Kubernetes-in-the-Loop: Enriching Microservice Simulation Through Authentic Container Orchestration

Martin Straesser[1](✉), Patrick Haas[1], Sebastian Frank[2], Alireza Hakamian[3], André van Hoorn[2], and Samuel Kounev[1]

[1] University of Würzburg, Würzburg, Germany
{martin.straesser,samuel.kounev}@uni-wuerzburg.de,
patrick.haas@informatik.uni-wuerzburg.de

[2] University of Hamburg, Hamburg, Germany
{sebastian.frank,andre.van.hoorn}@uni-hamburg.de

[3] University of Stuttgart, Stuttgart, Germany
mir-alireza.hakamian@iste.uni-stuttgart.de

Abstract. Microservices deployed and managed by container orchestration frameworks like Kubernetes are the bases of modern cloud applications. In microservice performance modeling and prediction, simulations provide a lightweight alternative to experimental analysis, which requires dedicated infrastructure and a laborious setup. However, existing simulators cannot run realistic scenarios, as performance-critical orchestration mechanisms (like scheduling or autoscaling) are manually modeled and can consequently not be represented in their full complexity and configuration space. This work combines a state-of-the-art simulation for microservice performance with Kubernetes container orchestration. Hereby, we include the original implementation of Kubernetes artifacts enabling realistic scenarios and testing of orchestration policies with low overhead. In two experiments with Kubernetes' *kube-scheduler* and *cluster-autoscaler*, we demonstrate that our framework can correctly handle different configurations of these orchestration mechanisms boosting both the simulation's use cases and authenticity.

Keywords: Kubernetes · Microservices · Container orchestration · Discrete event simulation · Cloud computing · Software performance

1 Introduction

Microservices are a modern architectural style for developing complex software systems with a steadily increasing adoption in practice [20]. Containers are the most popular deployment technology for microservices enabling improved elasticity and faster start times compared to virtual machines [38,39,43,49]. As manual management of hundreds to thousands of containers is impractical, container

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2024
Published by Springer Nature Switzerland AG 2024. All Rights Reserved
E. Kalyvianaki and M. Paolieri (Eds.): VALUETOOLS 2023, LNICST 539, pp. 82–98, 2024.
https://doi.org/10.1007/978-3-031-48885-6_6

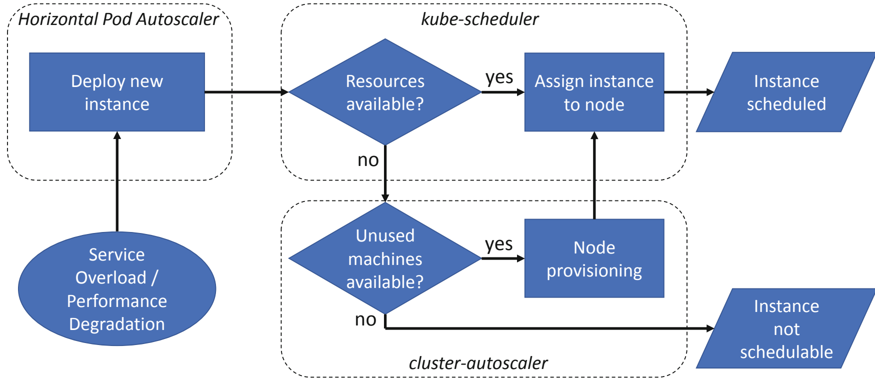


Fig. 1. Interactions between Kubernetes components.

orchestration (CO) frameworks are widely used, with Kubernetes being the most popular platform [13]. CO frameworks must maintain an acceptable quality of service of the managed applications, fulfilling many performance-relevant tasks, such as autoscaling, scheduling, or container networking [41]. As previous studies have shown, CO frameworks can significantly influence performance metrics of deployed applications, like response times or CPU usage [35,37,44,45].

Consequently, microservice performance models have to consider the CO frameworks' behavior for accurate performance prediction. However, several challenges arise here. First, the orchestration mechanisms to be simulated are highly complex. For example, the Kubernetes scheduling algorithm has nine extension points that can be individually configured for specific use cases [28]. Second, there are dependencies between single orchestration mechanisms or application-level patterns [40]. Figure 1 shows how Kubernetes' Horizontal Pod Autoscaler, *kube-scheduler* and *cluster-autoscaler* work together when new service instances need to be deployed reacting to a performance degradation. Third, CO frameworks are regularly updated, leading to a change in behavior and quickly causing performance models to be outdated. Previous work in microservice performance modeling and simulation integrates either no or only self-implemented, simplified runtime orchestration mechanisms [8,17,22,46].

This work aims to establish a link between microservice performance simulation and modern CO frameworks. We present an approach that enables connecting a discrete event simulation and an event-based system. We use this approach to extend the microservice simulator MiSim [17] with Kubernetes orchestration mechanisms using their original code artifacts. By this, orchestration mechanisms can be integrated with simulation in their full complexity without requiring abstract models, making the simulation more authentic. Our implementation uses an adapter that translates events from the simulation to Kubernetes events and vice versa. Actions of the Kubernetes components directly modify simulated entities. We validate our approach in experiments with Kubernetes' kube-

scheduler and cluster-autoscaler. We show that different complex scheduling and autoscaling configurations can be tested in the simulation with low overhead.

Microservice application designers can use our framework to evaluate realistic scenarios with impact factors present at runtime early in the development process. Furthermore, performance engineers could use the developed framework to study the performance of a microservice application in what-if scenarios in conjunction with different workloads and Kubernetes component configurations. Container orchestration researchers can use the framework to design new orchestration mechanisms and evaluate their behavior with different applications. Combining a microservice simulation with original Kubernetes components allows the use of the full complexity of the orchestration mechanisms in each of these use cases, leading to a more expressive and realistic simulation. All in all, the contribution of this paper is twofold:

- We present an approach to connect a discrete-event simulation with the event-based system of Kubernetes and use it to extend the state-of-the-art microservice performance simulator MiSim [17].
- We analyze the usability and overhead of our approach in two detailed case studies on (i) container scheduling and (ii) cluster autoscaling.

The remainder of this paper is structured as follows. In the next section, we present the foundations of this work: the microservice simulator MiSim, as well as information about the architecture of Kubernetes. Section 3 discusses related works and the differences to this paper. Section 4 explains our approach in detail, while Sect. 5 presents our case studies with the kube-scheduler and cluster-autoscaler as examples of integrated Kubernetes components. Section 6 discusses the advantages, disadvantages, and limitations of our approach, while Sect. 7 concludes the work.

2 Foundations

2.1 MiSim: A Discrete Event Simulator for Microservice Performance and Resilience

Frank et al. [17] propose MiSim, a simulator for microservice performance and resilience. Technically, MiSim utilizes a discrete-event simulation (DES) realized through the DES framework DESMO-J [32]. As an input, a lightweight architectural description with microservice descriptions and active resilience mechanisms is required. Microservice properties include instance numbers, available operations and their dependencies, CPU demands, and more. These elements are simulated together with dynamic aspects, like user requests. Notably, MiSim supports the simulation of common mechanisms relevant to request behavior and thus performance, i.e., circuit breakers, connection limiters, retries, load balancers, and autoscalers. MiSim can process time-varying workloads and outputs several performance metrics, such as response times and CPU utilization. MiSim is conceptually and technically developed to be extensible. In particular, it uses

the Strategy design pattern for many components, allowing to substitute their behavior easily. For example, four CPU scheduling policies are implemented. By default, MiSim uses Multi-Level Feedback Queues and the SARR algorithm proposed by Matarneh [34] that schedules computation time based on the median of the remaining burst time of all scheduled processes.

2.2 Kubernetes Architecture and Communication Patterns

In this work, we exploit the internal architecture and communication of Kubernetes [27] to connect to MiSim. The Kubernetes control plane consists of four essential components: an *etcd database*, the *kube-controller-manager*, the *kube-scheduler*, and the *kube-apiserver*. The etcd database persists the states of the cluster resources. The controller manager runs a series of processes that, for example, monitor nodes or jobs. The scheduler assigns containers, organized in so-called pods, to worker nodes for execution. The central component is the kube-apiserver, which handles every communication between the control plane and worker nodes. It is based on the Kubernetes API [25] and serves endpoints for every resource type in the cluster (e.g., nodes, pods). When a consumer (e.g., the kube-scheduler) requests information about a resource type in the cluster (e.g., nodes), it queries the corresponding endpoint. Kubernetes uses a "list-then-watch" principle to distribute information to all interested consumers. After starting a Kubernetes component, it requests the latest list of selected resources with a list request. This list contains a resource version, which is later used as a reference to this list. The consumer stores this list in an internal cache. Then the consumer sends a watch request with the resource version of its cached list as a query parameter. This request sets up an HTTP streaming connection between the kube-apiserver and the consumer. If a change to subscribed resources happens (e.g., a new node is added to the cluster), a watch event is emitted. The watch event has a type (*added*, *modified*, or *deleted*) and a representation of the affected resource. The consumer receives this event, modifies its cache, and performs an action if necessary. We refer to the official documentation for more information about the Kubernetes API [25].

3 Related Work

There are different ways to evaluate the performance of container orchestration (CO) frameworks. COFFEE [41] is a benchmarking framework for CO frameworks that allows the characterization of the performance of an orchestrator (configuration) with different metrics. Other works use benchmarking approaches, primarily focusing on Kubernetes [11,21,37,47]. While benchmarking can provide the most accurate results and realistic scenarios, it requires an extensive infrastructure. Simulations usually give less precise results but are much more cost-efficient. There are several simulations for component-based systems that cover models of selected CO mechanisms, e.g., load balancing [9,50],

autoscaling [7] or networking [46]. In contrast to these works, we aim to combine microservice simulation with multiple CO tasks and mechanisms using their original implementation, i.e., without hand-crafted models.

Previous works have already dealt with the connection between simulations and real code. *Software-in-the-Loop* [14] is a term in this area that describes this connection, inspired by Hardware-in-the-Loop simulations. Such approaches focus primarily on testing control software in autonomous systems [10,19]. Erb and Kargl [16] note general analogies between discrete-event simulations and event-driven architectures. In the area of software performance simulations, a similar concept has been used by Von Massow et al. [33], combining a simulation and an adaptation controller. In this work, we apply a similar concept to microservice simulation and Kubernetes using scheduling and cluster autoscaling as exemplary CO mechanisms. In general, both Kubernetes scheduling and cluster autoscaling are active research areas, as confirmed by recent articles [12,42,48]. For Kubernetes scheduling, there is a community project called kube-scheduler-simulator [30] where different scheduling policies can be tested similarly to our experiment in Sect. 5.1 but without evaluating their impact on the performance of deployed services.

4 Approach

In this section, we explain our concept to integrate Kubernetes orchestration mechanisms in the microservice simulation MiSim. First, some entities and basic CO models need to be integrated into MiSim. Section 4.2 presents our approach to combine discrete event simulation and event-based systems, while Sect. 4.3 shows how we handle incompatible models and events. We conclude this section with a detailed look at Kubernetes scheduling and cluster autoscaling. Our implementation [1] is to be seen as a wrapper that is 100% compatible with the MiSim core and builds on its extension points.

4.1 Integration of Basic Container Orchestration Models

The MiSim core does not have any deployment models but only models for microservice architectures and resilience patterns. CPU resources are considered independently from nodes. For this reason, we add some basic entities and associated events relevant at deployment time in the simulation: A *node* is modeled as a bunch of resources. A *cluster* is a graph with a set of nodes connected with edges indicating network latencies between individual nodes. A *container* has a 1:1 relationship to a microservice instance from the MiSim core and is deployed on exactly one node. Next, we need to define which aspects of container orchestration should be considered in our simulation. We build on the work of Straesser et al. [41] in which eight performance-relevant tasks of container orchestration frameworks are identified: container deployment, scheduling, resource allocation, availability, health monitoring, scaling, load balancing, and networking. We add

support for all of these tasks in the simulation and implement simple orchestration mechanisms directly.

Performance overheads due to the deployment of containers are taken into account in the simulation by startup times. Each container has a certain resource requirement, which has to be considered during scheduling. When the scheduler assigns a container to a node, a certain amount of resources are reserved and unavailable for other containers on the same node. The scheduler has to check if enough resources are available on the node. We implement two standard scheduling algorithms: random and round-robin. Health monitoring is represented in the simulation by periodic events that check the status of all deployed containers. Restarts can be simulated if a container has been crashed (e.g., by MiSim's chaos monkey function). Networking is determined by the cluster network graph indicating the mean latency and standard deviation between nodes. Furthermore, we have adapted one autoscaling and several load balancing strategies already available in MiSim to work with the new container and node models.

4.2 Connecting a Discrete Event Simulation and an Event-Driven System

General Concept. This section presents our abstract concept for combining a discrete event simulation (DES) with an event-driven system (EDS). Our goal is to extend the authenticity and use cases of the DES by including mechanisms/algorithms used in the EDS. A discrete-event simulation [31] is a sequence of events where each event $e \in E$ is associated with a time point t in the simulation time. Here, E denotes the set of all possible types of events. Each event e is associated with a function $F(e)$, which is executed as soon as e occurs. This function can, for example, change modeled entities in the simulation. An event-driven system [36] is a popular architectural style for developing component-based software. Two components communicate with each other by the sender (producer) emitting an event e' and sending it via a proxy and middleware (channel) to the receiver (consumer). In the following, we denote the set of defined event types in the EDS as E'.

Consider a set of special event types $\Theta \subseteq E$ in the DES. We modify all functions $F(e)$ for all events $e \in \Theta$ such that these events are passed to the EDS. If such an event e occurs at simulation time t in the simulation, two more steps are needed to make the event processable by the EDS. First, we need a transformation τ that maps the event e to an event or a sequence of events $e' \in E'$, as usually not all events in the DES are represented in the EDS and vice versa. Second, passing a representation of the simulation state $S(t)$ at time t may be necessary. This is especially necessary if not all events $e \in E$ are passed to the EDS. $S(t)$ should contain the states of all simulated entities relevant to the EDS. If the interface of the EDS allows only event-based communication, the simulation state $S(t)$ must be transformed into a sequence of events from E'. When sending e' and $S(t)$, the DES acts as an event producer from the EDS' point of view. The EDS forwards all received events to consuming components. The consumers now execute a black-box logic and generate responses $r \in E'$

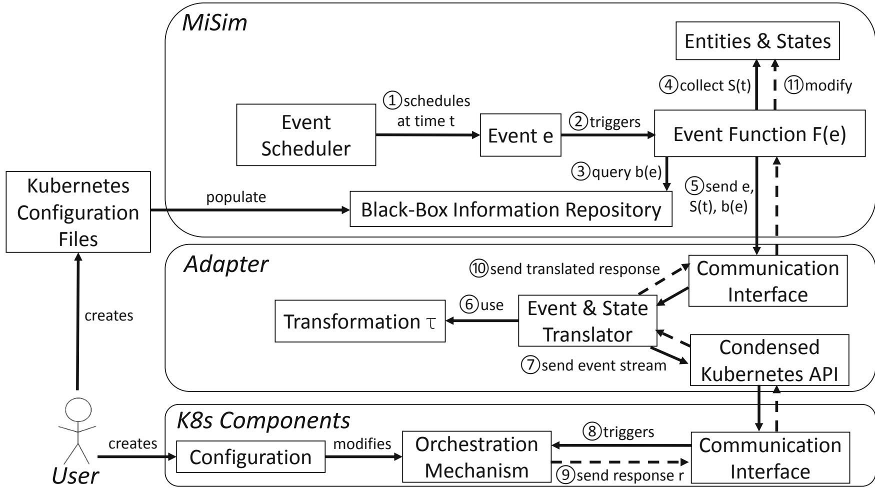


Fig. 2. Overview of our approach and component interactions.

as reactions to received events. The responses are passed back to the DES and processed by $F(e)$. As a consequence, entities or states in the simulation might be modified.

Application to MiSim and Kubernetes. In our use case, we connect the DES MiSim with event-based orchestration mechanisms of Kubernetes. Figure 2 shows an overview of our developed framework. To bridge the gap between MiSim and Kubernetes, we decided to use an adapter [2]. This has the advantage that the simulation remains slim, and if no Kubernetes components are to be used, no unnecessary code must be loaded and executed. We select a set of events Θ from MiSim, that are relevant for Kubernetes components. Θ includes all events affecting containers and nodes, while several other MiSim events, e.g., related to CPUs or logging, are excluded. The simulation sends selected events and the state of the simulated entities to the adapter. The adapter is responsible for the event transformation τ and the processing and transformation of the components' responses r. It implements relevant parts of the Kubernetes API, especially endpoints for pods and nodes. Hence, it acts like the kube-apiserver from the Kubernetes components' point of view. Because the kube-apiserver handles all communication in the Kubernetes control plane, we only need this one adapter for different components. The adapter and Kubernetes components are started prior to the simulation. We initialize the component caches with empty lists for all resource types. When the simulation is started, events for modeled resource types (like pods or nodes) are forwarded to the Kubernetes components.

4.3 Handling Incompatible Events and Models

General Concept. In the following, we consider cases where the DES and the EDS use different models or entities that cannot be transformed into each other. In the following, let M_D be the set of models and entities which appear in the DES but have no equivalent in the EDS. Similarly, let M_E be the set of models and entities that exist in EDS but have no equivalent in the DES. Let M_{D+E} be the set of models and entities with equivalences in DES and EDS. First, we consider elements from M_D. Since, in our case, we are only interested in the results of the DES, these do not pose a problem. We simply exclude all events concerning entities M_D from the set Θ. We just have to ensure that we correctly model interactions between entities from M_D and entities from M_{D+E}.

We can divide elements from M_E into two groups. We ignore the group of models or entities from M_E that have no interaction with elements from M_{D+E} or whose interactions should not be considered in the DES. The second group of elements from M_E that have relevant interactions with elements from M_{D+E} has to be considered in our approach. These can, for example, influence the response r to an event e from the DES and thus influence the simulation run. To solve this problem, we propose to use a black-box information repository (BIR). The DES receives a set of additional EDS-specific information B before starting a run, as well as information about which events $e \in \Theta$ should use which information $b \in B$. The simulation does not interpret or change elements in B but stores them in the repository. The function $F(e)$ passes them to the EDS for specific events e together with the simulation state $S(t)$. This way, entities not modeled in the DES can still be considered in simulation runs and in the black-box logic of the EDS components.

Application to MiSim and Kubernetes. In this paper, we choose scheduling and cluster autoscaling as two Kubernetes orchestration mechanisms to integrate into MiSim. The scheduling algorithm comprises nine extension points where user-defined plugins can be attached [28]. As these plugins contain arbitrary logic, it is impossible to integrate this configuration space in the simulation using traditional models. Similarly, the *cluster-autoscaler* has more than 30 configuration options [18]. Simulative evaluation is especially beneficial since cluster autoscaling can only be performed with special infrastructure where nodes can be added or deleted on demand. Both mechanisms use entities that are not present in MiSim (like node affinities, labels, or machine set definitions). Kubernetes uses YAML files to define these entities. Our framework stores these user-created files in the BIR and forwards the contents on specific events. In the following, we look deeper into how the kube-scheduler and cluster-autoscaler are integrated into MiSim using the aforementioned concepts.

4.4 A Detailed Look at Scheduling and Cluster Autoscaling

The kube-scheduler receives a request from the adapter whenever an event in MiSim is triggered that creates a new container. This request contains information about the container to be deployed (e.g., its resource requirements) and all

other currently deployed containers (the simulation state). The selected node or an error (e.g., if all resources in the cluster are reserved) is expected as a response. The adapter converts this request into a series of Kubernetes watch events and sends them to the scheduler. The kube-scheduler determines nodes with enough resources and selects a node for the container according to its scheduling policy. This policy is set as a configuration [28] at the start, like in a real cluster. If no node is available, the kube-scheduler returns an error to the adapter, which passes it on to the simulation.

Scheduling policies in Kubernetes can also be influenced by other factors (e.g., pod affinities [24]). These are examples of critical elements from the set M_E, i.e., entities that exist in Kubernetes but do not exist in the simulation and yet affect relevant tasks in the simulation. For the example mentioned above, we create node definitions with labels and pod definitions with affinities in the form of Kubernetes YAML files that populate the BIR at the simulation start. The simulation and adapter forward them on events concerning nodes or pods, respectively. The kube-scheduler uses this information for scheduling. With this approach, we can cover the scheduling policies' maximum complexity without implementing new logic ourselves. We demonstrate the different usage of scheduling policies in Sect. 5.1.

As a second Kubernetes component, we integrate the cluster-autoscaler (CA) into the simulation. It becomes active whenever the kube-scheduler cannot deploy a pod (upscaling, see Fig. 1) or when the utilization of a node falls below a certain threshold (downscaling). Hence, the CA subscribes events for pods and nodes; our adapter forwards all decisions of the kube-scheduler directly to the CA. Furthermore, the CA needs endpoints to manage node groups. Here, different implementations for cloud providers exist [26]. We use the generic open-source Kubernetes Cluster API [29]. Note that the chosen cloud provider implementation does not influence the autoscaling logic. By now, we support the integration of *MachineSets* from the Cluster API. A MachineSet is a set of machines with equal resources that can be scaled from a specified minimum value (≥ 0) to a maximum value. The user can specify definitions of MachineSets as part of the BIR at the simulation start. We compare two different upscaling policies for the CA in Sect. 5.2.

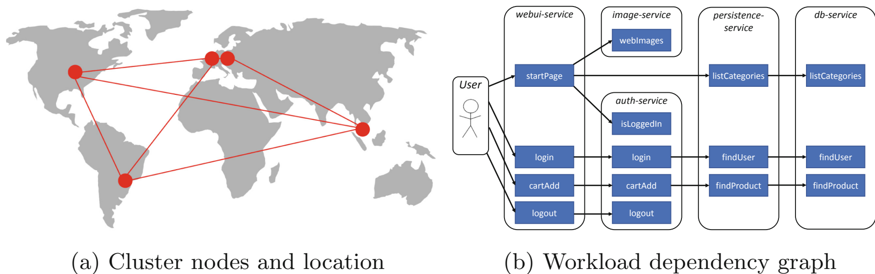
5 Evaluation

In the following, we provide empirical evidence on the usefulness of our approach for evaluating orchestration policies and validate the behavior of the included Kubernetes components by conducting two experiments. First, we consider a microservice application whose services are deployed in a global cluster. We examine the interactions between different scheduling policies, the application, and the cluster architecture. In the second experiment, we deploy an increasing number of service instances in a heterogeneous cluster with two machine types. We look at different expansion policies of the CA and show that our framework can correctly capture the interactions between the CA and the kube-scheduler. We provide a CodeOcean capsule [3] where all experiments can be reproduced.

5.1 Scheduling Policies in a Global Cluster

Overview. In this experiment, we model a cluster with nodes in different geographic regions. A microservice reference application is deployed in this cluster. We simulate a constant load and analyze the response time of different user operations. Three different scheduling policies are tested, causing services to be deployed on different nodes and experiencing different network latencies. We show that our framework can use the kube-scheduler in a way that it behaves the same as in the real cluster and that the simulation can reflect the effects of different scheduling policies on the simulated test application.

Cluster Environment. We use a cluster with five workers and one master in the Google Cloud, as shown in Fig. 3a. All machines are of type e2-standard-4 with four CPU cores. Two workers (eu1 and eu2) and the master are deployed in Germany. The remaining workers run in Singapore, Brazil, and Iowa (USA). Every node has its compute zone as a Kubernetes node label. Before the experiment, we measure the network latency between all nodes using 100 ping packets. The network latencies are given to the simulation as means and standard deviations. We use Kubernetes YAML files to specify the five workers for the simulation. At the simulation start, this information is forwarded to the kube-scheduler, which extracts available resource capacities, node labels, and more. The master node is not considered in the simulation.



(a) Cluster nodes and location (b) Workload dependency graph

Fig. 3. Modeled cluster environment and test application workload.

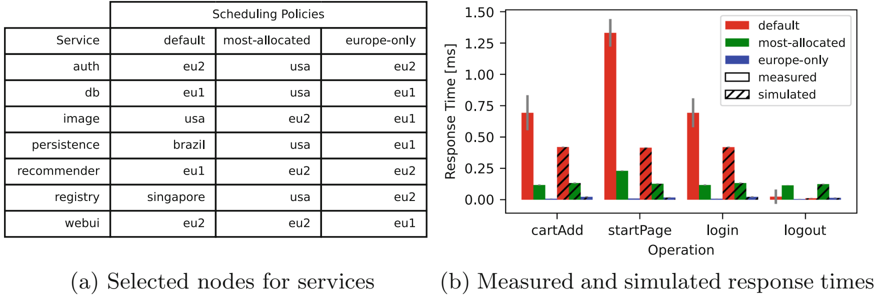
Microservice Test Application and Workload. We use the popular benchmarking application TeaStore [23] with seven microservices in our experiments. All microservices request 0.8 CPU cores per instance. In the real cluster, we use the HTTP load generator [4] to generate a constant load of 20 requests per second for five minutes. A user behavior consisting of four steps is simulated. First, a user accesses the start page and then logs in with his account data. Afterward, the user adds a random product from the store to his cart and then logs out again. In the simulation, we build on the architecture model of the TeaStore used by Frank et al. [17] and extend it with the database service. The defined workload stimulates a total of five services and 15 endpoints. The dependency graph of the operations for our workload is shown in Fig. 3b.

Scheduling Policies. We deploy one instance of each TeaStore service in our cluster. Hence, the total CPU core requirement is 5.6 cores, meaning that at least two nodes are needed for the deployment. We evaluate three scheduling policies: *default*, *most-allocated*, and *europe-only*. The default scheduling policy deploys the pod to the node where the least resources have been reserved so far. This results in the services being distributed to all nodes. Kubernetes offers two general options to influence scheduling. First, different configurations of the kube-scheduler can be used, resulting in a different algorithm being used for all pods. The most-allocated policy is an example of this. We use a profile of the kube-scheduler [28], which causes the next pod to be deployed to the node with the most resources reserved but still enough resources to run the pod. This policy requires two nodes to run the services. The scheduling policy is set at the start of the kube-scheduler. The second option to influence the scheduling in Kubernetes is to set special properties of the pods. These will influence the scheduling only for selected pods. For our europe-only policy, we use node affinities. Precisely, we specify that our services can only be deployed on nodes with a label indicating that they belong to the compute zone Europe. This causes the services to be distributed to the nodes eu1 and eu2. Both in the simulation and the real cluster, we use Kubernetes deployment files to specify the affinities.

Start State. Before deploying the services, all nodes have no reserved resources. Therefore, the nodes are equally suitable for selection. Furthermore, the order in which the containers are to be deployed plays a role in the scheduling since the kube-scheduler processes the pods one after the other. In initial tests, both factors led to different placements occurring during repeated simulation and real cluster runs. We made two adjustments to ensure the experiment's comparability and repeatability. First, we fixed the order in which the containers should be deployed to the recommended deployment order of the TeaStore [5]. Furthermore, we deployed other containers on four of the five worker nodes that occupy different resources at the nodes: Singapore (0.05 cores), Brazil (0.1 cores), eu2 (0.15 cores), and USA (0.2 cores). This setup leads to the scheduling decisions not being random and the experiment being repeatable.

Results. Fig. 4a gives an overview of the scheduling for different policies. These scheduling decisions are the same in the real cluster and the simulation. This shows that our approach can correctly represent different scheduling policies in the simulation without implementing them. The second question to be answered is whether the effects of different scheduling policies on the performance of the modeled microservice application can be simulated. Figure 4b shows the simulated and measured response times with the standard deviation for all four considered request types. We see that the scheduling policies significantly impact the response times. The default policy that distributes the services worldwide has the highest response times, while the europe-only policy has the lowest. This is visible in both the measured and simulated results. Furthermore, using the logout operation as an example, we see an interaction between application architecture and scheduling. As shown in Fig. 3b, the logout operation depends only on one operation of the auth service. According to Fig. 4a, the

default policy causes webui and auth to be deployed in Europe. In contrast, in the most-allocated policy, the auth service is deployed in the USA, and the webui service is in Europe. This explains the higher response time for this operation in the most-allocated policy compared to the default policy.



	Scheduling Policies		
Service	default	most-allocated	europe-only
auth	eu2	usa	eu2
db	eu1	usa	eu1
image	usa	eu2	eu1
persistence	brazil	usa	eu1
recommender	eu1	eu2	eu2
registry	singapore	usa	eu2
webui	eu2	eu2	eu1

(a) Selected nodes for services (b) Measured and simulated response times

Fig. 4. Effects of different scheduling policies in a global cluster.

Overall, the effects of different scheduling policies on microservice performance can be qualitatively modeled. However, some operations also have significant errors in the response time prediction. These are caused by inaccuracies in the MiSim performance model. For example, the large deviation for the startPage operation in the default policy is caused by large payloads with variable sizes (here: images) sent over the network. MiSim lacks support for modeling different payload sizes and parametric dependencies. Hence, the real network overhead cannot be predicted accurately in this case. However, this is a limitation of MiSim's performance model rather than a weakness of our approach for including orchestration mechanisms, as further discussed in Sect. 6. We also analyzed the overhead of our adapter and the integration in total. In this experiment, we measure only a tiny overhead. The simulation runs take, on average, about 9.2 s, of which only about ten milliseconds are spent on the communication with the adapter and kube-scheduler. This shows that our approach has a reasonable overhead and can enrich MiSim with authentic Kubernetes container orchestration mechanisms.

5.2 Expansion Policies for Cluster Autoscaling

Overview. In this section, we demonstrate our simulation with the CA and show its ability to capture the interactions between the cluster-autoscaler and the kube-scheduler (see Fig. 1). We model a heterogeneous cluster with two different node groups. The CA offers the possibility to define *expansion policies*, i.e., to configure which node groups should be prioritized during upscaling. We compare two expansion policies and prove they work as expected in our framework.

Simulated Cluster Environment. We define a cluster with two machine sets. Machines in the *small-set* have 4 CPU cores and in the *large-set* 8 cores. Both

machine sets can be scaled from one to ten nodes. One node from each machine set is deployed at the start. The analogy in a real cluster would be to deploy different instance groups with different resources and pricing models. Unfortunately, we cannot directly compare this experiment to a Google Kubernetes cluster where a proprietary, not freely configurable version of the CA is used.

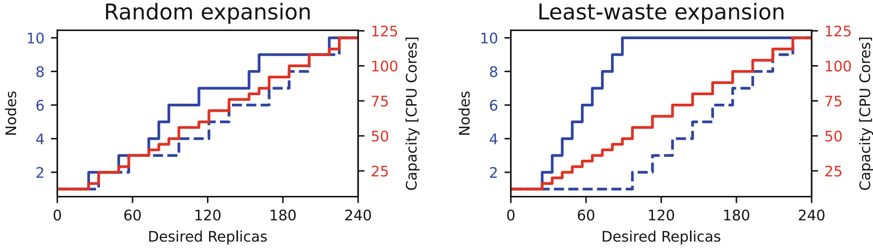


Fig. 5. Cluster scaling with nodes from small-set (solid) and large-set (dashed).

Expansion Policies and Test Application. The expansion policies of the CA can be adjusted via a command line flag. The CA triggers upscaling whenever the kube-scheduler cannot assign a pod to a node due to a lack of resources. We consider two expansion policies. The *random* policy randomly selects a node group for upscaling. In contrast, the *least-waste* policy prefers the node group, where as few resources as possible (here: CPU cores) are wasted after the deployment of the new pods. In this case, the machine set with four cores is always preferred. We use a Kubernetes deployment file of the TeaStore registry service as a test application, with each replica requesting 0.5 cores. For demonstration purposes, we use an autoscaler that requests a new instance every 15 s.

Results. Fig. 5 shows the two expansion policies in comparison. Since the random policy returns different results with repeated runs, Fig. 5 shows only a selected test run. Both expansion policies behave as expected: The random expander deploys alternating instances with 4 and 8 CPU cores on average (both blue lines increase early). The least-waste expander first scales the machine set with 4 CPU cores up to the maximum number of 10 (only the solid blue line rises first). This leads to a slow increase in the number of provided CPU cores (red line) and many upscaling decisions being necessary initially. The random policy makes capacity increase more stable, making it a policy that can be used when nothing is known about workload changes. The least-waste policy is a cost-optimized policy that can be used when slowly increasing loads and few upscaling are expected. Both policies stop the expansion when the maximum size of the machine sets is reached. Overall, we show that different CA configurations can be correctly executed in our simulation. The basis for this is the interaction of the CA with the kube-scheduler, as shown in Fig. 1. Hence, we show that multiple Kubernetes components can be used in parallel in our framework.

6 Discussion

This section summarizes our approach's strengths, weaknesses, and limitations. As already mentioned, there are different user groups of our framework. We enable designers of microservice architectures to simulate realistic scenarios with authentic container orchestration. We provide a new lightweight test platform for Kubernetes or container orchestration developers, where real code and new configurations can be tested without requiring a cluster. A significant advantage is that orchestration mechanisms can be included in the simulation in their full complexity without having to model them. The drawback is that expert knowledge of the orchestrator interfaces is required to integrate its code. This could be an obstacle for microservice designers, less for developers of CO frameworks.

To include new orchestration mechanisms in our framework, the interfaces required by the mechanism must be identified and implemented in the adapter. In a concrete use case, the question of whether integration in this form is worthwhile or whether a simple model of the mechanism is sufficient must be answered. The integration is worthwhile if the mechanism has a complex configuration and logic. If it is a relatively simple algorithm without regular updates, a simple model should be preferred. Generally, we assume that the integrated components are independently deployable and work event-based. A current limitation of our approach is that component responses are always executed immediately in the simulation. While this is a good approximation for the kube-scheduler, cluster autoscaling includes non-negligible node start times. Extensions in our framework for these purposes are planned. Since we do not change the performance models in MiSim, the challenge for accurate predictions for response times and similar metrics remains the correct calibration of the performance models as well as a correct modeling of parametric dependencies [15]. However, these weaknesses concern the MiSim core rather than the extension presented in this paper, which in theory, would also support other performance models.

7 Conclusion

This paper presented a new approach for connecting a microservice simulation with Kubernetes components based on a general concept for combining discrete-event simulation and event-driven systems. We validated our approach in experiments with the kube-scheduler and cluster-autoscaler. We have shown that different component configurations can be represented in the simulation by using the actual Kubernetes components and without abstract models or manual re-implementation. Our framework can be used, for example, by microservice application developers to simulate real-world scenarios and by CO framework developers to test new orchestration policies. Future work will focus on evaluating complex scenarios, simulation-based optimization of CO mechanisms, and increasing the simulation accuracy, e.g., by considering startup times.

Acknowledgements. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 510552229. We thank the German Federal Min-

istry of Education and Research (dqualizer FKZ: 01IS22007B and Software Campus 2.0 — Microproject: DiSpel, FKZ: 01IS17051), the Google Cloud Research Credits program (GCP213506809) and our student researchers Lion Wagner and Lukas Mönch for supporting this work. The work was conducted in the context of the SPEC RG DevOps Performance Working Group [6].

References

1. https://github.com/DescartesResearch/misim-orchestration
2. https://github.com/DescartesResearch/misim-k8s-adapter
3. https://doi.org/10.24433/CO.4913288.v1
4. https://github.com/joakimkistowski/HTTP-Load-Generator
5. https://github.com/DescartesResearch/TeaStore
6. https://research.spec.org/devopswg
7. Aslanpour, M.S., Toosi, A.N., Taheri, J., Gaire, R.: Autoscalesim: A simulation toolkit for auto-scaling web applications in clouds. Simul. Modelling Pract. Theory (2021)
8. Bao, L., Wu, C., Bu, X., Ren, N., Shen, M.: Performance modeling and workflow scheduling of microservice-based applications in clouds. IEEE Trans. Parallel Distrib. Syst. (2019)
9. Becker, M., Becker, S., Meyer, J.: Simulizar: design-time modeling and performance analysis of self-adaptive systems. Softw. Eng. (2013)
10. Ben Ayed, M., Zouari, L., Abid, M.: Software in the loop simulation for robot manipulators. Eng., Technol. Appl. Sci. Res. (2017)
11. Bozóki, S., Szalontai, J., Pethő, D., Kocsis, I., Pataricza, A., Suskovics, P., Kovács, B.: Application of extreme value analysis for characterizing the execution time of resilience supporting mechanisms in Kubernetes. In: Dependable Computing—EDCC 2020 Workshops (2020)
12. Carrión, C.: Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges. ACM Comput, Surv. (2022)
13. Cloud Native Computing Foundation: Cncf annual survey 2022 (2023). https://www.cncf.io/reports/cncf-annual-report-2022/
14. Demers, S., Gopalakrishnan, P., Kant, L.: A generic solution to software-in-the-loop. In: MILCOM 2007—IEEE Military Communications Conference (2007)
15. Eismann, S., Walter, J., von Kistowski, J., Kounev, S.: Modeling of parametric dependencies for performance prediction of component-based software systems at run-time. In: IEEE International Conference on Software Architecture (ICSA) (2018)
16. Erb, B., Kargl, F.: Combining discrete event simulations and event sourcing. In: Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques. SIMUTools **14** (2014)
17. Frank, S., Wagner, L., Hakamian, A., Straesser, M., van Hoorn, A.: Misim: a simulator for resilience assessment of microservice-based architectures. In: IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS) (2022)
18. Friedman, N.: Kubernetes cluster autoscaler command line options (2023). https://gist.github.com/neerfri/4bd7477920cb33a2a229807ed10c29c2
19. Hellerer, M., Schuster, M.J., Lichtenheldt, R.: Software-in-the-loop simulation of a planetary rover. In: The International Symposium on Artificial Intelligence, Robotics and Automation in Space (2016)

20. IBM Market Development & Insights: Microservices in the enterprise (2021). https://www.ibm.com/downloads/cas/OQG4AJAM
21. Jawarneh, I.M.A., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R., Palopoli, A.: Container orchestration engines: a thorough functional and performance comparison. In: IEEE International Conference on Communications (ICC) (2019)
22. Jindal, A., Podolskiy, V., Gerndt, M.: Performance modeling for cloud microservice applications. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering (2019)
23. von Kistowski, J., Eismann, S., Schmitt, N., Bauer, A., Grohmann, J., Kounev, S.: Teastore: a micro-service reference application for benchmarking, modeling and resource management research. In: IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS) (2018)
24. Kubernetes Documentation: Assign pods to nodes using node affinity (2023). https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity/
25. Kubernetes Documentation: Kubernetes API concepts (2023). https://kubernetes.io/docs/reference/using-api/api-concepts/
26. Kubernetes Documentation: The kubernetes cluster autoscaler (2023). https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/README.md
27. Kubernetes Documentation: Kubernetes components (2023). https://kubernetes.io/docs/concepts/overview/components/
28. Kubernetes Documentation: Scheduling framework (2023). https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/
29. Kubernetes SIG Cluster API: Kubernetes cluster API (2023). https://cluster-api.sigs.k8s.io/
30. Kubernetes SIG Scheduling: Kube-scheduler-simulator (2023). https://github.com/kubernetes-sigs/kube-scheduler-simulator
31. Law, A.M., Kelton, W.D.: Simulation modeling and analysis, vol. 3. Mcgraw-hill New York (2007)
32. Lechler, T., Page, B.: Desmo-j: An object oriented discrete simulation framework in java. In: Proceedings of the 11th European Simulation Symposium (ESS) (1999)
33. von Massow, R., van Hoorn, A., Hasselbring, W.: Performance simulation of runtime reconfigurable component-based software architectures. In: 5th European Conference on Software Architecture (ECSA) (2011)
34. Matarneh, R.J.: Self-adjustment time quantum in round robin algorithm depending on burst time of the now running processes. Am. J. Appl. Sci. (2009)
35. Mekki, M., Toumi, N., Ksentini, A.: Microservices configurations and the impact on the performance in cloud native environments. In: IEEE 47th Conference on Local Computer Networks (LCN) (2022)
36. Michelson, B.M.: Event-driven architecture overview. Patricia Seybold Group (2006)
37. Pan, Y., Chen, I., Brasileiro, F., Jayaputera, G., Sinnott, R.: A performance comparison of cloud-based container orchestration tools. In: IEEE International Conference on Big Knowledge (ICBK) (2019)
38. Schmoll, R., Fischer, T., Salah, H., Fitzek, F.H.P.: Comparing and evaluating application-specific boot times of virtualized instances. In: 2nd IEEE 5G World Forum (2019)

39. Straesser, M., Bauer, A., Leppich, R., Herbst, N., Chard, K., Foster, I., Kounev, S.: An empirical study of container image configurations and their impact on start times. In: 23rd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid) (2023)
40. Straesser, M., Grohmann, J., von Kistowski, J., Eismann, S., Bauer, A., Kounev, S.: Why is it not solved yet? challenges for production-ready autoscaling. In: Proceedings of the ACM/SPEC on International Conference on Performance Engineering (2022)
41. Straesser, M., Mathiasch, J., Bauer, A., Kounev, S.: A systematic approach for benchmarking of container orchestration frameworks. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering (2023)
42. Tamiru, M.A., Tordsson, J., Elmroth, E., Pierre, G.: An experimental evaluation of the Kubernetes cluster autoscaler in the cloud. In: IEEE International Conference on Cloud Computing Technology and Science (CloudCom) (2020)
43. Tesfatsion, S.K., Klein, C., Tordsson, J.: Virtualization techniques compared: Performance, resource, and power usage overheads in clouds. In: ACM/SPEC International Conference on Performance Engineering (2018)
44. Truyen, E., Bruzek, M., Van Landuyt, D., Lagaisse, B., Joosen, W.: Evaluation of container orchestration systems for deploying and managing nosql database clusters. In: IEEE 11th International Conference on Cloud Computing (CLOUD) (2018)
45. Truyen, E., Van Landuyt, D., Lagaisse, B., Joosen, W.: Performance overhead of container orchestration frameworks for management of multi-tenant database deployments. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. SAC **19** (2019)
46. Valera, H.H.A., Dalmau, M., Roose, P., Larracoechea, J., Herzog, C.: Draceo: a smart simulator to deploy energy saving methods in microservices based networks. In: IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE) (2020)
47. VMware: K-bench (2023). https://github.com/vmware-tanzu/k-bench
48. Wang, M., Zhang, D., Wu, B.: A cluster autoscaler based on multiple node types in Kubernetes. In: IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC) (2020)
49. Xavier, B., Ferreto, T., Jersak, L.: Time provisioning evaluation of KVM, Docker and Unikernels in a cloud platform. In: 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (2016)
50. Zhang, Y., Gan, Y., Delimitrou, C.: μqsim: enabling accurate and scalable simulation for interactive microservices. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (2019)