Container-Based Simulation: A Concept For Large-Scale Simulation Environments

Daniel Seufferth[1*], Falk Stefan Pappert[1], Heiderose Stein[1], Oliver Rose[1]

[1]Institute of applied computer sciences , University of the Bundeswehr Munich, Werner-Heisenberg-Weg 39, 85577 Neubiberg, Germany; [*]*daniel.seufferth@unibw.de*

Abstract. When simulation experiments grow in scale, a simulation environment providing appropriate computational resources is needed to produce results within a reasonable time. Containerization is a promising method for creating such a scalable environment. This method needs to be adapted to the simulation users' demands and should be easy to use for a project. Therefore, we provide a concept for a container-based environment, supporting various simulation projects, that dynamically scales and adapts the simulation workload to the available hardware. The concept encapsulates the environment technologies and provides an access service to the user for defining and editing simulation experiments and retrieving results. We discuss requirements for combining simulation and containerization to support the transition to such a container-based simulation environment. As a result, we see an opportunity to enhance large-scale simulation experiments using container methods and identify areas for further research.

Introduction

There are several fields of study in and adjacent to the broader simulation domain with a significant rise in demand for computational power. Prominent methods are Simulation Based Optimization (SBO), Machine Learning (ML), and data-farming.

SBO combines simulation models with optimization algorithms to find optimal or near-optimal solutions for problems [1]. Various approaches utilize SBO to assist in decision-making and finding ways to optimize real-world systems (e.g., see Lidberg et al. [2], Nikolopoulou and Ierapetritou [3], or Nguyen et al. [4]). As most optimization algorithms require multiple iterations over a system, increasing the number of simultaneously running versions of the model speeds up the optimization process significantly.

The importance of ML as a research field was shown in a recent study by Nature, which asked 1600 researchers about how ML and Artificial Intelligence (AI) in general will change in their future research. Among the main benefits are improvements to data processing and acquisition, and increased productivity [5]. Rai et al. [6] comprehensively review the importance and recent advances of ML for Industry 4.0 applications in manufacturing and production systems. A substantial amount of training data is necessary to create comprehensive ML-tools.

At the same time, the increasing complexity of simulation models and the number of simulation experiments challenge the simulation environment: As computational resources are constrained, the workload required for large-scale and complex simulations limits the feasibility of performing these experiments.

Efforts to speed up simulation experiments have been an active field of research since the 1970s [7]. The ongoing trend of cloud computing, virtualization, and containerization offers an exciting opportunity for creating a scalable simulation environment. The research in this field is diverse and growing. Król et al. [8] developed Scalarm, an infrastructure for distributing simulation workloads onto multiple computing nodes in heterogeneous environments. In more recent work, Anagnostou et al. [9] evaluated technological approaches with their work on simulation experimentation frameworks, applying a microservice-based auto-scaling approach utilizing MiCADO. MiCADO focuses on efficiently utilizing cloud resources. It extends Kubernetes' Application Programming Interface (API) objects with its Application Description Templates (see [10]), making it less universal.

Although these technologies have been around for some years, a significant uptick in adopting these methods is not yet visible in the simulation community, neither from the practitioner nor the software developer side. We attribute this to the perceived steep learning curve of using containers and container management in general. Especially packaging simulation workloads in containers and their convenient use is still an open issue. To approach this, we propose a container-based sim-

ulation architecture that supports different experiment setups, e.g., SBO, data-farming, or machine-learning-based approaches. Our goal is to provide a foundation for a heterogeneous environment that can dynamically support multiple projects to scale and adapt the distribution of computing resources.

This paper is structured as follows: First, we describe our concept for a container-based simulation environment from a simulationist's point of view. We briefly introduce the key technologies we need for this concept: containers and container orchestration. The following sections provide an overview of the key components of our concept. The third section discusses the requirements arising from combining simulation with containers in the context of large-scale experiments. Lastly, we summarize our research findings and give future research opportunities.

1 Introduction to containerization and container orchestration.

Container technology is a type of operating system virtualization, isolating a part from the underlying host Operating System (OS) for the processes of the software packaged in the container. Unlike a Virtual Machine (VM), containers utilize the OS and access the hardware of the host computer directly. This makes containers lightweight and flexible, as only the dependencies to run the containerized software are necessary. Thus, a container only brings its purpose-specific binaries and files and uses general libraries and binaries shared by the host OS. This is implemented by a software called container runtime, which enables containers to communicate with the host kernel and run processes, as stated by Hitchcock [11].

Containers are commonly created based on a container image. These are created using container engines, such as Docker, Podman, or Apptainer engines. All container engines follow a similar approach to image creation, utilizing a descriptive file that defines all steps necessary to containerize a software package. As a generalization, we call this file the containerization file in the following. Typically, a containerization file defines the base image and the more specific parts of the container built atop.

Containers are highly portable software packages, able to run on various hardware configurations, from multiple office Personal Computers (PCs) to full-fledged servers, as long as the underlying OS supports the specific container. Container orchestrators are used to manage large amounts of containers. Therefore, they are vital components for highly scalable simulation environments, where they handle large numbers of simultaneously running simulator instances.

There are several different orchestration software packages on the market, but one of the most prominent ones is Kubernetes, with which Google launches about 4 billion containers per week [12]. More information on containers and Kubernetes can be found in Huawei Technologies [13] and Poulton and Joglekar [14].

2 Container-based simulation environment

Modern software systems that need to be highly scalable are typically built using a service-based architecture, which is comprised of individually running services that offer functionality to both other services and the user. An approach to designing and scaling up these software systems is containerization, where each service runs within containers. Therefore, the number of container instances providing a service can be increased or decreased using container orchestration based on the service's demand and the system's current load. This approach efficiently addresses and uses large amounts of heterogeneous hardware and maximizes the utilization of existing hardware. Hence, enabling the dynamical adjustment of available computing resources to different simulation projects, even when these projects employ different simulation software packages.

Figure 1 provides an overview of the functional concept of the proposed container-based simulation environment and its main components. The *Design of Experiment Service* represents the core user strategy on what simulation settings need to be run; this can be an optimizer, a reinforcement learning agent, or another generator for the design of experiments. It generates scenarios and commissions the system with their evaluation. The *Scenario Manager* is responsible for managing the scenarios once they are in the system, scheduling and monitoring their execution. Within our architecture, we consider a scenario to be a complete data set representing a single instance of the modelled system. We recognize three types of scenario data depending on the viewpoint, with only the second one needed for a complete system understanding.

1. Abstract meta-data, stored in the *Scenario Status*

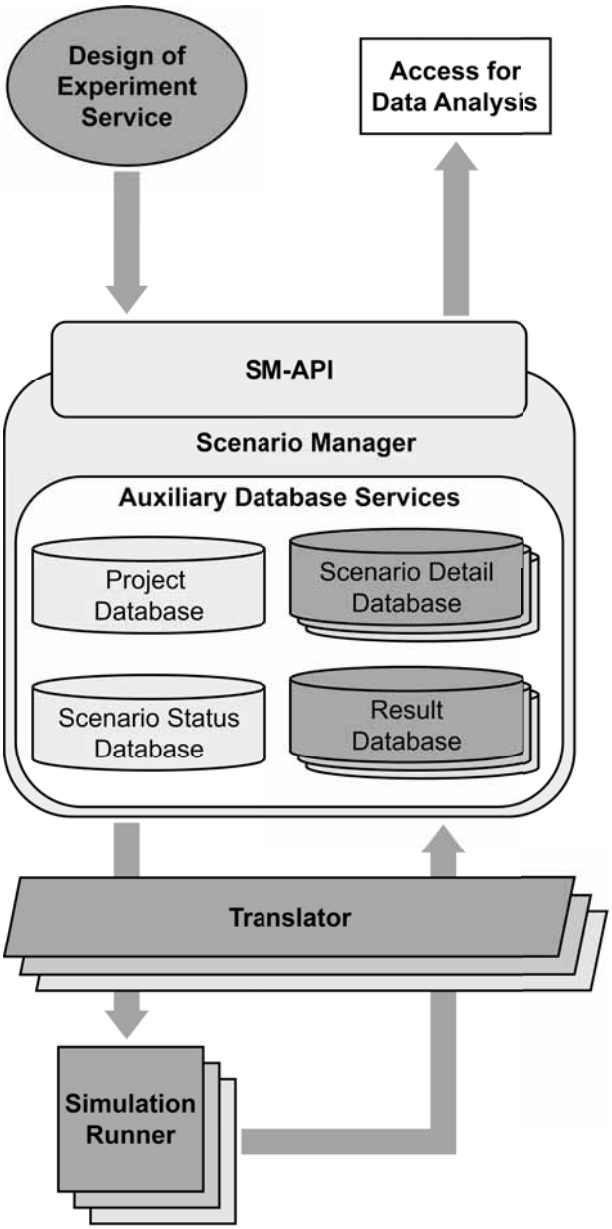

Figure 1: Concept of a container-based simulation environment, consisting of four main components: *Design of Experiment Service* (DoES), *Scenario Manager* (SM), *Translator* and *Simulation Runner* (SR).

Database, like a global scenario ID or the processing state of simulations desired for the scenario.

2. System description, stored in the *Scenario Detail Database*, provides all system information to create a valid model for the purpose.

3. Result data, stored in the *Result Database*, keeps records of all Key Performance Indicators (KPIs) and logs information desired from a scenario's simulation runs.

The *Translator* generates simulation-software-specific instances of a scenario and followingly makes the simulation executable by the simulator. These executable instances are run by the *Simulation Runners*, which report results to the *Scenario Manager* after execution.

Each of these components is implemented as a service within its container. This allows the available computing resources to be adjusted depending on the demand for these components and the system's general load. This setup, especially the *Simulation Runners*, enables the user to benefit from dynamically addressing large amounts of computing resources. Therefore, it allows for the adaptive provisioning of resources for different simulation projects using different software packages.

From our point of view, there are two general ways to implement our concept: focusing on project-specific components or aiming at reaching a universal system. In the universal approach, the semantic transformation from a project-specific to a universal model is done by the *Design of Experiment Service* (DoES). All scenarios handled and evaluated can, therefore, be stored in the same structure, allowing the *Scenario Database*, *Result Database*, and *Translator* to be universal as well. The downside of this approach is the need to transform and store scenarios, and therefore their model descriptions, in a generic format. This makes the creation of these three universal components very complex.

The more pragmatic approach is to design these components in a project-specific way, as indicated by the colouring scheme in Figure 1. This is, in our view, the more worthwhile option. However, this requires individually building database schemes and *Translators* for each project. The complexity of these components is significantly smaller, as a scenario entry only needs to hold a few key parameters, and the *Translator* can be specialized for the project. A further benefit of the project-specific approach is the improved possibilities for fine-tuning and adapting the model structure to the project-related domain.

2.1 Design of Experiment Service

As we mentioned earlier, the DoESs represent the scenario-creating elements of a simulation-based system: for example, the optimizer creating solution can-

didates and sending them off to be evaluated or the data-farming service implementing a given design of experiments. Therefore, the DoES represents the interface to the simulation environment and is necessary to enter simulation workloads. Via this interface, the setup of all scenarios is defined, which comprises the data required for generating the models, necessary computing restrictions, and other requirements.

2.2 Scenario Manager

The *Scenario Manager* (SM) is the management component of our concept and has various functions. Based on the project-specific simulation data provided by the DoES, it manages scenarios and coordinates their execution on the *Simulation Runner*s (SRs) using the *Translators* for model generation. Furthermore, it actively monitors the SRs, ensuring comprehensive oversight by consistently tracking the status of each scenario. Lastly, the SM also organizes the simulation results in databases and provides access to analyze the generated data.

The SM and its databases are implemented as containers. While the *Project Database* and *Scenario Status Databases* are universal, the *Scenario Detail Database* and *Scenario Result Databases* are project-specific deployments. The particular configuration of each project-specific database deployment depends on the scale of the experiment. Therefore, the configuration can range from one database container to complex deployments consisting of multiple database containers and their infrastructure, e.g., load balancers.

2.3 Translator

The *Translators* generate simulator-specific simulation models based on the scenarios passed to it by the SM. The data needed is pulled from the *Scenario Detail Database*. The specific implementation of the *Translator* highly depends on the respective project approach and the capabilities of the chosen modelling tool. A simple *Translator* would be used to forward parameters from the *Scenario Detail Database* to the SRs that already have a base model implemented. More sophisticated *Translators* may use the data from the *Scenario Detail Database* to automatically generate executable simulation models. The main technologies used to implement *Translators* come from the fields of model transformation and automated model generation. We refer to published research for the specific implementation of *Translators*. Thiers et al. [15] comprehensively introduce this research subject. They furthermore discuss the large variety of system description languages and propose a methodology for one system description language combined with a transformation step that loosely couples other languages to the back-end bridging abstraction language.

2.4 Simulation Runner

The SRs provide simulation execution as a service. These containers encapsulate the simulation engine and, when started, run the actual simulations. Different containers providing different simulation engines can be deployed, matching the projects run. After the scenarios are generated by the *Translator*, they get forwarded to the SR with the specified simulation engine required by the design.

The number of SRs is highly dynamic and is automatically scaled up and down based on the available computing resources. Figure 2 illustrates this breathing characteristic of the SRs. During idle time, only the template of a SR exists (1). When the *Translator* generates models that require an idle SR, the deployment increases the number of SRs to a size that fits the available hardware and the number of simultaneous instances defined for this project, visualized in (2) and (3). If demand decreases, the number of SRs decreases (4) and goes back to zero if this specific simulation engine is no longer needed.

3 Requirements to use containerization for simulation

To make simulation environments based on containers practicable for a user, some requirements must be met for simulation packages. Although containerization and running software as services from inside containers are generally not new concepts, they are rather unexplored in the simulation community. Therefore, it is necessary to illustrate how the specific use case of simulation is affected by containerization. We found that containerization, in combination with simulation, sets specific requirements that can be grouped into four general fields:

- Modeling
- Containerization

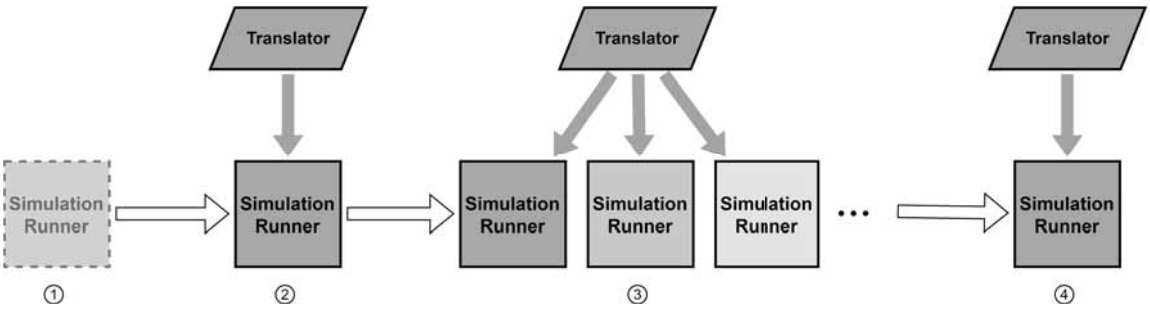

Figure 2: The *Simulation Runners* automatically scales up and down, based on the demand created by the scheduled scenarios of the *Scenario Manager*.

- Integration
- Licensing

We amend these requirements from [16] with the findings from our concept in the following.

3.1 Modeling requirements

The key modelling requirement on a simulation package for usage in large-scale systems is the model generation capability by external means. We found that simulation packages allow the automation of model building to different degrees, which we categorize into five levels:

1. No automation
2. Parameterization of a hand-made model
3. Bootstrapping models based on external data
4. External generation of model files
5. Online model generation using an API

In the proposed architecture for a container-based simulation environment, the *Translator* component handles automated model generation. To this end, some degree of external model generation needs to be supported by the simulation tool. Depending on the project, higher levels of model generation may be required, restricting the choice of simulators.

3.2 Containerization Requirements

The containerization requirements focus on what is needed to use simulation software efficiently in container-based, scalable environments. They are categorized into three topics: needs imposed by the OS, constraints due to the ephemeral nature of containerized applications, and requirements of the containerization process.

3.2.1 Operating System

An important requirement for the container-based simulation environment is providing the OS needed for the simulation software. As we mentioned in Section 1, containers are a form of OS virtualization, requiring a host OS that supports the container. The majority of currently used containers are Linux-based, requiring a host that runs Debian, RedHat, Ubuntu, or a similar OS. Consequently, most of the current container ecosystem, i.e., the applications and tools used for running containers in a scalable environment, is tuned for Linux containers.

In contrast, as shown in Table 1, a significant amount of software currently used by the simulation community for modelling and simulation is based on Windows. For simulation software that solely runs on Windows, Windows containers are required. Running Windows containers in orchestrator systems leads to hybrid environments, consisting of both Linux and Windows hosts, as the management layers of container orchestrators like Kubernetes require Linux hosts. Such hybrid environments can be considered less efficient than their Linux-only counterparts, as resources for both host systems need to be provided and managed. This results in worse utilization of the available hardware or increased efforts due to the need to dynamically adjust host system allocation depending on the observed demand. An ideal situation would be the availability of Linux options for all simulation software packages; as this is not the case, we see at least larger systems working on a hybrid basis for the foreseeable future.

Simulation tool	Supports Windows?	Supports Linux?	Supports macOS?
ANSYS	yes	yes	no
AnyLogic	yes	yes	yes
FlexSim	yes	no	no
MATLAB/Simulink	yes	yes	yes
OpenModelica	yes	yes	yes
Simio	yes	no	no

Table 1: List of simulation tools and the support of Linux-based OSs.

3.2.2 Ephemeral simulation containers

Containers may be killed at any time during their lifecycle. This can be caused by errors in their internal software or changes in maintenance or load control. This characteristic is called ephemeral. A system that depends on containers to provide functionality, therefore, needs to address this properly. The following are ideas to approach this problem:

Restarting failed simulation containers intuitively emerges as the simplest method to address this issue. Most container orchestrators are declarative, which means they try to achieve the desired state defined in the files posted to the API server of the orchestrator. Controllers continuously check the observed state of the cluster against the desired state and perform the tasks necessary to achieve the desired state if deviation is detected. To effectively utilize this feature of container orchestrators for large-scale simulation experiments, a controller that keeps track of the failed scenarios is essential. This functionality would be covered by the *Scenario Manager* in the previously proposed architecture.

If a simulation container is ended prematurely, the state of the model is lost, and the allocated resources for this process are wasted. Extending the restart of the containers by **keeping track and storing the progress of the simulation run** is a comprehensible next step. For this, continuous updates of the current simulation state, or at the least regular snapshots, must be kept in storage. This causes a significant amount of data to be transferred and stored. These resource-intensive tasks likely harm simulation speed: the theoretical performance gained through restarting the simulation container from a cached image of the last state will most likely be compromised by the processes necessary to create these images. From the authors' point of view, this approach is only feasible in an environment where successfully finishing simulation runs is otherwise unlikely, e.g., due to extensive simulation execution time or unstable computing hardware.

Most simulation experiments utilize replications to achieve sufficient confidence intervals. Therefore, **increasing the number of replications for each scenario** would be an efficient way to add redundancy and counteract the loss of simulation runs. Depending on stability, this method would not create a significant margin of load to the cluster, ensuring efficient utilization of the computing hardware available. However, increasing the number of replications for the scenarios does not ensure the successful execution of any scenario, as even a large number of replications could fail. Suppose the probability of failure of the container/model can be estimated. In that case, the required number of replicas needed to achieve a certain coverage can easily be calculated within the desired confidence.

3.2.3 Container creation

The last containerization requirement addresses constraints for containerizing simulation software.

To containerize a simulation model, it is necessary to either have an existing base image of your simulator or a base image that supports your simulator of choice. The latter option requires manually adding the simulation engine to the base image, which needs additional effort. This would usually be done with the help of a package manager or an installation method that does not require user interaction. If the simulation tool does not support these methods, a more tedious way of packaging the simulation engine must be utilized, e.g., emulating user inputs in the containerization file.

An optimized containerization process for simulation models would mean that simulation package developers and vendors provide usable base images of their

simulation software. With functioning base images, the SRs can easily be populated by the generated models from the *Translators*. Although some simulators already provide base images, e.g., MATLAB, and some container images for simulators created by the community, e.g. for SimPy, this is not the norm.

3.3 Integration requirements

The integration requirements describe how well a simulation package can be integrated into a container-based software system, e.g., how it can be started and executed and how results are managed. Containerized software usually runs headless, meaning that a Graphical User Interface (GUI) is typically unavailable during runtime. Therefore, in a containerized setup, it is strongly preferred to have a simulator that supports execution without a GUI. Suppose the simulation software is dependent on interactions with a graphical user interface. In that case, it is not well suited for a scaled-up environment as the manual interaction also needs to be scaled accordingly, which is typically not feasible.

Besides running, starting the simulation is also a concern for integration. Although an essential and everywhere available option in past days, starting the simulation run from the command line is no longer generally available. Similar to manual interactions with the model during runtime, using a GUI is also unfeasible for environments where many simulations are started in parallel. Although there are makeshift approaches, including the emulation of inputs from keyboard and mouse, that help automate these interactions, their implementation is typically less than ideal.

Most simulation execution environments used in simulation packages are not (yet) designed for distributed infrastructures. An experiment manager and an external way of triggering a simulation run are necessary for such systems. The most common way to do this is by using command-line interfaces. Starting simulations using an API is an even more convenient option.

Other important considerations are connectivity to databases and the possibilities to write results. If this is not possible, other options are required to parameterize the simulation model and gain access to results. At least the results must be exportable to an external file, which can then be evaluated and transferred by additional software in the container. This, of course, makes the container much more complex to create, but especially for older simulation packages, it may be the only available option.

3.4 Licensing Requirements

The last requirement for combining container technologies and simulation we see is the licensing of simulation software. Software vendors have different approaches to licensing their products, which may impact the cost structure of using this software in a containerized, highly scalable environment. Simulation packages, where costs only depend on the model development environment, allow very flexible scaling of the simulation to new or different projects. Licensing on a per-core/seat/user basis is also suitable for a containerized computing infrastructure but can get quite costly if sufficient hardware is available. Moreover, the licensing agreement should be thoroughly reviewed to determine whether the use in container environments is allowed.

Besides cost and permissibility, another noteworthy side of licensing is its technical enforcement. For a dynamic environment, concurrent licensing via licensing servers is ideal. In contrast, licensing schemes enforced by hardware restrictions put significant limitations on a containerized simulation platform. Examples of this approach would be licensing codes tailored to a specific computer or hardware keys provided as USB dongles.

4 Conclusion and further research

This paper proposes a high-level concept for a container-based simulation environment primed to meet the growing demand for large-scale simulation experiments. We introduced the four key components of our concept, consisting of the *Design of Experiment Service* (DoES) as a central service that allows access to the environment and defines all specifics of an experiment setup, the *Scenario Manager* (SM) as a management service for handling large numbers of scenarios, the *Translator* as a translation interface that generates simulation models from different description languages, and the *Simulation Runner*s (SRs) that provide containerized simulators which can be dynamically scaled up and down based on demand.

To support the transition to such a container-based simulation environment, we investigated what requirements arise from the combination of simulation and containerization. The described requirements cover various fields and must be considered when container-

izing simulation models. General modelling requirements describe the importance of populating the simulation containers with models.

The next step of our research will be to implement this concept, as we see a need to enhance large-scale simulation experiments and use container methods. This includes containerizing and testing different simulation software packages and assessing to what extent the simulation environment can support their dynamic management. Another future step is to evaluate the environment on different hardware setups. Other exciting areas of future research may look at different concepts, for example, a more universal DoES or strategies targeted to a specific hardware configuration.

Acknowledgement

We want to thank Uwe Langer and Alexandros Karagkasidis for their continuous support of the hardware infrastructure.

This research is funded by dtec.bw - Center for Digitization and Technology Research of the Bundeswehr. dtec.bw is funded by the European Union – NextGenerationEU.

References

[1] Fowler J, Sawah SE, Turan HH. Recent Advances in Simulation-Based Optimization for Operations Research Problems. *Annals of Operations Research*. 2023;320(2):545–546.

[2] Lidberg S, Aslam T, Pehrsson L, Ng AHC. Optimizing Real-World Factory Flows Using Aggregated Discrete Event Simulation Modelling. *Flexible Services and Manufacturing Journal*. 2020;32(4):888–912.

[3] Nikolopoulou A, Ierapetritou MG. Hybrid Simulation Based Optimization Approach for Supply Chain Management. *Computers & Chemical Engineering*. 2012;47:183–193.

[4] Nguyen AT, Reiter S, Rigo P. A Review on Simulation-Based Optimization Methods Applied to Building Performance Analysis. *Applied Energy*. 2014; 113:1043–1058.

[5] Van Noorden R, Perkel JM. AI and Science: What 1,600 Researchers Think. *Nature*. 2023; 621(7980):672–675.

[6] Rai R, Tiwari MK, Ivanov D, Dolgui A. Machine Learning in Manufacturing and Industry 4.0 Applications. *International Journal of Production Research*. 2021;59(16):4773–4778.

[7] Taylor SJ. Distributed Simulation: State-of-the-Art and Potential for Operational Research. *European Journal of Operational Research*. 2019;273(1):1–19.

[8] Król D, Wrzeszcz M, Kryza B, Dutka Ł, Kitowski J. Massively Scalable Platform for Data Farming Supporting Heterogeneous Infrastructure. In: *The 4th International Conference on Cloud Computing, Grids, and Virtualization: (Cloud Computing 2013) ; Valencia, Spain, 27 May - 1 June 2013 ; Held at ComputationWorld 2013*, edited by Zimmermann W, pp. 144–149. Wilmington: IARIA. 2013;.

[9] Anagnostou A, Taylor SJE, Abubakar NT, Kiss T, DesLauriers J, Gesmier G, Terstyanszky G, Kacsuk P, Kovacs J. Towards a Deadline-Based Simulation Experimentation Framework Using Micro-Services Auto-Scaling Approach. In: *2019 Winter Simulation Conference (WSC)*, edited by Navonil Mustafee, Ki-Hwan G Bae, Sanja Lazarova-Molnar, Markus Rabe, C Szabo, Peter Haas, and Young-Jun Son. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc. 2019; pp. 2749–2758.

[10] Project M. Application Description Template - MiCADO. https://micado-scale.github.io/adt/. 2023.

[11] Hitchcock K. Containers. In: *Linux System Administration for the 2020s*, pp. 155–200. Berkeley, CA: Apress. 2022;.

[12] Google. Why Choose GKE as Your Kubernetes Service. https://cloud.google.com/blog/products/containers-kubernetes/why-choose-gke-as-your-kubernetes-service. 2023.

[13] Huawei Technologies Co Ltd n. Container Technology. In: *Cloud Computing Technology*, pp. 295–342. Springer, Singapore. 2023;.

[14] Poulton N, Joglekar P. *The Kubernetes Book*. [United Kingdom]: Nigel Poulton, january 2022 ed. 2022.

[15] Thiers G, Sprock T, McGinnis L, Graunke A, Christian M. Automated Production System Simulations Using Commercial Off-the-Shelf Simulation Tools. In: *2016 Winter Simulation Conference (WSC)*. 2016; pp. 1036–1047.

[16] Seufferth D, Stein H, Pappert F, Rose O. On the Usage of Container and Container Orchestrators as a Computational Infrastructure for Simulation Experiments. *20 ASIM Fachtagung Simulation in Produktion und Logistik 2023*. 2023;pp. 393–401.