FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Elastic Business Process Management in the Cloud

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Philipp Hoenisch BSc.
Matrikelnummer 0725710

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Schahram Dustdar
Zweitbetreuung: Ass. Prof. Dr.-Ing. Stefan Schulte

Diese Dissertation haben begutachtet:

Schahram Dustdar	Ralf Steinmetz

Wien, 24. August 2015

Philipp Hoenisch

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Elastic Business Process Management in the Cloud

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Philipp Hoenisch BSc.
Registration Number 0725710

to the Faculty of Informatics
at the TU Wien

Advisor: Univ. Prof. Dr. Schahram Dustdar
Second advisor: Ass. Prof. Dr.-Ing. Stefan Schulte

The dissertation has been reviewed by:

Schahram Dustdar	Ralf Steinmetz

Vienna, 24th August, 2015

Philipp Hoenisch

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Philipp Hoenisch BSc.
Brunner Gasse 56-58
2380 Perchtoldsdorf
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.

Wien, 24. August 2015

Philipp Hoenisch

Acknowledgements

About 4 years ago I would have never though that I would ever start or even finish a PhD. However, after roughly 3 years, this thesis finally marks the finish line for one of the highest academic degrees one can earn. It is obvious that I can not claim all credits for myself as there are a couple of people who strongly supported me whenever I was either stuck or just slacking of.

First, I want to thank Prof. Schahram Dustdar who welcomed me in his group and provided a productive and pleasant work environment over the last few years. I am also thankful for Prof. Ralf Steinmetz for serving as the second reviewer of this thesis. Next, I want to thank all my colleagues from the Distributed System Group for all the collaborations, discussions and support. A special thank goes to Christoph Hochreiner, Black Coffee and Lorem Ipsum for their insightful comments and encouragement, but also for the hard questions which incented me to widen my research from various perspectives.

Another special thanks goes to all the researchers with whom I enjoyed working. Most notably, Dieter Schuller for his invaluable personal and technical experiences. Christian Janiesch and Srikumar Venugopal for their academic support in several publications. And Ingo Weber who hosted me at NICTA in Sydney as a short term research intern. I hope our paths will cross again in the near future.

My special thank goes to Stefan Schulte. I would like to express my deepest gratitude to him as he supported me from the very first moment on. He greatly guided me through the dissertation and has always been on hand with help and advice for me.

Last but not least, a special word of thanks goes to my family and my dearest friends for their continuous support, encouragement and especially the needed distractions after work. Without you, I would not be where I am right now.

The research in this dissertation has received funding from the European Community's Seventh Framework Programme for research, technological development and demonstration (FP7-ICT) under the grant agreement number 318201 (SIMPLI-CITY).

Danksagung

Diese Arbeit wurde durch das “European Community’s Seventh Framework Programme” für Forschung, Technologischer Entwicklung und Demonstration (FP7-ICT)” im Rahmen des *SIMPLI-CITY*-Projektes (grant agreement no. 318201) finanziert.

Abstract

Business Process Management is a multifaceted approach covering several aspects of organizational, management and technical facets of business processes. Recently, this technology gained great attention in the field of many different industries including the finance industry or the energy domain where computational resources are used to carry out business processes automatically. Business processes are composed from human or software-based services. Especially in larger companies, an extensive number of different processes are available, each made up from various process steps which are realized by software services, each needing a different amount of resources. As the amount of process instances to be executed varies over time, a *Business Process Management System* is needed which is able to serve the ever-changing demand of needed resources. For example, the amount of required resources during peak times will be much higher than during off-peak times. Permanent provisioning of resources is obviously not the best choice, as resources which are able to handle peak loads will hardly be used during off-peak times. Hence, this leads unwanted high cost (i.e., *over-provisioning*). Contrary, providing less resources may lead to an *under-provisioned* system, which might not be able to carry out all processes during peak times or will suffer from a low Quality of Service. With the upcoming of cloud computing, it is possible (i) to lease and release resources in an on-demand, utility-like fashion, and to provide the means of (ii) rapid elasticity through scaling the infrastructure up or down, based on (iii) pay-per-use through metered service. However, so far, only few researchers have provided methods and solutions to facilitate *Elastic Processes*, i.e., processes which are carried out on cloud-based resources while considering all three dimensions of elasticity: *resource*, *quality* and *cost*.

Hence, this thesis presents novel approaches to apply cloud-based computational resources for Business Process Management. First, we show how leased resources can be used more efficiently if the process' future resource demand is considered during scheduling and resource allocation. Second, by constantly extending this approach, we present a cost-based optimization model for sequential business processes. As in real-world complex business processes are more realistically, we present afterwards a multi-objective optimization model for complex business processes. Finally, we introduce an extension to this model, allowing to lease resources from a public and private cloud equally, leading to a hybrid cloud environment. We evaluate our approaches extensively and compare the results against state of the art baselines in order to show its potential of reducing process execution time and avoiding unnecessary cost while still ensuring Service Level Agreements.

Kurzfassung

Geschäftsprozess-Management ist ein vielschichtiger Ansatz, der sich mit der Gestaltung, Implementierung und Steuerung von Geschäftsprozessen beschäftigt. Zuletzt erlangte diese Technologie große Aufmerksamkeit in vielen verschiedenen Branchen, wie insbesondere in der Finanz- und Energiebranche, in der Rechenressourcen für die Automatisierung von Geschäftsprozessen verwendet werden. Geschäftsprozesse bestehen in der Regel aus Mensch- oder Software-basierten Diensten. Vor allem in großen Unternehmen ist es üblich, dass eine Vielzahl verschiedener Geschäftsprozesse zur Verfügung steht, die zum Teil aus rechenintensiven Software-basierten Diensten bestehen. Da die Menge der Prozessinstanzen, die ausgeführt werden sollen, variiert, wird ein Geschäftsprozess-Management-Programm benötigt, das in der Lage ist, die ständig wechselnden Anfragen zu beantworten und deren Ausführung zu steuern. Es ist offensichtlich, dass die Anzahl der benötigten Ressourcen zu Spitzenzeiten höher ist als zu weniger intensiv genutzten Zeiten. Aus diesem Grund ist eine permanente Bereitstellung von Rechenressourcen nicht die beste Wahl, denn Ressourcen, die in der Lage sind Spitzenlasten zu tragen, werden außerhalb dieser Zeiten kaum benützt. Dies führt zu unerwünscht hohen Kosten. Andererseits kann es, wenn zu wenige Ressourcen zur Verfügung stehen, zu einer niedrigen Dienstgüterqualität kommen. Mit dem Aufkommen von Cloud Computing, ist es nun möglich: (i) Rechenressourcen nur bei Bedarf zu mieten, und (ii) die Infrastruktur schnell nach oben oder unten zu skalieren, während nur die gemieteten Ressourcen bezahlt werden müssen. Bisher haben jedoch nur wenige Wissenschaftler an der Realisierung von Elastischen Prozessen - Prozessen, die auf Cloud-basierten Ressourcen ausgeführt werden - gearbeitet. Bei diesen sind drei Dimensionen der Elastizität zu berücksichtigen: Ressource, Qualität und Kosten-Elastizität.

In dieser Arbeit werden neue Ansätze präsentiert, um Geschäftsprozesse auf Cloud-basierten Ressourcen auszuführen: Der erste Teil der Arbeit zeigt, wie die geleasten Ressourcen effizient genutzt werden können. Anschließend wird ein kostenbasiertes Optimierungsmodell für sequentielle Geschäftsprozesse dargestellt. Da jedoch in der realen Welt komplexe Geschäftsprozesse realistischer sind, soll dieser Ansatz erweitert werden und ein “Multi-Objective” Optimierungsmodel für komplexe Geschäftsprozesse erstellt werden. Anschließend wird gezeigt, wie dieses Optimierungsmodel erweitert werden kann, um Geschäftsprozesse in einer Hybrid Cloud auszuführen. Diese Ansätze werden intensiv evaluiert und die Ergebnisse mit Baselines verglichen, die dem aktuellen Stand der Technik entsprechen. In der vorliegenden Arbeit wird sich zeigen, dass mit diesem Ansatz Ausführungszeit und Kosten verringert werden können.

Contents

List of Figures

List of Tables

List of Algorithms

Earlier Publications

This thesis is based on work published in scientific conferences, workshops, journals and surveys. These papers found the core of this thesis and are listed here only once and not explicitly referenced throughout this thesis. Parts of these papers are contained in verbatim. Please refer to Appendix C for a full publication list of the author of this thesis.

- **Philipp Hoenisch**, Stefan Schulte, and Schahram Dustdar. Workflow Scheduling and Resource Allocation for Cloud-based Execution of Elastic Processes. In 6*th IEEE International Conference on Service Oriented Computing and Applications (SOCA)*, 1-8. IEEE, 2013

- Stefan Schulte, Dieter Schuller, **Philipp Hoenisch**, Ulrich Lampe, Schahram Dustdar, and Ralf Steinmetz. Cost-Driven Optimization of Cloud Resource Allocation for Elastic Processes. *International Journal of Cloud Computing (IJCC)*, 1(2):1-14, 2013

- **Philipp Hoenisch**, Dieter Schuller, Stefan Schulte, Christoph Hochreiner, and Schahram Dustdar. Optimization of Complex Elastic Processes (accepted for publication). *IEEE Transactions on Services Computing (TSC)*, Volume NN, Number NN, NN-NN, 2015

- **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 (CLOUD)*, pages 17-24. IEEE, 2015

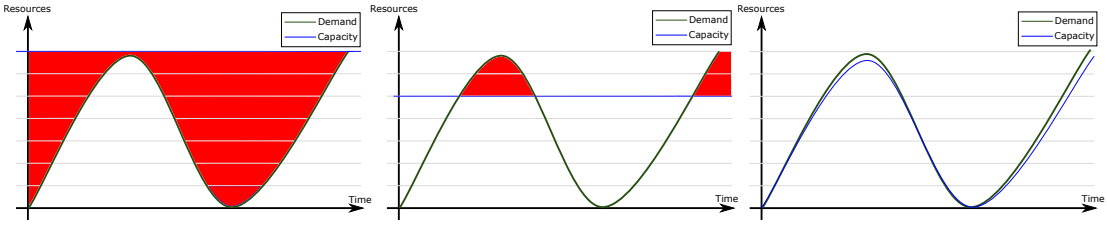
- Stefan Schulte, Christian Janiesch, Srikumar Venugopal, Ingo Weber, and **Philipp Hoenisch**. Elastic Business Process Management: State of the Art and Open Challenges for BPM in the Cloud. *Future Generation Computer Systems*, Volume 46, 36-50, 2015

CHAPTER 1

Introduction

Business Process Management (BPM) is a field in operations management covering different aspects of organizational, management and techniques of business processes. According to van der Aalst et al., BPM "includes methods, techniques, and tools to support the design, enactment, management, and analysis of operational business processes" [110]. Business processes are generally composed of different services, involving human-provided as well as software-based services, in order to present and realize a particular business logic or product [92]. In several different industrial sectors, BPM has gained more attention in the last few years. For example, in the financial industry where trade settlement and execution control are meant to be carried out automatically [40], as well as in the energy domain where data has to be collected from a very large number of sensors and processed in almost real-time [91, 93].

One of the most important aspects of BPM is the automatic execution of processes using computational resources which are referred as workflows[74]. Notably, within the remainder of this work, we stick to the term *business processes* before getting executed and *elastic processes* or *processes* when executed using cloud-based computational resources. Doing so, efficient and effective resource allocation is a key issue in elastic process executions and a Business Process Management System (BPMS) has to be able to administrate, schedule and execute several hundreds or thousands of process instances simultaneously. In order to do so, a BPMS has to assign computational resources to each task involved in the processes. Due to the nature of software-based services and their functional diversity, the demand of computational resources varies. Especially if a large amount of data has to be processed in a short time under a specified Quality of Service (QoS), service executions may be very resources and time-intensive. Further, due to different priority levels, some processes need to be carried out immediately, while others may run regularly and others may be postponed to a certain point in the future adhering to a specific deadline. Naturally, the computational resources needed for process executions are subject to temporal variations, i.e., the amount of needed computational resources during peak time, may be much higher than during off-peak times [32].



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

Figure 1.1: Fixed Resources vs. Cloud Provisioning

1.1 Problem Statement

A common service provider problem is that the traditional permanent provisioning of fixed computing capacities (e.g., traditional data centers) may reach its limits when cost- and quality-efficiency should be achieved [5]. As it can be seen in Figure 1.1, the provisioning of fixed resources that are able to handle peak loads may hardly be used during off-peak times, which may lead to very high cost (*over-provisioning*, shown in red in Figure 1.1a). Contrary to that, during peak times, some processes may not be carried out at all, or with an unacceptable level of QoS, if there is a shortage of resources (*under-provisioning*, shown in red Figure 1.1b).

With the emergence of cloud computing, it is nowadays possible to use computational resources in an on-demand, utility-like fashion, while the risk of over- or under-provisioning can be prevented [18] (Figure 1.1c). The virtually unlimited amount of leasable cloud-based computational resources is especially promising for business process enactments as such resources can be acquired on-demand in a self-service way. Further, the underlying cloud infrastructure provides the possibility to scale up and down by adding or removing resources (*rapid elasticity*) [79]. Executing business processes using cloud resources can be called *elastic processes* [32, 79], i.e., reflecting the three major benefits of cloud computing: (i) *Rapid elasticity* of single processes as well as the process landscape based on the actual demand for computational resources, (ii) leasing and releasing the needed computational resources in an *on-demand*, *utility-like* fashion, and (iii) pay-per-use of the resources through *metered service*.

With these new possibilities, also new challenges and responsibilities came up for BPMSs [98]. Scheduling process instances among cloud-based resources, which can be leased and released in a dynamic way, is a much more complex task than creating a scheduling plan on a fixed limited amount. As it is possible that process instances may be requested ad hoc, i.e., after the scheduling plan was created, the BPMS needs to be able to react in time to the ever-changing demand for resources. Creating such a scheduling plan needs to take into account the individual Service Level Agreements (SLAs) of processes, such as deadlines, execution times, or other QoS attributes, as well as the current process landscape, available resources, and the amount of queued process instances, i.e., those which are meant to be executed in the near future. Beside of creating such a scheduling plan, the BPMS is also responsible that the schedule is carried out and

resources are allocated accordingly. By the nature of such systems, one cannot rule out the possibility that unexpected errors occurs. Therefore, the BPMS needs to monitor the actual execution in order to conduct countermeasures in the case of service or resource failures.

To the best of our knowledge, in literature and practice, surprisingly little effort has been investigated in the field of business process execution using cloud resources in order to realize *Elastic Processes* [4, 32]. Current approaches mostly consider only single service [18, 63, 71] for scaling resources. Only a few researchers have already addressed the topic of business processes or scientific workflows in the cloud [53]. In addition, the work so far on business processes mainly considers a single process instances a time neglecting the possibility to *share* services instances and resources among others [9, 52, 117]. Hence, new techniques for running business processes in the cloud in order to realize elastic processes are needed.

1.2 Research Question

The research conducted throughout this thesis has been motivated by the challenges identified in Section 1.1, i.e., the functionalities a BPMS needs to provide when business processes are executed on cloud-based computational resources. Hence, in this work, we address the following research questions:

Research Question I:
How can cloud-based resources be used for business process management in a cost-efficient way?

As discussed in Section 1.1, cloud-based computational resources can be used to address the problem of under- and over-provisioning when executing business processes. However, to make use of cloud-based resources, BPM becomes more complex than for fixed-provisioned resources. First, a BPMS in the cloud has to be able to lease and release cloud resources on-demand and deploy the required services among it. Second, the BPMS has to create a scheduling plan for the process executions, i.e., a detailed plan defining when a particular process instance needs to be carried out. For that, the scheduling plan needs to take into account the process' deadline and other QoS attributes. Third, the BPMS has to ensure that the scheduling plan is carried out and allocate required resources to single services and service instances accordingly. Last but not least, the BPMS needs to conduct countermeasures in the case of service or resources failures.

Existing work mostly proposed scheduling solutions where full resources were allocated for a single process instance. However, as it is very likely that several different instances of a particular business process may be executed simultaneously, sharing services and resources among them will eventually lead to less cost. Hence, a BPMS needs to consider the full process landscape at once during scheduling and executions. This means, the BPMS should be able to handle several process instances of the same, or different time, allocate resources to services and create a detailed scheduling plan about *where* and *what* service needs to be invoked *at what time* in order to fulfill a particular process instance.

Research Question II:
How can leasing cloud-based resources be optimized for enacting complex business processes?

Since a different amount of business processes may be requested at any time, the demand of computational resources is under a continuous change. Hence, utilizing cloud-based computational resources for business process enactments may lead to cost savings. However, the more complex business processes get, the more complex is the leasing and scheduling process. Complex business processes comprise different process patterns, including (but not limited to) AND-block, XOR-block or Repeat loops. These have to be handled specifically during scheduling and resource planning as the process' future is not deterministic. Hence, new techniques for utilizing cloud-based computational resources for complex business process enactments are needed.

Research Question III:
What problems need to be considered when executing complex business processes in a hybrid cloud environment?

Nowadays a wide range of different cloud providers are available on the market (Amazon Web Services (AWS)[1], Windows Azure[2], Google Cloud Computing[3]). Each provider offers different services with different SLAs of a different price. Beside of having the possibility to have even more resources available by combining different cloud providers, depending on the current need, it may be useful to lease resources simultaneously from various cloud providers leading to a hybrid cloud environment. Enacting business processes in a hybrid cloud will lead to new challenges as different cost types and pricing models as well as privacy aspects have to be considered.

1.3 Scientific Contribution

Guided by the research questions stated in Section 1.3, throughout this thesis we made the following scientific contributions to the state of the art of BPM:

Contribution I:
A reasoning and scheduling algorithm for sequential elastic processes.

When enacting business processes in the cloud, a BPMS has to account for scheduling and resource allocation equally. For doing this efficiently, the BPMS has to *look into the future*, i.e., plan ahead. We present first a reasoning and scheduling algorithm for sequential business processes which takes into account the near future. Hence, during the scheduling, the BPMS is able to pre-pone future process steps and use resources more efficiently. This algorithm will be implemented in a prototype. For that, we reuse

[1] `https://aws.amazon.com`
[2] `https://azure.microsoft.com`
[3] `https://cloud.google.com`

Vienna Platform for Elastic Processes (ViePEP) from [97] and extend the framework. The extension will be evaluated running different scenarios and comparing it against a state of the art baseline. Details are presented in Chapter 4. Contribution I was originally presented in [44].

Contribution II:
A cost-based optimization model and heuristic for sequential elastic processes.

The result of the reasoning and scheduling algorithm from Contribution I may or may not lead to an optimal and cost-efficient solution. Hence, we introduce a novel optimization model for sequential elastic processes which aims at cost-efficiency. Again, we extend ViePEP by adding additional functionalities and components. Based on the cost-based optimization model we implement a heuristic and run several evaluations which are compared against a baseline. Details are presented in Chapter 5. Contribution II was originally presented in [99].

Contribution III:
A cost-based optimization model for complex elastic processes.

We continue with the approach from Contribution III, i.e., the mutli-objective optimization model aimed at complex elastic processes. Complex elastic processes differ to sequential business processes as several additional factors need to be considered. For example, the decision of which branch a process flow chooses or the amount of loop repetitions will have an impact on the scheduling strategy. Hence, we provide a multi-objective optimization model which considers different SLAs simultaneously. Solving this problem is defined as the *Service Instance Placement Problem (SIPP)*. In order to evaluate the applicability of this optimization model, we extend ViePEP and run comprehensive experiments. Details are presented in Chapter 6. Contribution III was originally presented in [43].

Contribution IV:
A cost-based optimization model for complex business elastic in a hybrid cloud focusing on data transfer cost.

After having a multi-objective optimization model proposed in Contribution III, we consider now business process enactment in a hybrid cloud environment. By combining a public cloud and a private cloud, even more resources are available for executing business processes. However, it is common that cloud providers charge users for in-bound and out-bound traffic, i.e., uploading data to and downloading data from the cloud. We consider this aspect in an extension of the SIPP model and provide a data-aware and cost-based optimization model for complex elastic processes in hybrid clouds. Comprehensive experiments have been performed in order to verify the efficiency of the extension. Details of this optimization model are presented in Chapter 7. Contribution IV was originally presented in [42].

1.4 Organization of this Thesis

The remainder of this thesis is structured as follows:

- Chapter 2 provides background information and introduces the basics which are used throughout this thesis. Specifically, background information about BPM is introduced in Section 2.1. Further, we give a short introduction on cloud computing in Section 2.2 and the concept of elastic processes in Section 2.3. In addition, we introduce the original ViePEP 1.0 which is used as a base of this thesis and which was originally presented in [97].

- In Chapter 3 we present related work from the field of resource allocation for single service execution (Section 3.1), resource allocation for scientific workflows (Section 3.2) and the most related field: resource allocation for business processes (Section 3.3).

- In Chapter 4 we present a basic process scheduling and resource allocation approach for cloud-based executions of elastic processes. Within this chapters, we extend ViePEP, re-introduce its architecture and present a process scheduling and resource allocation algorithm.

- Afterwards, Chapter 5 extends this approach and presents a cost-based optimization model for sequential elastic processes. We realize this optimization model as an heuristic and evaluate it against a state of the art baseline.

- In Chapter 6 we address resource optimization and scheduling of complex elastic processes. For that, we extend ViePEP and map the complex search space for process scheduling and resource optimization as a multi-objective optimization problem.

- Chapter 7 extends the multi-objective optimization problem in order to support a multi-cloud environment. For that, we address the challenges of data-aware process scheduling.

- Finally, we conclude this thesis in Chapter 8 and discuss the presented contributions. Last but not least, we offer an outlook on our ongoing and future research.

CHAPTER 2

Background

In this chapter, we introduce background knowledge needed to understand this thesis, i.e., the needed basic concepts that are used in this thesis. First, we introduce the basics of BPM. Second, we illustrate the fundamental properties and functionalities of cloud computing. Afterwards, we present ViePEP as it was firstly introduced in [97]. The prototype ViePEP served as the basis for this thesis and each contribution (as presented in Section 1.3) has been implemented as an extension.

2.1 Business Process Management

A basic use of Web services is composing them into value-added structures. This approach is referred as service compositions (or sometimes orchestration) and is the underlying technology of business processes [49]. The multidisciplinary approach, covering the organizational, management and technical aspects of service compositions is referred BPM [49, 70]. BPM "includes methods, techniques, and tools to support the design enactment, management and analysis of operational business processes" [110]. Processes are generally defined as sequences of tasks performed within (or across) companies or organizations [119]. In this context, BPM refers to a collection of tools and methods for achieving an understanding of, managing, and improving an enterprises' process portfolio [126].

Within such a portfolio, processes are commonly classified into two categories:

1. value-adding core processes, which are considered to contain corporate expertise and produce products or services that are delivered to customers [26],
2. and non-value-adding supplementary processes, which facilitate the ongoing operation of the core processes.

For example, while processes concerning design and production are usually seen as core processes, human-provided processes are seen as supplementary processes.

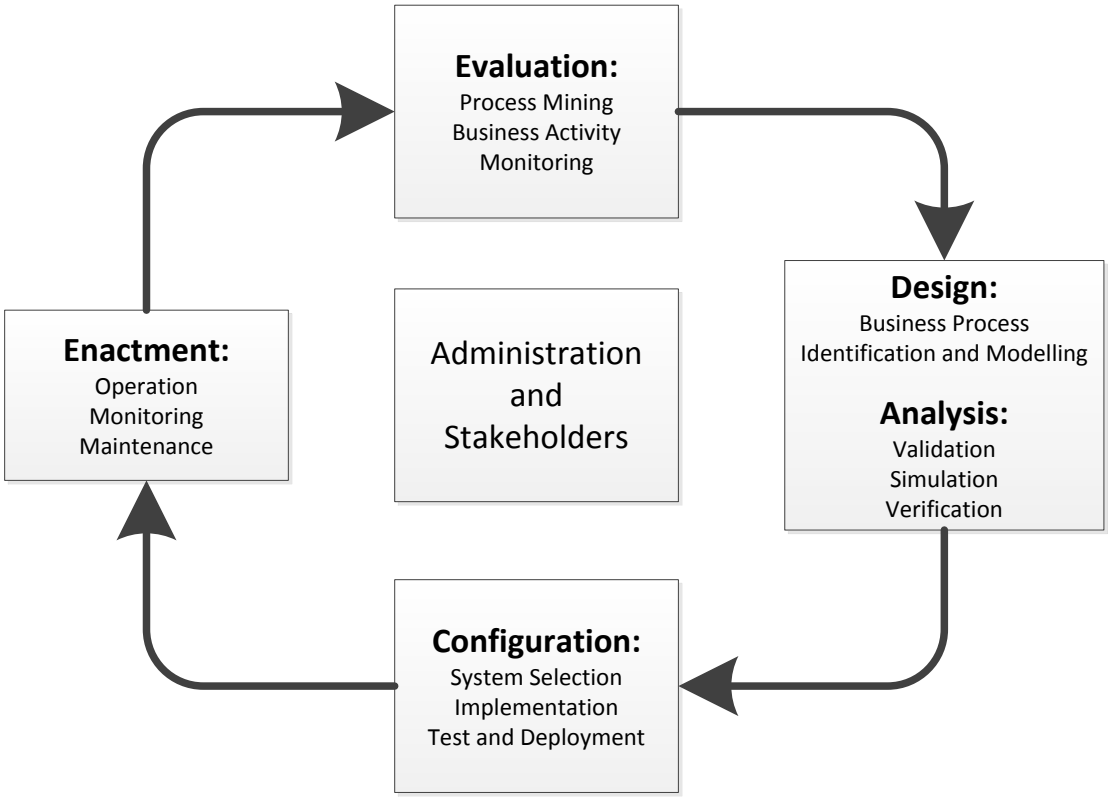


Figure 2.1: Business Process Life-Cycle (adapted from [118])

In contrast to business process engineering (which is a greenfield methodology), BPM considers generally planning, controlling, and monitoring of intra- and/or inter-organizational processes. Further, this is done with regards to existing operational sequences and structures in a consistent, continuous, and iterative way of process improvement [8].

Figure 2.1 presents a general BPM life-cycle which usually consists of four main phases: Design & Analysis, Configuration, Enactment, and Evaluation [118]. Notably, within this thesis only the Enactment phase is of further interest.

A system which provides the technical means to support the realization of BPM is referred as a BPMS. In general, such a BPMS allows to store process definitions (i.e., process models), manages the process instances and monitors the enactments. This can be achieved through logging the process executions. Process models hereby specify process orchestrations and activities along them with execution constraints and relations between them. A process model can be seen as a formal description of a business process assigning single atomic work units (or *tasks*) to agents (or services) for execution. Such a work unit (or task) refers to a work which needs to be performed, either manually or automatically by system. Following the principle of Service Oriented Computing (SOC), applications are used to fulfill the automated tasks. These are commonly implemented

as (software) services [88, 110].

The automation of such a sequence of tasks within a process by a system is often referred as a *workflow* [74]. Before executing a process, its process model needs to be instantiated which is commonly called a process *instance*. Further, the system executing and controlling these processes is called BPMS. Process requesters and service providers may define QoS constraints in form of SLAs. This can be done either for the full process model, for a specific process instance, or just for single tasks within a process instance. A SLA consists out of one to many Service Level Objectives (SLOs) which address execution deadlines (i.e., a specific point of time or a maximum amount of time for the process or task execution), cost or other QoS metrics [55, 78].

It is a common approach to have a large number of concurrent process instances, which is referred to a process *landscape*. Such a process landscape may span several different organizations or companies [15]. In order to allow such a collaboration, the employed BPMSs executing the process orchestrations need to communicate with each other by sending and receiving messages over a pre-defined channel [118].

Due to the nature of such a process landscape, the BPMS needs to deal with different arrival scenarios at any time. Further, the management efforts of scheduling and enactment get more complicated with the degree of process model complexity, these may vary from simple sequential processes (Chapter 4 and Chapter 5) to complex process model (Chapter 6 and Chapter 7).

2.2 Cloud Computing

Cloud computing is a relatively new paradigm for delivering computational resources in a an on-demand fashion to customers similar to other utilities like water, electricity or gas [5, 6, 14, 16, 18, 35, 90, 111]. Examples of such resources are networks, servers, storage, software (applications) and other services. In addition, with the market entrance of major players like Amazon or Google, this new computing paradigm is already pretty advanced [77] and the worldwide public cloud services market is worth 131 Billion US dollars, with an estimated compound annual growth rate of about 17% from 2011 to 2017 [38].

Compared to traditional data centers, where resources are provided in a static way (see Figure 1.1), in cloud computing the foundational characteristics are: *on-demand self-service*, i.e., consumers can lease computing resources (server time, network storage, etc.) from cloud providers without human interaction and prior commitments. These resources are available over the internet and can be accessed through heterogeneous client platforms (*broad network access*). A main functionality is the so-called *resource pooling* which allows to serve multiple consumers simultaneously by assigning virtual resources dynamically according to their demand. Further, these resources can be elastically provisioned and released, either automatically or manually. Hence, *rapid elasticity* is a key feature to the consumer [79].

The above mentioned properties enable the realization of elastic processes which can dynamically adjust to the demand of resources which are needed to fulfill a service or

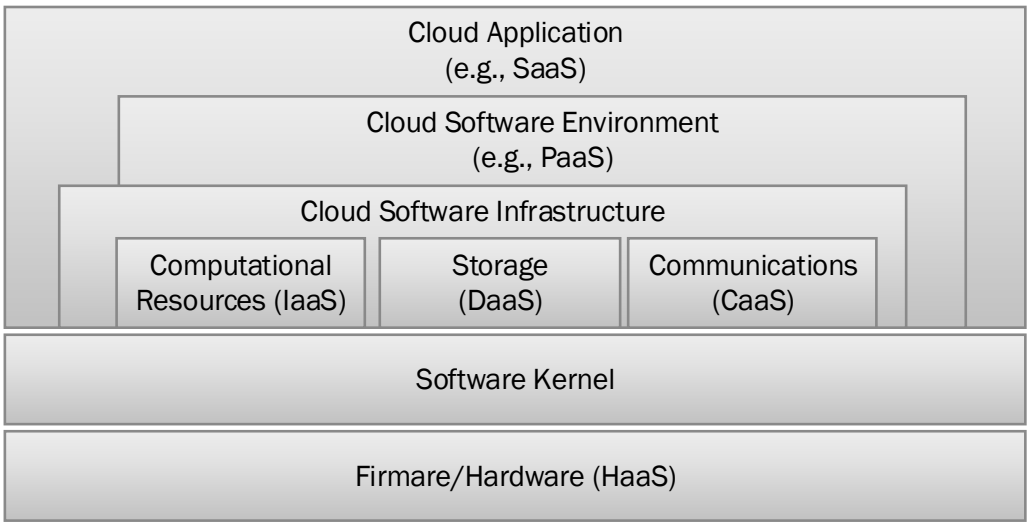


Figure 2.2: Layers of the Cloud Computing Stack (adapted from [122])

business task. This utility driven nature is based on the Service Oriented Architecture (SOA) principle, hence, offered service can be consumed in a service-oriented way and without major user interaction [18]. These services are offered by *cloud providers* who are "responsible for making a service available to interested parties". Interested parties are referred as *cloud consumers*, i.e., "a person or organization that maintains a business relationship with, and uses service from, cloud providers" [14].

In recent years, a well known architecture of the cloud computing technology stack has emerged. This technology stack involves three core layers: Infrastructure as a Service (IaaS),Platform as a Service (PaaS) and Software as a Service (SaaS) [79, 122] (as shown in Figure 2.2). However, there are no limits set to add additional layers, e.g., the newly upcoming principle of Linux Container (LxC) technologies, such as Docker Containers, gain more and more attentions [46].

At the bottom of the stack, one can find Hardware as a Service (HaaS). It provides the means of leasing real hardware. However, this layer is seldom provided to cloud consumers and rather known from the traditional way of server hosting, i.e., physical machines. Mostly referred to as computational resources, the IaaS layer is often further sub divided into additional services, e.g., communication or storage. Further, the IaaS layer provides the means of dynamic provisioning of computational resources such as Virtual Machines (VMs), storage discs, network devices, etc. Operating directly on the IaaS level provides a higher degree of flexibility to cloud consumers but also requires to to manage the leased resources on their own. Hence, this requires them to have the necessary expertise in server management. Well known examples for IaaS providers are

Amazon's Elastic Compute Cloud (EC2)[1], Microsoft Azure Compute[2], Google's Cloud Platform[3], VMWare vCloud[4] and DigitalOcean[5]. Virtualization technologies allow to run several VMs on a single physical machine. Each VM is isolated in terms of its applications and resources. Beside of VMs, several IaaS provider offer additional services at the same level, e.g., services for storing and retrieving data, i.e., Data as a Service (DaaS) (e.g., Amazon's Simple Storage Service (S3)[6]).

On top of the IaaS layer is the PaaS layer. It is meant to reduce the administrative overhead of managing the virtualized infrastructure and provides different products to consumers. Commonly these products offer pre-configured software environments to the user providing the means to simplify deploying, managing and scaling applications, while provisioning of physical hardware, power, data center are managed by the cloud provider [65]. However, when using PaaS products, consumers sacrifice some flexibility compared to operating directly on the IaaS layer by relying on the provider managed infrastructure. A variety of different PaaS products are offered by cloud providers. The most common ones assist developers by offering a set of pre-configured components which can easily be composed and then deployed into separate containers to ease development setup. More advanced are managed VM containers offering a full Operating System (OS) including a software stack for a certain use. However, developers still have a certain freedom to customize and modify this environment (e.g., Heroku[7]). Most general, PaaS products are meant to increase software developers productivity and reduce software cost [65]. In addition, providers can leverage additional information about the deployed software applications which would not be available on the IaaS level.

As Figure 2.2 depicts, the most upper layer is the SaaS layer providing cloud applications. While its offers are rather targeted at end users, IaaS and PaaS products are more for application developers and system operators. Usually, SaaS are meant to meet a certain business need in a specific domain aiming at human interaction, i.e., a web interface or a native graphical user interface is provided. Well known examples for SaaS products are Dropbox[8], Office 360[9], Spotify[10], Netflix[11], and others.

The cloud paradigm offers new possibilities for business process enactment as resources can be provisioned dynamically in an on-demand fashion, i.e., leased only when really needed and released when unneeded. However, new possibilities also bring new challenges: most importantly, a BPMS for elastic processes, which needs to be able to lease and release cloud-based computational resources. Afterwards, the required services need to be deployed among the available resources in an efficient way. Second, the BPMS has to

[1] https://aws.amazon.com
[2] https://azure.microsoft.com
[3] https://cloud.google.com
[4] https://vcloud.vmware.com/
[5] https://www.digitalocean.com/
[6] https://aws.amazon.com/s3
[7] https://www.heroku.com/
[8] https://dropbox.com
[9] https://login.microsoftonline.com
[10] https://spotify.com
[11] https://netflix.com

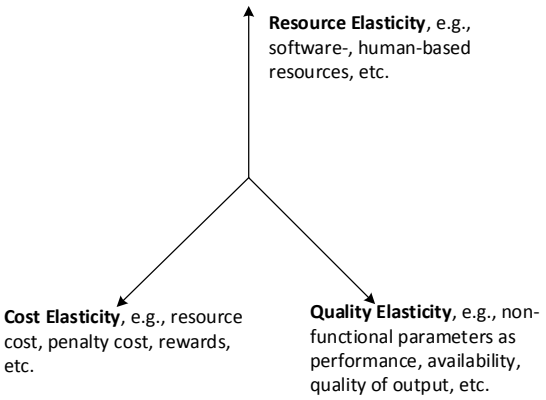


Figure 2.3: Three Dimensions of Elasticity

create a scheduling and resource allocation plan, i.e., a detailed plan telling at what point of time what amount of resources are required including where a specific process instance should be executed. Third, the BPMS needs to monitor the process executions in order to be able to detect SLA violations in time and arrange corresponding countermeasures [68].

2.3 Elastic Processes

With the upcoming of cloud computing [79], advanced resource allocation and virtualization capabilities are now possible which have fostered a new trend towards elastic computing [32, 67]. The term *elasticity* is well-understood in economics (e.g., price elasticity [102]) as well as in other fields like physics or chemistry (e.g., material science [7, 103]). In general terms, elasticity expresses the potential to change some variables in response to the change of other variables. For example, in the computing domain, an elastic system is a system which is able to dynamically adjust itself (or adjust key properties) in response to external events, i.e., user requests. These elastic properties can be generally divided into three categories and be seen as the the ability to change the amount of used resources (*resource elasticity*), the system responsiveness in terms of cost (*cost elasticity*) and the resulting QoS (*quality elasticity*)[32], see Figure 2.3.

- *Resource Elasticity:*
  It is common today to understand elasticity purely from a resource-management point of view[24, 32]. However, this is rather restrictive, as resources' requirements are not determined only by the application and services using them. Even more, considering elasticity solely from the resource perspective may lead to limitations in terms of scalability and quality. A general assumption is that the more resources are added to an application, the higher the quality gets, however, this is only true to a certain point as the resource-quality relation can hardly be presented as a linear function. For example, an application for performing image optimization

operations will need a certain amount of computational resources, but, if more resources are added, it is not likely that the quality of the image may increase, but the result will be available faster[32].

- *Quality Elasticity:*
  Quality elasticity measures how responsive the quality is when the amount of available resources changes[32]. This elasticity follows the simple assumption that the achievable QoS increases the more resources get consumed. However, the main issue here is that a service is needed which is able to measure the quality. Further, this service needs to consider a cost function for computing the resource requirements in order to achieve a certain level of QoS, such as execution speed. In this case, the service's result is deterministic, but the execution speed can be scaled, based on the amount of available resources. An example for such a service is MapReduce, which is a scalable programming framework that lets users process data elastically [27]. MapReduce has a desired elasticity that allows to scale the execution speed with the amount of available computational resources.

  Execution time (or response time which includes the network latency) is not the only quality a service may have. Other quality measurements account for the result quality of a service's computation process. For example, the Aqua approximate query answering system which was developed by Bell Labs is able to make trade-offs considering quality aspects in query processing [2]. Traditionally, query processing aims at generating correct and exact answers, however, the Aqua approximate query answering system is able to execute queries faster than traditional answering systems by providing only approximate answers instead of a correct one which may take an unacceptable long time. The trade-off between the QoS outcome and the execution time (response time) needs to be considered when allocating resources.

- *Cost Elasticity:*
  Cost elasticity refers to the responsiveness of cost to changes of resource provisions. This means, service providers may apply cost elasticity when defining the pricing model for offered resources. Within this elasticity resources (such as computational services provided by VMs, data transmission, etc. are charged based on a pay-as-you-go model [79]. By defining such a model, service providers (such as cloud providers) need to consider different cost items: e.g., the initial investment, provisioning and maintenance of the hardware and network infrastructure. Based on this information, providers can create a dynamic pricing model which is based on the cost elasticity concept. For example, in Amazon's EC2 pricing model, customers are offered with two different pricing models[12]: *On-demand instances* which are purely based on a pay-per-use model. Customers do not have long-term commitments and are free from planning. The applied cost need to be paid per hour. And *spot instances*, for which a different prices needs to be paid which may fluctuate over time according to the actual supply-demand status (notably, Amazon may also consider additional

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

factors for their spot prices, but these are not open for the public). In this pricing mode, potential customers bid a maximum they are willing to pay for resources. The resources are provided as long as the spot price is smaller or equally to the bidding prices or until the resource is explicitly released (e.g., the VM instance gets terminated).

Using this pricing schema, Amazon can offer higher prices during peak times and lower prices during off-peak time in order to shape customers behaviors. This may lead to a balanced resource usage during the day, as more flexible customers would tend to consume more resources when the prices are cheaper (off-peak times) and avoid to lease resources during peak times.

2.3.1 Downside of Elastic Processes

If not properly designed, elasticity may result in unwanted system behavior or unexpected cost[59]. Allowing an elastic system to change its state elastically can cause slight modifications which in the worst case may become uncontrollable or irreversible. An elastic system mainly relies on efficient monitoring techniques and has to decide whether the available information is valid and in what way it can be useful. For example, if invalid information is provided, e.g., the system monitor returns wrong information, wrong transformation happened, etc. the elastic system may acquire resources in an uncontrolled manner, i.e., leasing too many resources which will eventually lead to uncontrolled high cost.

Even more, it may happen that the elastic system "decides" that a better placement of services needs to be found, and constantly moves services between the leased resources. This could prevent a needed scale down, i.e., releasing unneeded resources, which again, will lead to unwanted spending. In contrast to that, if the system fails to scale up, i.e., additional resources are not leased on time, the needed level of QoS can not be achieved which may result in SLA violations [50].

However, as mentioned in [32], elastic processes also consider process dynamics from the field of human computing. Human computing can be seen as human-centric *technologies* and *services* (e.g.,social compute units) [31, 87]. Although human computing is still in its early states, promising approaches have already been proposed in different fields such as context sensing machine analysis of human affective and social behavior and smart environments [23, 86, 113]. However, scaling human resources is a challenge on its own as QoS for human-provided services needs to consider different aspects. For example, a possible way to consider quality for resource allocation of human services would be by considering the people's heterogeneous skills and interests.

Considering these different attributes can lead to better results that meet a predefined level of quality [94]. In addition to that, a single human resource, e.g., a single person can only be scaled to a certain extent. For example, a system administrator gets automatically notification in case of unknown system errors, or errors which can not be easily recovered automatically. The system administrator will only be able to solve a single problem a time. Even more, if new notifications are raised before he was able to finish a former

task, he will very likely be stressed, annoyed or unwilling to take up new tasks. Hence, the scalability of human resources needs to be handled differently compared to scaling of computational resources.

2.3.2 Upside of Elastic Processes

If properly designed, a system which considers all three elastic dimensions equally (i.e., resource, quality and cost elasticity) while realizing elastic processes has several benefits over a system which considers only one elasticity a time, e.g., most of the time, only resource elasticity is considered. However, such a system needs to define cost and quality trade-offs if it is aimed at realizing elastic processes.

The system needs to deal with these trade-offs and find an optimal solution. Nonetheless, in some cases there is no optimal solution. Hence, the most fitting solution would be the one which satisfies all given SLAs, e.g., deadline constraints or an upper limit for cost a user is willing to pay. The resulting advantages are that the system is able to handle different resource requirements depending on the current demand, e.g., through leasing more resources during peak times and releasing them during off-peak times, the system is able to provide a certain level of cost-efficiency. In addition, such a system is able to provide a different level of QoS for different users/customers, e.g., depending on their needs and given SLAs.

However, in order to do so, reliable and effective provisioning of elastic applications requires precise engineering approaches and tools. An *elastic reasoning mechanism* is needed which is able to cover the multidimensional dynamics of elastic processes[32]. This means, this elastic reasoning mechanism needs to decide how to utilize the leasable resources in an optimal way. For example, such an algorithm can take the dynamic resource and cost information from a cloud provider and the desired quality of a service in order to reason on a possible action. Examples for such algorithm can come from the field of reinforcement learning, which describes a wider field of machine learning and artificial intelligence. It tackles a problem faced by an agent through trial-and-error interactions with a dynamic environment [54]. The system keeps on trying different action until "better" state has been reached. Alternatives can be decision trees [89], or complex algorithm which make use of optimization approaches while taking into account multiple objectives and variables in order to reason on an optimal solution [76].

In the scope of this thesis, resource and quality elasticity are the major driven elastic properties. However, dynamic changes in cost are achieved as positive side effects. We show that considering these two elastic dimensions equally during resource planning and scheduling. Hence, this will eventually lead to a better overall system status. This means, cost can be saved while the same (or even better) level of QoS can be provided.

2.3.3 Example Scenario

In the following paragraphs, we provide a basic example scenario to motivate this thesis and to illustrate how elastic processes can be useful in the real-world. We consider a scenario from the financial industry, since in banks IT cost account for 15%-20% of

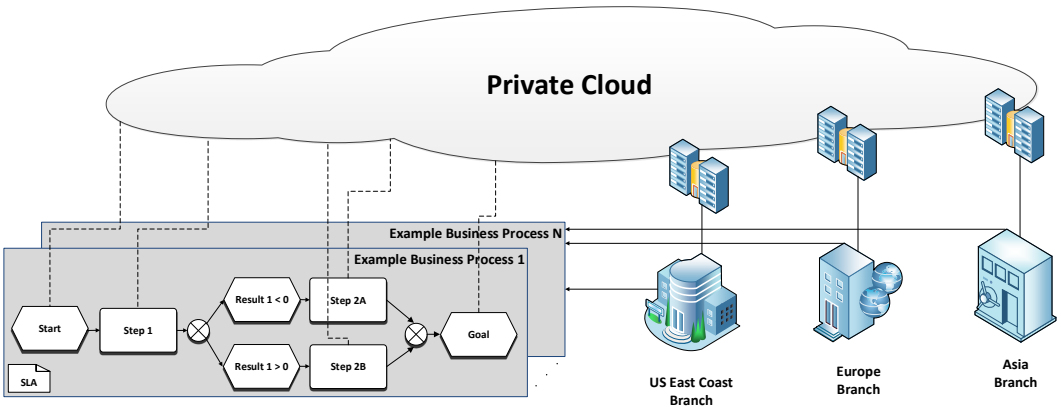


Figure 2.4: Example Scenario

the overall administrative expenses [64]. Hence, cost-efficient process enactment is an important goal in this domain. Notably, the presented example scenario is illustrative only and should neither be seen as complete nor exclusively for the applied banking domain. The work presented in this chapter can be easily used in any domain which features an extensive process landscape and needs to be able to adjust efficiently to an ever-changing number of process requests, e.g., the manufacturing industry [96] or Smart Grids [91].

Figure 2.4 shows a graphical representation of our example scenario: We consider an international bank featuring several branches which are worldwide distributed from *US East Coast* over *Europe* to *Asia*. As each of the branches provides similar products to their customers, the bank maintains a private cloud spanning all data centers of the distributed branches. This cloud provides all business processes which are used within the bank's products and services.

Every branch of the bank has access to the business processes in the cloud. Notably, the bank's branches have a large number of different business processes at their disposal. These may range from long-running data analytic processes to short-running trading processes. Especially in the latter kind of processes, time constraints are critical as even a short delay can lead to revenue loss or penalty payments. In order to ensure time constraints, a process is therefore equipped with a SLA which defines its deadline, i.e., the point of time at which the process enactment has to be finished.

Assuming the discussed bank simultaneously serves several hundreds or even thousands of customers, a very large number of business processes with different priorities may be requested at every point of time. This leads to high fluctuations of needed resources during peak and off-peak times. Also, the bank's process landscape is ever-changing – new orders and therefore process requests may arrive, while running process instances finish or have to be repeated. Thus, the usage of principles of elastic processes is an obvious approach [32]. Not surprisingly, elasticity and scalability have been named as primary reasons for the usage of cloud-based computational resources in the financial industry

[41]. For this, approaches to cost-efficient process scheduling and resource allocation – as discussed in this thesis – are needed.

2.4 ViePEP 1.0 – The Vienna Platform for Elastic Processes

In this section we present ViePEP which was first introduced in [97] and serves as the base for this thesis. ViePEP was created as BPMS testbed to foster research on elastic processes [32].

We take up the basic concepts of ViePEP in this thesis and gradually extend it resulting in a completely new framework. Although the concepts remained the same, the underlying architecture has been completely rewritten for this thesis. Hence, in later chapters we reintroduce and present the newly created ViePEP 1.1 and later on the final version: ViePEP 2.0 (see Section 6.4.1). However, before doing so, we introduce the original ViePEP 1.0 – the Vienna Platform for Elastic Processes.

As Figure 2.6 depicts, ViePEP consists out of five top-level entities acting as a unified framework. In order to motivate ViePEP's components, we apply self-adaptation techniques and concepts from the field of Autonomic Computing [57]. These techniques also include self-healing, i.e., the ability of a system to detect and recover from potential problems and continue to function smoothly, self-configuration, i.e., the ability of a system to configure and reconfigure itself under varying and unpredictable conditions, and self-optimization, i.e., the ability to detect suboptimal behavior and optimize itself to improve its execution [57].

The functionalities and components needed to provide self-optimization of a Cloud-based process landscape, are motivated according to the well-known MAPE-K cycle shown in Figure 2.5. This is done, since a system enacting business processes is commonly under permanent change, i.e., due to permanently arriving process requests and changing cloud resource utilization, a continuous alignment to the new system status is necessary. Using the MAPE-K cycle, the process landscape is continuously monitored and optimized based on knowledge about the current system status. In the following, we will briefly discuss the four phases of this cycle:

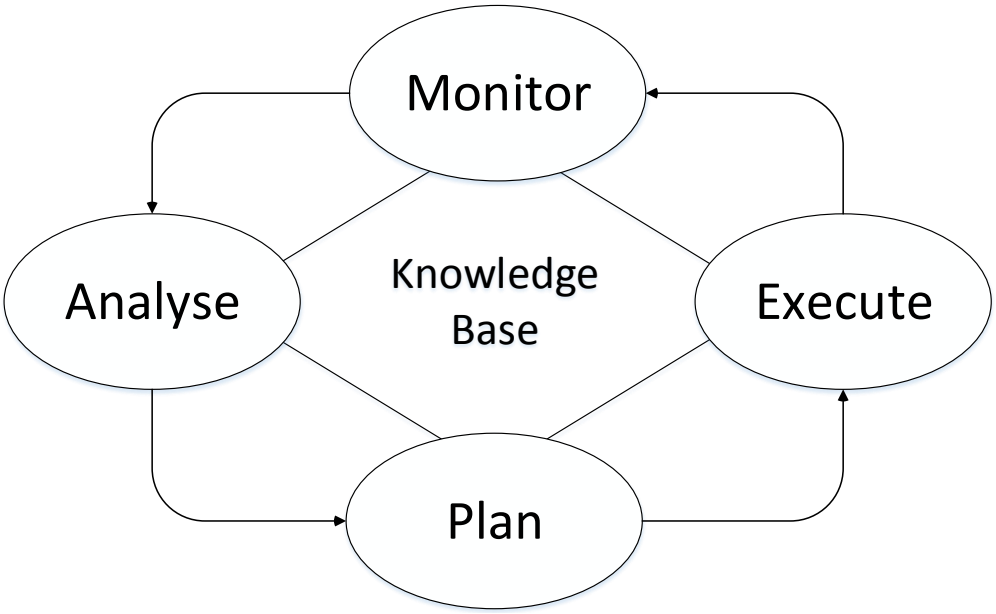


Figure 2.5: MAPE-K Cycle (adopted from [57])

- *Monitor*:
  In order to adapt a system, it is first of all necessary to monitor the system status. In the scenario at hand, this includes the monitoring of the status of single VMs in terms of CPU, RAM, and network utilization, and the non-functional behavior of services in terms of response time and availability.

- *Analyze*:
  To achieve Elastic Process execution, it is necessary to analyze the monitored data and reason on the general knowledge about the system. In short, this analysis is done in order to find out if there is currently under- and over-provisioning regarding the leased cloud-based computational resources (VMs) and to detect SLA violations in order to carry out corresponding countermeasures (e.g., provide further VMs or re-invoke a service instance).

- *Plan*:
  While the analysis of the monitored data and further knowledge about the system aims at the current system status, the planning also takes into account the future resource needs derived from the knowledge about future process steps, their SLAs, and the estimated resource requirements and runtimes. For this, a process scheduling and resource allocation plan needs to be generated

- *Execute*:
  As soon as the plan is set up, each service is executed corresponding to this plan.

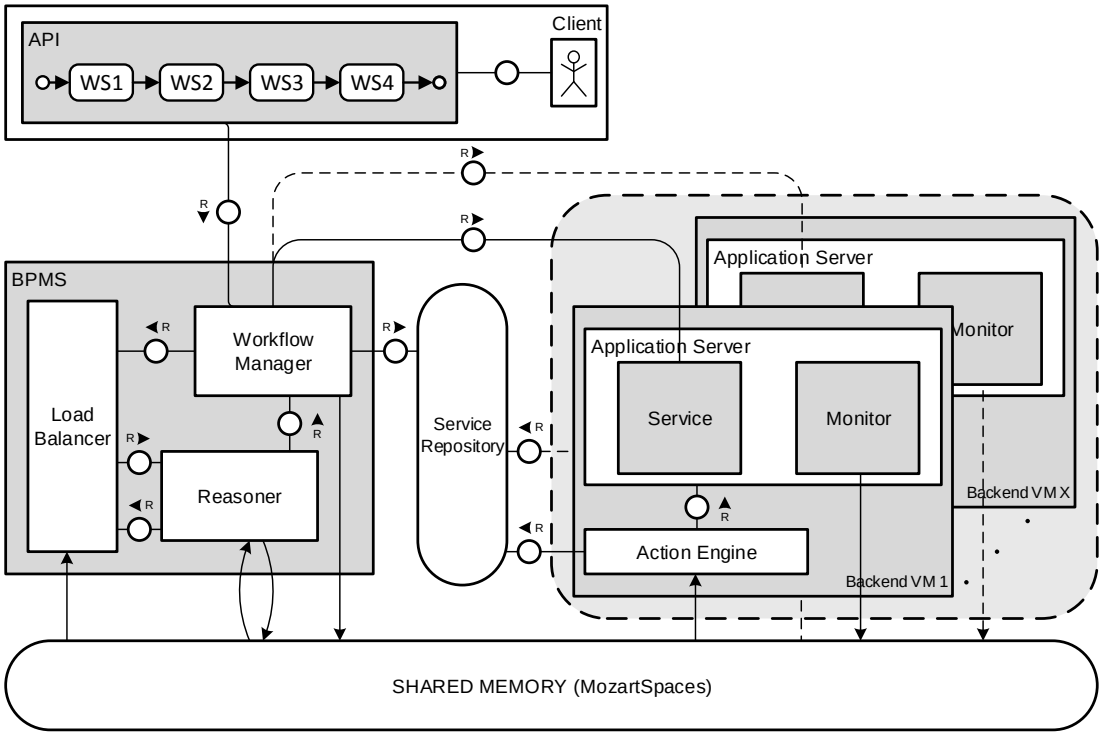


Figure 2.6: ViePEP 1.0 – Architecture

- *Knowledge Base*:
  While not really a part of the cycle, the Knowledge Base stores information about the system configuration. In our case, this is the knowledge about which service instances are running on which VMs, how many VMs are currently part of the system, and of course the knowledge about requested process and service instances

In order to provide self-adaptation, ViePEP supports all four phases of the cycle. To do so, ViePEP consists out of two major top level and two three helper components:

First, the *Client* models service-based business processes, i.e., a composition of several services steps which have to be invoked. In addition to that, the Client can provide optional deadlines, i.e., a particular point of time on which the process's execution has to be finished. Clients may request many processes consecutively or even simultaneously and ViePEP is able to serve several clients in parallel.

Second, the *BPMS VM* provides the core functionality of ViePEP in terms of a BPMS and cloud resource control solution. The subcomponents of the ViePEP BPMS are placed in a individual VM to avoid that it becomes a bottleneck. Through vertical scaling, it is possible to provide ViePEP with additional resources if the business process and cloud landscape becomes increasingly complex. The BPMS VM consists out of:

- The *Workflow Manager* is responsible for accepting client-issued business process requests during runtime. It stores these processes in a queue, where they are

stored until being scheduled for execution. Therefore, another responsibility of the Workflow Manager is the actual execution of these processes. Further, it measures the execution time for services, which is a prerequisite to detect and counteract against deviations from expected QoS behavior.

- The *Load Balancer* is able to balance the service invocations on the available resources in order to have an equal load on each VM.
- The *Reasoner* provides the core functionality of ViePEP. This component is responsible for creating a detailed scheduling plan according to the given deadlines. In addition, it arranges leasing and releasing the required amount of cloud-based computational resources.

A *Backend VM* hosts a particular service. Usually, several Backend VMs exist at the same time, each hosting a particular service (several instances of the same service may be hosted at different Backend VMs). Backend VMs are managed by the BPMS VM. The Backend VM hosts an *Application Server* on which the single services are deployed. In order to monitor the services regarding several QoS attributes, such as response time, the VM's CPU and RAM load, a *Monitoring* component is installed. Another important component is the *Action Engine* which is able to deploy and undeploy the services if requested by the Reasoner from the BPMS VM.

The *Shared Memory* and *Service Repository* are helper components. The Shared Memory is used to provide the possibility of sharing data and information between the BPMS VM and the different Backend VMs. In this version we chose MozartSpaces for our needs, as it provides the means of a peer-to-peer-based, distributed shared memory. We use this shared memory as a storage for the monitoring data. Further, we make use of the notification feature of MozartSpaces for sending action commands from the BPMS VM to the single Backend VMs.

The *Service Repository* hosts the service descriptions as well as their implementations as portable Web Application ARchive (WAR)-files which can be easily deployed on the Application Server. The Service Repository is queried by the Action Engine in order to find a particular service and deploy it on its Application Server.

ViePEP 1.0 was implemented as a standalone application using Java 1.6. It includes RESTful interfaces which are provided for the public, e.g., using these interfaces, Clients can request business processes for executions. As communication technology between Backend VMs and the BPMS VM we employ a shared memory, i.e., MozartSpaces[13] [60]. Beside of simple storage functionality, we use this shared memory for issuing notifications from the BPMS VM (i.e., from the Reasoner) to the Backend VM (i.e., towards the Action Engine). The single services are deployed using WAR-files which can be deployed on any arbitrary application server, e.g., in our case Apache Tomcat 6[14] is deployed. Last but not least, for monitoring the services and the underlying architecture in terms of CPU load and RAM usage we use PIS-Probe[15].

[13] `http://www.mozartspaces.org/2.3-SNAPSHOT/`
[14] `https://tomcat.apache.org/download-60.cgi`
[15] `https://github.com/psi-probe/psi-probe`

CHAPTER 3

Related Work

In the following chapter the most important related work in the field of this thesis is presented. This includes work from the field of resource allocation and service provisioning for single service invocations (Section 3.1), for Scientific Workflows (Section 3.2) and for business processes (Section 3.3).

3.1 Resource Allocation for Single Service Invocations

The main focus of this thesis is on business processes and their executions using cloud-based computational resources. However, resource allocation and automated service provisioning has a longer history for single service executions than for business processes. Hence, many methods and algorithms to allocate resources and schedule single service requests in the cloud have been proposed in recent years[17, 63]. In general these approaches focus on several different aspects. Cost optimization in terms of SLA penalties and resources utilization is one of the major topics. For example, Lampe et al. make use of a Knapsack-based heuristic in order to solve an assignment problem in which services on a SaaS level are assigned to particular VMs on the IaaS level [63]. The authors define this problem as Software Service Distribution Problem (SSDP). Similar to that, Lim et al. address the topic of automatic resource controlling by creating an horizontal platform for web applications, i.e., this platform is able to grow to serve a higher amount of requests by acquiring additional VMs. The underlying optimization model is based on linear regression and is supposed to be able to detect the *right* moment when a new VM should be leased or another one can be released. The authors continued their work in [72] in order to support a scalable storage controller. Urgaonkar et al. in [105] propose a dynamic provisioning technique for multi-tier web applications in the cloud, By using a flexible queuing model, their approach is able to determine how many resources are required for each tier including a predictive and reactive method which is able to determine when to acquire these resources.

Cao et al. propose an optimized algorithm for task scheduling based on ABC (activity based costing) [21]. Li and Venugopal provide an automatic mechanism to scale applications up or down on the IaaS level. In order to do so, they apply machine learning, i.e., a reinforcement learning approach [71]. This algorithm earns points if moving a service, starting or terminating a VM results in a better situation, e.g., better QoS such as throughput, response time, etc.

In other work, QoS and SLA validation and enforcement are also taken into account: In order to deliver such a solution where the needed QoS is considered in a cost-efficient way, Buyya et al. proposes the federation of distributed and independent cloud resources [17]. Further, Cardellini et al. modeled cloud resource management in terms of VM allocations for single service invocations as a mixed integer linear optimization problem and propose heuristics to solve it [22].

Wu et al. discuss dynamic resource allocation from a different perspective [120]. The authors aim at profit maximization for SaaS providers while scheduling service requests based on pre-defined SLAs between service provider and consumers. Beside of this, Litoui et al. aim at optimizing at different layers in order to gain profit for different stakeholders e.g., IaaS providers, PaaS providers, SaaS providers and end users [73]. Similar to that, Lee et al. consider also a three-tier cloud structure while addressing the challenge of service request scheduling in the cloud [66]. This structure consists of infrastructure vendors, service providers and consumers. Similar to that, Truong et al. propose a novel PaaS which aims at supporting dynamic life-cycle development and execution of elastic systems. Within the PaaS, the authors consider cost, resource and quality elasticity each associate with single components of the system [104].

The way how the profit can be maximized or how a better resource utilization can be realized differs for many approaches, e.g., by making use of a just in-time resource allocation mechanism, i.e., only reserve resources when they are actually needed [124]. Cost-efficient scheduling can also be reached through optimizing resource utilization. This has been addressed by Emeakaroha et al. in [34]. The authors consider different SLA parameters, such as the amount of CPU required, network bandwidth, memory and storage within their scheduling approach for deploying applications in the cloud.

The important characteristics of cloud-based services are the assurance of non-functional guarantees in the form of SLAs, such as guarantees of execution time and prices. In order to do so, Kertesz et al. propose a self-manageable architecture for SLA-based service virtualization. This provides a way to ease interoperable service invocations in the cloud [58].

Nguyen Van et al. do not try to acquire additional resources but try to change the settings of already existing ones. For that, the authors propose SLA-aware virtual resource management for the cloud [84]. The authors continue their work and propose an automatic virtual resource management approach for service hosting platforms[83].

Resource optimization for single services is an interesting research topic on its own. However, the discussed approaches lack of a process perspective across the utilization of resources. This means, the main focus was on single independent service scaling, i.e., there is no relation between two distinct services. A prominent approach is to scale

reactive, i.e., as soon as a specific event appear, the system needs to be scaled, e.g., if the CPU load exceeds a particular threshold or a service response time is above a specific value over a timespan. Other approaches propose the usage of proactive scaling, i.e, the amount of needed resources is predicted and resources are allocated in advance. The presented scaling strategies give a good insight and parts are applicable for our work.

3.2 Resource Allocation for Scientific Workflows

Beside of optimization for single services, a number of different scheduling and optimization approaches for scientific workflows have been proposed. However, it has to be noted that a direct application of these approaches to business processes is not possible as there are particular differences between scientific workflow and business processes [74]:

- First, business processes have a different goal than scientific workflows. While scientific workflows are driven by (scientific) experiments and evaluations, business processes are more business driven, i.e., business processes are meant to create a certain product or service to customers. Because of this, QoS and further SLAs have a much higher importance to business processes than scientific workflows.
- Second, another difference is that scheduling for scientific workflows usually only considers single workflows, while for business processes scheduling has to take care of a complex landscape of potentially hundreds or thousands of concurrent processes.
- Third, while business processes are rather control flow oriented, scientific workflows are more data driven. Consequently, scheduling for scientific workflows usually takes into account data transfer aspects [85], while this is rarely the case for business process scheduling approaches.
- Fourth, for scientific workflows, scheduling approaches apply always different process models, i.e., instead of sharing services concurrently between workflows, they are instantiated and carried out individually on a particular VM. As soon as the service invocation has finished, the VM hosting the service will either be released or reused for other services. This is a similar scenario as for single service invocation scaling. Hence, we see overlapping approaches in both fields.

However, some scheduling approaches present interesting aspects which may be also applicable for business process scheduling. For example, Hoffa et al. or Juve and Deelman also utilize cloud resources for the execution of scientific workflows [47, 53]. cloud-based computational resources are also used by Pandey et al. who propose the usage of Particle Swarm Optimization for scheduling scientific workflows [85]. The authors aim at minimizing total cost, which are combined cost of storage and data transfer cost. However, the authors neglect considering QoS and further SLA aspects which are an important factor in business process scheduling. Szabo and Kroeger apply evolutionary algorithms in order to schedule data intensive scientific workflows on a fixed amount of

VMs [101]. Again, the authors do not consider deadlines and consider only one workflow at a time. The latter constraint also applies to work by Byun et al. [19]. The authors present a scheduling and resource allocation approach which considers cost aspects and deadlines defined by users. The same is also addressed by Abrishami et al. who explicitly consider deadline constraints for IaaS providers [1].

Similar to the approaches used for scaling single service invocations, the presented approaches for scientific workflow scheduling and resource optimization offer interesting ideas and insights. However, the mentioned differences between scientific workflows and business processes prevent a direct adaptation [74].

3.3 Resource Allocation for Business Processes

Although the presented related work of scheduling and resource allocation for single service invocations (Section 3.1) or scientific workflows (Section 3.2) give an interesting insight into this subject matter, they can not be directly applied to business process scheduling. In fact, approaches which directly address (elastic) business processes are still scare. However, a number of researchers have already presented corresponding work.

Xu et al. share the same basic assumptions as we made for the work at hand, i.e., that processes are interdependent and that services can be shared among them [121]. The authors consider optimization of scheduling in respect to cost and time. However, QoS and further SLAs are not considered within their approach.

Business processes are also considered by Juhnke et al. in [52]. The authors provide an extension to a standard Business Process Execution Language (bpel) workflow engine. This is able to make use of cloud-based computational resources to execute business processes. Since this workflow engine is based on bpel, processes are composed from services, which mirrors the assumptions from our work. Through this extension, Juhnke et al. made it possible to execute an arbitrary amount of processes in parallel while optimizing scheduling and resource allocation. This is done in respect to cost and the overall execution time. The authors created an optimization problem which considers, apart from VM leasing cost, also data transfer cost. In order to solve this optimization problem, a genetic algorithm is applied. However, as deadlines are not considered, this approach can be compared to the above mentioned related work on scientific workflow scheduling.

Similar to that, Duipmans et al. propose a transformation-based approach for business process management in the cloud [30]. The authors assume that processes are composed from single steps, hence, their approach allows companies to control their business processes in a way that particular parts (steps) can either be executed on their own premise or on additionally leased resources in the cloud. For that, the authors propose an intermediate language and a transformation chain which allows to split business processes according to data and activity distribution.

The same assumptions were made by Bessai et al. [9, 10]. The authors also assume that processes are composed form single steps which are represented by software services. Within their work, the authors present different methods to optimize resource utilization

and process scheduling. The driving factor is cost and also time optimization or to find a pareto-optimal solution covering both, cost and time. An important aspect within their work is that process steps, and the corresponding software services can be shared concurrently among different processes [12]. However, in contrast to our work, the services will not share the same VM concurrently, i.e., the authors consider a one service per VM approach. Similar to Juhnke et al. [52], deadlines are not regarded. Further, both authors are do to consider different priorities for different processes, e.g., may it be through postponing particular processes to the future timeslot or through pre-poning processes to the current timeslot.

Wei and Blake and Wei et al. [114, 115] propose a similar approach: as in the other work, processes are build from single software-based services and the main focus in on optimizing resource allocation. However, although service instances may be shared among different processes [115], the authors do not consider parallel service invocations in different process instances, i.e., a particular service instance is only invoked by one process instance at a time. In their later work [116], Wei et al, extend their previous work and propose an adaptive and proactive resource management algorithm. The goal of the algorithm is to keep the total amount of leased resources for each service neither over- nor under-provisioned.

Contrary to that, in our work, we follow the *classic* service composition model, i.e., sharing and invoking service instances among different process models. Beside of this, although the authors do not consider any SLAs or deadline constraints, a process owner may define generic QoS constraints [117]. Since deadlines are not taken into account, a priority-based scheduling is not given. Although the authors do not consider cost explicitly, they provide mechanism aiming at saving cost. Similar to Bessai et al. in [11], Wei et al. also do not implement a testbed to evaluate their algorithms, but rather use simulation techniques in their evaluation runs. However, despite of the mentioned differences between our work and the work by Wei et al., there are several commonalities such as the allowance of different sized VMs and the attempt to predict a future resource demand.

Amziani et al. discuss the modeling of single elastic processes using Petri nets and simulate elasticity strategies, but do not take into account scheduling [3]. However, in his later work, Amziani and Mohamed et. al., propose a generic model that allows adding autonomic management facilities to any Cloud resource [81]. This model is designed according the MAPE-K cycle as used in our earlier work in [97] and within this thesis (see Section 4.2.1).

Janiesch et al. provide an extensive conceptual model for business processes in [51]. In order to evaluate their approach, the authors provide a corresponding testbed which makes use of Amazon Web Services. Similar to our approaches, the authors take into account SLAs, such as deadlines and aim at cost optimization. However, an automatic scheduling and resources allocation mechanism was not provided. Further, in contrast to our work, the authors do not use monitoring to gather data and compute the resource demand for upcoming service invocations, but assume that a correlation is given between the resource demand of different steps in a process.

A complementary scenario to the approach which is presented in this work was also applied by Gambi and Pautasso in [37]. The authors define design principles for executing RESTful business processes using cloud-based computational resources. However, as the authors propose a one-process-per-VM approach, i.e., a particular process instance is deployed on one particular VM, sharing computational resources or service instances among different process instances is not possible.

Only a few researchers, e.g., Cai et al. [20], Bessai et. al., [10, 12], Wei et. al., [114, 117] discuss business process scheduling in a way as we did. So far, most researcher see solely resource elasticity as the major driving factor. For that, different approaches have been proposed ranging from simple ad hoc-based scaling (Li et al. [71]) to proactive resource optimization (Wei and Bessai [116]). However, considering only resource elasticity is rather restrictive as neglecting the other dimensions of elastic processes will eventually lead to either unneeded high cost or a low level of QoS. Hence, in this thesis we aim at realizing elastic business process enactment in the cloud, i.e., we aim at considering all three dimensions equally (resource, quality and cost elasticity).

CHAPTER 4

Scheduling and Resource Allocation for Sequential Elastic Processes

As mentioned in the introduction section, BPM is a matter of great importance in many different industrial sectors, e.g., the financial industry as means to carry out trade settlement or execution control [40], or in the energy domain as means to carry out essential decision processes [91, 93]. One of the most important aspects of BPM is the automatic execution of processes and a large number of different approaches for modeling and executing such processes have been presented in the past [33, 82].

Efficient and effective resource allocation is a key issue in process executions [61], and a corresponding BPMS has to be able to administrate, schedule and execute hundreds of process instances simultaneously, which are each made up from several process steps. Such a BPMS is also responsible to assign needed computational resources to the single services representing the process steps [48]. Based on the functional diversity of these services, the requirements in computational resources do vary. Many of these services may be resource-intensive, especially if a large amount of data has to be processed in a short time [91]. Furthermore, some processes need to be carried out immediately, others may have to be done regularly, and others may have to be done until a specific deadline. Naturally, the computational resources needed to execute processes are subject to temporal variations – during peak times, the needed computational resources will be much higher than during off-peak times [32]. On the one hand, a BPMS equipped with a fixed amount of computational resources that could handle peak loads in a resource-intensive process landscape would lead to very high cost based on the *over-provisioning* of resources in off-peak times, since resources will not be utilized most of the time. On the other hand, a BPMS might not be able to carry out processes at all or with an unacceptable QoS, if there is a shortage (*under-provisioning*) of resources.

We presented such a BPMS in Section 2.4: ViePEP 1.0 – A BPMS which has several responsibilities: First, it has to create a scheduling plan for the requested processes, i.e., a detailed plan which defines at what time a particular process instance and its services need to be carried out in order to adhere to the given SLA. Second, it has to create a resource allocation plan and lease or release cloud-based computational resources. Third, ViePEP has to make sure that the scheduling plan is carried out and allocate resources according to the need of the single services. Last but not least, it has to conduct countermeasures in the case of service or resource failures.

ViePEP 1.0 is already partially able to execute business processes using cloud-based computational resources, i.e., it is able to realize *elastic processes* [4, 32]. However, the first version lacks a resource-optimization algorithm. Hence, we hereby extend ViePEP 1.0 by the means of a dynamic business process scheduling and resource allocation which makes use of cloud-based computational resources in an efficient way.

The remainder of this chapter is organized as follows: First, in Section 4.1 we define some prelimiaries including the overall scenario description. In Section 4.2 we reintroduce ViePEP and extend it. For that, we present a slightly adapted version of the well-known MAPE-K cycle [57]. Afterwards, we introduce the used scheduling (Section 4.3) and reasoning algorithms (Section 4.4). In Section 4.5, we evaluate the algorithms. The chapter closes with a summary in Section 4.6.

4.1 Preliminaries

We assume that a business process landscape is made up from a large number of business processes, which can be carried out automatically. The part of a business process which can be executed using machine-based computational resources is also known as a workflow [74], however, within this thesis we use the term processes.

While we assume that processes are composed of software services, we do not expect that companies are willing to outsource important services fully to external providers, since these services will then be outside of the control domain of the process owner. In contrast, we make use of private and public cloud-based computational resources to host service instances, which are then invoked as process steps and leaves the control with the process owner: If particular service instances or VMs fail, the process owner or the admin of the business process landscape is able to deploy further service instances. Making use of VMs to host particular services allows sharing resources among processes, as the same service instance might be invoked within different processes at the same time. Notably, it is important to distinguish between *services* (from a specific *type*), which are the basic building blocks of *processes*, *service instances*, which are hosted by VMs, and *service invocations*, which denote the unique execution of a service instance in order to serve a particular process request, i.e., a process instance.

In our scenario, processes may be requested at any time by process owners (called *clients* in the following) and may be carried out in regular intervals or nonrecurring, i.e., ad hoc. For that, clients can choose from a set of *process models* and request them by instantiating one or more, i.e., resulting in a particular *process instance*.

It is the duty of a software framework (in our case, ViePEP), which combines the functionalities of a BPMS and a cloud-based computational resource management system, to accept incoming process requests, schedule the process steps, and lease and release computational resources based on the process scheduling plan. The clients are able to define different QoS constraints as SLA on the level of process instances. Without a doubt, the deadline is the most important SLO: Some business-uncritical process instances may have very loose deadlines, while other process instances need to be carried out immediately and finished as soon as possible. Usually, process instances will have a defined maximum execution time or a defined deadline. The clients are able to define complex processes which feature AND-block, XOR-block or Repeat loop constructs. However, we assume that the next steps in a particular process instance are always known. We assume that the business process landscape is volatile, i.e., ever-changing, since process requests may arrive at anytime. Furthermore, changes may have to be necessary since services or VMs are not delivering the expected QoS, or the next steps in a process instance are not as planned.

For achieving an efficient scheduling and invocation of processes and corresponding services, cloud-based computational resources in terms of VMs are used. The software services are deployed on the VMs in order to be invoked. Each service is of a specific *service type* and needs to be deployed on an arbitrary VM resulting in a *service instance*. There might be more than one service instance of the same service type at the same time. The same applies for VMs, i.e., there might be an unspecific amount of VMs of each type, i.e., *VM instances*.

Overall, the total cost arising from leasing aforementioned cloud-based computational resource are subject to be minimized. In addition, it has to be made sure that given QoS constraints such as deadlines, until which corresponding process instances have to be finished, are satisfied. The closer the deadline is for a certain process instance, the higher is the importance to schedule and execute corresponding services accomplishing its steps. If not carefully considered and scheduled, further additional cloud-based computational resources will have to be leased and paid in order to execute process instances that cannot be delayed any further. For avoiding such situations in which extra resources have to be leased due to an inefficient scheduling strategy, the execution of process instances along with the leasing and releasing of cloud-based computational resources has to be optimized. This makes it necessary to facilitate self-adaptation and self-optimization of the overall business process landscape through replanning of process scheduling and resource allocation.

4.2 ViePEP 1.1

ViePEP 1.0 was firstly introduced in [97] and described in Section 2.4. It provides mechanisms and architectural components that are beyond those of traditional BPMS approaches [32], as it combines the functionality of a cloud resource management system and a BPMS. The platform has to manage a potentially highly volatile business process landscape, as process requests permanently arrive, leading to a changing demand of

required resources. This makes it necessary to continuously align and optimize the system landscape, which is made up

1. from the business process landscape and
2. the leased cloud-based computational resources.

ViePEP operates on the PaaS level [5], i.e., ViePEP is able to lease and release cloud-based computational resources in terms of VMs from a cloud provider and deploy services on them. For this, ViePEP is able to schedule process instances and single process steps based on predefined rules, and allocate cloud-based resources to execute them. ViePEP provides the capabilities to estimate the resource demand for carrying out single services and is able to lease and release cloud-based computational resources based on this information.

In order to support process scheduling and resource allocation on a more complex level, we extend ViePEP 1.0 (Section 2.4) and present here a novel approach to reason about the amount of needed resources. For that, we apply techniques from the field of autonomic computing and self-adaptation for elastic process executions according to a modified MAPE-K cycle [57]. Further, we split up the core component of ViePEP: *Reasoner* into two subcomponents: (i) *Reasoner* and (ii) *Scheduler*.

4.2.1 Self-Adaption for Elastic Process Execution

Self-adaptation is a common concept from the field of autonomic computing and includes *self-healing*, i.e., the ability of a system to detect and recover from potential problems and continue to function smoothly, *self-configuration*, i.e., the ability of a system to configure and reconfigure itself under varying and unpredictable conditions, and *self-optimization*, i.e., the ability to detect suboptimal behavior and optimize itself to improve its execution [57].

The BPMS, i.e., ViePEP 1.0 has been designed as a self-aware BPMS and is able to setup and reconfigure itself if the system landscape has changed (*self-configuration*), to detect and recover from an error in order to be fully functional again (*self-healing*), and to optimize itself in order to improve its execution (*self-optimization*). This approach has been motivated by techniques and methods from the field of autonomic computing and more in particular, ViePEP 1.0 was designed according to the MAPE-K cycle. However, in order to provide a more sophisticated scheduling and resource optimization approach as presented in Section 4.3 and Section 4.4, we slightly adapt this MAPE-K cycle and split up the *Planning* phase into a *Resource Planning (Reasoning)* and *Scheduling Planning (Scheduling)* phase (Figure 4.1). This is done with the motivation that these two phases directly influence each other. For the sake of completeness, the full MAPE-K cycle consists out of:

- *Monitor:*
  A system has to be aware of its current state in order to be able to adapt itself to internal and external influences such as the changing demand of required resources.

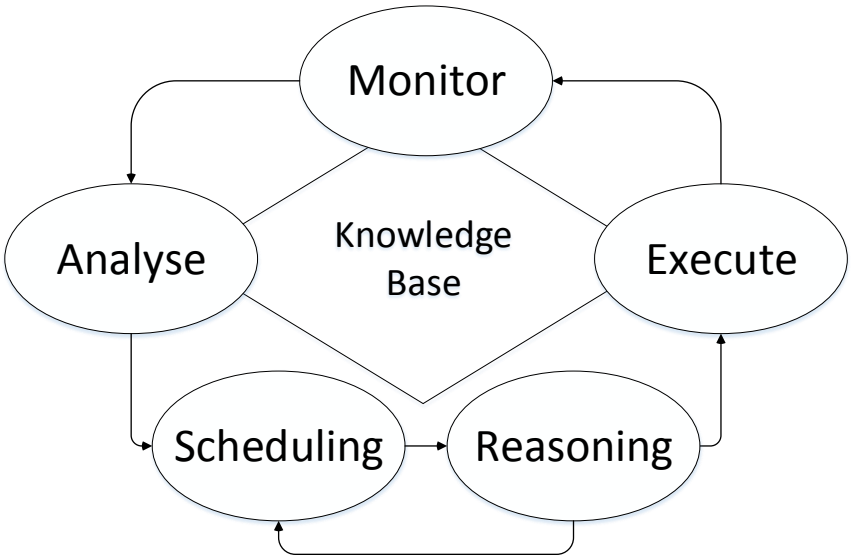


Figure 4.1: Adapted MAPE-K Cycle (adapted from [57])

To achieve this, ViePEP provides the means of monitoring the status of single VMs in terms of CPU load and RAM usage. Further, non-functional behavior of the deployed services, such as response time and availability, is monitored. This functionality was already provided in ViePEP 1.0 and has only been slightly improved.

- *Analyze:*
  In order to create an appropriate scheduling plan, it is crucial to take into account the monitored data and to reason the general knowledge about the system. This analysis aims at detecting if the system is over- or under-provisioned in terms of leased cloud-based computational resources. In addition to that, the analysis of historical data aims at detecting potential deadline violations in time in order to find appropriate countermeasures.

- *Scheduling:*
  In order to make sure that process deadlines are met, it is necessary to create a detailed execution plan of each single process step, i.e., at which point of time a particular service has to be invoked. This scheduling plan is highly dependent on the resource planning. Therefore ViePEP 1.1 will implement a mutual communication

flow between those two steps as can be seen in the adapted MAPE-K cycle in Figure 4.1 and in ViePEP's 1.1 architecture in Figure 4.2.

- *Reasoning:*
  While the previous phase aims at creating a detailed scheduling plan, this phase is responsible for calculating the amount of required computational resources in order to fulfill exactly this plan. For that, the service invocations are assigned to the available resources arranged in timeslots according to their deadlines. If there are not enough resources available, additional ones will be leased; if resources are superfluous, resources will be released.

- *Execute:*
  As soon as the schedule planning and resource planning has been done, each service is invoked as planned and resources are leased or released according to this plan.

- *Knowledge Base:*
  While not really a part of the cycle, the Knowledge Base stores information about the system configuration. In our case, this is the knowledge about which service instances are running on which VMs, how many VMs are currently part of the system, and the knowledge about queued processes.

4.2.2 ViePEP 1.1 – Architecture

As ViePEP 1.0, ViePEP 1.1, the updated version consists out of five top level entities (depicted in Figure 4.2):

Although not really part of the framework, the *Client* can request business processes for execution. These are forwarded to the *BPMS VM*, which hosts the core functionalities of ViePEP in terms of a BPMS and cloud resource control solution. While the components *Process Manager* and *Load Balancer* remain the same, the main difference to the earlier version of ViePEP is that the newly updated version contains a *Scheduler* and a *Reasoner*, which are responsible for creating a detailed scheduling plan according to the process instance deadlines, and lease or release the required cloud-based computational resources. For a detailed descriptions how the Scheduler and Reasoner operate, please refer to Section 4.3 and Section 4.4.

On the BPMS VM, the Reasoner is able to control a number of *Backend VMs*, each hosting a particular service on an *Application Server*. The *Action Engine* is able to deploy and undeploy services if requested by the Reasoner. The Reasoner can issue a request to the cloud provider to either start a Backend VM and when its Action Engine is ready, to deploy a new service instance, or (in case the Backend VM is already running) the Action Engine is directly able to execute the corresponding request. ViePEP allows the following actions:

- *Start* a new Backend VM, which includes deploying a new service instance on it.

- *Duplicate* a Backend VM including the deployed service instance.

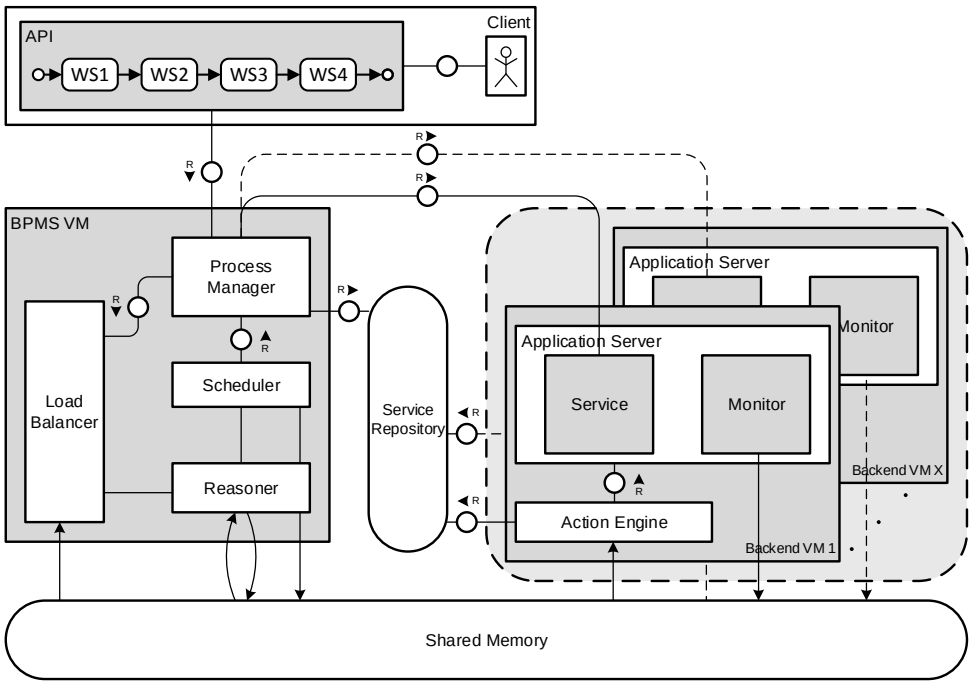


Figure 4.2: ViePEP 1.1 – Architecture

- *Terminate* a Backend VM, which marks the Backend VM as *phasing out* and will prevent the Load Balancer from requesting further service invocations from this Backend VM. Once all previously assured service invocations have been finished, the VM is terminated.
- *Exchange* the hosted service instance on the Backend VM. This will also prevent the Load Balancer from requesting any further invocations of this particular service instance. Once all assured service invocations have been finished, the instantiated service is replaced by another one.
- *Move* a service instance from one Backend VM to another one (with different computational resources).

Last but not least, the *Shared Memory* and *Service Repository* provide data storage capabilities for services and monitoring data.

ViePEP 1.1 was developed as a standalone application using Java 1.6. For the shared memory we chose MozartSpaces [60], which provides the means of a peer-to-peer-based distributed shared memory. It is used as a storage for the monitoring data. Further, we make use of the notification feature of MozartSpaces to send action commands from the BPMS VM to the Backend VMs. The services are portable WAR-files which can be easily deployed on an arbitrary Application Server, in our case Apache Tomcat 6[1].

[1] `https://tomcat.apache.org/download-60.cgi`

Further, the services are monitored using PSI-Probe[2].

So far, the process scheduling and reasoning algorithms of ViePEP 1.0 only considered deadline constraints, i.e., in form of priorities: the closer a process instance's deadline is, the higher is its priority. However, there has been no optimization of resource utilization considering deadlines more in detail. Within the next two sections, we present a more sophisticated scheduling and reasoning approach which enables resource-efficient process scheduling.

4.3 Scheduling

4.3.1 Problem Statement

A scheduling algorithm for elastic processes is responsible for finding a process execution plan which makes sure that all process instances are carried out under given constraints. These constraints are defined SLAs. Based on this scheduling plan, the Reasoner is able to allocate, lease, and release cloud-based computational resources.

Scheduling has to be done continuously for an unknown duration of time, across a system landscape including the business process landscape as well as cloud-based computational resources, so that:

- All SLAs defined for process instances are met.
- Resources are utilized in an efficient way, i.e., the cost for leasing VMs over the reckoned timespan should be optimized.
- Scheduling and reasoning needs to be redone once the system landscape changes, as new process instance requests arrive, or the predicted resource utilization of VMs does not apply.
- In addition, scheduling and reasoning are done in regular intervals. This interval can be set by a system administrator.

In order to find such a scheduling plan, we make use of the following constraints:

- Each Backend VM hosts exactly one service instance, i.e., it is not possible that different service instances (from different types) are instantiated at the same Backend VM.
- All VMs are from the same type, i.e., they offer the same capabilities in terms of computational resources and cost.

[2] `https://github.com/psi-probe/psi-probe`

4.3.2 Process Descriptions and Monitoring Data

ViePEP accepts new process instance requests at any time, and maintains a queue of process instances which are ordered by their individual priorities, i.e., they are sorted by the client-defined deadlines for the process instnace execution. In order to accept new process instances and to create a scheduling plan for the queued process steps, a process description language is needed. Formally, a process instance is defined as following:

$$P_i = \{sla_p, s_1, s_2, ..., s_n\}$$

$$s_i = \{sla_s, id\}$$

$$sla_i = \{slo_1, slo_2, ..., slo_n\}, i \in \{p, s\}$$

P_i defines a process instance with ID i which consists out of 1 to n single sequential service steps expressed by s_1, ... ,s_n. Each service step must provide a mandatory field id which is the unique identifier of the service representing this step. A process instance can have an optional SLA (sla_i) consisting out of several SLOs expressed by slo_1, ... , slo_n on process instance (sla_p) or service level (sla_s). For now, we consider solely the deadline as SLO, i.e., the scheduling takes into account that process instances have to be finished before a particular point of time; deadlines for single steps are then derived from this SLO if they are not explicitly defined. For this, we define

$$sla_p = deadline_p$$

$$sla_s = deadline_s$$

In the following, we assume that SLA and deadlines defined by the process owner (client) are valid and achievable. For example, regarding deadlines this means that the execution duration of the single steps in a process instance does not exceed the set process execution deadline and that the deadline is in the future.

ViePEP monitors service invocations continuously regarding execution duration and needed cloud-based computational resources in terms of RAM and CPU. These values are assigned to the corresponding services, thus monitoring data for a particular service can be described as follows:

$$m_{ID} = \{d, cpu, ram, |ID|\}$$

m_{ID} defines one monitoring data entry for the service with the id ID, i.e., the monitoring data for one particular service invocation of service s_i, i.e., a service with the ID ID. d defines the service execution duration, cpu defines the CPU usage, ram defines the RAM usage, and $|ID|$ defines the number of concurrent invocations of service s_i at the time of monitoring at the particular VM it is running on. For now, the CPU value is used for the resource demand prediction (see Section 4.4) and the duration value is needed to create a efficient scheduling plan.

Based on this data, we are able to derive the needed resources and duration of a single service invocation, as presented in [45].

4.3.3 Scheduling Algorithm

ViePEP maintains a queue of process instances, which are sorted by the client-defined process instance execution deadlines. A client is able to specify an execution deadline, i.e., the latest date and time when the process instance execution has to be finished.

If this is the case, we can calculate the overall process instance execution duration t_p as follows:

$$t_p = \sum_{i=1}^{n} t_{s_i} + \epsilon$$

As can be seen, t_p is the sum of each single service invocation duration (t_{s_i}) plus a safety margin ϵ. t_{s_i} has been calculated based on the monitoring data as described above. ϵ can be set independently for each single process instance by the client. A large ϵ indicates a low risk to not meet the execution deadline, as there is a certain time buffer if a service invocation takes longer than estimated. Vice versa, if ϵ has been set to a small value or even 0, there is a high risk that the execution deadline cannot be met as there is no time buffer left.

Based on t_p, the latest starting time $start_p$ can be calculated:

$$start_p = deadline_p - t_p$$

where $deadline_p$ is the given SLA which defines the process instance's execution deadline and t_p is the process instance execution duration calculated in the last step.

In order to compute at which point of time a particular process step has to be executed, thus the given SLA is not violated, we have to compute the latest starting time for each single process step s_i accordingly:

$$start_{s_i} = start_{s_{i+i}} - t_{s_i}$$

where $start_{s_{i+i}}$ defines the latest starting time of the next step in the process instance, and t_{s_i} defines the execution duration of step s_i. As can be seen, this is a recursive function, as it is necessary to derive the latest starting time of the next step. However, for the last step s_n in a process instance, the execution deadline is identical with the process execution deadline:

$$deadline_{s_n} = deadline_p$$

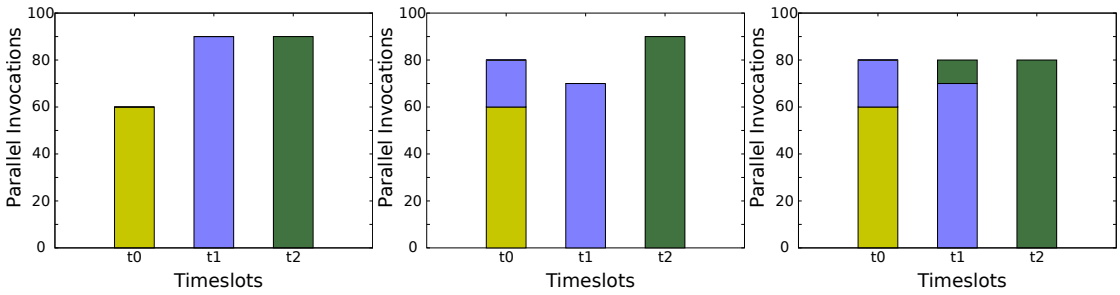
Hence, the latest starting time for the last step in a process instance is:

$$start_{s_n} = deadline_p - t_{s_n}$$

In case deadlines have been set for single services, the starting time for step s_i is set by:

$$start_{s_i} = deadline_{s_i} - t_{s_i}$$

After the starting time is calculated for each single process step in the queue, a full scheduling plan for the process instance executions is available, i.e., ViePEP has now the information at which point of time how many service invocations have to be performed. This information is forwarded to the Reasoner (Section 4.4) which is responsible for reasoning about the required demand of resources.



(a) Initial State (b) First Round Optimized (c) Final State

Figure 4.3: Timeslot Scheduling

4.4 Reasoning

Algorithm 1 Method Reasoning()

```
1:  procedure REASONING
2:      for i = 1 to timeSlotsList.size() do
3:          timeSlot_t0 = timeSlotsList.get(i);
4:          timeSlot_t1 = timeSlotsList.get(i + 1);
5:          demand_t0 = computeResourceDemand();
6:          if demand_t0.isFullyUtilized() then
7:              continue();
8:          end if
9:          while demand_t0.isNotFullyUtilized() do
10:             moveInvocations(timeSlot_t0, timeSlot_t1);
11:             demand_t0 = computeResourceDemand();
12:         end while
13:         reasoning();
14:     end for
15: end procedure
```

In the previous section, we described how deadlines of individual process steps and the corresponding starting times are assigned in order to fulfill a given SLA. The result of this step is a detailed plan of which process step should be executed at which point of time. Based on this information, it is now possible for the Reasoner to derive the resource demand. To compute the needed amount of computational resources, each different service is considered separately, as only one service instance per Backend VM is allowed to be instantiated.

Figure 4.3 represents a scheduling plan for a specific service instance. On the horizontal axis, the time periods are shown, while the vertical axis show the number of planned parallel service invocations at a specific timeslot t_i. Figure 4.3a shows the initial state as it would be provided by the Scheduler: In t_0 there would be a total of 60 parallel

invocations, in t_1 90 and in t_2 also 90. By using Ordinary Least Square (OLS) linear regression, we can compute the total amount of required resources for one timeslot based on historical monitoring data [112]. This process instance is based on the notion that a dependent variable – corresponding to the resource utilization of a Backend VM for a particular service instance – can be derived through the linear combination of a set of independent variables, corresponding to the individual monitoring results for this service instance. In the following, we will describe the outcome of the OLS-based regression analysis based on the example from Figure 4.3.

In the example, the regression analysis has led to the outcome that a single Backend VM is able to handle 80 invocations of a particular service instance in one single timeslot. Please note that this analysis would be done for all service instances in parallel. Furthermore, the degree of maximum utilization of a Backend VM can be set as a parameter by the system admin. A high maximum utilization would lead to a higher saturation of the computational resources of a VM, but would also yield a higher risk of under-provisioning, as there is only a small resource buffer.

Taking the original scheduling plan into account, in the first timeslot t_0, we have 60 parallel service invocations scheduled. The Reasoner would calculate the demand of resources for these 60 invocations and would derive that it is possible to do even more invocations in t_0, as there are still resources available and not exploiting them would lead to a waste of resources. Therefore, we present a reasoning algorithm which is able to move the scheduled invocations from one timeslot to another in order to fully utilize the acquired resources. The algorithm is presented in form of pseudocode in Algorithm 1.

The input for this algorithm is the list of queued process steps (services) which is sorted by their execution deadline (*timeSlotList*). Each timeslot is exactly as long as one service invocation lasts; however, as reasoning is done for each service separately, different services in a process instance will have different timeslot lengths. From line 2–14, we iterate over these timeslots and consider each timeslot separately. Since the Scheduler has assigned a number of process steps to each timeslot, we can compute the amount of required resources for this timeslot (see line 5 in the Algorithm 1). This is done by calling the method *computeResourceDemand*() which is based on OLS linear regression.

In the next step, we calculate if the required resources are fully utilized (line 6). If this is the case, we continue with the next timeslot (line 7), if not, service invocations from the next timeslot are pre-poned to the current timeslot in order to try to fully utilize the acquired resources (lines 9–12). Naturally, this leads to a lower number of service invocations in the next timeslot. Hence, it is necessary to replan the scheduling. Therefore, the *reasoning*() method is called recursively (see line 13).

For the example, Figure 4.3b presents the result of the first round of scheduling: 20 invocations were shifted from t_1 to t_0. In the next step, the second timeslot will be considered. After the first round of replanning the scheduling, only 70 invocations are scheduled for timeslot t_1. Since the available resources are able to handle more invocations in parallel, 10 further invocations are shifted from the t_2 to t_1. The result is presented in Figure 4.3c. At this point, we want to state that the presented numbers

in this example are only for illustration and do not echo the result of a real evaluation. Evaluation results will be presented in the next section.

Based on the outcome of the replanning of the original scheduling, the Reasoner is now able to deduct the required resources and will correspondingly trigger the start and termination of Backend VMs, duplicate a Backend VM, move a running service to another Backend VM, or exchange the service hosted by a Backend VM by another service.

4.5 Evaluation

In the following, we will first shortly describe the evaluation scenarios and secondly we will present the results of the evaluation runs.

We used ViePEP 1.1 for our evaluation and implemented the scheduling and reasoning algorithms (see Section 4.2). As test environment we used an OpenStack-based cloud.

4.5.1 Evaluation Scenario

For the evaluation, we chose a particular business process which has to be processed 20,000 times. This process model consists out of 5 individual steps realized by RESTful services, i.e., a *Dataload Service* which loads data from a sensor, a *Pre-Processing Service* which performs some simple computations with the loaded data, a *Calculation Service* which processes the given data, a *Reporting Service* which generates a report out of the results, and a *Mailing Service* which delivers a mail containing the report. Each individual service needs a different amount of computational resources in terms of CPU and RAM. The maximum utilization parameter for the Backend VMs is set to 80%, i.e., scheduling and resource allocation will only utilize max. 80% of the offered resources of a Backend VM. The concrete value of 80% was chosen based to achieve a high resource utilization on each VM, while maintaining necessary reserves in case of unexpected errors, e.g., another VM needs longer to start or fails and needs to be restarted. In addition, a the needed amount of CPU per invocation is not deterministic, hence, having *spare* 20% will allow slight deviations.

4.5.2 Process Request Arrival Patterns

We make use of three distinct process request arrival patterns:

- *Constant Arrival Pattern*:
  In this arrival pattern, the process requests arrive in a constant manner. An automated client sends 200 requests simultaneously to the BPMS every 10 seconds, i.e., $y = 200$.
- *Linear Arrival Pattern*:
  In this arrival pattern, the process requests arrive in a linear rising function, i.e., $y = k * \lfloor \frac{x}{3} \rfloor + d$. The number of constantly rising parallel requests is increased by 10 every 10 seconds.

- *Pyramid Arrival Pattern*:
  In this arrival pattern, the process requests are sent to the server according to a pyramid-like function. We start at 0 parallel requests and increase it to a randomly chosen peak. As soon as this peak is reached we decrease the number again to 0 and repeat this procedure until the maximum number of process requests has been covered (here: 20,000).

The different arrival patterns have been chosen in order to show how the algorithms perform under different degrees of volatility. While the number of process requests is quite predictable in the constant and linear arrival patterns, the pyramid-based arrival pattern follows a more unpredictable arrival pattern. The arrival patterns are shown as "Process Requests" in Figure 4.4.

4.5.3 Metrics

In order to get reliable numbers, we executed each arrival pattern three times and evaluated the results against the following three quantitative metrics:

- *Total Makespan in Minutes:* This metric defines the total execution makespan of all 20,000 process instances, i.e., this is the timespan from the first process instance to the moment when the last process step has been processed.
- *Max. Active Cores:* The second metric is the amount of concurrently leased number of cores, i.e., the maximum of leased (CPU) cores.
- *Cost:* This metric is a combination of the former two, i.e., it defines the cost emerging from the acquired resources. The cost are defined in VM-Minutes, i.e., one VM-Minute is defined as one VM running for one minute. VM-Minutes can be directly mapped to common pricing schemes, e.g., from Amazon EC2.

In the evaluation runs, we compare the "Scheduling + Reasoning" approach against a baseline which makes use of an ad hoc approach ("Pure Scheduling"), i.e., applies scheduling and resource allocation, but does not apply the reasoning. This means that in the baseline scenario additional VMs are leased in an ad hoc approach, i.e., whenever the threshold is exceeded additional VMs are acquired, or if the load falls short of a lower threshold (20%), unneeded VMs are released. The lower threshold of 20% has been chosen in order to tolerate the CPU load the OS will approximately need.

4.5.4 Results

The results of our evaluation in terms of the average number of all evaluation runs can be found as numbers in Table 4.1-4.3 and as graphical presentation in Figure 4.4. "Pure Scheduling" denotes the ad hoc approach without reasoning, while "Scheduling + Reasoning" marks the application of the complete approach as presented in Section 4.3-4.4.

Table 4.1-4.3 presents the observed metrics as presented in Section 4.5.3 for all three arrival patterns (constant, linear, and pyramid). The tables state the average

Table 4.1: Evaluation Results – Constant Arrival

	Constant Arrival	
	Pure Scheduling	Scheduling Reasoner
Number of Process Requests	20,000	
Interval between two Request Bursts (in Seconds)	10	
Number of Requests in one Burst	$y = 200$	
Total Makespan in Minutes (Standard Deviation)	25.67 (σ=2.09)	23.93 (σ=0.42)
Max. Active Cores (Standard Deviation)	21 (σ=*)	14 (σ=*)
Cost in VM-Minutes (Standard Deviation)	302.5 (σ=13.60)	277.44 (σ=7.18)

**unfortunately this value was not available*

Table 4.2: Evaluation Results – Linear Arrival

	Linear Arrival	
	Pure Scheduling	Scheduling Reasoner
Number of Process Requests	20,000	
Interval between two Request Bursts (in Seconds)	10	
Number of Requests in one Burst	$y = 10 * \lfloor \frac{x}{3} \rfloor + 10$	
Total Makespan in Minutes (Standard Deviation)	44.52 (σ= 3.01)	42.04 (σ= 6.91)
Max. Active Cores (Standard Deviation)	24 (σ=*)	24 (σ=*)
Cost in VM-Minutes (Standard Deviation)	545.05 (σ=20.62)	537.72 (σ=14.03)

**unfortunately this value was not available*

of the evaluation runs including the standard deviation (σ). Figure 4.4 completes the presentation of the average evaluation results by depicting the arrival patterns ("Process Requests") over time and the number of running VMs until all process instances have been served. Again, evaluation results for "Pure Scheduling" and "Scheduling + Reasoning" are shown. In some cases it may happen that the "Process Requests" are ending before the whole scenario is finished. This can be reduced to the fact that each process instance has a deadline in the future and some process instances are still waiting in the queue which have to be processed.

Table 4.3: Evaluation Results – Pyramid Arrival

	Pyramid Arrival	
	Pure Scheduling	Scheduling Reasoner
Number of Process Requests	20,000	
Interval between two Request Bursts (in Seconds)	20	
Number of Requests in one Burst	Based on a Random Number	
Total Makespan in Minutes (Standard Deviation)	40.22 (σ=0.67)	38.5 (σ=1.2)
Max. Active Cores (Standard Deviation)	21 (σ=*)	20 (σ=*)
Cost in VM-Minutes (Standard Deviation)	555.16 (σ=25.98)	545.56 (σ=17.30)

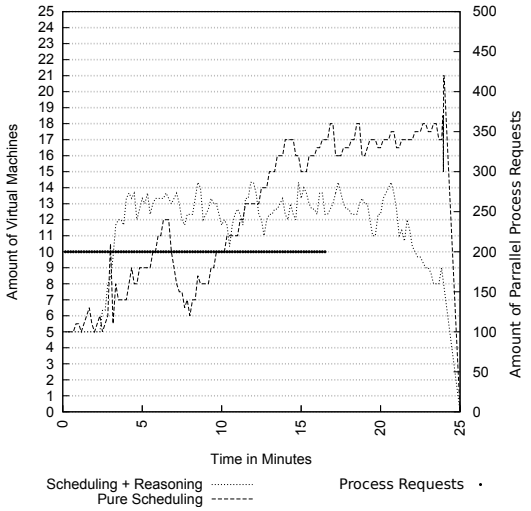
**unfortunately this value was not available*

As presented in Table 4.1-4.3, for all three observed arrival patterns, the application of "Scheduling + Reasoning" leads to lower cost if compared with "Pure Scheduling": The constant arrival pattern is seen as the best case scenario. Both the "Pure Scheduling" and the combined approach lead to relatively low cost, i.e., 302.5 VM-Minutes vs. 277.44 VM-Minutes. The combined approach leads to cost savings of 8.28%. This shows that in this arrival pattern, our approach was able to reduce the total makespan in minutes (from 25.67 to 23.93 minutes, i.e., 6.78%) while allocating less resources. In contrast to that, the savings from the combined approach in the linear arrival pattern ($\sim$ 1.34%) and the pyramid arrival pattern ($\sim$ 1.73%) are comparably small. This can be traced back to the fact that in these cases, there is not as much information available about future resource demands, as the number of process instances changes permanently and therefore it is more difficult to pre-pone service invocations to earlier timeslots. Furthermore, these arrival patterns require more changes in the numbers of Backend VMs per service, which also leads to higher cost.

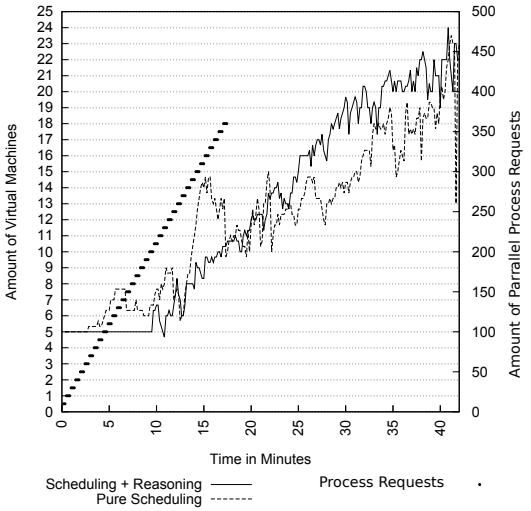
Regarding the time savings, the constant arrival pattern features the largest saving for the "Scheduling + Reasoning" approach ($\sim$ 6.78%) , while the linear arrival pattern ($\sim$ 5.57%) and the pyramid arrival pattern ($\sim$ 4.28%) feature smaller numbers.

Taking the cost and time savings into account, it can be deduced that the combined approach speeds up the total processing time, but the Reasoner is not able to fill up the Backend VMs to the same degree for all arrival patterns. Once again, this can be traced back that in the linear arrival and pyramid arrival pattern, information about process instance arrives in a more unpredictable manner and the number of Backend VMs per service is more volatile. As we can achieve the smallest cost savings in the pyramid arrival pattern, it can be deduced that this arrival pattern is the worst case scenario for our approach. Hence, it will be interesting to investigate such a arrival pattern in more detail in our future work.

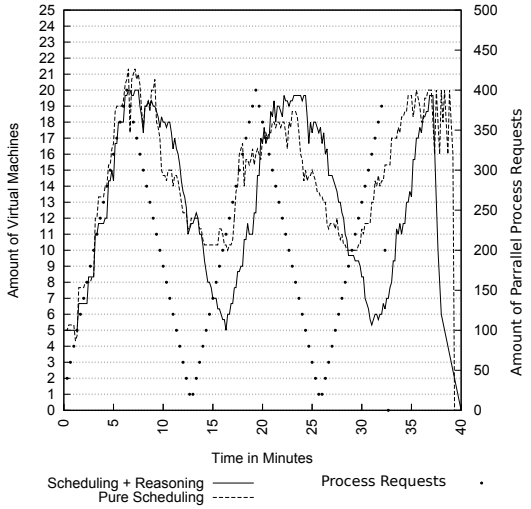
Last but not least, it should not be forgotten that the good results for the "Pure



(a) Constant Arrival Pattern



(b) Linear Arrival Pattern



(c) Pyramid Arrival Pattern

Figure 4.4: Evaluation Results

Scheduling" approach are due to the fact that our approach already takes into account the single deadlines for individual process steps. In general, both the "Pure Scheduling" and the "Scheduling + Reasoning" approaches led to a quite efficient resource allocation of cloud-based computational resources. Therefore, through the application of these algorithms, ViePEP is able to decrease the risk of unwanted spendings produced by over-provisioning. Furthermore, the system also decreases the risk of process instance execution errors by preventing under-provisioning of resources.

4.6 Conclusion

In this chapter, we presented scheduling and reasoning algorithms for sequential elastic processes, i.e., for the execution of business processes using cloud-based computational resources. The decision at what point of time a particular process instance should be enacted is based on client-defined non-functional requirements such as execution deadlines. Depending on a computed scheduling plan, the ViePEP Reasoner is able to deduce the resource demand and allocate resources accordingly. Furthermore, the Reasoner changes the scheduling plan based on the current resource situation. As we have shown in the evaluation, our "Pure Scheduling" approach already leads to good results in terms of cost and total execution time; the "Scheduling + Reasoning" approach leads to further cost and time savings.

Research on elastic process landscapes in the cloud, including dynamically leasing and releasing of computational resources, is just at the beginning and we are sure that more and more approaches addressing this topic will follow in the next years. In our own work, we take into account more complex process patterns – in this chapter, we focused on sequential processes only, however, loops, branches and further patterns are more realistic and closer to real-world scenarios. This will be considered in Chapter 6 and Chapter 7.

In the next chapter, i.e., Chapter 5 we will extend our work from this chapter and provide an algorithm aiming at saving cost, i.e., we present a cost-based optimization model for sequential elastic processes.

CHAPTER 5

Cost-based Optimization for Sequential Elastic Processes

We showed in the former chapter (Chapter 4) how cloud-based computational resources can be used for elastic process enactments in an efficient way. Within this chapter we extend that approach and aim at optimizing the used resources by providing an optimization model and according heuristic.

5.1 Overview

In Section 2.4 we presented ViePEP 1.0, which combines the functionalities of a BPMS with that of a cloud resource management system. We extended this BPMS and presented ViePEP 1.1 in the former chapter: A BPMS which is able to schedule complete processes as well as the involved single tasks, lease and release cloud-based computational resources in terms of VMs while taking into account SLAs defined by the process owners.

Within this chapter, we extend our former approach by addressing the problem of online cloud-based computational resource allocation for elastic processes using an optimization model and a heuristic. In our scenario, it is necessary to schedule service invocations and lease and release cloud-based computational resources in order to carry out single process steps under given SLAs. To encounter the complexity of this scenario, it is necessary to compute the resource demands of service invocations, develop a cost model, compute the cost, and perform a cost/performance analysis. This has to be done continuously, as new process requests arrive, software services representing single process steps do not behave as predicted, or a process instance is changed by the process owner. The goal is to provide cost-optimized process scheduling that takes into account the given SLAs and leases and releases cloud-based computational resources in order to optimize cost. Hence, the main contributions in this chapter are as follows:

- We formulate a model for scheduling and resource allocation for Elastic Processes.
- We design a heuristic based on the proposed model.
- We integrate this work into ViePEP 1.1.

The remainder of this chapter is organized as follows: In Section 5.2, we introduce some preliminaries. Afterwards, we present a scheduling and resource allocation solution for sequential elastic processes. For this, we define an integer linear optimization problem and a corresponding heuristic (Section 5.3.2 and Section 5.3.3). We evaluate the scheduling and resource allocation algorithms using ViePEP 1.1–based testbed experiments (Section 5.4). Last but not least we close this chapter with a summary in Section 5.5.

5.2 Preliminaries

As in the former chapter (see Section 4.1), we assume a business process landscape to be made up from a large number of business processes which need to be carried out automatically using cloud-based computational resources. For that, a BPMS is deployed (here ViePEP 1.1 see Section 4.2).

In this process landscape, clients can choose from a set of process models and request processes for execution, i.e., clients can instantiate the process models resulting in a particular process instances. Each process instance consists out of a set of process steps which are realized by invoking a corresponding (software) service. A service is of a specific service type and in order to be invokable, it needs to be deployed on a particular VM instance resulting in a service instance. Such processes may be requested at any time and may be carried out in regular intervals or ad hoc and nonrecurring. Users can define SLAs for process instances. Within such a SLA, different SLOs may be defined. In this chapter we consider solely the process deadline.

ViePEP, the deployed BPMS is employed to accept these process requests, schedule their executions, compute the resource demand and lease or release cloud-based computational resources according to the created scheduling plan. Summarizing, the core functionalities of ViePEP can be described as following:

- Provisioning of an interface to the cloud, allowing leasing and releasing cloud-based computational resources (VMs) on-demand
- Execution of process steps by instantiating services on VMs and invoking the service instances of process instances
- Dynamic scheduling of incoming requests for processes based on their QoS requirements, i.e., timeliness in terms of maximum execution time or a deadline
- Monitoring the deployed VMs in terms of resource utilization and monitor the QoS of service invocations

So far, the information which services need to be invoked at what point of time was generated by the Scheduler and the Reasoner using a deadline-based scheduling approach as presented in Section 4.3 and Section 4.4. These two components aimed at resource-efficiency only, i.e., the objective was to use less resource with the major limitation that only single-core VMs were leased.

However, cloud providers allow leasing different types of VMs, each comprising a different level of resource supply, e.g., a different amount of CPU cores or RAM. For example AWS provides several different VM types[1]. As the Reasoner has now more possibilities, the decision whether to scale up, down and in what dimension, gets more complicated. Hence, we extend the functionality of the Reasoner and provide an optimization model aiming at cost optimization.

Notably, the basic functionality of the Reasoner remains the same: it interacts with the Load Balancer in order to estimate which Backend VM provides how many free resources for particular service instances. In addition, the Reasoner is aware of the queued process requests and a Scheduler is employed to create a detailed scheduling plan. The functionality of this subcomponent is defined through the optimization approach and heuristics presented in Section 5.3.

Within the next chapter, we present the revised approach for scheduling processes and their single service invocations as an optimization model.

5.3 Solution Approach

We formulate the problem of scheduling process instances and their individual services, respectively, as an optimization problem. The general approach proposed in this chapter for achieving an optimized scheduling and resource allocation is presented in Section 5.3.1. A formal specification of the corresponding optimization problem is provided in Section 5.3.2. Finally, in Section 5.3.3, we describe a heuristic solution method for efficiently solving the optimization problem.

5.3.1 General Approach

For achieving the necessary process scheduling and resource allocation, the deadlines indicating the time when the corresponding process instances have to be finished are considered. In addition, in order to minimize the total cost of VMs leased for executing process requests and corresponding services, respectively, the leased VMs should be utilized as much as possible, i.e., leased but unused resource capacities should be minimized. Thus, in order to reduce the necessity of leasing additional VMs, we try to invoke services that cannot afford further delays first on already leased VMs before leasing additional VMs if the remaining resource capacities of already leased VMs are sufficient. Only if the resource requirements of processes cannot be covered by already leased VMs, additional VMs will be leased. For instance, if the deadlines for certain process instances allow delaying their service invocations to another period, it can be beneficial to release leased

[1] `http://aws.amazon.com/ec2/instance-types/`

resources and delay such service invocations. However, it has to be considered that those service invocations have to be scheduled for one of the subsequent optimization periods to ensure that corresponding deadlines are not violated.

In addition, since VMs are leased only for a certain time period and the execution of already scheduled invocations will be finished at a future point in time, the scheduling strategy to be developed cannot be static, i.e., the optimization approach for scheduling service invocations should not be applied only once. It rather has to be applied multiple times at different optimization points in time. In this respect, it has to be noted that potentially further requested process instances may have to be served, which additionally have to be considered when carrying out an optimization step.

Thus, an efficient scheduling strategy has to review allocated service instances and scheduled service invocations periodically and carry out further optimization steps for considering dynamically changing requirements and keeping the amount of leased but unused resource capacities low. Furthermore, the resource demand and average runtime for single service instances needs to be known in advance – for this, an approach based on OLS linear regression will be applied.

5.3.2 Optimization Problem

In this section, we model the scheduling and allocation of processes and services as an optimization problem. Since cloud-based computational resources are only leased for a limited time period, it has to be decided whether those resources or even additional resources have to be leased for another period, or whether leased resources can be released again. In this respect, we aim at utilizing leased cloud-based computational resources, i.e., VMs, to their full capacities. Further, additional resources will only be leased if capacities of already leased VMs are not sufficient to cover and carry out further service invocations that cannot be delayed.

For considering the different (optimization) time periods, the index t will be used. Depending on these periods, the parameter τ_t refers to the actual point in time, respectively, indicated by a time period t. For scheduling different processes, multiple process templates are considered. The set of process templates is labeled with P, where $p \in P = \{1, ..., p^{\#}\}$ refers to a certain process template. The set of process instances that have to be considered during a certain period t corresponding to a certain process template p is indicated by I_p, where $i_p \in I_p = \{1, ..., i_p^{\#}\}$ refers to a specific process instance.

The total number of process instances that have to be considered in period t is indicated by $i_p^{\#}$. Please note that considering a certain process instance i_p in period t does not necessarily result in invoking corresponding service instances in this period. It rather makes sure that it is considered for the optimization step conducted in period t, which may lead to the decision that this instance is further delayed – until another optimization step is carried out in a subsequent period. The remaining execution time for executing a certain prorcess instance i_p, which might involve invoking multiple service instances for accomplishing the different steps of a certain process instance, is indicated by the parameter e_{i_p}. Thus, by specifying the parameter e_{i_p} as the remaining execution

time of a certain process instance, we account for the fact that certain steps of this instance might have already been accomplished by having invoked corresponding service instances in previous periods.

It has to be noted that executing process instances refers to invoking software-based services representing a process step j_{i_p}. The *next* step of a process instance is labeled with $j^*_{i_p}$. For example, if a process instance consists of five steps, its remaining execution time e_{i_p} is determined by the sum of the execution times $e_{j_{i_p}}$ of the steps J_{i_p} which are required for accomplishing the different steps of the process instance, i.e., $e_{i_p} = \sum_{j_{i_p} \in J_{i_p}} e_{j_{i_p}}$

The set J_{i_p} thereby represents the set of process steps (including the services) that have to be invoked for accomplishing all steps of process instance i_p. Allocating and executing this instance refers to invoking the next service $j^*_{i_p}$ that accomplishes the next step, i.e., the first step in this example. Thus, four steps of this process instance still remain unaccomplished. After having the underlying service instance of $j^*_{i_p}$ invoked, the remaining execution time e_{i_p} for this process instance is reduced by the execution time $e^*_{j_{i_p}}$ of the invoked service. Thus, e_{i_p} can be determined by adding up the services' execution times for the remaining four steps of this process instance. For a more detailed description of how the execution time is calculated, refer to Section 4.3. The corresponding process instance needs to be executed again, i.e., a service instance accomplishing the second step has to be invoked. For accomplishing this second step, the corresponding service becomes the next service $j^*_{i_p}$.

The deadline at which the execution of a process instance has to be finished is indicated by the parameter d_{i_p}. Assuming a continuous flow of time, d_{i_p} refers to an actual point in time as, for instance, "27.09.2013, 13:37:00". For executing a certain process instance i_p, i.e., for invoking the service instance of the next step $j^*_{i_p}$, of a process instance i_p, a certain amount of computational resources from a VM are required. The resource requirement for the corresponding next services is indicated by r_{i_p}.

Regarding the leasing of cloud-based computational resources, we assume different types v of VMs. The set of VM types is indicated by the parameter V, where $v \in V = \{1, ..., v^\#\}$ refers to VM type v. The corresponding resource supply of a VM (in terms of CPU, RAM, bandwidth) of type v is indicated by the parameter s_v. For counting and indexing leased VM instances of type v, the variable k_v is used. Although in theory unlimited, we assume the number of leasable VM instances of type v in a time period t to be restricted by $k^\#_v$ for modeling reasons, i.e., in order to make the idea of indefinite resources, provided by the cloud, tangible. Thus, we assume a maximum number $k^\#_v$ of leasable VMs. The set of leasable VM instances of type v is indicated by K_v, where $k_v \in K_v = \{1, ..., k^\#_v\}$. The cost for leasing one VM instance of type v is indicated by c_v.

For finally deciding at period t which service to instantiate and invoke, binary decision variables $x_{(i_p,k_v,t)} \in \{0, 1\}$ are used. A value $x_{(i_p,k_v,t)} = 1$ indicates that the next service for step $j^*_{i_p}$ of process instance i_p should be allocated and invoked in period t at VM k_v, whereas a value $x_{(i_p,k_v,t)} = 0$ indicates that the invocation of the corresponding service should be delayed, i.e., no service of process instance i_p should be invoked in period t. For indicating, whether a certain VM instance k_v of type v should be leased in period t,

another decision variable $y_{(k_v,t)} \in \{0,1\}$ is used. Similar to $x_{(i_p,k_v,t)}$, a value $y_{(k_v,t)} = 1$ indicates that instance k_v of VM type v is leased. The total number of VMs of type v to lease in period t is labeled with $\gamma_{(v,t)}$.

Using these parameters and variables, we formulate the optimization problem in (5.1) for deciding which process instances i_p and corresponding services for the step j_{i_p}, respectively, should be allocated and invoked in period t.

$$\min \sum_{v \in V} c_v \cdot \gamma_{(v,t)} + \sum_{v \in V} \sum_{k_v \in K_v} y_{(k_v,t)} \cdot f_{(k_v)} \tag{5.1}$$

$$\tau_{(t+1)} + e_{i_p} - e^*_{j_{i_p}} \cdot x_{(i_p,k_v,t)} \leq d_{i_p} \forall p \in P, i_p \in I_p \tag{5.2}$$

$$\tau_{(t+1)} \leq \tau_t + e^*_{j_{i_p}} \cdot x_{(i_p,k_v,t)} + (1 - x_{(i_p,k_v,t)}) \cdot M \ \forall p \in P, i_p \in I_p \tag{5.3}$$

$$\tau_{(t+1)} \geq \tau_t + \epsilon \tag{5.4}$$

$$\sum_{p \in P} \sum_{i_p \in I_p} r_{i_p} \cdot x_{(i_p,k_v,t)} \leq f_{(k_v)} \ \forall v \in V, k_v \in K_v \tag{5.5}$$

$$\sum_{k_v \in K_v} y_{(k_v,t)} \leq \gamma_{(v,t)} \forall v \in V \tag{5.6}$$

$$y_{(k_v,t)} \cdot s_{(k_v)} - \sum_{p \in P} \sum_{i_p \in I_p} r_i \cdot x_{(i_p,k_v,t)} \leq f_{(k_v)} \forall v \in V, k_v \in K_v \tag{5.7}$$

$$x_{(i_p,k_v,t)} = 1 \ \forall p \in P, i_p \in I_p, v \in V, k_v \in K_v \mid \text{if } i_p \text{ runs} \tag{5.8}$$

The constraints in (5.2) make sure that the deadlines d_{i_p} for process instances i_p will not be violated. For this, the sum of the remaining execution time e_{i_p} and the next optimization point in time $\tau_{(t+1)}$ has to be lower or equal to the deadline. By scheduling a service invocation, the corresponding remaining execution time e_{i_p} is reduced, because the execution time for the next service $e_{j^*_{i_p}}$ will be subtracted in this case.

The constraints in (5.3) and (5.4) determine the next optimization point $\tau_{(t+1)}$. In order to avoid optimization deadlocks, which would result from not advancing the next optimization time $\tau_{(t+1)}$, $\tau_{(t+1)}$ is restricted in (5.4) to be greater or equal to τ_t plus a small value $\epsilon > 0$. In order to replan the scheduling and the execution of process instances as soon as a service invocation has been finished, $\tau_{(t+1)}$ should be lower or equal to $\tau_{(t+1)}$ plus the minimum execution time of the services invoked in period t. Using an additional parameter $M \geq \max(e_{j_{i_p}})$, the last term in (5.3) thereby makes sure that services that are not invoked in this period do not restrict $\tau_{(t+1)}$, to be lower or equal to τ_t. However, a replanning will anyhow be triggered if events such as the request of further process instances or the accomplishment of a certain invoked service occur.

The constraints in (5.5) make sure that the resource capacities required by the next services for process instance i_p assigned to instance k_v of VM type v are lower or equal to the capacities these VM instances can offer. If service instances are assigned to a certain VM instance k_v of type v, the corresponding decision variable $y_{(k_v,t)}$ will assume a value of 1. The sum of all decision variables $y_{(k_v,t)}$ determines the total number $\gamma_{(v,t)}$ of instances for VMs of type v, as indicated in (5.6). In (5.7), the amount of unused capacities for VM instance k_v, which is indicated by the variable $f_{(k_v)}$, is determined. In order to account for running services invoked in previous periods, corresponding decision variables $x_{(i_p,k_v,t)}$ are set to 1, which is indicated in (5.8).

The objective function, which is specified in (5.1), aims at minimizing the total cost for leasing VMs. In addition, by adding the amount of unused capacities $f_{(k_v)}$ of leased VMs to the total cost, the objective function also aims at minimizing unused capacities of leased VM instances.

5.3.3 Heuristic Approach

For efficiently solving the optimization problem presented in the last subsection, we develop a heuristic solution method. This heuristic basically examines which process instances i_p and services representing the step j_{i_p}, respectively, have to be scheduled and invoked in the current optimization period t in order to avoid violating corresponding deadlines d_{i_p}.

For this, the heuristic initially determines the point in time $\tau_{(t+1)}$, when the next optimization step has to be carried out – at the latest. Corresponding to this (latest) subsequent optimization time, a virtual *time buffer* is calculated for each process instance, which will be referred to as slack, indicating the time the corresponding process instance i_p may be delayed at maximum, before a violation of the corresponding deadline d_w takes place. Those process instances, for which the slack is lower than 0, i.e., those process instances, for which a further delay would result in violating a process deadline, are considered as critical. Thus, at first, we try to invoke the critical services instances on such VM instances that are already leased and running. Afterwards, we lease new VM instances such that all remaining critical service instances are allocated and invoked in the current period. Finally, in order to minimize unused resources of leased VM instances, we invoke service instances on the leased VM instances corresponding to their slack.

This heuristic solution method is provided in Algorithm 2 using pseudocode. Corresponding methods used in Algorithm 2 are indicated in Algorithm 3, Algorithm 4 and Algorithm 5. In lines 1-13 of Algorithm 2 the required parameters and variables are initialized. For instance, the deadlines d_{i_p} for process instances i_p of process template p are initialized in line 2. In this respect, it has to be noted that the corresponding parameter d[p,i] represents an array containing all deadlines d_{i_p} of process instances i_p that are considered at optimization period t. Analogously, e[p,i] represents an array for the remaining execution times of process instances i_p, which is initialized in line 3. The resource supply s_v for a VM of type v is initialized in line 4, whereas the number of instances k_v of leased VMs of type v is initialized in line 5. Note that VM instances

Algorithm 2 Heuristic Solution Approach

```
1: procedure MAIN(String[] args)
2:     d[p,i];                                   ▷ //Deadline for instance i of process p
3:     e[p,i];                                   ▷ //Remaining execution time for instance i of process p
4:     s[v];                                     ▷ //Resource supply for VM of type v
5:     k[v];                                     ▷ //Number of already leased VMs of type v
6:     leasedVM[v,k];                            ▷ //k^th VM instance of type v
7:     unusedRes[v,k];                           ▷ //Unused resources for k^th VM of type v
8:     sl[p,i] ← new Double[p#,i#];              ▷ //Array for slack
9:     r_crit ← 0;                               ▷ //Aggregated resource demand for critical instances
10:    vmTypeList;                               ▷ //List of available VM types, sorted by size ascending
11:    sortList ← new List();                    ▷ //Sorted List corresponding to slack
12:    critList ← new List();                    ▷ //List containing critical instances
13:    τ_(t+1) ← d[1,1];                         ▷ //Initialize next optimization point in time
14:                                              ▷ //Compute τ_(t+1)
15:    for p = 1 to p# do
16:        for i = 1 to i# do
17:            if d[p,i]-e[p,i]-τ_(t+1) then
18:                τ_(t+1)=d[p,i]-e[p,i];        ▷ //Get minimum τ_(t+1)
19:            end if
20:            if τ_(t+1) ≤ τ_t then
21:                τ_(t+1)=τ_t+ϵ;                ▷ //Avoid deadlocks
22:            end if
23:        end for
24:    end for
25:                                              ▷ //Compute slack sl[p,i]
26:    for p = 1 to p# do
27:        for i = 1 to i# do
28:            sl[p,i]=d[p,i]-e[p,i]-τ_(t+1);    ▷ //Get slack
29:            if s[p,i] < 0 then
30:                critList.add(getInst(p,i))
31:                r_crit=r_crit+getInst(p,i).resNextService()
32:            else
33:                sortList.insert(getInst(p,i),sl[p,i]);
34:            end if
35:        end for
36:    end for                                   ▷ //Invoke critical instances on leased but unused resources
37:    usedRes=placeOnUnusedRes(critList)
38:    r_crit= r_crit-usedRes
39:                                              ▷ //lease new VMs until r_crit is satisfied
40:    leaseNewVMs(r_crit)
41:                                              ▷ //Place further non-critical instances on unused resources
42:    placeOnUnusedRes(sortList)
43: end procedure
```

Algorithm 3 Method PlaceOnUsedRes(List)

```
1:  procedure placeOnUsedRes(List)
2:                                                    ▷ //Variable Initialization
3:      usedRes ← 0;
4:      removeList ← new List();
5:      for iter=1 to list.size() do
6:          inst = list.get(iter);
7:          r = inst.resNextService();
8:          placed = false;
9:          for v=1 to v# do
10:             if !placed then
11:                 for k=k[v]; k≥1; k=k-1 do
12:                     if (r≤unusedRes[v,k]) then
13:                         placeInst(inst,leasedVM[v,k]);
14:                         placed = true;
15:                         unusedRes[v,k]=unused[v,k]-r;
16:                         usedRes = unused[v,k]-r;
17:                         removeList.add(inst);
18:                         break;
19:                     end if
20:                 end for
21:             end if
22:         end for
23:     end for
24:     list.remove(removeList);
25:     return usedRes;
26: end procedure
```

potentially have been leased in previous periods. Thus, the number of instances k_v is not necessarily 0 when carrying out an optimization step.

Correspondingly, leased VMs as well as unused resources of already leased VMs, which are indicated by the arrays leasedVM[v,k] and unusedRes[v,k], have to be considered (cf. lines 6-7). For computing the slack of all process instances and aggregating the resource demands of the critical instances, the array sl[p,i] and the variable r_{crit} is used (cf. lines 8-9). Since in this heuristic, different sizes of VMs are considered, i.e., they differ in the amount of available resources, a list of the available VM types is stored in vmTypeList in line 10. This list is sorted in ascending order by the VMs' sizes, i.e., the smallest VM comes first. In lines 11-12, empty lists are created for storing the critical instances as well as the non-critical instances, which are sorted in the list sortList corresponding to their slack. Finally, the point in time $\tau_{(t+1)}$, where the next optimization step has to be carried out at latest, is initialized with an arbitrary deadline, as, e.g., $d_{(1_1)}$.

Having initialized required parameters and variables, $\tau_{(t+1)}$ is determined in lines

Algorithm 4 Method PlaceInst(List,VM)

```
1: procedure placeInst(List, VM)
2:                                                    ▷ //Variable Initialization
3:     v = VM.getType();
4:     removeList = new List();
5:     unusedRes = supply[v];
6:     for iter=1 to list.size() do
7:         inst = list.get(iter);
8:         r = inst.resNextService();
9:         if unused≥r then
10:            placeInst(inst,leasedVM[v,k]);
11:            unusedRes=unusedRes-r;
12:            removeList.add(inst);
13:        end if
14:    end for
15:    list.remove(removeList);
16:    return supply[v]-unusedRes;
17: end procedure
```

15-24 by computing the minimum difference between deadlines d_{i_p} and corresponding remaining execution times e_{i_p} for all process instances. For this point in time, the difference between remaining execution time and deadline, i.e., the slack, will be 0 for at least one process instance. In order to avoid optimization deadlocks that will result if the subsequent optimization time $\tau_{(t+1)}$ is equal to the current time $\tau_{(t+1)}$, a small value $\epsilon > 0$ is added (cf. lines 20-21).

In lines 26-36, the slack for each process instance is computed (cf. line 28) and the corresponding process instances are either added to the list of critical instances (cf. line 30) or inserted into a sorted list of (non-critical) instances (cf. line 33) corresponding to their slack. In addition, the resource requirements for the next services of the critical process instances are aggregated (cf. line 31). The corresponding critical service instances need to be allocated and invoked in the current period – either on already leased and running VM instances or on further VM instances that have to be additionally leased in this period. Invoking critical service instances on already leased VM instances is accounted for in line 37 by calling the method *placeOnUnusedRes*, which is provided in Algorithm 3. Within Algorithm 3, the method *placeInst* is called, which is provided in Algorithm 4.

In line 38 (of Algorithm 2), the resource requirements of the successfully invoked critical process instances are subtracted from the aggregated resource demand of the remaining critical service instances. In line 40, additionally required resources are acquired. For this, the method *leaseNewVMs* is called. Its pseudocode is presented in Algorithm 5.

The types of the new VMs are chosen according to the following procedure: In general we successively try to acquire VMs with rather high resource supply aiming at reducing

Algorithm 5 Method LeaseNewVMs(Res)

```
1: procedure LEASENEWVMS(Res)
2:                                                    ▷ //Variable Initialization
3:     vm;                                            ▷ //Temp variable for new VM
4:     while r_crit >0 do
5:         for iter=1 to vmTypeList.size() do
6:             v ← vmTypeList.get(iter);
7:             if supply[v]≥r_crit OR iter == vmTypeList.size() then
8:                                                    ▷ //start a new VM of type v
9:                 vm = leaseVM(v);
10:                k[v] = k[v]+1;
11:                leasedVM[v,k[v]-1] = vm;
12:                unusedRes[vm,k[v]-1] = supply[v];
13:                usedRes = placeInst(critList,vm, k[v]-1);
14:                r_crit = r_crit-usedRes;
15:                break;
16:            end if
17:        end for
18:    end while
19: end procedure
```

unused resource capacities due to having a large number of rather small-sized VMs. In addition, the cost of VMs in our scenario is proportional (see Section 5.2), i.e., larger VMs will provide a proportionately lower basic load that is not available to service instances. Therefore, as long as more resources are required, i.e., $r_{crit} > 0$ (cf. line 4 in Algorithm 5), a new VM having the least amount of provided resources, but still bigger or equal to the required resource demand r_{crit} (cf. line 7), will be leased. If no VM type fulfills this requirement, the biggest available VM type will be leased (cf. line 9, i.e., if the end of *vmTypeList* is reached). Subsequent to that, a new VM having this type will be leased (cf. line 9). If a new VM is leased, the resource demand r_{crit} will be reduced by the amount of actually used resources, i.e., after having placed critical instances on this VM (cf. line 13-14). Before this, we update in lines 9-12 the number of leased VM instances of type v, i.e., k_v, store the corresponding VM instance, and set the value of its unused resource supply to the (maximum) supply of a VM of type v. Using the method *placeInst*, which is provided in Algorithm 5, we allocate critical service instances on the newly leased VM such that no further critical service instances can be placed on it due to its limited resource capacity. Subsequently, a "break" statement will stop the for loop and the procedure will be repeated until enough resources are available.

Corresponding to the presented algorithm for leasing additional resources (Algorithm 5), it is possible that resources are still available on the leased VMs, i.e., they are not fully utilized. Therefore, we invoke further scheduled service instances on (already) leased VM instances in order to reduce the amount of unused VM resources. This is realized

in line 42 of Algorithm 2 by calling the method *placeOnUnusedRes* (cf. Algorithm 3) another time.

Based on the results of the algorithms, the Reasoner is able to lease/release resources based on the calculated resource demand. In addition, the Process Manager gets the information at which point in time to invoke which service instance as part of which process instance.

5.4 Evaluation

In the following we present our general evaluation approach (Section 5.4.1), i.e., the evaluation scenario including the evaluation criteria. The experiment's results are discussed in Section 5.4.2. As in the former chapter, we use ViePEP 1.1 (Section 4.2) for our evaluation. Notably, although the evaluation scenarios are similar to the former evaluation, we are not able to compare the results directly with each other due to some signifficant differences, e.g., in the former evaluation only single-core VMs were used and resource cost (VM leasing cost or VM-Minutes) are calculated differently.

5.4.1 General Evaluation Approach

While ViePEP and the presented reasoning approach are applicable in arbitrary process landscapes and industries, we evaluate the heuristic using a real-world data analysis process from the finance industry. Choosing this particular process does not restrict the portability of our approach to other domains. Again, we apply a testbed-driven evaluation approach, i.e., real cloud-based computational resources are deployed. For the single services, we simulate differing workloads regarding CPU and RAM utilization and service runtime as discussed below. However, real services are deployed and invoked during process executions.

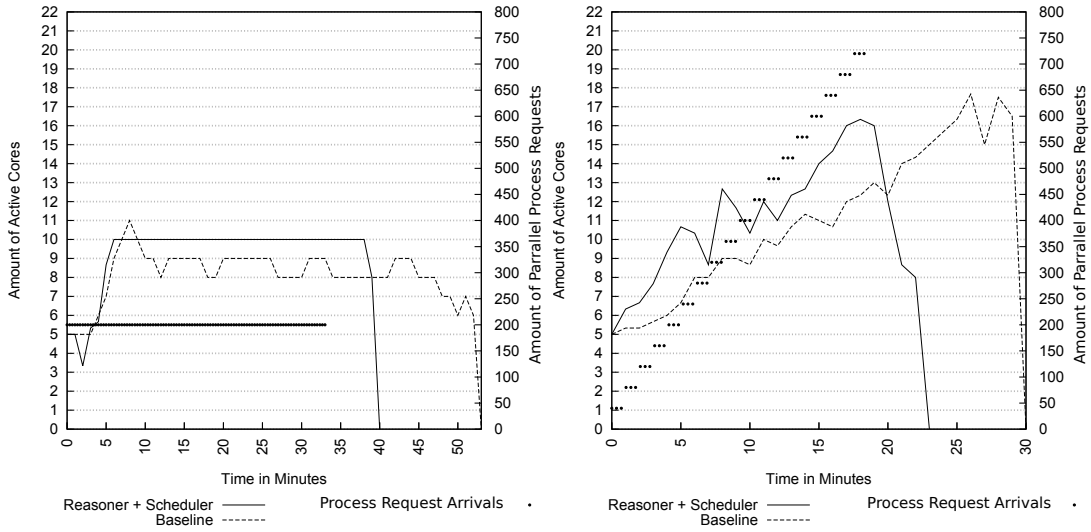
As in our former evaluations (Section 4.5), we decided to make use of one single process model which will be processed 20,000 times. This sequential process consists out of five individual service steps: The light-weight *Dataloader Service* simulates the loading of data from an arbitrary source; afterwards, the more resource-intensive *Pre-Processing Service* is invoked; next, the *Calculation Service* simulates data processing, which leads to high CPU load; then, the *Reporting Service* generates a simple report – it generates a load similar to the one of the *Pre-Processing Service*. Last, the *Mailing Service* sends the report to different recipients – this is a lightweight service comparable to the *Dataloader Service*. The user-defined maximum execution time for process instances has been set to 5 minutes (from the time of the request).

In order to test our optimization approach against a baseline, we have implemented a basic ad hoc approach. This approach is only able to take into account currently incoming process requests in an ad hoc way. While this includes the scheduling of process requests, the baseline approach does not take into account future resource demands. Instead, whenever a Backend VM is utilized more than 80%, it leases an additional Backend VM for the according service. When the utilization is below 20%, the VM is released again.

Notably, the baseline approach will only lease single-core VMs, as it is not able to take into account future resource demands.

Process Request Arrival Patterns

Similar to our former evaluation in Section 4.5, we make use of two distinct process request arrival patterns: In the *constant arrival* pattern, the process requests arrive in a constant manner. This means, the same amount of processes arrives in a regular interval. In our evaluation, the number of simultaneously executed processes is set to 200 and is send to ViePEP every 20 seconds. In the *linear arrival* pattern, the processes are executed following a linear rising function, i.e., $y = k \cdot \frac{x}{3} + 40$ where y is the amount of concurrent requests and 40 the start value. This value is increased by $k = 40$ in an interval of every 60 seconds.



(a) Constant Arrival – Results

(b) Linear Arrival – Results

Figure 5.1: Evaluation Results

Metrics

In order to get reliable numbers, we executed each arrival pattern three times and evaluated the results against three quantitative metrics. First, we measure the total makespan in minutes which is needed to process all 20,000 process instances (*Total Makespan in Minutes*). This is the timespan from the arrival of the first process request until the last step of the last process was processed successfully. The second metric is the amount of the concurrently leased number of cores, i.e., the sum of (CPU) cores of the leased VMs (*Max. Active Cores*). The combination of the first two metrics, results in *Cost in Core-Minutes*, i.e., these tell us the resulting cost of the overall evaluation. The Core-Minutes are calculated following a similar pricing schema as Amazons EC2, i.e., the VMs cost increase proportionally with the number of provided resources. Our evaluation environment, i.e., the private cloud we are running ViePEP in, provides four different VM types, a single-core VM, a dual-core VM, a triple-core and a quad-core VM. In order to get the resulting cost we sum up the active cores over time and get the overall Core-Minutes.

5.4.2 Results and Discussion

Table 5.1-5.2 and Figure 5.1a-5.1b present our evaluation results in terms of the average numbers from the conducted evaluation runs. Table 5.1-5.2 present the observed metrics as discussed in the last section for both arrival patterns. For each arrival pattern, the numbers for evaluation runs are given for the baseline scenario as well as the deployed optimization approach. The tables also state the standard deviation (σ) from the average values. In general, the observed standard deviation is low, and therefore indicates a low

Table 5.1: Evaluation Results – Constant Arrival

	Constant Arrival	
	Pure Scheduling	Reasoner + Scheduler
Number of Process Requests	20,000	
Number of Requests in one Burst	$y = 200$	
Total Makespan in Minutes (Standard Deviation)	52 (σ=2.16)	39 (σ=0.81)
Max. Active Cores (Standard Deviation)	11 (σ=0)	10 (σ=0)
Cost in Core-Minutes (Standard Deviation)	443.67 (σ=7.72)	370.33 (σ=5.90)

Table 5.2: Evaluation Results – Linear Arrival

	Linear Arrival	
	Pure Scheduling	Reasoner + Scheduler
Number of Process Requests	20,000	
Number of Requests in one Burst	$y = 40 \cdot \lfloor \frac{x}{3} \rfloor + 40$	
Total Makespan in Minutes (Standard Deviation)	28.67 (σ= 1.25)	22 (σ= 0.82)
Max. Active Cores (Standard Deviation)	18 (σ= 0)	16.66 (σ= 0.47)
Cost in Core-Minutes (Standard Deviation)	314.71 (σ=25.36)	243.59 (σ=6.70)

dispersion in the results of the different evaluation runs. Figure 5.1a and Figure 5.1b complete the presentation of the average evaluation results by depicting the arrival patterns ("Process Request Arrivals") over time and the number of active cores until all process requests have been served. Again, evaluation results with deployed optimization and applying the baseline approach are shown.

The numbers in Table 5.1 and Table 5.2 indicate a substantial performance difference between the baseline and the optimization approach. Most importantly, the cost in terms of Core-Minutes is lower in both cases, leading to 16.53% cost savings for the constant arrival patter and 22.60% for the linear arrival pattern. Hence, we can derive that the optimization approach helps to achieve a significantly better utilization of VMs, thus preventing additional cost arising from over-provisioning of cloud-based computational

resources. Also, the optimization approach is faster in absolute numbers, as it needs 25.00% less time to execute all process requests following the constant arrival pattern and 23.26% in the linear arrival pattern.

For both arrival patterns, the baseline approach is in many cases not able to comply with the process deadlines (5 minutes), as can be seen from the backlog after all process requests have arrived (see Figure 5.1). This can be traced back to the applied ad hoc approach, i.e., it takes the applied ad hoc scaling approach in the baseline scenario too long to react to new process requests and adjust the number of leased VMs accordingly.

Interestingly, for the constant arrival pattern, Figure 5.1a shows clearly that ViePEP was able to optimize the system's landscape almost perfectly, i.e., the number of active VMs vary only a few times during the experiment. It can be perfectly seen that the optimization approach (i.e., "Reasoner + Scheduler") is not only faster than the baseline, but also acquired overall less computational resources, i.e., VMs.

For the linear arrival pattern, the number of active cores increases in about the same for the optimization and the baseline. However, the biggest difference is that in the "Reasoner + Scheduler" approach, VMs with more than one core are acquired while in the baseline approach only single-core VMs are acquired. This results in a slower processing of the whole process queue since the overhead of the OS is comparable higher in a single-core VM than in a quad-core VM. Single-core VMs are therefore less able to handle a high load.

To summarize, the evaluation results show that the proposed optimization approach indeed leads to a more efficient allocation of cloud-based computational resources to single steps. While in our former evaluations (see Section 4.5.4) we achieved an average saving of 3.79%, using an optimization approach as presented in this chapter, we were able to achieve an average saving of 19.56% in terms of cost. Complementary to that, in terms of time, in our former evaluation we were 6.71% (in average) minutes faster and following the new approach we are now 24.14% faster than the baseline.

As a result, we can say that ViePEP 1.1 is able to provide a higher cost-efficiency than approaches which do not take the process perspective into account – in our evaluation, such approaches where represented by the baseline.

5.5 Conclusion

In this chapter, we:

- Presented an optimization model for resources allocation and scheduling for sequential elastic processes.
- Presented a heuristic based on the proposed optimization model.
- Implemented the heuristic in ViePEP 1.1.
- Evaluated the heuristic and compared it against a baseline which uses ad hoc-based scaling.

- Showed that using this heurist we were able to significantly save cost (19.56%) and execution time (24.14%).

In the following sections we will extend the basic model of our process scheduling and resource allocation approach by allowing more complex VM models (e.g., non-proportional cost for VMs), minimum lease periods for VMs from a cloud provider (Chapter 6), include data transfer cost when scheduling process steps in a hybrid cloud environment (Chapter 7), and take into account more complex process patterns such as XOR-block, AND-blocks and Repeat loops. While ViePEP 1.0 and ViePEP 1.1 were conceptualized for a usage in a private cloud only, in Chapter 7, ViePEP is extend by the possibility to combine public and private cloud-based computational resources resulting in ViePEP 2.0.

CHAPTER 6

Cost-based Optimization for Complex Elastic Processes

In Chapter 4 we presented how cloud-based computational resources can be used for sequential elastic process executions and in Chapter 5 we showed how leasing of such resources can be optimized in a cost-efficient way. So far, we considered sequential processes which represent only a small subset of real-world business processes, i.e., more complex processes are very common in real-world business processes [108]. Hence, in this chapter we extend the former approach by considering complex process patterns like XOR-blocks, AND-blocks and Repeat loops (including their according joins). In addition, we consider two additional aspects, first, penalty cost which accrue if SLAs get violated, and second, the Billing Time Unit (BTU) which expresses the cost per leasing period.

6.1 Overview

So far, we provided solely resource scheduling and optimization approaches for sequential elastic processes (see Chapter 4 and Chapter 5). Nonetheless, more complex process patterns like XOR-blocks, AND-blocks or Repeat loop constructs have not been covered yet, even though these process patterns are very common in real-world business processes [108]. Also, despite being an important cost factor, penalty cost as well as the BTU, which expresses the cost per leasing period, have not been regarded so far.

Hence, in this chapter, we substantially extend our former work on elastic process scheduling and resource allocation by the following:

- We define a system model for elastic process landscapes, taking into account complex process patterns, penalty cost, and BTUs.
- We present the Service Instance Placement Problem (SIPP), an optimization model which aims at minimizing the total cost arising from enacting an elastic process

landscape. Solutions to the SIPP describe execution plans in terms of process scheduling and resource allocation. This includes the assignment of service instances to VMs, the scheduling of service invocations, and the leasing and releasing of VMs.

- We implement the SIPP and create a new version of ViePEP resulting in ViePEP 2.0.
- We evaluate our approach extensively, showing its advantages compared to existing ad hoc resource allocation and process scheduling strategies.

The remainder of this chapter is organized as following: Section 6.2 defines some preliminaries for the approach presented in this chapter. Afterwards, Section 6.3 presents the SIPP optimization problem. The results of the evaluation for the scheduling and resource allocation algorithm through testbed experiments are described in Section 6.4. Last but not least Section 6.5 concludes this chapter.

6.2 Preliminaries

We hereby define further preliminaries similar to Section 4.1. Although the basics remain the same, we need to define some additional preliminaries before modeling the multi-objective optimization model (SIPP):

As prior, we deploy ViePEP in the cloud which serves on the one hand as a BPMS and on the other hand as a cloud controller in an intra or inter-organizational process landscape. Process owners may define *process models* and request their enactment, resulting in single *process instances*. Process models are composed of *process steps*. To execute a process step, a *service* is deployed on a particular VM resulting in a certain *service instance* and will then be *invoked* (*service invocation*). In the process landscape, we distinguish between several different services, i.e., each service instance is of a particular *service type*. The *execution time* of a *service invocation*, i.e., the timespan from starting the invocation until it is finished, may take from a few seconds to several minutes. For practical reasons, a VM may only run a single service instance.

However, this service instance may be invoked several times simultaneously and a service (of a specific type) may be deployed several times on different VMs. Along with the process request, process owners define deadlines, which are part of a SLA and therefore an important constraint for the SIPP. Penalties accrue if a certain process instance does not meet its SLA [69]. It is the general goal to minimize the cost of process enactments, taking into account VM leasing cost and penalty cost.

There might be several process owners in the process landscape, requesting different process instances at different points of time. As a result, the process landscape is volatile and ever-changing, which needs to be taken into account when solving the SIPP. Process landscapes could become very large, since the cloud offers theoretically unlimited resources [59].

During scheduling, the BPMS needs to take into account process requests, running process instances, leased VMs, service instances, the current and planned workloads of

the leased VMs, as well as the execution times and workloads of different service types on different VMs. Importantly, the BPMS is aware of *future* service invocations, since it knows the next steps of requested process instances.

In contrast to our former work we do not only consider simple, sequential processes, but also address more complex process patterns. Referring to [108], our scheduling approach accounts for structured processes comprising parallel invocations, i.e., *AND-blocks* (AND-splits with corresponding joins), as well as exclusive invocations, i.e., *XOR-blocks* (XOR-splits with corresponding joins). Also *Repeat loops* that allow a repeated execution of sub-processes are covered. Considering these complex process patterns aggravates the scheduling problem substantially, since the *next* step is not always straight forward in XOR-blocks or Repeat loops. For these, and for AND-blocks, several different possibilities have to be considered during resource allocation and scheduling. Applying the recursive pattern interlacing approach as presented in [95], the scheduling approach proposed in this chapter is also capable of considering interlaced structures. Depending on the openness to risk, a worst-, best- or average-case has to be performed prior scheduling, i.e., taking the longest, the shortest or the average path through a process structure. In our approach, we consider a worst-case scenario.

6.3 Complex Process Scheduling

For the enactment of complex elastic processes, cloud-based computational resources in terms of VMs are used. In this respect, we aim at achieving an optimal scheduling and placing of service instances, realizing corresponding process instances. For this reason, leasing and releasing of cloud-based computational resources has to be realized so that the cost for leasing aforementioned cloud-based computational resources is minimized. Furthermore, we need to make sure that given constraints on QoS attributes such as deadlines for the process instances are satisfied. In the work at hand, we exclusively focus on execution time as QoS attribute. Other QoS attributes from the field of service composition, e.g., availability, reliability or throughput [100], are not explicitly covered. It should be noted that availability and reliability are partially regarded in our optimization model, since process requests are carried out as long as there are computational resources from a cloud provider available. Since clouds offer virtually unlimited resources, the limiting factor for throughput is the missing capability of the BPMS to handle an unlimited number of processes at the same time. Scalability of the BPMS is however a research topic on its own.

If a violation of QoS constraints and therefore a SLA breach takes place, penalties accrue. Thus, the closer the deadline for a certain process instance is, the higher is the importance for scheduling and invoking corresponding service instances to enact the particular process instance in time. If not carefully considered and scheduled, the BPMS provider will either have to lease and pay for additional VMs to host service invocations that cannot be delayed any further, or to pay the aforementioned penalties. To avoid such situations where extra resources have to be leased or penalties have to be paid due to an inefficient scheduling strategy, the scheduling of service invocations along with

the leasing and releasing of cloud resources has to be optimized. Hence, we formulate the problem of scheduling and placing service instances on VMs for realizing process instances – the SIPP – as a Mixed Integer Linear Programming (MILP) optimization problem.

The applied system model is described in the subsequent Section 6.3.1. Afterwards, a formal specification of the corresponding optimization model is provided in Section 6.3.2, and finally, the model is extended in Section 6.3.3 to enable multi-period scheduling.

6.3.1 System Model

Within this section, we provide the system model needed for process scheduling and resource allocation as an optimization model. It has to be noted that a similar model is already presented in Section 5.3.2. However, while in the former chapter we provided an optimization model for sequential elastic processes, within the current chapter, we provide an optimization model for complex elastic processes. In addition, beside of considering complex process patterns (AND-blocks, XOR-blocks, and Repeat loops), we also consider the BTU, i.e., the cost per leasing period, different types of resources, e.g., CPU and RAM, penalty cost and others. Hence, we present in the following a coherent description of the needed system mode.

It has to be noted that the optimization problem described in this chapter per se refers to a certain time period. For considering different time periods, we use the parameter t as index which indicates the start of a period[1]. The concrete *time period* is indicated by the parameter τ_t and is given by a concrete *point in time* (e.g., "27.09.2013, 13:37:00" CET) that corresponds to the beginning of that (optimization) time period. In order to account for different types of process instances, we consider multiple process models. The set of process models is labeled with P, where $p \in P = \{1, ..., p^{\#}\}$ indicates a certain process model. The set of process instances that have to be considered during a certain period starting at t according to a certain process model p is indicated by I_p, where $i_p \in I_p = \{1, ..., i_p^{\#}\}$ refers to a certain process instance. In this respect, it has to be noted that *considering* a certain process instance i_p in the period starting at t does not necessarily result in invoking corresponding service instances in this period. It rather ensures that respective service invocations are acknowledged as potential candidates for scheduling. Thus, they *may* be scheduled in the time period starting at t or not.

Service instances that have to be invoked for accomplishing a process instance i_p are covered in the set J_{i_p}, where $j_{i_p} \in J_{i_p} = \{1, ..., j_{i_p}^{\#}\}$ refers to a certain service invocation for accomplishing a specific step in process instance i_p. Service invocations that have already been scheduled in previous optimization periods are referred to as ${j_{i_p}}^{run}$. They are covered in the set ${J_{i_p}}^{run}$. Since the steps within a certain process instance have to be invoked in a certain order, we may only schedule service invocations for the *next* step(s) in the optimization period starting at t. We label the corresponding service invocations accomplishing the *next* step(s) with $j_{i_p}^{*} \in J_{i_p}$. The *execution time* of service invocation j_{i_p}, i.e., the duration of the service invocation, is indicated by $e_{j_{i_p}}$.

[1]For a concise overview of the parameters used in this chapter, see Table 6.2

In order to schedule j_{i_p}, the corresponding service type has to be instantiated on a VM. In this chapter, we account for different VM types. The set of VM types is indicated by the parameter V, where $v \in V = \{1, ..., v^{\#}\}$ refers to VM type v. The corresponding resource supply of a VM of type v in terms of processing units (CPU) and main memory (RAM) is indicated by s_v^C and s_v^R. The unit of CPU resources is *percent*, i.e., a single-core VM has 100% CPU, a dual-core VM has 200% CPU, and so on, and the unit for RAM is *Mega Bytes* (MB). Analogously, the resource demand of a certain service invocation j_{i_p} with respect to CPU in percent and RAM in MB is indicated by $r^C_{(j_{i_p},k_v)}$ and $r^R_{(j_{i_p},k_v)}$. An instance k of a VM of type v is referred to as k_v. Although virtually unlimited, we assume the number $k_v^{\#}$ of leasable VMs of type v to be limited in a certain time period starting at t and specify the set of VM instances of type v as $K_v = \{1, ..., k_v^{\#}\}$, where $k_v \in K_v$. The cost for leasing one VM instance of type v is indicated by c_v. The remaining, *free* resource capacities of a VM instance k_v regarding CPU and RAM after scheduling and placing of service invocations is referred to as $f^C_{k_v}$ and $f^R_{k_v}$.

As previously stated, a service has to be deployed on a VM in order to be invoked (resulting in a particular service instance). The corresponding *deployment time* depends on the type of the service, i.e., the *service type* which is referred to as st_j. The service deployment time is indicated by Δ_{st_j} and expressed in milliseconds. Parameter $z_{(st_j,k_v,t)} \in \{0, 1\}$ indicates if a service of type st_j is already deployed ($z_{(st_j,k_v,t)} = 1$) at VM instance k_v in the time period starting at t. In addition, the respective VM instance needs to be up and running in order to enable service deployment and invocation. The corresponding VM *start-up time* (in milliseconds) for a VM instance of type v is labeled with Δ_v, whereas $\Delta = \max_{v \in V}(\Delta_v)$ refers to the maximum start-up time. Whether the VM instance k_v is already up and running in a time period starting at t is indicated by the parameter $\beta_{(k_v,t)} \in \{0, 1\}$ – similar to $z_{(st_j,k_v,t)}$.

The *remaining* execution time in milliseconds for a process instance i_p is indicated by e_{i_p}. It can be computed by aggregating the execution times $e_{j_{i_p}}$ of service invocations j_{i_p} according to the structure of process instance i_p. As previously stated, we account for XOR-blocks, AND-blocks, and Repeat loops in addition to Sequences. Corresponding aggregation specifications accounting for a worst-case analysis are provided in Table 6.1 – without an index for a specific process instance i_p. For the sake of simplicity, we assume the services' execution times not to be dependent on the concrete VM instance k_v. Accounting for VM-dependent service execution times could be achieved straightforwardly, since it does not affect our approach for computing optimal solutions to the SIPP. Only the way of computing the execution times would have to be adapted. Instead, we account for VM resources in terms of CPU and RAM as previously stated.

For a Sequence, the execution times of the respective service invocations have to be added up. In order to account for a worst-case analysis, we need to additionally consider the corresponding service deployment times Δ_{st_j} and the worst VM start-up time Δ, each expressed in milliseconds. With respect to an AND-block, it is necessary to account for the different *paths* l within the AND-block. The set of all paths is indicated by $L = \{1, ..., l^{\#}\}$, where l refers to a certain path. In order to separate the set of paths for AND-blocks from the set of paths for XOR-blocks, we use additional indices a (for AND)

Table 6.1: Worst-Case Aggregation Specifications

Pattern	Execution time (e)
Sequence	$e^{seq} = \sum_{j \in J^{seq}} (e_j + \Delta_{st_j} + \Delta)$
AND-block	$e^{AND} = \max_{l \in L_a}(\sum_{j \in J^l} (e_j + \Delta_{st_j} + \Delta))$
XOR-block	$e^{XOR} = \max_{l \in L_x}(\sum_{j \in J^l} (e_j + \Delta_{st_j} + \Delta))$
Repeat loop	$e^{RL} = re \cdot e^{seq}$

and x (for XOR). In order to obtain the execution time for an AND-block, we need to compute the execution times for each of the paths $l \in L_a$ separately and then take the maximum, which is indicated in Table 6.1. We thereby implicitly assume the steps within a path of the AND-block to be arranged sequentially, so that the corresponding execution time can be computed according to the aggregation specification for a Sequence. In order to account for interlaced structures, we apply a technique for recursively combining the provided aggregation specifications.

The aggregation specification for computing the execution time for an XOR-block is basically the same as for an AND-block – at least in terms of a worst-case analysis since we need to consider the *worst* path in terms of execution time, i.e., the longest path. But in contrast to an AND-block, only one of the paths within an XOR-block will finally be invoked. With respect to Repeat loops, the structure that has to be repeated is invoked multiple times. For a worst-case analysis, we assume a maximum number re of repeated invocations. With respect to Table 6.1, we consider a Sequence, indicated by e^{seq}, as the structure to be repeatedly invoked. However, also single process steps or whole processes comprising AND-/XOR-blocks could be subject for repeated invocations and, thus, for Repeat loops.

The deadline DL_{i_p} indicates at which point in time the enactment of process instance i_p has to be finished. If DL_{i_p} is violated, penalty cost will accrue. The penalty cost depends on time and duration, respectively, the invocation of process instance i_p took longer than restricted by the deadline, indicated by $e^p_{i_p}$, and on the penalty cost per time unit, referred to as $c^p_{i_p}$.

For finally deciding whether to schedule a certain service invocation j_{i_p} in the optimization period starting at t, we use binary decision variables $x_{(j_{i_p},k_v,t)} \in \{0,1\}$. A value $x_{(j_{i_p},k_v,t)} = 1$ indicates that the service invocation j_{i_p} of process instance i_p should be scheduled in the period starting at t on the k-th VM of type v (and invoked afterwards), whereas a value $x_{(j_{i_p},k_v,t)} = 0$ indicates that the scheduling and invocations can be delayed. For indicating if we need to lease a certain VM instance k_v of type v in the period starting at t, the decision variable $y_{(k_v,t)} \in \mathbb{N}_0$ is used. In contrast to $x_{(j_{i_p},k_v,t)}$, $y_{(k_v,t)}$ may take values greater than 1 indicating that VM instance k_v should be leased in a optimization period starting at t for $y_{(k_v,t)}$ BTUs. The BTU is the minimum leasing duration. Thus, releasing a VM before the end of the BTU corresponds to wasting paid resources. The actual leasing duration of a particular VM is always a multiple of the

BTU. The *remaining* leasing duration for a specific VM instance k_v in the period starting at t is labeled with $d_{(k_v,t)}$. The total number of VMs of type v to lease in the period starting at t is indicated by $\gamma_{(v,t)}$.

Having described the underlying system model in this section, the subsequent sections present the optimization problem. As described, the optimization problem takes as input a set of process instances I_p including their process steps and corresponding service invocations J_{i_p}. In addition to that, a set of computational resources is needed (V). An overview of all veraibles is presented in Table 6.2.

Table 6.2: Variable Descriptions for the SIPP

Variable Name	Description
$v \in V = \{1, ..., v^{\#}\}$	V specifies the set of VM types and v is a specific type.
$k_v \in K_v = \{1, ..., k_v^{\#}\}$	K_v specifies the amount of VMs of type v.
k_v	Defines the k^{th} VM instance of type v.
$p \in P = \{1, ..., p^{\#}\}$	P specifies the set of process models and p is a specific process model.
$i_p \in I_p = \{1, ..., i_p^{\#}\}$	I_P is the set of all process instances, and i_p represents a specific process instance of process model p.
$j_{i_p}, j_{i_p}^* \in J_{i_p}$	J_{i_p} is the set of process steps of a process instance i_p which have to be executed to fulfill i_p. j_{i_p} is a specific process step of the process instance i_p and $j_{i_p}^*$ is the *next* process step of process instance i_p.
${j_{i_p}}^{run} \in J_{i_p}^{run}$	$J_{i_p}^{run}$ defines a set of running process steps of the process instance i_p and ${j_{i_p}}^{run}$ defines specific running process step of the process instance i_p.
e_{i_p}	e_{i_p} is the remaining execution duration of process instance i_p.
$e_{j_{i_p}}, e_{j_{i_p}}^{run}, e_{j_{i_p}}^{run_{k_v}}$	$e_{j_{i_p}}$ is the remaining execution duration of step j of the process instance i_p. $e_{j_{i_p}}^{run}$ (or $e_{j_{i_p}}^{run_{k_v}}$) is the remaining execution duration of the already running process step j of process instance i_p (on the k-th VM of type v).
$ex_{j_{i_p}}$, $ex_{j_{i_p}^*}$	$ex_{j_{i_p}}$, $ex_{j_{i_p}}^*$ are helper variables defining the combined remaining execution, remaining deployment time and VM startup time if the process step j is scheduled.
$\hat{e}_{i_p}^l, \hat{e}_{i_p}^s$	Defines the combined remaining execution, remaining deployment time and VM startup time if the process step j is not yet scheduled or running. The indices l and s define if this step is part of a complex pattern (AND-block, XOR-block or Repeat loops) (l) or a sequential process (s).
$e_{i_p}^p$	Defines the amount of penalties which accrue if the process instance i_p is delayed.
DL_{i_p}	Defines the deadline for the process instance i_p, i.e., a specific point in time represented as the time elapsed since 01/01/1970 in milliseconds.
t, τ_t, τ_{t+1}, τ_{t_s}	t defines the beginning of a time period, τ_t defines the current time period, and τ_{t+1} defines the next time period, i.e., a point of time in the future and τ_{t_s} defines a specific point of time.
s_v^C, s_v^R	Defines the amount of total resources in terms of CPU (s_v^C) and RAM (s_v^R) for a particular VM of type v.

Continued on next page

Table 6.2 – Continued from previous page

Variable Name	Description
$f_{k_v}^C$, $f_{k_v}^R$	Defines the available resources of the VM v in terms of CPU and RAM.
$r_{(j_{i_p,k_v})}^C$, $r_{(j_{i_p,k_v})}^R$	Defines the amount of required resources for a specific service invocation j_{i_p,k_v} of process instance i_p on a specific VM instance k_v in terms of CPU (r^C) and RAM (r^R).
st_j	Defines the service type of the process step j.
Δ_{st_j},$\Delta_{j_{i_p}}$	Defines the time it takes to deploy a service of type st of the process step j or of a specific process step j of process instance i_p.
Δ_v, Δ	Δ_v defines the time it takes to start a new VM of type v expressed in milliseconds. Δ defines the max of starting a VM of any type, i.e, $\max_{v \in V}(\Delta_v)$.
$z_{(st_j,k_v,t)}$	Indicates whether a specific service type st_j is deployed on the VM k_v in time period t.
$\beta_{(k_v,t)}$	Indicates if the VM k_v is/was running in the time period starting at t.
$e_{i_p}^{seq}$, $e_{i_p}^{L_a}$, $e_{i_p}^{L_x}$, $e_{i_p}^{RL}$	Defines the execution time for a sequence ($e_{i_p}^{seq}$), AND-block ($e_{i_p}^{L_a}$), XOR-block ($e_{i_p}^{L_x}$), or Repeat loop ($e_{i_p}^{RL}$) for a specific process instance i_p.
$l \in L = \{1, ..., l^{\#}\}$, L_a, L_x, L_{re}	L indicates the set of all paths and l indicates a specific path within a process. L_a, L_x, L_{re} define the paths for AND-blocks, XOR-blocks or Repeat loops.
re	Defines the maximum amount of repetitions within a Repeat loop.
$c_{i_p}^p$	Defines the cost per time unit of delay for the process instance i_p.
$x_{(j_{i_p},k_v,t)}$	Defines if the process step j of process instance i_p should be invoked on VM k_v in the time period t.
$y_{(k_v,t)}$	Defines how often a VM k_v should be leased in time period t.
BTU	Defines the *Billing Time Unit* (BTU), i.e., *one* leasing duration, i.e., a certain time period in milliseconds.
M	M is a constant needed to give some constraints a higher weight.
ϵ	Defines a short time period in milliseconds which is used to prevent deadlocks.
$d_{(k_v,t)}$, $d_{(k_v,t-1)}$	Defines the remaining leasing duration of VM k_v in time period t or $t-1$.
$\gamma_{(v,t)}$	Defines the amount of leased VMs of type v in the time period starting at t.
c_v	Defines the leasing cost of VM type v.
$g_{k_v,t}$	Helper variable indicating if the VM k_v is running in the time period starting at t or needs to be started.

Continued on next page

Table 6.2 – Continued from previous page

Variable Name	Description
ω_f^C, ω_f^R	Helper variables, representing constant values to give more weight to a term in the optimization function.
$z_{(j_{i_p},k_v,t)}$	This variable indicates if the service type of a service invocation j_{i_p} has the same service type as the service instance which is deployed at VM k_v in the time period starting at t.

6.3.2 Optimization Problem

In this section, we model the scheduling and placement of service invocations for enacting corresponding process instances – the SIPP – as an optimization problem. Using the system model provided in Section 6.3.1, we gradually develop the corresponding optimization model for the current optimization period starting at t in this section. A multi-period extension of the optimization problem is discussed in Section 6.3.3.

$$\begin{aligned}\min \quad & \sum_{v \in V} c_v \cdot \gamma_{(v,t)} \\ & + \sum_{p \in P} \sum_{i_p \in I_p} c_{i_p}^p \cdot e_{i_p}^p \\ & + \sum_{v \in V} \sum_{k_v \in K_v} (\omega_f^C \cdot f_{k_v}^C + \omega_f^R \cdot f_{k_v}^R) \\ & - \sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in J_{i_p}^*} \frac{1}{DL_{i_p} - \tau_t} x_{(j_{i_p},k_v,t)}\end{aligned} \tag{6.1}$$

In (6.1), the objective function, which is subject for minimization, is depicted. It comprises four terms. In the first term, i.e, $\sum_{v \in V} c_v \cdot \gamma_{(v,t)}$, we compute the total cost accruing due to leasing $\gamma_{(v,t)}$ VM instances of type v at a cost of c_v per VM instance in the period starting at t. The second term, i.e., $\sum_{p \in P} \sum_{i_p \in I_p} c_{i_p}^p \cdot e_{i_p}^p$, accounts for penalties arising due to violating deadlines. For computing these penalties, a linear penalty function is applied [69]: We multiply the durations $e_{i_p}^p$, i.e., the time units the invocations of process instances i_p took longer than restricted by the corresponding deadline, with the cost per time unit, represented by $c_{i_p}^p$. The third term, i.e., $\sum_{v \in V} \sum_{k_v \in K_v} (\omega_f^C \cdot f_{k_v}^C + \omega_f^R \cdot f_{k_v}^R)$, regards the sum of *free* resource capacities $f_{k_v}^C$ (CPU) and $f_{k_v}^R$ (RAM) for all (leased) VM instances – weighted with corresponding weights ω_f^C and ω_f^R. Finally, in the fourth term, we calculate the difference between the deadlines DL_{i_p} for all process instances and the current period τ_t and compute the corresponding reciprocal value. This way, we deduce a measure for the *urgency* and *importance*, respectively, for each process instance, since the closer the deadline is for a process instance i_p, the larger is $\frac{1}{DL_{i_p} - \tau_t}$.

With the first term, we aim at minimizing the total cost. With the second term, we aim at minimizing penalty cost. With the third term, we aim at minimizing leased

but unused cloud-based computational resource capacities. With the fourth term, we aim at *maximizing* (note the minus in front of the term) the relative importance of the scheduled service invocations. Since the minimization of the leasing cost and the penalty cost should have highest priority, we set the weights ω_f^C and ω_f^R to a very low value such as 0.000001. Since the actual values for the deadlines DL_{i_p} and the current period τ_t are quite large (as they are represented as the time elapsed since 01/01/1970 in milliseconds), their difference remains large, so that the reciprocal value becomes rather small. Thus, the minimization of leasing cost and penalty cost receives the highest (relative) weights and thereby priorities within the objective function.

The constraints in (6.2) demand the deadlines DL_{i_p} not to be violated for all $p \in P$, $i_p \in I_p$, $j_{i_p} \in J_{i_p}^{run}$. For this, the sum of the *remaining* execution time e_{i_p} and the next optimization period starting at τ_{t+1} has to be lower or equal with respect to the deadline. The length of the current optimization period, consequently, the start of the next optimization is defined by $t+1$ and is restricted in (6.3). τ_{t+1} has to be greater or equal to the current optimization period at τ_t plus a small value $\epsilon > 0$. ϵ is needed to avoid optimization deadlocks resulting from a too small or negative value for τ_{t+1}.

$$\tau_{t+1} + e_{i_p} + e_{j_{i_p}}^{run} \leq DL_{i_p} + e_{i_p}^{p} \tag{6.2}$$

$$\tau_{t+1} \geq \tau_t + \epsilon \tag{6.3}$$

The remaining execution time e_{i_p} is computed in (6.4) for all $p \in P$, $i_p \in I_p$. Depending on the structures of the process instances, different remaining execution times apply, i.e., for sequences using (6.5), for AND-blocks (6.6), for XOR-blocks (6.7) and for Repeat loops (6.8). Notably, a process instance may consist of a combination of different process patterns, which have to be summed up (see (6.4)).

As indicated in (6.5)-(6.8), the remaining execution time e_{i_p} can be reduced if a service invocation j_{i_p} is scheduled on a VM instance k_v. The helper variables $e_{i_p}^s$ and $e_{i_p}^l$ are defined in (6.10) and (6.11). As described in (6.9), in this case, the corresponding execution time $e_{j_{i_p}^*}$ along with the service deployment time $\Delta_{j_{i_p}^*}$ and VM start-up time Δ will be added up, resulting in the variable $ex_{j_{i_p}^*}$ and subtracted from the remaining execution time e_{i_p}. Note that execution times $e_{j_{i_p}}$ for service invocations that have already been scheduled in previous optimization periods will be set to zero. Further, since steps within AND-blocks may be invoked in parallel, it is possible to schedule multiple *next* service invocations – one *next* service invocation per branch of the AND-block.

$$e_{i_p} = e_{i_p}^{seq} + e_{i_p}^{L_a} + e_{i_p}^{L_x} + e_{i_p}^{RL} \tag{6.4}$$

$$e_{i_p}^{seq} = \begin{cases} \hat{e}_{i_p}^s - ex_{j_{i_p}^*} & \text{, if } x_{(j_{i_p}^*, k_v, t)} = 1 \\ \hat{e}_{i_p}^s & \text{, else} \end{cases} \tag{6.5}$$

$$e_{i_p}^{L_a} = \begin{cases} \max_{l \in L_a}(\hat{e}_{i_p}^{l} - ex_{j_{i_p}^*}) & \text{, if } x_{(j_{i_p}^*, k_v, t)} \\ \max_{l \in L_a}(\hat{e}_{i_p}^{l}) & \text{, else} \end{cases} \tag{6.6}$$

$$e_{i_p}^{L_x} = \begin{cases} \max_{l \in L_x}(\hat{e}_{i_p}^{l} - ex_{j_{i_p}^*}) & \text{, if } x_{(j_{i_p}^*, k_v, t)} \\ \max_{l \in L_x}(\hat{e}_{i_p}^{l}) & \text{, else} \end{cases} \tag{6.7}$$

$$e_{i_p}^{RL} = \begin{cases} re \cdot \hat{e}_{i_p}^{s} - ex_{j_{i_p}^*} & \text{, if } x_{(j_{i_p}^*, k_v, t)} \\ re \cdot \hat{e}_{i_p}^{s} & \text{, else} \end{cases} \tag{6.8}$$

$$ex_{j_{i_p}^*} = \sum_{v \in V} \sum_{k \in K_v} ((e_{j_{i_p}^*} + \Delta_{j_{i_p}^*} + \Delta) x_{(j_{i_p}^*, k_v, t)}) \tag{6.9}$$

$$\hat{e}_{i_p}^{s} = \sum_{j_{i_p} \in J_{i_p}^{seq}} (e_{j_{i_p}} + \Delta_{j_{i_p}} + \Delta) \tag{6.10}$$

$$\hat{e}_{i_p}^{l} = \sum_{j_{i_p} \in J_{i_p}^{l}} (e_{j_{i_p}} + \Delta_{j_{i_p}} + \Delta) \tag{6.11}$$

Since it might be the case that in an optimization period starting at t certain service invocations j_{i_p} are currently running, we need to add the corresponding remaining execution times, indicated by $e_{j_{i_p}}^{run}$, in (6.2). If the deadlines DL_{i_p} were violated, the corresponding durations $e_{i_p}^{p}$ would increase which in turn would lead to higher penalty cost (see (6.1)).

The constraints in (6.12) make sure that for all $v \in V$, $k \in K_v$, VM instances k_v will be leased if service invocations j_{i_p} are to be scheduled on them. This is indicated by a value of 1 for the corresponding decision variables, i.e., $x_{(j_{i_p}, k_v, t)} = 1$. The corresponding VM instance k_v has already been leased (and paid for) in a previous optimization period, which is indicated by a value of 1 for the parameter β_{k_v}, i.e., $\beta_{(k_v,t)} = 1$. Or we need to set the corresponding decision variable $y_{(k_v,t)}$ accordingly, i.e., $y_{(k_v,t)} \geq 1$. Since it may be the case that multiple service invocations are intended to be scheduled on VM instance k_v, the sum of the left-hand side of (6.12) might exceed a value of 1. Thus, in order to satisfy these constraints, we multiply $(\beta_{(k_v,t)} + y_{(k_v,t)})$ with a sufficiently large value M, i.e., 1,000,000. We chose this value since it is unlikely that there is a VM which is able to host more than 1,000,000 parallel service invocations.

$$\sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in J_{i_p}} x_{(j_{i_p}, k_v, t)} \leq (\beta_{(k_v,t)} + y_{(k_v,t)}) \cdot M \tag{6.12}$$

With respect to scheduling service invocations on a VM instance k_v, we demand the corresponding service types st_j in (6.13) to be the same. This way, we aim at achieving that on a specific VM instance only one service instances of the same type is deployed.

Differently stated, service instances with different service types may not be deployed at the same VM instance.

$$x_{(j^1_{i^1_{p^1}},k_v,t)} + x_{(j^2_{i^2_{p^2}},k_v,t)} \leq 1 \tag{6.13}$$

The constraints in (6.14) and (6.16) make sure that for all $v \in V$, $k \in K_v$ the resources (with respect to CPU and RAM) required by the service invocations that either already run on VM instance k_v or are scheduled to run on it, do not exceed the respective capacity of a VM of the type v. The remaining *free* capacities $f^C_{k_v}$ and $f^R_{k_v}$ are determined in (6.15) and (6.17). Note that we consider *free* capacities $f^C_{k_v}$ and $f^R_{k_v}$ for VMs that are either already running or leased in the period starting at t. For this, we use an additional variable $g_{(k_v,t)} \in \{0,1\}$, which takes a value of 1 only if VM k_v is either running ($\beta_{k_v} = 1$) or leased ($y_{(k_v,t)} \geq 1$) in the period starting at t. This is indicated in (6.18)-(6.20) for all $v \in V$, $k \in K_v$.

$$\sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in (J^*_{i_p} \cup J^{run}_{i_p})} r^C_{(j_{i_p},k_v)} x_{(j_{i_p},k_v,t)} \leq s^C_v \tag{6.14}$$

$$g_{(k_v,t)} \cdot s^C_v - \sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in (J^*_{i_p} \cup J^{run}_{i_p})} r^C_{(j_{i_p},k_v)} x_{(j_{i_p},k_v,t)} \leq f^C_{k_v} \tag{6.15}$$

$$\sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in (J^*_{i_p} \cup J^{run}_{i_p})} r^R_{(j_{i_p},k_v)} x_{(j_{i_p},k_v,t)} \leq s^R_v \tag{6.16}$$

$$g_{(k_v,t)} \cdot s^R_v - \sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in (J^*_{i_p} \cup J^{run}_{i_p})} r^R_{(j_{i_p},k_v)} x_{(j_{i_p},k_v,t)} \leq f^R_{k_v} \tag{6.17}$$

$$g_{(k_v,t)} \geq \beta_{(k_v,t)} \tag{6.18}$$

$$g_{(k_v,t)} \geq y_{(k_v,t)} \tag{6.19}$$

$$g_{(k_v,t)} \leq \beta_{(k_v,t)} + y_{(k_v,t)} \tag{6.20}$$

The constraints in (6.21) demand that the remaining leasing duration d_{k_v} plus the number $y_{(k_v,t)}$ of BTU for VM k_v (for all $v \in V$, $k \in K_v$) is larger or equal to the sum of remaining execution time $e_{j_{i_p}}$, the deployment time $\Delta_{j_{i_p}}$ (if the corresponding service type is not yet deployed, i.e., $z_{(j_{i_p},k_v,t)} = 0$), and the VM start-up time Δ, provided the VM is not yet running ($\beta_{(k_v,t)} = 0$). As we conduct a worst-case analysis, this remaining leasing duration has to be greater or equal to the service execution times of all *next* service invocations (for all $p \in P$, $i_p \in I_p$, $j_{i_p} \in J^*_{i_p}$) that should be scheduled on this VM including the corresponding service deployment and VM start-up time.

This way, we make sure that service invocations scheduled on a VM instance will not be *moved* to another VM instance during their invocation. The same restriction is provided by the constraints in (6.22) for all currently running service invocations. In order to make sure that the running service invocations can be finished on the concrete VM k_v where they have previously been assigned to, we explicitly consider this VM k_v by using an additional index k_v for $e_{j_{i_p}}^{run_{k_v}}$ in (6.22). Since these service invocations are already running, we do not need to consider service deployment and VM start-up times.

$$(e_{j_{i_p}} + \Delta_{j_{i_p}} \cdot (1 - z_{(j_{i_p},k_v,t)}) + \Delta \cdot (1 - \beta_{(k_v,t)}))x_{(j_{i_p},k_v,t)} \leq d_{(k_v,t)} + y_{(k_v,t)} \cdot BTU \quad (6.21)$$

$$e_{j_{i_p}}^{run_{k_v}} \leq d_{(k_v,t)} + y_{(k_v,t)} \cdot BTU \quad (6.22)$$

The sum of the $y_{(k_v,t)} \geq 1$ (for $v \in V$) values indicates the total number of VM instances and for how many $BTUs$ they have to be leased. This sum determines $\gamma_{(v,t)}$ in (6.23). In this respect, it has to be noted that $y_{(k_v,t)}$ includes both, the decision, which concrete VM instance k_v to lease and for how many $BTUs$.

$$\sum_{k \in K_v} y_{(k_v,t)} \leq \gamma_{(v,t)} \quad (6.23)$$

Finally, in (6.24), we make sure that for all $p \in P$, $i_p \in I_p$, $j_{i_p} \in J_{i_p}^*$ each service invocation can be scheduled only on one VM instance. In (6.25), we set the decision variables for all service invocations already running (for all $p \in P$, $i_p \in I_p$, $j_{i_p} \in J_{i_p}^{run}$, $v \in V, k \in K_v$) in the period starting at t to 1. The constraints in (6.26)-(6.29) restrict the decision variables $x_{(j_{i_p},k_v,t)}$ (for all $p \in P$, $i_p \in I_p$, $j_{i_p} \in J_{i_p}^*$, $v \in V, k \in K_v$), $y_{(k_v,t)}$ (for all $v \in V$, $k \in K_v$), and $e_{i_p}^p$ (for all $p \in P$, $i_p \in I_p$) to take values from $\{0,1\}$, $\mathbb{N}_0$, and $\mathbb{R}^+$, respectively.

$$\sum_{v \in V} \sum_{k \in K_v} x_{(j_{i_p},k_v,t)} \leq 1 \quad (6.24)$$

$$x_{(j_{i_p},k_v,t)} = 1 \quad (6.25)$$

$$x_{(j_{i_p},k_v,t)} \in \{0,1\} \quad (6.26)$$

$$g_{(k_v,t)} \in \{0,1\} \quad (6.27)$$

$$y_{(k_v,t)} \in \mathbb{N}_0 \quad (6.28)$$

$$e_{i_p}^p \in \mathbb{R}^+ \quad (6.29)$$

The optimization model for the current optimization period starting at t is obtained by assembling (6.1)-(6.29).

Having defined the optimization model used for computing an optimal solution to the SIPP in the period starting at t, we are now able to realize a multi-period scheduling as described in the next section.

6.3.3 Multi-period Scheduling Approach

Applying the approach for computing an optimal solution to the SIPP as presented in the previous Section 6.3.2, we obtain a scheduling plan, i.e., which service invocation to schedule in the period starting at τ_t. But as new requests for the invocation of further process instances i_p may arise during the scheduled service invocations, the previously computed *optimal* solution might no longer be optimal. Thus, we may not only account for one single optimization period but need to conduct multiple optimization steps.

In each optimization step, we only schedule the *next* service invocations for the process instances i_p which cannot be delayed any further according to (6.2). For this, we extract the decision variables $x_{(j_{i_p},k_v,t)}$, lease and start corresponding VMs k_v (if they have not been leased and started yet), deploy respective service invocations j_{i_p} on aforementioned VMs (if they have not been deployed yet), and initiate and monitor their invocation in terms of success and execution time.

The next optimization period in time, i.e., where the next optimization step will be carried out, is indicated by the variable τ_{t+1} – as a result of the optimization. If the invocation of a process instance i_p is finished until τ_{t+1}, we determine whether a QoS violation occurred, i.e., we compute $e^p_{i_p}$. In case of a QoS violation, i.e., if $e^p_{i_p} > 0$, corresponding penalties accrue.

Immediately prior to the next optimization step, i.e., for the next optimization period at τ_{t+1}, which then *becomes* the new current optimization period at τ_t in (6.3), we need to update certain parameters as described subsequently. We set:

- $\beta_{(k_v,t)} = 1$ if VM k_v runs in τ_t; 0 otherwise.
- $z_{(j_{i_p},k_v,t)} = 1$ if service invocation j_{i_p} with the same service type $st_j = st_{j_{i_p}}$ is already deployed at VM k_v in τ_t; 0 otherwise.
- $d_{(k_v,t)} = d_{(k_v,t-1)} + y_{(k_v,t)} \cdot BTU - (\tau_t - \tau_{t-1})$ to account for the time elapsed between the previous (τ_{t-1}) and the current (τ_t) optimization period when computing the remaining leasing duration of VM k_v in the optimization period τ_t.

Service invocations that have already been scheduled in the previous periods are referred to as $j^{run}_{i_p}$. For such service invocations, we determine the remaining execution times $e^{run}_{j_{i_p}}$ as follows: We subtract the time elapsed since the point in time where the corresponding service invocations have been scheduled (τ_{t_s}) from the sum of the services' execution times, the VM start-up times, and the services' deployment times. This is shown in (6.31) whereas $\hat{e}_{j_{i_p}}$ is defined in (6.30). Optimization period τ_{t_s} refers to the period where service invocation j_{i_p} has been scheduled.

$$\hat{e}_{j_{i_p}} = e_{j_{i_p}} + \Delta_{j_{i_p}} + \Delta \tag{6.30}$$

$$e^{run}_{j_{i_p}} = \begin{cases} 0 & \text{, if } j_{i_p} \text{ is finished} \\ \max\left(0, \hat{e}_{j_{i_p}} - (\tau_t - \tau_{t_s})\right) & \text{, else} \end{cases} \tag{6.31}$$

Referring to (6.31), the remaining execution times are considered as 0 if the corresponding service instances (j_{i_p}) have already been invoked, i.e., the service invocation is finished. Further, by taking the maximum in (6.31), we make sure not to consider negative remaining execution times. This means, if an invocation lasts longer than expected, the SIPP has to include this information in the next optimization period, i.e., the corresponding process instance will have a higher priority as it may get delayed otherwise.

Having set up and updated the necessary parameters for the next period, we conduct another optimization step. According to the results of this newly conducted optimization step, i.e., according to the values of the decision variables $x_{(j_{i_p},k_v,t)}$, we schedule and invoke corresponding service instances. Continuing this optimization and scheduling procedure results in an efficient scheduling strategy, which takes optimal scheduling decisions (at the optimization period τ_t) into account and minimizes unused, free VM capacities (see the usage of $f^C_{k_v}$ and $f^R_{k_v}$ in (6.1)).

6.4 Evaluation

As a proof of concept, the proposed SIPP model has been thoroughly evaluated. We apply our prototype and testbed framework ViePEP as introduced in Section 2.4 and Section 4.2. However, since we follow a different approach, i.e., using a optimization model for process scheduling and resource optimization, we had to re-implement ViePEP resulting in ViePEP 2.0.

In the following subsections, we present first ViePEP 2.0 in Section 6.4.1, second, the evaluation setting in Section 6.4.2, third the applied metrics in Section 6.4.3, and last but not least, we discuss the quantitative evaluation results in Section 6.4.4.

6.4.1 ViePEP 2.0

Figure 6.1 shows the high-level architecture of the enhanced ViePEP. As in the former version, ViePEP 2.0 depicts five top-level entities: A Client can model business processes and define optional SLAs, i.e., defining deadline constraints for the full process. The business process can be requested for execution to the BPMS VM through the Process Manager.

The Backend VM remains mostly unchanged. It hosts the software-based services on an application server, e.g., Apache Tomcat and a Monitor which monitors the deployed services in terms of CPU and RAM usage. The monitored data is forwarded to the BPMS VM through a JMS-based Message Queue.

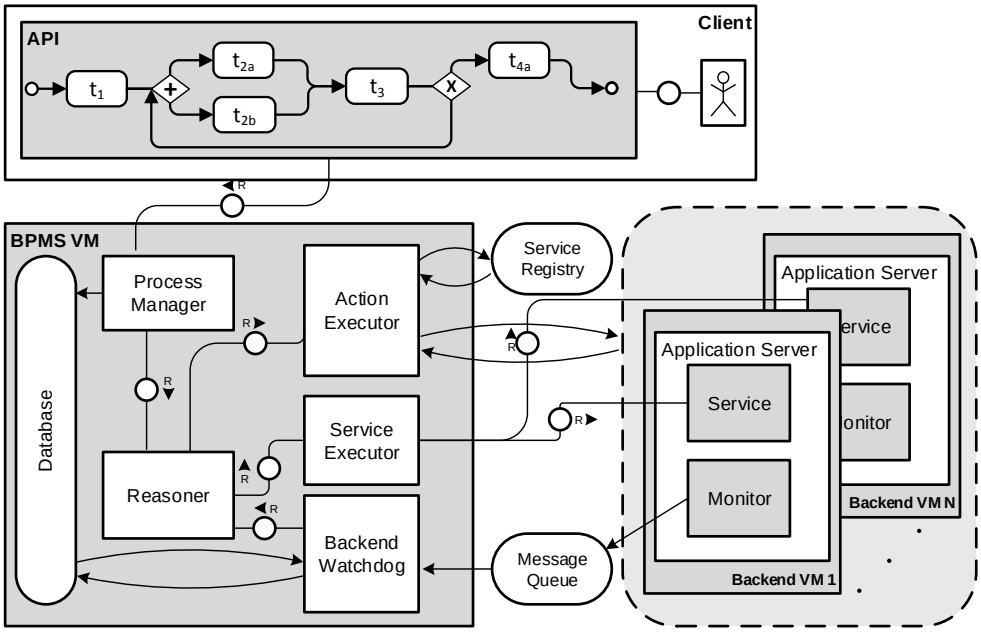


Figure 6.1: ViePEP 2.0 – Architecture

The BPMS VM has partly been reimplemented. First, it hosts the Process Manager which acts as a single-point entry component to the Client. Business processes get requested through it. After being requested, the process instances are immediately stored within a Database and the scheduling and resource optimization is triggered. While in the latest version of ViePEP, i.e., version 1.1, two components were responsible for doing that, in ViePEP 2.0 it is just one component, i.e., the Reasoner which implements the SIPP model from Section 6.3.2.

The output of the SIPP model is two-fold. First, it defines how many resources should be leased or which already leased resources can be released. Hence, the Reasoner is connected with an Action Executor which is connected to the cloud infrastructure. It can instantiate or terminate VMs. In addition, the Action Executor is connected with a Service Registry which hosts the needed software-based services in form of WAR-files. Hence, the Action Executor is also responsible for deploying the needed services among the leased resources. The second output of the SIPP model is a complete scheduling plan, i.e., this plan defines on which VM instance a specific service instance should be invoked. In order to do this, the Action Executor is connected with the Service Executor which either invokes the requested services immediately or maintains a queue for future invocations. Beside of these subcomponents, a Backend Watchdog is employed. It receives new monitoring data via the Message Queue and is able to verify whether the data indicates a critical state or is still in a normal state. In case of an alert, e.g., the CPU or RAM load is too high on a specific VM instance, the Backend Watchdog can notify the

Table 6.3: Evaluation Process Models

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	0	1	0
6	9	1	1	0
7	8	0	0	0
8	3	0	1	0
9	4	1	1	1
10	20	0	4	0

Reasoner in order to trigger resolving the SIPP model with updated information to find an appropriate countermeasure.

ViePEP 2.0 is implemented in form of an OSGi 4.3-based framework[2], i.e., Apache Karaf 2.3.2[3]. As a communication engine between the Backend VMs and the BPMS VM we apply a JMS-based Message Queue, i.e., Apache ActiveMQ 5.7.0[4]. ViePEP 2.0 itself is implemented using Java 1.7. Both the BPMS VM and the Backend VMs are running in a private cloud testbed running OpenStack (OpenStack Folsom[5]). To solve the optimization problem, the SIPP is modeled using Java ILP[6]. It provides a Java interface to MILP solvers – in ViePEP 2.0, IBM CPLEX[7] is applied.

6.4.2 Setting

Test Collection

To evaluate the proposed optimization approach, we choose a subset of the SAP reference model [25, 56]. The SAP reference model has been analyzed and exploited in various scientific papers and provides a well-known and widely accepted foundation for our evaluation [80]. Out of the 604 process models in the SAP reference model, we choose 10 exemplary process models which feature different degrees of complexity in terms of process patterns (see Appendix B).

Table 6.3 shows the basic characteristics of the 10 different models, i.e., if a model includes AND-blocks, XOR-blocks, Repeat loops or a combination of them. Notably, each split (AND, XOR) also includes a join. Figure 6.2 shows two example process models, namely No. 5, which contains an XOR-block, and No. 9, which contains an AND leading

[2] http://www.osgi.org/Main/HomePage
[3] http://karaf.apache.org/
[4] http://activemq.apache.org
[5] http://www.openstack.org/software/folsom/
[6] http://javailp.sourceforge.net/
[7] http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

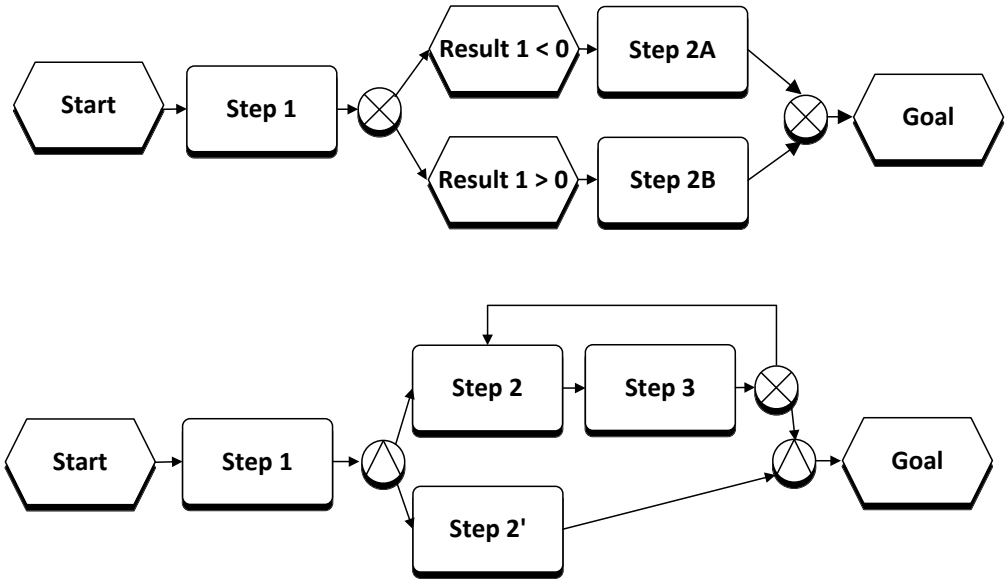


Figure 6.2: Process No. 5 (top), Process No. 9 (bottom)

to two new branches. No. 9 also contains a Repeat loop. A full presentation of all 10 used business process models can be found in Appendix B.

While process models in the SAP reference model usually contain some human-provided services, we are focusing on software services only. Hence, for all services in the process models, we deploy services with differing degrees of computational complexity, resource demands, and execution duration. For this, we apply the *lookbusy* load generator[8], which is a configurable tool able to generate a particular CPU load for a particular timespan.

Following our assumption that services are shared among processes, we generate 10 different software services as described in Table 6.4. Duration and CPU Load are mean values μ_{cpu} and μ_{dur} of normal distributions as described in (6.32) and (6.33). We assume that the services' actual resource consumption varies to some extent for each service invocation. We assume $\sigma_{cpu} = \frac{\mu_{cpu}}{10}$ respectively $\sigma_{dur} = \frac{\mu_{dur}}{10}$ and we only select values between 95% and 105% of the provided mean value to conduct a reproducible evaluation.

$$f(x, \mu_{cpu}, \sigma_{cpu}) = \frac{1}{\sigma_{cpu}} \phi \left(\frac{x - \mu_{cpu}}{\sigma_{cpu}} \right) \tag{6.32}$$

$$f(x, \mu_{dur}, \sigma_{dur}) = \frac{1}{\sigma_{dur}} \phi \left(\frac{x - \mu_{dur}}{\sigma_{dur}} \right) \tag{6.33}$$

To estimate the need for computational resources for a service invocation on different VMs, we assume that each service invocation can be fully parallelized among the available CPUs. This means, while the mean execution time stays the same, the amount of required

[8] `http://devin.com/lookbusy/`

CPU load is divided by the number of available cores. For example, if a service invocation needs 100% of a single-core VM, only a quarter of it will be used on a quad-core VM, i.e., 25% for each core. This enables 4 times more simultaneous invocations of that particular service instance on a quad-core VM compared to a single-core VM.

Table 6.4: Evaluation Services

Service No.	CPU Load in % (μ_{cpu})	Service Makespan in sec. (μ_{dur})
1	5	30
2	10	80
3	15	120
4	30	100
5	45	10
6	55	20
7	70	40
8	125	20
9	125	60
10	190	30

Applied SLAs

SLAs are defined on process level and contain the deadline for the complete process enactment. To evaluate our optimization approach under different settings, two different SLAs are linked to each process model and evaluated in separate runs. The first set of SLAs provides rather "lenient" values, i.e., the SLAs are defined with significant leeway for delays, while the second set provides more strict values. For the lenient SLAs, the average violation threshold is set to $2.5 * ED_{I_p}$, while for the strict SLAs, the threshold is $1.5 * ED_{I_p}$, where ED_{I_p} is the mean execution time of a process model. Those values were chosen with having in mind the start-up time for VMs Δ and the time which is needed to deploy the respective service onto them, i.e., the service deployment time Δ_{st_j}.

Process Request Arrival Patterns

Similar to our former evaluations from Section 4.5 and Section 5.4 we apply different process request arrival patterns in order to evaluate our approach under different load. The first scenario follows a *Constant* arrival pattern, i.e., in regular intervals (120 seconds), we choose 5 instances of different process models, i.e., in the first round we request process model No. 1 to 5, in the second round No. 6 to 10, the third round again No. 1 to 5 etc. This is repeated until a total of 50 process instance requests are sent to ViePEP.

The second arrival pattern follows a *Pyramid*-like function and the process instances requested at particular points of time are randomly chosen: A total of 100 process instances are put in a randomly shuffled queue and a different amount of process instance requests is sent simultaneously. We start with a low number, e.g., with 1 instance a time,

increase it to a peak, and decrease it again to 1 instance request. After a few iterations, the amount increases again slowly to a peak until all 100 instance requests are sent to ViePEP. The detailed function can be found in (6.34).

$$f(n) = a \begin{cases} (n+1)/(n+1) & \text{if } 0 \leq n \leq 3 \\ \lceil (n+1)/4 \rceil & \text{if } 5 \leq n \leq 17 \\ 0 & \text{if } 18 \leq n \leq 19 \\ (n/n) & \text{if } 20 \leq n \leq 35 \\ \lceil (n-9)/20 \rceil & \text{if } 36 \leq n \leq 51 \end{cases} \tag{6.34}$$

In this function, n represents the time in minutes, which is used to calculate a and a represents the amount of process instance requests which will be sent to ViePEP. Between each bunch of a requests we assign a waiting period of 60 seconds. It remains to mention that for repeated evaluation runs, the same order of process models in the queue is applied in order to generate reproducible results.

Notably, in contrast to our former evaluations (Section 4.5 and Section 5.4), we reduced the total amount of requested process instances. However, contrary to before, we choose 10 process models of different complexities. In addition, we choose 10 different software services which need to be invoked, each needing a different amount of CPU and RAM. The result is many-faceted realistic evaluation scenario.

The arrival patterns are depicted in Figure 6.3-6.4.

Baseline

To compare our optimization approach against a baseline, we apply a basic strategy for resource provisioning and scheduling which is based on existing work on process scheduling [36, 39, 69]. It should be noted that we adapted the strategy slightly to fit our basic assumption that service instances may be shared simultaneously among process instances and that Backend VMs host only one particular service instance but there might be several concurrent service invocations.

Applying the baseline strategy (*X-VM for Each*), ViePEP leases a new quad-core Backend VM for a particular service type once the workload on a particular VM is above an upper threshold of 80%. The VM is released again once the workload is below a lower threshold of 20%. These values have been chosen based on experiences collected in our former evaluations from Section 4.5.4 and Section 5.4.2 and from our former work [99]: The 20% lower threshold was chosen due the fact that the OS needs up to 15%, which means, a overall system load of less than 20% means that there are only few service invocations running, and the VM is not needed anymore. The 80% upper threshold has been chosen to be able to handle unexpected deviations of needed resources for running service invocations. However, this does not mean that a VM will not be used 100%.

Hence, there might be several VMs for the same service type. This approach applies a basic scheduling of process instances and in addition takes into account the near future, i.e., VMs are leased for a fixed period which allows to pre-pone future steps in order to use leased resources more efficiently. Further, the baseline also considers the deadline, i.e.,

process instances with an earlier deadline have to be enacted before processes instances with a later deadline.

6.4.3 Metrics

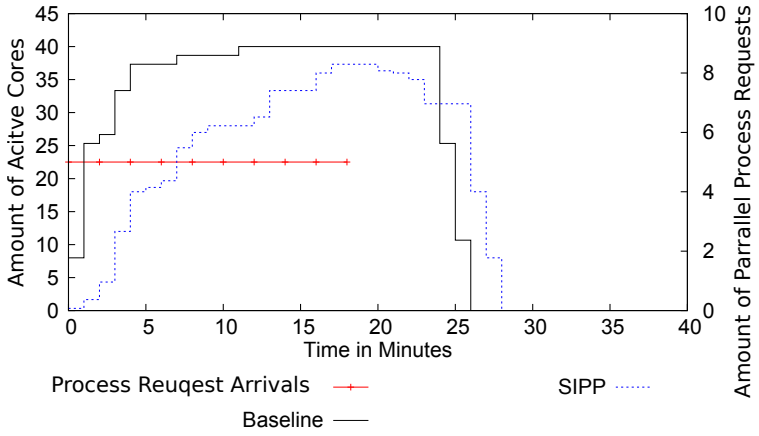
To assess the quality of our optimization approach, we use different metrics. For all numbers, we also calculate the standard deviation σ. Since our optimization problem aims at cost minimization, comparing the *total cost* arising from VM leasing *and* penalties is an obvious option. First we compute the amount of *Max. Active Cores*, then, we apply the following cost model: We assume that it is cheaper to lease a quad-core VM than 4 single-core VMs (respectively a dual-core VM is cheaper than 2 single-cores, etc.). We apply a linear penalty cost model based on [69]: We assign 1 unit of *penalty cost* per 10% of time units of delay. Penalty cost and VM leasing cost over the runtime of all process instances result in the *Total Cost* metric. Second, we measure the *SLA Adherence*, i.e., the percentage of process requests which have been fulfilled on time. Third, we measure the overall duration to process all requests (*Total Makespan*), starting with the point of time when the first request arrives and stopping once all service invocations have been finished.

6.4.4 Results and Discussion

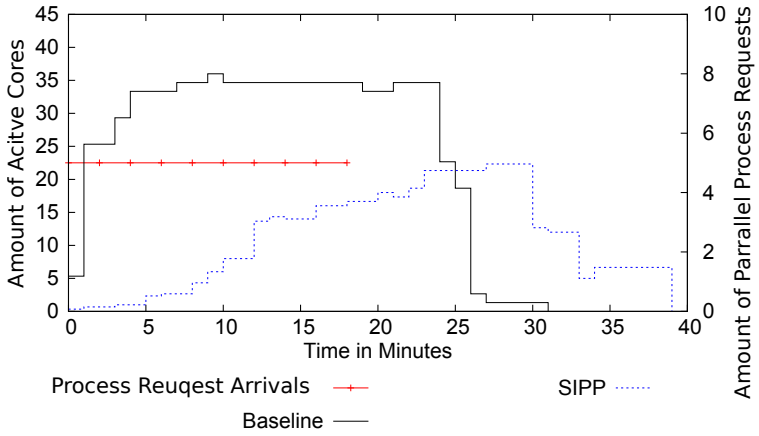
In order to get representative numbers, each scenario (with the SIPP model and the baseline, with lenient and strict SLAs) was executed 3 times over a timespan of 7 days. This has been done to avoid corruption of results due to differing base loads in the OpenStack-based cloud testbed. While Table 6.5 and Table 6.6 presents the average of the conducted evaluation runs including their standard deviations as numbers, Figure 6.3 and Figure 6.4 show the results as charts. For each chart, on the horizontal axis, the time in minutes is presented, on the *left* vertical axis the amount of active (CPU) cores is shown and on the *right* vertical axis, the amount of parallel process requests is presented. It remains to mention that while the metric *Total Makespan in Minutes* in Table 6.5 and Table 6.6 represents the time of executing all process requests, the *Time in Minutes* in Figure 6.3 and Figure 6.4 shows the timespan of leasing the first VM until releasing the last one.

First, we discuss the Constant Arrival scenario for both SLA levels, lenient and strict. As it can be seen in Table 6.5, the SLA adherence using our optimization approach is always above 94.0% (with a rather small standard deviation of 2.0%). This means, our optimization approach was capable of detecting potential shortcomings in time, and rescheduled the service invocations. However, the few SLA violations can be reduced to the fact that it may be cheaper in some situations to accept a short delay than leasing additional VMs.

The SLA adherence for the baseline is much lower: As the numbers show, almost a third of the process requests were delayed resulting in four times higher penalty cost (42.67) than using our SIPP model (10.0) for the strict SLA level. The low amount of penalty cost for SIPP can be explained by our leasing and releasing policy, i.e., we try to



(a) Strict SLA

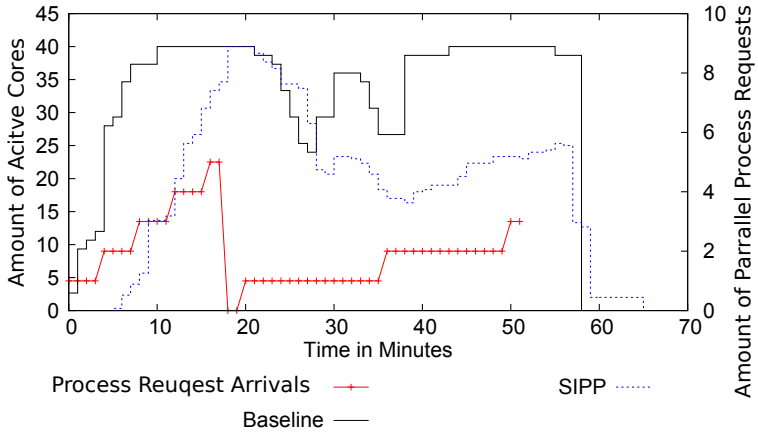


(b) Lenient SLA

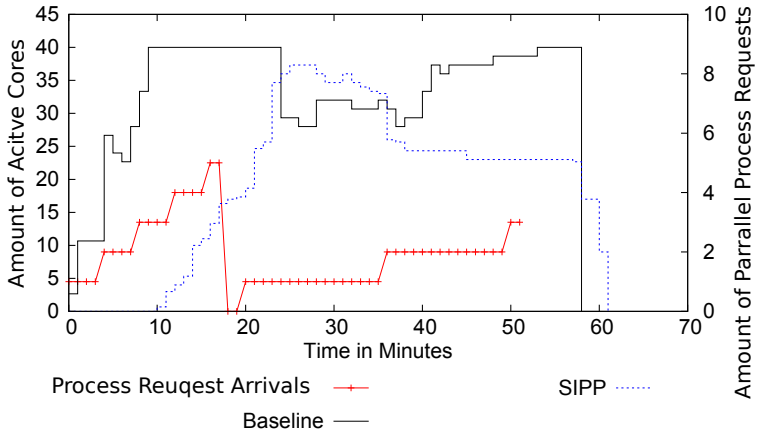
Figure 6.3: Evaluation Results – Constant Arrival

lease exactly as many resources as needed, while the baseline leases additional resources ad hoc, i.e., when a specific threshold has been reached.

The fact that by applying the baseline each time a new quad-core VM was leased, resulted in much higher cost, namely in average 41.66% higher than applying the SIPP model. Comparing the overall makespans, we see that the difference between lenient and strict SLA levels for the baseline can be neglected, i.e., 22.67 vs. 25.67 minutes. Comparing these numbers, we see that the baseline was even faster than our approach. This is due to the fact that leasing a quad-core VM allows much more service invocations at a certain point of time. However, due to the limited amount of requested process instances, the leased resources were not fully utilized, but have to be paid for the full BTU. This can be seen in Figure 6.3a and Figure 6.3b around minute 25, i.e., the amount of leased CPUs literally falls from 40 to 0, compared to our approach, where releasing



(a) Strict SLA



(b) Lenient SLA

Figure 6.4: Evaluation Results – Pyramid Arrival

resources follows more a step-like function.

Second, we discuss the Pyramid Arrival scenario (see Table 6.6). Again, using our approach, we achieve a close to 100% SLA adherence, thus we can assume that the scheduling plan created by the SIPP model considered all given SLAs and assigned the service invocations accordingly. The fact that the baseline with a strict SLA level ended up in more than 30% SLA violations, can be linked to the leasing and releasing policy of cloud-based computational resources, i.e., in the baseline scenario, cloud-based computational resources are leased or released based on a threshold. Comparing the baseline with our approach with regard to the time, the overall makespan for our approach is lower in every case, however, less resources are needed as our SIPP is able to distinguish what types of VMs are needed, i.e., whether a single-, dual-, triple-, or quad-core Backend VM should be leased.

Table 6.5: Evaluation Results – Constant Arrival

Arrival Pattern	Constant Arrival			
	SIPP		Baseline	
SLA Level	Strict	Lenient	Strict	Lenient
Number of total Process Requests	50			
Interval between the Process Request	120 seconds			
Number of parallel Process Requests	$y = 5$			
SLA Adherence in % (**Standard Deviation**)	94.0 (σ=2.0)	96.0 (σ=2.0)	68.66 (σ=18.33)	92.66 (σ=1.55)
Total Makespan in Minutes (**Standard Deviation**)	23.33 (σ=0.58)	31.33 (σ=4.51)	22.67 (σ=1.15)	25.67 (σ=2.89)
Max. Active Cores (**Standard Deviation**)	37 (σ=4.36)	22.33 (σ=7.36)	40 (σ=6.06)	36 (σ=3.19)
Leasing Cost (**Standard Deviation**)	1136.0 (σ=80.88)	780.0 (σ=78.0)	1738.33 (σ=20.21)	1493.33 (σ=40.41)
Penalty Cost (**Standard Deviation**)	10.0 (σ=3.61)	5.33 (σ=4.93)	42.67 (σ=15.31)	7.0 (σ=4.36)
Total Cost (**Standard Deviation**)	1146.0 (σ=82.66)	785.33 (σ=81.13)	1781.0 (σ=18.74)	1500.33 (σ=38.89)

Notably, the SLA adherence for the strict and lenient SLA levels differ for the SIPP. The evaluation shows that applying stricter SLAs requires more resources in less time. This can perfectly be seen in Figure 6.4a-6.4b. Since a higher-cored VM is comparably cheaper than several lower-cored ones, they are more likely to be leased when a strict SLA is applied (see Figure 6.4a) as more invocations have to be processed in shorter time. Since this enables a faster invocation of the process instances, we observe a smaller number of SLA violations. Comparing this to the lenient SLAs (see Figure 6.4b), we see that less resources are acquired at the beginning. Due to that, and since our services mirror real-world services where the CPU load and execution time is not deterministic, we experience a few SLA violations ($\sim$1%), which is acceptable by the definition of our SIPP model.

While we can not compare the numbers from this evaluation directly with the results from the evaluation runs from the former chapters, we can see that following a reasoning only approach (i.e., from Chapter 4) we could save -3.79% cost and were approximately 6.71% minutes faster than the baseline. In our later approach, i.e., a simple optimization approach realized using an heuristic for sequential elastic processes (i.e., from Chapter 5), we were able to save in average 19.56% cost and were 24.13% faster than the baseline. Within this chapter we achieved even better values: By using the SIPP approach, we were able to reduce the overall cost by 41.66% for the constant arrival pattern and 36.86%

Table 6.6: Evaluation Results – Pyramid Arrival

Arrival Pattern	Pyramid Arrival			
	SIPP		Baseline	
SLA Level	Strict	Lenient	Strict	Lenient
Number of total Process Requests	100			
Interval between the Process Request	60 seconds			
Number of parallel Process Requests	$f(n)$ (see (6.34))			
SLA Adherence in %	100.0	99.67	67.33	96.33
(Standard Deviation)	(σ=0.0)	(σ=0.58)	(σ=3.06)	(σ=0.58)
Total Makespan in Minutes	58.67	57.67	57.0	53.33
(Standard Deviation)	(σ=2.52)	(σ=0.58)	(σ=0.0)	(σ=1.15)
Max. Active Cores	40	37.33	40	40
(Standard Deviation)	(σ=10.83)	(σ=10.76)	(σ=7.79)	(σ=7.79)
Leasing Cost	2339.33	2176.33	3628.33	3430.0
(Standard Deviation)	(σ=226.89)	(σ=268.59)	(σ=40.41)	(σ=121.24)
Penalty Cost	0.0	0.33	88.0	7.33
(Standard Deviation)	(σ=0.0)	(σ=0.58)	(σ=18.36)	(σ=3.21)
Total Cost	2339.33	2176.67	3716.33	3437.33
(Standard Deviation)	(σ=226.89)	(σ=268.78)	(σ=37.55)	(σ=124.42)

for the pyramid arrival pattern. However, the baseline achieved a shorter total makespan (in average 9.18% faster). Hence, it can be deducted that leasing of additional resources in an ad hoc manner may end up in faster process enactments, but will also result in higher cost, since resources are always leased for a defined BTU. Thus, the resources have to be paid, but may not be needed later on.

In addition, our evaluations have shown that using an optimization model for creating a scheduling plan for complex process patterns will create much lower cost than using an ad hoc approach. Although a MILP solver is used to find an optimal solution to such problems, having real-world factors in the evaluation, such as a fluctuation of resource usage, the overall outcome will always be affected. However, the fact that we experienced less than 5% of SLA violations and an average cost improvement of almost 40% definitely shows that using an optimization model for scheduling service instances and their invocations among cloud-based computational resources should always be preferred over an ad hoc approach.

6.5 Conclusion

Using cloud-based computational resources to enact business processes seems to be an obvious choice. However, there is still a lack of BPMS frameworks which are able to lease and release resources for process enactment in a cost- and time-efficient manner. Within this chapter we made the following contributions:

- We defined a system model needed for complex elastic process enactments, i.e., a system model taking into account sequences as well as complex process patterns (XOR-blocks, AND-blocks and Repeat loops). In addition, we considered penalties and the BTU.
- We defined the SIPP, a multi-objective optimization model to achieve complex process scheduling. The SIPP model is meant to minimize the total cost arising from resource leasing cost and potential penalty cost.
- We presented ViePEP 2.0 and integrated the SIPP.
- We evaluated our approach extensively and showed its advantages over ad hoc-based resource scalling and process scheduling.

In the next chapter (Chapter 7) we will extend this model and consider the problems and challenges which apply when enacting complex elastic processes in an hybrid cloud environment.

CHAPTER 7

Cost-based Optimization for Complex Elastic Processes in Hybrid Clouds

So far, we presented two different approaches for utilizing cloud-based computational resources in order to enact sequential elastic processes (see Chapter 4 and Chapter 5). We have shown that using a basic scheduling algorithm can achieve cost savings compared to ad hoc scaling baselines. However, since sequential processes are only a small subset of real-world business processes we considered in the last chapter (see Chapter 6) complex elastic processes and presented a multi-objective optimization model called the SIPP. By solving this model, we were able to use cloud-based computational resources for complex elastic process enactment in a cost-efficient way. In order to evaluate this model, we created ViePEP 2.0 (Section 6.4.1). So far, only a single cloud provider has been considered. However, leasing resources only from one cloud provider is rather restrictive as cloud providers may offer different resources with various configurations to different prices. By allowing ViePEP to lease resources from a hybrid cloud, i.e., a combination from private as well as public clouds, more different resources (i.e., different VM types for different prices) can be leased which could lead to a better overall performance and decreases the risk of vendor lock-in. However, scheduling business processes in a hybrid cloud environment is more complicated as several additional aspects need to be considered.

Hence, in this chapter we substantially extend the work from Chapter 6 by the following:

- We extend the system model for complex elastic process scheduling in order to be able to lease resources and schedule services invocations in a hybrid cloud.

- Doing this, we extend the SIPP model and consider *data transfer time* and *data transfer cost*.
- We extend ViePEP 2.0 in order to allow leasing resources from multiple cloud providers.
- We implement the extended SIPP model into ViePEP 2.0.
- We evaluate this new approach extensively and compare its results against a baseline following ad hoc-based scaling.

The remainder of this chapter is structured as follows: We start with a motivational scenario in Section 7.1 and state different prerequisites for our approach in Section 7.2. Afterwards, we present our scheduling approach in Section 7.3. In Section 7.4, our solution is evaluated using a testbed-driven approach using ViePEP 2.0 (see Section 6.4.1). Last but not least, Section 7.5 concludes the chapter.

7.1 Example Scenario

In this section, we provide an example scenario to motivate the contribution presented in this chapter. As in Section 2.3.3, we consider a scenario from the financial industrial sector [64], e.g., an international bank. Notably, the magnitude of the process landscape and the strict process execution deadlines can be also found in other domains, e.g., the manufacturing domain [96] as the manufacturing industry is currently undergoing a massive transition towards more effective and interconnected factories [29].

The considered bank features several worldwide distributed branches, i.e., from the US East Cost over Europe to Asia. Each branch of this bank provides similar products to their customers, hence, the bank maintains a private cloud spanning all data centers of the single branches. In addition, each branch has access to similar business process models which are used within the bank's products and services. The single branches can lease resources from this private cloud for a particular time span (BTU) and execute their business processes. Resources need to be paid according to the BTU, hence, a major objective is to achieve cost-efficiency. This includes an optimal process enactment of the corresponding business processes and an optimal resource leasing policies.

The available business processes contain different services which range from short, but resource intensive operations to long-running analytic processes. Figure 7.1 represents one exemplary business process which is composed of five steps (Step1 – Step5). In order to enact this process model, five corresponding software-based services need to be invoked (S1 – S5). The services deal with complex computations and their results have to be transferred from one service to the next one. This transfer requires time that has to be considered by the BPMS.

The maintained private cloud provides computational resources which are sufficient to instantiate all software-based services in off-peak times. However, during peak times, when there are numerous parallel process enactments, it is required to lease additional

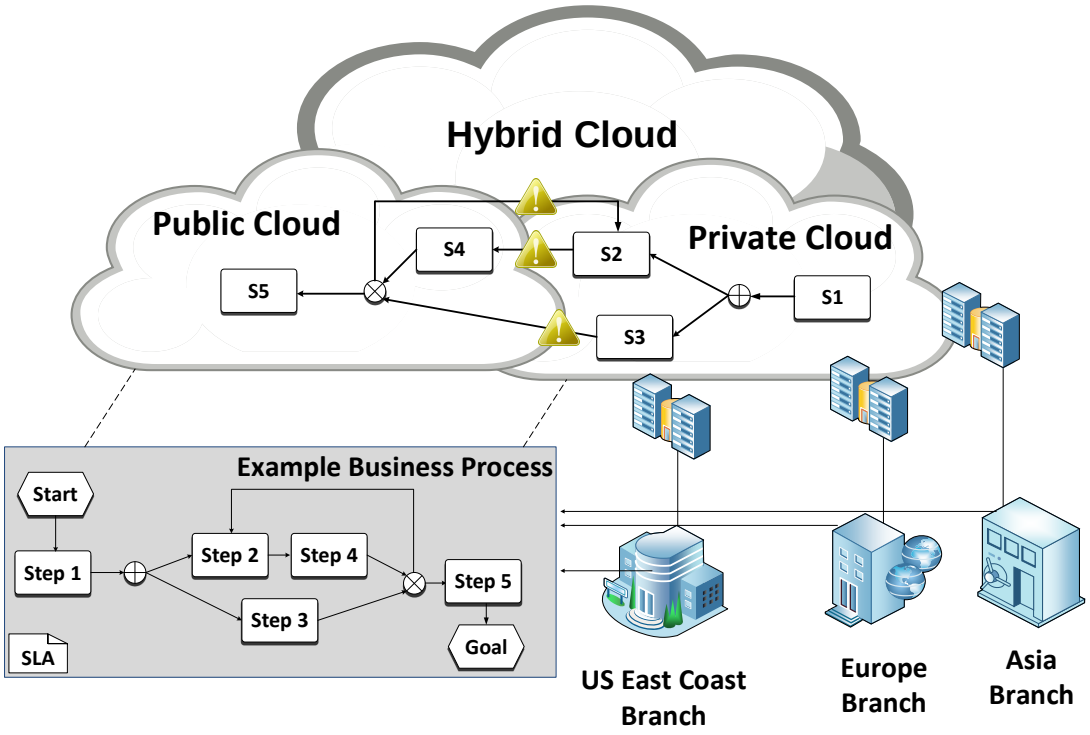


Figure 7.1: Example Scenario

resources from a public cloud to cope with the resource requirements for the services. Figure 7.1 represents a snapshot regarding the resource allocation for a peak time scenario, where services S1, S2 and S3 are deployed using cloud-based computational resources leased from the private cloud and services S4 and S5 are deployed on resources leased from the public cloud since there are too little resources on the private cloud. The data transfer capabilities within one cloud are very good, so that data transfer of date only requires a short period of time. In contrast, the transfer capabilities between the private cloud and the public cloud are limited, since they are routed over the Internet. These inter-cloud data transfers also issue data transfer cost [1]. They are marked with the exclamation marks in Figure 7.1 and the goal is to reduce or avoid these costly data transfers during process enactment. Since the in-time completion of the process executions is vital for a cost-efficient process enactments, each process execution is assigned with a deadline and when a process execution does not meet this deadline, penalty cost accrue.

7.2 Preliminaries

Based on the example scenario, we define some preliminaries to realize a data transfer aware scheduling and resource provisioning approach in hybrid clouds. We extend in this chapter the SIPP model from Section 6.3, hence, we refer to the preliminaries from

Section 6.2. Notably, altough the original SIPP model was conceptualized for a private cloud setting and so far neither considers hybrid clouds nor any data transfer aspects, the underlying preliminaries remain the same for the extension mentioned in this chapter.

As already stated in the example scenario, we consider a hybrid cloud that is composed of a private cloud and a public cloud hosted in two geographically different locations. The BPMS is deployed in the private cloud and covers both the functionalities of a BPMS and of a cloud controller. Although both, the private and the public cloud, are capable of hosting all service types, we assume that the private cloud should be utilized first as its running cost, e.g., energy cost or maintenance cost, are cheaper compared to the leasing cost for the VMs in the public cloud. Therefore, the natural goal of the optimization model is to achieve a high resource utilization of the private cloud before leasing VMs from the public cloud. In general, we only consider inter-cloud data transfer in terms of data transfer cost, as intra-cloud data transfers are free of charge [1]. However, intra-cloud data transfer also takes some time, which has to be considered. Further, we assume that every service instance maintains its own data repository and that data is only transferred on a process step to process step communication basis.

7.3 Process Scheduling in Hybrid Clouds

Based on the discussed preliminaries, we are now able to define our elastic process scheduling approach. For this, we first present how to extend the SIPP model from Section 6.3.2. Afterwards, we extend the scheduling problem to also support hybrid clouds while considering data transfer aspects.

7.3.1 Extending the Service Instance Placement Problem (SIPP)

To execute elastic processes, software services are deployed on cloud-based computational resources, i.e., VMs. Companies often suffer from the provisioning problem, i.e., the challenge to lease enough resources to handle peak loads but prevent over-provisioning during off-peak times. Hemce, the challenge is to lease as little resources as needed while still satisfying a certain level of QoS. In order to satisfy these aspects, we defined a system model and an optimization model which is aiming at minimizing the total *leasing cost* for cloud-based computational resources, potential *penalty cost* (see Section 6.3.1) and through our extension: *data transfer cost*. Besides of minimizing the cost, the most important outcome of the optimization problem is an assignment of service invocations to computational resources, i.e., VMs, in an optimal manner. The SIPP model, including all its constraints, variable definitions and utility functions is described in more detail in Section 6.3 and a table of used variables can additionally be found in Table 6.2.

7.3.2 Optimization Model

$$
\begin{aligned}
\min \; & \sum_{v \in V} c_v \cdot \gamma_{(v,t)} \\
& + \sum_{p \in P} \sum_{i_p \in I_p} c^p_{i_p} \cdot e^p_{i_p} \\
& + \sum_{v \in V} \sum_{k_v \in K_v} (\omega^C_f \cdot f^C_{k_v} + \omega^R_f \cdot f^R_{k_v}) \\
& - \sum_{p \in P} \sum_{i_p \in I_p} \sum_{j_{i_p} \in J^*_{i_p}} \frac{1}{DL_{i_p} - \tau_t} x_{(j_{i_p}, k_v, t)} \\
& + C_D
\end{aligned}
\tag{7.1}
$$

Equation (7.1) shows the main objective of the extended SIPP model. The main objective function comprises five terms. The first four terms form are from the original SIPP model from Section 6.3.2 and are shortly summarized in the following. The fifth term, i.e., C_D is an extension to the SIPP, i.e., this term is dedicated to enable and optimize hybrid cloud deployment of service instances and their invocations, which is explained in detail in Section 7.3.3.

The first term in (7.1) computes the total cost accruing due to leasing $\gamma_{(v,t)}$ VMs of type v in a certain leasing period τ_t. The second term computes the cost which may accrue if deadlines are violated, i.e., penalty cost which have to be paid for delayed process instances. For computing these penalties we apply a linear penalty function as described in [69]. This means that each process instance includes defined penalty cost $c^p_{i_p}$ which are due if the process instance gets delayed. This value is multiplied with the actual period of time $e^p_{i_p}$ for which the process instance is delayed. The third term in the equation is used to consider the cost of *wasted* resources, i.e., the sum of free resource capacities $f^C_{k_v}$ and $f^R_{k_v}$ in terms of CPU and RAM. The fourth term is used to compute the urgency of process instances, i.e., we subtract the current period in time τ_t from the deadlines DL_{i_p} and compute the corresponding reciprocal value. This way, a process instance having a closer deadline is assigned with a higher priority since the value $\frac{1}{DL_{i_p} - \tau_t}$ gets larger.

7.3.3 Scheduling Elastic Processes in Hybrid Clouds

Usually, private clouds provide a fixed amount of computational resources and therefore suffer from over- and under-provisioning. Having this problem in mind, many companies choose to extend their private cloud with a public cloud to be able to lease virtually unlimited additional resources to deploy resource-intensive services [13, 106].

The proposed optimization model from (7.3.2) is able to optimize the scheduling and placement of service invocations on computational resources provided by a single cloud (independent from the fact whether it is either a private or a public cloud). However, the support of hybrid clouds raises additional challenges especially in terms of data transfer, i.e., the duration needed to transfer data and the cost which accrue when a large amount

of data is moved from or into a cloud. In order to address these challenges, we extended the original SIPP model in (7.1) with the fifth term C_D. This, and additional constraints and utility functions (Equations (7.2)-(7.6)) are explained in the following.

Data Transfer Cost

First we consider the data transfer cost, which are represented by the new variable C_D (see (7.4)). This variable represents all data transfer cost which are issued by the inter-cloud data transmission between currently running respectively already finished process steps and future process steps that are currently considered in the scheduling plan.

To distinguish between private cloud VMs and public cloud VMs, we extend the concept of the VM variable K_v: We introduce two subtypes of this variable, $K_{v_{priv}}$ which represents VMs in the private cloud and $K_{v_{pub}}$ which represents VMs in the public cloud. Since the original system model of SIPP does not consider any data transfer between two sequenced process steps, we introduce the additional variables: $j_{i_p}^{past} \in J_{i_p}^{past} = \{1, ..., j_{i_p}^{past^{\#}}\}$. Variable $J_{i_p}^{past}$ represents the set of all process steps, and $j_{i_p}^{past}$ one particular process step, which is/are currently running or have already been completed to determine whether two adjoining process steps issue any data transfer cost. Two successive process steps can follow four different deployment scenarios: In the first two scenarios both process steps are either deployed in the private cloud or in the public cloud. As intra-cloud data transfer is free of charge, no data transfer cost will accrue within this scenario. In contrast to that, in the two remaining deployment scenarios, the predecessor step can either be deployed in the private cloud and the successor step in the public cloud or vice versa. Within this scenario, data transfer cost will accrue and have to be considered.

The actual deployment situation is evaluated using the utility function $l(j_{i_p}^{past}, j_{i_p}^{*})$ (see (7.2)). In case a scheduling plan consists of a deployment spanning across private and public clouds, we assign data transfer cost based on the output result size of the first task, since hosting providers usually only charge for outgoing data transfer (see (7.3)). In any other case, we assign no data transfer cost. These individual transfer cost $C_{D_{j_{i_p}}}^{\#}$ are accumulated by an additional constraint (see (7.4)) to be considered for the overall optimization objective (see (7.1)) as C_D.

$$l(j_{i_p}^{past}, j_{i_p}^{*}) = \begin{cases} 0, & \text{if } k_1, k_2 \in K_{v_{priv}}, x_{(j_{i_p}^{past}, k_1, t)}, \\ & x_{(j_{i_p}^{*}, k_2, t)} \\ 0, & \text{if } k_1, k_2 \in K_{v_{pub}}, x_{(j_{i_p}^{past}, k_1, t)}, \\ & x_{(j_{i_p}^{*}, k_2, t)} \\ 1, & \text{else} \end{cases} \tag{7.2}$$

$$C^{\#}_{D_{j_{i_p}}} = \begin{cases} C_{D_{j_{i_p}}} & \text{, if } l(j^{past}_{i_p}, j^{*}_{i_p}) = 1 \\ 0 & \text{, else} \end{cases} \tag{7.3}$$

$$\sum_{v \in V} \sum_{k \in K_v} \sum_{j_{i_p} \in J_{i_p}} x_{(j_{i_p}, k_v, t)} * C^{\#}_{D_{j_{i_p}}} \leq C_D \tag{7.4}$$

Collocation of Service Invocations

The amount of accruing data transfer cost depends on the service type in combination with the deployment location, i.e., whether it is deployed in the public or private cloud. Using this information, the collocation of successive service invocations is realized based on the data transfer cost constraints in an implicit manner, as the objective function aims at minimizing the overall cost.

Selective Usage of Public Resources

To implement the conditional usage of the public cloud, we introduce the utility function u (see (7.5)) and the functionality to assign the future leasing cost for the VMs leased from the public cloud based on the current usage of the private cloud. The utility function u sums up the cores of all currently leased private VMs. In order to have a VM-independent resource usage calculation function, the usage calculation is carried out on the basis of VM cores. As long as the usage of the private cloud is below 70%, the leasing cost of the public VMs are multiplied with 100 to make them a non-desirable option for the scheduling plan (see (7.6)). The concrete figure of 70% was chosen in order to achieve a high resource utilization for the private cloud, while maintaining necessary reserves in case of VM failures to achieve a data transfer optimized scheduling plan, even if some VMs have to be replaced [125].

$$u = \sum_{k_v \in K_{v_i}} core(g(k_{v_{priv}}, t)) \tag{7.5}$$

$$c_{v_{ex}} = \begin{cases} c_{v_{ex}} & \text{, if } u > \left(\sum_{k_v \in K_{v_i}} core(k_{v_{priv}}) \right) * 0.7 \\ c_{v_{ex}} * 100 & \text{, else} \end{cases} \tag{7.6}$$

This constraint does not completely rule out the usage of VMs provided by the public cloud, since the scheduling plan may require a specific type of a VM, which is currently not available in the private cloud and an expensive VM may be a better option than paying penalty cost. Nevertheless, since the cost are higher than for the private resources, the selection of such an expensive VM is unlikely. In general, we assign leasing cost for the private cloud as well as the public cloud. The cost structure for the private cloud enables us to implement a fine-grained selection of different VM types to use the private cloud as efficient as possible.

7.4 Evaluation

For evaluating our scheduling and provisioning approach we made use of ViePEP 2.0 from Section 6.4.1. As ViePEP was originally designed for a single private cloud environment, we implemented an extension allowing to lease VMs from an Openstack-based cloud[1] as well as from Amazon EC2 (AWS)[2]. While the private cloud is restricted in terms of computational resources, the public cloud provides virtually unlimited resources. The BTU for both clouds is set to 5 minutes.

On the one hand ViePEP operates on a PaaS level and accepts new process instance requests and on the other hand it operates on the IaaS level to lease and release computational resources in order to enact the requested process instances. Further, ViePEP makes use of IBM CPLEX [3] for solving the SIPP model. The result is a full scheduling and resource allocation plan, i.e., a scheduling plan telling on which VM instance a particular service instance has to be invoked in order to fulfill certain steps of a particular process instance.

In the following subsections, we first present the evaluation setup in Section 7.4.1. Afterwards, we present the quantitative evaluation including a discussion in Section 7.4.2.

7.4.1 Evaluation Setup

Process Models

We select 10 representative process models from the SAP reference model [25] to perform realistic evaluations. Notably, these are the same 10 process models as used in Section 6.4.2. The 10 selected process models can be found in Appendix B. The SAP reference model has been evaluated and used for many scientific papers, e.g., [80] and provides a solid basis for our evaluation. The selected process models are composed from different process patterns and different levels of complexity, i.e., the process models may be composed of AND-block, XOR-block or Repeat loops. The former two types consist of a split (AND, XOR) and the appropriate merge pattern, which either is blocking (AND-block) or simply continues the process execution, as soon as 1 optional process step is completed (XOR-block).

Test bed

As in our previous evaluation (Section 6.4) we simulated 10 different software services, each needing a different amount of computational resources and makespan. In this chapter we extend our approach by considering data transfer time and cost. In order to *force* service deployment on computational resources leased from the public cloud as well, we updated the simulated services. Hence, the simulated services differ from the services used in Section 6.4.

[1] http://www.openstack.org/software/folsom/

[2] http://aws.amazon.com/ec2/

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

Table 7.1: Service Types

Service Type No.	CPU Load in % (μ_{cpu})	Makespan in sec. ($\mu_{makespan}$)	Outgoing data cost
1	50	30	2
2	75	80	4
3	75	120	6
4	100	20	2
5	120	100	4
6	125	30	6
7	150	40	2
8	175	20	4
9	250	60	6
10	310	30	2

Table 7.1 provides a detailed overview about the 10 simulated service types. The table states the required mean CPU load in percent, the mean makespan in seconds, and the size of outgoing data in cost units. The services' CPU load describes the required amount of CPU in percent on a single-core VM. If a service is deployed on a multi-core VM, the load is equally distributed among all of them, i.e., we assume that each service is fully parallelizable. This leads to the fact that only the first 3 services of Table 7.1 can be deployed on a single-core VM. All other services require at least a dual-core VM or an even larger one. Notably, with an increasing number of CPU cores, the makespan remains the same. For generating the CPU load, we use the lookbusy load generator[4]. This is a configurable tool to generate CPU load on a VM for a given timespan and enables us to simulate the resource consumption. The values μ_{cpu} and $\mu_{makespan}$ provided in Table 7.1 represent mean values of 2 general normal distributions where we assume $\sigma_1 = \mu_{cpu}/10$ respectively $\sigma_2 = \mu_{makespan}/10$. ViePEP is able to lease 7 different VM types: 4 VM types from the private Openstack-based cloud, and 3 VM types from Amazon EC2 public cloud.

Applied SLAs

As in our former evaluation, we applied each process instance with a SLA which defines the process' deadline, i.e., the latest point of time when the process instance has to be finished. To evaluate the applicability of our approach, we choose two different SLA scenarios as used in Section 6.4. First, we apply a *strict* scenario, i.e., the deadline is defined by 1.5 times the process model's average makespan. Second, we apply a *lenient* scenario, i.e., the deadline is defined by 2.5 times the process model's average makespan. The two values were chosen due to the fact that VM instances require some time for the deployment of the services and individual VM instances may fail, so that ViePEP has to lease replacement VMs which affects the overall process execution negatively.

[4] `http://devin.com/lookbusy/`

Process Request Arrival Pattern

Again, we apply two different process request arrival patterns: First, we follow a *constant* request pattern, for which we select 7 process models every 120 seconds. In order to vary the process models, we create different instances based on the round-robin principle, i.e., in the first round process models 1 to 7 are requested, in the next round 8 to 10 and 1 to 4, and so on. This is repeated until a total of 70 process instances are requested in a time frame of about 20 minutes.

The second request pattern follows a *pyramid*-like function, which is represented by (7.7). Variable a represents the amount of process instance requests at the single points of time n every 120 seconds. Notably, this request pattern differs from the pyramid arrival pattern from Section 5.4.1: we changed this arrival pattern in order to lease resources from both clouds, the private as well as from the public cloud. The actual process selection aims for a realistic distribution among common and rare process models. To achieve this variable distribution, we start for every new request batch with process number 1 and increment the process number to match the amount of process requests. The second request pattern issues in total 118 process instances within a time frame of 48 minutes.

$$f(n) = \begin{cases} a = n & \text{if } 0 \leq n \leq 5 \\ a = n/2 & \text{if } n = 6 \\ a = 1 & \text{if } 7 \leq n \leq 10 \\ a = n - 10 & \text{if } 11 \leq n \leq 19 \\ a = 10 & \text{if } 20 \leq n \leq 24 \end{cases} \tag{7.7}$$

Baseline for the Evaluation

As in our former evaluations (Section 4.5-6.4), we compare our evaluation results against a *baseline* which follows a basic ad hoc-based scheduling and resources provisioning strategy [36]: As soon as a service instance needs to be invoked and no VM instance hosting a specific service is available, a new VM will be instantiated. In this evaluation, a new quad-core VM is leased and the corresponding service type is deployed. In addition, to prevent under-provisioning, an additional quad-core VM will be leased and the needed service type will be deployed, if a VM instance's resource utilization level reaches 80%. If a VM instance's resource load is below 20%, the VM instance will be released as soon as all service invocations on this VM have been finished. Within the baseline, VMs are equally instantiated from the public as well as from the private cloud without considering any data transfer aspects. Nevertheless, all restrictions applied to the SIPP model, like only 1 service type per VM, QoS constraints, i.e., the given deadline, and the issued cost are also applied for the baseline approach.

Metrics

To assess the performance of our scheduling approach, we apply similar metrics as used prior in Section 6.4: First, the *Total Makespan in Minutes* is measured, i.e., the overall timespan from the first process instance request until the successful termination of the last process step of the last process instance. Second, the *Total Cost* are computed. For that, we first compute the amount of maxium leased CPU cores (*Max. Active Cores*). Second, contrary to our former evaluations, the total cost are now composed from *Private Leasing Cost* representing the leasing cost for the private cloud, *Public Leasing Cost* which represent the leasing cost for the public cloud and *Data Transfer Cost* which are charged for transferring data across the different clouds. In addition, the Total Cost also include *Penalty Cost* which accrue if a process instance is delayed. For that, we apply according to the delay, 1 cost unit penalty for every 10% of the overall makespan of the process model. We further assess the *SLA Adherence*, i.e., the percentage of process executions which are compliant with their SLA.

To receive significant data, we perform three iterations for each process request pattern combined with the two different SLA levels resulting in a total of 12 runs for our approach as well as 12 for the baseline.

7.4.2 Results and Discussion

The following section presents and discusses the results of our evaluation by listing all key metrics in Table 7.2 for the constant request pattern and in Table 7.3 for the pyramid request pattern. A graphical representation can be found in Figure 7.2 for the constant request pattern and in Figure 7.3 for the pyramid request pattern. Notably, the numbers in the table, as well as the numbers below in the discussion represent the mean value over the three runs. The standard deviation σ is also presented in the tables.

First we discuss the constant request pattern for both SLA scenarios (see Table 7.2): The most prominent difference between the baseline and our extended SIPP approach are the higher total cost for the baseline approach. Using the extended SIPP model we were able to achieve total savings in terms of cost of 61.38% for the lenient SLA scenario and for the strict scenario total savings of 45.70%. As it can be seen in Table 7.2, the leasing cost represent the major part of the overall cost. Overall, in our approach, less VMs from the public cloud were leased leading to significant smaller leasing cost. These results show that the selective usage of public resources works as intended and the extended SIPP only leases public resources when they are required to comply with the SLA. This has a large impact on the overall data transfer cost. The baseline issued about 14 times as much data transfer cost which represent almost a third of its total cost, i.e., 26% for the strict SLA scenario and 40% for the lenient one. Compared to our approach, where data transfer cost amounts to only 4.9% of the total cost for the strict scenario (and none for the lenient one). While observing the SLA adherence, we see that hardly any SLAs are violated in the constant request pattern. The fact that we experienced no SLA violations in the strict scenario for our approach, can be reduced to the fact that the overall deadlines are *close* which results in a higher risk of delay compared to the

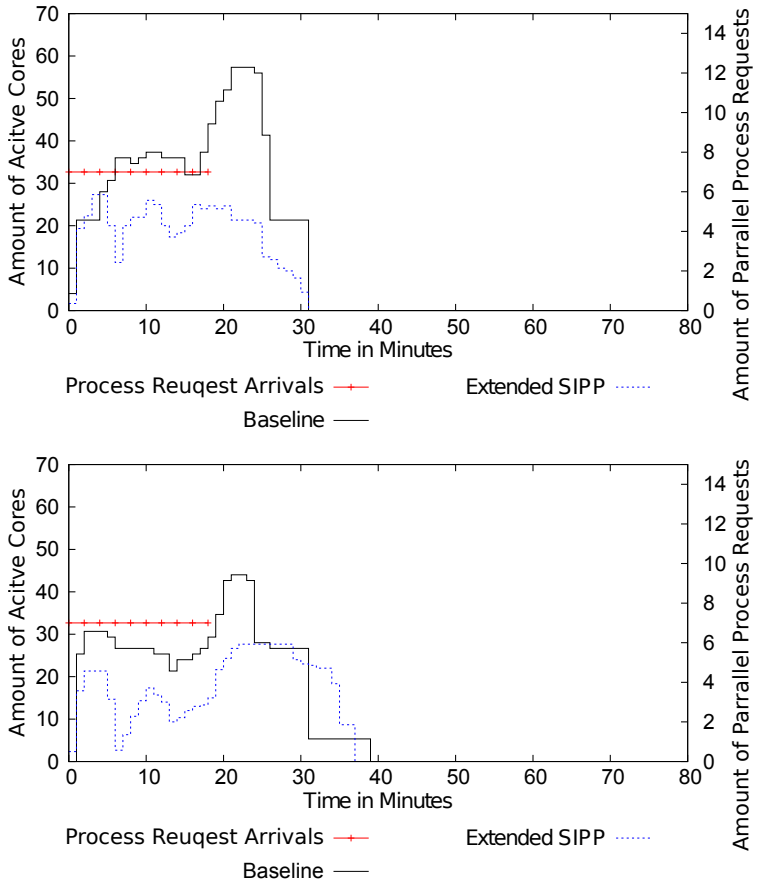


Figure 7.2: Evaluation Results – Constant Arrival

lenient scenario. As penalty cost are an important factor, closer deadlines *force* earlier scheduling. If a VM is already leased, the SIPP model aims at reducing *wasting* resources, hence, future process instances are pre-poned.

Next to the SLA adherence it remains to discuss the overall makespan: In general, the strict scenario finishes faster than the lenient one, and further, the baseline finishes faster than our approach. This can be reduced to the fact that the baseline leases more VMs, hence, has more resources available and is able to perform more service invocations in parallel. In contrast, the extended SIPP only leases additional resources when they are cost-efficient in respect to the penalty cost model. However, although the baseline was faster than the extended SIPP, it also results in much higher cost ($\sim$2.1 times).

Second, we discuss the pyramid request pattern where 48 additional process instances were requested (see Table 7.3). Due to the nature of this request pattern, both approaches had to lease in total more resources than in the constant request pattern leading to higher leasing cost of both, public VMs and private VMs. Interesting to see is the ratio between data transfer cost and leasing cost: For our approach, this ratio amounts to

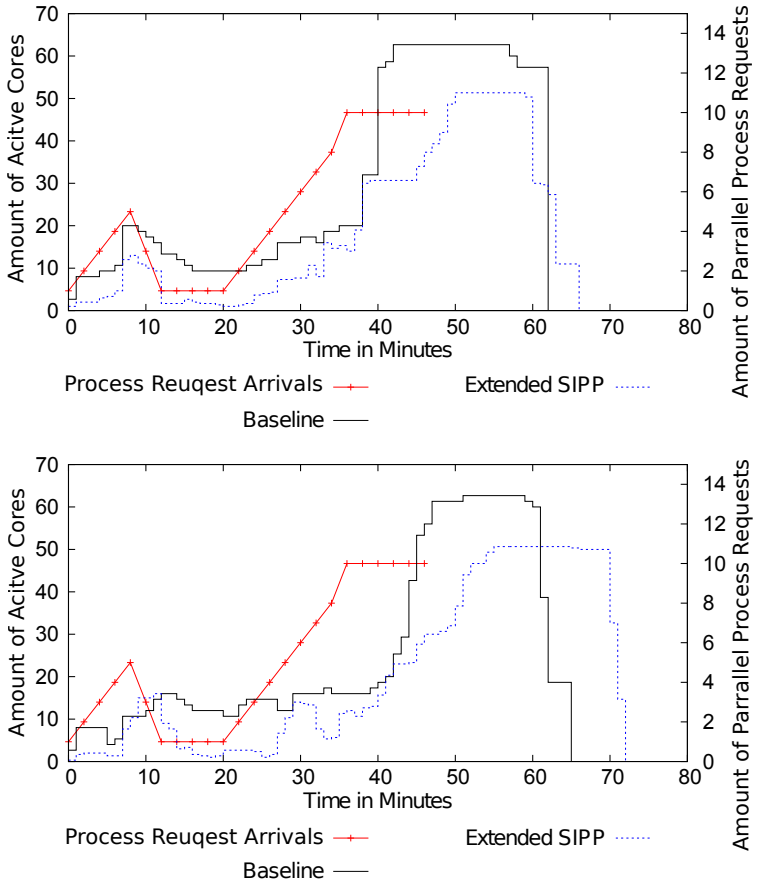


Figure 7.3: Evaluation Results – Pyramid Arrival

6.7% for the strict SLA and 5.1% for the lenient one. For the baseline it is ∼25% for both scenarios, i.e., for the strict SLA scenario and the lenient SLA scenario. This means that our approach preferred to schedule service invocations within the same cloud over switching between the public and private cloud. As the baseline does not consider any data transfer cost for its scheduling plan, it results in much higher cost. Due to the used penalty cost model, the overall SLA adherence is in general lower (∼57%-86%) compared to the constant request pattern (∼90%-100%) as penalty cost are comparably cheaper than leasing additional resources. The reason that we experienced more SLA violations using the extended SIPP model in the pyramid request pattern than for the constant request pattern, can be attributed to the fact that in total more process instances are processed.

The evaluations have shown that the usage of our approach for creating a scheduling plan which considers leasing cost, penalty cost, and data transfer cost significantly decreases the overall cost compared to an ad hoc approach. The numbers in Table 7.2 and Table 7.3 reveal that the extended SIPP approach was able to reduce the overall

Table 7.2: Evaluation Results – Constant Arrival

	Constant			
	Extended SIPP		Baseline	
SLA Level	Strict	Lenient	Strict	Lenient
Number of Process Requests	70			
Interval between two Process Requests	120 seconds			
Number of Parallel Process Requests	$y = 7$			
Total Makespan in Minutes (Standard Deviation)	26.33 (σ=2.30)	33.00 (σ=3.00)	23.67 (σ=1.15)	32.67 (σ=7.51)
Max. Active Cores (Standard Deviation)	27.33 (σ=0.81)	27,67 (σ=5.72)	57.33 (σ=16.5)	44.00 (σ=13,74)
SLA Adherence in % (Standard Deviation)	100.00 (σ=0.00)	90.00 (σ=7.95)	94.76 (σ=6.60)	100.00 (σ=0.00)
Penalty Cost (Standard Deviation)	0.00 (σ=0.00)	10.33 (σ=8.02)	15.00 (σ=24.24)	0.00 (σ=0.00)
Private Leasing Cost (Standard Deviation)	963.33 (σ=77.36)	1284.00 (σ=300.39)	1061.67 (σ=20.20)	385.00 (σ=121.24)
Public Leasing Cost (Standard Deviation)	45.00 (σ=0.00)	0.00 (σ=0.00)	945.00 (σ=242.49)	1061.67 (σ=80.83)
Data Transfer Cost (Standard Deviation)	52.00 (σ=13.85)	0.00 (σ=0.00)	724.00 (σ=187.06)	937.33 (σ=244.79)
Total Cost (Standard Deviation)	1060.33 (σ=91.22)	1294.33 (σ=308.39)	2745.67 (σ=474.00)	2384.00 (σ=204.38)

cost by an average of ∼53% for the constant request pattern and an average of ∼40% for the pyramid request pattern.

To recap, in our former evaluations using the original SIPP model (see Section 6.4.4) we achieved an overall cost reduction of 41.66% for the constant request pattern and 36.86% for the pyramid request pattern. If we compare these numbers to the results from this chapter (∼53% for the constant and ∼40% for the pyramid request pattern), we can see that using the SIPP model should always be preferred over ad hoc scaling when cost-efficiency should be achieved. However, in terms of overall makespan, the baseline was again faster than our approach (∼6.78%). Hence, we can conclude that leasing additional resources in an ad hoc manner will indeed lead to faster process enactments. However, the offside is that this leads eventually to higher cost.

Table 7.3: Evaluation Results – Pyramid Arrival

	Pyramid			
	Extended SIPP		Baseline	
SLA Level	Strict	Lenient	Strict	Lenient
Number of Process Requests	118			
Interval between two Process Requests	120 seconds			
Number of Parallel Process Requests	$f(n)$ (see Equation (7.7))			
Total Makespan in Minutes (Standard Deviation)	62.00 (σ=2.00)	67.33 (σ=0.59)	58.67 (σ=1.91)	61.67 (σ=2.21)
Max. Active Cores (Standard Deviation)	51.33 (σ=8.70)	50.67 (σ=11.19)	62,67 (σ=1.91)	62,67 (σ=1.91)
SLA Adherence in % (Standard Deviation)	57.91 (σ=4.67)	76.27 (σ=5.29)	60.92 (σ=5.06)	86.72 (σ=3.42)
Penalty Cost (Standard Deviation)	155.33 (σ=34.43)	58.00 (σ=19.28)	123.33 (σ=15.00)	27.33 (σ=9.87)
Private Leasing Cost (Standard Deviation)	1793.00 (σ=47.70)	1934.67 (σ=199.58)	1458.33 (σ=253.20)	1540.00 (σ=185.20)
Public Leasing Cost (Standard Deviation)	420.00 (σ=108.17)	548.33 (σ=137.69)	1796.67 (σ=225.02)	1563.33 (σ=359.21)
Data Transfer Cost (Standard Deviation)	170.67 (σ=94.77)	136.67 (σ=28.94)	1184.67 (σ=193.67)	1051.33 (σ=204.92)
Total Cost (Standard Deviation)	2539.00 (σ=111.53)	2677.67 (σ=216.15)	4563.00 (σ=153.27)	4182.00 (σ=481.38)

7.5 Conclusion

Within this chapter we have shown that considering data transfer aspects (data transfer cost and data transfer time) while creating a scheduling plan for hybrid clouds can heavily reduce the total cost. We made the following contributions in this chapter:

- In order to consider *data transfer time* and *data transfer cost* in the SIPP model, we extended first the underlying system model.
- Afterwards, we extended the SIPP model in order to reduce or avoid data transfer cost.
- We extended ViePEP 2.0 in order to allow leasing resources from multiple cloud providers.
- We implemented the extended SIPP model into ViePEP 2.0.

- We evaluated the extended SIPP model and compared the results against a baseline.

Our evaluations have shown that our optimization model is able to schedule service invocations among hybrid cloud-based computational resources in a cost-efficient way. It also does not just reduce the leasing cost but also reduces the data transfer cost compared to an ad hoc baseline. Using the extended SIPP model we achieved total cost savings of up to 40%-53%, depending on the process requests.

However, our evaluations have also shown that the cost reduction depends on the underlying cost model, e.g., we experienced some SLA violations since it was *cheaper* to accept a delay instead of leasing additional resources. Based on this insight, we plan to use different cost models for penalty cost, data transfer cost and leasing cost in our future work. In addition, up to now we focused on process step to process step communications, i.e., the output of one process step is directly used as input for the next process step. However, real-world use cases often incorporate more complex data patterns, like shared global storage systems. Future evaluations will consider these complex data patters, a larger set of services as well as different cloud providers.

Another important research area for hybrid clouds are privacy restrictions for service types or the processed data. These privacy restrictions could either be issued by the legislation for sensitive data or are based on precautions not to deploy sensitive trade secrets, e.g., algorithms or customer data, on a public cloud. These constraints require an extended set of SLAs which have to be considered. For a more fine grained discussion about future extensions, pleas refer to Section 8.3.

CHAPTER 8

Conclusions and Future Work

In this final chapter we summarize the main results of this thesis. Section 8.1 presents the general outcome of the conducted research and describes how the state of the art was advanced. In Section 8.2 we revisit the research questions from Section 1.2 and analyze them. Finally, Section 8.3 concludes this thesis and discusses possible future work on open topics.

8.1 Summary

In this thesis we presented different approaches for business process enactment using cloud-based computational resources, i.e., we were aiming at realizing elastic processes. The main contributions in this thesis were implemented in ViePEP. For that, ViePEP has been completely re-designed and re-implemented, resulting in two new versions, i.e., ViePEP 1.1 (see Section 4.2) and ViePEP 2.0 (see Section 6.4.1). ViePEP is a BPMS designed according to the MAPE-K cycle which consists out of monitoring, analyzing, planning and executing phase. This MAPE-K cycle has been slightly adapted in order to fit our needs. The main contributions in this thesis can be assigned to the planning phase:

First, we created a novel approach for using cloud-based computational resources to enact business processes. On the one hand, the main driving factor was to avoid over- and under-provisioning of the leased resources at any time. On the other hand, it was important to provide a certain level of QoS and to ensure SLA adherence. Hence, our first contribution is that we assigned cloud-based computational resources to process executions. As we experienced that even the smallest VM might not be fully utilized at all time, we had a "look into the future", i.e., we considered future process executions and pre-poned the corresponding service invocations in order to fully utilize the leased resources. This allowed us to save cost and reduce the overall process execution time.

Second, we created an optimization model and transformed it into a heuristic-based algorithm which is able to schedule sequential elastic processes and to assign their service

invocations to cloud-based computational resources. The main outcome here is two-fold: (i) a detailed leasing plan which gives information about how many resources are needed at what point of time, and (ii), a detailed scheduling plan which assigns service invocations to the leased resources. The heuristic is able to consider different priorities of queued process instances according to the defined deadlines.

Third, we created a multi-objective optimization model which is able to realize complex elastic process executions on cloud-based computational resources. This model is called Service Instance Placement Problem (SIPP) and, as in the former contribution, its outcome is a detailed scheduling and resource allocation plan. By the nature of complex business processes, this optimization model has to consider different process patterns such as branches, joints and loops. Doing so, we arranged a worst case analysis taking the longest path for AND- and XOR-branches and assigned a maximum amount of possible Repeat loop iterations. By performing extensive evaluations and comparing the results against a state of the art baseline, we showed that following this optimization model, we were able to significantly save cost and execution time while still ensuring the defined SLAs.

The final contribution in this thesis is an extension to the formerly created optimization model (i.e., the SIPP) in order to be able to execute complex elastic processes in a hybrid cloud environment. The main driving factor is the consideration of data transfer aspects as especially data-intensive tasks may produce additional cost when executed in a hybrid cloud as cloud providers may account for in- and out-bound traffic. Hence, the extended SIPP considered a co-location of related tasks within the same data center in order to reduce the overall traffic. By considering data transfer aspects during the scheduling, we were able to reduce the overall cost compared to an ad hoc-based scaling baseline.

Beside of the planning phase, the other three phases have also been considered in our contributions, i.e., the next step in the MAPE-K cycle is to execute the formerly created scheduling and resource allocation plan. This means, software-based services are invoked according to this scheduling plan and resources are either leased or released according to the resource allocation plan.

The other two phases are monitoring and analyzing: As each process step is mapped to a software-based service, the service invocations are monitored in terms of execution time and used resources, i.e., the amount of needed RAM and CPU. The monitored data is analyzed in order to be able to detect SLA violations in time and in order to arrange countermeasures in time. Hence, the analyzing result is directly used within the planning phase.

Designing ViePEP according to the MAPE-K cycle allowed us to use cloud-based computational resources in a cost-efficient way for elastic process enactments. We considered user-defined SLAs and adjusted the system landscape accordingly in case of potential violations. The newly created extensions of ViePEP have been evaluated extensively and compared against baselines which represent the current state of the art.

8.2 Research Questions Revised

We introduced our research questions in Section 1.2 which guided the work in this thesis. Hence, in this section, we revisit these questions and provide a summary of how they have been answered within the context of this work along with limitations of our contributions. The latter one will be recapped in Section 8.3 in form of possible future work.

Research Question I:
How can cloud-based resources be used for business process management in a cost-efficient way?

The original version of ViePEP 1.0 (see Section 2.4) was designed to use cloud-based computational resources for process enactments. However, it was limited in its efficiency as neither cost nor SLAs have been considered. Hence, in this work, we extended ViePEP resulting in a fully functional BPMS for elastic process enactments by considering several aspects including complex process patterns, BTUs and SLAs. Further, by accounting future process executions during scheduling decisions. By pre-poning future service invocations, we were able to use the leased resources more efficiently and reduce the overall cost. However, using the present techniques, only the near known future is taken into account, i.e., the queued process requests. Using prediction techniques in order to forecast the arrival of future process requests could lead to more fine grained scheduling decisions.

Research Question II:
How can leasing cloud-based resource be optimized for enacting complex business processes?

After we showed how cloud-based computational resources can be used for business process enactments, we widened our research area and considered complex business processes. Complex business processes may include different process patterns, such as XOR-branches, AND-branches or Repeat loops. In order to achieve this, we used worst-case analysis which consider the longest possible path over the available XOR- and AND-branches. In addition to that, we provided a maximum value for allowed Repeat loop iterations. However, beside of worst-case analysis, there are other ways of how scheduling decisions can be made for complex business process which have not been considered in this work, e.g., best-case or average-case analysis. In addition to that, business processes may have additional patterns to the already considered XOR-, AND-branches and Repeat loops. Examples would be synchronized blocks, multiple choices, i.e., a complex XOR/AND-branch combinations, implicit terminations, etc. [107, 109]. Considering them in scheduling and resource optimization will need more complex analysis and techniques about the process structure.

Research Question III:
What problems need to be considered when executing complex business processes in a hybrid cloud environment?

There are several reasons why a single cloud is not sufficient for business process enactment. For example, a private cloud may only be able to provide a limited amount of resources, or another cloud provider may offer resources of different types, or simply the fact that leasing resources from another cloud provider is possibly cheaper. However, almost every cloud provider charges their user for in- and/or out-bound traffic. This means, when a hybrid cloud environment is used for elastic process enactment, the scheduling should take data transfer aspects into account. Exactly this was done and implemented into ViePEP. While our presented approach is able to reduce the overall cost (consisting out of leasing, penalty and data transfer cost), it does not consider any privacy aspects. For example, some process instances may not be allowed to be executed outside a certain area or country as sensible data is used. In addition, so far, we considered task-to-task communication, however, in some cases, a shared memory or a database may be used. Again, for this database different cost and privacy aspects may apply.

8.3 Future Work

In this thesis we presented several approaches to execute business processes on cloud-based computational resources, i.e., we showed how to realize elastic process enactments. However, as already shortly mentioned in Section 8.2, there are still open aspects which should be considered when enacting elastic processes in the cloud:

- The presented approaches of scheduling and resource optimization would benefit from a fine grained prediction algorithm of future process requests. The more precisely we know how many future invocations are expected, the better we are able to plan ahead and lease the required amount of resources, i.e., lease enough resources to handle peak loads, and release resources when they are not needed anymore. Such a prediction function needs to take into account several different aspects, included (but not limited to): past invocations, irregular events, regular or scheduled events (e.g., a new IPhone gets announced), etc. [28, 62, 75, 123].

- So far, we considered three different process patterns, AND-branches, XOR-branches and Repeat loops including their corresponding joins. However, there are several more complex process patterns available which may be included by customers or clients. In order to make reliable and efficient scheduling and resource allocation decisions, these different concepts need to be considered. Examples for such complex process patterns are: basic control flow patterns, parallel splits, synchronizations, exclusive choices, simple merges, multi-choices, synchronizing merges, multi-merges or different loop constructs [107, 109].

- Scheduling complex business processes requires to make assumptions about the future process flow in order to be able to know what services need to be invoked next. There are different ways to do this. So far, we followed a worst-case analysis, i.e., we took the longest path of XOR-blocks or AND-blocks, and assigned a maximum of allowed repetitions for Repeat loops, however, in some cases a best-case or average-case analysis may lead to better results.

- When enacting business processes in a hybrid cloud environment, a BPMS does not just have more resources available for its disposal, but also more challenges need to be tackled. So far, we considered different pricing models from different cloud providers, as well as data transfer cost, which apply when moving data into our out of the cloud. However, some services may operate on private or sensitive data which should not be moved into an arbitrary cloud, for example, hospital data should always remain in specific country. This fact needs to be considered during scheduling decisions. In addition, real-world use cases often incorporate more complex data patterns, like shared global storage memories or systems.

- Last but not least, the presented optimization model in Section 6.3.2 makes use of a linear pricing model for the VMs. However, cloud providers like Amazon offer different pricing models for different VM types, depending on their setting. Hence, in future work, different pricing models for the leasable resources should be considered as well as an extension to lease spot instances[1]. Although more variety of possible resources means scheduling is getting more complex, it also allows to make more cost-efficient scheduling decisions.

We expect that further work on resource optimization and elastic process enactments in the cloud will indeed take these aspects into account.

[1] `http://aws.amazon.com/en/ec2/purchasing-options/spot-instances/`

Bibliography

[1] Saeid Abrishami, Mahmoud Naghibzadeh, and Dick H. J. Epema. Deadline-constrained workflow scheduling algorithms for Infrastructure as a Service Clouds. *Future Generation Computer Systems*, 29(1):158–169, 2013.

[2] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 574–576, New York, NY, USA, 1999. ACM.

[3] Mourad Amziani, Tarek Melliti, and Samir Tata. Formal Modeling and Evaluation of Stateful Service-Based Business Process Elasticity in the Cloud. In *On the Move to Meaningful Internet Systems (OTM 2013)*, volume 8185 of *LNCS*, pages 21–38. Springer, 2013.

[4] Vasilios Andrikopoulos, Tobias Binz, Frank Leymann, and Steve Strauch. How to Adapt Applications for the Cloud environment – Challenges and Solutions in Migrating Applications to the Cloud. *Computing*, 95(6):493–535, 2013.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53:50–58, 2010.

[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report 4, EECS Department, University of California, Berkley, New York, NY, USA, April 2010.

[7] Ellen M. Arruda and Mary C. Boyce. A three-dimensional constitutive model for the large stretch behavior of rubber elastic materials. *Journal of the Mechanics and Physics of Solids*, 41(2):389 – 412, 1993.

[8] Jörg Becker, Martin Kugeler, and Michael Rosemann, editors. *Process Management: A Guide for the Design of Business Processes.* Springer, 2011.

[9] Kahina Bessai, Samir Youcef, Ammar Oulamara, and Claude Godart. Bi-criteria strategies for business processes scheduling in cloud environments with fairness metrics. In *IEEE 7th Intern. Conf. on Research Challenges in Information Science (RCIS 2013)*, pages 1–10. IEEE, 2013.

[10] Kahina Bessai, Samir Youcef, Ammar Oulamara, Claude Godart, and Selmin Nurcan. Resources allocation and scheduling approaches for business process applications in Cloud contexts. In *4th IEEE Intern. Conf. on Cloud Computing Technology and Science (CloudCom 2012)*, pages 496–503. IEEE, 2012.

[11] Kahina Bessai, Samir Youcef, Ammar Oulamara, Claude Godart, and Selmin Nurcan. Business Process scheduling strategies in Cloud environments with fairness metrics. In *IEEE 10th Intern. Conf. on Services Computing (SCC 2013)*, pages 519–526. IEEE, 2013.

[12] Kahina Bessai, Samir Youcef, Ammar Oulamara, Claude Godart, and Selmin Nurcan. Scheduling strategies for business process applications in cloud environments. *IJGHPC*, 5(4):65–78, 2013.

[13] Luiz Fernando Bittencourt and Edmundo Roberto Mauro Madeira. HCOC: A Cost Optimization Algorithm for Workflow Scheduling in Hybrid Clouds. *Journal of Internet Services and Applications*, 2(3):207–227, 2011.

[14] Robert B. Bohn, John Messina, Fang Liu, Jin Tong, and Jian Mao. Nist cloud computing reference architecture. In *Proceedings of the 2011 IEEE World Congress on Services*, SERVICES '11, pages 594–596, Washington, DC, USA, 2011. IEEE Computer Society.

[15] Ruth Breu, Schahram Dustdar, Johann Eder, Christian Huemer, Gerti Kappel, Julius Köpke, Philip Langer, Jürgen Mangler, Jan Mendling, Gustaf Neumann, Stefanie Rinderle-Ma, Stefan Schulte, Stefan Sobernig, and Barbara Weber. Towards Living Inter-Organizational Processes. In *15th IEEE Conf. on Business Informatics (CBI 2013)*, pages 363–366. IEEE, 2013.

[16] Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011.

[17] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N. Calheiros. InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services. In *10th Intern. Conf. on Algorithms and Architectures for Parallel Processing (ICA3PP 2010)*, volume 6081 of *LNCS*, pages 13–31. Springer, 2010.

[18] 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 Computing Systems*, 25(6):599–616, 2009.

[19] 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.

[20] Zhicheng Cai, Xiaoping Li, and Jatinder N.D. Gupta. Critical Path-Based Iterative Heuristic for Workflow Scheduling in Utility and Cloud Computing. In *11th Intern. Conf. on Service Oriented Computing (ICSOC 2013)*, volume 8274 of *LNCS*, pages 207–221. Springer, 2013.

[21] Qi Cao, Zhi-Bo Wei, and Wen-Mao Gong. An Optimized Algorithm for Task Scheduling Based on Activity Based Costing in Cloud Computing. In *3rd Intern. Conf. on Bioinformatics and Biomedical Engineering (ICBBE 2009)*, pages 1–3. IEEE, 2009.

[22] Valeria Cardellini, Emiliano Casalicchio, Francesco Lo Presti, and Luca Silvestri. SLA-aware Resource Management for Application Service Providers in the Cloud. In *First Intern. Symp. on Network Cloud Computing and Applications (NCCA '11)*, pages 20–27. IEEE, 2011.

[23] Diane Cook and Sajal Das. *Smart Environments: Technology, Protocols and Applications (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2004.

[24] Georgiana Copil, Daniel Moldovan, Hong Linh Truong, and Schahram Dustdar. Multi-level elasticity control of cloud services. In *11th Intern. Conf. on Service-Oriented Computing (ICSOC 2013)*, volume 8274 of *LNCS*, pages 429–436. Springer, 2013.

[25] Thomas Aidan Curran and Gerhard Keller. *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. Prentice Hall PTR, Upper Saddle River, 1997.

[26] Thomas H. Davenport. *Process Innovation: Reengineering Work Through Information Technology*. Harvard Business School Press, Boston, MA, 1993.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[28] PeterA. Dinda and DavidR. O'Hallaron. Host load prediction using linear models. *Cluster Computing*, 3(4):265–280, 2000.

[29] Shad Dowlatshahi and Qing Cao. The relationships among virtual enterprise, information technology, and business performance in agile manufacturing: An industry perspective. *European Journal of Operational Research*, 174(2):835–860, 2006.

[30] Evert Ferdinand Duipmans, Luís Ferreira Pires, and LuizOlavo Bonino da Silva Santos. A transformation-based approach to business process management in the cloud. *Journal of Grid Computing*, 12(2):191–219, 2014.

[31] Schahram Dustdar and Kamal Bhattacharya. The social compute unit. *Internet Computing, IEEE*, 15(3):64–69, May 2011.

[32] Schahram Dustdar, Yike Guo, Benjamin Satzger, and Hong Linh Truong. Principles of Elastic Processes. *IEEE Internet Computing*, 15(5):66–71, 2011.

[33] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *Intern. J. of Web and Grid Services*, 1(1):1–30, 2005.

[34] Vincent C. Emeakaroha, Ivona Brandic, Michael Maurer, and Ivan Breskovic. SLA-Aware Application Deployment and Resource Allocation in Clouds. In *COMPSAC Workshops 2011*, pages 298–303. IEEE, 2011.

[35] Thomas Erl, Ricardo Puttini, and Zaigham Mahmood. *Cloud Computing: Concepts, Technology & Architecture*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2013.

[36] Marc E. Frincu, Stéphane Genaud, and Julien Gossa. On the Efficiency of Several VM Provisioning Strategies for Workflows with Multi-threaded Tasks on Clouds. *Computing*, 96:1059–1086, 2014.

[37] Alessio Gambi and Cesare Pautasso. RESTful Business Process Management in the Cloud. In *5th Intern. Workshop on Principles of Engineering Service-Oriented Systems (PESOS 2013) in conjuction with the 35th Intern. Conf. on Software Engineering (ICSE 2013)*. IEEE, 2013.

[38] Gartner, Inc. *Forecast: Public Cloud Services, Worldwide, 2011-2017, 4Q13 Update*. Gartner, Inc., 2013.

[39] Stéphane Genaud and Julien Gossa. Cost-wait Trade-offs in Client-side Resource Provisioning with Elastic Clouds. In *4th Intern. Conf. on Cloud Computing (CLOUD 2011)*, pages 1–8. IEEE, 2011.

[40] Heiko Gewald and Jens Dibbern. Risks and benefits of business process outsourcing: A study of transaction services in the German banking industry. *Information & Management*, 46(4):249–257, 2009.

[41] Asif Qumer Gill, Deborah Bunker, and Philip Seltsikas. An Empirical Analysis of Cloud, Mobile, Social and Green Computing – Financial Services IT Strategy and Enterprise Architecture. In *IEEE Ninth Intern. Conf. on Dependable, Autonomic and Secure Computing (DASC 2011)*, pages 697–704. IEEE, 2011.

[42] 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 (CLOUD 2015)*, pages 17–24. IEEE, 2015.

[43] Philipp Hoenisch, Dieter Schuller, Stefan Schulte, Christoph Hochreiner, and Schahram Dustdar. Optimization of complex elastic processes. *Services Computing, IEEE Transactions on*, PP(99):1–1, 2015.

[44] Philipp Hoenisch, Stefan Schulte, and Schahram Dustdar. Workflow Scheduling and Resource Allocation for Cloud-based Execution of Elastic Processes. In *6th IEEE Intern. Conf. on Service Oriented Computing and Applications (SOCA 2013)*, pages 1–8. IEEE, 2013.

[45] Philipp Hoenisch, Stefan Schulte, Schahram Dustdar, and Srikumar Venugopal. Self-Adaptive Resource Allocation for Elastic Process Execution. In *6th Intern. Conf. on Cloud Computing (CLOUD 2013)*, pages 220–227. IEEE, 2013.

[46] 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*, Goa, India, nov 2015.

[47] Christina Hoffa, Gaurang Mehta, Timothy Freeman, Ewa Deelman, Kate Keahey, Bruce Berriman, and John Good. On the Use of Cloud Computing for Scientific Workflows. In *IEEE Fourth Intern. Conf. on e-Science (eScience'08)*, pages 640–645. IEEE, 2008.

[48] Zhengxing Huang, Wil M. P. van der Aalst, Xudong Lu, and Huilong Duan. Reinforcement learning based resource allocation in business process management. *Data & Knowledge Engineering*, 70(1):127–145, 2011.

[49] Michael N. Huhns and Munindar P. Singh. Service-oriented computing: key concepts and principles. *Internet Computing, IEEE*, 9(1):75–81, Jan 2005.

[50] Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. How a consumer can measure elasticity for cloud platforms. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 85–96, New York, NY, USA, 2012. ACM.

[51] Christian Janiesch, Ingo Weber, Jörn Kuhlenkamp, and Michael Menzel. Optimizing the Performance of Automated Business Processes Executed on Virtualized Infrastructure. In *47th Hawaii Intern. Conf. on System Sciences (HICSS 2014)*, pages 3818–3826. IEEE, 2014.

[52] Ernst Juhnke, Tim Dörnemann, David Bock, and Bernd Freisleben. Multi-objective Scheduling of BPEL Workflows in Geographically Distributed Clouds. In *4th Intern. Conf. on Cloud Computing (CLOUD 2011)*, pages 412–419. IEEE, 2011.

[53] Gideon Juve and Ewa Deelman. Scientific Workflows and Clouds. *ACM Crossroads*, 16(3):14–18, 2010.

[54] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *J. Artif. Int. Res.*, 4(1):237–285, May 1996.

[55] 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.

[56] Gerhard Keller and Thomas Teufel. *SAP R/3 Process Oriented Implementation: Iterative Process Prototyping.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[57] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[58] Attila Kertesz, Gabor Kecskemeti, and Ivona Brandic. An Interoperable and Self-adaptive Approach for SLA-based Service Virtualization in Heterogeneous Cloud Environments (forthcoming). *Future Generation Computer Systems*, NN(NN), 2013.

[59] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *2010 ACM SIGMOD Intern. Conf. on Management of Data (SIGMOD '10)*, pages 579–590. ACM, 2010.

[60] Eva Kühn, Richard Mordinyi, Mario Lang, and Adnan Selimovic. Towards zero-delay recovery of agents in production automation systems. In *Web Intelligence and Intelligent Agent Technologies, 2009. WI-IAT '09. IEEE/WIC/ACM International Joint Conferences on*, volume 2, pages 307–310, Sept 2009.

[61] Akhil Kumar, Wil M. P. van der Aalst, and Eric M.W. Verbeek. Dynamic Work Distribution in Workflow Management Systems: How to Balance Quality and Performance? *J. of Management Information Systems*, 18(3):157–194, 2002.

[62] Dara Kusic and Nagarajan Kandasamy. Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems. In *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pages 74–83, June 2006.

[63] Ulrich Lampe, Thorsten Mayer, Johannes Hiemer, Dieter Schuller, and Ralf Steinmetz. Enabling Cost-Efficient Software Service Distribution in Infrastructure Clouds at Run Time. In *4th IEEE Intern. Conf. on Service-Oriented Computing and Applications (SOCA 2011)*, pages 1–8. IEEE, 2011.

[64] 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 Conf. on Information Systems (AMCIS 2013)*. AIS, 2013.

[65] George Lawton. Developing software online with platform-as-a-service technology. *Computer*, 41(6):13–15, June 2008.

[66] Young Choon Lee, Chen Wang, Albert Y. Zomaya, and Bing Bing Zhou. Profit-Driven Service Request Scheduling in Clouds. In *10th IEEE/ACM Intern. Conf. on Cluster, Cloud and Grid Computing (CCGrid 2010)*, pages 15–24. IEEE, 2010.

[67] Sebastian Lehrig, Hendrik Eikerling, and Steffen Becker. Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics. In *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, QoSA '15, pages 83–92, New York, NY, USA, 2015. ACM.

[68] Philipp Leitner, Johannes Ferner, Waldemar Hummer, and Schahram Dustdar. Data-driven and automated prediction of service level agreement violations in service compositions. *Distributed and Parallel Databases*, 31(3):447–470, 2013.

[69] Philipp Leitner, Waldemar Hummer, Benjamin Satzger, Christian Inzinger, and Schahram Dustdar. Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud. In *5th Intern. Conf. on Cloud Computing (CLOUD 2012)*, pages 213–220. IEEE, 2012.

[70] Frank Leymann, Dieter Roller, and Marc-Thomas Schmidt. Web services and business process management. *IBM Systems Journal*, 42(2):198–211, 2002.

[71] Han Li and Srikumar Venugopal. Using Reinforcement Learning for Controlling an Elastic Web Application Hosting Platform. In *8th Intern. Conf. on Autonomic Computing (ICAC 2011)*, pages 205–208. ACM, 2011.

[72] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated Control for Elastic Storage. In *7th Intern. Conf. on Autonomic Computing (ICAC 2010)*, pages 1–10. ACM, 2010.

[73] Marin Litoiu, Murray Woodside, Johnny Wong, Joanna Ng, and Gabriel Iszlai. A Business Driven Cloud Optimization Architecture. In *25th Symp. on Applied Computing (SAC 2010)*, pages 380–385. ACM, 2010.

[74] Bertram Ludäscher, Mathias Weske, Timothy M. McPhillips, and Shawn Bowers. Scientific Workflows: Business as Usual? In *7th Intern. Conf. on Business Process Management (BPM 2009)*, volume 5701 of *LNCS*, pages 31–47. Springer, 2009.

[75] Fumio Machida, Masahiro Kawato, and Yoshiharu Maeno. Just-in-time server provisioning using virtual machine standby and request prediction. In *Autonomic Computing, 2008. ICAC '08. International Conference on*, pages 163–171, June 2008.

[76] R. Timothy Marler and S. Jasbir Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004.

[77] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. Cloud computing – The business perspective. *Decision Support Systems*, 51:176–189, 2011.

[78] Michael Maurer, Ivona Brandic, and Rizos Sakellariou. Enacting slas in clouds using rules. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6852 of *Lecture Notes in Computer Science*, pages 455–466. Springer Berlin Heidelberg, 2011.

[79] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. Recommendations of the National Institute of Standards and Technology, 2011.

[80] Jan Mendling, H. M. W. Verbeek, Boudewijn F. van Dongen, Wil M. P. 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.

[81] Mohamed Mohamed, Mourad Amziani, Djamel Belaïd, Samir Tata, and Tarek Melliti. An autonomic approach to manage elasticity of business processes in the cloud. *Future Generation Computer Systems*, 50:49 – 61, 2015. Quality of Service in Grid and Cloud 2015.

[82] Bela Mutschler, Manfred Reichert, and Johannes Bumiller. Unleashing the Effectiveness of Process-Oriented Information Systems: Problem Analysis, Critical Success Factors, and Implications. *IEEE Trans. on Systems, Man, and Cybernetics, Part C*, 38(3):280–291, 2008.

[83] Hien Nguyen Van, Frédéric Dang Tran, and Jean-Marc Menaud. Autonomic virtual resource management for service hosting platforms. In *2009 Workshop on Software Engineering Challenges of Cloud Computing (CLOUD) at 31st Intern. Conf. on Software Engineering (ICSE 2009)*, pages 1–8. IEEE, 2009.

[84] Hien Nguyen Van, Frédéric Dang Tran, and Jean-Marc Menaud. SLA-Aware Virtual Resource Management for Cloud Infrastructures. In *Ninth IEEE Intern. Conf. on Computer and Information Technology (CIT '09)*, pages 357–362. IEEE, 2009.

[85] 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 IEEE Intern. Conf. on Advanced Information Networking and Applications (AINA 2010)*, pages 400–407. IEEE, 2010.

[86] Maja Pantic, Anton Nijholt, Alex Pentland, and Thomas S. Huanag. Human centred intelligent human computer interaction: How far are we from attaining it? *Int. J. Auton. Adapt. Commun. Syst.*, 1(2):168–187, August 2008.

[87] Maja Pantic, Alex Pentland, and Anton Nijholt. Special issue on human computing. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 39(1):3–6, Feb 2009.

[88] Michael P. Papazoglou, Paolo. Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, Nov 2007.

[89] J. Ross Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[90] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A taxonomy and survey of cloud computing systems. In *Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC*, NCM '09, pages 44–51, Washington, DC, USA, 2009. IEEE Computer Society.

[91] Sebastian Rohjans, Christian Dänekas, and Mathias Uslar. Requirements for Smart Grid ICT Architectures. In *Third IEEE PES Innovative Smart Grid Technologies (ISGT) Europe Conf.*, pages 1–8. IEEE, 2012.

[92] Michael Rosemann and Jan vom Brocke. The Six Core Elements of Business Process Management. In *Handbook on Business Process Management 1*, pages 107–122. Springer, 2010.

[93] Enrique Santacana, Gary Rackliffe, Le Tang, and Xiaoming Feng. Get Smart. *IEEE Power and Energy Magazine*, 8(2):41–48, 2010.

[94] Benjamin Satzger, Harald Psaier, Daniel Schall, and Schahram Dustdar. Stimulating skill evolution in market-based crowdsourcing. In Stefanie Rinderle-Ma, Farouk Toumani, and Karsten Wolf, editors, *Business Process Management*, volume 6896 of *Lecture Notes in Computer Science*, pages 66–82. Springer Berlin Heidelberg, 2011.

[95] Dieter Schuller, André Miede, Julian Eckert, Ulrich Lampe, Apostolos Papageorgiou, and Ralf Steinmetz. QoS-based Optimization of Service Compositions for Complex Workflows. In *Intern. Conf. on Service Oriented Computing (ICSOC 2010)*, pages 641–648, 2010.

[96] Stefan Schulte, Philipp Hoenisch, Christoph Hochreiner, Schahram Dustdar, Matthias Klusch, and Dieter Schuller. Towards Process Support for Cloud Manufacturing (accepted for publication). In *18th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2014)*, pages 142–149. IEEE Computer Society, Washington, DC, USA, 2014.

[97] Stefan Schulte, Philipp Hoenisch, Srikumar Venugopal, and Schahram Dustdar. Introducing the vienna platform for elastic processes. In *In Performance Assessment and Auditing in Service Computing Workshop (PAASC 2012) at 10th International*

Conference on Service Oriented Computing (ICSOC 2012), volume 7759 of *Lecture Notes in Computer Science*, pages 179–190. Springer Berlin Heidelberg, 2013.

[98] 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.

[99] Stefan Schulte, Dieter Schuller, Philipp Hoenisch, Ulrich 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.

[100] Anja Strunk. QoS-Aware Service Composition: A Survey. In *8th IEEE European Conf. on Web Services (ECOWS 2010)*, pages 67–74. IEEE, 2010.

[101] Claudia Szabo and Trent Kroeger. Evolving Multi-objective Strategies for Task Allocation of Scientific Workflows on Public Clouds. In *IEEE Congress on Evolutionary Computation (CEC 2012)*, pages 1–8. IEEE, 2012.

[102] Gerard J. Tellis. The price elasticity of selective demand: A meta-analysis of econometric models of sales. *Journal of Marketing Research*, 25(4):pp. 331–341, 1988.

[103] Leslie R.G. Treloar. *The Physics of Rubber Elasticity.* Oxford Classic Texts in the Physical Sciences. OUP Oxford, 2005.

[104] Hong Linh Truong, Schahram Dustdar, Georgiana Copil, Alessio Gambi, Waldemar Hummer, Duc-Hung Le, and Daniel Moldovan. Comot - A platform-as-a-service for elasticity in the cloud. In *2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, March 11-14, 2014*, pages 619–622, 2014.

[105] Bhuvan Urgaonkar and Abhishek Chandra. Dynamic provisioning of multi-tier internet applications. In *Proceedings of the Second International Conference on Automatic Computing*, ICAC '05, pages 217–228, Washington, DC, USA, 2005. IEEE Computer Society.

[106] Ruben Van den Bossche, Kurt Vanmechelen, and Jan Broeckhove. Online cost-efficient scheduling of deadline-constrained workloads on hybrid clouds. *Future Generation Computing Systems*, 29(4):973–985, 2013.

[107] Wil M. P van der Aalst, Alistair P. Barros, Arthur H. M. ter Hofstede Hofstede, and Bartek Kiepuszewski. Advanced workflow patterns. In *Proceedings of the 7th International Conference on Cooperative Information Systems*, CooplS '02, pages 18–29, London, UK, UK, 2000. Springer-Verlag.

[108] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[109] Wil M. P van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, July 2003.

[110] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske. Business process management: A Survey. In *Intern. Conf. on Business Process Management (BPM 2003)*, volume 2678 of *LNCS*, pages 1–12. Springer, 2003.

[111] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008.

[112] Marno Verbeek. *A Guide to Modern Econometrics*. John Wiley & Sons, Hoboken, NJ, 3rd edition, 2008.

[113] Alessandro Vinciarelli, Maja Pantic, and Hervé Bourlard. Social signal processing: Survey of an emerging domain. *Image Vision Comput.*, 27(12):1743–1759, November 2009.

[114] Yi Wei and M. Brian Blake. Adaptive Service Workflow Configuration and Agent-Based Virtual Resource Management in the Cloud. In *2013 IEEE Intern. Conf. on Cloud Engineering (IC2E 2013)*, pages 279–284. IEEE, 2013.

[115] Yi Wei and M. Brian Blake. Decentralized Resource Coordination across Service Workflows in a Cloud Environment. In *22nd IEEE Intern. Conf. on Collaboration Technologies and Infrastructures (WETICE 2013)*, pages 15–20. IEEE, 2013.

[116] Yi Wei and M. Brian Blake. Proactive virtualized resource management for service workflows in the cloud. *Computing*, 96(7):1–16, 2014.

[117] Yi Wei, M. Brian Blake, and Iman Saleh. Adaptive Resource Management for Service Workflows in Cloud Environments. In *2013 IEEE 27th Intern. Symp. on Parallel and Distributed Processing (IPDPS 2013) Workshops and PhD Forum*, pages 2147–2156. IEEE, 2013.

[118] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2nd edition, 2012.

[119] Stephen A White. *Business Process Model and Notation (BPMN) Version 1.2*. Object Management Group Inc., January 2009.

[120] 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 IEEE/ACM Intern. Symp. on Cluster, Cloud and Grid Computing (CCGrid 2011)*, pages 195–204. IEEE, 2011.

[121] Meng Xu, Li-Zhen Cui, Haiyang Wang, and Yanbing Bi. A Multiple QoS Constrained Scheduling Strategy of Multiple Workflows for Cloud Computing. In *IEEE Intern. Symp. on Parallel and Distributed Processing with Applications (ISPA 2009)*, pages 629–634. IEEE, 2009.

[122] Lamia Youseff, Mario Butrico, and Dilma Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, Nov 2008.

[123] Honglin Yu, Lexing Xie, and Scott Sanner. Predicting YouTube video viewcount with twitter feeds. In *the 22nd acm international conference on multimedia*, Orlando, Florida, USA, nov 2014.

[124] Jia Yu and Rajkumar Buyya. A Novel Architecture for Realizing Grid Workflow using Tuple Spaces. In *5th IEEE/ACM Intern. Workshop on Grid Computing (GRID 2004)*, pages 119–128. IEEE, 2004.

[125] Qin Zheng and Bharadwaj Veeravalli. Utilization-based pricing for power management and profit optimization in data centers. *Journal of Parallel and Distributed Computing*, 72(1):27–34, 2012.

[126] Michael zur Muehlen and Marta Indulska. Modeling languages for business processes and business rules: A representational analysis. *Information Systems*, 35(4):379–390, 2010.

APPENDIX A

Abbreviations

AWS	Amazon Web Services
bpel	Business Process Execution Language
BPM	Business Process Management
BPMS	Business Process Management System
BTU	Billing Time Unit
DaaS	Data as a Service
HaaS	Hardware as a Service
IaaS	Infrastructure as a Service
MILP	Mixed Integer Linear Programming
OLS	Ordinary Least Square
OS	Operating System
PaaS	Platform as a Service
QoS	Quality of Service
SaaS	Software as a Service
SIPP	Service Instance Placement Problem
SLA	Service Level Agreement
SLO	Service Level Objective
SOA	Service Oriented Architecture

SOC	Service Oriented Computing
ViePEP	Vienna Platform for Elastic Processes
VM	Virtual Machine
WAR	Web Application ARchive

APPENDIX B

Process Model Collection

This chapter lists all process models which are used for the evaluation runs of Chapter 6 and Chapter 7.

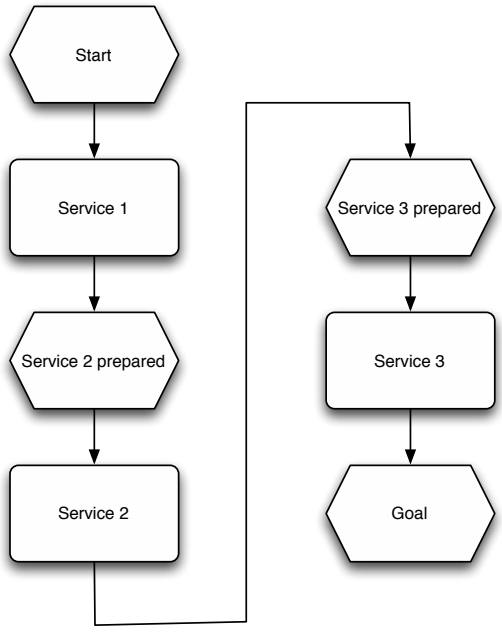


Figure B.1: Process Model 1

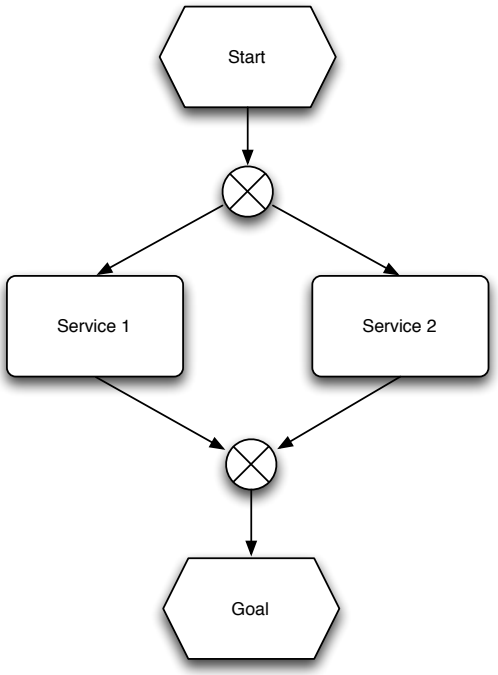


Figure B.2: Process Model 2

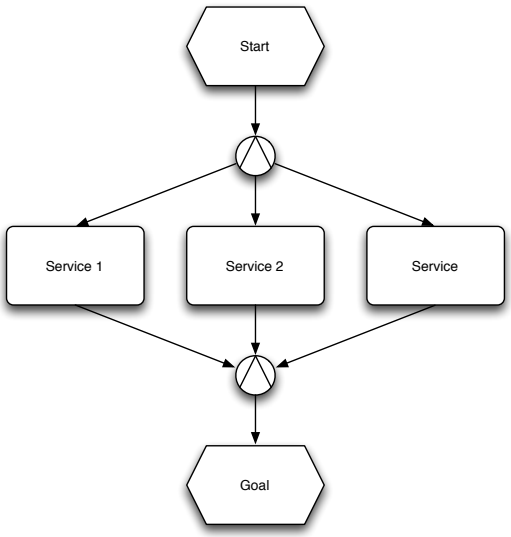


Figure B.3: Process Model 3

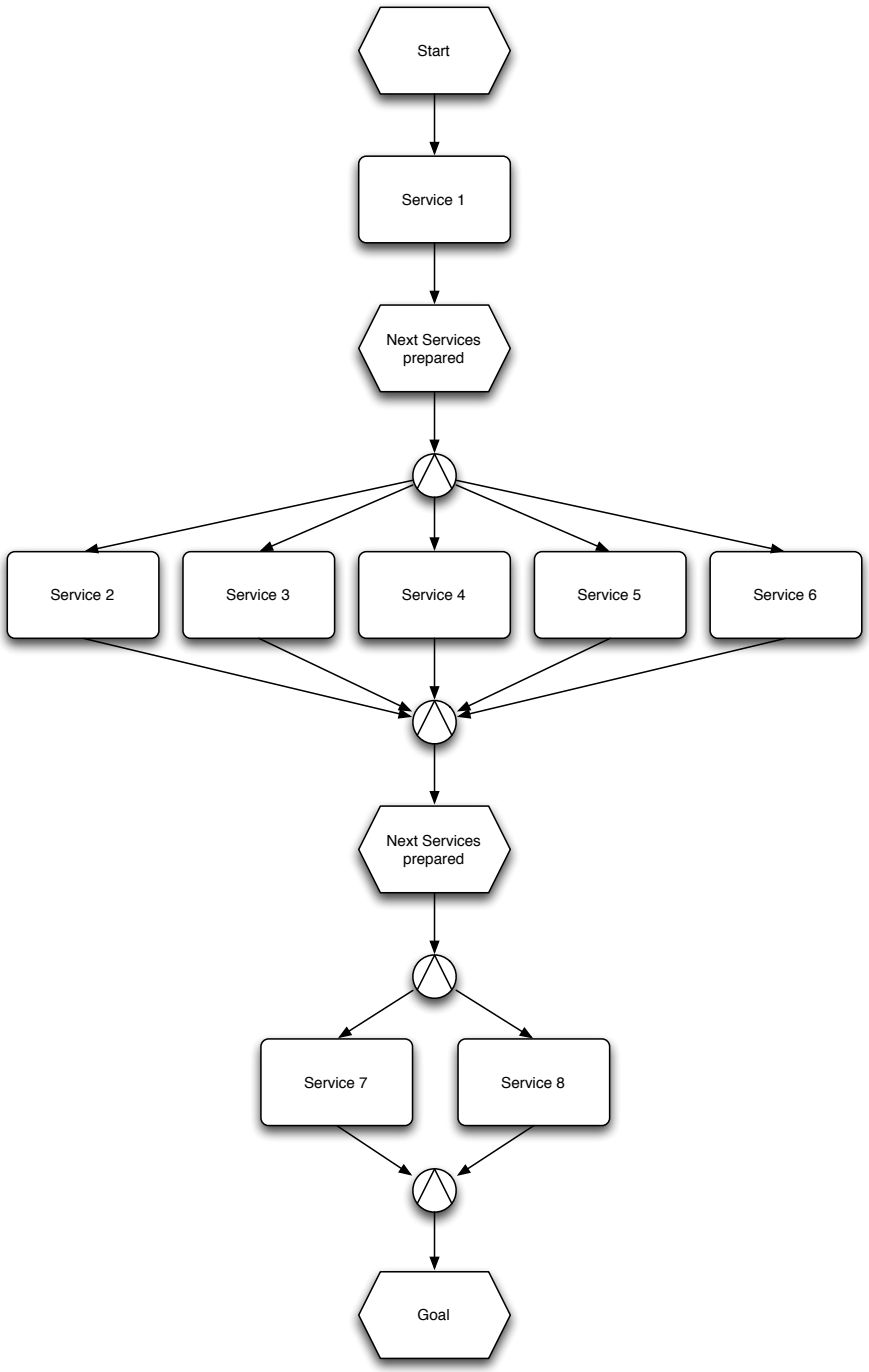


Figure B.4: Process Model 4

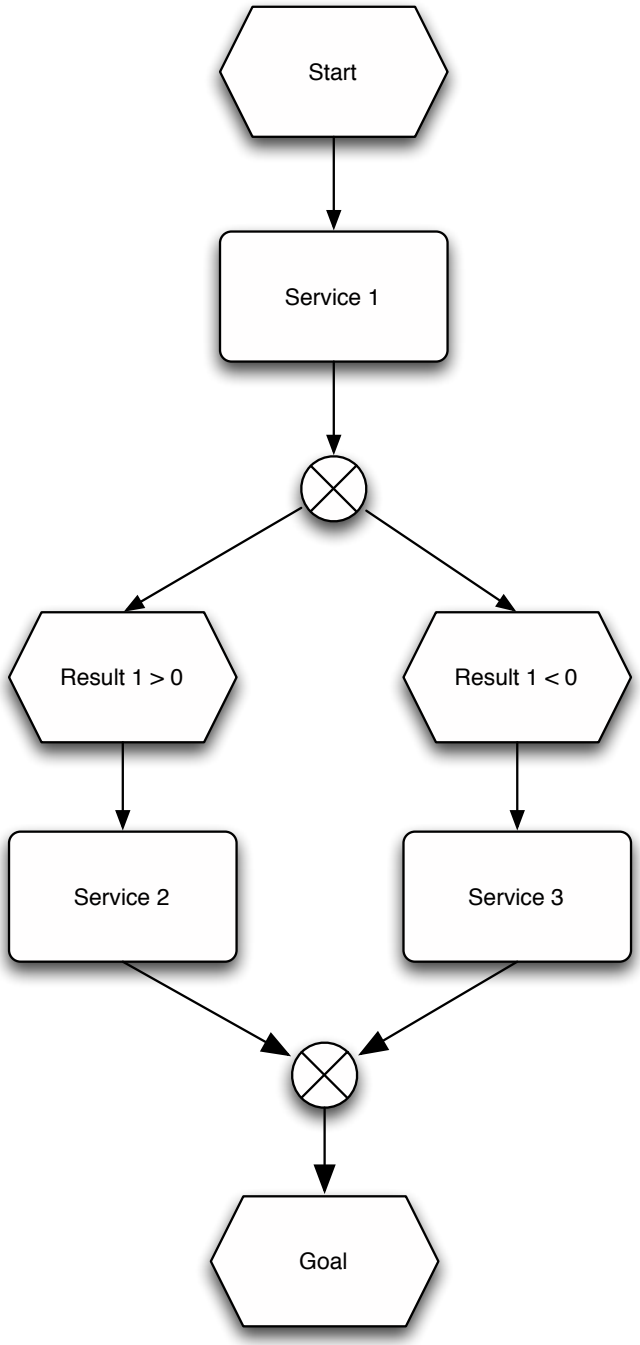


Figure B.5: Process Model 5

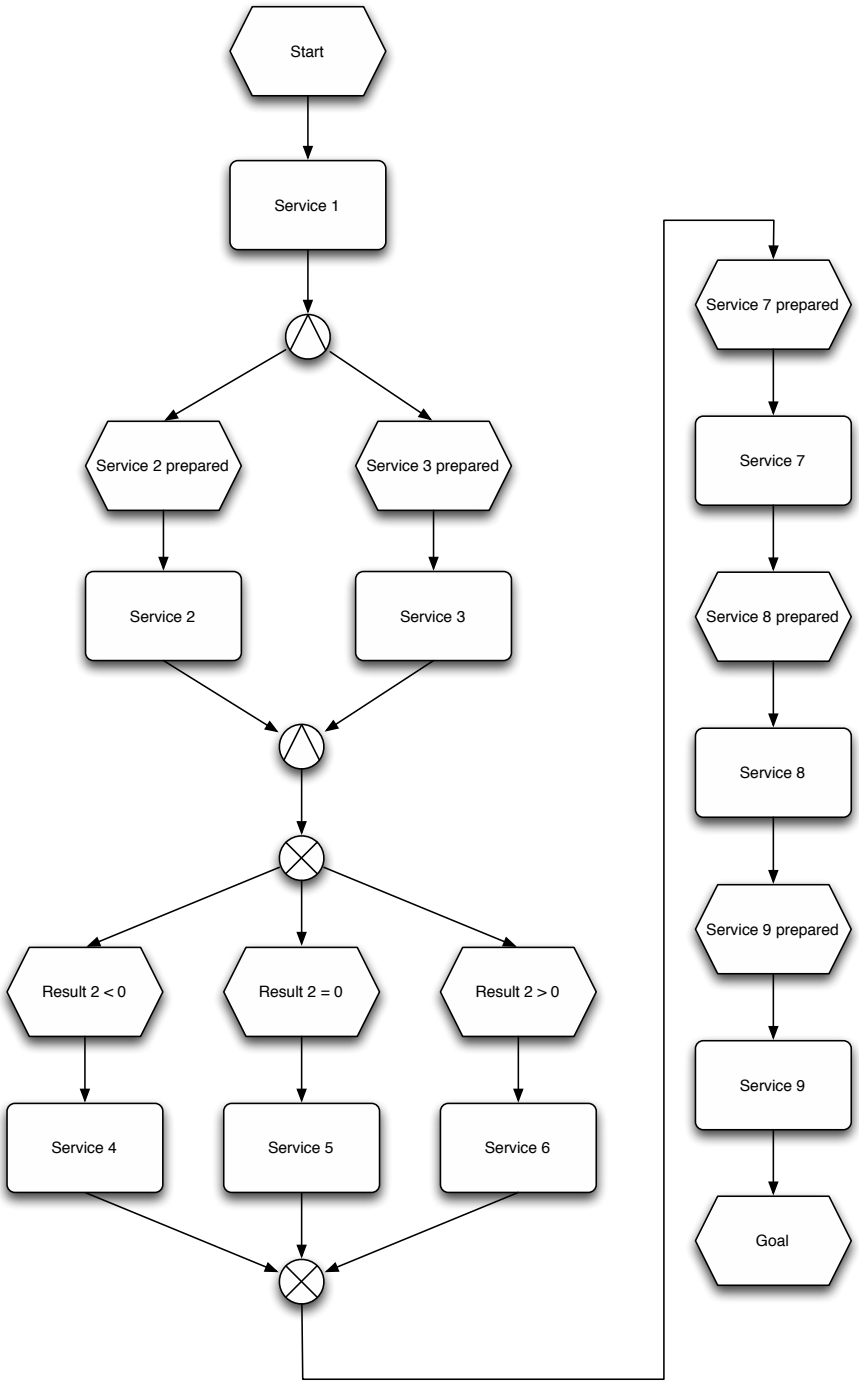


Figure B.6: Process Model 6

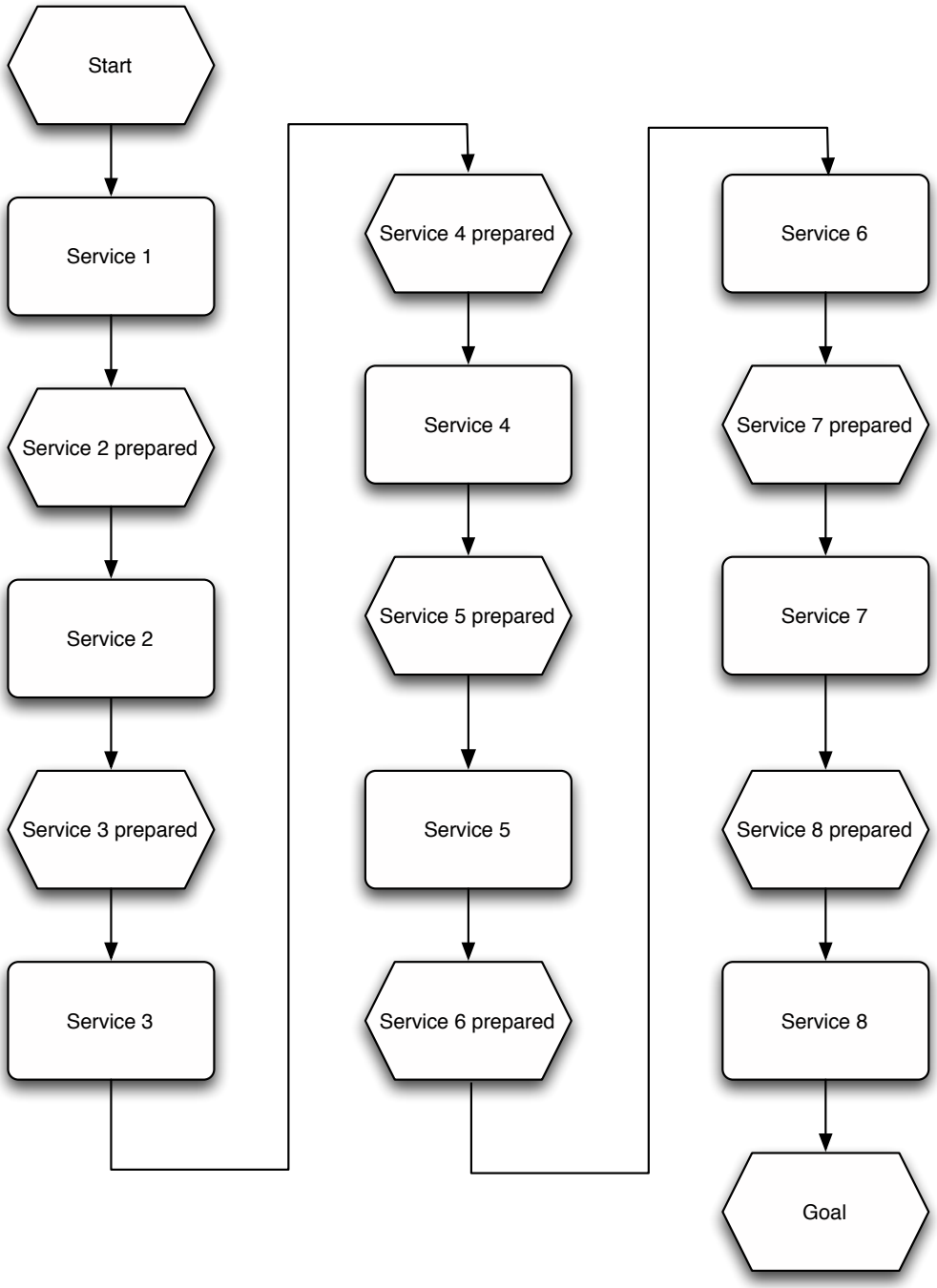


Figure B.7: Process Model 7

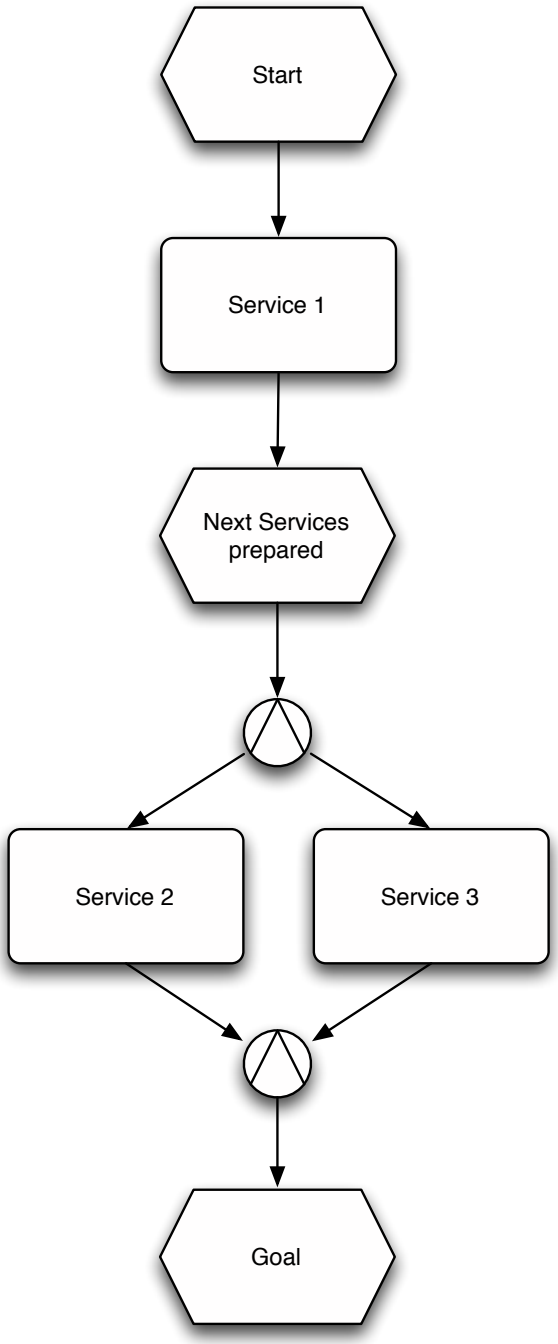


Figure B.8: Process Model 8

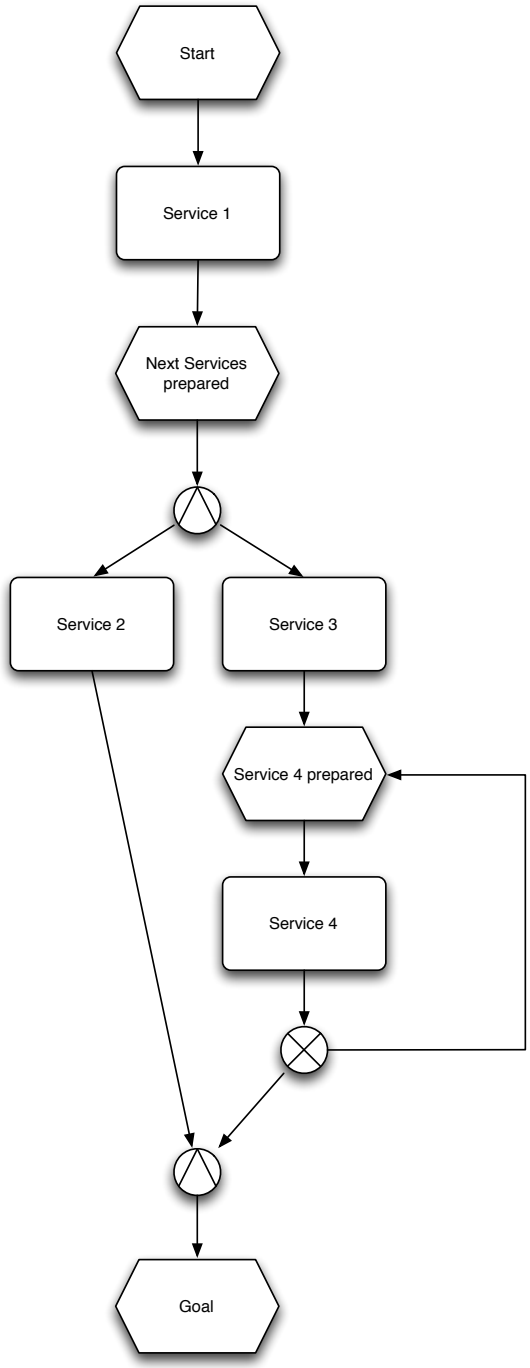


Figure B.9: Process Model 9

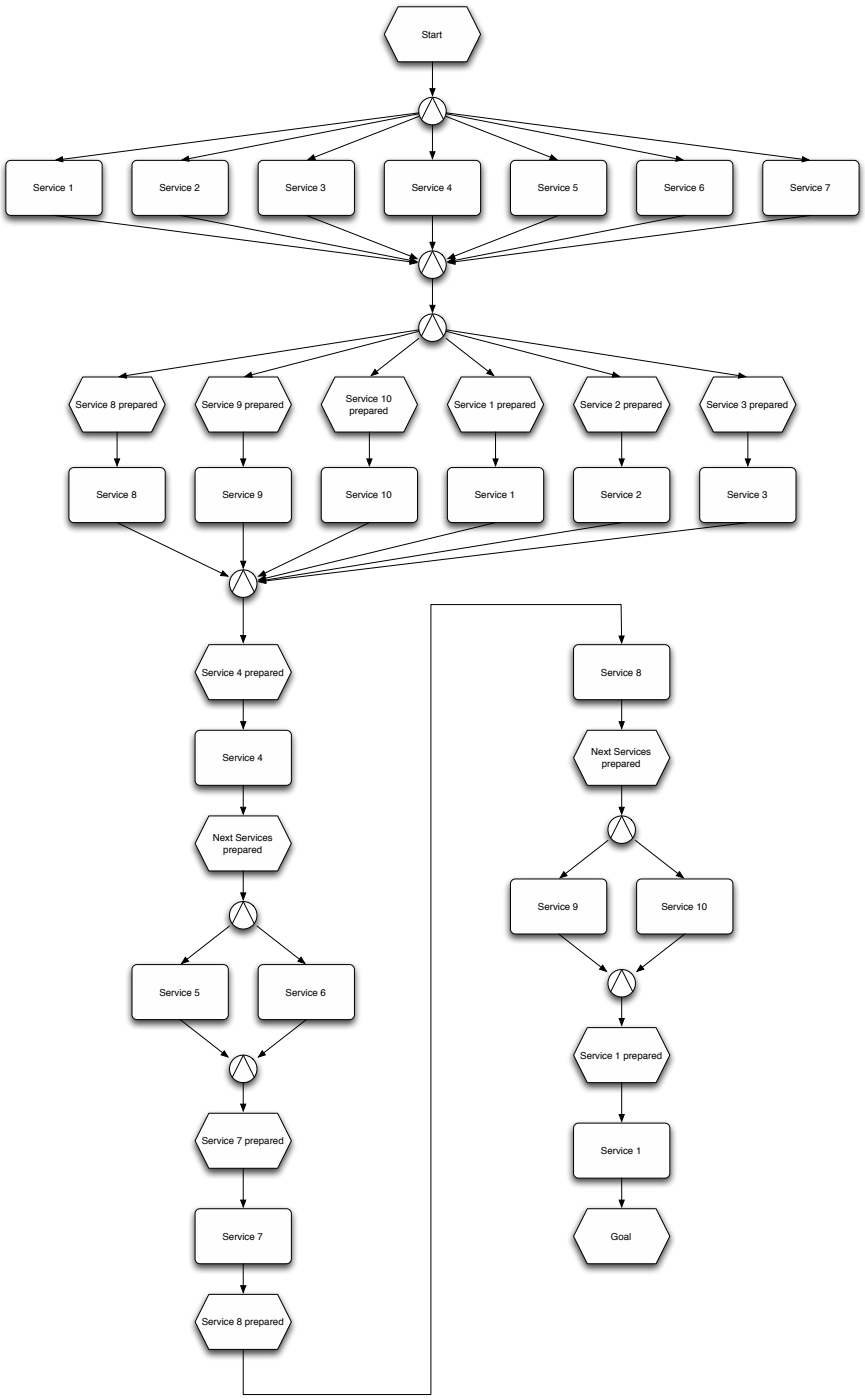


Figure B.10: Process Model 10

APPENDIX C

Curriculum Vitae

Personal Information

Name	Philipp Hoenisch
Address	Brunner Gasse 56-58, 2380 Perchtoldsdorf, Austria
Email	philipp@hoenisch.at
Web	https://www.linkedin.com/in/phoenisch
GitHub	https://github.com/bonomat
Born	May 27, 1988

Work Experiences

02/2015 – 05/2015	Researcher within the SSRG at NICTA, Sydney, Australia
09/2012 – ongoing	Project Assistant at TU Wien, Vienna, Austria
02/2012 – 07/2012	Research Intern at Service-Oriented Computing Group, University of New South Wales (UNSW), Sydney, Australia
07/2011 – 09/2011	Freelancer Project with FTW Telecommunications Research Center Vienna, Austria
03/2010 – 02/2012	Software Engineer at Plandata, Vienna, Austria
02/2010 – 02/2012	Software Engineer at OpenEngSB, Open Source Project, Vienna, Austria
08/2009 – 09/2009	Software Engineer at Siemens, Vienna, Austria

Education

2012 – 2015	**Ph.D.** in Computer Science at the Distributed Systems Group, TU Wien, Austria
2010 – 2013	**M.Sc. (Dipl.Ing.)** in Software Engineering & Internet Computing, TU Wien, Austria
2007 – 2010	**B.Sc.** in Software & Information Engineering, TU Wien, Austria

Publications

- **Philipp Hoenisch**, Ingo Weber, Stefan Schulte, Liming Zhu, and Alan Fekete. Four-fold auto-scaling on a Contemporary Deployment Platform using Docker Containers (accepted for publication). *In International Conference on Service Oriented Computing (ICSOC 2015)*, Goa, India, Nov 2015

- **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 (CLOUD 2015)*, pages 17-24. IEEE, 2015

- **Philipp Hoenisch**, Dieter Schuller, Stefan Schulte, Christoph Hochreiner, and Schahram Dustdar. Optimization of Complex Elastic Processes. *Services Computing, IEEE Transactions on (TSC)*, Volume NN, Number NN, Pages NN-NN, 2015

- 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*, Volume 46, Pages 36-50, 2015.

- **Philipp Hoenisch**, Dieter Schuller, Christoph Hochreiner, Stefan Schulte, Schahram Dustdar. Elastic Process Optimization – The Service Instance Placement Problem, *Technical Report TUV-1841-2014-01, Distributed Systems Group, TU Wien*, 2014

- Daniel Burgstahler, Stefan Schulte, Sven Abels, Kristof Kipp, **Philipp Hoenisch**, Schahram Dustdar and Ralph Steinmetz. Informationssysteme für Verkehrsteilnehmer: Datenintegration, Cloud-Dienste und der Persönliche Mobilitätsassistent. Praxis der Informationsverarbeitung und Kommunikation (PIK), Volume 37, Number 3, 243-250, 2014.

- Stefan Schulte, **Philipp Hoenisch**, Christoph Hochreiner, Schahram Dustdar, Matthias Klusch, and Dieter Schuller. Towards Process Support for Cloud Manufacturing (accepted for publication). *In 18th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2014)*, pages 142-149. IEEE Computer Society, Washington, DC, USA, 2014

- **Philipp Hoenisch**. Viepep - A BPMS for elastic processes. *In , Proceedings of the 6th Central-European Workshop on Services and their Composition, ZEUS 2014*, Potsdam, Germany, February 20-21, 2014., volume 1140 of CEUR Workshop Proceedings, pages 61–68. CEUR-WS.org, 2014

- Stefan Schulte, Dieter Schuller, **Philipp Hoenisch**, Ulrich Lampe, Schahram Dustdar, and Ralf Steinmetz. Cost-Driven Optimization of Cloud Resource Allocation

for Elastic Processes. *International Journal of Cloud Computing (IJCC)*, 1(2):1-14, 2013

- **Philipp Hoenisch**, Stefan Schulte, and Schahram Dustdar. Workflow Scheduling and Resource Allocation for Cloud-based Execution of Elastic Processes. In 6*th IEEE Intern. Conf. on Service Oriented Computing and Applications (SOCA 2013)*, pages 1-8. IEEE, 2013

- **Philipp Hoenisch**, Stefan Schulte, Schahram Dustdar, and Srikumar Venugopal. Self-Adaptive Resource Allocation for Elastic Process Execution. *In 6th Intern. Conf. on Cloud Computing (CLOUD 2013)*, pages 220–227. IEEE, 2013

- Franz Wotawa, Marco Schulz, Ingo Pill, Seema Jehan, Philipp Leitner, Waldemar Hummer, Stefan Schulte, **Philipp Hoenisch**, and Schahram Dustdar. Fifty Shades of Grey in SOA Testing. *In 9th Workshop on Advances in Model Based Testing (A-MOST 2013) at Sixth IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*, pages 154-157, 2013.

- 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 10th International Conference on Service Oriented Computing (ICSOC 2012)*, volume 7759 of Lecture Notes on Computer Science, pages 179-190. Springer, Berlin Heidelberg.

- Stefan Schulte, **Philipp Hoenisch**, Srikumar Venugopal, and Schahram Dustdar. Realizing Elastic Processes with ViePEP. *In 10th International Conference on Service Oriented Computing* - Demos, volume 7759 of Lecture Notes on Computer Science, pages 439-443. Springer, Berlin Heidelberg, 2012.