g., 100, as an upper bound of possible container deployments per VM instance.

$$\sum_{st \in ST} \sum_{c_{st} \in C_{st}} a_{(c_{st},k_v,t)} \leqslant g_{(k_v,t)} * M \tag{4.23}$$

Constraint 4.24 makes sure that for all container instances $st \in ST, c_{st} \in C_{st}$ a container instance c_{st} will be deployed or is already running on a VM instance k_v if service invocations are scheduled for execution on the container. To that end, we calculate the number of service invocations that will be scheduled or are already running on a container instance c_{st} and make use of our decision variable $a_{(c_{st},k_v,t)}$ that indicates whether a container should be deployed on a VM instance k_v, as well as a helper variable N that presents an arbitrarily large number, e.g., 10,000, as an upper bound of possible parallel service invocations for each container instance.

$$\sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in (J^*_{ip} \bigcup J^{run}_{ip})} x_{(j_{i_p},c_{st},t)} \leqslant a_{(c_{st},k_v,t)} * N \tag{4.24}$$

Constraint 4.25 makes sure that for all VM instances $v \in V, k_v \in K_v$ and all container instances $st \in ST, c_{st} \in C_{st}$ a VM instance k_v will be leased at least for the full next optimization period if container instances c_{st} are allocated on it. On the lefthand side, we consider the decision whether or not a container should be deployed on a VM instance k_v and calculate the time until the start of the next optimization period, as computed by the optimization model with the current knowledge. On the righthand side, we demand that if a VM instance k_v is already leased, its remaining leasing duration $d_{(k_v,t)}$ is greater or equal to the calculated time until the start of the next optimization period. If the currently remaining leasing duration is smaller, the corresponding VM instance k_v needs to be leased for $y_{(k_v,t)}$ additional BTUs to meet the equation.

$$a_{(c_{st},k_v,t)} * (\tau_{(t+1)} - \tau_t) \leqslant d_{(k_v,t)} * \beta_{(k_v,t)} + BTU_{k_v} * y_{(k_v,t)} \tag{4.25}$$

Due to the multiplication of the binary variable $a_{(c_{st},k_v,t)}$ with the continuous variable $\tau_{(t+1)}$ Equation 4.25 is nonlinear. Therefore, we need to perform a linearization for the nonlinear term $a_{(c_{st},k_v,t)} * \tau_{(t+1)}$ and replace it with the new variable $a_{(c_{st},k_v,t)}\_times\_\tau_{(t+1)}$. The linearization of the product of the binary variable $a_{(c_{st},k_v,t)}$ and the continous variable $\tau_{(t+1)}$ [56] is expressed by the Constraints 4.26 - 4.29.

$$a_{(c_{st},k_v,t)}\_times\_\tau_{(t+1)} \leqslant \tau_{(t+1)}\_UpperBound * a_{(c_{st},k_v,t)} \tag{4.26}$$

$$a_{(c_{st},k_v,t)}\_times\_\tau_{(t+1)} \leqslant \tau_{(t+1)} \tag{4.27}$$

$$\begin{aligned} a_{(c_{st},k_v,t)}\_times\_\tau_{(t+1)} + \tau_{(t+1)}\_UpperBound \\ \geqslant \tau_{(t+1)} + \tau_{(t+1)}\_UpperBound * a_{(c_{st},k_v,t)} \end{aligned} \tag{4.28}$$

$$a_{(c_{st},k_v,t)}\_times\_\tau_{(t+1)} \geqslant 0 \tag{4.29}$$

Constraint 4.30 makes sure that for all VM instances $v \in V, k_v \in K_v$, all container instances $st \in ST, c_{st} \in C_{st}$ and all running or schedulable process steps $j_{i_p} \in (J^*_{i_p} \bigcup J^{run}_{i_p})$, the containers c_{st} with scheduled service invocations j_{i_p} will not be moved to other VM instances during their services invocations. On the lefthand side, we consider the remaining execution time of a process step j_{i_p} for each process step scheduled on a container instance ($a_{(c_{st},k_v,t)} = 1$) and each container scheduled on a VM instance ($x_{(j_{i_p},c_{st},t)} = 1$). In case that the process step j_{i_p} is not running yet, we also have to consider the time to pull a new image and starting the needed container, if the image does not exist and the container is not running on the scheduled VM instance ($z_{(st_j,k_v,t)} = 0$). Furthermore, if the required VM instance k_v is not up and running yet ($\beta_{(k_v,t)} = 0$) , we also need to add the time for starting the new VM instance k_v to our calculation of the remaining execution time. Similarly to Constraint 4.25, we demand the calculated remaining execution time to be smaller or equal to the remaining leasing duration $d_{(k_v,t)}$ of the allocated VM instance k_v. Again, if the currently remaining leasing duration is smaller, the corresponding VM instance k_v needs to be leased for $y_{(k_v,t)}$ additional BTUs to satisfy the equation.

$$\begin{aligned} \big(e_{j_{i_p}} + (\Delta_{c_{st_j}} + \Delta_{st_j}) * (1 - z_{(st_j,k_v,t)}) + \Delta * (1 - \beta_{(k_v,t)})\big) * a_{(c_{st},k_v,t)} * x_{(j_{i_p},c_{st},t)} \\ \leqslant d_{(k_v,t)} * \beta_{(k_v,t)} + BTU_{k_v} * y_{(k_v,t)} \end{aligned} \tag{4.30}$$

Due to the multiplication of the two binary variable $a_{(c_{st},k_v,t)} * x_{(j_{i_p},c_{st},t)}$ Equation 4.30 is nonlinear. Therefore, we need to perform a linearization for the nonlinear term and replace it with the new variable $a_{(c_{st},k_v,t)}\_times\_x_{(j_{i_p},c_{st},t)}$. The linearization of the product of the two binary variables $a_{(c_{st},k_v,t)}$ and $x_{(j_{i_p},c_{st},t)}$ [142] is expressed by the Constraints 4.31 - 4.33.

$$a_{(c_{st},k_v,t)}\_times\_x_{(j_{i_p},c_{st},t)} \leqslant a_{(c_{st},k_v,t)} \tag{4.31}$$

$$a_{(c_{st},k_v,t)}\_times\_x_{(j_{i_p},c_{st},t)} \leqslant x_{(j_{i_p},c_{st},t)} \tag{4.32}$$

$$a_{(c_{st},k_v,t)}\_times\_x_{(j_{i_p},c_{st},t)} \geqslant a_{(c_{st},k_v,t)} + x_{(j_{i_p},c_{st},t)} - 1 \tag{4.33}$$

Similar to Constraint 4.30, Constraint 4.34 together with the Constraints 4.38 and 4.39, ensure that all service invocations that are already running on certain container instances on a VM instance at the start of τ_t, will not be migrated to another container or VM instance during their invocation. We explicitly reference the already assigned container instance c_{st} and VM instance k_v by the additional indices in Constraint 4.34 and do not need to consider additional times for pulling container images or starting VM instances.

$$e_{j_{i_p}}^{run_{c_{st},k_v}} \leqslant d_{(k_v,t)} * \beta_{(k_v,t)} + BTU_{k_v} * y_{(k_v,t)} \tag{4.34}$$

Constraint 4.35 defines the variable $\gamma_{(v,t)}$ for all VM types $v \in V$. $\gamma_{(v,t)}$ indicates the total number of BTUs to lease VM instances of type v, meaning it includes the decisions which VM instances k_v to lease and for how long (how many BTUs) to lease them. To define $\gamma_{(v,t)}$, we build the sum over all individually leased BTUs for each VM instance of type v.

$$\sum_{k_v \in K_v} y_{(k_v,t)} \leqslant \gamma_{(v,t)} \tag{4.35}$$

Constraint 4.36 demands for all next schedulable process steps $j_{i_p} \in J_{i_p}^*$ that each service invocation is scheduled on only a single container instance c_{st} corresponding to the service instances service type st_j or postponed for a later optimization period.

$$\sum_{c_{st_j} \in C_{st_j}} x_{(j_{i_p},c_{st_j}t)} \leqslant 1 \tag{4.36}$$

Constraint 4.37 demands for all container instances $st \in ST, c_{st} \in C_{st}$ that each container instance can only be deployed on one VM instance k_v.

$$\sum_{k_v \in K_v} a_{(c_{st},k_v t)} \leqslant 1 \tag{4.37}$$

Constraint 4.38 sets the decision variable $x_{(j_{i_p},c_{st},t)}$ for each already running service invocation $j_{i_p}^{run}$ on a corresponding running container instance $c_{st_j}^{run}$ to 1. Furthermore, Constraint 4.39 sets the decision variable $a_{(c_{st},k_v,t)}$ for each already running container from Constraint 4.38 on a corresponding running VM instance k_v to 1. Those two constraints correspond to inheriting the scheduling decision already made in a previous period.

$$x_{(j_{i_p}^{run},c_{st_j}^{run},t)} = 1 \tag{4.38}$$

$$a_{(c_{st_j}^{run},k_v,t)} = 1 \tag{4.39}$$

Constraint 4.40 restricts the decision variable $x_{(j_{i_p},c_{st_j},t)}$ that decides whether or not to schedule a service invocation j_{i_p} on a certain container c_{st} for all $p \in P, i_p \in I_p, j_{i_p} \in J_{i_p}^*, st \in ST, c_{st} \in C_{st}$ to be a boolean.

$$x_{(j_{i_p},c_{st_j},t)} \in \{0,1\} \tag{4.40}$$

Constraint 4.41 restricts the decision variable $a_{(c_{st},k_v,t)}$ that decides whether or not to deploy a container instance c_{st} on a certain VM instance k_v for all $st \in ST, c_{st} \in C_{st}, v \in V, k_v \in K_v$ to be a boolean.

$$a_{(c_{st},k_v,t)} \in \{0,1\} \tag{4.41}$$

Constraint 4.42 restricts the decision variable $y_{(k_v,t)}$ that decides for how many additional BTUs to lease a VM instance k_v at time t for all $v \in V, k_v \in K_v$ to be a positive integer ≥ 0.

$$y_{(k_v,t)} \in \mathbb{N}_0^+ \tag{4.42}$$

Constraint 4.43 restricts the decision variable $\gamma_{(v,t)}$ that decides for how many total BTUs to lease any VM instances of type v at time t for all $v \in V$ to be a positive integer ≥ 0.

$$\gamma_{(v,t)} \in \mathbb{N}_0^+ \tag{4.43}$$

Constraint 4.44 restricts the variable that indicates the penalty execution time $e_{i_p}^p$ to be a positive real number.

$$e_{i_p}^p \in \mathbb{R}^+ \tag{4.44}$$

Constraint 4.45 restricts the variable $g_{(k_v,t)}$ to be a boolean. The variable has been defined in Constraints 4.21 and 4.22 and takes the value of 1 if a VM instance is either already running ($\beta_{k_v} = 1$) or being leased for at least one BTU ($y_{(k_v,t)} \geq 1$) in τ_t.

$$g_{(k_v,t)} \in \{0, 1\} \tag{4.45}$$

Constraint 4.46 restricts the helper variable $a_{(c_{st},k_v,t)}\_times\_\tau_{(t+1)}$ used for the linarization of Constraint 4.25 in Constraints 4.26 - 4.29 to be a positive real number.

$$a_{(c_{st},k_v,t)}\_times\_\tau_{(t+1)} \in \mathbb{R}^+ \tag{4.46}$$

Constraint 4.47 restricts the helper variable $a_{(c_{st},k_v,t)}\_times\_x_{(j_{i_p},c_{st},t)}$ used for the linarization of Constraint 4.30 in Constraints 4.31 - 4.33 to be a boolean value.

$$a_{(c_{st},k_v,t)}\_times\_x_{(j_{i_p},c_{st},t)} \in \{0, 1\} \tag{4.47}$$

Problem Simplification through Additional Restriction

Up until this point the generally defined optimization model does not restrict the deployment of multiple containers of the same service type on the same VM instance. In our preconditions, we stated that we only want to allow one single container per service type st on each VM instance.

It should be noted that the deployment of multiple containers of the same service type on the same VM instance can be desirable and even come with a number of advantages, especially in terms of security requirements in a multi-tenant environment. It may be required that either each individual service request or certain specific service requests are executed in isolation.

Typical reasons for demanding stronger isolation of service requests and their used resources are the following:

- **Security**: A higher security can be guaranteed for service executions if requests run in isolated containers [42]. This is especially the case for multi-tenant applications where situations of cross-tenant data leakage need to be prevented and the associated risks have to be minimized [21] [105].

- **Regulations**: Often higher security measures have to be incorporated due to legal and compliance reasons and data privacy standards [74]. Different regulations, e.g., from different companies or countries [85], may also impose different requirements to the execution of the same services, such that they have to be packed into different container instances .

- **QoS guarantees in multi-tenant environments**: The isolation of requests from different tenants might be necessary for realizing different advanced multi-tenant QoS guarantees [60] [145]. Other than the already considered execution deadlines, cloud providers can, for example, offer different QoS levels for storage and encryption [100]. A provider might have to offer different methods to save the state of containers [60], e.g., some customers might require persistent state storage while for others saving the state in an ephemeral storage can be sufficient. Also when offering different encryption methods for encryption in transit (e.g., using an encrypted connection like HTTPS vs. not offering encrypted data transmission) and encryption at rest (e.g., offering the possibility to save data in encrypted storage drives) the use of different containers for the same service types can be helpful.

- **Billing**: Resource isolation on a tenant basis is also helpful when defining certain resource limits per tenant, and especially for tracking activities and resource usage on a tenant basis [57] [140] [174]. Tracking a tenant's activities and usage is crucial for incorporating advanced billing mechanisms that allow the usage of on-demand pricing schemes.

It should be noted that also when looking at the advantage of resource sharing while using SOA for the deployment of business processes, such advantages are still present, even when each service request runs as an isolated process in its own container instance. The reason for this is that multiple container instances can be started from the same container image in a neglectable amount of time and containers can share the resources offered by a VM instance. Therefore, the additional isolation does not diminish advantages of concurrent execution on shared resources.

Nevertheless, current work on task-scheduling and resource allocation for elastic processes has not focused on these issues. Existing approaches as presented in Chapter 3 mostly utilize homogeneously configured VM instances for the execution of concurrent process requests. The closely related work from Hoenisch et. al [65] that makes use of Docker containers for isolation of applications on VM instances also only accounts for scheduling maximally one container of a certain application type on a VM instance without accounting for situations where multiple application containers for the same application could be reasonable.

Our optimization approach as presented so far does not account for decisions that favor the deployment of a process step on multiple containers in different situations as stated above, e.g., to meet legal and compliance regulations. Nevertheless, the approach generally allows the scheduling of multiple container instances of the same service type on a single VM instance, which offers a basis for introducing further constraints that tackle the above considerations. Defining and realizing such considerations in more detail goes beyond the scope of this thesis and should be explored by future work.

Additional restriction: In the work at hand, we provide a solution for the simplified assumption that maximally one container of the same service type st is allowed per VM instance. This also corresponds to similar assumptions made in related work.

To realize the simplified assumption of this work that for all $v \in V, k_v \in K_v$ each VM instance may only host one container of the same service type st, we introduce Constraint 4.48.

$$\sum_{c_{st} \in C_{st}} a_{(c_{st}, k_v t)} \leqslant 1 \tag{4.48}$$

Due to this additional restriction, our presented general optimization model can be significantly simplified. We now can abstract from having to schedule container instances on VM instances and service invocations on container instances. Instead, we can directly schedule service invocations on VM instances, while considering container properties. The actual container instances can be defined and allocated as needed by the scheduled service invocations.

Issues with current approaches: The related work on scheduling applications on container instances based on MILP optimization models [65] assumes a discrete set of container instances with underlying resources that can be assigned according to a predefined and discrete set of resource variables. Such a discrete pool of containers would also be necessary in our case when trying to directly solve the optimization problem as defined in this chapter using a MILP solver. It is important to consider that doing so would be associated with a number of limitations:

- **Large number of decision variables**: The defined optimization model would have to handle a very large number of decision variables.
  - First, the solver would need to decide for every VM instance in the set of leasable instances whether or not to lease the VM instance.

  - Second, the solver would need to decide for every container instance whether or not to deploy a container with a certain configuration on a certain VM instance.
  - Lastly, the solver would need to decide on which deployed container instance to schedule the service requests.

- **High computational complexity**: The computational complexity grows exponentially with the number of incoming requests and also with the number of VM instances and container instances that have to be managed. This can easily lead to unacceptably long computation times even for slightly increasing problem sizes.

  Previous optimization approaches have only considered scheduling over two dimensions (e.g., [63]), which entails a lower computational complexity than the problem we consider. Yet also in simlar previous MILP-based optimization approaches it could be observed that solutions struggled with increasing problem sizes [97].

- **Manage container instances**: Maintaining a large enough finite set of container instances that allow for a flexible system configuration is in itself very expensive. At the same time, when not enough container instances are available and managed by the system, the scheduling decisions can be artificially aggravated and lead to poor scheduling outcomes.

  Note that this problem did not manifest in the previous work by Hönisch et al. [65] where only two dimensions are considered and hence the managed finite pool of containers is considerably smaller than in our case.

To give an example, if the system manages five different container instances with different configurations for each of ten different service types, the system would already need to manage and make decisions for 50 different container instances. If the system had a pool of ten different VM instances to choose from, for each incoming service request the system would need to decide on which of the 50 available containers to schedule a service invocation and on which of the ten available VM instances to deploy the container. This computation is not only highly expensive but can also lead to ineffective results. By managing a pool of container instances, the system is restricted, although containers have the potential to provide a high flexibility. Out of the 50 containers in the pool, only five are actually of relevance to a certain service type and when a container of a certain service type with a certain configuration is already working at capacity, the system might have to choose another container that is configured to offer more computational resources than actually required for the scheduled service invocations. This would not only lead to a waste of computational resources due to the additional space that the container guarantees to its service invocations without an actual demand, but could also waste an even larger amount of resources provided by VM instances, as a larger container might require the lease of an additional VM instance, although an already leased VM instance might offer sufficient resources for executing the steps scheduled on a container, but not for offering the resources the pre-configured container actually demands.

Solving the limitations of discrete container pools: To overcome these limitations of a predefined set of container instances and by making use of the simplified assumption that only one

container instance for each service type may be deployed on each VM instance, we implement a solution that meets all the constraints defined by the optimization model described in this section, without having to actually manage a pool of containers. Our implementation solves a reduced problem only considering the scheduling of service invocations on VM instances, while considering container-based information. The reduced problem may again be formulated as a MILP optimization problem and solved using a MILP solver. Another possibility is to formulate the reduced optimization problem as an evolutionary based algorithm, e.g., a genetic algorithm. In both cases the input for the reduced problem are the process requests, possible container images, and a pool of VM instances. The output of the reduced model is a task-scheduling and resource allocation decision stating for all VM instances $v \in V, k_v \in K_v$ what VM instances to lease for how many BTUs ($y_{(k_v,t)}$) and for all next schedulable or running process steps $j_{i_p \in J^*_{i_p} \bigcup J^{run}_{i_p}}$ on which VM instance $v \in V, k_v \in K_v$ to schedule the service invocations ($x_{(j_{i_p},k_v,t)}$). This output is transformed using an algorithm that calculates a task-scheduling solution that creates container instances corresponding to the actual need as computed by the reduced model, such that no resources are wasted and an unlimited amount of container instances may be used.

Overall optimization approach summary: To sum it up, the overall optimization approach as presented in this chapter aims at finding a scheduling solution for service invocations on container instances that are deployed on VM instances. The overall inputs of the optimization problem are the client requests i_p to enact process instances within defined deadlines, a set of different VM types and a pool of VM instances, and a set of different container images for different software services. The overall outputs of the optimization problem are the following:

- the decision which VM instances to lease for how many BTUs ($y_{(k_v,t)}$)
- which containers with which configurations to deploy on the VM instances ($a_{(c_{st},k_v,t)}$)
- which service invocations to schedule on which container instance ($x_{(j_{i_p},c_{st},t)}$)

Reduced optimization approach summary: To realize the overall optimization approach in an effective way, we implement a reduced optimization model that processes the inputs of the overall approach and generates the following outputs:

- the decision which VM instances to lease for how many BTUs ($y_{(k_v,t)}$)
- which service invocations to schedule on which VM instance ($x_{(j_{i_p},k_v,t)}$)

The reduced output already considers container-based information and is transformed by an algorithm that creates container instances corresponding to the actual need as computed by the reduced model. The transformation processes the decision variables $x_{(j_{i_p},k_v,t)}$ and creates the two variables $a_{(c_{st},k_v,t)}$ and $x_{(j_{i_p},c_{st},t)}$ as described in the overall optimization output. This leads to a significantly reduced computational effort for realizing the optimization approach and also creates containers that perfectly correspond to the actual demand of the scheduled service invocations. The highly flexible resulting system structure is further discussed in Chapter 5.

4.5 Working Example Execution

To facilitate the understanding of the presented MILP optimization approach discussed in Section 4.4, in this section we demonstrate how the solution performs on the simplified working example scenario presented in Section 4.3.

4.5.1 Execution of the Process Requests of the Working Example

We explain the main functionalities and decisions of our BPMS for realizing a self-adaptive auto-scaling system for the execution of elastic processes in containerized cloud environments, by illustrating how the system behaves over time for executing the incoming process requests from Table 4.9 which are composed of service steps of different service types with certain computational requirements as detailed in Table 4.7, while using the available computational resources as described in Subsection 4.3.3.

For executing incoming process requests, the BPMS has to take all the details summarized in Section 4.3 into account. In addition to that, the startup times of VMs and the time to pull new Docker images on running machines have to be considered. In the following example, we assume a general VM startup time of two minutes and a container image pull time of 30 sec. Once a container image exists on a VM we assume a negligible container startup time of very few seconds and will not consider this in the further illustration of the execution of the ongoing example.

Our designed BPMS prioritizes the scheduling and execution of process invocations depending on the corresponding process deadlines. To accomplish this, our system uses a worst-case analysis to estimate the priority of the next process steps. This means for every step that has to be executed within a process instance, we consider the makespan of the process step and additionally, assume that a new VM has to be leased and the corresponding container image has to be pulled on the new machine for executing the step. Therefore, for each process step we also consider the VM startup time and container pull time when calculating the latest time a process step can be scheduled to still meet its deadline.

When looking at the incoming process requests from Table 4.9, we can see that with a worst-case analysis the process instances P1.1, PN.1, P3.2, and PN.2 have a very strict deadline and need to be scheduled for execution as soon as they arrive, so they do not violate their deadline constraints. On the other side, process instance P3.1 has a very lenient deadline and the execution of the first process step P3.1.B can be postponed for a rather long time to still meet the process deadline. To facilitate the explanation of scheduling decisions Figure 4.3 shows all incoming process requests with all process steps and the scheduling deadlines when performing a worst-case analysis for each process step.

In the following Figures 4.4 to 4.16, the next process steps of process requests that can be scheduled as soon as their predecessors are finished are outlined by a bold red frame ($j^*_{i_p}$). Already scheduled but not yet running steps are outlined by a bold black frame ($\in j^{run}_{i_p}$). Already finished process steps and process steps that are currently being executed have a white filling in the overviews of running process requests. Process steps that are being executed are furthermore outlined by a bold blue frame ($\in j^{run}_{i_p}$) and can be found in containers (c_{st}) on VMs (k_v) in the right part of Figures 4.4 to 4.16, in which the currently leased and running VMs are illustrated.

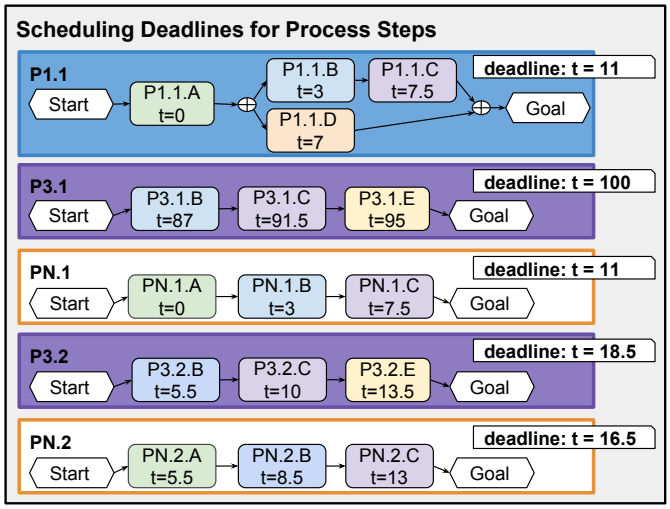


Figure 4.3: Scheduling Deadlines for Process Steps

VM instances that have been requested but are still booting up, are indicated by a dotted line. The figures also show the already pulled Docker images (st) on the top right corner for each VM instance. Docker images that are currently being pulled are also indicated by a dotted line and shaded in gray.

System Landscape at t = 0

We assume to start with an empty system landscape, with no running process instances and no leased VM resources. At $t = 0$, the system receives three new process requests as presented in Table 4.9 and illustrated in Figure 4.3. The process landscape after all decisions made at $t = 0$ is depicted in Figure 4.4. To begin with, the system analyzes the first executable process steps of each received process instance. The requests P1.1 and PN.1 have strict deadlines and require an immediate scheduling and start of execution for their first process steps P1.1.A and PN.1.A when performing a worst-case analysis, while P3.1 has a lenient deadline and the execution of P3.1.B can be postponed into the future. Therefore, the system decides to schedule the process steps P1.1.A and PN.1.A in an isolated container of image type A on the cheapest available computational resource VM4.1, which is large enough for both process requests. At $t = 0$ the system requests to lease VM4.1, which will only be up and running at $t = 2$, and leased until the end of one BTU at $t = 7$. The execution of the process requests can only be started at $t = 2.5$ when the image A which allows deploying containers that can execute service invocations of service type A is pulled on VM4.1.

System Landscape at t = 2.5

At $t = 2.5$ the system starts the execution of the two scheduled process steps P1.1.A and PN.1.A on the new container created from image A on VM4.1 as shown in Figure 4.5. Both process steps will take 30 seconds to be finished.

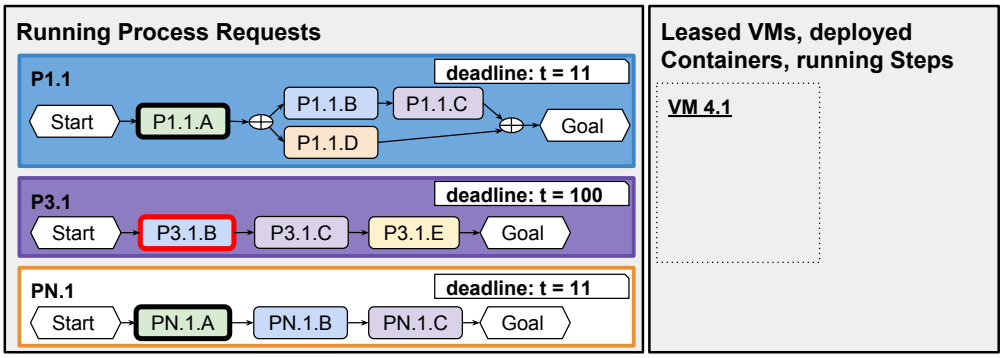


Figure 4.4: System Landscape at t = 0

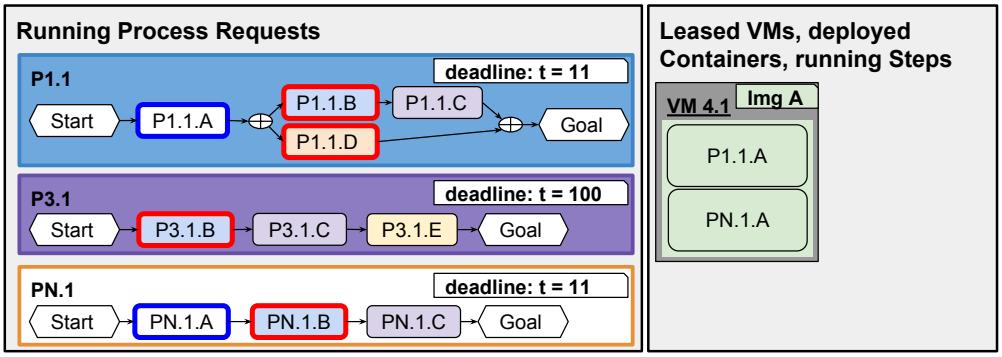


Figure 4.5: System Landscape at t = 2.5

System Landscape at t = 3

At $t = 3$, the execution of P1.1.A and PN.1.A is completed and the system can schedule the subsequent steps of P1.1 and PN.1 as can be seen in Figure 4.6. When making a scheduling decision, the system considers all next steps (indicated by a red frame) that may be scheduled and do not have a preceding step that is still under execution (indicated by a blue frame). Due to the worst-case analysis, the system needs to schedule the process steps P1.1.B and PN.1.B at this stage. As VM 4.1 offers enough space for executing all next steps of service type B and no more process steps of service type A are needed, the system shuts down the running container of type A on VM4.1 and downloads the image for starting containers of service type B. While doing so, the already existing image for containers of service type A remains on VM4.1. Note that the system decides to also schedule the step P3.1.B (indicated by a thick black frame), although the deadline is in the distant future, as enough computational resources are available for the execution of the step without incurring any extra costs and while optimizing the systems utilization. Furthermore, the system decides to postpone the scheduling of P1.1.D, as there are currently not enough computational resources available for its execution.

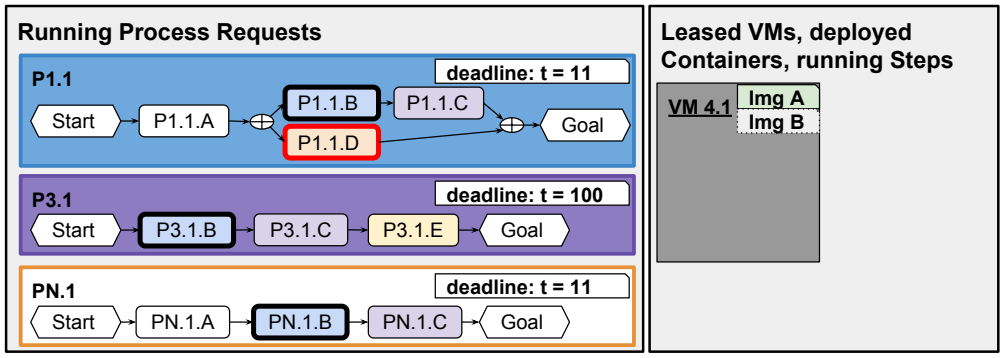


Figure 4.6: System Landscape at t = 3

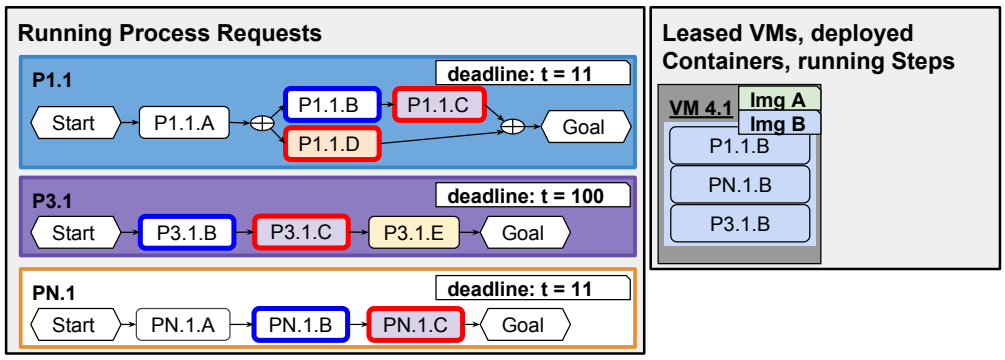


Figure 4.7: System Landscape at t = 3.5

System Landscape at t = 3.5

At $t = 3.5$, the image for containers of type B is fully downloaded on VM4.1 and the system starts with the execution of the three service invocations of type B as shown in Figure 4.7. At this stage, 90% of the available CPU resources of VM4.1 are being utilized.

System Landscape at t = 5.5

At $t = 5.5$, the system receives two new process requests as shown in Figure 4.8. Due to the worst-case analysis, the first process steps P3.2.B and PN.2.A of these new process requests need to be scheduled immediately.

Simultaneously, all previously running service invocations of service type B on VM4.1 are finished. Although the subsequent process steps P1.1.C, P3.1.C, and PN.1.C of the corresponding process instances may now be scheduled by the system, their scheduling is postponed as there are not enough computational resources available so the scheduling would incur additional leasing costs and the deadline allows for a later scheduling.

VM4.1 offers enough computational resources for executing the time-critical process steps P3.2.B and PN.2.A. The system downsizes the currently running container of type B and starts a new container of type A from the already existing image on VM4.1.

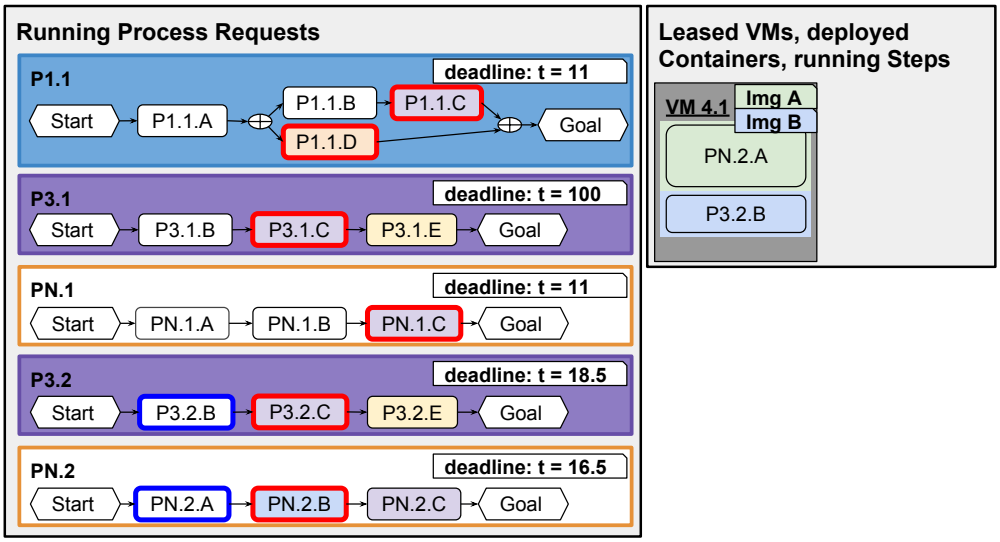


Figure 4.8: System Landscape at t = 5.5

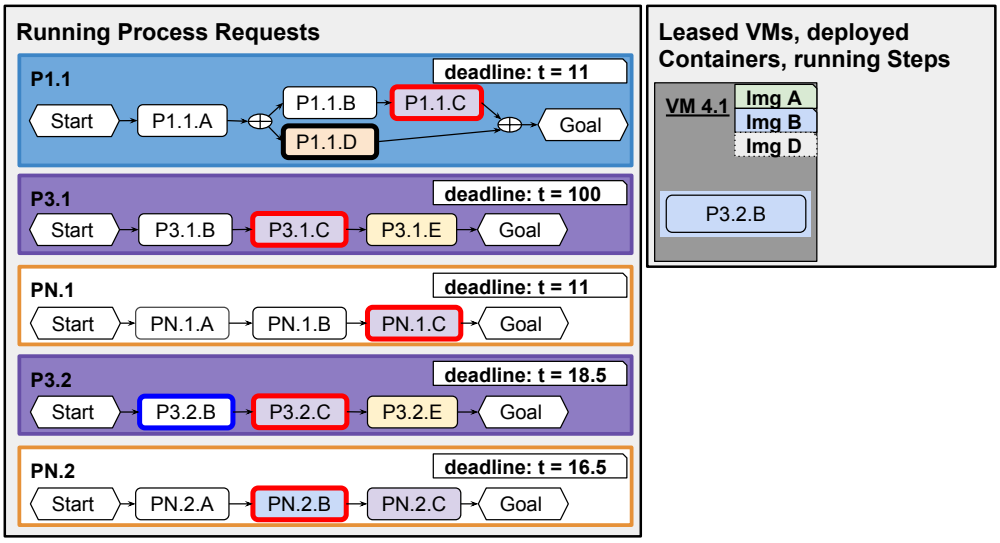


Figure 4.9: System Landscape at t = 6

As the remaining leasing duration of VM4.1 is only 1.5 more minutes and the execution of P3.2.B requires two minutes, the leasing duration of VM4.1 is extended by one more BTU and the process steps are both scheduled and executed on VM4.1.

System Landscape at t = 6

At $t = 6$, the process step PN.2.A is finished and the next process step PN.2.B can be scheduled. Because no further process steps of type A need to be scheduled, the corresponding container on

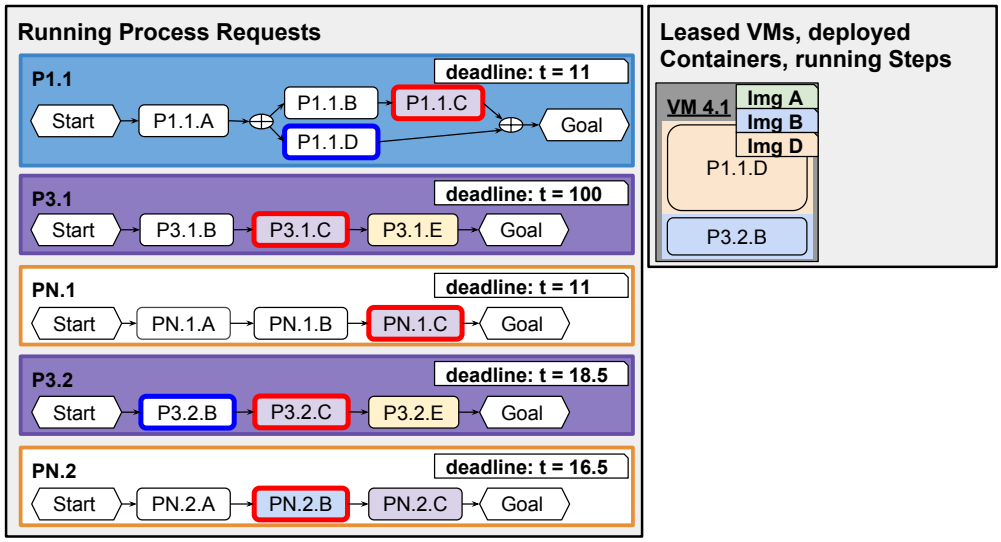


Figure 4.10: System Landscape at t = 6.5

VM4.1 is closed as shown in Figure 4.9. All next steps that may be scheduled can currently also be postponed. The system only offers enough resources for the execution of P1.1.D or PN.2.B. As the deadline of P1.1.D is closer, the system schedules the execution of P1.1.D on VM4.1 and initiates pulling the image for creating containers of type D on VM4.1.

System Landscape at t = 6.5

At $t = 6.5$, the image for creating containers of type D is fully pulled and the scheduled process step P1.1.D is started on a corresponding container as shown in Figure 4.10.

System Landscape at t = 7.5

At $t = 7.5$, the execution of P3.2.B finishes. The freed up resources on VM4.1 only allow to schedule and start the execution of PN.2.B as shown in Figure 4.11.

According to a worst-case analysis, the scheduling of P1.1.C and PN.1.C needs to be performed now, as a further delay might lead to a deadline violation. The system schedules the process steps on VM6.1 as it leads to the smallest additional costs while offering the needed computational resources for both process steps. As VM6.1 offers more resources than needed by the two process steps, the system additionally schedules P3.2.C on the free resources. At this stage, the system initiates leasing the required VM6.1.

System Landscape at t = 8

At $t = 8$, the execution of P1.1.D finishes and the corresponding container is shut down.

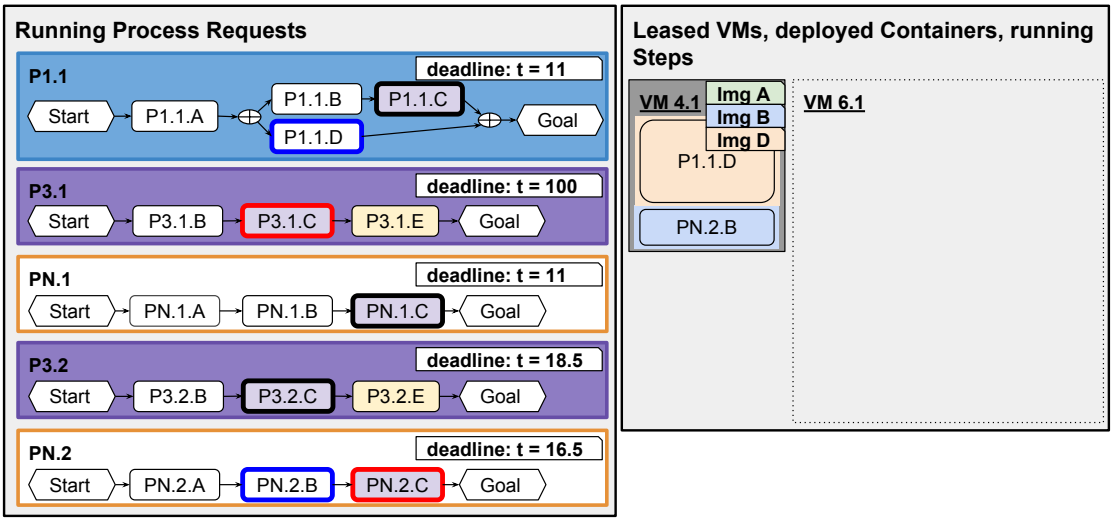


Figure 4.11: System Landscape at t = 7.5

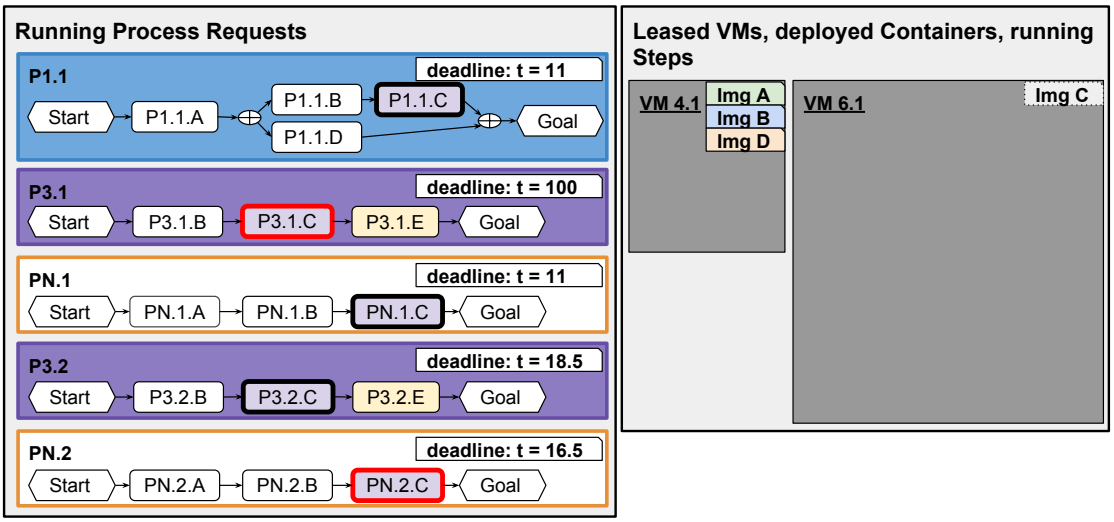


Figure 4.12: System Landscape at t = 9.5

System Landscape at t = 9.5

At $t = 9.5$, VM6.1 has fully booted up and is running, so the system starts pulling the required image for deploying containers of type C on the VM as shown in Figure 4.12. Also, the execution of PN.2.B finishes and the corresponding container is shut down.

System Landscape at t = 10

At $t = 10$, the image for deploying type C containers is pulled on VM6.1, so the system starts a container large enough for executing the three scheduled process steps as shown in Figure 4.13.

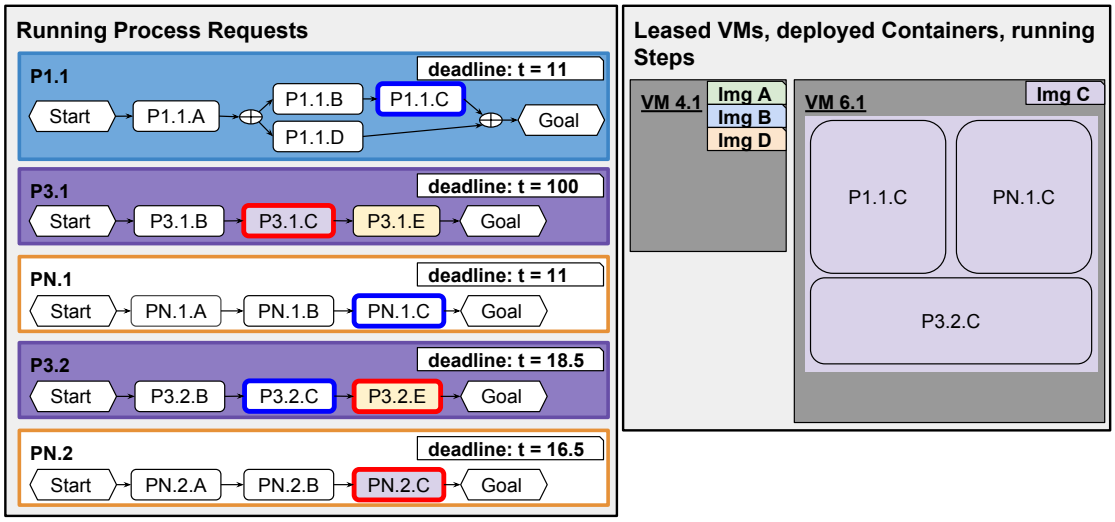


Figure 4.13: System Landscape at t = 10

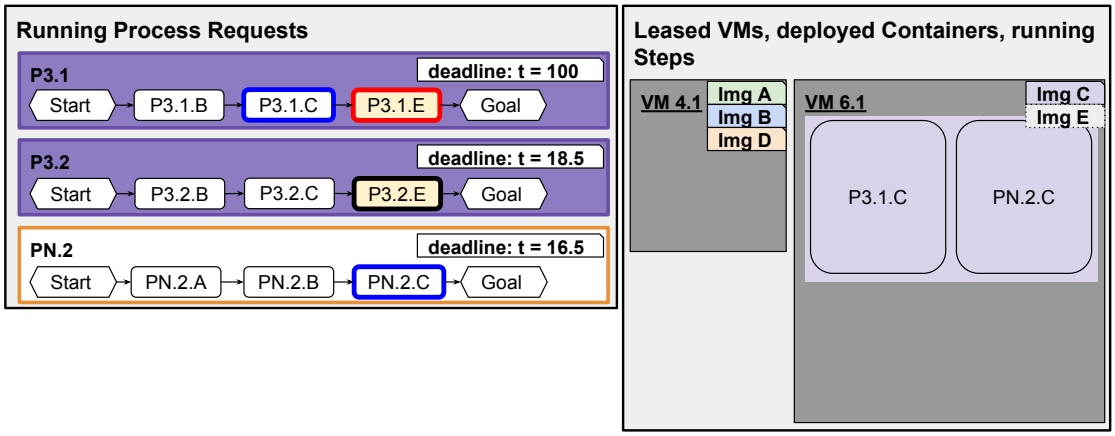


Figure 4.14: System Landscape at t = 11

System Landscape at t = 11

At $t = 11$, the execution of P1.1.C, PN.1.C and P3.2.C finishes and the corresponding process instances P1.1 and PN.1 also finish in time. The system schedules and starts the execution of all next process steps that can be scheduled, as VM6.1 offers enough computational resources. The execution of P3.1.C and PN.1.C is directly started on the resized container of type C, while for executing process step P3.2.E the system initiates pulling the image for creating a corresponding container of type E, as shown in Figure 4.14.

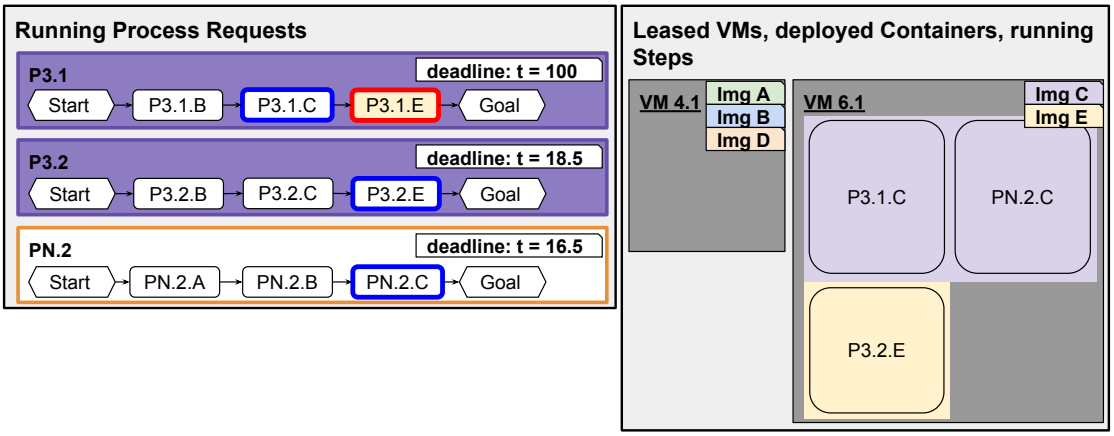


Figure 4.15: System Landscape at t = 11.5

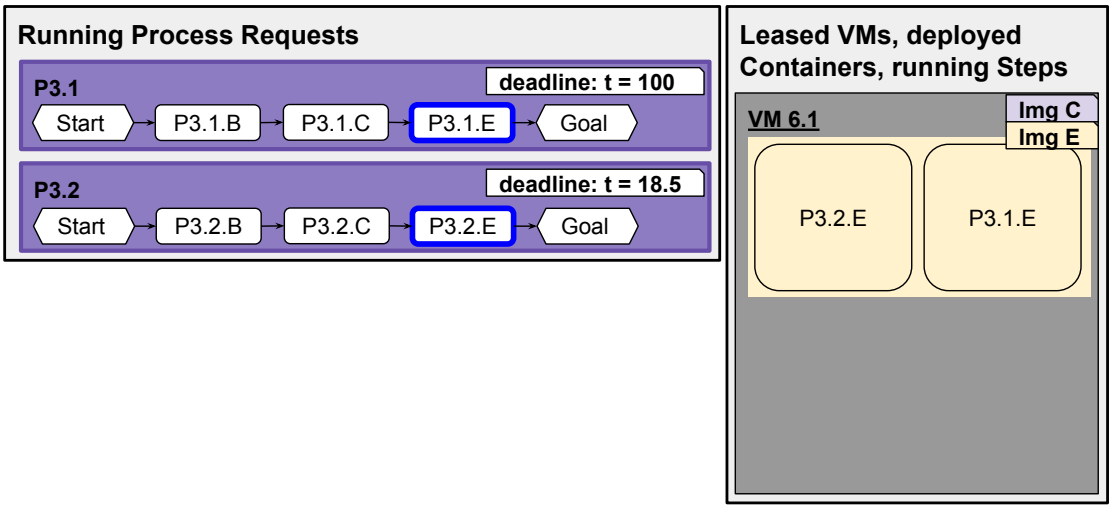


Figure 4.16: System Landscape at t = 12

System Landscape at t = 11.5

At $t = 11.5$, the image for starting containers of type E is fully pulled on VM6.1 and the system starts the execution of the scheduled process step P3.2.E as shown in Figure 4.15.

System Landscape at t = 12

At $t = 12$, the executions of P3.1.C and PN.2.C end and the process instance PN.2 is finished in time. The system closes the container of type C and schedules the execution of the next process step P3.1.E on the resized container instance of type E on VM6.1 as shown in Figure 4.16. At the same time, VM4.1 is released as it reached the end of its second BTU.

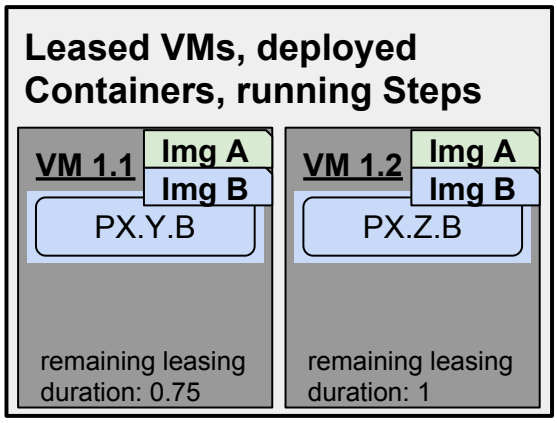


Figure 4.17: System Landscape at t = t_0

System Landscape at t = 14

At $t = 14$, the execution of process step P3.2.E ends and the process instance P3.2 finishes in time.

System Landscape at t = 14.5

At $t = 14.5$, the execution of process step P3.1.E ends and the process instance P3.1 finishes in time. The corresponding container is closed and as VM6.1 reached the end of its BTU it is released.

Now the system landscape has no leased computational resources and no running process instance.

Evaluation

We provide an evaluation and a more in depth discussion of the execution of our working example in Chapter 6.

4.5.2 Further Scenario to Explain Term 4 of the Objective Function

To clarify the reasoning behind Term 4 of objective function 4.1 that tries to maximize the future resource supply of already leased VM instances, we demonstrate a short scenario independent of the remaining presented working example. We start with an initial system structure at time t which is in a state as shown in Figure 4.17. The shown running steps of type B are assumed to be finished simultaneously at time $t = t_0 + 0.75$.

At $t = t_0 + 0.1$, our system needs to initiate the invocation of a process step PX.Y.A of type A that has a makespan of 30 seconds. As the needed container for this process step is deployed on both running VM instances VM1.1 and VM1.2, the invocation of process step PX.Y.A would be finished until $t = t_0 + 0.6$, no matter on which VM instance we were to schedule the step. With the current system knowledge there would be no cost implications for scheduling the process

step on VM1.1 or VM1.2. Due to Term 4 of the objective function, which prefers scheduling process steps on VM instances that have a shorter remaining leasing duration, the system will prefer to schedule the process step on VM1.1.

We assume another incoming process request PX.Z.A at $t = t_0 + 0.5$. This process request can also be immediately scheduled on VM1.2, as it still offers enough resources and a long enough remaining execution duration.

If we had made a decision against Term 4 of our optimization model, to schedule PX.Y.A on VM1.2 at time $t = 0.1$, at $t = 0.5$ we would face the situation of having a nearly fully utilized VM1.2 that can not process the invocation of PX.Z.A and also having VM1.1 that offers enough resources, but does not have a long enough remaining leasing duration. In this case, we would be forced to lease VM1.1 for another BTU and pay the associated leasing costs, such as to schedule the invocation of PX.Z.A.

This shows the reasoning and importance of Term 4, which can lead to lower costs in future optimization rounds.

CHAPTER 5

Implementation

"Genius is one percent inspiration, ninety nine percent perspiration."

– Thomas Edison

"A person who never made a mistake never tried anything new."

– Albert Einstein

This chapter discusses the implementation of the approach presented in Chapter 4. First, we introduce the overall system structure and components of the designed BPMS. Second, we explain the realization of the optimization model as discussed in Section 4.4, differentiating between the reduced optimization model including corresponding implementation approaches, and the container-based scheduling algorithm extracted from the reduced optimization solution. Finally, we present our extension that allows the simulation of process executions while incorporating a time discretization model.

5.1 System Overview

The overall system architecture of our BPMS prototype is depicted in Figure 5.1. Clients can send process requests to the BPMS which schedules the process steps and allocates resources according to the optimization model presented in Section 4.4. Among the main responsibilities of the BPMS are the definition and execution of the task-scheduling and resource allocation plan, which defines (1) the leasing and releasing of backend VM instances, (2) the deployment of containers on the leased VMs, and (3) the invocation of process steps on the deployed containers. We extended and reconstructed the research prototype ViePEP [137] to offer the discussed functionalities. In the following, we explain the implemented solution for the main system components.

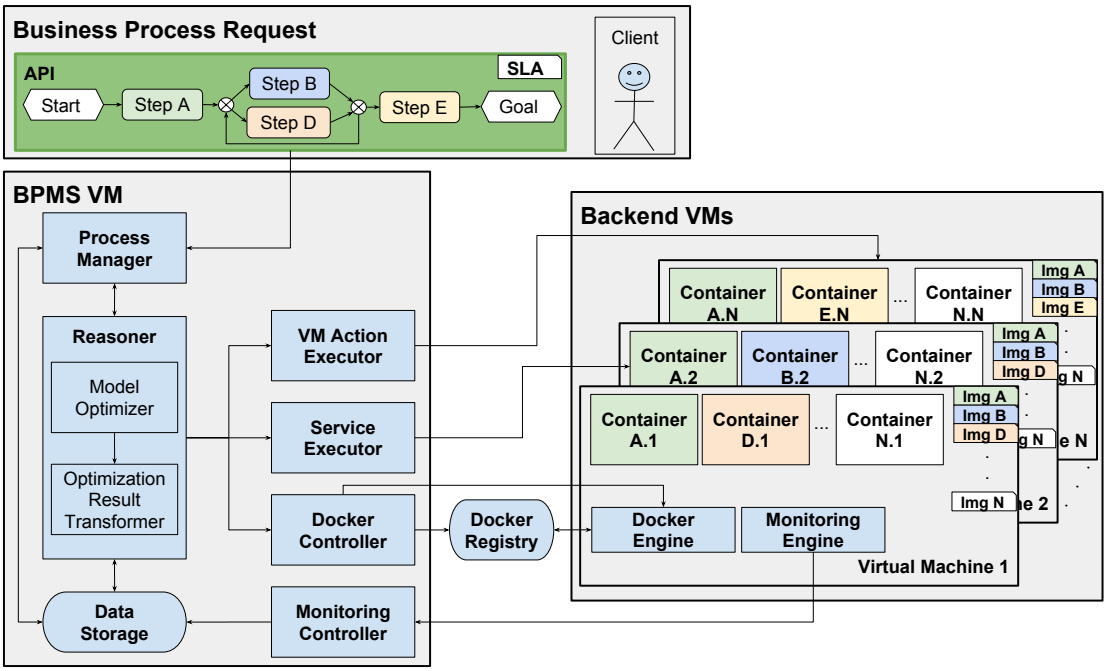


Figure 5.1: System Architecture Overview

5.1.1 Workflow Definition

Clients can choose from different process models and create multiple process instances composed of different services. Process execution requests with execution deadline constraints and penalty cost agreements are sent to the BPMS system. The BPMS provides a REST Application Programming Interface (API) that is called from Java using an HTTP client that sends the defined process instances as XML objects.

5.1.2 Process Manager

The process manager component is part of the BPMS and provides the REST API that receives the incoming process requests, transforms the XML objects into Java objects representing the complex elastic processes to be instantiated, and loads all process information into the data storage. This component is responsible for launching the reasoner whenever new process requests arrive or single process steps of process instances that are being executed are finished.

5.1.3 Data Storage

We make use of an in-memory caching system for a fast application performance. Details about finished workflow instances are persisted in a MySQL database.

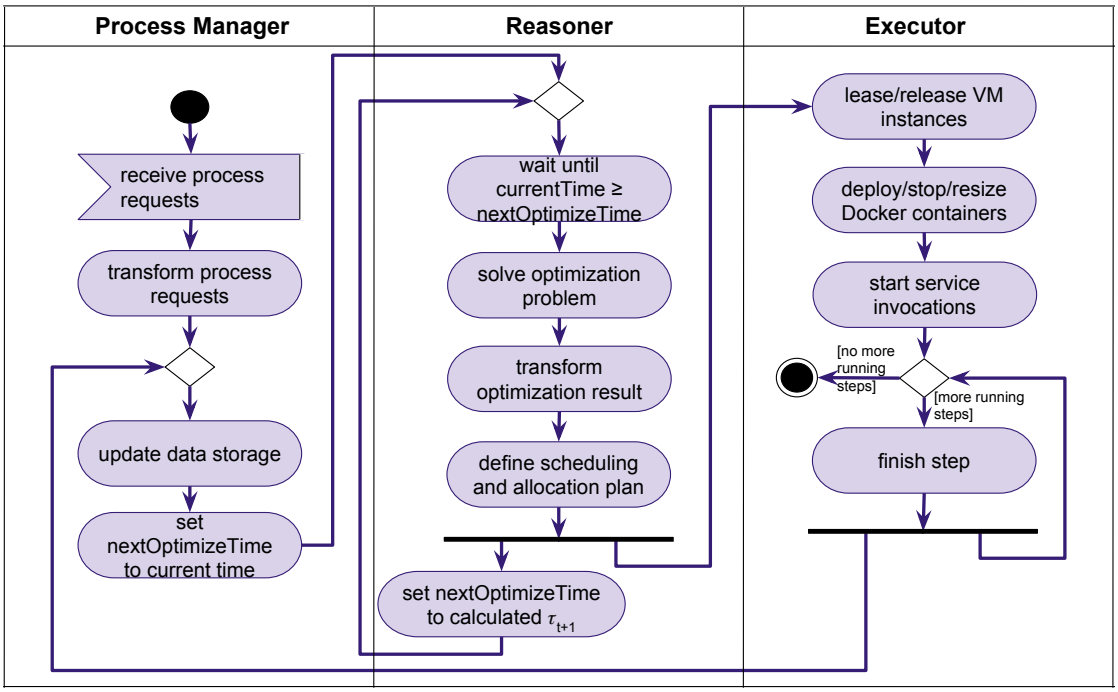


Figure 5.2: Triggering the Reasoner

5.1.4 Triggering the Reasoner

The Reasoner Component that calculates the optimized task-scheduling and resource allocation plan can be triggered by different events as shown in the UML activity diagram in Figure 5.2.

- When the process manager receives new requests, the reasoner is triggered to calculate a scheduling plan for the first schedulable process steps of the incoming requests. This is achieved by setting the $nextOptimizeTime$ to the current time. Previously schedulable but postponed steps are also considered in the scheduling decision.
- Whenever the execution of a process step is finished and leased resources are freed up, the reasoner is triggered to calculate further scheduling decisions for schedulable process steps. Such steps can either be the subsequent steps of the just finished process step or previously postponed process steps that have a distant deadline.
- Additionally, the reasoner component calculates the next best time to trigger the optimization during an optimization round based on the information available at that time. The calculated next optimization time τ_{t+1} is especially necessary for finding the best time to calculate a scheduling for process steps that can already be executed in τ_t but for which the scheduling decision and execution was postponed into the future. It should be noted that if the $nextOptimizeTime$ is set to an earlier point in time by subsequent actions, the calculated decision of starting the optimization at τ_{t+1} is overwritten and a new τ_{t+2} will be calculated during the newly triggered optimization round.

5.1.5 Reasoner Component

The reasoner component is responsible for calculating the optimized task-scheduling and resource allocation plan and is mainly split into two components, as can be seen in Figure 5.1. The model optimizer solves a reduced optimization model that schedules steps on VM instances while considering properties of the containerized architecture. The result of the model optimizer is transformed into a container-based resource allocation and scheduling plan by the optimization result transformer component. In Section 5.2 we present our implemented solution of the model optimizer and the reduced optimization problem in more detail. We define the reduced optimization problem using MILP and calculate a solution by utilizing CPLEX as our MILP solver. The optimization component can be implemented using different approaches, therefore, we also discuss an alternative evolutionary-based implementation approach for the reduced problem. In Section 5.3 we explain the implementation of the optimization result transformer and the creation of the container-based execution plan.

5.1.6 Process Execution

As soon as the reasoner creates the container based resource allocation and scheduling plan, the BPMS needs to execute it. In Figure 5.3 we depict the execution sequence of incoming process requests. The calculated execution plan that indicates which steps to place on which container instance and which container instance to place on which VM instance is handed over to the Execution Controller which then generates maps of VMs and their deployed containers, as well as the scheduled service invocations per container instance. The Execution Controller is also responsible for setting the container configuration, including the minimum guaranteed resources available to single container instances as to fulfill the scheduled service invocations. The tasks of the Execution Controller can be regarded as the last transformation step before the actual execution is initiated. We will go into more detail about the optimization and transformation steps in the subsequent Sections 5.2 and 5.3. Once the scheduling plan is fully transformed, the Execution Controller initiates the lease of all VM instances that are not yet leased and running, as defined by the scheduling plan. Once a VM instance is up and running, the further scheduling of containers and service invocations can be initiated.

VM Action Executor

The VM Action Executor leases new VM instances or extends the lease of existing instances according to the scheduling plan. The component is also responsible for releasing VM instances at the end of their BTU. Furthermore, it initiates the deployment of the needed Docker containers with the needed configuration by initating the Docker Controller.

Docker Controller

The Docker Controller pulls the required Docker images from the Docker registry onto the running VM instances as required and makes use of the Docker Engine on the running VM instances to start, stop or resize container instances such that they correspond to the configurations as defined by the scheduling plan.

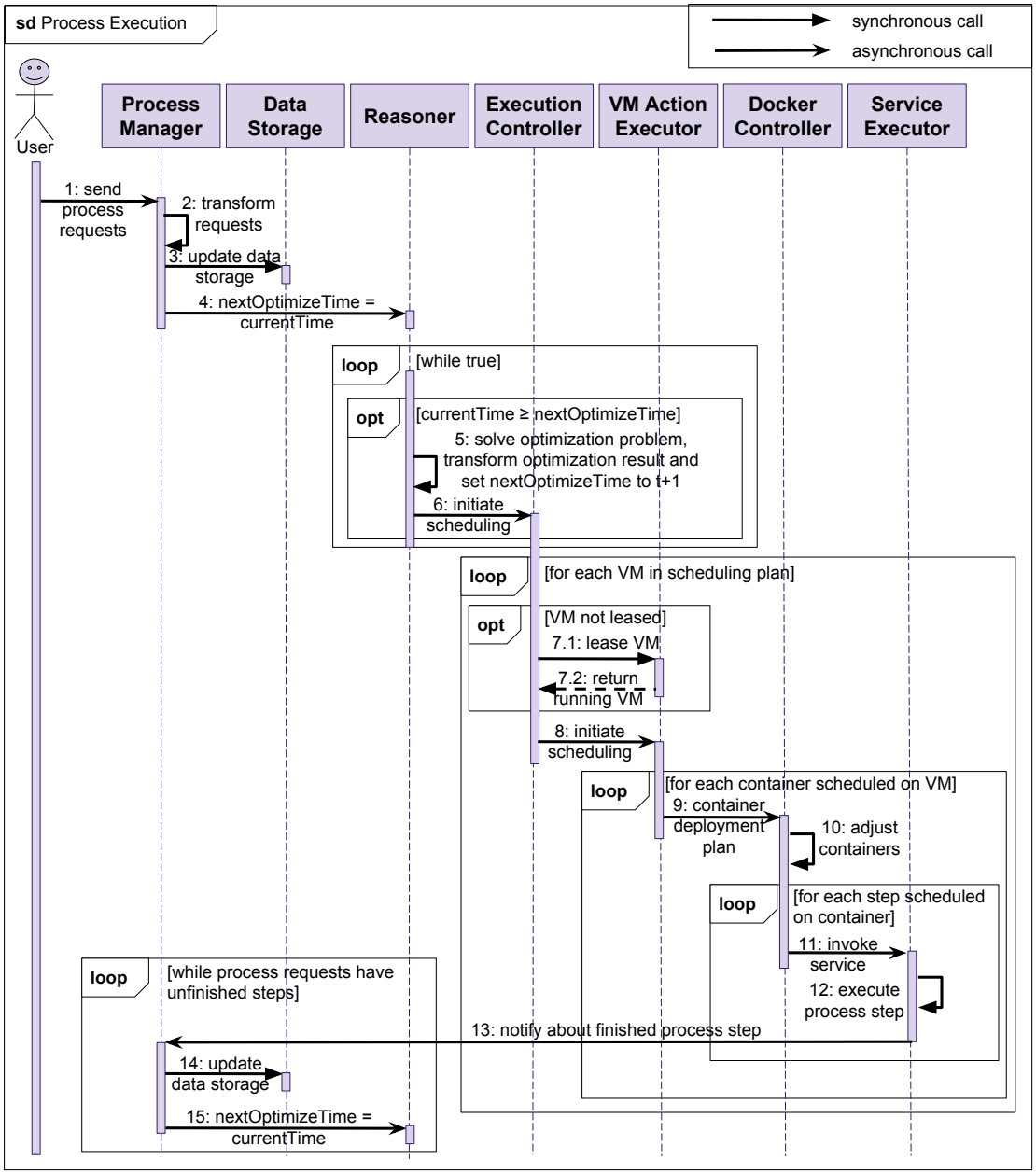


Figure 5.3: Execute Requests

Service Executor

Once a container is up and running according to the configuration, the scheduled process steps can be invoked and executed by the Service Executor. After each service invocation, the Service Executor notifies the Process Manager Component, so it can update the Data Storage and trigger a new optimization round by setting the $nextOptimizeTime$ variable to the current time.

5.2 Implementation of the Reduced Optimization Model

In Section 4.4, we introduced the general optimization model for task-scheduling and resource allocation of elastic processes in container-based cloud environments using MILP. We also discussed advantages of realizing the scheduling approach by making use of a reduced optimization model that first schedules tasks on VM instances and afterwards computes a corresponding container placement.

In this section, we introduce two possible implementations for the reduced model. First, we present a MILP-based solution that can be solved using a MILP solver. Second, we discuss another possible implementation of the reduced model using a genetic algorithm-based approach. In the course of this thesis, we only implement the MILP-based solution. The presented solutions correspond to different possible implementations of the optimization component of our BPMS as shown in our system's architecture in Figure 5.1.

The solution of the reduced model, as described in this section, is transformed to the scheduling plan as presented in Section 5.3. The result of the two reasoning steps, meaning, the solution to a reduced model, and the transformation step, generate a solution for the full optimization problem as presented in 5.2.1, which corresponds to the solution for one optimization round as offered by the reasoner component and executed by the respective execution components.

5.2.1 Mathematical Programming Based Implementation

This work focuses on extending existing MILP-based exact optimization approaches for scheduling and resource allocation of elastic processes on hypervisor-based cloud infrastructures [63] and single applications in Docker-based cloud environments [65]. To achieve the functionality described in Section 4.4 and to overcome the discussed limitations that arise when directly solving the general optimization model using a MILP solver, we introduce a reduced MILP problem, making use of the assumption that only one container of a certain service type shall be deployed on a VM instance. This assumption has already been made by related work [63]. The reduced MILP problem directly schedules process steps on VM instances, while considering the current information about the deployed containers. The details about the container instances are afterwards added to the scheduling plan during the transformation step. Generally speaking, the transformation will make sure that on each VM instance, on which process steps of a certain service type are scheduled, one container instance of the same service type will be deployed, which guarantees the exact resources, as needed by the scheduled service invocations on the VM.

Main Modifications of the General Model to the Reduced Model

In the following, we explain how we transform the general optimization model from Section 4.4 to a reduced model that directly schedules process steps on VM instances. The resulting reduced optimization Model can be found in Appendix A. We exchange the previous decision variable $x_{(j_{ip},c_{st},t)}$, which referred to scheduling of process steps on certain container instances, with the variable $x_{(j_{ip},k_v,t)}$, which refers to the direct scheduling of process steps on VM instances. We also eliminate all decisions for variables $a_{(c_{st},k_v,t)}$ which describe the container placement.

Reduced Objective Function

The terms of the reduced objective function remain the same for Term 1, Term 2, and Term 5 as described for the general objective function in Equation 4.1.

The purpose behind Term 3 as presented in Equation 4.1 remains the same, meaning we still want to minimize the deployment time of containers, but instead of directly considering the scheduling of container instances, we consider the placement of service steps on the VM instance and use the service type of the scheduled process steps to calculate the potential deployment time for container instances that are generated during the transformation step. The new term has the form $\sum_{p \in P} \sum_{i_p \in I_p} \sum_{j^*_{i_p} \in J_{i_p}} \sum_{v \in V} \sum_{k_v \in K_v} \left(\omega_z * (1 - z_{(st_j,k_v,t)}) * x_{(j_{i_p},k_v,t)}\right)$. Again, as the term is minimized, the system prefers choosing VM instances for assigning service invocations where the cache for deploying the needed container instance already exists.

Also, Term 4 as presented in Equation 4.1 is modified to serve the same purpose. To maximize the future resource supply of already leased VM instances, we now give a preference to directly scheduling service invocations on VMs with a shorter remaining leasing duration. The new term has the form $\sum_{p \in P} \sum_{i_p \in I_p} \sum_{j^*_{i_p} \in J_{i_p}} \sum_{v \in V} \sum_{k_v \in K_v} \left(\omega_d * d_{(k_v,t)} * x_{(j_{i_p},k_v,t)}\right)$.

Term 7 as presented in Equation 4.1 is only slightly modified to serve the same purpose, meaning it still aims at maximizing the importance of scheduled service invocations. Instead of considering the variable $x_{(j_{i_p},c_{st_j},t)}$ we now consider the variable $x_{(j_{i_p},k_v,t)}$. The new term has the form $\sum_{p \in P} \sum_{i_p \in I_p} \sum_{j^*_{i_p} \in J_{i_p}} \sum_{v \in V} \sum_{k_v \in K_v} \left(\omega_{DL} * (DL_{j^*_{i_p}} - \tau_t) * x_{(j_{i_p},k_v,t)}\right)$.

Term 6 as presented in Equation 4.1 is eliminated from the reduced model, as the sum of unused container resources is minimized in the transformation step, where containers are assigned just as many resources, as the scheduled service invocations actually require.

Reduced Problem Constraints

Many of the constraints needed for the reduced objective function remain the same or are only slightly modified, compared to the general MILP problem definition, while a large number of constraints can now be eliminated.

Constraints and Equations 4.2 to 4.12 remain the same, with slight modifications of Equations 4.6 to 4.10, where we simply need to exchange the decision variable $x_{(j_{i_p},c_{st_j},t)}$ by the now used decision variable $x_{(j_{i_p},k_v,t)}$. Equation 4.10 therefore now needs to build the sum over VM instances instead of over all container instances and the resulting equation has the form as shown in Equation A.10 in Appendix A.

Constraints 4.13 and 4.14 need to be modified such that they consider for each VM instance k_v the sum of CPU and RAM resources required by all service invocations that either already run or are scheduled to run in any container on the VM instance in τ_t. These resources need to be smaller or equal to the resource supply in terms of CPU (s^C_v) and RAM (s^R_v) that is offered by VM instances of type v. To achieve this, we simply need to exchange the decision variable $x_{(j_{i_p},c_{st_j},t)}$ by the now used decision variable $x_{(j_{i_p},k_v,t)}$ on the left side of the Constraints 4.13 and 4.14 and exchange the offered resources of the individually considered container instances ($s^C_{c_{st_j}}$ and $s^R_{c_{st_j}}$) by the offered VM resources (s^C_v and s^R_v) on the right side of the Constraints. These changes are expressed by Equation A.13 and A.14 as shown in Appendix A.

Constraints 4.15 to 4.18 from the general optimization problem can now be eliminated, as we do not consider the container instances in the reduced model and the fulfillment of the constraints is later on assured by the transformation step.

Constraints 4.19 and 4.20 need to be modified such that they define for each VM instance $v \in V, k_v \in K_v$ the remaining free capacities in terms of CPU and RAM that are not directly allocated to service instances (which are later packed into container instances). The resulting change is indicated by Constraint A.15 and A.16 in Appendix A.

Constraints 4.21 and 4.22 from the general optimization problem remain the same also for the reduced model, as we still need the definition of the same helper variables.

Constraint 4.23 is modified such that it expresses that a VM instance k_v will be leased or is running if service invocations are to be placed (within later defined containers) on it. The change is indicated by Constraint A.19 in Appendix A.

Constraints 4.24 to 4.29 can all be eliminated, as the details about container instances are not considered in the reduced model and the fulfillment of the constraints is again assured later by the transformation step.

Constraint 4.30 is only slightly changed in the reduced model, as we again do not need to consider information about container instances and can remove the decision variable $a_{(c_{st},k_v,t)}$ from the constraint. This change of Constraint 4.30 also eliminates Constraints 4.31 to 4.33 as we no longer need to perform the linearization. The simple change of Constraint 4.30 is shown in Constraint A.20 in Appendix A.

Constraint 4.34 remains the same, we only need to remove the container instance c_{st} from the indices, as we do not consider it. Also, Constraint 4.35 remains unchanged.

Constraint 4.36 needs to be changed such that it demands each service invocation to be scheduled on only one VM instance instead of one container instance. The resulting change is expressed in Constraint A.23 in Appendix A.

Constraint 4.37 can again be fully eliminated, as we do not need to consider the container details in the reduced optimization model.

Constraint 4.38 can be changed such that the steps currently running on a VM instance remain on the VM instance ($x_{(j_{ip}^{run},k_v,t)} = 1$). The related Constraint 4.39 can again be removed for the reduced model.

Constraint 4.40 only needs to be changed as to consider the variable $x_{(j_{ip},k_v,t)}$ instead of the container-based variable $x_{(j_{ip},c_{st},t)}$. Constraint 4.41 is eliminated as the decision variable $a_{(c_{st},k_v,t)}$ is no longer considered. Constraint 4.42 to 4.45 remain the same and Constraint 4.46 and 4.47 are eliminated as the helper variables are not used in the reduced model.

Solving the problem

The reduced optimization problem as fully defined in Appendix A is implemented using Java ILP[1], which provides a Java interface to MILP Solvers. To solve the problem, we utilize IBM CPLEX[2]. Compared to the general optimization model, computing an optimal solution to the reduced model involves a significantly lower computational effort for increasing problem sizes when using a MILP solver.

A solution to the reduced problem described using MILP can also be modeled and solved using evolutionary-based algorithms, as we discuss in the following section.

5.2.2 Genetic Algorithm-Based Heuristic

The described problem can be formulated and solved using evolutionary-based algorithms, like, for example, genetic algorithms. Such algorithms do not guarantee to find an optimal solution, but, when well designed, they can lead to good solutions in a relatively small amount of time. Especially for large and complex problems, it is often desirable to accept a good trade off between the solution quality and the runtime of algorithms. Genetic algorithms can be utilized as a global search method and are helpful for functions with multiple local optima. Nevertheless, when more specialized algorithms exist for a problem, genetic algorithms may not be the best optimization tool. In this subsection, we will introduce a possible general design of a heuristic based on genetic algorithms for solving our task-scheduling and resource allocation problem. Note, however, that the main focus of this work is on extending existing VM-based exact optimization algorithms for our described purpose of running services in isolated lightweight container instances. Therefore the evolutionary-based approach will not go into details about the algorithm configuration and we will not present an according evaluation.

Main components of the genetic algorithm

Genetic algorithms are based on Darwin's idea of evolution by the principles of natural selection, which are [110]:

- **Variation**: A variety of traits is present in the population.
- **Selection**: A mechanism exists, that selects the "fittest" individuals of a population that will reproduce and pass their genetic information on to future generations. This principle is often also known as "survival of the fittest".
- **Heredity**: A process exists, by which children inherit genetic properties of their parents.

Figure 5.4 shows the main components of our designed genetic algorithm. The algorithm starts by initializing a population of possible solutions, evaluates their fitness, performs a selection and reproduction process with crossover and mutation operations to generate a new population, until a fitting solution is found. In the following, we go into the details of every phase and the necessary steps.

[1] http://javailp.sourceforge.net/

[2] http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

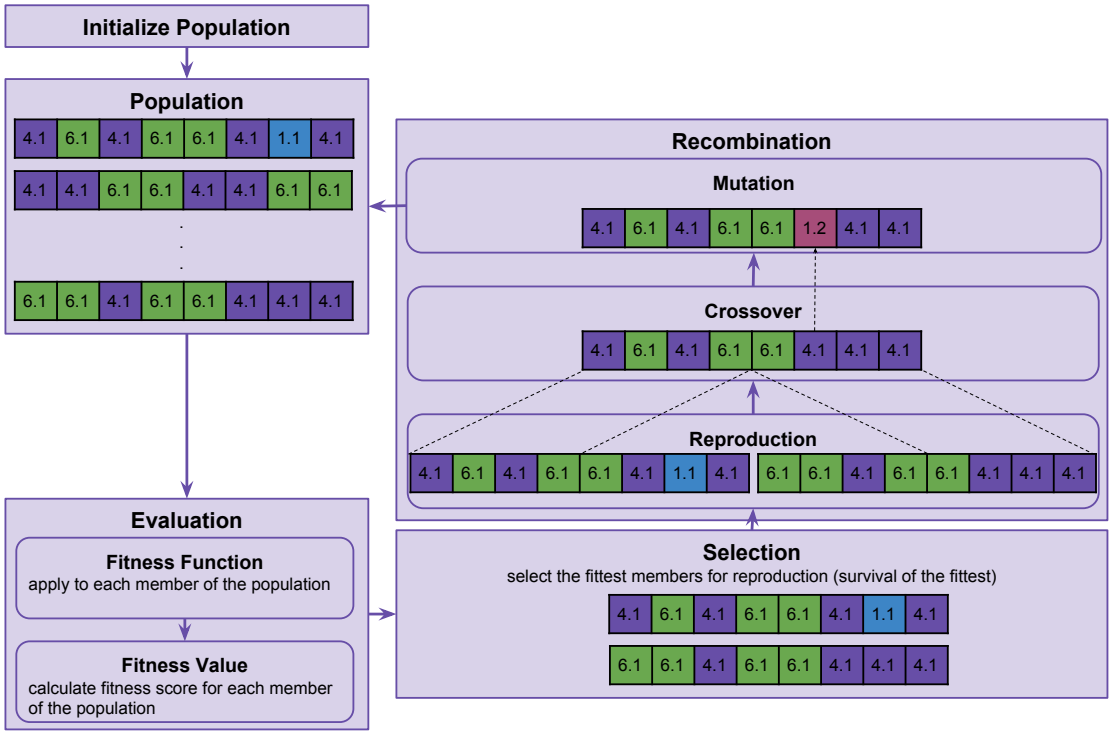


Figure 5.4: Genetic Algorithm Evolution

Population Encoding

Defining the encoding is one of the most important steps when designing a genetic algorithm. In our case we choose a value encoding, our values being the different leasable VM names, and the chromosome length corresponding to the number of process steps that may be scheduled in an optimization round. In other words, each slot in a chromosome as shown in Figure 5.4 corresponds to a process step that may be scheduled during the optimization round, and the encoding indicates on which VM the process step should be executed. This algorithm has some similarities to the MILP-based solution we explained in the previous Subsection 5.2.1 and is based on the same assumptions. Again, this encoding directly decides on which VM to schedule a process step, but we have to take container-based information into account. The corresponding containers will afterwards be created such that they can fulfill the designed scheduling. If the goal is to drop Constraint 4.48 (i.e., allow multiple containers per VM for a single service type), then the linear value encoding is not sufficient and we suggest using a tree encoding, considering containers on one level and VMs on another level.

Initialization of a Population

The population needs to be initialized with N different possible chromosomes that correspond to possible scheduling solutions. The initial population may be created by randomly assigning any leasable VM instance to the slots. To achieve better performing solutions, the initialization

could make sure to only create feasible solution chromosomes, e.g., make sure that not more process steps than the actual capacity of a VM allows to execute are assigned to a VM instance.

Fitness Function

The fitness function needs to calculate a fitness score for each chromosome in the population. A fitness score should be high for chromosomes that lead to low scheduling costs and can be calculated by using similar but normalized equations as the ones we use for the objective function of our reduced optimization problem in Subsection 5.2.1, while leaving out the summation of all possibilities and only considering one combination as indicated by the chromosome. Values for the decision variables $x_{(j_{i_p},k_v,t)} \in \{0,1\}$ of the optimization model can be mapped to our chromosome encoding. The fitness in constrained optimization problems can generally be measured by a solution's feasibility and the value of its objective function.

Selection Strategy

The chromosomes with the best fitness scores can be chosen in combination with a probability function. The chosen chromosomes are used for reproduction and creation of a new population. In Figure 5.4 we only show the selection of two parents for illustrative reasons, but the selection strategy, the actual number of needed parents, and the required population size have to be defined in future work.

Crossover Operation

Different crossover operations may be performed using the selected parents. In Figure 5.4 we show a simple single point crossover operation, but future work on the topic may also use other forms of crossover operations, like two-point, uniform, or arithmetic crossovers.

Mutation Operation

After the crossover operation, the new chromosomes can be mutated. The exact mutation algorithm also has to be defined in future work. At this stage, we would design the mutation stage such that the resulting chromosomes form feasible solutions for our optimization domain.

New Population

At the end of the reproduction algorithms the necessary amount of children have been generated and combined to a new population, which replace the old population.

Exit Condition

The algorithm is repeated until a good enough solution is found in the population. Generally the fitness score of the overall population should increase over time. A possible exit condition might be, that no improvements in the population could be made for a certain number of rounds.

5.3 Transformation of the Reduced Model Results for Container-based Execution

Once a solution for the reduced optimization problem that schedules process steps on VM instances is generated, we need to transform the result and create the final scheduling plan which also considers the containers for execution. The resulting solution corresponds to the overall optimization problem as discussed in Section 4.4.

5.3.1 General Transformation of the Reduced Model Solution

The transformation as described in Algorithm 5.1 can generally be applied to the outcome of any solution approach utilized to solve the reduced optimization problem.

To transform the result of the reduced optimization model to a result for the general optimization problem we need to iterate over all VM instances k_v of the reduced scheduling plan and extract all process steps that are scheduled for execution on a VM instance. For all process steps j_{i_p} of the same service type st a container instance c_{st} is defined (line 7) with a configuration that makes sure that the containers minimum resource supply is set to be exactly as large as the sum of scheduled service invocations of type st on k_v (line 13). We can then exchange the variable $x_{(j_{i_p},k_v,t)}$ from the reduced model solution for the newly introduced variables $x_{(j_{i_p},c_{st},t)}$ and $a_{(c_{st},k_vt)}$ which we set to 1 (lines 11, 13). This means, our scheduling plan schedules all service invocations j_{i_p} of service type st that were scheduled on k_v on the newly defined container instance c_{st}, while the container instance c_{st} is scheduled on k_v. This transformation, when starting with an optimal solution for the reduced problem, guarantees an optimal solution for the full optimization approach as defined in 5.2.1.

Algorithm 5.1: Transformation of the Reduced Solution

```
1  forall the k_v in the reduced scheduling plan do
2      VMServiceMap ← new Map(st, List(j_ip));
3      forall the j_ip where x_(j_ip,k_v,t) = 1 do
4          VMServiceMap ← (st_j, j_ip)
5      end
6      forall the st in VMServiceMap.getKeys() do
7          c_st ← new Container(st);
8          containerSize ← 0;
9          forall the j_ip in VMServiceMap.getValues(st) do
10             containerSize ← (containerSize + j_ip.getNeededResourceSize());
11             x_(j_ip,c_st,t) ← 1;
12         end
13         c_st.setSize(containerSize);
14         a_(c_st,k_vt) ← 1;
15     end
16 end
```

5.3.2 Executing the Transformed Scheduling Plan

When executing the transformed scheduling plan, the main tasks of the BPMS are to lease or release VM instances, to deploy, stop, or adjust containers, and to place the service invocations. For each task different rules have to be considered.

Executing the VM Plan

Whenever the decision variable $y_{(k_v,t)}$ takes a value > 0 the corresponding VM instance k_v needs to be leased or the lease of the VM instance needs to be extended by more BTUs. If during an optimization round no containers are scheduled on a VM instance k_v, the instance will still continue to be up and running until the end of its leasing duration.

Executing the Container Plan

The container instances are more flexible than the VM instances. Whenever the reasoner delivers the solution $a_{(c_{st},k_v,t)} = 1$ for a container instance c_{st}, either the deployment of the instance is initiated by the BPMS if a container of type st is not running on k_v at time t, or a currently running container of type st on k_v is potentially resized to match the defined container configuration. If a container instance of service type st on a VM instance k_v is currently running at τ_t but the scheduling plan does not define a corresponding container scheduling of the form $a_{(c_{st},k_v,t)} = 1$, the running container is immediately shut down to free up resources for other containers.

Executing the Service Invocation Plan

All scheduled service invocations $x_{(j_{i_p},c_{st},t)} = 1$ that are not yet running are invoked by the BPMS on the defined container c_{st}. As the scheduling plan also contains running steps, the execution of those service invocations that have already been scheduled for the first time in a previous optimization round remains unchanged.

5.4 Extending ViePEP

To implement our approach, we extended the research prototype ViePEP [137], such that it offers both, solely hypervisor-based scheduling as already used for previous work (e.g., [62] [63] [64]), as well as container-based scheduling as explained in the previous sections of this chapter.

ViePEP was built for the execution of elastic processes in public and private cloud environments and offers interfaces that allow to lease VMs. Furthermore, ViePEP also offers a simple simulation mode that provides the functionality to simulate the enactment of elastic processes on the cloud-based infrastructure.

We extended ViePEP such that it allows the deployment of Docker containers for placing service invocations. In the following, we explain our extension of ViePEP for allowing to switch between different optimization modes, changes in considering the time for the optimization models, and our enhancement of the original simulation mode, to allow for more sophisticated simulation scenarios by making use of a discrete time model.

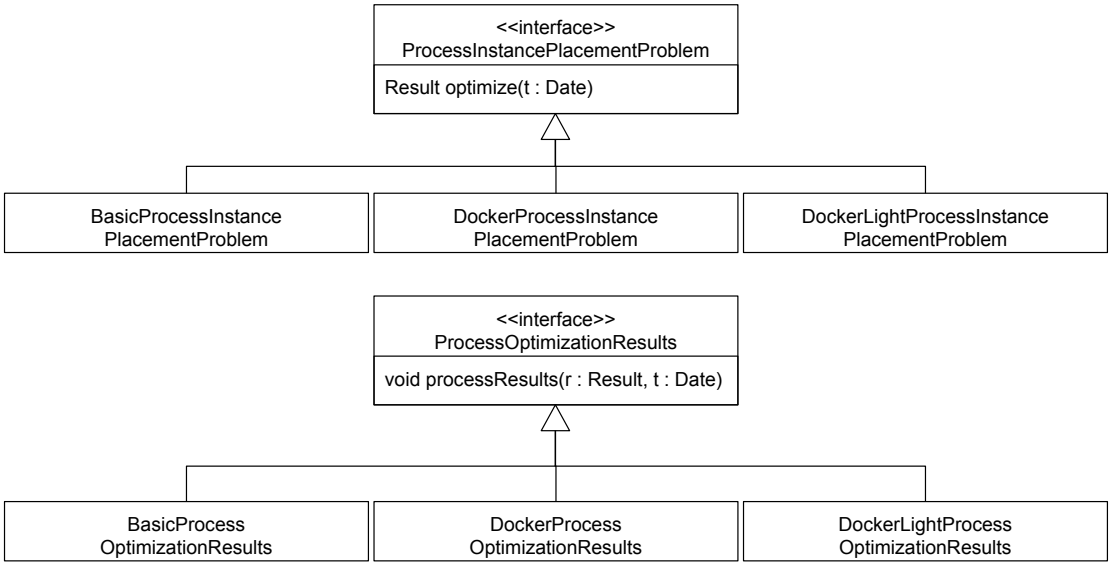


Figure 5.5: Interfaces for Different Reasoning and Result Processing Classes

5.4.1 Allowing Different Optimization Modes

Our extension of ViePEP allows an easy exchange of the reasoning component as well as the processing of the optimization results. ViePEP is implemented using the Spring application framework[3]. The class diagram in Figure 5.5 shows the different available implementations for solving optimization problems and processing the results. We set up the three Spring profiles *'basic'* for executing the hypervisor-based optimization problem following Hönisch et al. [63], *'docker'* for utilizing a MILP-based implementation of the full optimization problem without any further transformation as introduced in Section 4.4, and *'dockerLight'* for calculating the reduced optimization model and performing the discussed transformation.

The profiles can easily be changed by setting the active profile using the corresponding Spring property. The implementation of our approach as discussed in this chapter can be utilized by activating the profile *'dockerLight'* in the `application.properties` file (or via a command line parameter):

```
spring.profiles.active = dockerLight
```

Listing 5.1 shows the Spring configuration class for the profile *'dockerLight'*. Similar configurations also exist for the other profiles. The configuration makes sure that when the profile *'dockerLight'* is activated, Spring initializes and injects the beans corresponding to the chosen profile. This allows us to change ViePEP's used optimization approach (i.e., the used implementation for calculating a solution for an optimization problem and the processing of the optimization results) by simply changing the profile.

[3] https://projects.spring.io/spring-framework/

Listing 5.1: Using Profiles to Configure the Reasoning

```
1  @Configuration
2  @Profile("dockerLight")
3  @PropertySource(value = "application-docker.properties")
4  public class DockerLightOptimizerConfiguration {
5          @Bean
6          public ProcessInstancePlacementProblemService initializeParameters() {
7                  return new DockerLightProcessInstancePlacementProblemServiceImpl();
8          }
9          @Bean
10         public ProcessOptimizationResults processResults() {
11                 return new DockerLightProcessOptimizationResults();
12         }
13 }
```

5.4.2 Considering the Time in Optimization Models

CPLEX can be numerically sensitive when the difference between the smallest and the largest variables in the model is rather large [29]. This can lead to the calculation of wrong optimization results by the solver. In our optimization model, we determine the time at the beginning of an optimization round τ_t and try to find the next best time to start the following optimization round τ_{t+1}. The Java implementation of

```
new Date().getTime();
```

will deliver the number of milliseconds since January 1, 1970, 00:00:00 GMT (UNIX epoch). Compared to the other numbers used in the optimization this number is too large and also remains rather large when dividing the outcome by 1,000 and only considering the number of seconds since the UNIX epoch. To overcome this problem, we calculate with a closer epoch in our optimization model by using the constant $START\_EPOCH$ shown in Listing 5.2 which captures the number of seconds since the UNIX epoch when our BPMS is started.

Listing 5.2: Capturing the Start of the BPMS as a Start Epoch for Reasoning Calculations

```
1  public class Constants {
2          public static final long START_EPOCH = TimeUtil.now() / 1000;
3  }
```

The method call

```
TimeUtil.now()
```

will at this point deliver the value of

```
new Date().getTime();
```

The class $TimeUtil$ performs the time discretization of our system and we will explain its functionality in more detail in the following subsection.

Our optimization models receive the current time in milliseconds starting from the UNIX epoch as an input for τ_t, but immediately transform the time such that only the seconds that have passed since the start of the BPMS are considered. The transformed time is used as our τ_t for all calculations in the optimization model. The transformation is achieved as follows:

```
tau_t=new Date(((tau_t.getTime()/1000)-START_EPOCH)*1000);
```

Whenever the optimization model considers other times, for example, the deadline for process instances in Constraint A.2, the times need to be adjusted to match the newly introduced start epoch:

```
(workflowInstance.getDeadline()/1000)-START_EPOCH;
```

As the calculated τ_{t+1} is also adjusted to the newly introduced start epoch, after performing the optimization we need to retransform the calculated start time for the next round to the time in milliseconds since the UNIX epoch:

```
tau_t_1=(START_EPOCH+optimize.get("tau_t_1").longValue())*1000;
```

5.4.3 System Simulation with Time Discretization

We introduce a discrete time model for our system, to facilitate system simulations. The class `TimeUtil` is used throughout the BPMS for receiving the current time or performing `Thread.sleep(milli` operations.

To start the time discretization, the timer has to be activated, e.g., when starting the application, by calling `startTicking()` in `TimeUtil`, shown in Listing 5.3. The discrete time will start at the actual current time.

Listing 5.3: TimeUtil for Time Discretization - Start Ticking

```
1  public class TimeUtil extends Thread {
2      public static final AtomicLong TIME = new AtomicLong(-1);
3      public static final AtomicBoolean RUNNING = new AtomicBoolean();
4      [...]
5      public static void startTicking() {
6          if(TIME.get() < 0) {
7              TIME.set(new Date().getTime());
8          }
9          RUNNING.set(true);
10     }
11     [...]
12 }
```

The discrete Time is incremented as shown in Listing 5.5, according to the adjustable time ticking speed defined in Listing 5.4. The variable `TIME_INCREMENTS_MS` indicates the actual discrete time increment, while the variable `SEC_SLEEP_TIME_MS` indicates after how many actually passed real-time milliseconds the time increment should occur. The speed can be adjusted by using the method `setSpeed(speed)`. Demanding a speed of 1 is equivalent to setting the time ticker to tick in real-time, whereas demanding a speed of 20 sets the

time ticker to tick 20 times faster than real-time. When running our BPMS in simulation mode, which simulates the leasing of VM instances, the placement of Docker containers, and the execution of service invocations, we set the time ticking speed to 20 for performing the reasoners task-scheduling and resource allocation plan, but during optimization runs, we demand that the discrete time ticks in real-time, as to correctly register the fluctuating time needed to solve the optimization problem.

Listing 5.4: TimeUtil for Time Discretization - Set Speed

```
1  public class TimeUtil extends Thread {
2      public static final long TIME_INCREMENTS_MS = 100;
3      public static final AtomicLong SEC_SLEEP_TIME_MS = new AtomicLong(10);
4      [...]
5      public static void setRealTime() {
6          setSpeed(1);
7      }
8      public static void setFastTicking() {
9          setSpeed(20);
10     }
11     public static void setSpeed(double speed){
12         SEC_SLEEP_TIME_MS.set((long)(TIME_INCREMENTS_MS / speed));
13     }
14     [...]
15 }
```

Listing 5.5 shows how the time increment is performed. After each discrete time increment, all threads waiting for a certain change in the current time are notified. After the time increment, the timer thread sleeps for `SEC_SLEEP_TIME_MS` milliseconds before performing the next time increment. The method `doSleepSafe(millisec)` performs a real-time `Thread.sleep(millisec)` operation.

Listing 5.5: TimeUtil for Time Discretization - Increment Time

```
1  public class TimeUtil extends Thread {
2      [...]
3      @Override
4      public void run() {
5          [...]
6          while(true) {
7              if(RUNNING.get()) {
8                  synchronized (TIME) {
9                      long delta = TIME_INCREMENTS_MS;
10                     TIME.set(TIME.get() + delta);
11                     TIME.notifyAll();
12                 }
13             }
14             doSleepSafe(SEC_SLEEP_TIME_MS.get());
15         }
16     }
17     private void doSleepSafe(long millis) {
18         try {
19             Thread.sleep(millis);
20         } catch (InterruptedException e) {
21             throw new RuntimeException(e);
22         }
23     }
24     [...]
25 }
```

Every time the BPMS needs the current time, it calls `TimeUtil.now()` to either receive the discrete time when in simulation mode or the actual current time when simulation mode is turned off as shown in Listing 5.6.

Listing 5.6: TimeUtil for Time Discretization - Get Current System Time

```
1  public class TimeUtil extends Thread {
2      private static TimeUtil INSTANCE;
3      @Value("${simulate}")
4      private boolean simulate;
5      [...]
6      public static long now() {
7          [...]
8          if(!INSTANCE.simulate || TIME.get() < 0) {
9              return new Date().getTime();
10         }
11         return TIME.get();
12     }
13     [...]
14 }
```

The BPMS calls `TimeUtil.sleep(millisec)` shown in Listing 5.7 whenever it needs to perform a sleep operation, to either wait until the discrete time variable was incremented to the target time when in simulation mode, or to perform a real `Thread.sleep(millisec)` operation using the method `doSleepSafe(millisec)` introduced in Listing 5.5.

Listing 5.7: TimeUtil for Time Discretization - Sleep And Notify Waiting Threads

```
1  public class TimeUtil extends Thread {
2      [...]
3      public void sleep(long millis) {
4          [...]
5          if(!simulate) {
6              doSleepSafe(millis);
7              return;
8          }
9          long targetTime = now() + millis;
10         try {
11             long timeNow = now();
12             while(targetTime > timeNow) {
13                 [...]
14                 synchronized (TIME) {
15                     TIME.wait();
16                 }
17                 timeNow = now();
18             }
19         } catch (InterruptedException e) {
20             throw new RuntimeException(e);
21         }
22     }
23     [...]
24 }
```

In simulation mode, the optimization is started with the current simulated time as received by `TimeUtil.now()`. During the optimization run, the discrete time ticker is set to tick in real-time. Afterwards the time ticks faster than real-time to accelerate the simulation of the task-scheduling and resource allocation plan as shown in the sequence diagram in Figure 5.6.

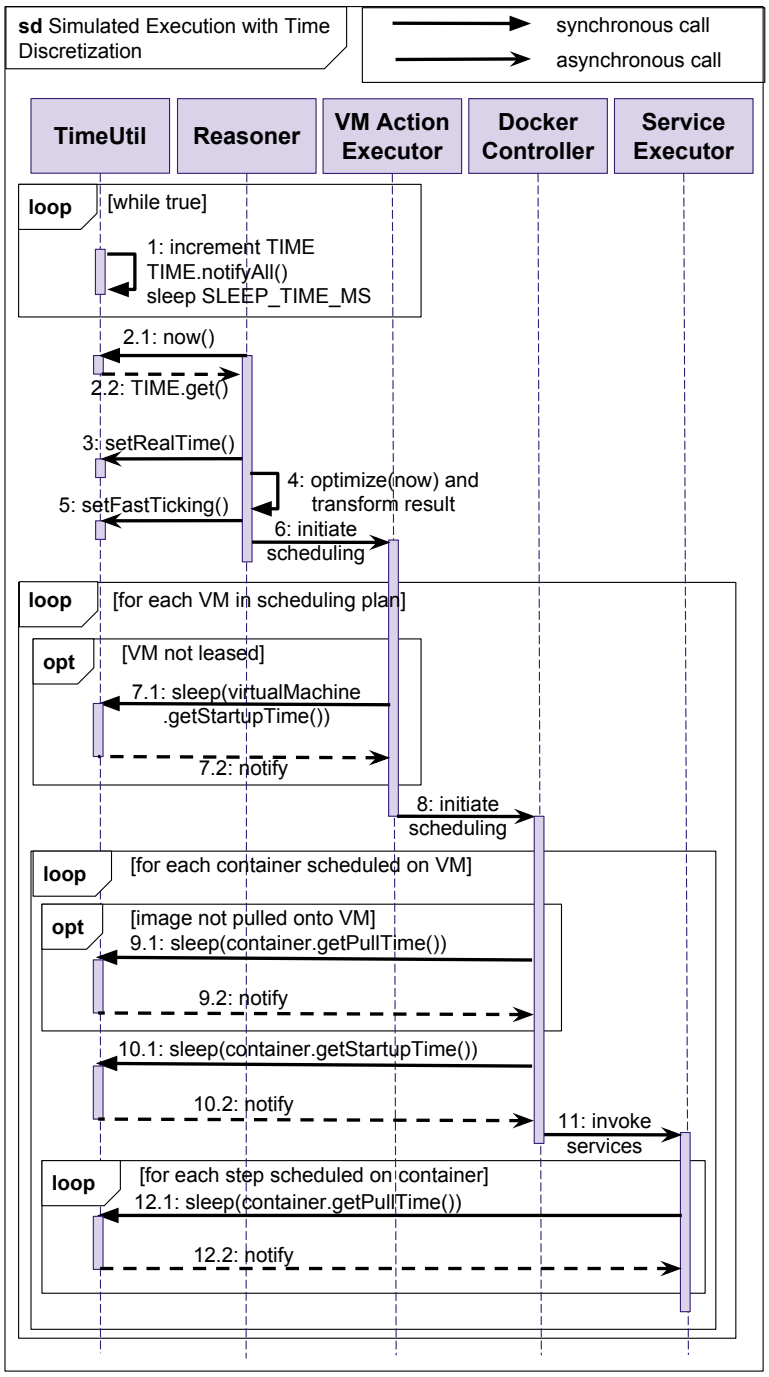


Figure 5.6: System Simulation with Time Discretization

CHAPTER 6

Evaluation and Discussion

"There are two possible outcomes: if the result confirms the hypothesis, then you've made a measurement. If the result is contrary to the hypothesis, then you've made a discovery. "

– Enrico Fermi

"Everybody is a genius. But if you judge a fish by its ability to climb a tree, it will live its whole life believing that it is stupid."

– Albert Einstein

In this chapter, we conduct a qualitative analysis of our approach by presenting an evaluation of the working example from Section 4.3. We discuss the execution as specified by the optimization model in Section 4.4 and as illustrated in Section 4.5. Additionally, we discuss a modified solution that allows for pre-scheduling decisions and point out important observations, some of which can be tackled in future work. As a comparison to previous work, we also discuss the execution of the working example from Section 4.3 using an hypervisor-based approach and contrast the advantages of container-based to hypervisor-based approaches.

Furthermore, we present a quantitative evaluation of the implementation of our approach as presented in Section 4.4 and Chapter 5. Therefore, we define and execute an experimental design. We discuss the evaluation settings and scenario, the used metrics and final results while comparing our approach to a baseline.

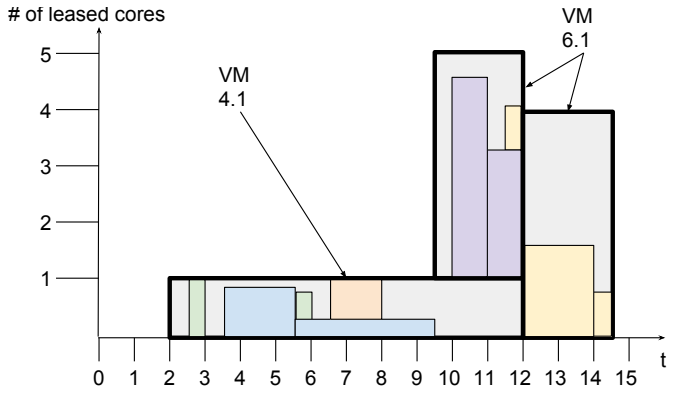


Figure 6.1: Number of Leased VM Cores and Utilization by Containers

6.1 Evaluation of the Working Example

For evaluating the working example, we consider the type and amount of leased resources as well as their utilization at each point in time and calculate costs that arise from the leasing and scheduling decisions.

6.1.1 Evaluation of the Presented Working Example

In Figure 6.1, we abstract from the execution of single process steps of the discussed working example and evaluate the number of leased VM cores and their utilization by deployed containers. The figure shows us that we leased two VMs. VM4.1 was leased for a total of two BTUs and VM6.1 was leased for the duration of one BTU. Each deployed container for executing service invocations of service type A (green), service type B (blue), service type C (purple), service type D (orange), and service type E (yellow) is marked in Figure 6.1. The assigned resources to containers correspond to the respective resource utilization for every point in time.

As we met all deadlines, we do not have to consider any penalty costs and can easily calculate the costs for the process executions by adding the costs of leasing VM4.1 for two BTUs and VM6.1 for one BTU. This amounts to a total cost of 25 units.

6.1.2 Evaluation of the Working Example with Pre-Scheduling

As can be seen from the working example, the designed system calculates a scheduling for process steps only if a scheduling is possible on available and free computational resources without incurring any additional leasing costs, while prioritizing steps with a close deadline, or if the worst-case analysis, based on the process deadline as well as the execution and deploy times of the step and all following steps, requires an immediate scheduling decision.

An alternative approach would be to schedule steps on newly leased computational resources before their worst-case scheduling deadline if a good enough resource utilization is calculated. To illustrate this, we look at our example scenario and make only one different pre-scheduling decision at $t = 3$, when the worst-case analysis of process instance P1.1 requires the scheduling

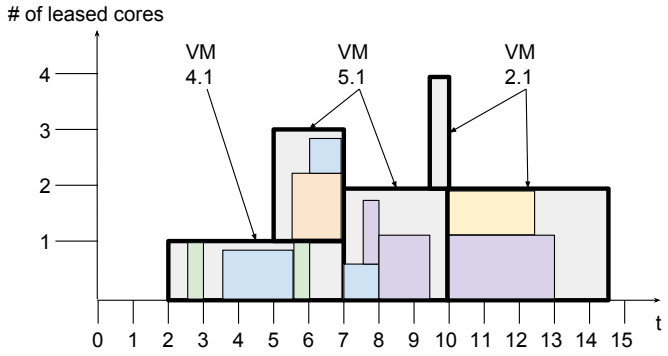


Figure 6.2: Number of leased VM cores and utilization by Containers with early scheduling of Step P1.1.D

of P1.1.B, we also pre-schedule the process step P1.1.D in the parallel split branch, although the process step in the parallel branch could and would be postponed at that stage by our presented optimization approach as discussed in Section 4.4. Except for this change, all other decisions are made following the exact same pattern as previously explained. Without going into the details for each subsequent scheduling decisions, this small change leads to a final system evaluation over time as demonstrated in Figure 6.2.

We can see that the decision to schedule P1.1.D at $t = 3$ leads to leasing a dual-core VM from the private cloud which is up and running at $t = 5$. The additional resource offered by VM5.1 leads to an earlier scheduling of process steps of type C and also eliminates the need to extend the lease of VM4.1. Furthermore, the process steps with close deadlines ($t \leq 20$) could be finished by $t = 12.5$. While our first described solution, as summarized in Figure 6.1, finishes all process instances, the changed example summarized in Figure 6.2 does not schedule the last process step P3.1.E with a distant deadline, as the system does not offer enough resources for its execution.

If the system does not receive any further process requests, the last process step P3.1.E of process P3.1 would be scheduled for execution at $t = 95$ according to its worst-case analysis. The system would choose the cheapest computational resource VM4.1, which would be up and running at $t = 97$. If other process requests enter in the future and require to lease computational resources before $t = 95$ and if enough resources are available, the process step might also be scheduled at an earlier point in time.

When only looking at the illustrated time frames of $t \leq 15$ we can see that the alternative with pre-scheduling leases less computational resources and offers a slightly better resource utilization than the originally presented scheduling approach. Nevertheless, also when disregarding that additional computational resources are needed for the execution of the last process step P3.1.E, the cost of all computational resources VM4.1 and VM5.1 from the private cloud and VM2.1 from the public cloud amounts to 28 units, which is higher than the previously calculated 25 units for finalizing all process instances.

6.2 Important Observations

When looking at the presented execution of our working example that uses our optimization approach and when regarding comparisons to other approaches we can make a number of important observations.

First of all, it should be noted that in the presented execution of our working example we made some simplifications, like not considering any time for starting, stopping and resizing container instances and most importantly we also did not consider any time for actually calculating the scheduling decision. The calculation of a solution for the MILP optimization problem will take some time which, depending on the problem size, may range from a few seconds to multiple minutes and has an impact on the overall system performance.

When abstracting from the simplifications, we can observe the following major points regarding the behavior of the system:

- **Optimization rounds**: Optimization rounds need to be initiated when changes in the system landscape occur. Such changes include newly arriving process requests and the finished execution of already scheduled process steps. Furthermore, the start of new optimization rounds is also a result of previous optimization rounds and especially also of postponing decisions.

- **Postponing optimization steps**: During optimization rounds, the system decides which of the next schedulable process steps to schedule on which container on the available VM resources. The system may decide to postpone the scheduling of some schedulable process steps into another optimization round if scheduling the steps would incur increased costs due to the requirement of leasing additional computational resources although the process deadline constraints would allow for a later scheduling without incurring penalty costs.

- **Optimization knowledge**: The optimizations are based on the system knowledge at the beginning of an optimization round and depend on previously made optimization decisions and system performances.

- **Rescheduling**: Sometimes the system is in a state where previously made scheduling decisions could easily be changed without having to consider any migrations, as the execution of the process steps could not be started yet. This can happen, for example, when a process step is scheduled for execution on a certain container on a VM instance and either the container image is still in the process of being pulled onto the machine or the VM instance is still in the process of booting up, while a new optimization round is initiated. This type of rescheduling can lead to some performance gains but is not further considered in our approach, as we are focusing on the advantages that come from the use of containers for task-scheduling and resource isolation compared to simple hypervisor-based approaches and the described rescheduling has not been considered in the related work discussed in Chapter 3.

- **Migration**: Furthermore, the optimization approach could benefit from considering migrations. Migrations would, for example, be helpful for utilizing remaining free resources on leased VM instances for service requests that run longer than the remaining leasing duration of the instances. In that case, the remaining leased resources could be utilized and at the end of the leasing duration, the running containers could be migrated onto other available resources, without having to extend the lease of an old and unneeded VM instance. Approaches considering migrations could also split the execution of service requests such that deadlines are considered but also resources are utilized more efficiently. Migrations come with a wide number of necessary considerations and additional overheads that a BPMS would need to take into account and that also go far beyond the scope of this thesis.

- **Pre-Scheduling**: Pre-scheduling refers to the scheduling of process steps before their scheduling deadline when performing a worst-case analysis. Our system performs pre-scheduling whenever the system offers sufficient computational resources for schedulable process steps which may be postponed into the future. This type of pre-scheduling does not incur any additional leasing costs and allows for a better utilization of the leased resources. Still, it has to be mentioned that the pre-scheduling decision is solely made using current information and optimizes the current system state, without considering any future implications.

  Another form of pre-scheduling would allow the scheduling of process steps that may still be postponed on computational resources which would incur additional leasing costs. A possible scenario would be if enough schedulable but postponable process steps were in line such that they could fully utilize a new leased instance. Yet, also in this special case, the pre-scheduling may lead to an overall worse outcome for the performance of the system over time, as the scheduled steps may, in future rounds, have been scheduled on otherwise free remaining computational resources, which were not foreseen in the current optimization round.

  An example of the pre-scheduling scenario that accepts leasing costs ahead of scheduling deadlines was already discussed in the evaluation of the working example execution in Section 6.1. As we could observe when comparing the system outcomes from Figure 6.1 and Figure 6.2, the pre-scheduling lead to higher execution costs, although one might have gained the impression of a better resource utilization. This shows that pre-scheduling decisions can often lead to worse outcomes when only considering the current system state, but might lead to better scheduling decisions when combined with prediction algorithms.

- **Prediction**: The presented approach considers the future in terms of making scheduling decisions that consider future deadlines, including decisions to postpone scheduling of process steps. When making scheduling decisions for a certain optimization round, the system analyzes the current utilization of its leased resources, without performing extended predictions about the systems future state. This may lead to some disadvantages. A good example for this would be when a process step PX.Y.A needs to be scheduled for execution at $t = x$ as it cannot be postponed any further without incurring penalty costs.

If at $t = x$ there are not enough computational resources available for scheduling the process step, the system will lease a new VM instance and pull the corresponding image for the process step. Therefore, the process step scheduled at $t = x$ will only be able to start its execution after the VM is up and running, the corresponding Docker image is pulled and a container is deployed at around $t = x + 2.5$. If some other process steps would finish their execution by say $t = x + 1$ and free up enough computational resources for the execution of the already scheduled process PX.Y.A, the decision at $t = x$ would have lead to a longer execution time for process instance PX.Y and might lead to higher total costs due to leasing the additional VM instance although it might have been prevented, if a more extended analysis of currently running steps and the future state would have been performed.

Yet in highly dynamic system landscapes a meaningful prediction would need to consider more than just the currently available information about scheduled and running process steps and the future state of currently known process instances and leased resources, as new process instances can arrive at any time and impact the future system state such that optimized decisions considering only the future execution of currently known process instances might lead to scheduling plans that turn out to be suboptimal in the future when new requests are processed. The use of predictive analytics and machine learning approaches might help in finding better overall system enactments over multiple time periods.

Advanced rescheduling, migration, pre-scheduling and prediction algorithms can help in developing cost-effective and efficient scheduling techniques, but all of them require a multitude of further considerations and the discussed related work in Chapter 3 lacks in solutions that consider such advanced scheduling methods for elastic processes. In this work, we focus on extending the related work on elastic processes to account for container-based architectures, therefore, our solution does not directly approach these important observations, although they should be considered in future work.

6.3 Advantages of Container-Based Approaches compared to Hypervisor-Based Approaches

In this section we compare the presented container-based scheduling approach to the solely hypervisor-based scheduling approach for elastic processes as presented by Hoenisch et. al [63] and explain the main strengths of container-based approaches with regard to scheduling decisions.

6.3.1 Execution of the Working Example using Hypervisor-Based Approaches

To compare the outcome of our container-based scheduling approach summarized in Figure 6.1 to hypervisor-based scheduling, we demonstrate the execution of the working example using the scheduling approach presented by Hoenisch et. al [63].

Scheduling Assumptions

The main scheduling assumptions and principles remain similar, with the following main differences:

- In the approach by Hoenisch et. al [63] software services are directly deployed on VM instances. Each VM instance hosts exactly one software service but a software service may be deployed on multiple VM instances.
- We again assume a VM startup time of two minutes. Instead of the time of 30 seconds that we calculated to pull a Docker image onto a VM, we now have to consider the time it takes to deploy a software service on a VM instance. We again assume the deployment time to be 30 seconds.

We use the same VM instances, software services, processes and process requests as already introduced in Section 4.3. In Figure 6.3 we again abstract from the actual execution of the single process steps of the working example and evaluate the number of leased VM cores and their utilization by the execution of service invocations. The boxes with bold black frames represent the leased VM instances for each point in time. As a VM instance can only process requests of one service type during an uninterrupted leasing period, the shade of the boxes indicates the service types the VM instances can execute. We use the same color coding as in the previous chapters, meaning green for service type A, blue for service type B, purple for service type C, orange for service type D, and yellow for service type E. The smaller filled boxes within the VM instances indicate the utilization of the VMs by executed service invocations at each point in time.

Scheduling Results

Looking at Figure 6.3, it is easy to see that we had to lease substantially more computational resources for the execution of the process instances. The utilization of each leased VM is very low, as each VM instance only allows the scheduling of one certain service type until it is released and the number of incoming requests in our working example is rather low.

Looking at the scheduling decisions in more detail we can observe that single-core VMs were leased for a total of six BTUs and one quad-core VM was leased for one BTU. The cheapest single-core VM (VM4.1) could be leased three times, the first time accepting service requests of tape A, the second time accepting service requests of type D, the third time accepting service requests of type E.

Also when abstracting from the container-based solution, we can clearly observe the same problems that arise from a lack of process predictions as discussed in Section 6.2 in this VM-based approach. For example, VM1.2 for service type B was unnecessarily leased from $t = 7.5$ to $t = 12.5$. The reason for this was that the scheduling decision was made at $t = 5.5$, when VM1.1 for service type B was nearly fully utilized and the system had to schedule the next process step 3.2.B due to the strict process deadline and worst-case analysis. The system did not consider that the already leased VM1.1 for service type B would soon offer the required resources, without the need to lease an additional VM instance. The last process step P3.1.E was

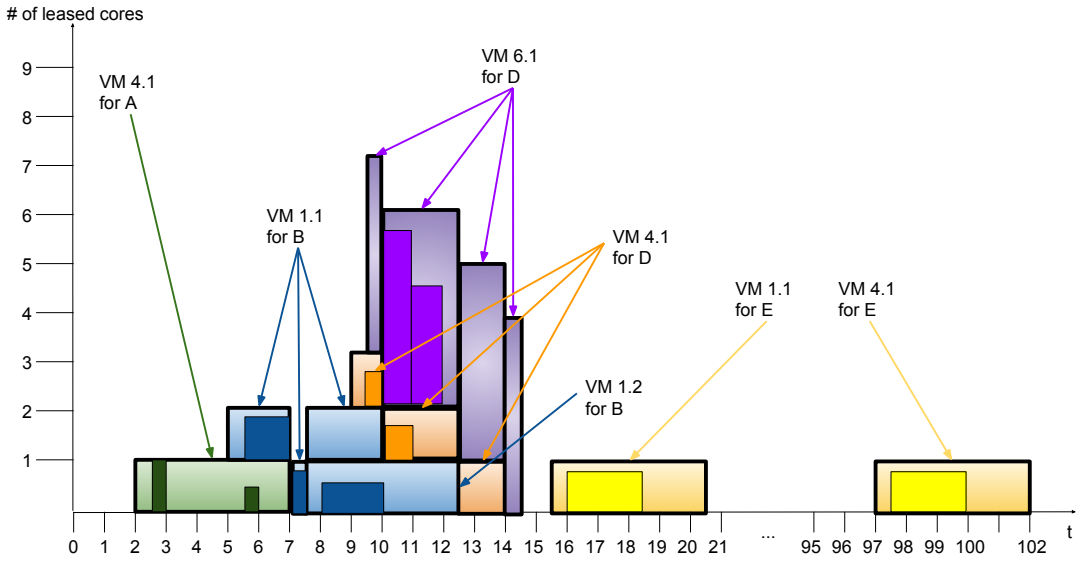


Figure 6.3: Number of Leased VM Cores and Utilization for Approach without Containers

scheduled at its latest possible scheduling deadline, because of the same situation we discussed in the evaluation of the working example with prescheduling in Subsection 6.1.2, assuming that no other process requests had to be executed before $t = 95$.

The total cost of this hypervisor-based task-scheduling and resource allocation approach is comprised of the total cost for leasing the resources depicted in Figure 6.3. VM4.1 was leased for a total of three BTUs, each to a cost of 5 units. Public cloud VMs of type 1 (VM1.1 and VM1.2) cost 8 units and were leased for a total of three BTU. Furthermore, one VM of type 6 was leased for one BTU at the cost of 15 units. This amounts to a total leasing cost of 54 units. This is more than twice the cost of the presented Docker-based approach of only 25 units.

6.3.2 Strengths of Container-Based Approaches

Next to the many advantages that are related to utilizing lightweight containerized technologies, as discussed in Subsection 2.1.5 and Chapter 3, like, for example, in the context of DevOps, the use of containers comes with the following advantages with regard to the presented task-scheduling and resource allocation approach for elastic processes:

- As lightweight containers can be created instantly from existing images, on running VMs, a container-based system can better utilize the leased resources, by starting and stopping containers on demand without wasting many resources for scheduling service steps that are not executed frequently on separately leased VM instances.

- While the existing hypervisor-based approaches require a full VM instance for each service type, in container-based approaches service types with low utilizations can easily

share VM instances. Larger VM instances, that offer resources for a lower price than multiple small instances, can be leased for scheduling a large number of container instances for the execution of process steps that require a small amount of computational resources.

- The sharing of underlying VM instances by multiple containers is especially favorable in the case of a low number of process requests, as well as in the case of a rather limited amount of computational resources available to the system. In such cases, as illustrated by our working example, container-based approaches can lead to substantially better results.

- When the number of incoming process requests is large and the pool of available resources is not highly limited, the BPMS can also save a large amount of costs and resources compared to hypervisor-based approaches. In such a case, the container-based resource allocation approach will most probably lease a sufficient amount of large VM instances, as in our considered scenario larger VM instances offer the same amount of resources for a cheaper price than multiple smaller VM instances, and extend the lease for those VM instances as long as the resources are needed. Once the images for starting containers of certain service types are pulled onto the running VM instances, multiple lightweight containers can be started, resized and stopped as needed, without incurring any additional costs, while making the best possible use of the rather cheaply leased computational resources.

6.4 Quantitative Evaluation

In the following section, we evaluate the implementation of our proposed approach by conducting experiments using the research prototype and BPMS testbed ViePEP with the extensions introduced in Section 5.4.

All test have been executed on a machine with a quad core CPU with 2.50 GHz, 8 GB of RAM, running Ubuntu 16.04 with Linux kernel 4.8.0.

6.4.1 Evaluation Models

We perform our evaluation using a representative subset of 10 different process models from the SAP reference model [35], which has been used for multiple scientific papers, e.g., [103], and provides a solid foundation for our evaluations. From the around 600 process models, we select 10 models with different process patterns and varying levels of complexity, including sequences, XOR-blocks, AND-blocks, and loops. All XOR- and AND-blocks start with a split and end with a merge pattern. The merge for AND-blocks is blocking, while the merge for XOR-blocks continues the process execution as soon as the last process step of 1 optional branch is finished. Our chosen process models range from simple sequences with 3 process steps to complex process models with up to 20 process steps and can be found in Appendix B. We generally only consider software-based services for process steps and no human-provided services.

In Table 6.1 we present the most important characteristics of the 10 considered process models.

Table 6.1: Evaluation Process Models

Process Name	**|Steps|**	**|XOR|**	**|AND|**	**|Loops|**
1	3	0	0	0
2	2	1	0	0
3	3	0	1	0
4	8	0	2	0
5	3	1	0	0
6	9	1	1	0
7	8	0	0	0
8	3	0	1	0
9	4	0	1	1
10	20	0	4	0

The process steps of the used process models consist of a mixture of the following 10 rather resource-intensive software services as shown in Table 6.2.

Table 6.2: Evaluation Services - Resource-Intensive

Service Type Name	**CPU Load in %** (μ_{cpu})	**Service Makespan in sec.** (μ_{dur})
A	45	40
B	75	80
C	75	120
D	100	40
E	120	100
F	125	20
G	150	40
H	175	20
I	250	60
J	333	30

A services' CPU load refers to the required amount of CPU resources in percent of a single core, meaning all services with a CPU requirement of more than 100, cannot be scheduled on a single-core VM and the corresponding containers will need to be placed on a multi-core VM instance. We assume services that are fully parallelizable and an increased number of available CPU cores to a service's container instance has no influence on the makespan of a single step. The values in Table 6.2 represent mean values of general normal distributions with $\sigma_{cpu} = \mu_{cpu}/10$ and $\sigma_{dur} = \mu_{dur}/10$.

Applied SLAs

For the execution of each process instance, different deadline SLAs may be defined on process level with regard to the complete process enactment. We evaluate our approach using 2 different process deadline scenarios. In the first scenario, we use strict deadlines and demand the latest point in time when the execution of process instances has to be finished to be 1.5 times the process model's average makespan from the point in time when the request is sent. The second scenario allows more lenient SLA values and we demand 2.5 times of the process model's average makespan as the maximum execution duration for the BPMS. These values were chosen to account for the startup time of VM instances and the time to pull Docker images and deploy container instances.

For calculating penalty costs, we apply a linear cost model which assigns 1 unit of penalty cost per 10% of time units of delay.

Process Request Arrival Patterns

Our experimental design accounts for 2 different process request arrival patterns. In one scenario we design a constant arrival pattern, meaning that the same number of process instance requests arrives in regular time intervals. Specifically, we choose to request 5 process instances every 2 minutes, alternating process instance requests for process models 1 to 5 and 6 to 10 until 50 process instance requests have been sent. In a second scenario, we follow a pyramid-like function for designing our arrival pattern. We send a total of 100 randomly shuffled process instance requests in different batch sizes, ranging from 1 to 5 instances at a time in an interval of 1 minute. Equation 6.1 represents the pyramid pattern, n representing the time in minutes used to calculate a, and a representing the number of new process requests.

$$f(n) = a \begin{cases} 1 & if 0 \leq n \leq 4 \\ \lceil (n+1)/4 \rceil & if 5 \leq n \leq 17 \\ 0 & if 18 \leq n \leq 19 \\ 1 & if 20 \leq n \leq 35 \\ \lceil (n-9)/20 \rceil & if 36 \leq n \leq 51 \end{cases} \tag{6.1}$$

6.4.2 Simulated Test Environment

A rather large amount of research on scheduling and resource allocation problems for the cloud uses self-constructed simulation environments, but also off the shelf simulation frameworks like, e.g., CloudSim[1] for evaluation purposes [97].

For allowing large and realistic test scenarios over multiple hours and with different pricing models we perform the optimization runs using the simulation mode of our extended ViePEP implementation as introduced in Section 5.4.

Our system considers 7 VM types with the characteristics listed in Table 6.3

[1] http://www.cloudbus.org/cloudsim/

Table 6.3: Evaluation VM Types

VM Type Name	Provider	CPU Cores	Costs per BTU
1	Private	1	10
2	Private	2	18
3	Private	4	30
4	Public A	1	15
5	Public A	2	25
6	Public A	4	35
7	Public A	8	50

As Table 6.3 shows, we assume that leasing a VM instance with x cores is cheaper than leasing 2 VM instances with $x/2$ cores.

We assume the system can lease up to 3 VM instances of each type at a time.

Assumptions for Startup and Deployment Times

In our simulation, we generally assume a VM startup time of 60 seconds, a time to pull Docker images from the central registry onto a VM instance of 30 seconds, and a time to start a new Docker container and the contained service from an existing image of 2 seconds.

These assumptions are based on existing literature. In earlier work, VM startup times have been found to take up to several minutes [42]. More recent findings indicate VM startup times to be below 30 seconds [163], however, there is an intrinsic additional overhead to provision a VM in a cloud environment. Hence, we assume a VM startup time of 60 seconds when leasing instances from a cloud provider like AWS. Container startup times have been shown to be in the range of 1 to 2 seconds, regardless of the image size [173]. Image sizes have been found to often be in the range of 200 to 300 MBs [58], which, under an assumed download speed of 80 MBit/s, results in an approximate download time of roughly 20 to 30 seconds.

6.4.3 Metrics

We assess the outcome of our experiments by using the following metrics:

- **Total costs**: Our optimization approach aims at minimizing the overall costs for cloud providers and process owners. We compare the total costs from leasing VM instances and penalty costs arising from delayed process enactments. The total costs include the VM leasing and penalty costs over all process executions during an experiment's runtime.
- **SLA Adherence**: We calculate the percentage of process requests which meet their SLAs, i.e., the process instances that finish their execution before the defined deadline.
- **Makespan**: We measure the overall duration for executing all incoming process requests, from the first received request to the last finished process step.

We also calculate the standard deviation σ for all metrics, by performing 3 different evaluation runs.

We present the calculated metrics in our result sections in form of tables and charts. The tables represent the numerical average values of the evaluation runs and the standard deviations. The charts plot the leased CPU cores over time and also show the corresponding arrival patterns. The horizontal axes show the time in minutes. The left vertical axes display the number of leased CPU cores while the right vertical axes represent the number of parallel process requests. It is important to distinguish between the total makespan in minutes as discussed in the tables, which refers to the time for executing all incoming process requests, and the timespan shown in the charts which refers to the leasing duration of VM instances.

6.4.4 Solving the Full Optimization Model with CPLEX

Before performing the evaluation of the presented implementation approach, in which we define a reduced optimization model that is solved by the MILP solver CPLEX and afterwards transformed such that the end result delivers a solution for the full optimization problem, we discuss the performance of the full optimization problem when directly solved with CPLEX. We compare the result of the full optimization problem with the transformed result of the reduced optimization problem.

Additional Considerations for the Test Environment

For calculating the placement of services on Docker containers we need to extend the considered test environment, such that a set of Docker containers with certain configurations are available for the solver to calculate an optimal placement.

In the following test runs, we set up the test environment for the full optimization problem such that it offers 2 containers for each of the 10 Docker images and for each possible resource configuration of a Docker image. We use 3 simple resource configurations for containers that guarantee a certain number of resources. We allow configurations that guarantee for each container the processing power of 1 core, 2 cores, or 4 cores, of an underlying VM. This allows our optimization model to choose from 60 different Docker containers for performing the task-scheduling and resource allocation decisions.

This test environment setup shows a substantial limitation when directly trying to solve the full optimization problem with a MILP solver like CPLEX. The quality and precision of the result depend on the number of available container instances. As each container includes the application code for a service type, we need to consider an increasing amount of container instances for an increasing amount of offered service types. At the same time, the number of containers increases with each considered configuration. When the system does not consider enough containers of a certain service type as an input, it might unnecessarily delay scheduling decisions although VM resources might still be available. A discrete set of containers with discrete configurations strongly limits the scheduling decisions for the BPMS and the scaling possibilities that can come with using container technologies. When a large amount of containers with fine-grained configurations is considered to minimize the scheduling restrictions, the flexibility comes with an exponentially growing overhead for calculating an optimal scheduling solution.

Note that the result of both our reduced model and previous work on VM-based task-scheduling and resource allocation strongly depend on the number of available VM resources and the single VM instances have to be considered when calculating an optimization result. In contrast to container instances, VM instances are generally only available with discrete configurations, which allows to precisely use all possible configurations for the optimization. Nevertheless, with an increased number of available resources, an overhead for calculating a scheduling decision will occur. This overhead can be limited by using heuristics that only use a certain subset of the available VM resources for performing an optimization run, e.g., only considering all running VMs and a limited number of unleased resources for each service type, depending on the number of next schedulable steps.

A comparable heuristic that starts the optimization model with a certain set of configured containers, depending on the next schedulable steps could help in managing the overhead that comes with considering single container instances in the optimization problem. Such a heuristic should be explored in future work that focuses on scheduling decisions that require a certain isolation of services in containers as discussed in Section 4.4. In our considered special case, when only 1 container type may be deployed per VM instance, the reduced model followed by the transformation step offers a scalable solution and will yield the best possible result for the defined problem, as a continuous assignment and reassignment of computational resources to containers is realized.

Results and Discussion

When comparing the results of example runs for the full model and the reduced model after transformation for the evaluation scenario with constant process request arrivals and strict deadlines, we achieve results as illustrated in Figure 6.4 and depicted in Table 6.4. Note that in this subsection we only present and compare the exemplary execution of one evaluation run.

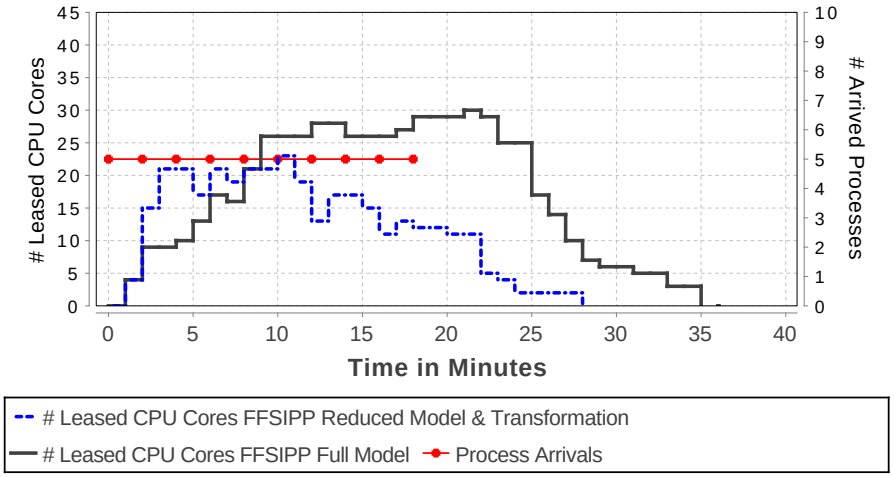


Figure 6.4: Solving the Reduced Optimization Problem and Performing the Transformation vs. Solving the Full Optimization Problem

Table 6.4: Solving the Reduced Optimization Problem and Performing the Transformation vs. Solving the Full Optimization Problem - Constant Arrival with Strict Deadlines

Arrival Pattern	**Constant Arrival**	
SLA Level	**strict**	
	FFSIPP reduced model and transformation	**FFSIPP** solving the full model
Number of Total Process Requests	50	
Interval Between Process Requests	120 Seconds	
Number of Parallel Process Requests	5	
SLA Adherence in %	96	10
Total Makespan in Minutes	28	32
Leasing Cost	1052	2069
Penalty Cost	3	162
Total Cost	1055	2231

The solution of the full optimization problem fully implemented with CPLEX yields a longer total makespan and leads to leasing costs that are twice as high when compared to the implemented reduced model after transformation. The main issue is the very low SLA adherence of only 10 percent and the associated penalty costs that are over 54 times higher for the full model.

The main reason for the poor performance is that with the given setting the full model was not able to calculate an optimal solution in slightly over 32 percent of all optimization rounds, only considering rounds where process requests existed and processes were still running. We generally demand every optimization calculation to be performed in under 1 minute. In the given scenario the solver failed to calculate an optimal solution within the time limit using the full optimization problem when around 10 or more running process requests had to be managed and scheduled at the same time. The enactments of these suboptimal solutions lead to the poor overall optimization outcome. In comparison, every result for the reduced optimization model calculates an optimal solution in under 5 seconds, also when regarding all subsequently presented evaluation runs.

As a proof of concept and to show the validity of the presented full optimization model, we perform 2 more example executions with a more relaxed arrival pattern. With a constant arrival of 2 requests every 3 minutes for a total of 20 process requests with strict deadlines, we achieve the following optimization results depicted in Figure 6.5 and presented in Table 6.5.

Again not all optimization rounds for the full model could find an optimal solution within the set time limit of 1 minute, but now only 0.85% of all rounds yield suboptimal feasible solutions. When calculating solutions with further slight relaxations of the arrival pattern, e.g., 2 requests every 4 minutes or 1 request every 2 minutes, optimal solutions can be found for all optimization rounds in under 1 minute.

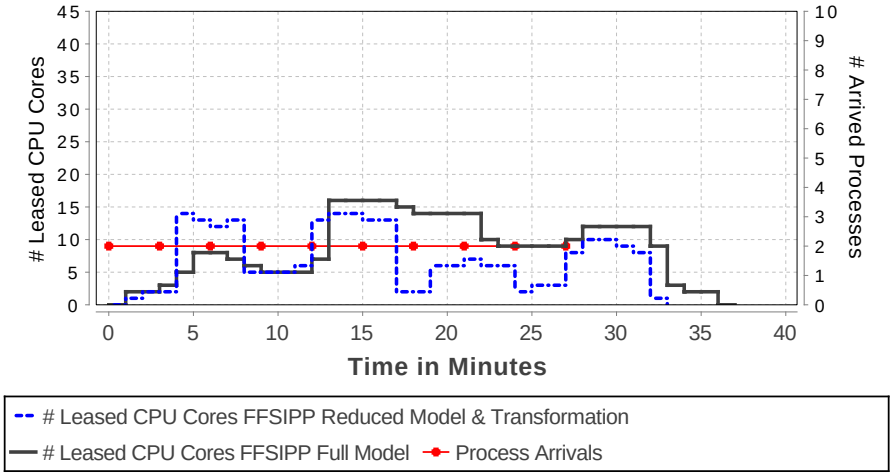


Figure 6.5: Solving the Reduced Optimization Problem and Performing the Transformation vs. Solving the Full Optimization Problem - Smaller Example

Table 6.5: Solving the Reduced Optimization Problem and Performing the Transformation vs. Solving the Full Optimization Problem - More Relaxed Arrival Frequency

Arrival Pattern	**Constant Arrival**	
SLA Level	**strict**	
	FFSIPP reduced model and transformation	**FFSIPP** solving the full model
Number of Total Process Requests	20	
Interval Between Process Requests	180 Seconds	
Number of Parallel Process Requests	2	
SLA Adherence in %	95	95
Total Makespan in Minutes	32	37
Leasing Cost	669	1000
Penalty Cost	1	1
Total Cost	670	1001

From the performed optimization runs we can see that now the SLA adherence of both, the full model and the reduced model after transformation are the same, while the full model still produces higher leasing costs, due to the fact that it does not allow for the same fine-grained resource allocation as the reduced model after transformation, because only a limited set of available container instances are provided as input to the full model.

6.4.5 Evaluation of the Reduced Model including Transformation

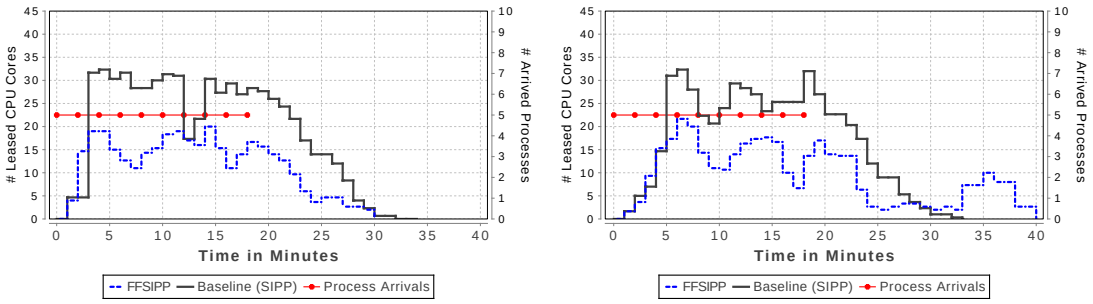
Now we perform the actual evaluation of our presented approach. We evaluate our Docker-based implementation of the optimization problem using the reduced model and the transformation against a hypervisor-based baseline optimization approach.

Baseline

We use a slightly modified version of the VM-based resource allocation and task-scheduling approach presented by Hönisch et al. [63] as a baseline for the evaluation of our presented solution. We already discussed the execution of the MILP-based optimization baseline approach in Section 6.3. The baseline approach uses an optimization model that directly schedules process requests on VM instances, while each leased VM instance is only able to execute process requests of 1 specific service type. The modifications of the originally presented baseline are made with regard to the utilized time constraints which we define according to the time constraints also used by our approach as presented in A.2 and A.3. Furthermore, we allow the change of the offered service type of a VM instance during an uninterrupted leasing duration whenever no more service requests are being executed by a VM instance. When changing the offered service type, a service deployment time of another 30 seconds is considered and again only 1 service type is offered at any point in time by a VM instance.

Results and Discussion

Constant Arrival of Resource-Intensive Processes: First, we compare the results for resource-intensive processes as presented in Table 6.1 and Table 6.2 for the constant arrival pattern, which are presented in Figure 6.6 and Table 6.6. Figure 6.6a shows the results for process requests with a strict deadline while Figure 6.6b depicts the results for lenient process requests.



(a) Strict SLA (b) Lenient SLA

Figure 6.6: Evaluation Results - Constant Arrival of Resource-Intensive Processes

Table 6.6: Evaluation Results - Constant Arrival (Resource-Intensive Processes)

Arrival Pattern	**Constant Arrival**			
	FFSIPP		**Baseline (SIPP)**	
SLA Level	**Strict**	**Lenient**	**Strict**	**Lenient**
Number of Total Process Requests	50			
Interval Between Process Requests	120 Seconds			
Number of Parallel Process Requests	5			
SLA Adherence in % (Standard Deviation)	98.67 $\sigma = 1.15$	100.00 $\sigma = 0.00$	86.00 $\sigma = 2.00$	98.67 $\sigma = 1.15$
Total Makespan in Minutes (Standard Deviation)	27.33 $\sigma = 1.15$	36.67 $\sigma = 2.08$	29.33 $\sigma = 0.58$	29.33 $\sigma = 0.58$
Leasing Cost (Standard Deviation)	1148.33 $\sigma = 20.50$	1102.00 $\sigma = 14.42$	2201.00 $\sigma = 55.56$	2052.67 $\sigma = 147.99$
Penalty Cost (Standard Deviation)	1.00 $\sigma = 1.00$	0.00 $\sigma = 0.00$	15.00 $\sigma = 2.65$	0.67 $\sigma = 0.58$
Total Cost (Standard Deviation)	1149.33 $\sigma = 20.01$	1102.00 $\sigma = 14.42$	2216.00 $\sigma = 57.51$	2053.33 $\sigma = 148.37$

As we can see from the results, our optimization approach yields a high SLA adherence of 98.67% with a very small standard deviation of only 1.15 percentage points and only 1 unit of penalty costs for process requests with strict execution deadlines. Compared to the baseline which only yields an SLA adherence of 86.00% with a standard deviation of 2.00 percentage points, and on average 15 units of penalty costs, our approach was considerably stronger in detecting potential SLA violations and rapidly calculating a good scheduling strategy. Furthermore, when regarding the leasing costs, our approach leads to only about half the costs needed when using the baseline with also a lower associated standard deviation.

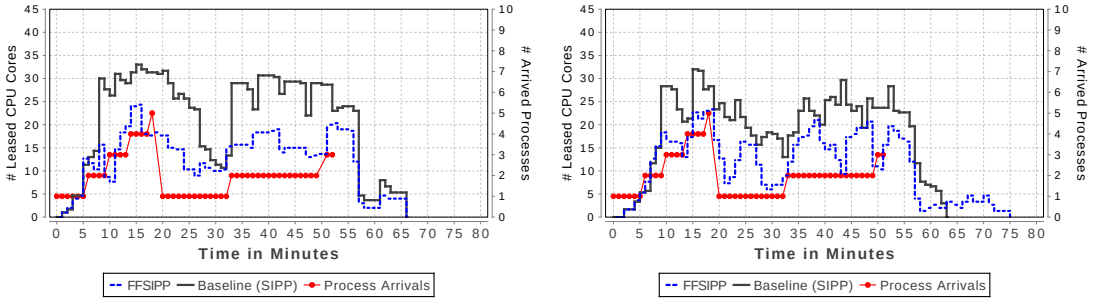
When comparing the result for more lenient deadlines, we can observe a much better SLA adherence of 98.67% for the baseline (as compared to strict deadlines), but again a better SLA adherence of 100.00% for all performed runs using our approach. This means our approach was able to finish all incoming process requests within the defined deadline constraints for all evaluation runs. The leasing costs of our approach are again nearly half as high compared to those of the baseline. The low leasing costs of our approach are a result of the near-optimal resource utilization that is realized through the dynamic assignment and reassignment of Docker containers for the execution of different service types that only reserve as much space from the underlying VM instances as they actually require.

As expected, the execution of process requests using our approach with lenient process deadlines leads to a longer total makespan and lower total leasing costs compared to the execution of process requests with a strict deadline. When looking at the number of leased resources in Figure 6.6b, we can clearly observe that our approach leases a smaller amount of computational resources than the baseline at most points in time, and that our approach was able to postpone the execution of process steps with a distant deadline, until postponing was no longer possible. This explains why after 33 minutes the number of leased resources increases after it had already declined. The baseline approach does not make use of the more distant deadline constraints in a similarly efficient way. The reason why the makespan for the baseline is similar for both the strict and the lenient process deadlines is that there are already sufficient computational resources leased at earlier points in time. These resources were not fully utilized and were able to perform the execution of process steps ahead of their scheduling deadlines.

Pyramid Arrival of Resource-Intensive Processes: Next, we compare the execution results for resource-intensive processes as presented in Tables 6.1 and 6.2 for the pyramid arrival pattern as introduced in Function 6.1. We achieve the results as presented in Figure 6.7 and Table 6.7. Figure 6.7a shows the results for process requests with a strict deadline while Figure 6.7b depicts the results for lenient process requests.

Again, when compared to the baseline, our approach results in a better SLA adherence and lower associated penalty costs as well as leasing costs that are approximately half as high. This time, the difference between the execution of process requests with a strict or a lenient deadline is relatively small. The explanation for the large saving of leasing costs of our approach is the same as for the constant arrival pattern and can be attributed to the very fine-grained resource utilization of our approach. Once again, our approach was also able to postpone steps into the future, leading to a longer total makespan for process requests with lenient deadlines, when compared to the baseline.

It should be noted that lenient deadlines lead to better results in both our approach as well as the baseline, but an important observation can be made when looking at the evaluation more closely. The performance benefit, especially in terms of costs is much higher when considering lenient deadlines for the baseline approach. Our approach also yields a slightly better result for lenient deadlines, but in comparison, this difference is rather small. For the baseline a large number of requests that have to be executed in close temporal proximity result in having to lease individual VMs for almost all service types. Some service types might only occupy a small part of a leased machine while for other service types multiple machines need to be leased. More lenient deadlines allow the use of already leased machines when computational resources are freed up in the near future, leading to the observed performance benefit. Depending on the defined preferences and costs considering penalties, the system can accept higher penalties for lower leasing costs. The higher the accepted penalties are, the more additional leasing costs could be saved. In contrast to the baseline, our approach is able to use free resources from any leased VM to deploy a container of any type with only as many assigned resources as the invocation of service requests will actually require. This flexible resource utilization leads to comparable leasing costs for strict and lenient deadlines, as the number of leased but unused resources is not increased as easily as for the baseline.



(a) Strict SLA

(b) Lenient SLA

Figure 6.7: Evaluation Results - Pyramid Arrival of Resource-Intensive Processes

Table 6.7: Evaluation Results - Pyramid Arrival (Resource-Intensive Processes)

Arrival Pattern	**Pyramid Arrival**			
	FFSIPP		**Baseline (SIPP)**	
SLA Level	**Strict**	**Lenient**	**Strict**	**Lenient**
Number of total Process Requests	100			
Interval between Process Requests	60 Seconds			
Number of Parallel Process Requests	$f(n)$ (see 6.1)			
SLA Adherence in % (Standard Deviation)	97.67 $\sigma = 0.58$	100.00 $\sigma = 0.00$	95.33 $\sigma = 0.58$	99.67 $\sigma = 0.58$
Total Makespan in Minutes (Standard Deviation)	63.33 $\sigma = 0.58$	69.67 $\sigma = 2.89$	62.00 $\sigma = 0.00$	61.67 $\sigma = 0.58$
Leasing Cost (Standard Deviation)	2322.00 $\sigma = 39.74$	2181.67 $\sigma = 89.44$	4413.33 $\sigma = 121.49$	3975.00 $\sigma = 118.87$
Penalty Cost (Standard Deviation)	3.33 $\sigma = 1.53$	0.00 $\sigma = 0.00$	10.67 $\sigma = 1.15$	0.33 $\sigma = 0.58$
Total Cost (Standard Deviation)	2325.33 $\sigma = 38.21$	2181.67 $\sigma = 89.44$	4424.00 $\sigma = 122.32$	3975.33 $\sigma = 119.43$

The results clearly show the main strength of our task-scheduling approach, which is the better resource utilization due to a fine-grained scheduling leading to overall lower leasing costs. We show this main strength while comparing our approach to an already strong baseline, which has been shown to deliver considerably better results in terms of cost and SLA adherence than simple threshold-based ad-hoc scheduling strategies [63]. The baseline approach already delivers a good SLA adherence as it detects potential violations in time and optimizes the scheduling decision accordingly. Nevertheless, our approach leads to even lower SLA violations.

Both approaches allow SLA violations if the associated penalty leads to lower total costs than leasing additional resources to finish process executions in time. As our approach allows for a more flexible scheduling of containers on VMs, it can perform cheaper scheduling decisions and make better use of already leased resources, resulting in lower SLA violations to leasing cost conflicts.

Generally, the results do not only depend on the number and type of process request arrivals and the available resources, but also on the attributes of the process steps. In particular, service types with larger resource requirements will lead to large containers taking up a great amount of the offered resources of a VM instance and therefore fewer resources remaining available for other large service requests of different service types. This consideration leads to the assumption that our approach is capable of outperforming the baseline even more when service instances with smaller resource requirements are considered. We evaluate this assumption in the following subsection.

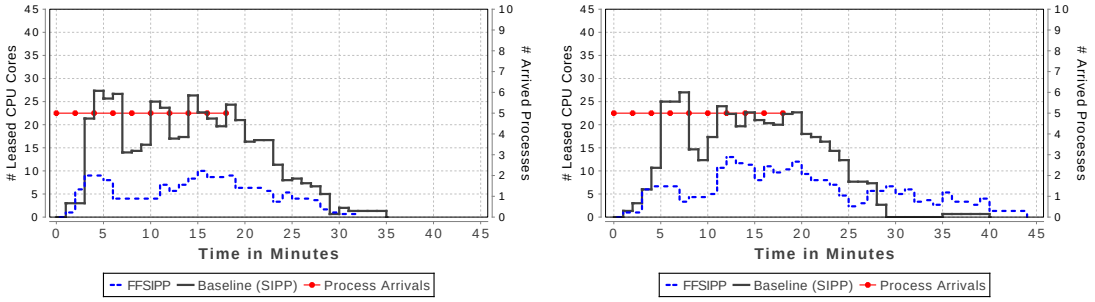
6.4.6 Evaluation of the Reduced Model including Transformation for Less Resource-Intensive Service Types

To evaluate the assumption that our approach leads to even better results when considering task-scheduling problems for service instances with smaller resource requirements, we perform the same evaluation as already discussed in Subsection 6.4.5, while now using the less resource-intensive service types introduced in Table 6.8, instead of the previously used service types from Table 6.2.

Table 6.8: Evaluation Services - Less Resource-Intensive

Service Type Name	**CPU Load in %** (μ_{cpu})	**Service Makespan in sec.** (μ_{dur})
A	5	40
B	10	80
C	15	120
D	30	40
E	45	100
F	55	20
G	70	40
H	125	20
I	125	60
J	190	30

Constant Arrival of Less Resource-Intensive Processes: Now, we compare the results for less resource-intensive processes as presented in Tables 6.1 and 6.8 for the constant arrival pattern and achieve the results as presented in Figure 6.8 and Table 6.9. Figure 6.8a shows the results for process requests with a strict deadline while Figure 6.8b depicts the results for lenient process requests.



(a) Strict SLA

(b) Lenient SLA

Figure 6.8: Evaluation Results - Constant Arrival of Less Resource-Intensive Processes

Table 6.9: Evaluation Results - Constant Arrival (Less Resource-Intensive Processes)

Arrival Pattern	**Constant Arrival**			
	FFSIPP		**Baseline (SIPP)**	
SLA Level	**Strict**	**Lenient**	**Strict**	**Lenient**
Number of Total Process Requests	50			
Interval Between Process Requests	120 Seconds			
Number of Parallel Process Requests	5			
SLA Adherence in %	97.33	99.33	93.33	99.33
(Standard Deviation)	$\sigma = 1.15$	$\sigma = 1.15$	$\sigma = 2.31$	$\sigma = 1.15$
Total Makespan in Minutes	27.33	37.33	29.67	31.00
(Standard Deviation)	$\sigma = 1.15$	$\sigma = 3.06$	$\sigma = 1.15$	$\sigma = 4.36$
Leasing Cost	479.33	512.00	1316.67	1283.67
(Standard Deviation)	$\sigma = 36.07$	$\sigma = 32.14$	$\sigma = 124.16$	$\sigma = 187.79$
Penalty Cost	2.33	1.33	6.67	0.33
(Standard Deviation)	$\sigma = 0.58$	$\sigma = 2.31$	$\sigma = 1.15$	$\sigma = 0.58$
Total Cost	481.67	513.33	1323.33	1284.0
(Standard Deviation)	$\sigma = 36.12$	$\sigma = 34.44$	$\sigma = 125.13$	$\sigma = 188.34$

As expected, the results for process requests composed of less resource-intensive process steps show even higher saving in terms of leasing costs when comparing our approach to the baseline, with around 2.5 times lower costs. The SLA adherence of our approach for less resource-intensive processes is comparable to the evaluation of the more resource-intensive processes as presented in Subsection 6.4.5, but the baseline shows a higher SLA adherence with the less resource-intensive service types.

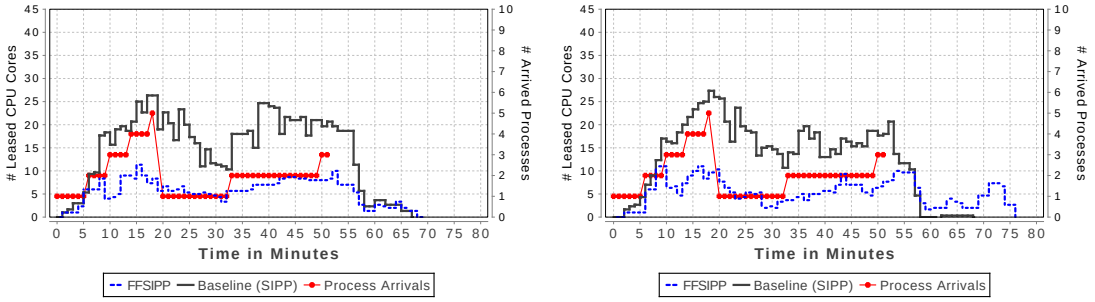
The improved SLA adherence for the baseline is easy to explain when considering that with smaller service types more service invocations of the same type can be executed on a VM instance, leading to a lower number of penalty cost versus leasing cost conflicts for the baseline approach. As our approach is able to place requests on any already leased VM instance, there already exists a relatively large amount of scheduling flexibility for the more resource-intensive service types, leading to similar SLA adherence values, independent of the required resources for process steps.

All other observations already discussed in Subsection 6.4.5 for more resource-intensive process steps can also be observed in the evaluation runs discussed in this section, e.g., the consequences of more lenient deadlines.

Pyramid Arrival of Less Resource-Intensive Processes: Next, we compare the execution results for less resource-intensive processes as presented in Tables 6.1 and 6.8 for the pyramid arrival pattern as introduced in Function 6.1. We achieve the results as presented in Figure 6.9 and Table 6.10. Figure 6.9a shows the results for process requests with a strict deadline while Figure 6.9b depicts the results for lenient process requests.

Table 6.10: Evaluation Results - Pyramid Arrival (Less Resource-Intensive Processes)

Arrival Pattern	**Pyramid Arrival**			
	FFSIPP		**Baseline (SIPP)**	
SLA Level	**Strict**	**Lenient**	**Strict**	**Lenient**
Number of total Process Requests	100			
Interval between Process Requests	60 Seconds			
Number of Parallel Process Requests	$f(n)$ (see 6.1)			
SLA Adherence in % (Standard Deviation)	97.33 $\sigma = 0.58$	100.00 $\sigma = 0.00$	98.67 $\sigma = 0.58$	100.00 $\sigma = 0.00$
Total Makespan in Minutes (Standard Deviation)	63.67 $\sigma = 2.31$	71.67 $\sigma = 0.58$	61.67 $\sigma = 1.15$	63.33 $\sigma = 4.04$
Leasing Cost (Standard Deviation)	1018.67 $\sigma = 25.97$	996.67 $\sigma = 48.33$	2619.00 $\sigma = 91.02$	2430.33 $\sigma = 114.29$
Penalty Cost (Standard Deviation)	4.33 $\sigma = 1.15$	0.00 $\sigma = 0.00$	3.6 $\sigma = 0.58$	0.00 $\sigma = 0.00$
Total Cost (Standard Deviation)	1023.00 $\sigma = 24.88$	996.67 $\sigma = 48.34$	2622.67 $\sigma = 90.47$	2430.33 $\sigma = 114.29$



(a) Strict SLA

(b) Lenient SLA

Figure 6.9: Evaluation Results - Pyramid Arrival of Less Resource-Intensive Processes

Again, we can observe that our approach leads to about 2.5 times lower leasing costs for process requests composed of less resource-intensive process steps when compared to the baseline. We can also make the same observations for SLA adherences as we already did for the constant arrival of less resource-intensive processes, as well as the other observations discussed in Subsection 6.4.5 for more resource-intensive process steps.

In this section, we have demonstrated that the good results presented for resource-intensive process requests with different arrival patterns in the previous section, can even be outperformed in terms of leasing cost minimization when regarding less resource-intensive processes. For the described settings we achieve a high SLA adherence of over 97%, also for strict deadlines.

6.4.7 Multiple Hour Evaluation Run with 60 Minute long BTUs

Finally, we present the exemplary result of an 8 hour long evaluation run that considers more realistic and common BTUs of 60 minutes. The presented results were achieved using the resource-intensive service types presented in Table 6.2. The used process request arrival pattern is defined as a mix of all already used arrival patterns and summarized in Table 6.11.

We start with the arrival of 150 process requests with strict deadlines, the first 50 following the constant arrival pattern and the other 100 following the pyramid arrival pattern. Afterwards, we send 150 more process requests with lenient deadlines, the first 50 following the constant arrival pattern and the other 100 following the pyramid arrival pattern. This arrival scenario is repeated 3 times, resulting in a total of 900 process request arrivals.

Table 6.11: Multiple Hour Arrival Pattern

Request Batch Size	Seconds between Request Batches	Process Deadlines	Total Requests	Minutes Sent
5	120	strict	50	0-18 144-162 288-306
$f(n)$ (see 6.1)	60	strict	100	20-71 164-215 308-359
5	120	lenient	50	72-90 216-234 360-378
$f(n)$ (see 6.1)	60	lenient	100	92-143 236-287 380-431

The results of the multi-hour evaluation for resource-intensive process requests and hour-long BTUs are shown in Figure 6.10 and Table 6.12.

We can again observe that our approach was able to cut the leasing costs nearly in half while maintaining a decent SLA adherence of over 94%. In contrast to the other evaluation runs, the SLA adherence of our approach is now on average lower than before. A reason for this can be that for this setting the system sacrifices some more SLA adherence in favor of lower leasing costs for the VM instances that are leased for a full hour. By defining higher penalty costs for SLA violations, this behavior can be controlled such that higher leasing costs are accepted for a better SLA adherence.

In contrast, the baseline approach shows on average a higher SLA adherence than in the previous runs with short BTUs. This can be explained by the fact that a rather large amount of computational resources are leased from the beginning, which allows for more pre-scheduling decisions to be made, such that more process instances are finished ahead of time. When looking at Figure 6.10, it can be clearly seen that a rather large amount of computational resources were leased during the first BTU of 60 minutes.

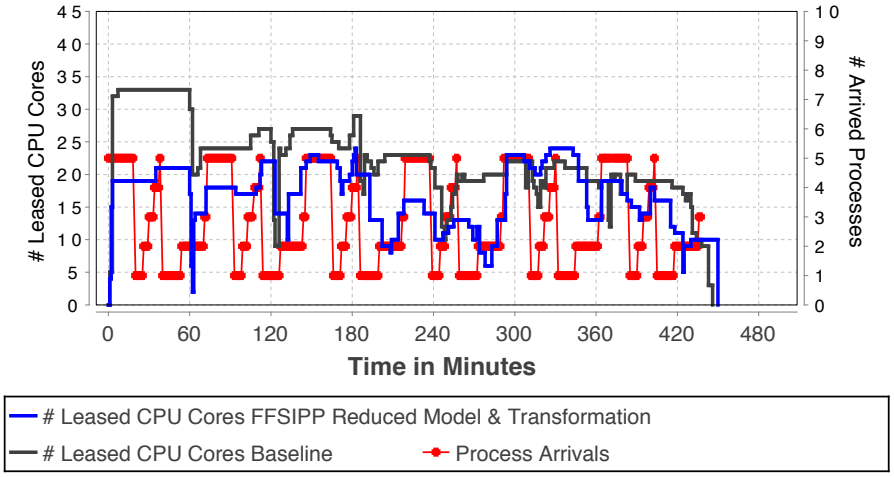


Figure 6.10: Multiple Hour Evaluation Run with 60 Minute BTUs

Table 6.12: Multiple Hour Evaluation Run with 60 Minute BTUs

Arrival Pattern	**Mixed** (see 6.11)	
SLA Level	**Mixed** (see 6.11)	
	FFSIPP	**Baseline**
Number of Total Process Requests	900	
Interval Between Process Requests	60 to 120 Seconds	
Number of Parallel Process Requests	1 to 5	
SLA Adherence in %	94.22	98.78
Total Makespan in Minutes	446	446
Leasing Cost	22759	39063
Penalty Cost	298	22
Total Cost	23057	39085

6.5 Summary

In this chapter, we presented an extensive evaluation of our approach, starting with a detailed discussion of the working example and its results as well as a comparison to other container-based approaches that allow for an earlier scheduling of process steps. We pointed out some important observations that could be made from the scheduling decisions and discussed potential extensions, such as rescheduling, migration, pre-scheduling, and prediction that can help in finding even better solutions. We also compared the execution and results of the working example when using a hypervisor-based approach to the execution and results achieved when using our approach, pointing out the strengths of container-based approaches.

In our detailed quantitative evaluation, we introduced a number of evaluation process models, different process request arrival patterns, and system settings and presented the associated execution results. We discussed the results achieved when directly solving the full model by using a MILP solver and explained its shortcomings. Afterward, we evaluated the results of the reduced model after transformation while comparing it to a strong hypervisor-based approach. We showed that due to the overall more efficient use of leased computational resources, our approach calculates scheduling results with 2 to 2.5 times lower leasing costs as well as a high SLA adherence when compared to the baseline.

CHAPTER 7

Conclusion and Future Work

"Go as far as you can see; when you get there, you'll be able to see farther."

– J.P. Morgan

"Those who have the privilege to know have the duty to act, and in that action are the seeds of new knowledge."

– Albert Einstein

In this final chapter, we summarize the main results achieved in this thesis. We outline our contributions and the way we enhanced the state of the art work in business process task-scheduling and resource allocation. We conclude the thesis by discussing open topics for future work and ongoing trends in related research areas which can build on our presented contributions.

7.1 Summary of Contributions

The advent of cloud computing has lead to a simple, cost-effective, and on-demand access to supposedly unlimited computational resources for everyone, from small startup companies to large corporations. Hereby the realization of elastic processes, i.e., business processes that are carried out on elastic cloud infrastructure, has been enabled. In this thesis, we tackle the problem of business process enactment on cloud-based infrastructure while making use of Docker containers to incorporate the benefits that come from the use of lightweight containerization technologies.

When looking at the state of the art solutions in related fields as presented in Chapter 3, we discussed a number of heuristics and optimization algorithms that have been proposed for hypervisor-based cloud architectures. Most solutions present approaches for the enactment of single services, while only little work has been proposed considering the process perspective.

When considering the current work on containerization technologies there exist only a small number of rather new approaches for single service task-scheduling problems, while, to the best of our knowledge, no other approaches for container-based scheduling of business processes have been proposed so far.

In this thesis we introduced multidimensional optimization approaches and heuristics to allow for the enactment of elastic business processes on container-based cloud architectures, making use of the lightweight properties of software containers. Our approaches enable a fine-grained resource allocation and task-scheduling for business processes while scaling the cloud-based architecture over four dimensions, i.e., horizontally and vertically, each for VM resources and Docker containers. Our solutions consider process-related quality measures defined in SLAs, in particular execution deadlines, and also account for complex process models.

We formulated a general multi-objective optimization model using MILP for solving scheduling problems during optimization rounds. The general model (also referred to as the full model) considers enactment costs and allows for an optimal allocation of resources, based on the knowledge during one optimization round, by minimizing states of resource over-provisioning or under-provisioning. The approach focuses on the minimization of VM leasing costs as well as penalty costs for SLA violations while considering times for VM and container leasing, startup, and deployment. Furthermore, the approach minimizes unused leased resources and considers the importance of scheduled service invocations. We extended the general approach to conform to our special case which only allows the deployment of one container of a certain service type on one individual VM instance, while still allowing the deployment of multiple containers of different service types on the same VM and the deployment of multiple containers of the same service type on different VM instances.

Furthermore, we proposed an alternative realization of the extended general approach, with a lower computational complexity. Therefor, we presented a reduced MILP model that directly schedules service invocations on VM instances, while considering container-based properties, and afterwards performs a transformation step to transform the result of the reduced model to a container-based scheduling result of the full model.

As an alternative to a rather heavyweight exact MILP-based optimization solution, we also proposed a general design of a genetic algorithm-based approach for solving the reduced optimization problem. The result of this genetic algorithm can be transformed in the same way we transformed the result of the reduced model, such that we receive a result for the full model.

We extended ViePEP, an initially hypervisor-based BPMS research prototype, to allow for a fine-grained, elastic enactment of concurrent business processes on flexible container-based cloud architectures. We implemented both, the full model, as well as the reduced model using a state-of-the-art commercial MILP solver. Furthermore, we implemented the result transformer for the reduced model.

Additionally, we extended ViePEP to allow for fast and efficient simulations of business process executions, by developing a new simulated time component that realizes a discretization of the system's time. This component can especially help with large-scale simulations that allow for a better design of representative real world scenarios and will also be helpful for future work.

Finally, we extensively evaluated our approach and the developed implementations. To begin with, we provided a detailed evaluation of the working example. We compared our approach to an existing hypervisor-based scheduling and resource allocation approach for elastic processes and discussed multiple important observations regarding scheduling performance and possible further improvements. Furthermore, we performed multiple evaluations of our implemented BPMS prototype. We showed the validity of the full model and compared its execution result to the reduced model after transformation. Multiple evaluations with different settings and preconditions for our container-based reduced model after transformation have demonstrated a strong performance gain, especially in terms of resource utilization of our approach. When compared to a hypervisor-based solution following previous work [63], our solution was able to cut leasing costs in half and in some cases lead to even higher savings.

7.2 Future Work

This thesis examines different approaches to realize the execution of elastic processes on container-based cloud infrastructure. Throughout this work, we mentioned some relevant points that should be considered in more detail in future work. Some of these remarks can be found in Section 6.2, as well as in Sections 4.4 and 5.2. We have identified the following points that should be considered and evaluated in more detail by future research on the topic of elastic process execution.

- To allow for better scheduling decisions, a prediction model for future process requests should be implemented. Such a model could be realized using predictive analytics and machine learning approaches. We currently use the information about the current and future state of the known process requests, but as new process instances can arrive at any time, a prediction model can help in making better scheduling decisions over multiple optimization time periods. One possible improvement could involve pre-scheduling decisions of postponable process steps and the early leasing of computational resources, which can optimize the future system landscape, and minimize total leasing costs.

- To calculate the future workload of indeterministic process executions, our approach currently performs a worst-case analysis. We consider the worst possible path and assume that we have to lease additional resources and deploy corresponding containers for all future process steps. Future work should experiment with with predictive approaches, and/or consider the average case or even the best case when evaluating the next steps and making scheduling decisions.

- Whenever a change of the system state is detected, past scheduling decisions that have not been enacted yet should be revised, leading to potential rescheduling decisions for scheduled but not running process steps. Such reschedulings can be implemented easily and could potentially lead to further performance gains.

- A more difficult form of rescheduling that may lead to cost savings are migrations of running process steps along with their container instances. By allowing migrations of containers from one VM instance to another, remaining leasing durations and free resources could be utilized more efficiently. At the same time, migrations come with a wide number of necessary considerations and additional overheads that need to be weighed out [171]. Comparable approaches for migration-aware optimization strategies on the level of VMs have been proposed recently [132] and could be integrated with our approach.

- As the performance of the proposed model strongly depends on the number of available resources that can be considered for scheduling decisions, the proposed system requires the realization of further heuristics to easily scale for larger system landscapes. When looking at the set of all available resources, the number and type of considered computational resources for an optimization round should be limited, according to the next schedulable process steps.

- We proposed a solution for the extended general MILP-based optimization model from Section 4.4. By removing the last Constraint 4.48, we would allow the deployment of multiple containers of the same service type on one VM instance. As already discussed, this additional isolation can be desirable in many cases, e.g., for regulatory and security reasons, state isolations, QoS guarantees, and billing in multi-tenant environments. Future work can build upon this work and introduce additional constraints and heuristics for realizing such use cases that demand a certain form of isolation for service executions. For a better system scaling, a comparable heuristic as described in the previous point for VM instances should also be explored on container level, i.e., an approach that starts the optimization model with a certain set of configured containers, depending on the next schedulable steps.

- In Section 5.2 of this thesis we also proposed the general concept and important guidelines of a genetic algorithm for solving our defined problem. Future work still needs to define this algorithm in more detail, especially considering the selection strategy, crossover, and mutation operations. It should be explored and evaluated if the use of such a genetic algorithm can lead to good results for larger problem instances in a short amount of time.

- Our approach can consider different linear pricing models, VM configurations, and BTUs for different VM types of multiple cloud providers. As there also exist different pricing models, e.g., setting individual lease durations for leased resources or bid-based leases like offered by EC2 spot instances[1], future work should consider their effect on optimization outcomes.

[1] https://aws.amazon.com/ec2/spot/

- When regarding different cloud offerings, our approach focuses on the leasing costs of the computational resources. When designing a system for hybrid cloud environments, also other measures have to be explored. Our approach can be extended to also consider data transfer costs [62], or to allow for different scheduling strategies for more data-intensive or more compute-intensive process steps [171]. Future work should also investigate methods to schedule sensitive data with scheduling restrictions, to optimize communication pathways between process steps, or to optimize the use of shared storage facilities while focusing on minimizing communication costs between nodes [97].

- The presented approach is designed for utilizing cloud offerings, while we assume that company-internal private clouds are provided and billed by an organizational unit, similar to external cloud offerings. Future work can extend this approach by additionally optimizing the use of internal computational resources on a physical level [97]. Large savings in terms of energy and maintenance costs may be achieved when also accounting for the placement of VMs on physical hosts that are directly controlled by a service provider.

- Finally, our approach was designed for standard automated business processes of different areas like manufacturing, finance, healthcare, or banking, but the presented concepts can also be applied and should be evaluated for the use in IoT scenarios and for microservice-based architectures.

APPENDIX A

Reduced Optimization Model

For completeness of contents in the following we fully write out the reduced MILP optimization problem solved by our MILP solver as described in Subsection 5.2.1.

A.1 System Model

This section presents the system model required for solving the reduced optimization problem.

Process, System and Decision Variables

Table A.1: System Model: Process Variables

Variable Name	Description
$p \in P = \{1, \ldots, p^{\#}\}$	The set of process models is represented by P, while p indicates a specific process model.
$i_p \in I_p = \left\{1, \ldots, i_p^{\#}\right\}$	The set of process instances of a specific process model p is represented by I_p, while i_p indicates a specific process instance of the process model p.
$j_{i_p} \in J_{i_p} = \left\{1, \ldots, j_{i_p}^{\#}\right\}$	The set of process steps not yet executed (and not yet started) to realize a specific process instance i_p is represented by J_{i_p}, while j_{i_p} refers to a specific process step of the process instance i_p.
$j_{i_p}^* \in J_{i_p}^* \subseteq J_{i_p}$	The set of next process steps that have to be scheduled for execution to realize a specific process instance i_p is represented by $J_{i_p}^*$, while $j_{i_p}^*$ refers to a specific next process step of the process instance i_p. Note: AND-Constructs can execute multiple paths in parallel and can, therefore, have multiple next process steps in the set $J_{i_p}^*$.

Continued on next page –>

Table A.1: System Model: Process Variables

$j_{i_p}^{run} \in J_{i_p}^{run} \subset J_{i_p}$	The set of currently running and not finished process steps of a specific process instance i_p is represented by $J_{i_p}^{run}$, while $j_{i_p}^{run}$ refers to a specific currently running process step of the process instance i_p. Note: AND-Constructs can execute multiple paths in parallel and can, therefore, have multiple currently running process steps in the set $J_{i_p}^{run}$.
DL_{i_p}	The deadline for the enactment of the process instance i_p, is defined by DL_{i_p} and refers to a certain point in time.
$DL_{j_{i_p}^*}$	The deadline for starting the enactment of a next process step $j_{i_p}^*$ such that it still meets its process deadline DL_{i_p}, while performing a worst-case analysis considering all subsequent steps is defined by $DL_{j_{i_p}^*}$ and refers to a certain point in time.
ω_{DL}	The weighting factor for controlling the effect of deadline constraints is indicated by ω_{DL}.
$r_{j_{i_p}}^C, r_{j_{i_p}}^R$	The resource demands of a process step j of the process instance i_p in terms of CPU ($r_{j_{i_p}}^C$) and RAM ($r_{j_{i_p}}^R$) are indicated by the respective terms.

Table A.2: System Model: Optimization Time Variables

Variable Name	**Description**
$t, t_{j_{i_p}}$	A specific point in time is specified by t. The point in time at which a specific service invocation j_{i_p} was scheduled is referred to as $t_{j_{i_p}}$
τ_t, τ_{t+1}	The current time period starting at a specific point in time t is indicated by τ_t. τ_{t+1} refers to the next time period starting at $t+1$. At the beginning of each time period the optimization model is calculated based on all currently available information and its results are enacted.
ϵ	The minimum time period that is at least needed between two optimization runs is defined by ϵ in milliseconds.

Table A.3: System Model: Virtual Machines

Variable Name	**Description**
$v \in V = \{1, \ldots, v^{\#}\}$	The set of VM types is represented by V, while v indicates a specific VM type.

Continued on next page –>

Table A.3: System Model: Virtual Machines

$k_v \in K_v = \left\{1, \ldots, k_v^{\#}\right\}$	The set of leasable VM instances of type v is represented by K_v, while k_v is the k^{th} VM instance of type v. Note: For a time period starting at t, we assume the number of leasable VMs of type v to be limited with $k_v^{\#}$.
BTU_v	The BTU of a VM instance k_v of VM type v is indicated by BTU_v. A BTU is a billing cycle and the minimum leasing duration for a VM instance k_v. A VM instance may be leased for multiple BTUs in a row, but releasing a VM before the end of a BTU leads to a waste of paid resources.
BTU^{max}	An arbitrary large upper bound of possible leasable BTUs for any VM instance k_v is expressed by the helper variable BTU^{max}.
c_v	This variable represents the leasing cost for VM instances of type v for one full BTU.
$\gamma_{(v,t)} \in \mathbb{N}_0^+$	This variable indicates the total number of BTUs to lease any VMs of type v in the time period τ_t.
$\beta_{(k_v,t)} \in \{0,1\}$	This variable indicates if the VM instance k_v is/was already running at the beginning of the time period τ_t.
$g_{(k_v,t)} \in \{0,1\}$	This variable indicates if the VM instance k_v was already running at the beginning of time period τ_t or is being leased during τ_t.
$d_{(k_v,t)}$	This variable indicates the remaining leasing duration of the VM instance k_v at the beginning of the time period τ_t.
ω_d	The weighting factor for controlling the effect of the remaining leasing duration on the optimization outcome is indicated by ω_d.
s_v^C, s_v^R	The resource supplies of a VM type v in terms of CPU (s_v^C) and RAM (s_v^R) are indicated by the respective terms.
$f_{k_v}^C, f_{k_v}^R$	The free resources in terms of CPU ($f_{k_v}^C$) and RAM ($f_{k_v}^R$) of a VM instance k_v are indicated by the respective terms.
ω_f^C, ω_f^R	Weighting factors for controlling the effect of free resources in terms of CPU (ω_f^C) and RAM (ω_f^R) on the optimization outcome are indicated by the respective terms.

Table A.4: System Model: Decision Variables

Variable Name	**Description**
$y_{(k_v,t)} \in \left\{\mathbb{N}_0^+\right\}$	This variable indicates how often (i.e., for how many BTUs) the VM instance k_v should be leased in the time period τ_t.
$x_{(j_{i_p},k_v,t)} \in \{0,1\}$	This variable indicates if process step j of process instance i_p should be invoked on VM k_v in the time period τ_t.

Table A.5: System Model: Containers and Services

Variable Name	Description
$st \in ST = \{1, \ldots, st^{\#}\}$	The set of all container types mapping all possible service types is represented by ST, while st indicates a specific container type for a certain service type.
st_j	The service type of a certain process step j is indicated by st_j.
$z_{(st,k_v,t)} \in \{0, 1\}$	This variable indicates if any containers of service type st have already been deployed on a VM instance k_v (and hence the container image for starting new container instances of service type st has been downloaded on k_v) until the time period τ_t.
ω_z	The weighting factor for controlling the effect of existing images on VM instances for the optimization outcome is indicated by ω_z.
M	An arbitrary large upper bound of possible container deployments for any VM instance k_v is expressed by the helper variable M.
N	An arbitrary large upper bound of possible service invocations for any container instance c_{st} is expressed by the helper variable N.

Startup, Deployment, Execution, and Penalty Times

For calculating execution times for process instances, our approach considers the execution times of single process steps, startup times for VMs, the time for pulling new Docker images onto running VMs instances, and the deployment times for starting Docker containers from Docker images. Our approach is based on a worst-case assumption, meaning for calculating remaining execution times and scheduling plans the system will assume the worst-case of having to lease a new VM instance, pull the needed image onto the instance and deploy a new container for each remaining step. Table A.6 summarizes the used variables for describing execution times.

Table A.6: System Model: Startup, Deployment, Execution, and Penalty Times

Variable Name	Description
Δ_v, Δ	The time in milliseconds to start a new VM of type v is indicated by Δ_v. The maximum VM-startup time of any type, i.e. $max_{v \in V}(\Delta_v)$ is expressed by Δ.
Δ_{st}	The time for pulling a container image for a service type st on a VM instance for the first time is expressed by Δ_{st}.
$\Delta_{c_{st}}$	The time for starting a container instance from an existing image for a service type st on a VM instance is expressed by $\Delta_{c_{st}}$.
e_{i_p}	The remaining execution duration of a process instance i_p is expressed by e_{i_p}. It does not include the execution and deployment time of the process steps that will be scheduled for the current optimization period τ_t, nor the remaining execution time of the currently already running steps.

Continued on next page –>

Table A.6: System Model: Startup, Deployment, Execution, and Penalty Times

$l \in L = \left\{1 \dots l^{\#}\right\},$	The set of all paths is represented by L, while l indicates a specific path within a process.
$L_a \cup L_x \cup RL \subseteq L$	The set of possible paths for AND-blocks L_a, XOR-blocks L_x and repeat loop constructs RL are expressed by the respective terms.
re	The maximum repetitions of a repeat loop are indicated by re.
$e_{i_p}^{seq}, e_{i_p}^{L_a}, e_{i_p}^{L_x}, e_{i_p}^{RL}$	The remaining execution duration of sequences ($e_{i_p}^{seq}$), AND-blocks ($e_{i_p}^{L_a}$), XOR-blocks ($e_{i_p}^{L_x}$) and repeat loop constructs ($e_{i_p}^{RL}$) of the process instance i_p are expressed by the respective terms. The terms do not include the remaining execution time of the currently running steps nor the execution and deployment times for process steps that will be scheduled for the current optimization period τ_t.
$\hat{e}_{i_p}^{s}, \hat{e}_{i_p}^{l}$	The remaining combined service execution duration, time for pulling needed container images, container deployment and VM startup time for all not yet scheduled service invocations of sequences or repeat loop blocks ($\hat{e}_{i_p}^{s}$) and paths in AND- or XOR-blocks ($\hat{e}_{i_p}^{l}$) that still need to be invoked to finalize the enactment of the process instance i_p are expressed by the respective terms. Those values do not include the remaining execution time of already running process steps $j_{i_p}^{run}$ or process steps that are being scheduled in the current optimization period τ_t.
$ex_{j_{i_p}^*}$	The combined execution duration, container deployment and VM-startup time of the next to be scheduled process step $j_{i_p}^*$ is expressed by $ex_{j_{i_p}^*}$.
$ex_{j_{i_p}^{run}}$	The remaining execution time of process steps already scheduled in previous periods and not finished until the start of τ_t is expressed by $ex_{j_{i_p}^{run}}$.
$e_{i_p}^{p}$	The penalty time units of a process instance i_p, i.e., the execution duration of i_p that occurs beyond the deadline DL_{i_p} of i_p, is expressed by $e_{i_p}^{p}$.
$c_{i_p}^{p}$	The penalty cost per time unit of delay of the process instance i_p is represented by $c_{i_p}^{p}$.

A.2 Objective Function

$$
\begin{aligned}
min & \sum_{v \in V} \left(c_v * \gamma_{(v,t)} \right) \\
& + \sum_{p \in P} \sum_{i_p \in I_p} \left(c^p_{i_p} * e^p_{i_p} \right) \\
& + \sum_{p \in P} \sum_{i_p \in I_p} \sum_{j^*_{i_p} \in J_{i_p}} \sum_{v \in V} \sum_{k_v \in K_v} \left(\omega_z * (1 - z_{(st_j,k_v,t)}) * x_{(j_{i_p},k_v,t)} \right) \\
& + \sum_{p \in P} \sum_{i_p \in I_p} \sum_{j^*_{i_p} \in J_{i_p}} \sum_{v \in V} \sum_{k_v \in K_v} \left(\omega_d * d_{(k_v,t)} * x_{(j_{i_p},k_v,t)} \right) \\
& + \sum_{v \in V} \sum_{k_v \in K_v} \left(\omega^C_f * f^C_{k_v} + \omega^R_f * f^R_{k_v} \right) \\
& + \sum_{p \in P} \sum_{i_p \in I_p} \sum_{j^*_{i_p} \in J_{i_p}} \sum_{v \in V} \sum_{k_v \in K_v} \left(\omega_{DL} * (DL_{j^*_{i_p}} - \tau_t) * x_{(j_{i_p},k_v,t)} \right)
\end{aligned}
\tag{A.1}
$$

The objective function in A.1 is subject to minimization under the subsequently following constraints. It considers six terms for calculating a scheduling plan, indicated by the decision variable $x_{(j_{i_p},k_v,t)}$, which decides on which VM instance to schedule a service invocation.

- **Term 1** - *minimize total VM leasing costs:* The first term, i.e., $\sum_{v \in V} \left(c_v * \gamma_{(v,t)} \right)$, calculates the total costs for leasing VM instances of each VM type v at time t, by considering the leasing costs c_v for the VM types and the total number of leased BTUs $\gamma_{(v,t)}$ that VM instances of the corresponding type v are leased at t. The aim is to minimize the total costs for leasing VMs.

- **Term 2** - *minimize penalty costs:* The second term, i.e., $\sum_{p \in P} \sum_{i_p \in I_p} \left(c^p_{i_p} * e^p_{i_p} \right)$, calculates the total penalty costs that arise from scheduling decisions that lead to execution times for process instances beyond their defined deadlines in time t or in the future. The penalty time $e^p_{i_p}$ of a process instance i_p is calculated according to a worst-case analysis. The aim is to minimize such penalty costs and therefore make sure that incoming process requests are executed in time, as defined in their SLAs.

- **Term 3** - *minimize container deployment time:* The third term minimizes the sum of time needed for deploying new container instances on VM instances k_v. As the actual container instances are only generated during the transformation step, we cannot directly consider the scheduling of container instances in the optimization problem. Instead, we consider the placement of service steps on the VM instance and use the service type of the scheduled process steps to indirectly calculate the potential deployment time for container instances that are generated during the transformation step. The subterm $(1 - z_{(st_j,k_v,t)})$ makes sure that the weighting factor ω_z is only considered when the cache for container deployment does not exist on the VM instance k_v. As the full term is minimized, the system prefers choosing VM instances for assigning service invocations where the cache for deploying the needed container instance already exists.

- **Term 4** - *maximize future resource supply of already leased VMs:* The fourth term tries to maximize the future resource supply of already leased VM instances, by giving a preference to directly scheduling service invocations on VMs with a shorter remaining leasing duration $d_{(k_v,t)}$ compared to other VMs with already existing longer leasing durations. Although at time t this preference does not lead to any cost-based advantages, it assures that the already leased resources offer more long-running resources for future optimization periods. The term calculates the remaining leasing duration of all VMs using a weighting factor. As we want to give a preference to deploying containers on VM instances with a small leasing duration, we minimize the term to maximize the future remaining leasing durations of available resources.

- **Term 5** - *minimize sum of unused present VM resources:* The fifth term minimizes the sum of all leased but unused computational resource capacities in terms of CPU ($f^C_{k_v}$) and RAM ($f^R_{k_v}$) of all leased VM instances. Due to this term, our approach makes sure to not only schedule all service invocations that can not be delayed any further without leading to penalty costs but to also pre-schedule service invocations that may already be executed but have a distant deadline. We consider this term using weighting factors for CPU and RAM resources, such that we give more importance on the first two terms of our objective function.

- **Term 6** - *maximize the importance of scheduled service invocations:* The last term calculates the difference between the last possible scheduling deadline of the next schedulable steps when performing a worst-case analysis based on the current time. The term considers all process instances and aims at giving a scheduling preference to process steps with closer deadlines. The term is responsible for defining the importance of schedulable process steps that do not necessarily have a pressing process deadline on remaining free resources. Note that in contrast to the related work [63], we do consider the actual enactment deadline for the next schedulable steps based on the worst-case analysis instead of simply comparing the overall process deadlines. We also define the term such that past process deadlines can be considered and steps that are past their deadline receive a higher scheduling importance. We consider a weighting factor ω_{DL} to assign a lower weight to the term.

A.3 Constraints

Constraint A.2 makes sure that deadlines are considered for each process instance i_p. The constraint does not guarantee that deadlines are not violated, but it defines the penalty times $e^p_{i_p}$ of process instances which are subject to minimization in the presented term 2 of the objective function. The constraint demands that the current time plus the remaining worst-case execution time of process instances is smaller or equal to the defined deadline plus the potential penalty time which occurs when process instances finish after their deadline. This constraint considers the remaining execution time including the time of process steps that are being scheduled during the current time period and process steps that have already been scheduled in previous periods and are still running.

$$\tau_t + ex^{run}_{j_{i_p}} + ex^{*}_{j_{i_p}} + e_{i_p} \leqslant DL_{i_p} + e^{p}_{i_p} \tag{A.2}$$

Constraint A.3 helps in defining the start of the next optimization period τ_{t+1}, by demanding that the start of the next optimization period τ_{t+1} plus the remaining worst-case execution time of process instances is smaller or equal to the respectively defined process deadlines plus the possible penalty times. It should be noted that in contrast to related work [63] our approach only considers the worst-case execution time of all process steps that have not been scheduled yet until τ_t and have to be scheduled in a later optimization period τ_{t+1} or later for the calculation of the start time of the next optimization period τ_{t+1}. At this stage, the approach especially does not consider the execution time of currently running process steps or process steps that are scheduled for execution in τ_t, as the earliest possible time when next steps of a process instance can be scheduled is when the previous steps, already running or scheduled in τ_t, are finished. Adding the remaining time of the currently running or scheduled steps to the left side of Equation A.3 can potentially lead to triggering premature optimization rounds at a $t+1$ where predecessor steps of the next schedulable steps are not finished yet (and hence no new steps can be scheduled).

Constraint A.4 makes sure that the next optimization time period τ_{t+1} is defined to be in the future. We use a value $\epsilon > 0$ to avoid optimization deadlocks arising from a too small value for τ_{t+1}.

The start of the next optimization period is directly calculated as a result of the previous optimization rounds. Nevertheless, as this optimization model only refers to one optimization round and only operates based on the current knowledge, the optimization can also be triggered by other events, like newly incoming requests or finalized process steps and run earlier than the here calculated time $t + 1$.

$$\tau_{t+1} + e_{i_p} \leqslant DL_{i_p} + e^{p}_{i_p} \tag{A.3}$$

$$\tau_{t+1} \geqslant \tau_t + \epsilon \tag{A.4}$$

Equation A.5 calculates the remaining execution duration for all process instances i_p. The remaining execution duration e_{i_p} does not include the execution and deployment time of the process steps that will be scheduled for the current optimization period τ_t, nor the remaining execution time of the currently already running steps.

$$e_{i_p} = e^{seq}_{i_p} + e^{L_a}_{i_p} + e^{L_x}_{i_p} + e^{RL}_{i_p} \tag{A.5}$$

The overall remaining execution duration e_{i_p} of a process instance i_p is the sum of the execution times of the different process patterns that the process instance is composed of. We aggregate the upper bound execution times of the different process patterns, following the approach by Jäger et al. [75]. The worst-case execution times for sequences are defined in Equation A.6, for AND-blocks in Equation A.7, for XOR-blocks in Equation A.8 and for Repeat Loop blocks in Equation A.9. For AND-blocks the longest remaining execution time has to be considered naturally. As we are using a worst-case analysis, we also consider the longest path for the calculation of the remaining execution time of XOR-blocks. For Repeat Loop blocks the sub path execution time is multiplied by the maximal possible amount of repetitions re.

Equations A.6 - A.9 consider the overall remaining execution time of all not yet running process steps for each process instance i_p as defined by the two helper variables $\hat{e}^s_{i_p}$ for sequences and $\hat{e}^l_{i_p}$ for XOR- and AND-blocks in Equations A.11 and A.12. Since we perform a worst-case analysis, both equations consider the worst-case VM startup time, the time for pulling the respective Docker image onto the running VM instance and the time for starting a new container instance for each remaining process step j_{i_p}. If a process step $j^*_{i_p}$ is being scheduled in the current time period τ_t, Equations A.6 - A.9 subtract the worst-case execution time of the scheduled steps as defined in Equation A.10 from the overall remaining execution time.

$$e^{seq}_{i_p} = \begin{cases} \hat{e}^s_{i_p} - ex_{j^*_{i_p}} & \text{, if } x_{(j^*_{i_p},k_v,t)} = 1 \\ \hat{e}^s_{i_p} & \text{, otherwise} \end{cases} \tag{A.6}$$

$$e^{L_a}_{i_p} = \begin{cases} \max\limits_{l \in La}(\hat{e}^l_{i_p} - ex_{j^*_{i_p}}) & \text{, if } x_{(j^*_{i_p},k_v,t)} = 1 \\ \max\limits_{l \in La}(\hat{e}^l_{i_p}) & \text{, otherwise} \end{cases} \tag{A.7}$$

$$e^{L_x}_{i_p} = \begin{cases} \max\limits_{l \in Lx}(\hat{e}^l_{i_p} - ex_{j^*_{i_p}}) & \text{, if } x_{(j^*_{i_p},k_v,t)} = 1 \\ \max\limits_{l \in Lx}(\hat{e}^l_{i_p}) & \text{, otherwise} \end{cases} \tag{A.8}$$

$$e^{RL}_{i_p} = \begin{cases} re * \hat{e}^s_{i_p} - ex_{j^*_{i_p}} & \text{, if } x_{(j^*_{i_p},k_v,t)} = 1 \\ re * \hat{e}^s_{i_p} & \text{, otherwise} \end{cases} \tag{A.9}$$

$$ex_{j^*_{i_p}} = \sum_{v \in V} \sum_{k_v \in K_v} \left((e_{j^*_{i_p}} + \Delta_{c_{st_j}} + \Delta_{st_j} + \Delta) * x_{(j^*_{i_p},k_v,t)} \right) \tag{A.10}$$

$$\hat{e}^s_{i_p} = \sum_{j_{i_p} \in J^{seq}_{i_p}} (e_{j_{i_p}} + \Delta_{c_{st_j}} + \Delta_{st_j} + \Delta) \tag{A.11}$$

$$\hat{e}^l_{i_p} = \sum_{j_{i_p} \in J^l_{i_p}} (e_{j_{i_p}} + \Delta_{c_{st_j}} + \Delta_{st_j} + \Delta) \tag{A.12}$$

Constraints A.13 and A.14 consider for each VM instance k_v the sum of CPU and RAM resources required by all service invocations that either already run or are scheduled to run in any container on the VM instance in τ_t. These resources need to be smaller or equal to the resource supply in terms of CPU (s^C_v) and RAM (s^R_v) that is offered by VM instances of type v.

$$\sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in (J^*_{i_p} \cup J^{run}_{i_p})} (r^C_{j_{i_p}} * x_{(j_{i_p},k_v,t)}) \leqslant s^C_v \tag{A.13}$$

$$\sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in (J^*_{i_p} \cup J^{run}_{i_p})} (r^R_{j_{i_p}} * x_{(j_{i_p},k_v,t)}) \leqslant s^R_v \tag{A.14}$$

Constraints A.15 and A.16 define for each VM instance $v \in V, k_v \in K_v$ the remaining free capacities in terms of CPU and RAM that are not directly allocated to service instances (which are later packed into container instances). As expressed by the variable $g_{(k_v,t)} \in \{0,1\}$, which is further defined in Constraints A.17 and A.18, we only consider VM instances that are either already running or are being leased in τ_t.

$$g_{(k_v,t)} * s^C_v - \sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in (J^*_{i_p} \cup J^{run}_{i_p})} (r^C_{j_{i_p}} * x_{(j_{i_p},k_v,t)}) \leqslant f^C_{k_v} \tag{A.15}$$

$$g_{(k_v,t)} * s^R_v - \sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in (J^*_{i_p} \cup J^{run}_{i_p})} (r^R_{j_{i_p}} * x_{(j_{i_p},k_v,t)}) \leqslant f^R_{k_v} \tag{A.16}$$

Constraints A.17 and A.18 define the value of the helper variable $g_{(k_v,t)} \in \{0,1\}$ which takes the value of 1 if a VM instance is either already running ($\beta_{k_v} = 1$) or being leased for at least one BTU ($y_{(k_v,t)} \geq 1$) in τ_t. $g_{(k_v,t)}$ is restricted to be a boolean variable in Constraint A.29.

$$g_{(k_v,t)} \leqslant \beta_{(k_v,t)} + y_{(k_v,t)} \tag{A.17}$$

$$\beta_{(k_v,t)} + y_{(k_v,t)} \leqslant BTU^{max} * g_{(k_v,t)} \tag{A.18}$$

Constraint A.19 makes sure that for all VM instances $v \in V, k_v \in K_v$ a VM instance k_v will be leased or is running if service invocations are to be placed (within later defined containers) on it. Therefore, we calculate the number of process steps that are scheduled or already running on k_v and make use of our helper variable $g_{(k_v,t)}$ as defined in Constraints A.17 and A.18, as well as a helper variable $(M * N)$ that presents an arbitrarily large number, e.g., 100*10,000, as an upper bound of possible container deployments per VM instance times possible parallel service invocations per container instance.

$$\sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in (J^*_{i_p} \bigcup J^{run}_{i_p})} x_{(j_{i_p},k_v,t)} \leqslant g_{(k_v,t)} * (M * N) \tag{A.19}$$

Constraint A.20 makes sure that for all VM instances $v \in V, k_v \in K_v$ and all running or schedulable process steps $j_{i_p} \in (J^*_{i_p} \bigcup J^{run}_{i_p})$, the scheduled service invocations j_{i_p} will not be moved to other VM instances. On the lefthand side, we consider the remaining execution time of a process step j_{i_p} for each process step scheduled on a VM instance ($x_{(j_{i_p},k_v,t)} = 1$). In case that the process step j_{i_p} is not running yet, we also have to consider the time to pull a new image and starting the needed container, if the image does not exist and the container is not running on the scheduled VM instance ($z_{(st_j,k_v,t)} = 0$). Furthermore, if the required VM instance k_v is not up and running yet ($\beta_{(k_v,t)} = 0$) , we also need to add the time for starting the new VM instance k_v to our calculation of the remaining execution time. We demand the calculated remaining execution time to be smaller or equal to the remaining leasing duration $d_{(k_v,t)}$ of the allocated VM instance k_v. If the currently remaining leasing duration is smaller, the corresponding VM instance k_v needs to be leased for $y_{(k_v,t)}$ additional BTUs to satisfy the equation.

$$\begin{aligned} \big(e_{j_{i_p}} + (\Delta_{c_{st_j}} + \Delta_{st_j}) * (1 - z_{(st_j,k_v,t)}) + \Delta * (1 - \beta_{(k_v,t)})\big) * x_{(j_{i_p},k_v,t)} \\ \leqslant d_{(k_v,t)} * \beta_{(k_v,t)} + BTU_{k_v} * y_{(k_v,t)} \end{aligned} \tag{A.20}$$

Similar to Constraint A.20, Constraint A.21 together with Constraint A.24 ensure that all service invocations that are already running in a container on a certain VM instance at the start of τ_t, will not be migrated to another VM instance (or container) during their invocation. We explicitly reference the already assigned VM instance k_v by the additional indices in Constraint A.21 and do not need to consider additional times for pulling container images or starting VM instances.

$$e^{run_{k_v}}_{j_{i_p}} \leqslant d_{(k_v,t)} * \beta_{(k_v,t)} + BTU_{k_v} * y_{(k_v,t)} \tag{A.21}$$

Constraint A.22 defines the variable $\gamma_{(v,t)}$ for all VM types $v \in V$. $\gamma_{(v,t)}$ indicates the total number of BTUs to lease VM instances of type v, meaning it includes the decisions which VM instances k_v to lease and for how long (how many BTUs) to lease them. To define $\gamma_{(v,t)}$, we build the sum over all individually leased BTUs for each VM instance of type v.

$$\sum_{k_v \in K_v} y_{(k_v,t)} \leqslant \gamma_{(v,t)} \tag{A.22}$$

Constraint A.23 demands for all next schedulable process steps $j_{i_p} \in J^*_{i_p}$ that each service invocation is scheduled on only one VM instance (and later executed within only one container instance) or postponed for a later optimization period.

$$\sum_{v \in V} \sum_{k_v \in K_v} \leqslant 1 \tag{A.23}$$

Constraint A.24 makes sure that all steps currently running on a VM instance k_v remain on the same VM instance. This is achieved by setting the decision variable $x_{(j_{i_p},k_v,t)}$ for each already running service invocation $j^{run}_{i_p}$ on a VM instance k_v to 1. This constraint corresponds to inheriting the scheduling decision already made in a previous period.

$$x_{(j^{run}_{i_p},k_v,t)} = 1 \tag{A.24}$$

Constraint A.25 restricts the decision variable $x_{(j_{i_p},k_v,t)}$ that decides whether or not to schedule a service invocation j_{i_p} on a certain VM instance k_v for all $p \in P, i_p \in I_p, j_{i_p} \in J^*_{i_p}$ to be a boolean.

$$x_{(j_{i_p},k_v,t)} \in \{0,1\} \tag{A.25}$$

Constraint A.26 restricts the decision variable $y_{(k_v,t)}$ that decides for how many additional BTUs to lease a VM instance k_v at time t for all $v \in V, k_v \in K_v$ to be a positive integer ≥ 0.

$$y_{(k_v,t)} \in \mathbb{N}^+_0 \tag{A.26}$$

Constraint A.27 restricts the decision variable $\gamma_{(v,t)}$ that decides for how many total BTUs to lease any VM instances of type v at time t for all $v \in V$ to be a positive integer ≥ 0.

$$\gamma_{(v,t)} \in \mathbb{N}^+_0 \tag{A.27}$$

Constraint A.28 restricts the variable that indicates the penalty execution time $e^p_{i_p}$ to be a positive real number.

$$e^p_{i_p} \in \mathbb{R}^+ \tag{A.28}$$

Constraint A.29 restricts the variable $g_{(k_v,t)}$ to be a boolean. The variable has been defined in Constraints A.17 and A.18 and takes the value of 1 if a VM instance is either already running ($\beta_{k_v} = 1$) or being leased for at least one BTU ($y_{(k_v,t)} \geq 1$) in τ_t.

$$g_{(k_v,t)} \in \{0, 1\} \tag{A.29}$$

APPENDIX B

Process Model Collection

For performing our evaluation in Section 6.4 we use the following process models listed in this chapter.

Figure B.1: Process Model 1

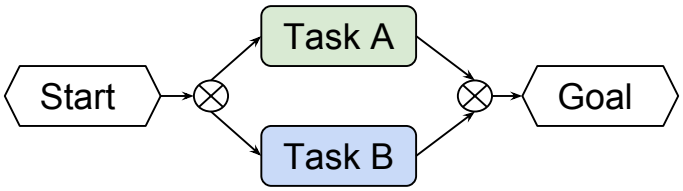


Figure B.2: Process Model 2

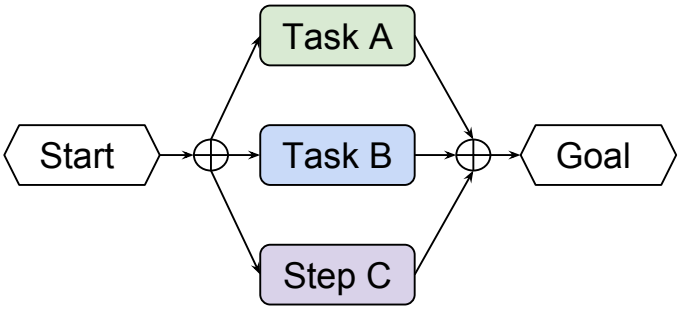


Figure B.3: Process Model 3

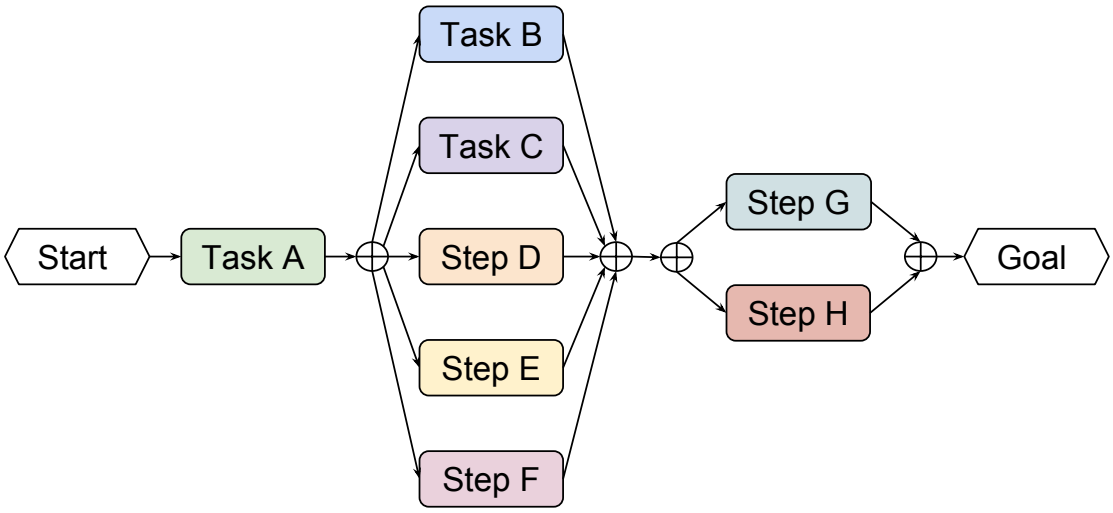


Figure B.4: Process Model 4

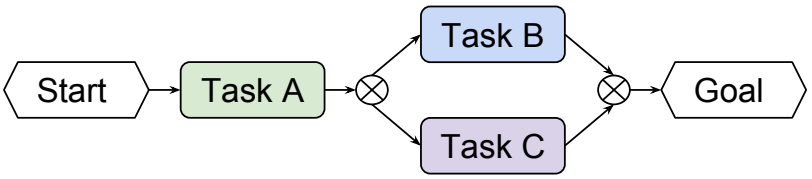


Figure B.5: Process Model 5

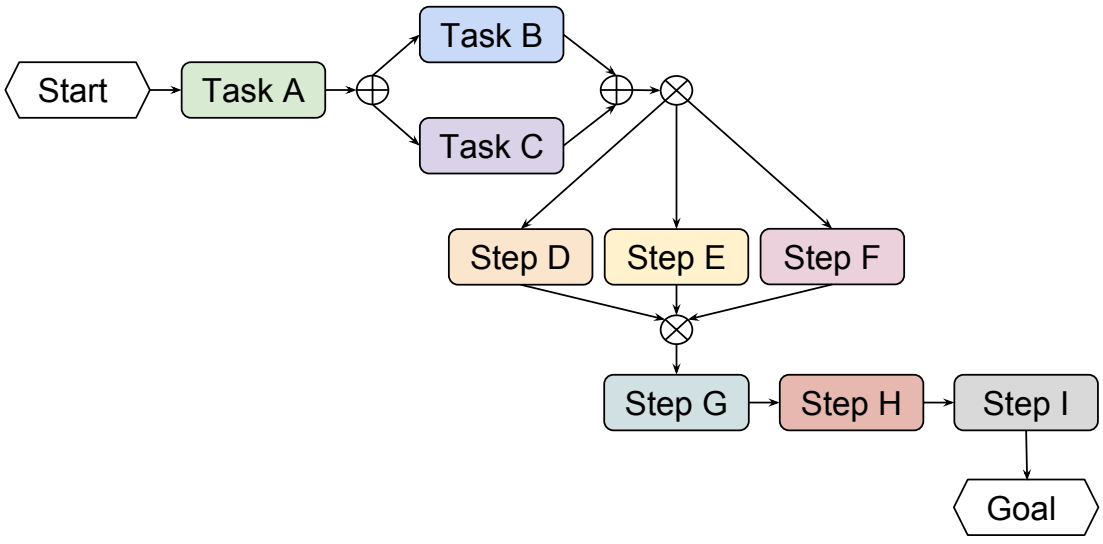


Figure B.6: Process Model 6

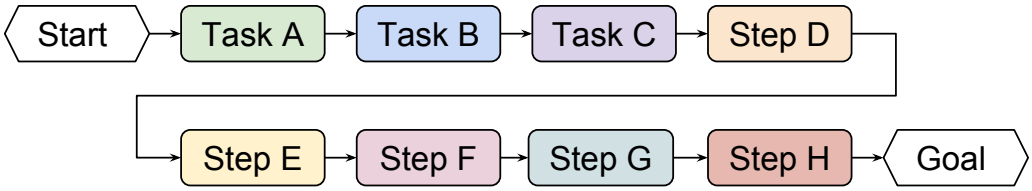


Figure B.7: Process Model 7

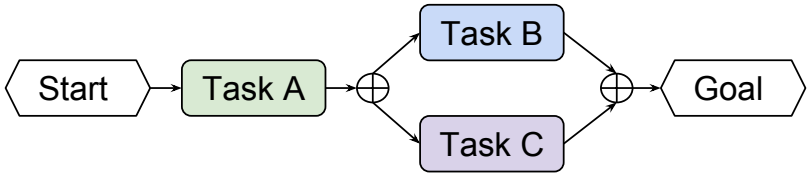


Figure B.8: Process Model 8

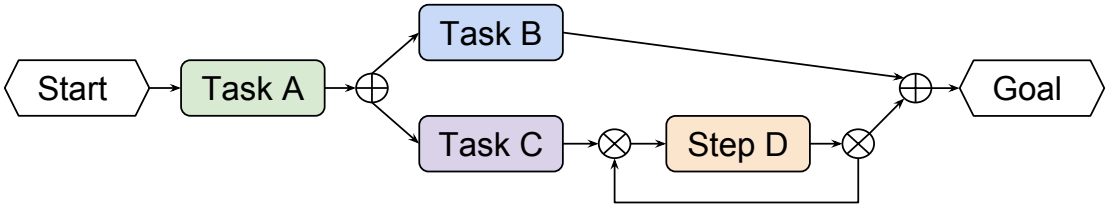


Figure B.9: Process Model 9

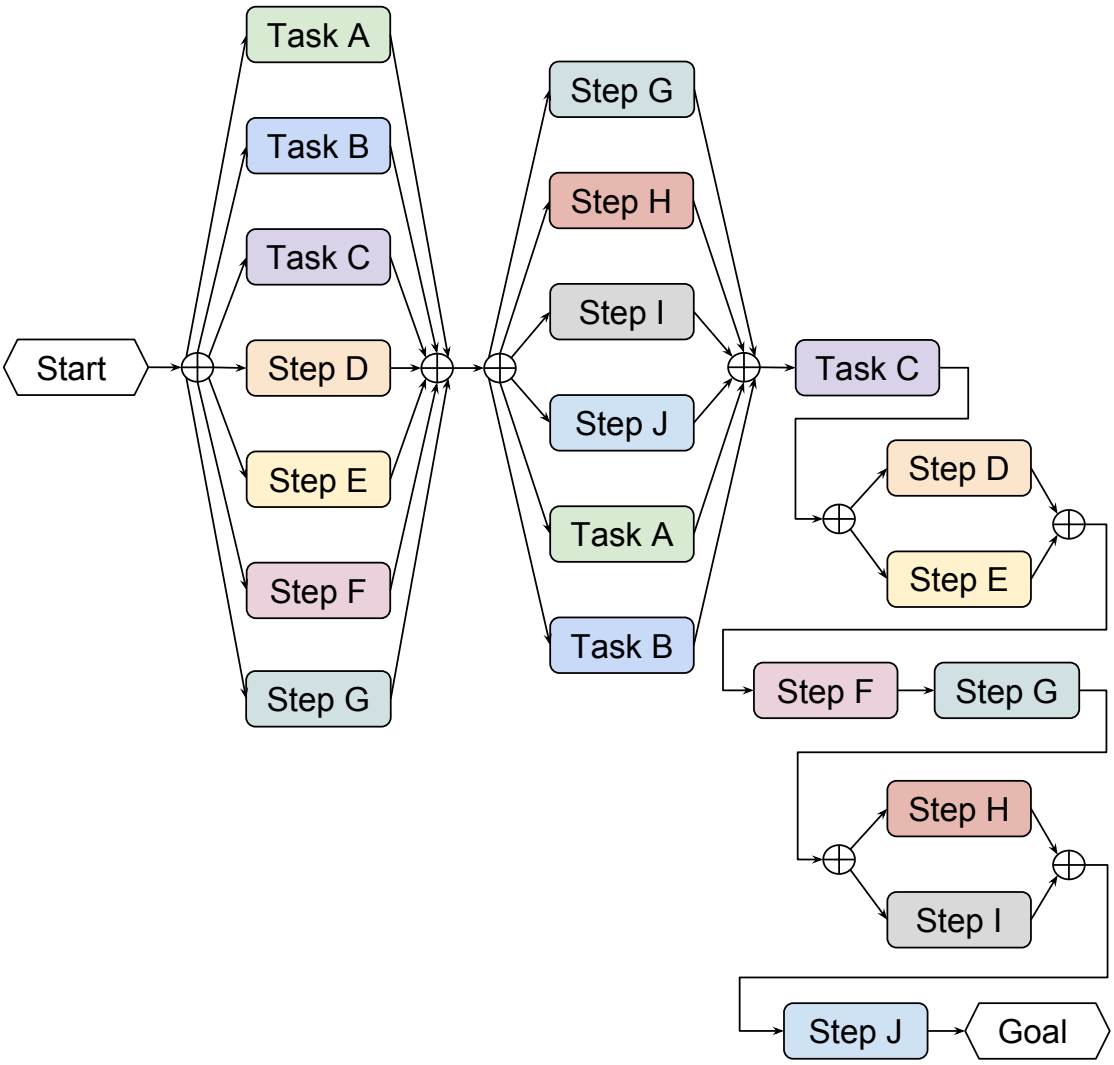


Figure B.10: Process Model 10

List of Acronyms

AI	Artificial Intelligence. 15, 17, 173
API	Application Programming Interface. 106, 173
AWS	Amazon Web Services. 23, 30, 35, 136, 173
BPaaS	Business Process as a Service. 34, 173
BPM	Business Process Management. x, xiii, 2, 3, 6, 9, 11, 32–34, 47, 53, 56, 59, 173
BPMS	Business Process Management System. x, xix, 5, 33, 34, 36, 54–57, 59, 60, 63, 64, 66, 74, 76, 77, 93, 105, 106, 108, 110, 117, 119–122, 129, 133, 135, 137, 155, 173
BTU	Billing Time Unit. xi, xiv, xv, 62, 65, 70, 73, 76, 79, 84–88, 92, 94, 97, 101–103, 108, 117, 126, 131, 132, 136, 148–150, 156, 161, 164, 168–171, 173
CoT	Cloud of Things. 14, 173
DaaS	Data as a Service. 18, 173
EC2	Elastic Compute Cloud. 7, 8, 18, 27, 30, 35, 48, 51, 156, 173
FFSIPP	Four Fold Service Instance Placement Problem. 78, 173
GB	Gigabyte. 26, 173
HDD	Hard Disc Drive. 16, 51, 173
I/O	Input/Output. 16, 23, 51, 173
IaaS	Infrastructure as a Service. 18, 19, 49, 51, 173
ILP	Integer Linear Programming. 48, 52, 173
IoT	Internet of Things. 5, 14, 15, 157, 173
IT	Information Technology. 32, 173
KVM	Kernel-based Virtual Machine. 22, 23, 173
M2M	Machine to Machine. 14, 173

MB Megabyte. 26, 173
MILP Mixed Integer Linear Programming. x, 43, 44, 55, 60, 78, 90–93, 108, 110, 111, 113, 114, 118, 128, 137, 141, 151, 154, 156, 159, 173
ML Machine Learning. 15, 17, 38, 49, 173

NASA National Aeronautics and Space Administration. 13, 173
NIST National Institute of Standards and Technology. 12, 173

OS Operating System. 2, 18, 19, 21–26, 30, 50, 173

PaaS Platform as a Service. 18, 19, 24, 50, 51, 60, 173

QCaaS Quantum Computing as a Service. 17, 173
QoS Quality of Service. 6–8, 19, 20, 31–33, 35, 36, 44, 49, 52–54, 56, 61, 67, 89, 156, 173

RoI Return on Investment. 43, 173

S3 Simple Storage Service. 18, 173
SaaS Software as a Service. 18, 19, 44, 49, 50, 173
SLA Service Level Agreement. 2, 19, 20, 48, 49, 52, 54, 56, 57, 61, 63, 64, 66, 74, 78, 79, 135, 136, 139, 140, 142–151, 154, 164, 173
SLO Service Level Objective. 20, 61, 74, 173
SOA Service-Oriented Architecture. 2, 6, 47, 75, 89, 173
SOC Service-Oriented Computing. 173
SWF Scientific Workflow. 6, 53, 55–57, 173

VM Virtual Machine. x, xiv, xv, 2, 6–8, 18, 21–26, 29, 30, 43, 48–52, 54–57, 60, 62–67, 70–74, 76–79, 82–94, 99, 102, 103, 105, 108, 110–117, 121, 126–138, 141–143, 145, 147, 149, 154, 156, 157, 160–165, 167–171, 173

WS Web Service. 2, 173

Bibliography

[1] Mohammad Aazam, Imran Khan, Aymen Abdullah Alsaffar, and Eui-Nam Huh. Cloud of things: Integrating internet of things and cloud computing and the issues involved. In *Applied Sciences and Technology (IBCAST), 2014 11th International Bhurban Conference on*, pages 414–419. IEEE, 2014.

[2] Moustafa Abdelbaky, Javier Diaz-Montes, Manish Parashar, Merve Unuvar, and Malgorzata Steinder. Docker containers across multiple clouds and data centers. In *Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on*, pages 368–371. IEEE, 2015.

[3] Saeid Abrishami, Mahmoud Naghibzadeh, and Dick HJ Epema. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Generation Computer Systems*, 29(1):158–169, 2013.

[4] Rafael Accorsi. Business process as a service: Chances for remote auditing. In *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pages 398–403. IEEE, 2011.

[5] Ejaz Ahmed, Ibrar Yaqoob, Abdullah Gani, Muhammad Imran, and Mohsen Guizani. Internet-of-things-based smart environments: state of the art, taxonomy, and open research challenges. *IEEE Wireless Communications*, 23(5):10–16, 2016.

[6] Sanjay P Ahuja, Sindhu Mani, and Jesus Zambrano. A survey of the state of cloud computing in healthcare. *Network and Communication Technologies*, 1(2):12, 2012.

[7] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK Users' guide*. SIAM, 1999.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[9] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[10] Bryan Bergeron. *Essentials of shared services*, volume 26. John Wiley & Sons, 2002.

[11] Data Science Department Berkeley School of Information. https://datascience.berkeley.edu/moores-law-processing-power/. Online: https://datascience.berkeley.edu/moores-law-processing-power/, 2014, last visited: 2017-08-29.

[12] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[13] Dimitri P Bertsekas. *Nonlinear programming*. Athena scientific Belmont, 1999.

[14] Kahina Bessai, Samir Youcef, Ammar Oulamara, Claude Godart, and Selmin Nurcan. Business process scheduling strategies in cloud environments with fairness metrics. In *Services Computing (SCC), 2013 IEEE International Conference on*, pages 519–526. IEEE, 2013.

[15] Maurice Anthony Biot. Theory of elasticity and consolidation for a porous anisotropic solid. *Journal of Applied Physics*, 26(2):182–185, 1955.

[16] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[17] Alessio Botta, Walter De Donato, Valerio Persico, and Antonio Pescapé. Integration of cloud computing and internet of things: a survey. *Future Generation Computer Systems*, 56:684–700, 2016.

[18] James Bottomley, Pavel Emelyanov, Antti Kantee, Justin Cormack, Dan Klein, Dina Bester, Mathew Monroe, and Andy Seely. Operating systems. *;login:*, 39(4):6–10, 2014.

[19] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger M Kienle, Marin Litoiu, Hausi A Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. *Software engineering for self-adaptive systems*, 5525:48–70, 2009.

[20] Kurt Bryan and Tanya Leise. The $25,000,000,000 eigenvector: The linear algebra behind google. *Siam Review*, 48(3):569–581, 2006.

[21] Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[22] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N Calheiros. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 13–31. Springer, 2010.

[23] Rajkumar Buyya, Chee Shin Yeo, and Srikumar Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on*, pages 5–13. IEEE, 2008.

[24] 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. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[25] Eun-Kyu Byun, Yang-Suk Kee, Jin-Soo Kim, and Seungryoul Maeng. Cost optimized provisioning of elastic resources for application workflows. *Future Generation Computer Systems*, 27(8):1011–1026, 2011.

[26] X-D Cai, Dian Wu, Z-E Su, M-C Chen, X-L Wang, Li Li, N-L Liu, C-Y Lu, and J-W Pan. Entanglement-based machine learning on a quantum computer. *Physical review letters*, 114(11):110504, 2015.

[27] Giuseppe C Calafiore and Laurent El Ghaoui. *Optimization models*. Cambridge university press, 2014.

[28] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1069–1075. ACM, 2005.

[29] IBM Knowledge Center. Numeric difficulties. Online: https://www.ibm.com/support/knowledgecenter/pl/SSSA5P_12.7.1/ilog.odms.cplex.help/CPLEX/UsrMan/topics/cont_optim/simplex/20_num_difficulty.html, last visited: 2017-08-29.

[30] Jens Clausen. Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen*, pages 1–30, 1999.

[31] Fernando J Corbató, Marjorie Merwin-Daggett, and Robert C Daley. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 335–344. ACM, 1962.

[32] James W Cortada. *The Essential Manager: How to Thrive in the Global Information Jungle*. John Wiley & Sons, 2015.

[33] Robert J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[34] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, and Vignesh Kalyanaraman. Blockchain technology: Beyond bitcoin. *Applied Innovation*, 2:6–10, 2016.

[35] Thomas Curran, Gerhard Keller, and Andrew Ladd. *SAP R/3 business blueprint: understanding the business process reference model*. Prentice-Hall, Inc., 1997.

[36] George B Dantzig, Alex Orden, Philip Wolfe, et al. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

[37] Simon J Devitt. Performing quantum computing experiments in the cloud. *Physical Review A*, 94(3):032329, 2016.

[38] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[39] Jack J Dongarra, Cleve B Moler, James R Bunch, and Gilbert W Stewart. *LINPACK users' guide*. SIAM, 1979.

[40] Giovanni Dosi. Technological paradigms and technological trajectories: a suggested interpretation of the determinants and directions of technical change. *Research policy*, 11(3):147–162, 1982.

[41] Charalampos Doukas and Ilias Maglogiannis. Bringing iot and cloud computing towards pervasive healthcare. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 922–926. IEEE, 2012.

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

[43] Marlon Dumas, Marcello La Rosa, Jan Mendling, Hajo A Reijers, et al. *Fundamentals of business process management*, volume 1. Springer, 2013.

[44] Schahram Dustdar, Yike Guo, Benjamin Satzger, and Hong-Linh Truong. Principles of elastic processes. *IEEE Internet Computing*, 15(5), 2011.

[45] Thomas Erl. *Service-oriented architecture*. Prentice Hall Englewood Cliffs, 2004.

[46] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 171–172. IEEE, 2015.

[47] Richard Phillips Feynman. *The principle of least action in quantum mechanics*. PhD thesis, Princeton University Princeton, New Jersey, 1942.

[48] Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

[49] Luciano García-Bañuelos, Alexander Ponomarev, Marlon Dumas, and Ingo Weber. Optimized execution of business processes on blockchain. In *International Conference on Business Process Management*, pages 130–146. Springer, 2017.

[50] Wolfgang Gerlach, Wei Tang, Kevin Keegan, Travis Harrison, Andreas Wilke, Jared Bischof, Mark D'Souza, Scott Devoid, Daniel Murphy-Olson, Narayan Desai, et al. Skyport: container-based execution environment management for multi-cloud scientific workflows. In *Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds*, pages 25–32. IEEE Press, 2014.

[51] Martin Glowik, Luca Mentuccia, and Marcello Tamietti. A new era for the automotive industry. Online: https://www.accenture.com/t20150914T170053__w__/us-en/_acnmedia/Accenture/Conversion-Assets/DotCom/Documents/Global/PDF/Industries_18/Accenture-Cloud-Automative-PoV.pdf, 2015, last visited: 2017-08-29.

[52] Charles David Graziano. A performance analysis of xen and kvm hypervisors for hosting the xen worlds project. 2011.

[53] Salvatore Greco, J Figueira, and M Ehrgott. Multiple criteria decision analysis. *Springer's International series*, 2005.

[54] BMW Group. Annual report 2015. Online: https://www.bmwgroup.com/content/dam/bmw-group-websites/bmwgroup_com/ir/downloads/en/2016/Annual_Report_2015.pdf, 2015, last visited: 2017-08-29.

[55] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[56] Akshay Gupte, Shabbir Ahmed, Myun Seok Cheon, and Santanu Dey. Solving mixed integer bilinear problems using milp formulations. *SIAM Journal on Optimization*, 23(2):721–744, 2013.

[57] P Harsh, K Benz, I Trajkovska, A Edmonds, P Comi, and T Bohnert. A highly available generic billing architecture for heterogenous mobile cloud services. In *Proceedings of the International Conference on Grid Computing and Applications (GCA)*, page 1, 2014.

[58] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *FAST*, volume 16, pages 181–195, 2016.

[59] Moeen Hassanalieragh, Alex Page, Tolga Soyata, Gaurav Sharma, Mehmet Aktas, Gonzalo Mateos, Burak Kantarci, and Silvana Andreescu. Health monitoring and management using internet-of-things (iot) sensing with cloud-based processing: Opportunities and challenges. In *Services Computing (SCC), 2015 IEEE International Conference on*, pages 285–292. IEEE, 2015.

[60] Parisa Heidari, Yves Lemieux, and Abdallah Shami. Qos assurance with light virtualization-a survey. In *Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on*, pages 558–563. IEEE, 2016.

[61] Nikolas Roman Herbst, Samuel Kounev, and Ralf H Reussner. Elasticity in cloud computing: What it is, and what it is not. In *ICAC*, pages 23–27, 2013.

[62] Philipp Hoenisch, Christoph Hochreiner, Dieter Schuller, Stefan Schulte, Jan Mendling, and Schahram Dustdar. Cost-efficient scheduling of elastic processes in hybrid clouds. In *8th International Conference on Cloud Computing*, pages 17–24. IEEE, 2015.

[63] Philipp Hoenisch, Dieter Schuller, Stefan Schulte, Christoph Hochreiner, and Schahram Dustdar. Optimization of complex elastic processes. *IEEE Transactions on Services Computing*, 9(5):700–713, 2016.

[64] Philipp Hoenisch, Stefan Schulte, and Schahram Dustdar. Workflow scheduling and resource allocation for cloud-based execution of elastic processes. In *Service-Oriented Computing and Applications (SOCA), 2013 IEEE 6th International Conference on*, pages 1–8. IEEE, 2013.

[65] Philipp Hoenisch, Ingo Weber, Stefan Schulte, Liming Zhu, and Alan Fekete. Four-fold auto-scaling on a contemporary deployment platform using docker containers. In *International Conference on Service-Oriented Computing (ICSOC)*, pages 316–323. Springer, 2015.

[66] Ingo Hofacker and Rudolf Vetschera. Algorithmical approaches to business process design. *Computers & Operations Research*, 28(13):1253–1275, 2001.

[67] Christina Hoffa, Gaurang Mehta, Tim Freeman, Ewa Deelman, Kate Keahey, Bruce Berriman, and John Good. On the use of cloud computing for scientific workflows. In *4th International Conference on eScience*, pages 640–645. IEEE, 2008.

[68] He-Liang Huang, You-Wei Zhao, Tan Li, Feng-Guang Li, Yu-Tao Du, Xiang-Qun Fu, Shuo Zhang, Xiang Wang, and Wan-Su Bao. Homomorphic encryption experiments on ibm's cloud quantum computing platform. *arXiv preprint arXiv:1612.02886*, 2016.

[69] Michael N Huhns and Munindar P Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[70] IBM. Frequently asked questions. Online: http://www-03.ibm.com/ibm/history/documents/pdf/faq.pdf, 2007, last visited: 2017-08-29.

[71] IBM. Cloud computing for banking. Online: https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=IBW03005USEN, 2013, last visited: 2017-08-29.

[72] IBM. Cloud computing for insurance. Online: https://public.dhe.ibm.com/common/ssi/ecm/ib/en/ibw03006usen/IBW03006USEN.PDF, 2013, last visited: 2017-08-29.

[73] SM Riazul Islam, Daehan Kwak, MD Humaun Kabir, Mahmud Hossain, and Kyung-Sup Kwak. The internet of things for health care: a comprehensive survey. *IEEE Access*, 3:678–708, 2015.

[74] Martin Gilje Jaatun, Siani Pearson, Frédéric Gittler, and Ronald Leenes. Towards strong accountability for cloud service providers. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 1001–1006. IEEE, 2014.

[75] Michael C Jaeger, Gregor Rojec-Goldmann, and Gero Muhl. Qos aggregation for web service composition using workflow patterns. In *Enterprise distributed object computing conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International*, pages 149–159. IEEE, 2004.

[76] Vipul Jain and Ignacio E Grossmann. Algorithms for hybrid milp/cp models for a class of optimization problems. *INFORMS Journal on computing*, 13(4):258–276, 2001.

[77] Ernst Juhnke, Tim Dornemann, David Bock, and Bernd Freisleben. Multi-objective scheduling of bpel workflows in geographically distributed clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 412–419. IEEE, 2011.

[78] Gideon Juve and Ewa Deelman. Scientific workflows and clouds. *ACM Crossroads*, 16(3):14–18, 2010.

[79] Michio Kaku. *Physics of the future*. Doubleday, 2011.

[80] Magnus Karlsson and Christos Karamanolis. Bounds on the replication cost for qos. *Technical report, Hewlett Packard Labs*, 2003.

[81] Alexander Keller and Heiko Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.

[82] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[83] Abdullah Konak, David W Coit, and Alice E Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, 2006.

[84] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 579–590. ACM, 2010.

[85] Andreas Kronabeter and Stefan Fenz. Cloud security and privacy in the light of the 2012 eu data protection regulation. In *International Conference on Cloud Computing*, pages 114–123. Springer, 2012.

[86] Ulrich Lampe, Thorsten Mayer, Johannes Hiemer, Dieter Schuller, and Ralf Steinmetz. Enabling cost-efficient software service distribution in infrastructure clouds at run time. In *International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–8. IEEE, 2011.

[87] Ulrich Lampe, Olga Wenge, Alexander Müller, and Ralf Schaarschmidt. On the relevance of security risks for cloud adoption in the financial industry. In *19th Americas Conference on Information Systems (AMCIS 2013)*. AIS, 2013.

[88] Frank Leymann and Dieter Roller. Production workflow: concepts and techniques. 2000.

[89] Han Li and Srikumar Venugopal. Using reinforcement learning for controlling an elastic web application hosting platform. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 205–208. ACM, 2011.

[90] Wubin Li, Ali Kanso, and Abdelouahed Gherbi. Leveraging linux containers to achieve high availability for cloud services. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 76–83. IEEE, 2015.

[91] Marin Litoiu, Murray Woodside, Johnny Wong, Joanna Ng, and Gabriel Iszlai. A business driven cloud optimization architecture. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 380–385. ACM, 2010.

[92] Haikun Liu, Hai Jin, Xiaofai Liao, Wei Deng, Bingsheng He, and Cheng-zhong Xu. Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines. *IEEE Transations on Parallel and Distributed Systems*, 26(5):1350–1363, 2015.

[93] Bertram Ludäscher, Mathias Weske, Timothy McPhillips, and Shawn Bowers. Scientific workflows: Business as usual? In *International Conference on Business Process Management (BPM)*, pages 31–47. Springer, 2009.

[94] Chris A Mack. Fifty years of moore's law. *IEEE Transactions on semiconductor manufacturing*, 24(2):202–207, 2011.

[95] Heike Mai, Bernhard Speyer, Deutsche Bank AG, and Ralf Hoffmann. It in banks: What does it cost? *Deutsche Bank Research*, 2012.

[96] Udi Manber and Peter Norvig. The power of the apollo missions in a single google search. Online: https://search.googleblog.com/2012/08/the-power-of-apollo-missions-in-single.html, 2012, last visited: 2017-08-29.

[97] Zoltán Ádám Mann. Allocation of virtual machines in cloud data centers—a survey of problem models and optimization algorithms. *ACM Computing Surveys (CSUR)*, 48(1):11, 2015.

[98] Ronny S Mans, Nick C Russell, Wil MP van der Aalst, Arnold J Moleman, and Piet JM Bakker. Schedule-aware workflow management systems. In *Transactions on Petri nets and other models of concurrency IV*, pages 121–143. Springer, 2010.

[99] John C McCallum. Price-performance of computer technology. In *The computer engineering handbook*. CRC Press, 2001.

[100] Sean McDaniel, Stephen Herbein, and Michela Taufer. A two-tiered approach to i/o quality of service in docker containers. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 490–491. IEEE, 2015.

[101] George R McGhee. *Convergent evolution: limited forms most beautiful*. MIT Press, 2011.

[102] Peter Mell and Timothy Grance. The nist definition of cloud computing. *NIST special publication*, 800:145, 2011.

[103] Jan Mendling, HMW Verbeek, Boudewijn F van Dongen, Wil MP van der Aalst, and Gustaf Neumann. Detection and prediction of errors in epcs of the sap reference model. *Data & Knowledge Engineering*, 64(1):312–329, 2008.

[104] Jan Mendling, Ingo Weber, Wil van der Aalst, Jan vom Brocke, Cristina Cabanillas, Florian Daniel, Soren Debois, Claudio Di Ciccio, Marlon Dumas, Schahram Dustdar, et al. Blockchains for business process management-challenges and opportunities. *arXiv preprint arXiv:1704.03610*, 2017.

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

[106] M. Michael, J. Moreira, D. Shiloach, and R. Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8. IEEE, 2007.

[107] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[108] Gordon E Moore. The experts look ahead. cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[109] Gordon E Moore. Progress in digital integrated electronics, digest of the 1975 international electron devices meeting, 1975.

[110] Timothy A Mousseau, Derek A Roff, et al. Natural selection and the heritability of fitness components. *Heredity*, 59(Pt 2):181–197, 1987.

[111] Katta G Murty. Optimization models for decision making: Volume. *University of Michigan, Ann Arbor*, 2003.

[112] Matteo Nardelli, Christoph Hochreiner, and Stefan Schulte. Elastic provisioning of virtual machines for container deployment. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 5–10. ACM, 2017.

[113] John C Nash. The (dantzig) simplex method for linear programming. *Computing in Science & Engineering*, 2(1):29–31, 2000.

[114] Nils J Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann, 2014.

[115] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

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

[117] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing*, 2017.

[118] Jian Pan, Yudong Cao, Xiwei Yao, Zhaokai Li, Chenyong Ju, Hongwei Chen, Xinhua Peng, Sabre Kais, and Jiangfeng Du. Experimental realization of quantum algorithm for solving linear systems of equations. *Physical Review A*, 89(2):022313, 2014.

[119] Suraj Pandey, Linlin Wu, Siddeswara Mayura Guru, and Rajkumar Buyya. A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments. In *24th International Conference on Advanced Information Networking and Applications*, pages 400–407. IEEE, 2010.

[120] Mike P Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12. IEEE, 2003.

[121] Beth Pariseau. Kvm reignites type 1 vs. type 2 hypervisor debate. Online: http://searchservervirtualization.techtarget.com/news/2240034817/KVM-reignites-Type-1-vs-Type-2-hypervisor-debate, 2011, last visited: 2017-08-29.

[122] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[123] James R Powell. The quantum limit to moore's law. *Proceedings of the IEEE*, 96(8):1247–1248, 2008.

[124] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *arXiv preprint arXiv:1609.09224*, 2016.

[125] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[126] Mijanur Rahaman and Md Masudul Islam. An overview on quantum computing as a service (qcaas): Probability or possibility. *doi*, 10(5815):16–22, 2016.

[127] Fahimeh Ramezani, Jie Lu, and Farookh Hussain. Task scheduling optimization in cloud computing applying multi-objective particle swarm optimization. In *International Conference on Service-Oriented Computing*, pages 237–251. Springer, 2013.

[128] Sumant Ramgovind, Mariki M Eloff, and Elme Smith. The management of security in cloud computing. In *Information Security for South Africa (ISSA), 2010*, pages 1–7. IEEE, 2010.

[129] IBM Press release. Ibm makes quantum computing available on ibm cloud to accelerate innovation. Online Press release: http://www-03.ibm.com/press/us/en/pressrelease/49661.wss, 2016, last visited: 2017-08-29.

[130] Paul Rimba, An Binh Tran, Ingo Weber, Mark Staples, Alexander Ponomarev, and Xiwei Xu. Comparing blockchain and cloud services for business process execution. In *Software Architecture (ICSA), 2017 IEEE International Conference on*, pages 257–260. IEEE, 2017.

[131] Michael Rosemann and Jan vom Brocke. The six core elements of business process management. In *Handbook on business process management 1*, pages 105–122. Springer, 2015.

[132] Guillaume Rosinosky, Samir Youcef, and François Charoy. Efficient migration-aware algorithms for elastic bpmaas. 2017.

[133] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27, 1995.

[134] Greg Satell. Ibm's new quantum computing move marks a new era for cloud computing. Online: https://www.forbes.com/sites/gregsatell/2016/05/04/ibms-new-quantum-computing-move-marks-a-new-era-for-cloud-computing/#2aac00466b87, 2016, last visited: 2017-08-29.

[135] KA Scarfone. *Guide to security for full virtualization technologies*, volume 800. DIANE Publishing, 2011.

[136] Stefan Schulte, Philipp Hoenisch, Christoph Hochreiner, Schahram Dustdar, Matthias Klusch, and Dieter Schuller. Towards process support for cloud manufacturing. In *Enterprise Distributed Object Computing Conference (EDOC), 2014 IEEE 18th International*, pages 142–149. IEEE, 2014.

[137] Stefan Schulte, Philipp Hoenisch, Srikumar Venugopal, and Schahram Dustdar. Introducing the vienna platform for elastic processes. In *Performance Assessment and Auditing in Service Computing Workshop (PAASC 2012) at International Conference on Service-Oriented Computing (ICSOC)*, pages 179–190. Springer, 2012.

[138] Stefan Schulte, Christian Janiesch, Srikumar Venugopal, Ingo Weber, and Philipp Hoenisch. Elastic business process management: State of the art and open challenges for bpm in the cloud. *Future Generation Computer Systems*, 46:36–50, 2015.

[139] Stefan Schulte, Dieter Schuller, Philipp Hoenisch, Urlich Lampe, Schahram Dustdar, and Ralf Steinmetz. Cost-driven optimization of cloud resource allocation for elastic processes. *International Journal of Cloud Computing*, 1(2):1–14, 2013.

[140] Aleksander Slominski, Vinod Muthusamy, and Rania Khalaf. Building a multi-tenant cloud service from legacy code with docker containers. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 394–396. IEEE, 2015.

[141] Brian T Smith, James M. Boyle, BS Garbow, Y Ikebe, VC Klema, and CB Moler. *Matrix eigensystem routines-EISPACK guide*, volume 6. Springer, 2013.

[142] J Cole Smith and Z Caner Taskin. A tutorial guide to mixed-integer programming models and solution techniques. *Optimization in Medicine and Biology*, pages 521–548, 2008.

[143] Kuyoro So. Cloud computing security issues and challenges. *International Journal of Computer Networks*, 3(5):247–55, 2011.

[144] Anja Strunk. Qos-aware service composition: A survey. In *8th European Conference on Web Services (ECOWS)*, pages 67–74. IEEE, 2010.

[145] Yu Sun, Jules White, Bo Li, Michael Walker, and Hamilton Turner. Automated qos-oriented cloud resource optimization using containers. *Automated Software Engineering*, 24(1):101–137, 2017.

[146] Melanie Swan. *Blockchain: Blueprint for a new economy*. O'Reilly Media, Inc., 2015.

[147] Claudia Szabo and Trent Kroeger. Evolving multi-objective strategies for task allocation of scientific workflows on public clouds. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 1–8. IEEE, 2012.

[148] Gerard J Tellis. The price elasticity of selective demand: A meta-analysis of econometric models of sales. *Journal of Marketing Research*, pages 331–341, 1988.

[149] James E Tomayko. Computers in spaceflight the nasa experience. *Computers in Spaceflight The NASA Experience*, 1, 1988.

[150] Andrea Tosatto, Pietro Ruiu, and Antonio Attanasio. Container-based orchestration in cloud: state of the art and challenges. In *Complex, Intelligent, and Software Intensive Systems (CISIS), 2015 Ninth International Conference on*, pages 70–75. IEEE, 2015.

[151] Lawrence J Trautman. Is disruptive blockchain technology the future of financial services? 2016.

[152] Ruben Van den Bossche, Kurt Vanmechelen, and Jan Broeckhove. Online cost-efficient scheduling of deadline-constrained workloads on hybrid clouds. *Future Generation Computer Systems*, 29(4):973–985, 2013.

[153] Wil MP van der Aalst, Arthur HM Ter Hofstede, Bartek Kiepuszewski, and Alistair Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[154] Wil MP Van Der Aalst, Arthur HM Ter Hofstede, and Mathias Weske. Business process management: A survey. In *International conference on business process management*, pages 1–12. Springer, 2003.

[155] Luis M Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.

[156] MingXue Wang, Kosala Yapa Bandara, and Claus Pahl. Process as a service distributed multi-tenant policy-based process runtime governance. In *Services computing (scc), 2010 ieee international conference on*, pages 578–585. IEEE, 2010.

[157] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, Quan Z Sheng, and Rajiv Ranjan. A taxonomy and survey of cloud resource orchestration techniques. *ACM Computing Surveys (CSUR)*, 50(2):26, 2017.

[158] Yi Wei and M Brian Blake. Adaptive service workflow configuration and agent-based virtual resource management in the cloud. In *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pages 279–284. IEEE, 2013.

[159] Yi Wei and M Brian Blake. Proactive virtualized resource management for service workflows in the cloud. *Computing*, 98(5):523–538, 2016.

[160] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M Göschka. On patterns for decentralized control in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II*, pages 76–107. Springer, 2013.

[161] Aaron Wright and Primavera De Filippi. Decentralized blockchain technology and the rise of lex cryptographia. 2015.

[162] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. Sla-based resource allocation for software as a service provider (saas) in cloud computing environments. In *11th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 195–204. IEEE, 2011.

[163] Bruno Xavier, Tiago Ferreto, and Luis Jersak. Time provisioning evaluation of kvm, docker and unikernels in a cloud platform. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 277–280. IEEE, 2016.

[164] Meng Xu, Lizhen Cui, Haiyang Wang, and Yanbing Bi. A multiple qos constrained scheduling strategy of multiple workflows for cloud computing. In *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*, pages 629–634. IEEE, 2009.

[165] Xin Xu, Huiqun Yu, and Xin Pei. A novel resource scheduling approach in container based clouds. In *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on*, pages 257–264. IEEE, 2014.

[166] Xiao Yue, Huiju Wang, Dawei Jin, Mingqiang Li, and Wei Jiang. Healthcare data gateways: found healthcare intelligence on blockchain with novel privacy risk control. *Journal of medical systems*, 40(10):218, 2016.

[167] Zeratul Izzah Mohd Yusoh and Maolin Tang. Composite saas placement and resource optimization in cloud computing using evolutionary algorithms. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 590–597. IEEE, 2012.

[168] Zeratul Izzah Mohd Yusoh and Maolin Tang. Composite saas scaling in cloud computing using a hybrid genetic algorithm. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pages 1609–1616. IEEE, 2014.

[169] Guangquan Zhang, Jie Lu, and Ya Gao. *Multi-Level Decision Making*. Springer, 2015.

[170] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[171] Dongfang Zhao, Nagapramod Mandagere, Gabriel Alatorre, Mohamed Mohamed, and Heiko Ludwig. Toward locality-aware scheduling for containerized cloud services. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 263–270. IEEE, 2015.

[172] Chao Zheng, Ben Tovar, and Douglas Thain. Deploying high throughput scientific workflows on container schedulers with makeflow and mesos. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 130–139. IEEE Press, 2017.

[173] Charles Zheng and Douglas Thain. Integrating containers into workflows: A case study using makeflow, work queue, and docker. In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*, pages 31–38. ACM, 2015.

[174] Yujian Zhu, Junming Ma, Bo An, and Donggang Cao. Monitoring and billing of a lightweight cloud system based on linux container. In *37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 325–329. IEEE, 2017.

[175] James R Ziegler. Time-sharing data processing systems. 1967.

[176] Guy Zyskind, Oz Nathan, et al. Decentralizing privacy: Using blockchain to protect personal data. In *Security and Privacy Workshops (SPW), 2015 IEEE*, pages 180–184. IEEE, 2015.