ViePEP-C: A Container-based Elastic Process Platform

Philipp Waibel, Christoph Hochreiner, Stefan Schulte, Agnes Koschmider, and Jan Mendling

Abstract—Business Process Management Systems (BPMS) need to be able to take into account the fluctuating demand for computational resources during the execution of business process activities. Today, BPMS rely on the leasing and releasing of virtual machines (VMs) on cloud resources, which leads to a rather coarse-grained allocation of computational resources. This may result in an increase in the execution cost, flexibility restrictions, and a negative impact on the Quality of Service.
In order to overcome these drawbacks, we introduce the Vienna Platform for Elastic Processes on Containers (ViePEP-C). ViePEP-C is an elastic BPMS that uses containers instead of VMs for the execution of business process activities on cloud resources, leading to a more fine-grained execution environment. To achieve this, ViePEP-C offers cloud controller, monitoring and business process execution functionalities and provides a platform for different resource and task scheduling algorithms. To evaluate the benefits of ViePEP-C, we further present a resource and task scheduling algorithm and show that, by using containers as execution environment, the execution cost can be decreased by over 20% (compared to a state-of-the-art VM-based scheduling algorithm) while considering a high service level.

Index Terms—Cloud Computing, Containers, Elastic Processes, Business Process Management

1 Introduction

Business processes are an integral part of any type of organization, including product ordering, complex manufacturing [1], [2], and various other kinds of processes. Business process management (BPM) and BPM systems (BPMS) support organizations with a flexible approach of composing services [3] by the help of control structures such as decision points, concurrent paths or loops [4], and by providing components for the execution and monitoring of business processes [5].

It is a challenge for a BPMS to deal with changing resources requirements during the execution of processes, which might leave the system often in a state of *over-* or *under-provisioning* of computational resources [6]. Over-provisioning of resources that are not utilized entail unnecessary cost, while under-provisioning harms the quality of services and leads to the denial of service in the worst case [6]. Cloud computing can help to address these problems, when designed in a way that is appropriate for the application at hand. Concepts of utilizing cloud computing for BPMS are referred to as *elastic* BPMS (eBPMS) and enacted processes are called *elastic processes* [6].

State-of-the-art eBPMS as suggested by, e.g., [7], [8], [9], apply virtual machines (VMs) as computational entities for the execution of elastic processes. However, VMs as heavy-weight entities have disadvantages in terms of deployment and start-up time, because of the requirement of an own operating system. Those disadvantages lead to increased deployment cost and reduce the flexibility and ad hoc elasticity [10]. *Containers* promise to provide various advantages over VMs such as no need for an own operating system [11], [12]. This leads to a more lightweight solution that deploys faster and, thus, adapts faster to changing requirements [13], [14], [15].

As presented in different other publications, e.g., [7], [16], [17], [18], the usage of resource and task scheduling algorithms during the execution of processes further lower the cost of process execution. To reap the benefits of containers, such as exemplified for data stream processing [19] or service-oriented computing [20], [21], appropriate algorithms have to be devised. These algorithms have to be able to cope with lightweight containers as entities of a potentially much higher amount as compared to coarse-grained VM-based solutions.

In this paper, we focus on how container technologies can be used to execute processes in a cost-efficient way. Our contribution is twofold. First, we present the novel *Vienna Platform For Elastic Processes on Containers (ViePEP-C)*, which represents a generic architecture that offers support for container-based scheduling of business processes and their activities. Second, we specify a novel scheduling algorithm for optimizing the execution of business processes and container deployments. The algorithm, called GeCoVM, is based on a genetic algorithm. Our evaluation demonstrates the efficiency of our algorithm by comparing it with state-of-the-art VM-based approaches in terms of the accruing amount of cloud resources and deployment times.

The remainder of this paper is organized as follows: Section 2 presents a motivational scenario to define the requirements of an eBPMS. In Section 3, we discuss relevant background information. Subsequently, in Section 4, we

- *P. Waibel, C. Hochreiner and S. Schulte are with the Distributed Systems Group, TU Wien, Austria.*
  E-mail: {p.waibel,c.hochreiner,s.schulte}@infosys.tuwien.ac.at
- *A. Koschmider is with the Institute of Applied Informatics and Formal Description Methods, KIT, Karlsruhe, Germany.*
  E-mail: agnes.koschmider@kit.edu
- *J. Mendling is with the Institute for Information Business, WU Wien, Austria.*
  E-mail: jan.mendling@wu.ac.at

Manuscript received September, 2018

describe the architecture of our eBPMS. The scheduling algorithm and the implementation of ViePEP-C are presented in Sections 5 and 6, respectively. Afterwards, we discuss the results of our evaluation in Section 7. Section 8 presents the related work and Section 9 concludes this paper and discusses our future work.

2 Motivational Scenario

To motivate our work, we use a scenario from the manufacturing industry. The scenario is taken from use cases observed in the European H2020 project CREMA (Cloud-based Rapid Elastic Manufacturing) [2]. The goal of CREMA is to offer *Cloud Manufacturing*-based, inter-organizational manufacturing processes. Cloud Manufacturing applies basic principles from the field of cloud computing and virtualization in order to achieve agile and scalable manufacturing processes [22], [23]. In brief, this is done by porting well-known approaches from the field of cloud computing to the manufacturing domain, so that it is possible (i) to lease and release manufacturing assets in an on-demand, utility-like fashion, (ii) to achieve rapid elasticity of manufacturing processes through scaling leased assets up and down, and (iii) to achieve pay-per-use through metered services.

In Cloud Manufacturing, manufacturing processes are composed from single process activities represented by their *digital twins*, e.g., a virtualized manufacturing machine [24]. After a digital twin has been modeled, it can be used as a service in manufacturing processes, thus allowing to integrate real-world data (e.g., from sensors) into a virtual representative of an asset. In these service-oriented processes, single manufacturing assets are integrated in a similar way as software and platform services are provided following the Software-as-a-Service (SaaS) or Platform-as-a-Service (PaaS) paradigms [25]. In addition, manufacturing processes may include software services which have no representation on the shopfloor, e.g., services for data analytics, automated ordering of parts, creation of invoices, model rendering, image processing, or product and process optimization. These software services can reach from simple calculations to more complex and long-running analytical tasks.

As it can be seen, it is therefore necessary and possible to execute and monitor real-world manufacturing processes by executing the single software services representing the activities in a process model. Notably, in order to provide and analyze real-time monitoring data about the status of a process instance, each single manufacturing process is represented by a software-based counterpart, i.e., there is a software-based process instance for each real-world manufacturing process instance. Thus, software-based processes act as digital twins of real-world manufacturing processes.

A graphical representation of our motivational scenario is illustrated in Fig. 1. As an example, we consider a company, called CarOne, that produces car parts, e.g., seats or doors. CarOne has several suppliers and sub-companies that produce different parts and perform assembly steps. For the daily business CarOne uses a vast amount of process models, e.g., that defines the steps that are needed to assemble the product or handle warehouse logistics. Processes range from small short-running processes, like assembling a connector, to more complex long-running inter-

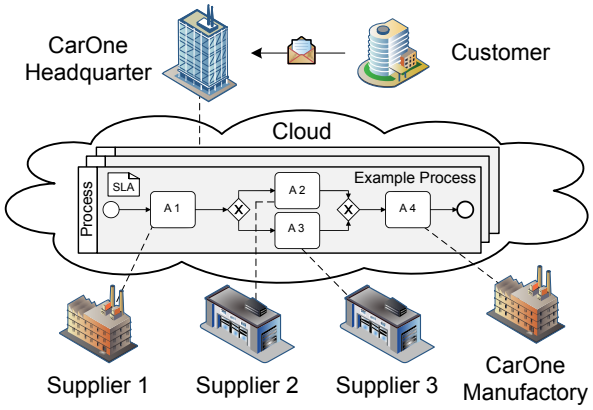


Fig. 1. Motivational Scenario

organizational processes, like building a car seat. It has to be noted that to simplify Fig. 1, the depiction only contains a small example process model and only a sub-part of all supplier companies. In reality, CarOne is part of an extensive process network which includes several suppliers. Also, the figure shows a quite static setting. In reality, process models need to be changed and process instances need to be adapted, e.g., because suppliers are not available any longer or there have been delays in a particular supply chain.

Since CarOne has to be able to compete with its competitors, it offers the possibility of ordering customized parts with a very short time for delivery and in small quantities, as well as, products in large quantities and with some lead time. This can lead to high fluctuations regarding the number of orders especially during peak times, e.g., Monday morning when several orders come in, and off-peak times, e.g., during the holidays. Products are build to order, i.e., once an order request has been received, a process model is instantiated [2].

In addition to the product orders, CarOne gets supplies for their production delivered at specific times, e.g., Tuesday around lunchtime. To store those supplies in the warehouse different processes are used, e.g., optimization of the warehouse or route calculation for the autonomous forklifts.

Thus, the company has to be able to cope with a potentially very large (yet volatile) number of process instances with different priorities at different points in time. This can lead to high fluctuations with regard to the need for computational resources, e.g., to break down the customer orders into raw materials, optimize the manufacturing schedule, or create potential cascades of replenishment orders. The lightweight feature of containers provide here a valid solution to cope with these high fluctuations.

Independent of the current order situation, CarOne has to fulfill different quantitative constraints. For instance, each order contains a defined deadline, i.e., the point in time until when the product has to be done.

To sum up the motivational scenario, we can derive the following requirements which need to be fulfilled by an eBPMS in order to support a Cloud Manufacturing-enabled process landscape:

- To dynamically react to the ever-changing needs for computational resources, cloud resources should be used for the execution of the process activities.

- Process instances may be enacted at any time.
- The execution of the process instances should consider Service Level Agreements (SLAs), e.g., a deadline until when the execution of a process instance has to be done. If the SLAs are violated, penalty cost may occur.
- For a resource- and cost-efficient execution of the processes, resource allocation and task scheduling optimization should be provided.
- The available cloud resources should be shared and optimized among all running process instances.
- The process instances and process activity execution should be monitored and corresponding countermeasures should be performed if required, e.g., if there have been delays in a process activity.

It has to be noted that the presented scenario is only for explanation purposes and should not be seen as exhaustive. Nor is the approach presented in this paper exclusively applicable to the manufacturing domain. In fact, the presented approach can be applied to any domain with an extensive process landscape, e.g., the Smart Grid [26], financial industry [27], or eHealth domains [28].

3 Background

3.1 Containers

As briefly mentioned in Section 1, a container is a lightweight, virtualized entity that does not need an own operating system. Instead, for each container, computational resources, e.g., physical hosts or VMs, are partitioned and an isolated user space instance is created that shares resources and an operating system with other containers [11], [12], [14]. Similar to a VM, a container is a configurable solution to execute software services in a virtualized manner, but in a resource-wise cheaper way due to the transparent operating system. This resource saving results in shorter startup, shutdown and migration times compared to rather heavyweight VMs. As shown in [29] the container deployment time can be six times lower than the one from a VM, and the memory footprint, i.e., the incremental memory cost of adding a container or VM, of a container can be up to eleven times lower than the one from a VM.

State-of-the-art container technologies enable to host potentially hundreds of containers on one physical machine [13]. These containers allow to offer self-contained services in an isolated computational environment. One popular use case for containers is hosting microservices.

For each container, a *container image* exists that contains the required configuration information and the executable software, i.e., the actual software of the service (e.g., a user login service) [11]. Container images are typically stored in a *container repository*. A user, or a system, can pull a container image from the repository and run it on a host system.

In comparison to a VM image, i.e., similar to a container image but on VM level, the size of a container image is up to three times smaller [30]. This smaller image size decreases the deployment time of a container in comparison to a VM.

3.2 Elastic Business Process Execution

"A business process consists of a set of activities that are performed in coordination in an organizational and technical environment" [5]. The order of the *process activities* is thereby defined in a *process model*. The process model structure can compose sequentially ordered process activities, as well as, more complex structures. More complex structures are, e.g., parallel or exclusive branches, or *loops* [4]. A parallel branch, called *AND-block*, is started by an AND-split and closed by an AND-join. The same goes for an exclusive branch, called *XOR-block*, which is started by an XOR-split and closed by an XOR-join.

If a process model is enacted, a new *process instance* is generated from the model. By using elastic cloud resources for the process execution, *elastic processes* are realized [6].

To execute a process activity, a software *service* is used. Each service is deployed in a container, resulting in a *service instance*. This service instance is then *invoked* during the process instance execution to fulfill a specific process activity.

3.3 Genetic Algorithms

A genetic algorithm is an iterative process that uses the principle of an evolutionary process by applying the genetic operations *selection*, *crossover*, and, *mutation* on each generation of possible solutions to a problem [31]. Each possible solution is called a chromosome, which is a composition of several genes. A composition of several chromosomes is called a generation. The size of a generation, i.e., the amount of chromosomes in a generation, is called population size.

During each iteration, a *fitness function* calculates a fitness score for each chromosome. This fitness score determines how well a given problem (e.g., process activity scheduling) is solved by a chromosome. This fitness score is used by the selection operator to select a subset of the generation as parent chromosomes for the next iteration. These parent chromosomes are then altered by the mutation and crossover operators to form a new generation of chromosomes. While the mutation operation changes random genes of a chromosome, the crossover operator swaps some genes of two chromosomes to form a new chromosome. In addition, each new generation gets a small number of unaltered elite chromosomes, i.e., chromosomes with the best fitness score, from the former generation.

The result of this iterative approach is the chromosome with the best fitness score that is achieved when a stop criterion is reached. With this approach, a genetic algorithm browses a large search space to find a near-optimal solution in polynomial time [32].

4 Design

To address the requirements defined in Section 2, we present ViePEP-C, an eBPMS that uses cloud resources for the execution of business processes.

4.1 General Approach

For the execution of the software services that compose elastic business processes, ViePEP-C uses containers on VMs which are hosted on cloud resources. By using VMs for the deployment of the containers, ViePEP-C becomes independent from cloud providers, since the containers can be deployed to arbitrary VMs. Due to the isolated user space characteristic of containers, one VM can host several

containers and still provides an isolated environment for each container, respectively its hosted service. This is not possible in a VM-based approach without containers since services executed in parallel on a single VM are not isolated from each other. Services executed in parallel on a VM without containers can lead, for instance, to conflicting library versions or filesystem accesses.

The general ViePEP-C approach is as follows: Clients are able to request a business process execution from ViePEP-C. Based on this request, the platform creates a process instance of the corresponding business process model. Subsequently, ViePEP-C analyzes the process structure (i.e., the given order of process activities) and SLA requirements, e.g., the process execution deadline, and creates a provisioning plan. The provisioning plan defines how the containers are allocated on VMs and how those VMs are hosted on cloud resources. The creation of the provisioning plan is done by a resource and task scheduling algorithm.

Since ViePEP-C can run multiple process instances concurrently, it considers all currently running and not yet running process instances, the corresponding SLAs, and the available cloud resources to create the provisioning plan.

After the provisioning plan is defined, ViePEP-C performs the tasks defined in the plan. Doing this, ViePEP-C deploys the required VMs, instantiates the container images on the VMs, and invokes the service instances on the containers. ViePEP-C further monitors the service instances and the execution environment, i.e., the containers and VMs. If the monitoring observes an unexpected behavior, e.g., failure of a service instance or a container, ViePEP-C performs corresponding countermeasures to guarantee the correct execution of the process instances. If required, ViePEP-C updates the provisioning plan during process runtime, e.g., if a new process execution request arrives or if a monitored event requires an update of the provisioning plan.

After a service instance completed its task, ViePEP-C continues with the execution of the subsequent process activity according to the provisioning plan. This is continued until all process activities of a process instance are completed, which results in the completion of the process instance and in the notification of the client (e.g., a customer or autonomous machine) that the execution is finished.

ViePEP-C belongs to the class of domain-agnostic process-aware information systems due to its universal approach and the generalized usability of business processes [1]. This means that ViePEP-C can be used in many different domains with an intensive process landscape, e.g., cloud manufacturing [2] (as discussed in Section 2), eHealth [28], or the financial industry [27].

Fig. 2 presents the high-level architecture of ViePEP-C. The architecture of ViePEP-C is based on our former work, called ViePEP [7], [33], [34]. Since ViePEP is not able to utilize containers to enact business processes, ViePEP-C constitutes a complete revision of the software. Several components had to be adapted, completely rewritten or were obsolete due to the new container-based approach. Which component had to be adapted or rewritten is depicted in Fig. 2 by different shadings.

As can be seen in Fig. 2, the platform consists of five top-level entities. Those entities are *Client API*, *eBPMS*, *Cloud Environment*, *Message Queue*, and *Container Registry*. In the following subsections, we will discuss those entities in detail. Notably, all components in ViePEP-C are loosely coupled and can be easily replaced by other implementations.

4.2 Client API

The *Client API* allows customers during design time to model and manage business processes. Moreover, the Client API is used to request the execution of processes. Process models are reusable, i.e., they can be stored and instantiated several times, even concurrently and by different process owners. ViePEP-C allows clients to model processes which include sequences as well as more complex process patterns, i.e., AND-blocks, XOR-blocks, and loops.

Besides the explicit request of process executions, an execution request can also be initialized by a triggered event, e.g., a customer places an order, or in predefined intervals, e.g., to automatically perform quality tests every five hours.

Each process execution request requires the presence of an SLA. Since the process execution can happen in an ad hoc manner or follows an interval approach, those SLAs can be set for each instantiation separately or defined as a default value for a process model. The default values are then used as long as no separate SLA has been defined for a particular process instance.

4.3 eBPMS

The second top-level entity is the eBPMS, which is responsible for the management of process instances and the execution environment, i.e., the cloud-based computational resources. For optimizing the utilization of cloud resources and to minimize the execution cost, the eBPMS entity performs resource allocation and task scheduling (see Section 4.3.2). Furthermore, the eBPMS monitors the execution of the service instances and conducts countermeasures in case of service instance or resource failures (see Section 4.3.6).

The eBPMS entity consists of several subcomponents:

4.3.1 Process Manager

The Process Manager is the entry component to the eBPMS. This component provides the interface that is used by the Client API to request the execution of a process. Each execution request includes the process model that should be enacted and the already discussed SLA. If a new request is received, the Process Manager creates a new process instance, writes the received information into the Database (see Section 4.3.8) and informs the Scheduler component (see Section 4.3.2) about the new process instance.

4.3.2 Scheduler

The Scheduler is responsible for creating and updating the provisioning plan that defines on which VM a container, holding a software service that represents a process activity, should be deployed and when its execution should start. Fig. 3 depicts an example provisioning plan that defines for five different process instances on which VM the container should be deployed, which cloud provider should be used

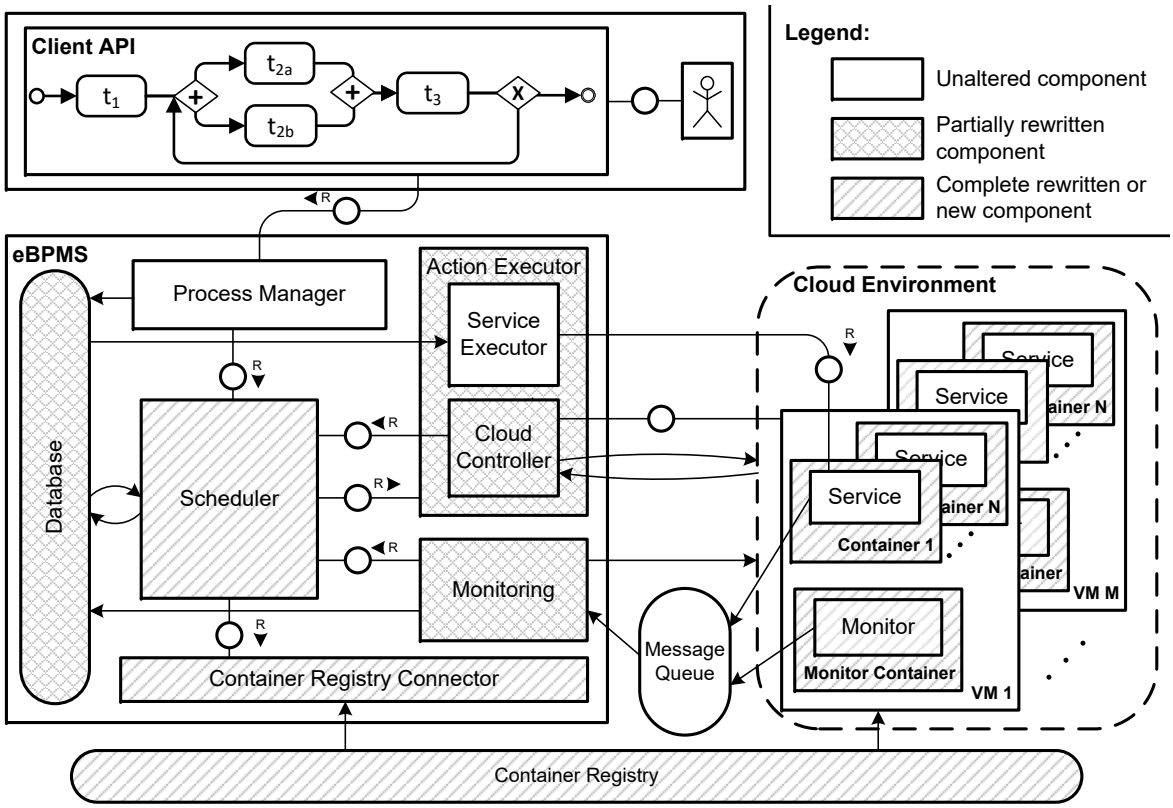


Fig. 2. ViePEP-C Architecture. The different shadings depict the degree of necessary adaptations, for the container-based execution, compared to our former eBPMS platform, called ViePEP.

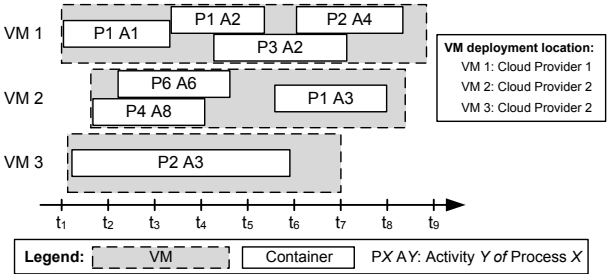


Fig. 3. Provisioning Plan Example

for the VMs and when the deployment and service invocation should be done.

The scheduling is done based on an algorithm, e.g., a genetic algorithm, or a formal model, e.g., following a Mixed-Integer Linear Programming approach. As mentioned before, all components in ViePEP-C are loosely coupled and can be easily replaced by other implementations. While we present a genetic algorithm within Section 5, the general approach and architecture of ViePEP-C is independent of the algorithm.

As input, the Scheduler gets the information about the running process instances, the not yet started process instances, the corresponding SLAs, the information how much computational resources (i.e., CPU, and RAM) a software service requires for a particular amount of invocations, how long the execution of the software service will be, and the monitoring data from currently running VMs and containers (e.g., resource utilization). How much computational resources a software service requires for a particular amount of invocations has to be known upfront, e.g., via monitored and historical data [35].

Eventually, the result of the scheduling is a provisioning plan that defines how the cloud resources should be used to execute all process instances while considering the SLAs.

The Scheduler is executed in predefined intervals, or if special events occur, e.g., if a new execution request arrives, a particular cloud resource is over-utilized, an SLA is violated, or a cloud resource fails. There are three different failure scenarios for which the Scheduler has to update the provisioning plan accordingly:

Failure of a container: For instance, the connection to a container is lost. Such a failure concerns the execution of the corresponding process activities and the corresponding process instances that are using the service on the particular container. If such a failure occurs, the provisioning plan has to be updated in a way that the container is redeployed and the execution of the service is restarted.

Failure of a VM: For instance, the connection to a VM is lost. Since a VM can host several containers, such a failure concerns all containers on that VM. Thus, the VM, as well as all affected containers, have to be redeployed and the service instances invoked.

The provisioning plan cannot be adhered to: For instance, the execution of a preceding process activity took longer than expected and, thus, the scheduled time of the next process activity cannot be adhered to. Such a failure affects all upcoming process activities of the affected process

instance. Thus, the provisioning plan has to be updated so that all upcoming process activity executions can be performed in the correct order, defined by the process model.

The detection of these different failure scenarios is the task of the Monitoring component (see Section 4.3.6).

4.3.3 Action Executor

The created provisioning plan is then handed over from the Scheduler to the Action Executor. As the name implies, this component is responsible for the execution of the provisioning plan. This includes the management of the cloud environment and the deployed VMs and containers. For this task, the component interacts with the Cloud Controller (see Section 4.3.4) and the Service Executor (see Section 4.3.5).

4.3.4 Cloud Controller

The Cloud Controller is responsible for the interaction with the cloud environment. This involves the tasks: (i) start a VM, (ii) stop a VM, (iii) deploy a container on a VM, and (iv) stop a deployed container. Which task has to be performed, and at which time, is defined in the provisioning plan.

4.3.5 Service Executor

As soon as the containers are instantiated according to the provisioning plan, the Service Executor is triggered. This component is responsible for invoking the service instances on the containers. The execution of each service instance is asynchronous, i.e., the Service Executor does not wait for the service instances to finish. If required, this component also transmits input variables to the service instances at the start of the execution.

4.3.6 Monitoring

The Monitoring component collects monitoring information about the state of the service instances (i.e., running, finished, failure) and the resource utilization of the containers. While the service instance itself provides the service instance state, the Monitor component (see Section 4.4.2) provides the resource utilization. A service instance is in the state *running* after it has been successfully invoked. The Monitoring component is informed about the state *finished* by the service instance via the Message Queue (see Section 4.5).

To detect the failure of a service instance, e.g., an exception during the execution of the software, the following methods are applicable: (i) a push model (i.e., the service instance notifies the Monitoring component), (ii) a pull model (i.e., the Monitoring component asks periodically if a failure occurred), (iii) a timeout approach, or (iv) a combination thereof [36]. The current architecture foresees a push model via the Message Queue (Section 4.5) to detect exceptions thrown by the software. This failure detection is reinforced with a timeout approach to detect if a service instance completely stopped running.

If the state of a service instance changes, the Monitoring component informs the Scheduler about the state change. The Scheduler then triggers the update of the provisioning plan and the execution of the updated plan.

The Monitoring component is also responsible for the monitoring of the cloud environment. In this matter, it monitors the connection and availability of the cloud environment, the instantiated VMs, and containers. This monitoring can follow again a push model, a pull model, or a combination thereof [36]. The current architecture foresees a pull model on the base of heartbeat messages. If a failure occurs, e.g., if an instantiated VM is not responding, the Scheduler component is informed about this situation. Thus, the Monitoring component is responsible for the detection of the container and VM failures that were discussed in Section 4.3.2.

Each state change is stored in the Database. Further, if the new state signalizes the end of a service instance execution, the Monitoring component receives the output variables of the service instance, if available.

4.3.7 Container Registry Connector

The Container Registry Connector is responsible for the connection to the external Container Registry (Section 4.5) to get the matching container image for a service of a process activity. The Scheduler triggers this during the creation of the provisioning plan.

As a result, the Container Registry Connector returns the information of the container image that is required to deploy the image on a VM, i.e., the container registry URL, the registry name, and the image name. This information is then transferred to the Cloud Controller, via the Action Executor.

4.3.8 Database

The last component in the eBPMS is the Database. The Database is used to persist execution-related information. This includes the status of the VMs, containers and service instances, the process activity variables (e.g., input and output variables of the service instances), the process instance state, the defined SLAs for each process instance, and the provisioning plan.

4.4 Cloud Environment

The third entity of ViePEP-C as depicted in Fig. 2 is the *Cloud Environment* that is used for the actual execution of the process activities, respectively the corresponding service instances. The cloud providers can be, for instance, Amazon EC2[1], Google Cloud Platform[2] or a private OpenStack-based[3] cloud. As stated before, each service instance is executed in a container. Each of those containers is deployed on a VM that is instantiated and hosted in the cloud. Each VM can deploy several containers for the services. To monitor the resource consumption of the containers and hosting VM, each VM has one dedicated monitoring service running on a separate container. This container and the monitoring service is deployed and started automatically after the VM is running.

4.4.1 Service

The Service component is the actual service implementation that represents a process activity. Such a service can be,

1. https://aws.amazon.com
2. https://cloud.google.com
3. https://www.openstack.org

e.g., a simple software service that performs an addition, a controlling software that is connected to a hardware machine, or a complex and long-running analysis software. Each software service needs to offer a remotely accessible interface, e.g., REST interface, that is used to invoke the service instance and to transfer necessary input variables to the service instance. Each service instance can be invoked several times simultaneously, provided that the hosting container instance has enough resources available.

If the service software recognizes an exception during its execution, which requires a restart of the service, the service informs the Monitoring component via the Message Queue. Thus, each service is responsible for detecting runtime exceptions and for notifying the Monitoring component. If the execution of a service completely stops, the exception may not be detected. Such situations are detected by the Monitoring component with a timeout approach as discussed in Section 4.3.6.

After the service instance finishes its task, it informs the Monitoring component about the state change via the Message Queue (see Section 4.5).

4.4.2 Monitor

The Monitor is responsible for monitoring the services' QoS and the executing environment, i.e., the containers. In this respect, it monitors the status of the used computational resources such as CPU, and RAM. This information is published in regular intervals on the Message Queue.

4.5 Message Queue & Container Registry

The Message Queue and Container Registry are the fourth and fifth entities shown in Fig. 2. While not providing core functionalities of ViePEP-C, these entities offer important helper functionalities necessary for the operation of the platform.

The Message Queue is used to send service instance states, e.g., "execution completed", and monitored computational resource information from the Monitor and Service components to the Monitoring component in the eBPMS. Furthermore, the Message Queue is used to transfer possible output variables, e.g., calculation results, back to the eBPMS.

The Container Registry contains all available container images which are then used to deploy the containers on the VMs. For each container image, the registry holds the information about the contained service, the container registry URL, the registry name, and the image name. This registry is then used by the Container Registry Connector component to get the container image information of a service and by the VMs to pull the container image.

5 The GeCoVM Algorithm

As described in Section 4.3.2, the resource and task scheduling algorithm aims to create a provisioning plan. Such a plan contains the information where a software service should be deployed (i.e., on which container instance and on which VM the container instance should be deployed) and when it should be invoked. The following section presents an algorithm, called GeCoVM (Genetic Container on VM), that is capable of creating such a provisioning plan.

5.1 Concepts of GeCoVM

To create a provisioning plan and to achieve a resource-efficient execution, GeCoVM performs two optimization steps:

In the *first* optimization step, GeCoVM performs scheduling of the process activity executions to achieve a timely overlapping of concordant process activities (i.e., activities that require the same service instance). Overlapping concordant process activities, which invoke the same service instance, are then allocated to the same container instance. This considers that a service instance can be invoked concurrently as part of different processes, as long as the container instance has sufficient computational resources. This results in the need for fewer container instances and, as a consequence, in reduced resource consumption and leasing cost. The output of this optimization step is a list of container instances and their deployment times.

The *second* optimization step gets the output of the first step and assigns to each of those container instances a VM. Each VM can get several container instances assigned (N:1), as long as the VM has enough computational resources.

By separating the optimization into two steps, the search space for finding a cost-efficient deployment for the containers on VMs is narrowed down. Instead of searching for a deployment location for each process activity, the second optimization step only has to search for a deployment location for the container instances. Since a container instance can handle more than one process activity, depending on the output of the first optimization step, the amount of container instances is smaller or equal to the number of process activities. Eventually, a narrowed search space speeds up the optimization and a good result can be found faster.

To further optimize the process activity execution, the deployments of the container instances and VMs are scheduled in a way that it is done during the execution of the preceding process activities. This minimizes the time a process activity execution has to wait for deployment and ensures that the container instances and VMs are already up and running at the time they are needed. However, this is not possible for the first process activities. Since only after the execution of GeCoVM it is known which container instances and VMs have to be deployed. To consider this, GeCoVM schedules the execution of the first process activities in a way that there is enough time to deploy the container instances and VMs.

As described in Section 4.3.2, GeCoVM gets all information about the currently running and not yet running process instances (including information about the process activities), the corresponding SLAs, and the monitoring information from currently running VMs and containers. In line with the related work [3], [37], [38], we consider as SLA the deadline until when the execution of a process instance has to be finished.As defined in Section 4 this SLA is user-defined for each process instance. If the deadline is not fulfilled, penalty cost are charged.

The optimization goal of GeCoVM is to minimize the process execution cost that is composed of the cloud resource *and* penalty cost.

Since the problem of task scheduling is NP-hard, GeCoVM is realized as a genetic algorithm [7]. As shown in different areas (e.g., [39], [40]) genetic algorithms have great

added after submission: Also apache airflow sees the dead-

potential for task scheduling in respect to optimization-runtime and quality of the result, i.e., convergence to the optimal solution.

GeCoVM is an extension of a scheduling algorithm presented in our former work [41]. So far, we introduced a genetic algorithm that performs scheduling of process activities in a way that a timely overlapping of concordant process activities is achieved, i.e., the first optimization step. In comparison to GeCoVM, the algorithm from [41] only considers the enactment of the process activities on containers and does not optimize the deployment of the container instances on the VMs, i.e., the second optimization step. In the following subsections, we will focus on the second step, but will give an overview about the first step wherever necessary for the understanding of GeCoVM.

5.1.1 Stopping Criterion

Since the execution duration of the algorithm affects the actual schedule of the activities, i.e., the algorithm needs to know when the first process activities can start, we use a time-based stopping criterion. This time-based stopping criterion stops the execution of the scheduling algorithm after a predefined time and returns the best result found until then. This stopping criterion is used for both optimization steps.

5.1.2 Chromosome Representation

As described before, GeCoVM uses two consecutive optimization steps so that the output of the first optimization step is the input of the second optimization step. Each optimization step uses a different chromosome representation.

For the first optimization step, the chromosome is composed of all running and not yet running process activities, which are the genes of the chromosome. Each gene holds the scheduled start time, i.e., the time when the execution of a process activity should start. Fig. 4a represents an example chromosome with a detailed representation of the process shown in Fig 4b.

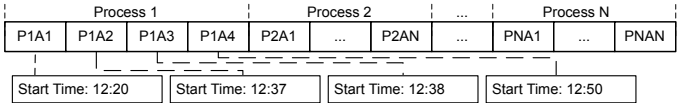
For the second optimization step, the chromosome is composed of all containers that are required to execute the process activities at the time that was determined by the first optimization step. Each gene holds the VM where the container instances should be deployed. Fig. 4c depicts an example chromosome for the second optimization step.

Each of those chromosomes represents a possible solution. In the following, the genetic operations (i.e., selection, crossover, and mutation) will be used on these chromosomes to alter them and to find an optimal solution.

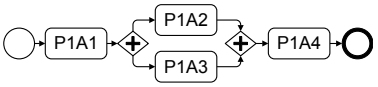
5.1.3 Initial Population

The initial population, i.e., the first generation of chromosomes used as input for a genetic algorithm [42], is composed by randomly assigned process activity start times for the first optimization step, and randomly assigned VMs for the second optimization step. A random selection of the start time and VMs, creates a high population diversity and helps to avoid premature convergence [42], [43].

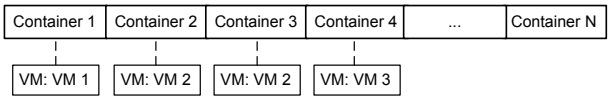
For the first optimization step, the initial population creation algorithm assigns to each process activity a random start time in a way that the order of the process activities, defined by the process model, is not violated. This algorithm



(a) Example Chromosome for the First Optimization Step with a Detailed Representation of Process 1



(b) Process 1 in BPMN



(c) Example Chromosome for the Second Optimization Step

Fig. 4. Example Chromosome Representation

Algorithm 1 Initial Population

Require: *containers*, *availableVMs*

```
1:  function ASSIGNVMS
2:      chromosome ← NULL
3:      for all c ∈ containers do
4:          if c.vm is empty then
5:              vm ← getVM(c, availableVMs)
6:              availableVMs ← availableVMs ∪ {vm}
7:              c.vm ← vm
8:          end if
9:          chromosome ← chromosome ∪ {c}
10:     end for
11:     return chromosome
12: end function
```

creates chromosomes of the kind represented by Fig. 4a. The algorithm uses the deadlines as *a priori* knowledge [44] and limits the random movement of the activities to not violate the deadlines. To achieve this, the algorithm assigns to each process activity a time frame in which the randomly assigned start time can be.

As discussed in Section 4.3.2, the Scheduler component, in this case GeCoVM, is executed several times (e.g., if a new process instance is requested or a failure occurs) and each call considers all currently running and requested process instances for the optimization. This can lead to the situation where the execution of a process activity already started or is already done when GeCoVM performs an optimization. This situation happens when a process instance was already the subject of an earlier optimization. If the execution of a process activity is already done or will be over before the optimization end time is reached, the process activity is ignored. If the process activity execution is still running when the optimization end time is reached, the start time of this process activity is used as *a priori* knowledge and not altered. A full description of the initial population creation algorithm for the first optimization step can be found in [41].

Algorithm 1 presents the initial population creation algorithm for the second optimization step. This algorithm assigns a random VM to each gene of a chromosome, as presented in Fig. 4c.

As input, the algorithm gets a list of all container in-

stances (*containers*) required to execute the process activities at the times defined by the first optimization step, and a list of already available VMs (*availableVMs*). The list of available VMs, contains VMs that will be available when the execution of GeCoVM is done, i.e., the stopping criterion is fulfilled. These VMs are for example VMs that were assigned to some process activities in a previous execution of GeCoVM.

Beginning with line 3, Algorithm 1 iterates over all container in the *containers* list. As mentioned before, GeCoVM is executed several times and each execution considers all running and requested process instances. Thus, previous executions of GeCoVM may already assigned a VM to a container *c*, this is checked by line 4. In such a situation, the algorithm does not change the deployment.

If the container *c* does not have a VM, a new VM is selected by the method *getVM* (line 5). The method *getVM* selects a VM that has enough computational resources left to deploy the container instance. This VM is taken from the *availableVMs* list, or a new VM is created. The decision, whether a VM is taken from the *availableVMs* list or a new VM is created, is done randomly to achieve a high population diversity [42], [43]. If the *availableVMs* list is empty or does not contain a VM that has enough space for the container instance, a new VM is created. The returned VM is stored in *vm*.

The VM, which is stored in *vm*, is then added to the *availableVMs* list (line 6) and to the container instance *c* (line 7). Eventually, the container *c* is added to the *chromosome* and the *chromosome* is returned in line 11.

Algorithm 1 is executed several times, depending on defined population size, to compile the initial population. The population size is a configureable value and defined upfront, e.g., by the eBPMS operator.

5.1.4 Fitness Function

The fitness function calculates for each chromosome the fitness score. The chromosome with the smallest fitness score represents the solution with the lowest VM leasing cost and penalty cost.

As defined before, the first optimization step aims at overlapping concordant process activities so that several process activities can use the same software service. Therefore, the fitness score of the first optimization step is composed of the amount of overlapping concordant process activities, how good an overlapping was achieved, and the penalty cost.

To define how good an overlapping of concordant process activities was achieved, the first optimization step calculates the computational resources used by a container instance to execute the process activities. A good overlapping of concordant activities yields in a minimized duration of a container instance being deployed and, thus, in a minimized computational resource usage. This overlapping of process activities is considered by (1).

$$\sum_{c \in C} (c_{cpu} * f_{cpu} + c_{ram} * f_{ram}) * c_{duration} * f_{container} \quad (1)$$

In (1), C is the list of container instances required to execute all process activities. A container instance is defined as $c \in C = \{c_1, c_2, ...\}$ and $c = (cpu, ram, duration)$ defines the computational size of the container instance (i.e., amount of CPU and RAM) and how long the container instance is deployed. The factors f_{cpu} and f_{ram} define how much the amount of CPU cores, respectively RAM, should be considered in the fitness score. The parameter $f_{container}$ defines the weight of the cost in the final fitness score of the first optimization step. f_{cpu}, f_{ram}, and $f_{container}$ are configureable values.

The penalty cost is composed of all the cost that arise due to missing deadlines, i.e., the time between the termination of the last process activity of a process and the deadline.

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

In (2), W is the set of all process instances of a chromosome and $w \in W = \{w_1, w_2, ...\}$ defines one process instance. A process is defined by $w = (end, deadline)$, where *end* defines the time when the last process activity terminates, and *deadline* is the defined process instance deadline. The factor $f_{penalty}$ defines the weight of the penalty cost in the final fitness score. If a process instance w violates the deadline or not, is considered by $x(w) \in \{0, 1\}$, i.e., $x(w) = 1$ the deadline is violated, $x(w) = 0$ the deadline is not violated.

The sum of (1) and (2) results in the final fitness score of the first optimization step.

The second optimization step considers the deployment of the container instances to the VMs. A good solution to this optimization step is when the leasing cost of the VMs are minimized. The VM leasing cost is the combination of all cost that arise due to the leasing of the required VMs used to deploy the container instances. Since the second optimization step does not change the process activity start times, the penalty cost is not considered.

$$\sum_{v \in V} (v_{cpu} * p_{cpu} + v_{ram} * p_{ram}) * v_{duration} * f_{VMleasing} \quad (3)$$

Eq. (3) is similar to (1), however, this time the VM resource consumptions, instead of the container resource consumptions, are considered. In (3), V is the list of VM deployments required to deploy all containers that are used for the enactment of the process activities. A VM is defined as $v \in V = \{v_1, v_2, ...\}$ and $v = (cpu, ram, duration)$ defines the computational size of the VM (i.e., amount of CPU and RAM) and how long the VM is deployed. The CPU and RAM prices of a VM are defined by p_{cpu}, i.e., the price for one CPU core, and p_{ram}, i.e., the prize for one GB of RAM. The parameter $f_{VMleasing}$ is a configurable factor that defines the weight of the leasing cost in the final fitness score.

Since the second optimization step is only concerned with the VM leasing cost, the fitness score is the result of (3).

5.1.5 Mutation

As explained in Section 3.3, the mutation operation varies a randomly selected gene of a chromosome. For the first optimization step, the mutation changes the start time of a process activity, and the second optimization step mutates the VM that should be used to deploy a container instance.

To ensure that the control flow of a process instance is not violated by mutating the start time, the mutation operation

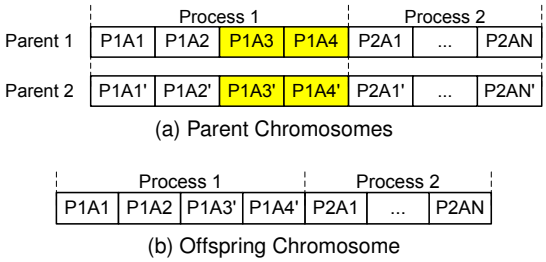


Fig. 5. Two-Point Crossover Operation

of the first optimization step defines boundaries in which the start time can be mutated, similar to the creation of the initial population. For instance, if the process activity is between two other process activities within a process model, the boundaries are the end time of the preceding process activity and the start time of the upcoming process activity minus the current process activity execution time. In between these boundaries, the mutation operation randomly selects a start time and assigns this start time to the gene. A detailed description of this mutation operation can be found in [41].

In case of the second optimization step, the already mentioned method *getVM* from Algorithm 1 is used to mutate the assigned VMs. The method gets the container of the current gene (*c*) and the list of available VMs (*availableVMs*) as input parameters and returns a random VM from the *availableVMs* list or a new VM.

In case of the second optimization step, the assignment of a new VM to a gene can lead to a situation where there is not enough time between the optimization end time (i.e., the end of the execution of GeCoVM) and the scheduled start time of the process activity to deploy the VM and container instance. To consider this, GeCoVM checks if this is the case and if so the mutation is reversed, and another random gene is selected.

5.1.6 Crossover

The crossover operation creates a new chromosome by splitting two chromosomes and combining them to a new chromosome. The two selected chromosomes are called parent chromosomes, and the new chromosome is called offspring (see Section 3.3).

For the first optimization step, GeCoVM uses a two-point crossover [31]. For this, the start gene of the crossover is selected randomly, i.e., a random process activity in a random process instance, and the end gene is the end of the process to which the process activity belongs. We decided to use a two-point crossover operation to consider one process instance at a time. Fig. 5 depicts such a two-point crossover operation. Fig. 5a depicts the two parent chromosomes with the two genes that are selected for the crossover highlighted in yellow. In this example, the gene P1A3 is the randomly selected gene, and P1A4 is the end of the process instance of chromosome Parent 1 (for Parent 2, chromosomes P1A3' and P1A4' are the representative process activities). Fig. 5b shows the offspring chromosome where the selected genes are from Parent 2 (i.e., P1A3' and P1A4') and the remaining genes are from Parent 1.

For the second optimization step, the crossover operation is a single-point crossover where only the start gene is randomly selected, and the end point is the end of the chromosome. Since the chromosome of the second optimization step is independent of the process instance, but a combination of containers, we decided to use a single-point crossover instead of a two-point crossover as for the first optimization step.

Again in the first optimization step, the crossover operation changes the start times of the process activities. In the second optimization step, the crossover operation changes the VM that should be used to deploy a container instance.

In both optimization steps, a crossover can lead to an incorrect provisioning plan. In case of the first optimization step, a crossover operation can lead to a violation of the control flow, defined by the process model. In the second optimization step, a situation where a VM does not have enough computational resources to deploy a container instance may occur. To consider this, GeCoVM checks if one of those violations occurs after the crossover and selects another crossover point, i.e., another gene, if necessary. If the offspring chromosome does not contain one of those violations, the offspring chromosome is returned.

6 Implementation

We implemented ViePEP-C in Java. The implementation uses the Spring Framework (vers. 2.1.1). As container technology, ViePEP-C applies Docker and as container registry Docker Hub[4] is used. As cloud provider, the current implementation provides connections to Google Cloud and Amazon AWS EC2. The default VM type in the current implementation is CoreOS 1800.5.0. However, the VM type is configurable. The execution of the services is monitored by a timeout approach in combination with a push model, if the service software detects an exception. The timeout is the multiple of the expected execution duration of a service (the default value is two times the expected execution duration). As message queue, ViePEP-C uses RabbitMQ 3.6.11[5], i.e., a JMS-based message queue. The GeCoVM implementation uses the Watchmaker framework[6] as evolution engine.

The implementation offers a rich set of configuration possibilities, e.g., to define the container images for the services, the cloud user information, or the different VM sizes. As database, MySQL 14.14[7] is used.

For interacting with the cloud, e.g., to start and stop a VM, the implementation uses the Google Cloud Platform Java library (vers. 0.21.1)[8] respectively the Amazon AWS EC2 library (vers. 1.11.121)[9]. For the interaction with the Docker environment, i.e., to deploy, start, stop and monitor the resource consumptions of the containers, the implementation uses the Spotify Container library (vers. 8.11.7)[10].

Since Google Cloud and AWS EC2 only allow the leasing of one CPU or an even amount of CPUs, e.g., 2, 4, 6, we split

4. https://hub.docker.com
5. http://www.rabbitmq.com
6. https://watchmaker.uncommons.org/
7. https://www.mysql.com/de/
8. https://cloud.google.com/java/docs/reference/
9. https://aws.amazon.com/sdk-for-java/
10. https://github.com/spotify/docker-client

up container instances that would require an odd amount of CPUs. For instance, if a container instance needs five CPUs to execute four process activities, the container instance is split up into two container instances with two process activities each. By doing this, we can further minimize the size of the leased VMs, since without this the next fitting VM size would be 6 CPUs. The size of RAM can be defined in 0.25 GB steps, hence, a segmentation is not needed.

The Monitor component, which observes the resource consumption of the individual VMs, is also written in Java, uses the Spring Framework, and the Spotify Container library to gather the required information.

The source code of ViePEP-C and the Monitor component, are available at [11] and [12].

7 Evaluation

In the following, we will evaluate ViePEP-C and discuss the benefits of a container-based process execution in comparison to a state-of-the-art VM-based approach. For the evaluation, we use ViePEP-C as discussed in Section 6 and employ the Google Cloud Platform as cloud environment for the eBPMS and the VMs for the containers. The source code of the Evaluation Client is available at Github[13].

7.1 Evaluation Setting

For the evaluation, we apply a setting that has already been used in our former work on elastic process execution [7], [41], and has been adapted for the purposes of the container-based elastic process execution discussed here.

7.1.1 Test Collection

For the evaluation of ViePEP-C, we use a subset of the SAP reference process models [45], [46], which is a common validation sample applied in many evaluations in the field of BPM [47]. For the evaluation, we choose ten representative process models with different degrees of complexity. Table 1a shows the basic characteristics of the chosen process models. The table presents the amount of process activities, AND-blocks, XOR-blocks, and loops. Each AND- and XOR-block includes a split and join gateway. Fig. 6 shows process No. 3 and process No. 9 as examples in BPMN.

For the evaluation, we use eight different software service types with different resource requirements and execution durations. Table 1b provides an overview of the required CPU load (in percentage), required memory (in MB), and total makespan (in seconds) of the different services. The required CPU load defines how much of a CPU is required by the service when one core of the CPU offers 100% and each further core adds another 100% (e.g., a dual-core CPU offers 200% overall). We assume that each service is fully parallelizable among the available CPUs. This means that a service with 100% load on a single-core CPU, has a 50% load on a dual-core (i.e., 50% on each core) with the same execution time, which results in 50% of the dual-core CPU not being used or being available to other services.

11. https://github.com/piwa/ViePEP-C
12. https://github.com/piwa/ViePEP-C-Backendmonitor
13. https://github.com/piwa/ViePEP-C-Testclient

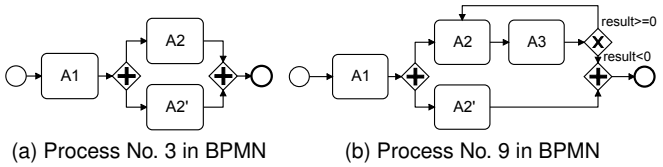
TABLE 1
Evaluation Process Models and Service Types

(a) Evaluation Process Models Characteristics

Name	|Activities|	$\|XOR\|$	$\|AND\|$	$\|loops\|$
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	0
9	4	1	1	1
10	20	0	4	0

(b) Evaluation Services

Service No.	CPU Load in % (μ_{cpu})	Memory Load in MB (μ_{ram})	Service Makespan in sec. ($\mu_{makespan}$)
1	15	270	40
2	20	450	320
3	25	720	480
4	40	720	80
5	55	960	400
6	65	1150	120
7	80	1150	160
8	135	1440	80



(a) Process No. 3 in BPMN (b) Process No. 9 in BPMN

Fig. 6. Evaluation Processes in BPMN

In addition, we assume that the actual CPU load, memory consumption, and total makespan of a service vary to some extend for each service invocation. For this we assume a normal distribution with $\sigma_1 = \mu_{cpu}/10$, $\sigma_2 = \mu_{ram}/10$, and $\sigma_3 = \mu_{makespan}/10$, each with a lower bound of 95%, and an upper bound of 105%.

If a service is invoked several times in parallel, the service utilizes more computational resources. To consider this aspect, we utilize insights from previous research [41] that has been found to best calibrate with a parameter of $2/3$. Accordingly, we add for each invocation $2/3$ of μ_{cpu} and μ_{ram} to the service resource consumption defined in Table 1b. For instance, two parallel invocations of service 1 will lead to $\mu_{cpu} = 15 + 15 * 2/3 = 25$.

Despite the fact that several SAP reference models contain human-provided services, we use for all process activities only software-based stateless services.

We implemented a test service to evaluate our approach[14]. This service is written in Java and uses the Spring Framework. To simulate the different resource demands and execution durations, the service uses the *fakeload* load generator [48]. This load generator simulates a configurable CPU and memory load for a particular time span. The test service offers a REST interface to start the execution. This execution takes as a parameter the resource consumption re-

14. https://github.com/piwa/ViePEP-C-Backendservice

quirements, i.e., CPU and RAM, and the service makespan.

7.1.2 Applied SLAs

Each process execution request also includes a deadline SLA, i.e., a point in time until when the execution of a process instance has to be finished. For the evaluation of ViePEP-C, we apply two different SLA levels. The first one is following a *strict* scenario, i.e., the deadline is 1.5 times of the makespan of the whole process. The process makespan is calculated upfront with the knowledge of the process model structure and the makespan of each service, taken from historical runs of the services. The second SLA level is *lenient*, i.e., the deadline is 2.5 times of the makespan of the whole process. Therefore, the lenient scenario allows a longer execution of a process instance without violating the SLA. Those two scenarios were chosen to tolerate the start-up time for the VMs, the containers and services.

7.1.3 Process Request Arrival Patterns

We apply two different process request arrival patterns. The first one requests a *constant* amount of process executions in a regular interval, i.e., this pattern requests the execution of five different process models every 120 seconds. The selection of the processes is done sequentially in a round-robin fashion according to Table 1a, i.e., the first iteration requests process model No. 1 to No. 5, the second round No. 6 to No. 10, the third one No. 1 to No. 5 etc. This is repeated 20 times.

The second process request arrival pattern follows a *pyramid*-like pattern described by (4). In the equation, a represents the amount of process execution requests at a specific point in time n. Analogue to before, the concrete process is chosen in a round-robin fashion according to Table 1a. In the end, a total amount of 100 instances are requested. The request time interval is set to 120 seconds.

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

7.1.4 Baseline

We compare ViePEP-C against a baseline which follows a state-of-the-art approach. The baseline provides an extended version of the algorithm *AllParExceed* discussed by Frincu et al. [49]. The algorithm assigns to each parallel process activity a new VM, or an existing one if available. A VM can thereby execute one service instance at a time to guarantee the necessary isolation of the service instances. However, a service can be invoked several times by process activities that require the same software service.

Fig. 7 presents a provisioning plan for the process shown in Fig. 6a, applying the AllParExceed approach.

In comparison to the algorithm presented in [49] we neglect the minimum leasing duration of a VM, called Billing Time Unit (BTU). This is done since most cloud providers do not consider the BTU time anymore or it is set to a small duration, e.g., 1 minute.

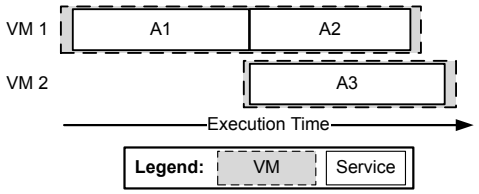


Fig. 7. Baseline Provisioning Plan Example

7.1.5 GeCoVM Parameter Setting

Based on preliminary experiments, we set the parameters in our evaluations as follows: A population size of 700, with 35 elite chromosomes per population, an optimization duration of 40 seconds for the first optimization step, and 40 seconds for the second optimization step.

For the fitness function, we use the settings: $p_{cpu} = 32$ and $p_{ram} = 4$ (which are rounded real-world cost of the Google Cloud Platform[15]), and $f_{cpu} = 32$ and $f_{ram} = 4$. The remaining parameters for the fitness functions (i.e., $f_{container}$, $f_{VMleasing}$, and $f_{penalty}$) will be set differently for different evaluation scenarios, as outlined below.

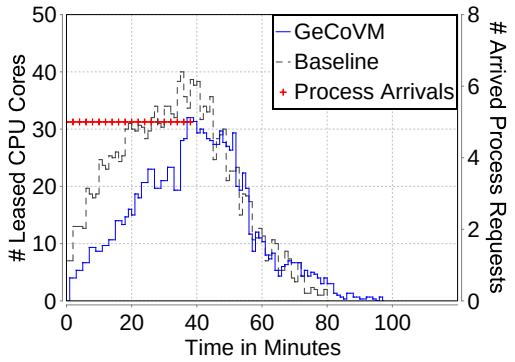
7.2 Metrics

To evaluate our approach, we use different metrics. To start with, the total *execution duration* of all process instances is measured, i.e., the sum of the elapsed time from a process execution request until the final process activity of the process instance terminates. The second metric is the *SLA adherence* in percentage, i.e., the amount of finished process instances without violating the SLAs in percentage. Third, the *leasing cost in core-minutes* tells us how many cores are leased for how many minutes, e.g., leasing two cores for two minutes results in four core-minutes. This is equivalent to the cost model of most cloud providers since they charge for the amount of used computational resources (e.g., amount of cores and duration of usage) following a linear cost model. Fourth, the *penalty cost* is the cost that is charged due to a SLA violation, i.e., in our case not keeping a deadline. For the penalty cost, we apply a linear cost model [50]: The model assigns for 10% of time units of delay, one unit of penalty cost. The penalty cost is measured in money units (MU). For all metrics, we calculate the mean value and the standard deviation σ.

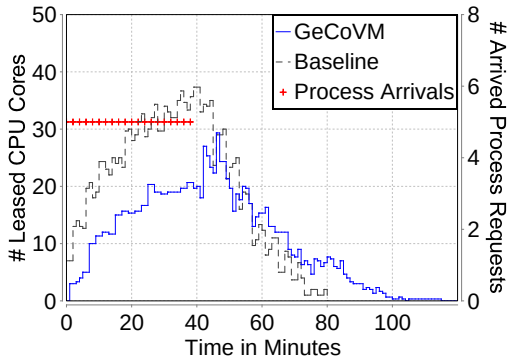
7.3 Evaluation Process

In the evaluation, we evaluate the efficiency of our container-based process execution approach in comparison to the state-of-the-art VM-based approach presented in Section 7.1.4. To fully evaluate ViePEP-C in combination with the task and scheduling algorithm GeCoVM, we perform and evaluate the following steps: First, we conduct a general evaluation where we analyze the influence of the two arrival patterns and SLA levels. Second, we analyze the influence of the fitness function parameters $f_{container}$, $f_{VMleasing}$, and $f_{penalty}$ by adapting them. Third, we analyze how ViePEP-C detects a failure situation by terminating a VM during a process activity execution.

15. https://cloud.google.com/compute/pricing

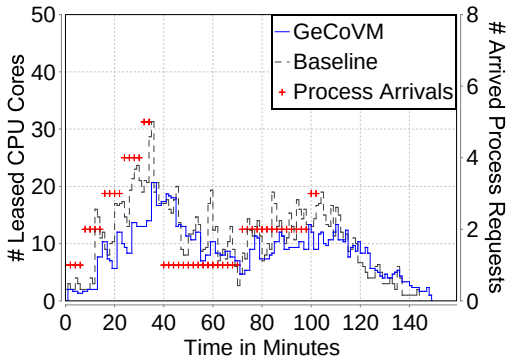


(a) Constant Arrival Pattern - Strict SLA

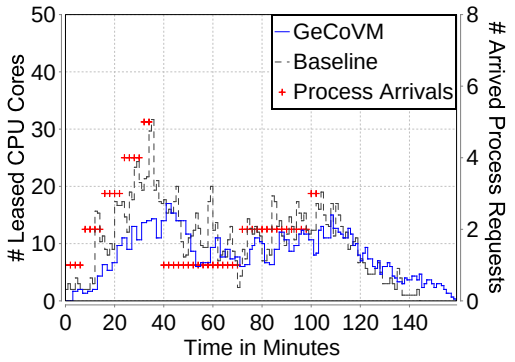


(b) Constant Arrival Pattern - Lenient SLA

Fig. 8. Constant Arrival Pattern



(a) Pyramid Arrival Pattern - Strict SLA



(b) Pyramid Arrival Pattern - Lenient SLA

Fig. 9. Pyramid Arrival Pattern

To minimize the effect of external influences, e.g., concurrently running applications, we execute each evaluation three times at different times of a day over two weeks.

7.4 Results and Discussion

In the following, we will present and discuss the results of the evaluation.

7.4.1 General Evaluation

In this scenario, we evaluate the behavior of ViePEP-C in combination with GeCoVM by using both arrival patterns, and both SLA level. For this evaluation scenario we set the fitness function parameters to the following values: $f_{container} = f_{VMleasing} = 10$, $f_{penalty} = 0.001$. This values yielded good results in several pre-evaluation executions.

Table 2 presents the resulting mean values and the standard deviation of the evaluation. Fig. 8 shows the results of the constant arrival pattern for the strict and lenient SLA level, and Fig. 9 the same for the pyramid arrival pattern.

In the following, we will first discuss the results of the constant arrival pattern for both SLA levels and then the results of the pyramid arrival pattern for both SLA levels.

As can be seen in Table 2, the SLA adherence of the baseline with the lenient SLA level is the highest (100%). Since the baseline executes one process activity directly after a preceding activity, basically no delay happens, which leads to 100% SLA adherence. For the baseline with the strict SLA level, a slightly lower result (98.67%) is achieved. This slightly lower result is due to the need for deploying the VMs, which may lead to a delay in the process activity execution. For the evaluation with active GeCoVM, an SLA adherence of 90.67% (for the lenient SLA level) and 89.67% (for the strict SLA level) is achieved. This lower SLA adherence, in comparison to the baseline, is because GeCoVM postpones process activity executions if an overlapping of process activities can be achieved. The lower SLA adherence also increases the penalty cost in case of active GeCoVM.

While the SLA adherence is lower in case of active GeCoVM, regarding the leasing cost GeCoVM achieves a cost saving in comparison to the baseline. For the strict SLA level, GeCoVM resultes in 20.20% lower leasing cost than the baseline, i.e., 1173.0 core-minutes instead of 1470.33 core-minutes for the baseline. For the lenient SLA level the leasing cost saving, achieved by GeCoVM in comparison to the baseline, is 21.60%. This additional cost saving is because GeCoVM is able to achieve more overlappings of process activities since the margin of postponing them is higher for the lenient SLA level than for the strict SLA level.

This can also be observed in Fig. 8. In Fig. 8a (strict SLA level) more CPUs at a time are leased than in Fig. 8b (lenient SLA level), i.e., partly over 30 CPUs for the strict SLA and always under 30 CPUs for the lenient SLA. However, in Fig. 8b the time the VMs are leased is long, i.e., over 110 minutes for the lenient SLA level and under 100 minutes for the strict SLA level.

In Fig. 8, it can also be observed that GeCoVM leases less CPU cores than the baseline. It can be further observed that after 50 minutes the number of leased CPU cores in case of GeCoVM are sometimes higher than for the baseline. This is the result of postponing the process activity executions in case of GeCoVM, which is not the case for the baseline.

For the pyramid arrival pattern, similar results are

TABLE 2
General Evaluation Results (Standard Deviations in Parenthesis)

	Constant Arrival Pattern				Pyramid Arrival Pattern			
	Strict		Lenient		Strict		Lenient	
	GeCoVM	Baseline	GeCoVM	Baseline	GeCoVM	Baseline	GeCoVM	Baseline
Execution Duration in Minutes	92.0 (8.66)	80.33 (4.67)	114.0 (10.54)	80.67 (0.57)	148.33 (0.58)	144.0 (0.53)	174.0 (17.35)	144.0 (0.0)
SLA Adherence (%)	89.67 (6.81)	98.67 (0.58)	90.67 (1.53)	100.00 (0.00)	91.33 (1.15)	99.67 (0.57)	96.67 (2.31)	100.0 (0.00)
Penalty Cost in MU	32.67 (20.11)	2.67 (2.89)	20.33 (5.51)	0.00 (0.00)	22.33 (4.73)	1.67 (2.89)	6.33 (4.04)	0.0 (0.0)
Leasing Cost in Core-Minutes	1173.0 (50.86)	1470.33 (11.68)	1154.33 (73.21)	1472.33 (8.08)	1203.33 (155.28)	1394.0 (2.65)	1198.67 (28.10)	1395.33 (10.11)

achieved. With the pyramid arrival pattern, the baseline with the lenient SLA level achieves an SLA adherence of 100%, and with the strict SLA level 99.67%. However, in comparison to the constant arrival pattern, GeCoVM achieves for both SLA levels better results regarding the SLA adherence, i.e., 91.33% for the strict SLA level and 96.67% for the lenient SLA level. This also results in lower penalty cost, in comparison to the constant arrival pattern.

Regarding the leasing cost, the cost savings in comparison to the baseline is lower in case of the pyramid arrival pattern than for the constant arrival pattern. With the pyramid arrival pattern, GeCoVM achieves a cost saving of 13.68% (strict SLA level) and 14.09% (lenient SLA level) in comparison to the baseline. This lower cost saving, in contrast to the cost saving with the constant arrival pattern, is due to the fact that the constant arrival pattern requests more process executions in parallel. This increases the chance that an overlapping of concordant process activities can be achieved, i.e., the possibilities of re-using a container instance for several process activities.

Concluding, it can be seen that by accepting a slightly lower SLA adherence (thus, slightly higher penalty cost) ViePEP-C is able to achieve much lower leasing cost in comparison to the baseline. This trade-off between cost and SLA adherence needs to be taken into account by the user, i.e., the process owner, and could be used as a foundation to select between different resource allocation and task scheduling strategies.

7.4.2 Parameter Influence Evaluation

In this evaluation, we analyze the influence of the GeCoVM parameters $f_{VMleasing}$,$f_{container}$, and $f_{penalty}$ (Section 5.1.4).

For this, we use the constant process execution request pattern with the strict SLA level and analyze the behavior by using three different parameter settings: (1) $f_{container} = f_{VMleasing} = 10$, $f_{penalty} = 0.001$; (2) $f_{container} = f_{VMleasing} = 1$, $f_{penalty} = 1$; (3) $f_{container} = f_{VMleasing} = 0.001$, $f_{penalty} = 10$. The first setting puts more weight on the optimization regarding the leasing cost than the penalty cost, the second setting weights the leasing cost and penalty cost the same, and the third setting puts more weight on the optimization of the penalty cost. The results of the evaluation are presented in Fig. 10 and Table 3.

As can be seen in Table 3 and Fig. 10 the first parameter setting ($f_{container} = f_{VMleasing} = 10$, $f_{penalty} = 0.001$) achieves the best result in respect of leasing cost and the third parameter setting ($f_{container} = f_{VMleasing} = 0.001$, $f_{penalty} = 10$)

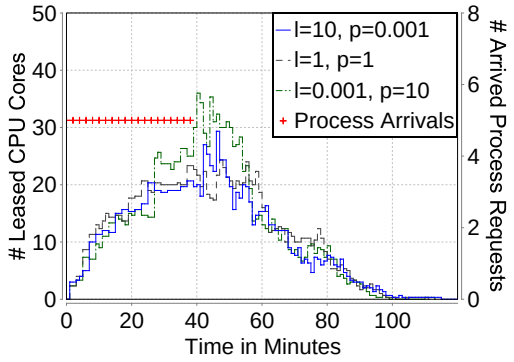


Fig. 10. Result of the GeCoVM Parameter Evaluation (legend: $l = f_{VMleasing} = f_{container}$; $p = f_{penalty}$

is the most expensive one. This is since the first parameter setting emphasis the optimization regarding leasing cost the most ($f_{container} = f_{VMleasing} = 10$) and the third parameter setting the least ($f_{container} = f_{VMleasing} = 0.001$).

For the penalty cost, it is the opposite. Here the third parameter setting achieves the best result regarding the SLA adherence (99.33%) and the first parameter setting the worst (90.67%). Again, this is due to the weight of the importance. Since the first parameter setting sets $f_{penalty} = 0.001$ and the third parameter sets $f_{penalty} = 10$, the optimization of the penalty is more important with the third parameter setting than with the first parameter setting.

The second parameter setting ($f_{container} = f_{VMleasing} = 1$, $f_{penalty} = 1$) distributes the weight equally between the optimization of the leasing cost and penalty cost (respectively SLA adherence). As can be seen in Table 3, the results for the leasing cost and penalty cost are between the first and third parameter setting. Regarding the leasing cost, the second parameter setting (1242.0 core-minutes) is approximately in the middle between the leasing cost of the first setting (1154.33 core-minutes) and the third setting (1314.33 core-minutes). For the SLA adherence and penalty cost the results are more in the direction of the third parameter setting.

In comparison to the baseline results from Section 7.4.1, for the constant arrival pattern with strict SLA, we can further see an improvement with the second and third parameter setting. Both settings achieve better results regarding the SLA adherence and leasing cost than the baseline from Section 7.4.1, e.g., the second parameter setting achieves: 99% SLA adherence (baseline: 98.67%) and 1242.0 core-minutes (baseline: 1470.33 core-minutes).

TABLE 3
GeCoVM Parameter Influence Evaluation Results (Standard Deviation in Parenthesis). For all settings we set $f_{container} = f_{VMleasing}$

	$f_{VMleasing} = 10$ $f_{penalty} = 0.001$	$f_{VMleasing} = 1$ $f_{penalty} = 1$	$f_{VMleasing} = 0.001$ $f_{penalty} = 10$
Execution Duration in Minutes	114.0 (10.54)	103.33 (6.81)	103.33 (8.39)
SLA Adherence (%)	90.67 (1.53)	99.0 (0.0)	99.33 (0.58)
Penalty Cost in MU	20.33 (5.51)	1.0 (0.0)	0.67 (0.58)
Leasing Cost in Core-Minutes	1154.33 (73.21)	1242.0 (82.94)	1314.33 (56.32)

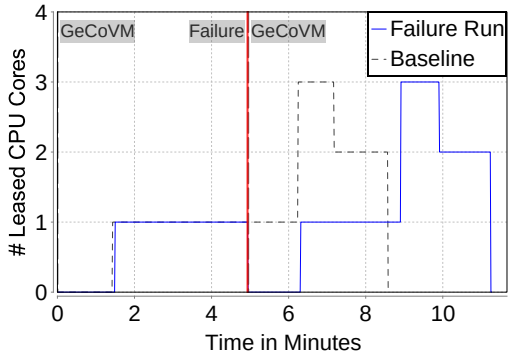


Fig. 11. Result of the Failure Detection Evaluation

7.4.3 Failure Detection Evaluation

In this evaluation, we analyze the behavior of ViePEP-C in a failure situation, e.g., the loss of a connection to a VM. As discussed in Section 4.3.2, this should trigger a re-scheduling of the provisioning plan.

For this evaluation, we request the execution of process 1 from Table 1a. This process is a sequential execution of the software services 6, 7, and 8 from Table 1b. During the execution of the second process activity, we externally (i.e., by a separate tool and not via ViePEP-C) terminate the executing VM. ViePEP-C should detect this as a VM connection failure and should trigger a re-scheduling of the provisioning plan. For the process execution, we use the strict SLA level. For the baseline, we request the execution of the same process, also with active GeCoVM. However, for the baseline, the VM is not terminated.

Fig. 11 presents the result of this evaluation. In Fig. 11 two markers, labeled *GeCoVM*, signalise when a scheduling, i.e., an execution of GeCoVM, took place. The third marker, labeled *Failure*, signalises when the VM was terminated.

As can be seen in Fig. 11, at the beginning (minute 0) the first execution of GeCoVM and the creation of the provisioning plan takes place (signalized by the first *GeCoVM* marker). Since we set the time-based stopping criterion to 80 seconds, the first VM is deployed at around 1:30 for the failure run evaluation. In the beginning, only one CPU is leased, since the first two process activities use service 6 and 7 that only need one CPU.

At around minute 5, we terminate the VM (signalized by the *Failure* marker). As can be seen in Fig. 11, this is detected by ViePEP-C and a re-scheduling takes place (signalized by the second *GeCoVM* marker). After another 80 seconds, GeCoVM updates the provisioning plan, and a new VM is deployed in case of the failure run evaluation.

At minute 9, a two-core VM is deployed during the failure run evaluation, which is used for the execution of the third process activity. At this time, the execution of the second process activity is not over yet, which leads to a leasing of three cores for a short period. After the execution of the third process activity is over, all VMs are released and the execution of the process instance is finished.

In case of the baseline, the same behavior as during the failure run can be observed at the beginning. The first VM is leased at a similar time as before. However, since the baseline does not suffer from a VM loss, it can use the same VM for the execution of the first and second process activity. Again, for the third process activity, a two core VM is leased while the one-core VM is still leased. This results in a short time where three CPUs are leased.

The final leasing cost of the failure run is 10 core-minutes and 9 core-minutes for the baseline. Regarding the penalty cost, the failure run is delayed 86 seconds which leads to a penalty cost of 3 MU. The baseline run is not delayed.

7.4.4 Concluding Discussion

Concluding, it can be said that ViePEP-C together with the container-based process execution approach and GeCoVM always achieves a better result regarding the leasing cost (Section 7.4.1 and 7.4.2). For instance, GeCoVM achieves 21.60% cost saving in comparison to the baseline for the constant arrival pattern and the lenient SLA level. Regarding the SLA adherence, the same evaluation results in 90.67% adherence.

Most of the cost-savings result from the fact that by using containers a higher VM utilization is achieved by offering an isolated environment for each service. This isolated environment allows the parallel execution of several services without risking conflicts, e.g., conflicting libraries.

With respect of failure detection and recovery, we have shown in Section 7.4.3 that ViePEP-C detects failures, e.g., a VM is not reachable anymore, and triggers the task and resource scheduling process. This scheduling process, performed by GeCoVM, updates the provisioning plan so that the execution of all process instances can be continued.

8 Related Work

In the following, we discuss selected contributions to the research field of elastic process execution on cloud resources.

In our former work [7], [33], [34], we presented an eBPMS called ViePEP. ViePEP uses VMs for the execution of process activities on cloud resources. ViePEP optimizes and schedules the execution on the available cloud resources in a cost-efficient way while considering SLAs, e.g., a deadline for the process execution. While ViePEP is the most comparable eBPMS to the work presented in this paper, the use of containers as execution environment requires a completely

new design and implementation of the eBPMS architecture and process execution handling, as shown in Figure 2.

Juhnke et al. [16] provide in their work an extended BPEL engine that leases and uses cloud-based computational resources in the form of VMs in an on-demand fashion to execute process activities. The presented scheduling algorithm considers the cost for the VMs and also the data transfer duration. Nevertheless, the approach does not regard SLAs nor container-based service instantiation.

Further publications regarding scheduling and execution of elastic processes on cloud resources are presented by Bessai et al. [51], Wei and Blake [17], Euting et al. [37], and Cai et al. [38]. All of them present different approaches for optimizing the usage of cloud resources, in form of VMs, to execute processes in a cost- and resource-efficient manner.

Rosinsky et al. [52] present a migration-aware optimization strategy for multi-tenant process execution in the cloud. In comparison to our work, they use VMs for the execution. Moreover, the approach does not execute the process activities on different cloud resources but on the same as the BPMS is running. This restricts the elasticity of the approach.

In [53], the authors present a highly scalable object-aware process management engine that is using distributed microservices on cloud resources. As an underlying architecture, the authors are using the actor model, where each microservice is an actor (comparable to our services) and connected to different other actors according to a data model. While this actor model allows high scalability, the integration of new objects into the data model is computationally very costly. Furthermore, each actor has to be aware of the process structure since the actors directly communicate with each other. This is not the case in our approach.

All of the approaches discussed so far aim at using cloud resources in the form of VMs for the execution of business processes, and none of the approaches consider the usage and resource optimization based on containers. However, there are several approaches which use containers for scheduling and resource allocation of arbitrary services in non-process settings: Especially the work of Vaquero et al. [54], Pahl [11], Xu et al. [15], Hoenisch et al. [20] and Nardelli et al. [21] have to be mentioned here. While they are discussing the usage of containers for the execution of applications to increase scalability and isolation of the applications, none of them consider containers for processes.

For instance, Hoenisch et al. [20] present a multi-objective optimization model that optimizes the deployment of applications on containers, which are themselves deployed on VMs, for cost-efficiency. In comparison to ViePEP-C and GeCoVM, this approach considers single applications and not processes. While a single process activity can be seen as a single application, this approach can not consider subsequent process activities for the optimization. The approach from [20] can only consider currently requested and running applications. Thus, a postponing of a process activity so that an overlapping with other process activities, whose executions will start shortly, is not possible. As shown in our evaluation, such a postponing further helps to achieve a cost-efficient execution.

Related work considering the execution of business processes on container-based cloud architectures is presented by Boukadi et al. [18]. Similar to the work at hand, Boukadi et al. present a business process execution approach that uses containers on VMs for the execution of the process activities. For this, they extend the simulation tool ContainerCloudSim and present a linear program that finds a global optimum solution, in respect of cost efficiency, for the deployment of the process activities. The objective function minimizes the deployment and data transfer cost and considers a minimum required VM QoS. As QoS, they consider VM security and availability. In comparison to our approach, the approach in [18] does not consider SLAs and does not allow slight violations of the SLAs to minimize the cost further. Besides, a simulator instead of a framework and testbed are used.

In [41], we present an optimization approach that performs an optimization of the start times of process activities to achieve an overlapping of them. This approach is one part of GeCoVM as presented in Section 5. The optimization presented in [41] only considers the start time of the process activities, which are then executed on containers. In comparison to the work at hand, the container deployment on the VMs is not optimized. Instead, the containers are directly executed on the cloud resources. While this is possible at some cloud providers (e.g., Amazon AWS and Google Cloud), the execution of containers on VMs is the norm [20].

Gerlach et al. [55] present in their work a scientific workflow (SWF) platform called Skyport that uses Docker containers for the deployment of the workflow services. By the usage of containers, they aim at providing a reproducible software deployment solution with isolated software applications that can be applied to SWFs. Zheng et al. [56] also aim at solving the problems that can occur in SWFs by using containers. They propose a solution with a two-level resource scheduling model to efficiently share resources among different SWFs. The authors show that by using a container-based scheduling platform, a higher system efficiency and a lower performance loss can be achieved. While this work is undoubtedly interesting, there are significant differences between SWFs and the business processes supported by ViePEP-C, which prevent the direct adaptation for our purposes [3].

In summary, most of the related work considers VMs as an execution environment for business processes. As discussed in Section 1, this leads to a rather coarse-grained deployment solution that reduces the flexibility and ad hoc elasticity of the deployment. None of the above-mentioned publications present a platform that uses containers as an execution environment for business processes.

9 Conclusion

In this paper, we have presented ViePEP-C, a novel eBPMS for the execution of business processes on cloud-based computational resources. ViePEP-C offers an API that can be used for the execution of business processes, composed of different process activities and different process patterns. ViePEP-C uses resource allocation and task scheduling algorithms, e.g., the presented GeCoVM algorithm, to achieve an efficient resource allocation while considering process-specific SLAs. The output of the algorithm is a provisioning

plan that is used for the process activity execution, respectively its corresponding services, on cloud resources by the use of containers on VMs. For this, ViePEP-C takes over the tasks of a cloud controller, i.e., deploys VMs, deploys containers on the VMs, and monitors the infrastructure. The monitored information is used to adapt the provisioning plan to current situations at runtime.

In our evaluation, we have shown that by using a container-based eBPMS, a significant improvement with respect to cost and deployment times, in comparison to a state-of-the-art approach, can be achieved. For instance, in case of the constant process execution arrival pattern with the lenient SLA level, ViePEP-C together with GeCoVM reduced the leasing cost by 21.60% in comparison to the state-of-the-art baseline. Furthermore, we have shown that ViePEP-C detects and handles failure situations, e.g., the loss of connection to a VM.

In our future work, we want to use ViePEP-C to analyze different scheduling algorithms and benchmark further optimization algorithms, e.g., based on a linear program. Furthermore, we plan to extend the scheduling algorithms in a way that not only CPU and RAM are considered, but also required disk space and disk accesses. We will also take a look into unikernels and how they can be used in ViePEP-C to decrease the resource consumption further and to increase the flexibility of the business process execution.

ACKNOWLEDGMENTS

This work is partially supported and funded by the Austrian Research Promotion Agency (FFG) via the "Austrian Competence Center for Digital Production" (CDP) under the contract number 854187, and by the FFG project 866270.

REFERENCES

[1] M. Dumas, M. L. Rosa, J. Mendling, and H. A. Reijers, *Fundamentals of Business Process Management, Second Edition*. Springer, 2018.
[2] S. Schulte, P. Hoenisch, C. Hochreiner, S. Dustdar, M. Klusch, and D. Schuller, "Towards Process Support for Cloud Manufacturing," in *IEEE 18th Int. Enterprise Distributed Object Comput. Conf.*, 2014, pp. 142–149.
[3] S. Schulte, C. Janiesch, S. Venugopal, I. Weber, and P. Hoenisch, "Elastic Business Process Management: State of the art and open challenges for BPM in the cloud," *Future Generation Computer Sys.*, vol. 46, pp. 36–50, 2015.
[4] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow Patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.
[5] M. Weske, *Business Process Management: Concepts, Languages, Architectures*. Springer, 2012.
[6] S. Dustdar, Y. Guo, B. Satzger, and H. L. Truong, "Principles of Elastic Processes," *IEEE Internet Comput.*, vol. 15, no. 5, pp. 66–71, 2011.
[7] P. Hoenisch, D. Schuller, S. Schulte, C. Hochreiner, and S. Dustdar, "Optimization of Complex Elastic Processes," *IEEE Trans. on Services Comput.*, vol. 9, no. 5, pp. 700–713, 2016.
[8] R. B. Halima, S. Kallel, W. Gaaloul, and M. Jmaiel, "Optimal Cost for Time-Aware Cloud Resource Allocation in Business Processes," in *IEEE Int. Conf. on Services Comput.*, 2017, pp. 314–321.
[9] Z. Wu, X. Liu, Z. Ni, D. Yuan, and Y. Yang, "A market-oriented hierarchical scheduling strategy in cloud workflow systems," *The Journal of Supercomput.*, vol. 63, no. 1, pp. 256–293, 2013.
[10] M. Mao and M. Humphrey, "A Performance Study on the VM Startup Time in the Cloud," in *IEEE 5th Int. Conf. on Cloud Comput.*, 2012, pp. 423–430.
[11] C. Pahl, "Containerization and the PaaS Cloud," *IEEE Cloud Comput.*, vol. 2, no. 3, pp. 24–31, 2015.
[12] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs Containerization to Support PaaS," in *IEEE Int. Conf. on Cloud Engineering*, 2014, pp. 610–614.
[13] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, 2014.
[14] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," *Advanced Science and Technology Letters*, vol. 66, no. 2, pp. 105–111, 2014.
[15] X. Xu, H. Yu, and X. Pei, "A Novel Resource Scheduling Approach in Container Based Clouds," in *IEEE 17th Int. Conf. on Computational Science and Engineering*, 2014, pp. 257–264.
[16] E. Juhnke, T. Dörnemann, D. Bock, and B. Freisleben, "Multi-objective Scheduling of BPEL Workflows in Geographically Distributed Clouds," in *Int. Conf. on Cloud Comput.*, 2011, pp. 412–419.
[17] Y. Wei and M. B. Blake, "Proactive virtualized resource management for service workflows in the cloud," *Comput.*, vol. 96, no. 7, pp. 1–16, 2014.
[18] K. Boukadi, R. Grati, M. Rekik, and H. B. Abdallah, "From VM to Container: A Linear Program for Outsourcing a Business Process to Cloud Containers," in *16th OTM Confederated Int. Conf. On the Move to Meaningful Internet Sys.*, 2017, pp. 488–504.
[19] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Cost-efficient enactment of stream processing topologies," *PeerJ Computer Science*, vol. 3, 2017, paper e141.
[20] P. Hoenisch, I. Weber, S. Schulte, L. Zhu, and A. Fekete, "Four-Fold Auto-Scaling on a Contemporary Deployment Platform Using Docker Containers," in *Int. Conf. on Service-Oriented Comput.*, 2015, pp. 316–323.
[21] M. Nardelli, C. Hochreiner, and S. Schulte, "Elastic Provisioning of Virtual Machines for Container Deployment," in *8th ACM/SPEC Int. Conf. on Performance Engineering Companion*, 2017, pp. 5–10.
[22] D. Wu, M. J. Greer, D. W. Rosen, and D. Schaefer, "Cloud manufacturing: Strategic vision and state-of-the-art," *Journal of Manufact. Sys.*, vol. 32, no. 4, pp. 564–579, 2013.
[23] X. Xu, "From cloud computing to cloud manufacturing," *Robotics and computer-integrated manufact.*, vol. 28, no. 1, pp. 75–86, 2012.
[24] R. Rosen, G. von Wichert, G. Lo, and K. D. Bettenhausen, "About The Importance of Autonomy and Digital Twins for the Future of Manufacturing," *IFAC-PapersOnLine*, vol. 48, no. 3, pp. 567–572, 2015.
[25] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.
[26] S. Rohjans, C. Dänekas, and M. Uslar, "Requirements for smart grid ICT-architectures," in *3rd IEEE PES Int. Conf. and Exhibition on Innovative Smart Grid Technologies*, 2012, pp. 1–8.
[27] A. Q. Gill, D. Bunker, and P. Seltsikas, "An Empirical Analysis of Cloud, Mobile, Social and Green Computing: Financial Services IT Strategy and Enterprise Architecture," in *IEEE 9th Int. Conf. on Dependable, Autonomic and Secure Computing*. IEEE, 2011, pp. 697–704.
[28] R. S. Mans, N. C. Russell, W. M. van der Aalst, A. J. Moleman, and P. J. Bakker, "Schedule-Aware Workflow Management Systems," in *Trans. on Petri Nets and Other Models of Concurrency IV*. Springer, 2010, pp. 121–143.
[29] K. Agarwal, B. Jain, and D. E. Porter, "Containing the Hype," in *Proceedings of the 6th Asia-Pacific Work. on Sys.*, 2015, p. 8.
[30] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and Virtual Machines at Scale: A Comparative Study," in *Proceedings of the 17th International Middleware Conference*, 2016, p. 1.
[31] D. Whitley, "A Genetic Algorithm Tutorial," *Stat. Comput.*, vol. 4, pp. 65–85, 1994.
[32] Z. Ye, X. Zhou, and A. Bouguettaya, "Genetic algorithm based QoS-aware service compositions in cloud computing," in *Int. Conf. on Database Sys. for Advanced Applications*, 2011, pp. 321–334.
[33] S. Schulte, P. Hoenisch, S. Venugopal, and S. Dustdar, "Introducing the Vienna Platform for Elastic Processes," in *Performance Assessment and Auditing in Service Comput. Works. at 10th Int. Conf. on Service-Oriented Comput.*, 2013, pp. 179–190.
[34] S. Schulte, D. Schuller, P. Hoenisch, U. Lampe, S. Dustdar, and R. Steinmetz, "Cost-Driven Optimization of Cloud Resource Allocation for Elastic Processes," *Int. Journal of Cloud Comput.*, vol. 1, no. 2, pp. 1–14, 2013.

[35] M. Borkowski, S. Schulte, and C. Hochreiner, "Predicting cloud resource utilization," in *2016 IEEE/ACM 9th Int. Conf. on Utility and Cloud Comput.*, 2016, pp. 37–42.
[36] N. Hayashibara, A. Cherif, and T. Katayama, "Failure detectors for large-scale distributed systems," in *IEEE Symposium on Reliable Distributed Sys.*, 2002, pp. 404–409.
[37] S. Euting, C. Janiesch, R. Fischer, S. Tai, and I. Weber, "Scalable business process execution in the cloud," in *2014 IEEE International Conference on Cloud Engineering*. IEEE, 2014, pp. 175–184.
[38] Z. Cai, X. Li, and J. N. Gupta, "Critical Path-Based Iterative Heuristic for Workflow Scheduling in Utility and Cloud Computing," in *Int. Conf. on Service-Oriented Comput.*, 2013, pp. 207–221.
[39] E. S. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *IEEE Trans. on Parallel and Distributed Sys.*, vol. 5, no. 2, pp. 113–120, 1994.
[40] M. Yoo, "Real-time task scheduling by multiobjective genetic algorithm," *Journal of Sys. and Software*, vol. 82, no. 4, pp. 619–628, 2009.
[41] P. Waibel, A. Yeshchenko, S. Schulte, and J. Mendling, "Optimized Container-Based Process Execution in the Cloud," in *17th OTM Confederated Int. Conf. On the Move to Meaningful Internet Sys.*, 2018, pp. 3–21.
[42] P. A. Diaz-Gomez and D. F. Hougen, "Initial Population for Genetic Algorithms: A Metric Approach," in *Proceedings of the 2007 International Conference on Genetic and Evolutionary Methods*, 2007, pp. 43–49.
[43] Y. Leung, Y. Gao, and Z.-B. Xu, "Degree of population diversity-a perspective on premature convergence in genetic algorithms and its markov chain analysis," *IEEE Trans. on Neural Networks*, vol. 8, no. 5, pp. 1165–1176, 1997.
[44] D. Bhandari, C. Murthy, and S. K. Pal, "Variance as a stopping criterion for genetic algorithms with elitist model," *Fundamenta Informaticae*, vol. 120, no. 2, pp. 145–164, 2012.
[45] T. A. Curran and G. Keller, *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. Prentice Hall PTR, 1997.
[46] G. Keller and T. Teufel, *SAP R/3 Process Oriented Implementation: Iterative Process Prototyping*. Addison-Wesley Longman Publishing Co., 1998.
[47] J. Mendling, H. Verbeek, B. F. van Dongen, W. M. P. van der Aalst, and G. Neumann, "Detection and prediction of errors in EPCs of the SAP reference model," *Data & Knowledge Engineering*, vol. 64, no. 1, pp. 312–329, 2008.
[48] M. Sigwart, C. Hochreiner, M. Borkowski, and S. Schulte, "FakeLoad: An Open-Source Load Generator," Distributed Systems Group, TU Wien, Tech. Rep. TUV-1942-2018-01, 2018. [Online]. Available: http://dsg.tuwien.ac.at/staff/mborkowski/pub/TR_FakeLoad.pdf
[49] M. E. Frincu, S. Genaud, and J. Gossa, "On the efficiency of several VM provisioning strategies for workflows with multi-threaded tasks on clouds," *Comput.*, vol. 96, no. 11, pp. 1059–1086, 2014.
[50] P. Leitner, W. Hummer, B. Satzger, C. Inzinger, and S. Dustdar, "Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud," in *Int. Conf. on Cloud Comput.*, 2012, pp. 213–220.
[51] K. Bessai, S. Youcef, A. Oulamara, and C. Godart, "Bi-criteria strategies for business processes scheduling in cloud environments with fairness metrics," in *Int. Conf. on Research Challenges in Information Science*, 2013, pp. 1–10.
[52] G. Rosinosky, S. Youcef, and F. Charoy, "Efficient Migration-Aware Algorithms for Elastic BPMaaS," in *Int. Conf. of Business Process Management*, 2017, pp. 147–163.
[53] K. Andrews, S. Steinau, and M. Reichert, "Engineering a Highly Scalable Object-aware Process Management Engine Using Distributed Microservices," in *17th OTM Confederated Int. Conf. On the Move to Meaningful Internet Sys.*, 2018.
[54] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 45–52, 2011.
[55] W. Gerlach, W. Tang, K. Keegan, T. Harrison, A. Wilke, J. Bischof, M. D'Souza, S. Devoid, D. Murphy-Olson, N. Desai, and F. Meyer, "Skyport: Container-based execution environment management for multi-cloud scientific workflows," in *5th Int. Works. on Data-Intensive Comput. in the Clouds*, 2014, pp. 25–32.
[56] C. Zheng, B. Tovar, and D. Thain, "Deploying High Throughput Scientific Workflows on Container Schedulers with Makeflow and Mesos," in *IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Comput.*, 2017, pp. 130–139.

Philipp Waibel is a PhD student at the Distributed System Group at TU Wien. He works at the Institute for Information Business at the Vienna University of Economics and Business. Philipp's research interests cover the whole spectrum of cloud computing, with the main focus on cost-efficient usage of cloud resources.

Christoph Hochreiner is a researcher and software engineer. His research interests cover the whole spectrum of cloud computing with a specific focus on data stream processing and security implications. Hochreiner has a PhD in computer science from TU Wien, Austria.

Stefan Schulte is Associate Professor at the Faculty of Informatics at TU Wien. His research interests span the areas of data engineering and the Internet of Things, and the application of blockchain technologies for data provenance and process execution. Findings from his research have been published in more than 90 refereed scholarly publications.

Agnes Koschmider is a senior researcher at the Institute of Applied Informatics and Formal Description Methods at KIT. She received her doctoral degree and venia legendi in Applied Informatics. Her current research focuses on predictive behavior analysis and extraction of process models from sensor logs in order to support decision-making processes. Her work was awarded with the Heilmann-Preis, junior fellowship of the German Informatics Society and a post-doctoral fellowship from the University of Pretoria (South Africa).

Prof. Dr. Jan Mendling is a Full Professor with the Institute for Information Business at Wirtschaftsuniversität Wien (WU Vienna), Austria. His research interests include various topics in the area of business process management and information systems. He has published more than 350 research papers and articles, among others in ACM Transactions on Software Engineering and Methodology, IEEE Transaction on Software Engineering, and Decision Support Systems. He is co-author of the textbooks Fundamentals of Business Process Management and Wirtschaftsinformatik.