Optimized Container-based Process Execution in the Cloud

Philipp Waibel[1,2][0000−0002−5562−4430], Anton Yeshchenko[1][0000−0002−5346−8358], Stefan Schulte[2][0000−0001−6828−9945], and Jan Mendling[1][0000−0002−7260−524X]

[1] Institute for Information Business, WU Wien, Austria
{philipp.waibel,anton.yeshchenko,jan.mendling}@wu.ac.at
[2] Distributed Systems Group, TU Wien, Austria
{p.waibel,s.schulte}@infosys.tuwien.ac.at

Abstract. A key challenge for elastic business processes is the resource-efficient scheduling of cloud resources in such a way that Quality-of-Service levels are met. So far, this has been difficult, since existing approaches use a coarse-granular resource allocation based on virtual machines.
In this paper, we present a technique that provides fine-granular resource scheduling for elastic processes based on containers. In order to address the increased complexity of the respective scheduling problem, we develop a novel technique called GeCo based on genetic algorithms. Our evaluation demonstrates that in comparison to a baseline that follows an ad hoc approach a cost saving between 32.90% and 47.45% is achieved by GeCo while considering a high service level.

Keywords: Elastic Processes, Optimization, Scheduling, Business Process Management, Software Containers.

1 Introduction

Business process management (BPM) is concerned with the efficient and effective organization of business processes [9]. Business Process Management Systems (BPMS) and other types of process-aware information systems help to flexibly redesign business processes, but have paid less attention to flexibility during process execution [28]. The advent of cloud computing provides new opportunities to make process execution at runtime more flexible by providing the means to scale the underlying computational resources in an ad hoc manner [26]. It is the ambition of elastic BPMS (eBPMS) to dynamically lease and schedule cloud resources on-demand in order to meet Quality-of-Service (QoS) levels while avoiding over- or under-provisioning [10,28]. While an over-provisioning scenario, i.e., more resources are allocated than required, leads to a waste of resources and, therefore, unnecessary cost, an under-provisioning scenario, i.e., less resources are available than required, can lead to decreased QoS [10].

Recent approaches to eBPMS utilize virtual machines (VMs) in order to achieve scalability, e.g., [13,14]. One of the downsides of VMs is their coarse-

granular packaging of functionality at the level of a full-fledged operating system. Beside the additional computational resources that are allocated by the operating system, also the deployment and start time of a VM are affected by the requirement of an own operating system, which reduces the ad hoc elasticity of VM-based systems [21]. To eliminate this problem inherent with VMs, the concept of *containers* is a promising solution. In comparison to a VM, a container eliminates the requirement of an own operating system, thus offering a lightweight virtualization solution [8,24].

While the usage of containers as an execution environment for business processes already offers a solution that helps to reduce resource consumption (in comparison to VM-based process execution) further reductions can be achieved by performing resource and task scheduling, aiming at resource efficiency [14]. However, state-of-the-art resource optimization and task scheduling approaches rely on VMs. The according optimization approaches aim for more coarse-grained solutions and are not tailored for the usage of containers, thus not fully mobilizing the potential of elastic business processes with regard to efficient resource utilization.

In this paper, we present a fine-granular task scheduling and resource allocation optimization approach, called *GeCo* (Genetic Container). This approach provides a cost-efficient process execution on containers that are deployed in a cloud environment while considering user-defined Service Level Agreements (SLAs). We address the increased complexity of the scheduling problem by using a genetic algorithm to meet our requirements. The evaluation demonstrates the efficiency of our approach and its capability to consider SLAs. In comparison to a baseline that follows an ad hoc approach, GeCo achieves a cost saving of 32.90% and 47.45% while considering the user-defined SLAs.

The remainder of the paper is organized as follows: Section 2 provides background information on containers and elastic business processes and discusses the preliminaries of our approach. Section 3 discusses our genetic algorithm optimization approach. The evaluated of our approach is presented in Sect. 4. The related work is discussed in Sect. 5 and Sect. 6 concludes this paper and presents our future work.

2 Background

Next, we discuss some necessary background information about containers, elastic process execution, as well as further preliminaries needed to comprehend the approach presented in this paper.

2.1 Containers

A container is, similar to a VM, a virtualization solution that bundles custom software and required dependencies (e.g., a database or specific libraries) together to form an executable package [24]. However, in comparison to a VM, a container uses the operating system of the host environment and, thus, needs

less computational resources than a VM. This is achieved by partitioning the host operating system and computational resources to create isolated user space instances for each container [1,8,24]. This leads to configurable virtualization solutions similar to VMs but with smaller resource requirements and shorter deployment and startup times [29].

A *container image* configures and holds the custom software and possible required additional software (e.g., a database, or libraries) [24]. The distribution of those container images is done by so-called *container registries* such as Docker Hub, by uploading (*push*), sharing, and downloading (*pull*) the container images. By deploying a container image, a *container instance* is created.

2.2 Elastic Business Process Execution

A business process is composed of activities, called *process steps* (in the following called *steps*), and transitions, described in a *process model* [33]. The structure of a process model can contain sequential steps, parallel branches, called *AND-blocks* (started by an AND-split and closed by a join), exclusive branches, called *XOR-blocks* (started by an XOR-split and closed by a join), and *loops*. By executing a business process, a *process instance* is generated corresponding to the specific process model. Unless explicitly stated otherwise, the term process denotes a process instance in the remainder of this paper. A process is called an *elastic process* when cloud resources are used for the elastic execution of the process [10]. In the work at hand we concentrate on structured process models.

For a step to be executed, a software *service* is used. By deploying the service on a container, a *service instance* is created. To fulfill a specific step, the corresponding service instance is *invoked*. Each invocation runs for a specific *execution time* to complete its task.

2.3 Preliminaries

For our optimization approach, we assume that an eBPMS exists that serves as a middleware between the process owner, which requests the execution of a process, and the cloud, which is used for the execution of the processes. An example for such a middleware is presented in [27]. Furthermore, we assume that the process contains only software-based process steps, respectively that all human-based process steps are represented by a software-based service.

Similar to the concept of a microservice, each container image holds exactly one service of a specific type, with this service representing a particular step. Each container image can be deployed at any time (also several times in parallel) in the cloud. The service instance which is deployed with the container, can then be invoked several times simultaneously (i.e., as part of different process instances) as long as the container has sufficient computational resources. Once the service instance is not needed anymore, the container instance is removed.

The process owner defines for each process execution the required SLAs. If these SLAs are not fulfilled, penalty cost are charged by the process owner. In the current state the deadline until when the execution has to be finished is defined

as a SLA. The deadline is also most commonly regarded in the related work [28]. The aim of our optimization approach is to optimize the process execution in a cost-efficient way by considering the computational resource *and* penalty cost.

3 The GeCo Algorithm

Our algorithm, called GeCo, aims to reduce the resource consumption of the process execution by performing scheduling of the steps in a way that a timely overlapping is achieved. Thus, the steps can share a container instance that can be invoked several times simultaneously (see Sect. 2.3). This results in a reduced resource consumption and, as a consequence, in reduced computational resource leasing cost. To also minimize the penalty cost, which are part of the overall process execution cost, the algorithm also considers the user-defined SLA, i.e., process execution deadline, during scheduling. Therefore, the result of GeCo is a schedule when each step should be executed in which particular software container, to minimize execution cost while considering the user-defined SLA.

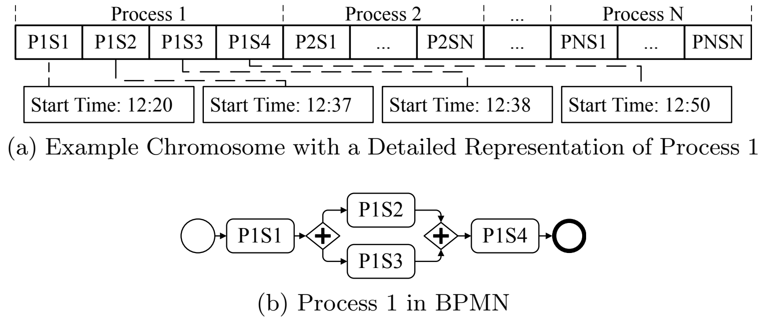
Since the problem of task scheduling is NP-hard [14], we use the concept of genetic algorithms in order to find a cost-efficient task scheduling. Genetic algorithms have shown their capability for task scheduling in different areas [16,36].

3.1 Genetic Algorithms

A genetic algorithm mimics an iterative, evolutionary process that applies the genetic operations *selection*, *mutation*, and *crossover* on each generation of possible solutions to a problem [34]. A generation contains several individual possible solutions called chromosomes, and each chromosome is a composition of several genes. For each iteration, the selection operator selects some chromosomes from the previous generation according to the fitness score of a chromosome. The fitness score is calculated by the *fitness function* for each chromosome and determines how well the chromosome solves a given problem (here: task scheduling). The selected chromosomes are then altered by the crossover and mutation functions to form a new generation. The crossover function swaps the genes of two chromosomes to form an offspring. The mutation function changes random genes to maintain the diversity of the generations. In addition to those newly created chromosomes a small number of elite chromosomes, i.e., chromosomes with the best fitness score, are added unaltered to the new generation. This process is repeated until a stop criterion, e.g., a defined optimization duration, is reached. At this point, the chromosome with the best fitness score is returned as a result. By this iterative approach, a genetic algorithm can browse a large search space to find a near-optimal solution in polynomial time [35].

3.2 Concepts of GeCo

As input, GeCo gets the current running and requested processes, their single steps (including the information if the execution of the step is currently ongoing,



(a) Example Chromosome with a Detailed Representation of Process 1

(b) Process 1 in BPMN

Fig. 1. Example Chromosome Representation

or has to be scheduled), the defined deadlines of the processes, and the status of the already running container instances. Furthermore, GeCo gets the information how long the service execution duration is and how much computational resources, with respect to CPU and RAM, are required for a particular amount of invocations. The execution duration and required computational resources have to be known upfront, e.g., via historical data [4].

The optimization is performed by GeCo each time a new process execution request is received. Furthermore, a re-optimization takes place if the execution times calculated by GeCo cannot be met, e.g., due to a longer execution duration of a preceding step. Otherwise, the calculated execution time is waited for before the execution of a step starts.

To further minimize the execution duration of the processes, GeCo deploys the container instance, required by a step, already during the execution of the preceding step to ensure that the container instance is up and running at the time it is needed. However, for the first step after the optimization, this is not possible since only after the optimization it is known which container instances have to be deployed. To consider this, GeCo schedules the execution time of the first steps in a way that there is enough time to deploy a container if needed.

Stopping Criteria As stopping criteria, we employ time-based criteria. These criteria can be of two kinds, i.e., according to the number of iterations or according to the calculation time of the genetic algorithm [3]. In GeCo, we employ a stopping criterion based on the execution time of the algorithm, since the execution time of the algorithm plays an important role in the problem at hand, as it affects the actual schedule of the service start times.

Chromosome Representation Our chromosome is a composition of all running and not yet running steps, which are the genes of the chromosome, and presents the schedule of the steps. Each gene holds the information when the execution of the step should start. This time is calculated by GeCo. Figure 1a shows a simplified example of a chromosome with a detailed representation of the process shown in Fig. 1b.

Algorithm 1 Initial Population

```
Require: processes, optEndTime
 1: function generateInitialChromosome
 2:     chromosome ← NULL
 3:     for all process ∈ processes do
 4:         lastStep ← getLastProcessStep(process)
 5:         maxBufferTime ← (process.deadline − lastStep.endTime)/process.steps.size
 6:         for all step ∈ process.steps do
 7:             randomTime ← getRandomNumber(0, maxBufferTime)
 8:             newStartTime ← getPrecedingStepEndTime(step) + randomTime
 9:             if timeForDeployment(newStartTime, optEndTime) is false then
10:                 newStartTime ← newStartTime + getReqTime(step, optEndTime)
11:             end if
12:             moveStep(step, newStartTime)
13:         end for
14:         chromosome ← chromosome.add(process)
15:     end for
16:     return chromosome
17: end function
```

Initial Population In GeCo, the initial population, i.e., first generation of chromosomes that is used as an input for the genetic algorithm [7], consists of randomly assigned task scheduling plans and, thus, already encodes possible solutions to the problem. As *a priori* knowledge, GeCo uses the deadlines of the processes and limits the random movement of the steps in a way that the deadlines are not violated in the initial population [3].

Algorithm 1 shows the process of creating a chromosome. This algorithm is executed several times, depending on the defined population size, to form the initial population. As an input, the algorithm gets all running and requested processes (*processes*), including the corresponding steps and the deadlines, and the end time of the optimization (*optEndTime*). The end time of the optimization is known since we use time-based stopping criteria. As a preliminary step (not shown in Algorithm 1), all start times of the steps are preset by setting the start time of a step to the end time of the preceding step. If a step is after a join gateway of an AND- or XOR-block, the end time of the latest step is used.

Beginning with line 3, Algorithm 1 iterates over all running and requested processes. The method *getLastProcessStep* returns the process step with the latest end time. To ensure that the deadline of a process is not violated, a maximal time that the execution start time of a step can be moved (*maxBufferTime*) is defined in line 5. This time is calculated by dividing the time between the process deadline (*process.deadline*) and the execution end time of the last step of the process (*lastStep.endTime*, as defined in line 4) by the number of process steps (*process.steps.size*). In case the process contains a loop structure, the algorithm uses a default amount of loop iterations, since the correct amount of loop iterations is not known at that point in time. This default amount of loop iterations has to be known upfront, e.g., via historical data.

In line 6–13, the random movement of the steps takes place. In line 7, a random number between 0 and *maxBufferTime* is selected and assigned to *randomTime*. This random number is then added to the end time of the latest preceding step of a step leading to the new start time *newStartTime* (line 8). The latest preceding step of a step is found by the method *getPrecedingStepEndTime*. If a step does not have a preceding step, i.e., it is the first step of a process, *optEndTime* is returned since this marks the time when the execution of the process can start.

As discussed before, the algorithm has to ensure that there is enough time to deploy the required container instances after the optimization is done. For this, the method *timeForDeployment* checks if *newStartTime* allows enough time for the deployment. If this is not the case, line 10 adds the time required to deploy the container instance, calculated by *getReqTime*, to *newStartTime*.

The start time of the *step* is then moved to the new start time in line 12. Eventually, the new process is added to the chromosome (line 14), and the chromosome is returned (line 16).

Fitness Function The fitness function assigns to each chromosome a fitness score. Finding a chromosome with the lowest container leasing cost and lowest penalty cost essentially yields a minimum fitness score.

The container leasing cost are the combination of all cost that arise due to the leasing of the required containers for the execution of the steps:

$$\sum_{c \in C} (c_{cpu} * p_{cpu} + c_{ram} * p_{ram}) * c_{duration} * f_{leasing} \tag{1}$$

In Eq. (1), C determines the list of required container deployments for the execution of the steps represented by the chromosome. One container is defined as $c \in C = \{c_1, c_2, ...\}$ and $c = (cpu, ram, duration)$ defines the required CPU and RAM of this container, and the duration how long the container has to be deployed. The price of a container is defined by p_{cpu}, i.e., the price for one CPU core, and p_{ram}, i.e., the prize for one GB of RAM. Finally, the configurable weighting parameter $f_{leasing}$ determines how much the leasing cost should be considered in the final fitness score.

The penalty cost is the combination of all cost that arise due to missed deadlines, i.e., the time between the termination of the last step and the deadline:

$$\sum_{w \in W} x(w) * (w_{end} - w_{deadline}) * f_{penalty} \tag{2}$$

In Eq. (2), W is the set of all processes of the chromosome and one process is defined as $w \in W = \{w_1, w_2, ...\}$. Each process contains the tuple $w = (end, deadline)$, where *end* is the execution end time of the process, i.e., the end time of the latest executed step, and *deadline* defines the deadline of the process. The weighting factor $f_{penalty}$ defines how much the penalty should be considered in the final fitness score. The term $x(w) \in \{0, 1\}$ considers if process w violates the deadline ($x(w) = 1$) or not ($x(w) = 0$).

The final fitness score is the sum of Eq. (1) and (2).

Mutation The mutation operation varies the start time of a step represented by a randomly selected gene of the chromosome. The resulting service execution time may not overlap the execution time of a preceding or upcoming step, which would violate the control flow defined by the process model.

How much the start time of a selected step (*step*) is changed is random. However, to ensure that no overlapping happens, the selection of how much the start time should be changed is bound by a lower (b_{lower}) and upper (b_{upper}) bound. There are three situations for b_{lower} and b_{upper}:

1. *step* is the first one in the process: In this situation, b_{lower} and b_{upper} are defined as shown in Eq. (3). If the container is not running, b_{lower} is defined by $opt_{endTime} + c_{deployDur}$, where $opt_{endTime}$ is the optimization end time and $c_{deployDur}$ the container deployment duration. Otherwise, i.e., the container was started by an already running step from another process, $b_{lower} = opt_{endTime}$. The bound b_{upper} is defined by the earliest start time of the next steps ($step_{ens}$) minus the duration of *step* ($step_{duration}$).

$$b_{lower} = \begin{cases} opt_{endTime} + c_{deployDur}, & \text{if deployment needed} \\ opt_{endTime}, & \text{otherwise} \end{cases}$$
$$b_{upper} = step_{ens} - step_{duration} \tag{3}$$

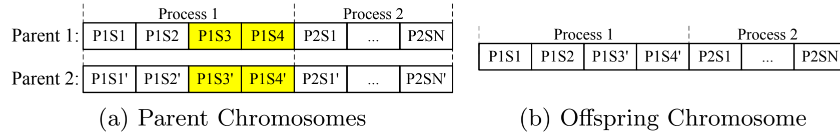
2. *step* is between two other steps: This situation is defined by Eq. (4) whereas $step_{lps}$ defines the latest end time of the preceding steps. Again for b_{lower} it has to be considered that maybe an additional deployment time is needed for the container, e.g., *step* is the second step in the process and the preceding step duration is shorter than the deployment time. This is considered by $additionalDeployDur = c_{deployDur} - (step_{lps} - opt_{endTime})$, where $c_{deployDur}$ and $opt_{endTime}$ are defined as for Eq. (3).

$$b_{lower} = \begin{cases} step_{lps} + additionalDeployDur, & \text{if deployment needed} \\ step_{lps}, & \text{otherwise} \end{cases}$$
$$b_{upper} = step_{ens} - step_{duration} \tag{4}$$

3. *step* is the last one in the process: This situation is defined by Eq. (5) whereas *deadline* is the deadline of the process and *additionalTime* a configurable time (e.g., 1 hour). The remaining terms are defined as for Eq. (3) and (4).

$$b_{lower} = \begin{cases} step_{lps} + additionalDeployDur, & \text{if deployment needed} \\ step_{lps}, & \text{otherwise} \end{cases}$$
$$b_{upper} = deadline + additionalTime \tag{5}$$

The final random time, which lies in the interval (b_{lower}, b_{upper}), is then used to adapt the start time of *step*, which resolves in a new chromosome.



(a) Parent Chromosomes (b) Offspring Chromosome

Fig. 2. Two-Point Crossover Operation

Crossover The crossover operation is a genetic algorithm operation that creates a new chromosome by splitting two chromosomes, called parent chromosomes, and combining them to create a new chromosome, called offspring chromosome. Our crossover operation is a two-point crossover [34] where the first point is a random gene in the chromosome, i.e., a random step of a random process, and the second position is the end of the process to which the step belongs.

Fig. 2 shows an example two-point crossover operation. The parent chromosomes are depicted in Fig. 2a. The highlighted genes show the genes that are selected for the crossover, i.e., P1S3 is the randomly selected gene and P1S4 is the end of the process (P1S3' and P1S4' are the representative steps in Parent 2). Eventually, the crossover operation results in the chromosome represented by Fig. 2b. It can be seen that the selected genes are taken from Parent 2 (i.e., P1S3' and P1S4') and the remaining genes are taken from Parent 1.

After the crossover is performed, a check if the offspring chromosome does violate the process structure, defined by the process model, or does not consider the deployment duration of the first container, is performed. If one of those constraints is violated, another crossover point, i.e., another gene, is selected. If both constraints are fulfilled, the offspring chromosome is returned.

4 Evaluation

As a proof of concept, GeCo has been thoroughly evaluated with respect to its allocation efficiency and SLA compliance. As underlying eBPMS, we apply the platform ViePEP [14,27]. A process owner can request the execution of processes by ViePEP, which then uses VMs on cloud resources for the execution of the services corresponding to the steps. Each service type has thereby an own VM image, similar to a container image as discussed in Sect. 2. Each VM can be deployed at any time, and the corresponding services can be invoked several times concurrently.

For our evaluation, we extend ViePEP in a way that it uses containers for the execution of the services. ViePEP and the services are written in Java. As application server for the services, we use Jetty. Each service uses the load generator library *fakeload*[3], which simulates a configurable RAM and CPU load for a given time span. For the evaluation, ViePEP is running on a VM on the Microsoft Azure Cloud[4] and the containers are deployed on the Azure Cloud

[3] https://github.com/msigwart/fakeload
[4] https://azure.microsoft.com

by using the functionality Container Instances. As container registry, the Azure Cloud Container Registry is used. The source code of ViePEP, the services, and the evaluation client are available at https://github.com/piwa/ViePEP-C, https://github.com/piwa/ViePEP-C-Backendservice, and https://github.com/piwa/ViePEP-C-Testclient.

4.1 Evaluation Setting

The evaluation is based on an adapted version of the settings used in [14].

Test Collection For the evaluation, we choose 10 representative process models from the SAP reference process models [6,18]. The SAP reference models are a common testset in the BPM field [22]. The 10 selected process models possess different levels of complexity. Table 1a presents the characteristics of the selected models by showing the number of steps, XOR-blocks, AND-blocks, and repeat loops. Each XOR- and AND-block contains a split and join gateway.

While the SAP reference models also contain human-provided services, in our evaluation we only use software-based services. For the evaluation, we use 8 different software service types, each with different resource requirements and execution durations. Table 1b summarizes the required CPU load (in percentage) and total makespan (in seconds) of the services for one invocation. We further consider that the actual CPU load and execution duration of a service can vary to some extend. To consider this variation, we assume a normal distribution $\sigma_1 = \mu_{cpu}/10$ of the CPU load and $\sigma_2 = \mu_{makespan}/10$ of the total makespan, both with a lower bound of 95% and an upper bound of 105%.

To consider that a service needs more computational resources when it is invoked several times in parallel, we add for each invocation 2/3 of μ_{cpu} defined in Table 1b. For instance, if a service of type 1 is invoked two times in parallel the resource requirements are $\mu_{cpu} = 15 + 15 * 2/3 = 25$, while the service makespan stays the same. This way we consider that for a second invocation no additional resources are required for secondary software, e.g., the application server. Since most cloud providers have a upper resource limit for container instances (e.g., the CPU limit is 4 cores at Azure) a second container is deployed if the amount of invocations requires more resources than this limit.

In addition, we assume that each service is stateless and fully parallelizable among the available CPUs. For instance, if a service has a load of 100% on a single-core CPU, then the same service has a 50% load on a dual core (i.e., 50% on each core) with the same execution time.

Applied SLAs To tolerate some execution delay, due to the container deployment time and service start-up time, we apply a *lenient* and a *strict* SLA level as process execution deadline. In the lenient scenario, the deadline is 2.5 times of the average makespan of the whole process. The average makespan is the time that the execution of the whole process needs by considering the historical execution duration of each step and the order of them defined by the process model. In the strict scenario, the deadline is 1.5 times of the average makespan.

Table 1. Evaluation Process Models and Service Types

(a) 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	1	0	0
6	9	1	1	0
7	9	1	0	0
8	3	0	1	1
9	4	1	1	1
10	20	0	4	0

(b) Evaluation Services

Service No.	CPU Load in % (μ_{cpu})	Service Makespan in sec. ($\mu_{makespan}$)
1	15	40
2	20	320
3	25	480
4	40	80
5	55	400
6	65	120
7	80	160
8	135	80

Process Request Arrival Patterns For the evaluation, we apply two different process request arrival patterns. The first one, called *constant* arrival pattern, requests in a 240 second interval the execution of 5 different processes. The processes are selected from Table 1a in a round-robin fashion. This is repeated 20 times, which results in the execution of 100 processes.

The second request arrival pattern, called *pyramid* arrival pattern, follows the pattern shown in Eq. (6), where n represents a point in time and a the amount of process execution requests at this time. The processes are again selected from Table 1a in a round-robin fashion in a 120 second interval.

$$f(n) = a \begin{cases} 1 & \text{if } 0 \leq n \leq 3 \text{ and } 20 \leq n \leq 35 \\ \lceil (n+1)/4 \rceil & \text{if } 4 \leq n \leq 17 \\ 0 & \text{if } 18 \leq n \leq 19 \\ \lceil (n-9)/20 \rceil & \text{if } 36 \leq n \leq 51 \end{cases} \tag{6}$$

Baseline We compare our optimization approach against a baseline that uses an ad hoc process execution approach. The baseline executes each process without task scheduling, but by executing one step after another according to the process model. However, for each new container that has to be deployed, it is checked if a compatible container is already deployed. If this is the case, this container is used, otherwise, a new container is deployed. When applying the baseline, containers are also deployed while a possible preceding step is running. This baseline is an adapted version of the *OneVMPerTask* provisioning strategy presented in [11].

Genetic Algorithm Parameter Setting For the evaluation, we use a population size of 2,000 with 100 elite chromosomes per population. The optimization duration is set to 40 seconds. Furthermore, we set $f_{leasing} = 10$ (see Eq. (1)), $f_{penalty} = 0.001$ (see Eq. (2)), and $additionalTime = 1\ hour$ (see Eq. (5)). These settings resolved in promising results in several pre-evaluation executions.

Table 2. Evaluation Results (Standard Deviation in Parenthesis)

	Constant Arrival Pattern				Pyramid Arrival Pattern			
	GeCo		Baseline		GeCo		Baseline	
SLA Level	Strict	Lenient	Strict	Lenient	Strict	Lenient	Strict	Lenient
SLA Adherence (%)	82.67 (3.06)	96.67 (2.08)	90.00 (0.00)	100.00 (0.00)	83.67 (2.31)	97.67 (1.15)	90.00 (0.00)	100.00 (0.00)
Process Makespan (min)	21.02 (15.20)	30.16 (22.97)	16.96 (12.74)	16.90 (12.69)	21.42 (15.92)	31.75 (25.46)	17.01 (12.73)	17.03 (12.75)
Leasing Cost	360.25 (19.71)	330.00 (15.22)	547.69 (16.26)	554.50 (9.32)	316.74 (2.67)	300.67 (13.90)	612.58 (17.67)	624.54 (31.94)
Penalty Cost	52.67 (5.77)	5.33 (5.13)	13.33 (0.58)	0.00 (0.00)	35.0 (4.58)	3.0 (1.0)	10.00 (0.00)	0.00 (0.00)
Total Cost	412.91 (14.56)	335.34 (17.82)	560.70 (16.26)	554.50 (9.32)	351.74 (7.13)	303.67 (14.60)	622.58 (17.67)	624.54 (31.94)

Metrics As evaluation metrics we use the average *process makespan*, the *SLA adherence*, and the *total cost* of the execution. The process makespan is the duration between receiving a process execution request and the termination of the final step in minutes. The SLA adherence determines how many processes terminated without violating the deadline in percentage.

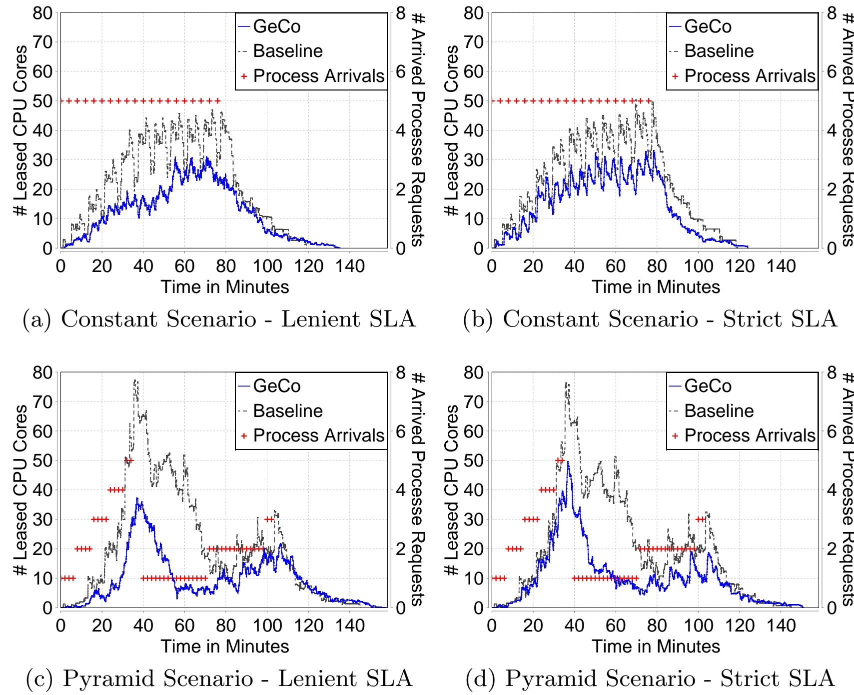
The total cost is composed of the *leasing cost*, i.e., the charged cost due to the leasing of the cloud resources, and the *penalty cost*, i.e., the charged cost due to a deadline violation. The penalty cost is calculated by a linear model based on [19]. This model assigns 1 unit of penalty cost for a delay of 10% of time units, i.e., seconds, of delay. For the leasing cost we use €0.0043 per GB-second and €0.0127 per CPU-second which are adapted real-world values from Azure prizing model[5]. For all evaluation results, we provide the mean value and the standard deviation.

4.2 Results and Discussion

In the evaluation, we evaluate the efficiency of our optimization approach, in comparison to the baseline, with both process request patterns and both SLA levels. Each evaluation step is performed three times over a timespan of 7 days to minimize external influences. The results of the evaluation are presented in Table 2. In addition, the constant arrival pattern results are presented in Fig. 3a and 3b and the pyramid arrival pattern results in Fig. 3c and 3d.

First, we discuss the constant arrival pattern results for both SLA levels, i.e., strict and lenient. In Table 2, it can be seen that our GeCo approach results in a lower SLA adherence than the baseline. This is due to the fact that the baseline executes each step directly after the preceding one without any delay between them. GeCo, in contrast, postpones the execution of a step if a more resource-efficient execution can be achieved. The postponing of the step execution in case

[5] https://azure.microsoft.com/en-us/pricing/details/container-instances/



(a) Constant Scenario - Lenient SLA
(b) Constant Scenario - Strict SLA
(c) Pyramid Scenario - Lenient SLA
(d) Pyramid Scenario - Strict SLA

Fig. 3. Evaluation Results

of GeCo also results in a longer process makespan as observable in Table 2. While the process makespan stays nearly the same for both SLA levels in case of the baseline (i.e., 16.96 and 16.90 min), it varies in case of GeCo. Since GeCo has less room to postpone a step (before the deadline is violated) in case of the strict SLA level the process makespan is shorter for this SLA level, i.e., 21.02 min, than for the lenient one, i.e., 30.16 min.

While the baseline SLA adherence is better than in case of GeCo, GeCo achieves in average 37.37% lower leasing cost than the baseline. This cost-saving results from a better resource utilization achieved by GeCo. This can also be seen in Fig. 3a and Fig. 3b, where GeCo leases fewer CPU cores than the baseline.

The lower SLA adherence, achieved by GeCo, results in higher penalty cost. However, the baseline results for both SLA levels in higher total cost, due to higher leasing cost. The total cost saving of GeCo, in comparison to the baseline, is 26.35% in case of the strict SLA level and 39.52% in case of the lenient one. This cost saving was achieved despite higher penalty cost than in case of the baseline. This shows us that in some situations accepting some penalty cost can help to reduce the leasing cost, and thus the total cost, to quite some extent.

Second, we discuss the pyramid arrival pattern results. As can be observed in Table 2, the SLA adherence and process makespan stayed nearly the same, in

comparison to the results of the constant arrival pattern. However, the leasing cost of GeCo and the baseline changed significantly. In case of GeCo, it can be seen that the pyramid arrival pattern allows GeCo to reduce the resource requirements even more. For the baseline the leasing cost increased, in comparison to the constant arrival pattern. This can be explained by the fact that the constant arrival pattern results in more parallel running processes, which increases the chances that an already running container can be re-used. In Fig. 3c and Fig. 3d, it can also be observed that the pyramid arrival pattern allows GeCo to achieve a much more resource-efficient solution than the baseline.

Table 2 shows that the penalty cost for GeCo and the pyramid arrival pattern with the strict SLA are lower than for the constant arrival pattern with the strict SLA, while the SLA adherence of both is nearly the same. This shows us that a similar amount of process deadlines are violated, however, in case of the pyramid arrival pattern the average duration between the end of the processes and deadlines is smaller. The total cost saving of GeCo in comparison to the baseline are 43.50% in case of the strict SLA level and 51.38% in case of the lenient one.

In summary, the best result, in respect of total cost, was achieved by our GeCo approach in case of the pyramid arrival pattern with the lenient SLA level with a cost saving of 51.38% and an SLA adherence of 97.67%. The second best result was achieved in case of the pyramid arrival pattern with the strict SLA level with a cost saving of 43.50% and SLA adherence of 83.67%. In respect of the SLA adherence our GeCo approach achieved for both lenient SLA levels an average of 83.17% and 97.17% for the strict one.

5 Related Work

The research in this paper relates to work on elastic BPM. In the following, we discuss selected contributions to this research field.

In our former work, we have presented the eBPMS platform ViePEP [27]. ViePEP offers several resource optimization and task scheduling approaches. Those approaches perform a resource allocation optimization and task scheduling for cost-efficient process execution while considering predefined SLAs. Especially, the work presented in [14] has to be mentioned, where the task scheduling and resource optimization problem is formulated as a Mixed Integer Linear Program (MILP). However, until now ViePEP relies on VMs as an execution environment for the services, which results in a much more coarse-grained deployment and may increase cost in comparison to the work at hand.

In [17], the authors present a BPEL engine containing a scheduling algorithm for the cost-efficient execution of process steps on cloud resources in the form of VMs. The scheduling algorithm is based on a genetic algorithm and allows the execution and optimization of several processes in parallel. The presented scheduling algorithm considers the leasing cost of the VMs and the data transfer duration. However, since the approach uses VMs instead of containers, it results in a much more coarse-grained solution and may increase resource

cost. Furthermore, in comparison to our work, it does not consider user-defined SLAs, e.g., the deadline until when the process has to be done. The same applies to the approach presented in [2]. In this work, the authors present a process scheduling approach that allows parallel execution of processes. The presented approach aims for a cost or execution time optimization or a pareto-optimal solution covering both, cost and execution time. Again, this approach uses VMs as the execution environment of the process steps and does not consider the process execution deadline. Hence, it is not possible to perform an optimization by postponing the execution of particular steps to the future as GeCo allows.

Wei and Blake present in [32] a resource utilization optimization approach that considers service levels. However, in contrast to our work, the approach does not consider a deadline for the process execution. Furthermore, while our approach follows the "classic" service composition model that allows the invocation of a service instance from different processes, this is not allowed in [32].

An approach for supporting a customer in finding the optimal cloud pricing strategy is presented in [13]. The approach selects a cost-efficient resource configuration (i.e., RAM and CPU size), the cloud provider, and the cloud pricing model (e.g., on-demand VM or reserved VM). Similar to our approach, the aim of the approach is to reduce the execution cost, including possible penalty cost, while considering QoS constraints and without violating temporal constraints. However, in comparison to our approach, the temporal constraints are on process step level and not on the process level.

In [25], the author's present migration-aware optimization strategies for multi-tenant process execution in the cloud. The presented strategies migrate a tenant, including the corresponding processes, from one BPMS to a different BPMS if the process execution needs more cloud resources or a new cost-efficient solution can be achieved. The BPMSs are thereby executed on VMs. The paper presents a linear optimization model and a heuristic optimization approach. The optimization aims to minimize the resource consumption while maintaining an acceptable migration amount for each tenant. While our approach considers the structure of the process and optimizes the execution of the steps on containers, the approach presented in [25] is more coarse-grained since it migrates the process as a whole and does not consider the single steps.

In [30] an approach for optimal resource provisioning for enterprise applications on cloud resources is presented. The authors present a linear program that finds the optimal setup of VM instances that minimizes the resources consumption while still able to fulfill all incoming requests. In comparison to our approach, which performs task scheduling to reduce the execution cost, their approach reduces the cost by finding the optimal VM instance configuration.

All of the aforementioned approaches are using VMs as an underlying execution environment and do not consider containers. As has been mentioned before, this leads to a more coarse-grained resource allocation, which may result in increased resource cost. For the usage of containers for scheduling and resource allocation of arbitrary services, several solutions have been proposed, e.g., [15,24,31,23]. Those publications discuss the usage of containers for scalable

and isolated application execution in the cloud. However, none of them consider the execution of processes that are composed of several, interdependent steps.

In [5], the authors present a linear program that finds a global optimal cost-efficient solution for the deployment of a business process on containers. These containers are then deployed on VMs running on cloud resources. While this approach is the most comparable to the work at hand, the algorithm does not consider SLAs and considers only one process at a time. Thus, the usage of the same container for several service invocations is not facilitated. Moreover, in comparison to our work, the containers are deployed on VMs. For the evaluation, the authors extended ContainerCloudSim, while we make use of a cloud-based testbed for the evaluation.

In [12], the scientific workflows (SWF) platform Skyport is presented. The platform uses Docker containers for the deployment of the workflow services to achieve a reproducible software deployment solution with isolated software applications. In [37], the authors present a two-level resource scheduling model for an efficient resource sharing among different SWFs. They show that a container-based scheduling platform increases the system efficiency while decreasing the risk of performance issues. However, the differences between SWFs and business processes prevent a direct adaptation of the approaches for our purposes [20].

To sum things up, most of the presented publications are considering VMs as execution environment. This leads to rather coarse-grained deployment solutions in comparison to a container-based deployment as provided by GeCo. In addition, most of the above-discussed publications do not consider the usage of an already deployed container, respectively the deployed service instance, several times. This further reduces the resource consumption and, thus, the overall cost.

6 Conclusion

Within this paper, we present a novel scheduling approach for the fine-granular execution of process steps on containers, which are deployed on cloud resources. This scheduling approach aims for a resource-efficient execution while considering user-defined SLAs, by scheduling the execution times of the steps of a complete process landscape. The resulting schedule minimizes the overall execution cost, which is a composition of the cloud resource leasing cost and penalty cost. The presented optimization approach, called GeCo, is based on a genetic algorithm. Our evaluation has shown that using such an optimization approach results in reduced leasing cost in comparison to an ad hoc baseline solution: In average our optimization approach issues 47.45% less cost for the pyramid arrival pattern and 32.90% less cost for the constant arrival pattern.

In our future research, we want to further analyze different genetic algorithm parameter settings. Furthermore, we want to examine different approaches for an automatic selection of the genetic algorithm parameter settings by using, for instance, machine learning. Another crucial point is the prediction of the service execution duration and the required computational resources. We will analyze in this respect how a combination of monitoring and prediction can be made.

Acknowledgments. This work is partially funded by FFG – Austrian Research Promotion Agency (FFG – project number: 866270).

References

1. Bernstein, D.: Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Computing **1**(3), 81–84 (2014)
2. Bessai, K., Youcef, S., Oulamara, A., Godart, C.: Bi-criteria strategies for business processes scheduling in cloud environments with fairness metrics. In: 7th Int. Conf. on Research Challenges in Information Science (RCIS). pp. 1–10 (2013)
3. Bhandari, D., Murthy, C., Pal, S.K.: Variance as a stopping criterion for genetic algorithms with elitist model. Fundamenta Informaticae **120**(2), 145–164 (2012)
4. Borkowski, M., Schulte, S., Hochreiner, C.: Predicting cloud resource utilization. In: 2016 IEEE/ACM 9th Int. Conf. on Utility and Cloud Computing (UCC). pp. 37–42 (2016)
5. Boukadi, K., Grati, R., Rekik, M., Abdallah, H.B.: From VM to Container: A Linear Program for Outsourcing a Business Process to Cloud Containers. In: OTM Confederated Int. Conf. On the Move to Meaningful Internet Systems. LNCS, vol. 10573, pp. 488–504 (2017)
6. Curran, T.A., Keller, G.: SAP R/3 Business Blueprint: Understanding the Business Process Reference Model. Prentice Hall PTR, Upper Saddle River (1997)
7. Diaz-Gomez, P.A., Hougen, D.F.: Initial population for genetic algorithms: A metric approach. In: Proceedings of the 2007 Int. Conf. on Genetic and Evolutionary Methods (GEM 2007). pp. 43–49 (2007)
8. Dua, R., Raja, A.R., Kakadia, D.: Virtualization vs Containerization to Support PaaS. In: 2014 IEEE Int. Conf. on Cloud Engineering. pp. 610–614 (2014)
9. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management, Second Edition. Springer (2018)
10. Dustdar, S., Guo, Y., Satzger, B., Truong, H.L.: Principles of Elastic Processes. IEEE Internet Computing **15**(5), 66–71 (2011)
11. Frincu, M.E., Genaud, S., Gossa, J.: On the efficiency of several vm provisioning strategies for workflows with multi-threaded tasks on clouds. Computing **96**(11), 1059–1086 (2014)
12. Gerlach, W., Tang, W., Keegan, K., Harrison, T., Wilke, A., Bischof, J., D'Souza, M., Devoid, S., Murphy-Olson, D., Desai, N., Meyer, F.: Skyport: Container-based execution environment management for multi-cloud scientific workflows. In: 5th Int. Works. on Data-Intensive Computing in the Clouds. pp. 25–32 (2014)
13. Halima, R.B., Kallel, S., Gaaloul, W., Jmaiel, M.: Optimal Cost for Time-Aware Cloud Resource Allocation in Business Processes. In: 14th Int. Conf. on Serv. Computing. pp. 314–321 (2017)
14. Hoenisch, P., Schuller, D., Schulte, S., Hochreiner, C., Dustdar, S.: Optimization of complex elastic processes. IEEE Trans. on Serv. Computing **9**(5), 700–713 (2016)
15. Hoenisch, P., Weber, I., Schulte, S., Zhu, L., Fekete, A.: Four-fold auto-ccaling on a contemporary deployment platform using docker containers. In: 13th Int. Conf. on Service-Oriented Computing. pp. 316–323. LNCS (2015)
16. Hou, E.S., Ansari, N., Ren, H.: A genetic algorithm for multiprocessor scheduling. IEEE Trans. on Parallel and Distributed systems **5**(2), 113–120 (1994)
17. Juhnke, E., Dörnemann, T., Bock, D., Freisleben, B.: Multi-objective Scheduling of BPEL Workflows in Geographically Distributed Clouds. In: 4th Int. Conf. on Cloud Computing. pp. 412–419 (2011)

18. Keller, G., Teufel, T.: SAP R/3 Process Oriented Implementation: Iterative Process Prototyping. Addison-Wesley Longman Publishing Co. (1998)
19. Leitner, P., Hummer, W., Satzger, B., Inzinger, C., Dustdar, S.: Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud. In: 5th Int. Conf. on Cloud Computing. pp. 213–220 (2012)
20. Ludäscher, B., Weske, M., McPhillips, T., Bowers, S.: Scientific workflows: Business as usual? In: Int. Conf. on Business Process Management. pp. 31–47 (2009)
21. Mao, M., Humphrey, M.: A performance study on the VM startup time in the cloud. In: 5th Int. Conf. on Cloud Computing. pp. 423–430 (2012)
22. Mendling, J., Verbeek, H., van Dongen, B.F., van der Aalst, W.M.P., Neumann, G.: Detection and prediction of errors in EPCs of the SAP reference model. Data & Knowledge Engineering **64**(1), 312–329 (2008)
23. Nardelli, M., Hochreiner, C., Schulte, S.: Elastic provisioning of virtual machines for container deployment. In: 8th ACM/SPEC on Int. Conf. on Performance Engineering Companion. pp. 5–10 (2017)
24. Pahl, C.: Containerization and the PaaS Cloud. IEEE Cloud Computing **2**(3), 24–31 (2015)
25. Rosinosky, G., Youcef, S., Charoy, F.: Efficient migration-aware algorithms for elastic bpmaas. In: 15th Int. Conf. of Business Process Management. vol. 10445, pp. 147–163 (2017)
26. Schulte, S., Hoenisch, P., Hochreiner, C., Dustdar, S., Klusch, M., Schuller, D.: Towards process support for cloud manufacturing. In: Int. Enterprise Distributed Object Computing Conf. pp. 142–149. IEEE (2014)
27. Schulte, S., Hoenisch, P., Venugopal, S., Dustdar, S.: Introducing the Vienna Platform for Elastic Processes. In: Performance Assessment and Auditing in Service Computing Works. at 10th Int. Conf. on Service-Oriented Computing. vol. 7759, pp. 179–190 (2013)
28. Schulte, S., Janiesch, C., Venugopal, S., Weber, I., Hoenisch, P.: Elastic business process management: State of the art and open challenges for BPM in the cloud. Future Generation Computer Sys. **46**, 36–50 (2015)
29. Seo, K.T., Hwang, H.S., Moon, I.Y., Kwon, O.Y., Kim, B.J.: Performance comparison analysis of linux container and virtual machine for building cloud. Advanced Science and Technology Letters **66**(105-111), 2 (2014)
30. Srirama, S.N., Ostovar, A.: Optimal resource provisioning for scaling enterprise applications on the cloud. In: Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on. pp. 262–271. IEEE (2014)
31. Vaquero, L.M., Rodero-Merino, L., Buyya, R.: Dynamically scaling applications in the cloud. ACM SIGCOMM Computer Comm. Review **41**(1), 45–52 (2011)
32. Wei, Y., Blake, M.B.: Proactive virtualized resource management for service workflows in the cloud. Computing **96**(7), 1–16 (2014)
33. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer, 2nd edn. (2012)
34. Whitley, D.: A Genetic Algorithm Tutorial. Stat. Computing **4**, 65–85 (1994)
35. Ye, Z., Zhou, X., Bouguettaya, A.: Genetic algorithm based QoS-aware service compositions in cloud computing. In: Int. Conf. on Database Systems for Advanced Applications. pp. 321–334 (2011)
36. Yoo, M.: Real-time task scheduling by multiobjective genetic algorithm. Journal of Systems and Software **82**(4), 619–628 (2009)
37. Zheng, C., Tovar, B., Thain, D.: Deploying high throughput scientific workflows on container schedulers with makeflow and mesos. In: 17th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing. pp. 130–139 (2017)