Hindawi
Complexity
Volume 2018, Article ID 8281079, 16 pages
https://doi.org/10.1155/2018/8281079

*Research Article*

# Orchestrated Platform for Cyber-Physical Systems

**Róbert Lovas, Attila Farkas, Attila Csaba Marosi, Sándor Ács, József Kovács, Ádám Szalóki, and Botond Kádár**

*Institute for Computer Science and Control, Hungarian Academy of Sciences (MTA SZTAKI), P.O. Box 63, Budapest 1518, Hungary*

Correspondence should be addressed to Róbert Lovas; lovas.robert@sztaki.mta.hu

Received 24 November 2017; Revised 21 March 2018; Accepted 19 April 2018; Published 5 July 2018

Academic Editor: Shun-Feng Su

One of the main driving forces in the era of cyber-physical systems (CPSs) is the introduction of massive sensor networks (or nowadays various Internet of things solutions as well) into manufacturing processes, connected cars, precision agriculture, and so on. Therefore, large amounts of sensor data have to be ingested at the server side in order to generate and make the "twin digital model" or virtual factory of the existing physical processes for (among others) predictive simulation and scheduling purposes usable. In this paper, we focus on our ultimate goal, a novel software container-based approach with cloud agnostic orchestration facilities that enable the system operators in the industry to create and manage scalable, virtual IT platforms on-demand for these two typical major pillars of CPS: (1) server-side (i.e., back-end) framework for sensor networks and (2) configurable simulation tool for predicting the behavior of manufacturing systems. The paper discusses the scalability of the applied discrete-event simulation tool and the layered back-end framework starting from simple virtual machine-level to sophisticated multilevel autoscaling use case scenario. The presented achievements and evaluations leverage on (among others) the synergy of the existing EasySim simulator, our new CQueue software container manager, the continuously developed Occopus cloud orchestrator tool, and the latest version of the evolving MiCADO framework for integrating such tools into a unified platform.

## 1. Introduction

As indicated in [1], we are the witnesses of a parallel but strongly interinfluencing development in computer and manufacturing sciences including the related rapid technology evolution. The achievements of computer and ICT research efforts were directly applied in manufacturing hardware, control systems, and software. Moreover, the complexity and challenges generated in the field of manufacturing continuously influenced the developments in computer science and ICT. In line with the latest research and development trends, big data-related tools (see Section 2), such as predictive analytics and simulations, have been contributing to the increasingly wider range of sectors, including manufacturing, agriculture, healthcare, and other services. In many of those cases, cloud computing serves as an elastic and efficient paradigm for implementing sensor data ingestion and simulation back-ends. With the emerging lightweight software container technologies (see Section 2), the feasible approaches and design options for such platforms have been significantly enriched for cyber-physical systems (CPSs).

CPSs, relying on such latest and foreseeable further developments of (i) computer science, (ii) information and communication technologies, and (iii) manufacturing science and technology, may lead to the 4th Industrial Revolution, frequently noted as Industry 4.0 [1]. In our paper, we present the evolution of our ultimate goal, an *orchestrated platform* for Industry 4.0, realized first within the Docker@SZTAKI project (see Section 3), which is responsible for collecting (among others) sensor data and enables sophisticated simulations using the data. The different versions, namely, the cloud VM-based and the software containerized variants, provide the three key requested features, that is, highly scalable, vendor-independent (cloud provider/technology agnostic), and open-source facilities that help protect the sensitive factory data by using, for example, in-house cloud or servers at the company premises but, at the same time, allow accessing public cloud services seamlessly when it becomes necessary (e.g., in the case of peek/unpredictable load or for high-availability functionalities). Therefore, they formed a robust and adaptive framework for further pilot application areas, for example, connected cars and precision farming (see Section 5), as the presented evaluation illustrates, for example, the elasticity and other parameters of the current implementation that has been ported under MiCADO (see Section 4.2).

As a vital part of our back-end platform, the main goal of the presented *orchestrated platform* (see Section 3.1) is to reliably receive and store incoming sensor data (including images) from multiple arrays of configured sensors that are deployed in the factory, on the trucks, and so on. One of the most crucial requirements concerning such back-ends is the capability to scale as the number of sensors and the volume of incoming data grow rapidly. It can be achieved by the appropriate architecture, solid fundamental building blocks, and exhaustive testing. Another important goal of the sensor data back-end is to serve the collected data for analytical purposes, for example, to feed the simulators with initial data or to compare the already collected data with the simulation results.

As another fundamental component of the orchestrated back-end platform, a *discrete-event simulation* (DES) kernel is utilized to forecast the behavior of a manufacturing system, denoted as EasySim hereafter (see Section 3.2). Specific details about the simulation kernel are provided in [2]; here, we just provide some highlights of EasySim. The simulation kernel is an implementation of the classical simulation modeling approach called discrete-event simulation (DES) and developed earlier in MTA SZTAKI. In a nutshell, the DES approach utilizes a mathematical/logical model of a physical system that portrays state changes at precise points in a simulated time horizon. Both the nature of the state changes and the time at which the change occurs mandate precise description. DES models are mainly flow models tracking the flow of entities through the factory. The tracking is done using times at which the various events occur and are sequentially ordered according to their occurrence time. In the modeling phase, the task of a modeler is to determine the state variables that capture the behavior of the system, the events that can change the values of those variables and the logic associated with each event. Executing the logic associated with each event in a time-ordered sequence produces a simulation of the system. As each event occurs and expires, it is removed from the sequence called an event list, and the next event is activated. This continues until all the events have occurred or an earlier-defined time window limit is achieved. Statistics are gathered throughout the simulation and reported with performance measures. Later in the paper, we provide the simulation scenarios, where the EasySim kernel is applied (see Section 4.1), but the main focus will be on the scenario generation and the evaluation of the simulation runs, which, contrary to the initial desktop environment, will run in parallel on the orchestrated back-end platform. With the parallelization support of this back-end platform, we are able to significantly speed up the evaluation of different scenarios which earlier run only sequentially in desktop environments. Additionally, it is important to mention that other simulation engines, even third-party, off-the-shelf simulation software, would have been suitable to model the scenarios presented in Section 3.2. EasySim was selected because of performance reasons.

## 2. Related Work

In this section, we discuss popular approaches, methods, and already-available services related to our ultimate goals and particularly addressed high-level features: (i) multilevel autoscaling in the cloud involving VMs and containers, (ii) cloud technology/provider agnostic mechanisms including high portability, and (iii) generic open-source implementation, which together enable the efficient deployment and operation of the wide range of CPS services.

Regarding cloud technologies, already, several authors underlined the fact that cloud computing is playing a significant role in realizing cyber-physical systems [3–5].

Amazon Web Services (AWS), Microsoft Azure, and Google Cloud may be considered the three dominant forces in public cloud computing, and all the three provide their own IoT [6] platform and services [7–9]. These are generic, hosted platforms and not available as an on-premise private solution. There are several proposals available for data processing that are aimed at providing a generic architecture rather than one that fits a single-use case [10–12]; thus, these can be exploited not just for strict big data scenarios.

The FIWARE big data architecture [13] was created within the FIWARE (Core Platform of the Future Internet) project as one of many generic enablers (GEs). A GE represents a functional building unit. A GE implementation supports a given set of functions over a set of APIs and interoperable interfaces that conform to the open specifications given for that GE [14]. The big data GE architecture expands the basic Apache Hadoop one. The Master Node has all management software and acts as a front-end for the users. Infinity is the permanent storage cluster (based on HDFS). Computing clusters have a lifecycle: they are created, they are used for computation, and, finally, they are removed. All data must be uploaded to Infinity beforehand. Data can be uploaded to and retrieved from Infinity via WebHDFS [15] or Cosmos CLI (a command line interface to WebHDFS). The big data GE specifies the use of SQL-like analytic tools like Hive, Impala [16], or Shark. Although the GE is based on Hadoop [17], it proposes several alternatives: (i) the Cassandra [18] File System can be used instead of HDFS, (ii) a distributed NoSQL database like HBase can be installed on top of HDFS, and (iii) use, for example, Cascading [19] as an extension or replacement.

DICE [20] is an EU Horizon 2020 research project that is aimed at providing a methodology and framework for developing data-intensive applications. It offers a framework consisting of an Eclipse-based IDE and tools and supports Apache Spark [21], Storm [22], Hadoop (MapReduce), Cassandra, and MongoDB [23]. By using its methodology, it allows the architecture enhancement, agile delivery, and testing for batch and stream processing applications.

Building on application containers and orchestration (e.g., via Docker [24] or Kubernetes [25]), serverless computing is an execution model for cloud computing where the cloud provider manages dynamically the underlaying machine resources. The pricing model is based on the actual resources consumed during execution (e.g., CPU execution time, memory, and network). All major public cloud providers support this model, for example, AWS Lambda, Google Cloud Functions, or Azure Functions. There are several open-source implementations like OpenLambda [26], OpenFaaS [27] (Open Function as a Service), Kubeless [28], Funktion, Iron Functions, and Fission.

Terraform [29] is an open-source tool for building, managing, and versioning virtual infrastructures in public or private cloud environments. Terraform allows defining whole virtual infrastructures via a configuration template. This can contain low-level information like machine types, storage, or network configuration but also high-level components like SaaS services or DNS entries. Based on the configuration, Terraform creates an execution plan and a resource graph to build the defined infrastructure. Topology and Orchestration Specification for Cloud Applications (TOSCA) [30, 31] is a standard language by OASIS [32] for describing collections or topologies of services, their relationships, components, and so on. It is similar to Amazon CloudFormation, OpenStack Heat, and Terraform. It is aimed at being an open standard that provides a superset of features (and grammar).

Regarding the representation and sharing industrial data in distributed systems, several initiatives exist. The National Institute of Standards and Technology (NIST) initiated the Smart Manufacturing Systems (SMS) Test Bed [33] in which data is collected from the manufacturing lab using the MTConnect (link is external) standard. That data is aggregated and published internally and externally of NIST via web services. Other initiative from General Electric is Predix [34], a huge platform enabling the collection and analysis of product- and asset-related data in order to improve and optimize operations. The SMS Test Bed is a source from where data can be retrieved and analyzed, but the Test Bed itself does not include solvers or simulators. Predix with its own multiple layers is designed for collection and analytics and includes tools for analysis, but building models which are later applied in the decision support here is also always necessary and at the same time almost the most difficult part. How to build these models is still not clear from the available sources of Predix. In our solution, to be presented in the simulation scenario, the model in question is built as discrete-event simulation with a tool developed earlier in an earlier project at MTA SZTAKI. This model is built automatically based on the Core Manufacturing Simulation Data (CMSD) standard, specifically designed for manufacturing simulation studies.

## 3. Docker@SZTAKI Project

The main motivation behind the Docker@SZTAKI project was to elaborate and demonstrate a Docker software container-based platform that can be formed on demand in a highly portable way, that is, according to the complexity and the actual needs of the CPS application in various IT environments. The supported environments (see Figure 1) include even the user's laptop, or the on-premise servers of the institute, and also a wide range of private/public/community clouds, for example, the Hungarian academic federated community cloud, the MTA Cloud (based on OpenStack middleware), or the public Amazon Web Services (AWS) cloud.

The Docker@SZTAKI platform (see Figure 1) consist of a private repository of Docker images, for example, the EasySim simulator (see Section 3.2), the various components of the presented sensor data back-end, and further auxiliary tools, such as the CQueue manager (see Section 3.2) or the Occopus [35] cloud orchestrator. CQueue plays a crucial role when the push model of the Docker Swarm clustering mechanism cannot be applied, and the pull model is more suitable for the application (e.g., in the case of EasySim). The Occopus cloud orchestrator is responsible for creating and managing the required VMs in the selected clouds when the Docker@SZTAKI user needs extra or 24/7 available IT capacities for their applications.

The platform has been used for demonstrating two major pillars of the CPS components: sensor data back-end and DES simulation.

### 3.1. Orchestrated Back-End Framework for CPS.

*3.1. Orchestrated Back-End Framework for CPS.* The goal of the framework is to reliably receive and store incoming sensor data (including images) from arrays of configured sensors and scale the number of sensors as needed. The framework also includes complementary user and administrative applications and connected analytics. In this section, we are going to detail the evolution of the framework in three iterations. Sensor data is usually generated by small microcontroller-based devices where usually raw data is from an arbitrary number of different instruments. Measurements are taken periodically and, thus, it generates a large number of small packets. Storing a large volume of this kind of data requires a tailored infrastructure with the capability to scale out (horizontally) as the volume of data grows.

*3.1.1. Virtual Machine-Based Architecture.* The architecture follows a three-tier layout as depicted in Figure 2. Each component of each tier can be deployed on a separate node. This allows easy scaling of the appropriate tiers.

The delivery tier (shown in Figure 2) accepts incoming sensor data and forwards it to one of the data collector application instances in the aggregation tier. The forwarding decision is made in two steps. First, based on a round-robin algorithm, a high-availability proxy and load balancer (based on HAProxy [36]) are selected. The proxy, in turn, will select an application server with the lowest load and forward the request to that one. A data collector instance in the aggregation tier (shown in Figure 2) will decode the received data and store them in the database tier (shown in Figure 2). Besides the data collector, other functionalities are also available and work similarly. Database services are provided by the Cassandra or MongoDB [23] database cluster, besides RDBMS-like MySQL. Cassandra is a decentralized structured

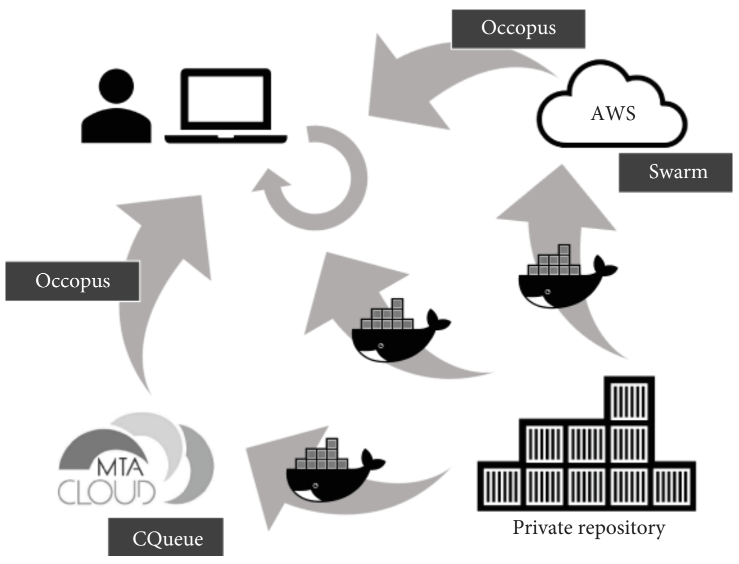


Figure 1: Docker@SZTAKI: main components with their typical usage scenarios.

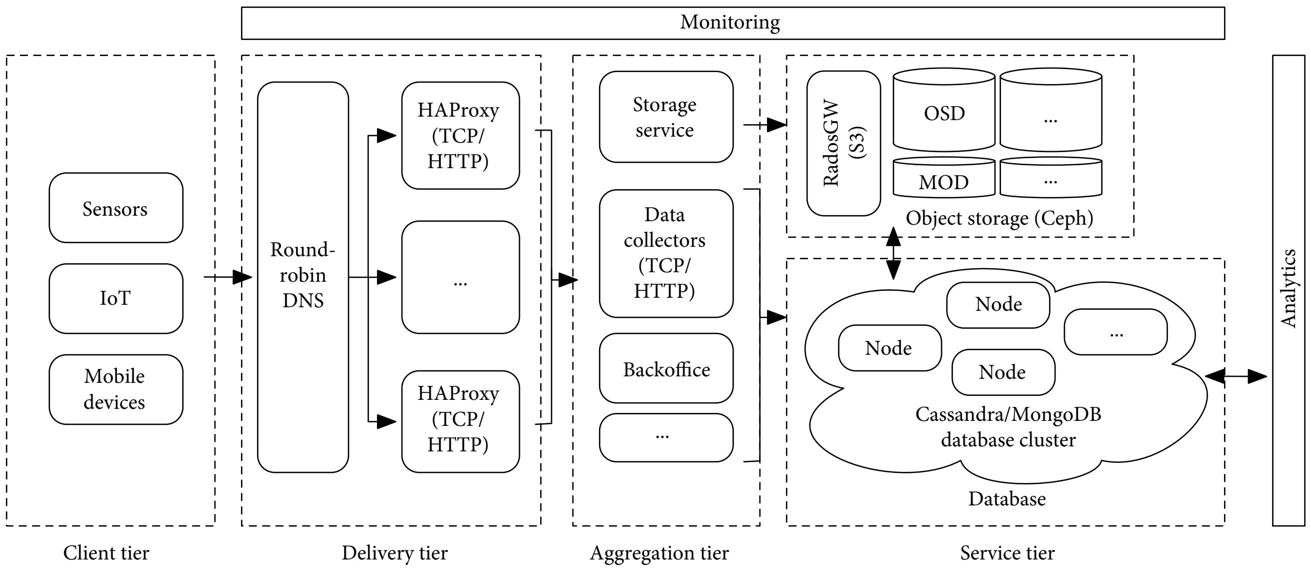


Figure 2: General architecture of the framework.

storage system that is well suited for storing time-series data like sensor data. As the volume of incoming data changes, Cassandra allows dynamically adding or removing new nodes to the database.

Data submission is initiated by the client tier resolving the DNS endpoint of a given service. The DNS endpoint may contain one or more load balancer address; in turn, they distribute the load between the available receiver instances. Using round-robin DNS techniques, it is possible to scale the number of load balancer nodes. It is a well-known simple method for load sharing, fault tolerance, and load distribution for making multiple redundant service hosts available. Next, HAProxy servers are responsible for balancing the load across multiple application servers (e.g., data collectors) after through the round-robin DNS the client contacts one. HAProxy also continuously monitors the health and performance of the application servers connected.

A data receiver application and connected components are depicted in Figure 3. It consists of the following: Chef is used as a deployment orchestrator for bootstrapping new nodes for the different tiers. The data processing component and Cassandra connector are implemented using the Flask web framework and Python. The sensor metadata decoder is responsible for interpreting the incoming data and passing it to the Cassandra connector. The Cassandra connector is used to store the decoded metadata in the database cluster. uWSGI [37] is used as a WSGI [38] application server, and, finally, Nginx [39] is connected to the wire protocol of uWSGI to achieve a high-performance WSGI-based web front-end.

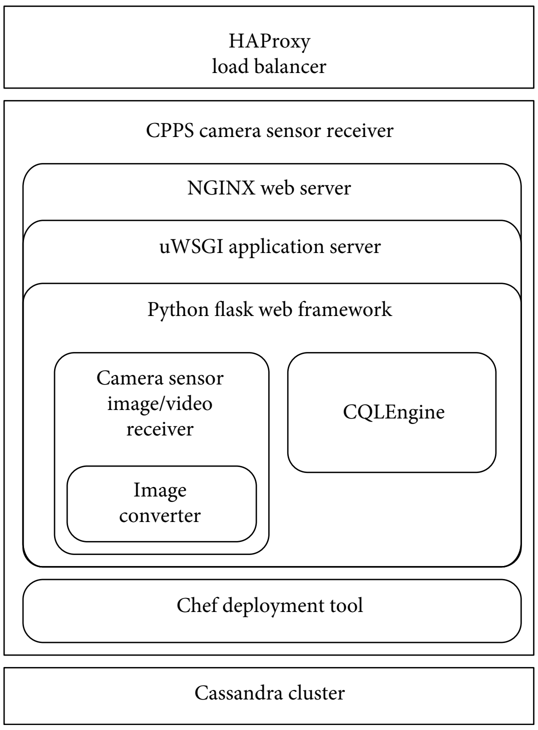


Figure 3: Architecture of a web-based data collector application for sensor image data.

*3.1.2. Container-Based Architecture.* The original data collector framework is based on virtual machines, and the components are run on separate nodes. This architecture is ideal to scale out or scale in the components based on the application utilization. On the other hand, this separation might have a negative effect on the resource utilization. To achieve better resource utilization, we have created the Docker version of the data collector infrastructure with smaller granularity. With the Docker container technology [24], the components of the architecture can be separated into containers; therefore, we can run more than one collector component on one particular virtual machine. The Docker version of the collector provides more efficient resource utilization than the virtual machine-based solution.

To create the Docker-based version, we built container images from each of the infrastructure components. We extended the application with a new configuration file which can be customized through the environment variables on the container start. This configuration is performed by the Docker entry point script at the start (this is the main configuration method in the Docker ecosystem). For the Cassandra Docker version, the official Cassandra image was selected from the Docker Hub but we applied some modifications; the official entry point script was extended to support the automatic Cassandra cluster creation at the start time on a Docker Swarm cluster. With these images, we created a Docker compose file to provide a simple container orchestration method. With the compose file, the main part of the collector infrastructure can be deployed by the service operator on one machine or on a Swarm cluster as a Docker stack, and the containers can be easily configured through the compose file with various environment variables.

The service operator can deploy the data collector framework as a Docker stack from the described Docker compose file on a cluster managed by Docker Swarm. Another important feature of Docker Swarm is the provided overlay network between the Swarm nodes for the containers. In this network, the containers can access each other like they are on one virtual machine. Furthermore, Swarm provides an ingress routing mesh on this network. With the routing mesh, the Swarm services can expose their ports on the virtual machines so they can be reached on every Swarm node from outside of the cluster. With that feature, Swarm provides an external load balancer between the application containers within a Docker service. Therefore, we decided to replace the HAProxy in the data collector infrastructure with the above-described routing mesh facility of Swarm. The resulting architecture is demonstrated in Figure 4. Prometheus [40] is used for monitoring the infrastructure with agents deployed on the nodes.

The infrastructure is deployed and managed by the Occopus [35] cloud orchestrating tool. Occopus is open-source software providing features to orchestrate, configure, and manage virtual infrastructures. It allows describing virtual infrastructures in a cloud agnostic way. We created the necessary description files to build and maintain the collector framework. As an additional benefit, the number of Swarm workers in the framework can be automatically scaled based on their CPU load.

*3.1.3. Extended Architecture.* In the next iteration of the data collector, we improved the data storage layer and separated the functions of the data collector layer to improve the disadvantages of the framework. In the first version, all metadata about the sensors and the measured data are stored in the Cassandra database. This is not an optimal database schema to store related data in a NoSQL database; therefore, we separated the stored data into two databases. The information and the corresponding metadata of the sensors are stored in an SQL database, and measurement data will be stored in a NoSQL database or distributed file system. Originally, data collectors served multiple purposes: receiving, processing, and storing the data in a database. These functions have been separated into distinct components. These streaming components push data to a streaming component, and dedicated processors store the data for further analytics or process them in-stream. This greatly reduces the stress on the data collector and furthers the architecture. The extended collector architecture is demonstrated in Figure 5.

### 3.2. EasySim Discrete-Event Simulation for CPS

*3.2.1. CQueue and Its Pull Execution Model for Tasks.* Since Docker does not provide a pull model for task execution (Swarm uses a push execution model), the new CQueue framework provides a lightweight queue service for processing tasks via application containers. The framework consists of four main components (see Figure 6): (i) one or more

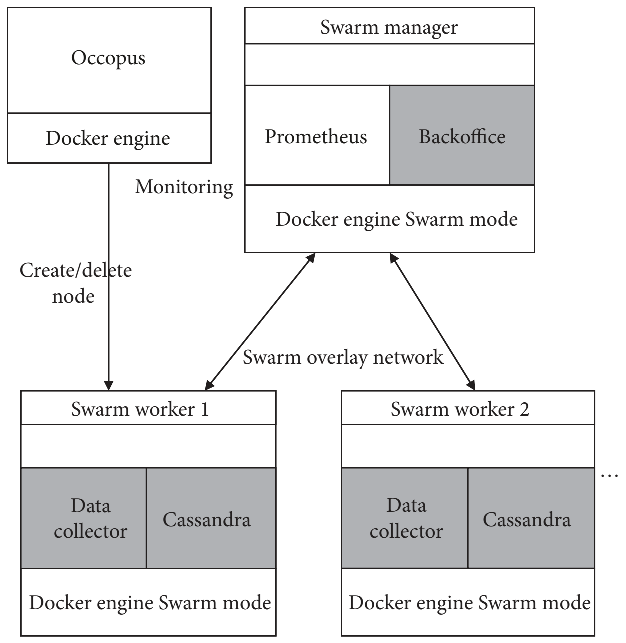


Figure 4: Container-based architecture of the sensor data back-end framework with VM-level autoscaling.

CQueue server, which acts as a front-end and receives the container-based task requests; (ii) a queue server which schedules the task requests for workers; (iii) CQueue workers that pull tasks from the queue server; and (iv) a key-value store that stores the state and the output of the finished tasks. Currently, queuing is handled by RabbitMQ, and Redis is used as the key-value store. The front-end server and the worker components are written in Golang, and they have a shared code base. All of the components are running inside Docker containers and can be scaled based on their utilization. The design goals of the framework are to use standard interfaces and components to create generic job processing middleware.

The framework is built for executing Docker container-based tasks with their specific inputs. Also, environment variables and other input parameters can be specified for each container. CQueue uses a unique ID to identify the pushed tasks, and the user has to specify it. The ID, the application container, and the inputs of the task must be specified in the standard JSON (JavaScript Object Notation) format. The CQueue server receives the tasks via a REST-like API. After this step, the server transforms the JSON-formatted tasks to standard AMQP (Advanced Message Queuing Protocol) messages and pushes them to the queue server. The workers pull the registered tasks from the queue server via the same AMQP protocol and execute them. One worker processes one task at a time. After the task is completed, the workers send a notification to the queue server, and this task will be removed from the queue. The worker continuously updates the status (*registered*, *running*, *finished*, or *failed*) of the task with the task's ID in the key-value store. When the task is finished or failed, the worker stores the *stdout* and *stderr* of the task in the key-value store as well. The status of a task and the result can be queried from the key-value store through the CQueue server. The output of the task is not processed; it is stored in the key-value store in its original format.

*3.2.2. Architecture from the Simulation Point of View.* Figure 7 illustrates the overall architecture of the system from a simulation experiment point of view. The graphical user interface, denoted by GUI in Figure 7, is the place where the configuration of an experiment with different parameters can be defined. The Experiment Manager then forwards the complete setup of the simulation and initiates the parallelized simulation runs accordingly. The main database is given by the DB symbol in the bottom of the figure, storing both the structural input of the simulation and the parameters of the simulation instances, and, later, after the simulation runs are terminated, all the outputs are streamed by the simulation runs. To better understand the process within this structure, the following section gives the details of the modules introduced above.

*3.2.3. Standardized Database.* Concerning the implementation of the persistence layer, MySQL has been selected to store all the necessary back-end information.

Regarding the standardization of manufacturing and logistic systems, there are different standards approved and offered by different organizations; the most known one is ISA 95, provided by the International Society of Automation [41]. Having a comparison on the base of applicability, finally, we selected the standard for Core Manufacturing Simulation Data (CMSD) [42] in order to have a standardized system with reusable components. In this way, we applied standard data formats for representing certain structures of the system related to the simulation module, namely, the SISO-STD-008-2010; the standard for Core Manufacturing Simulation Data (SISO CMSD) provided by SISO (http://www.sisostds.org, visited 2017-11-01) is applied in the research.

This standard addresses interoperability between simulation systems and other manufacturing applications. As such, it inherently includes the most relevant and simulation-related data for the simulation of manufacturing systems. The CMSD model is a standard representation for core manufacturing simulation data, providing neutral structures for the efficient exchange of manufacturing data in a simulation environment. These neutral data structures are applied to support the integration of simulation software with other manufacturing applications.

The CMSD standard has several packages, but not all of them are necessary in this application. Just as an example, the layout package was not used, as in our scenario the focus of the experiment which is the layout is not relevant. The standard itself is described as a UML model; furthermore, there are XML and representations in different programming languages. Within the context of the research, the back-end database was designed and implemented with an implementation of the CMSD standard in a relational database format, based on the initial UML version, forming the main data storage of different simulation forecasting

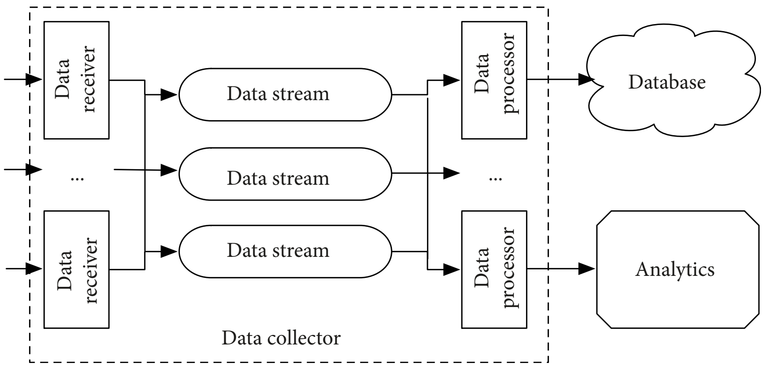


FIGURE 5: Data collectors in the extended sensor data back-end architecture.

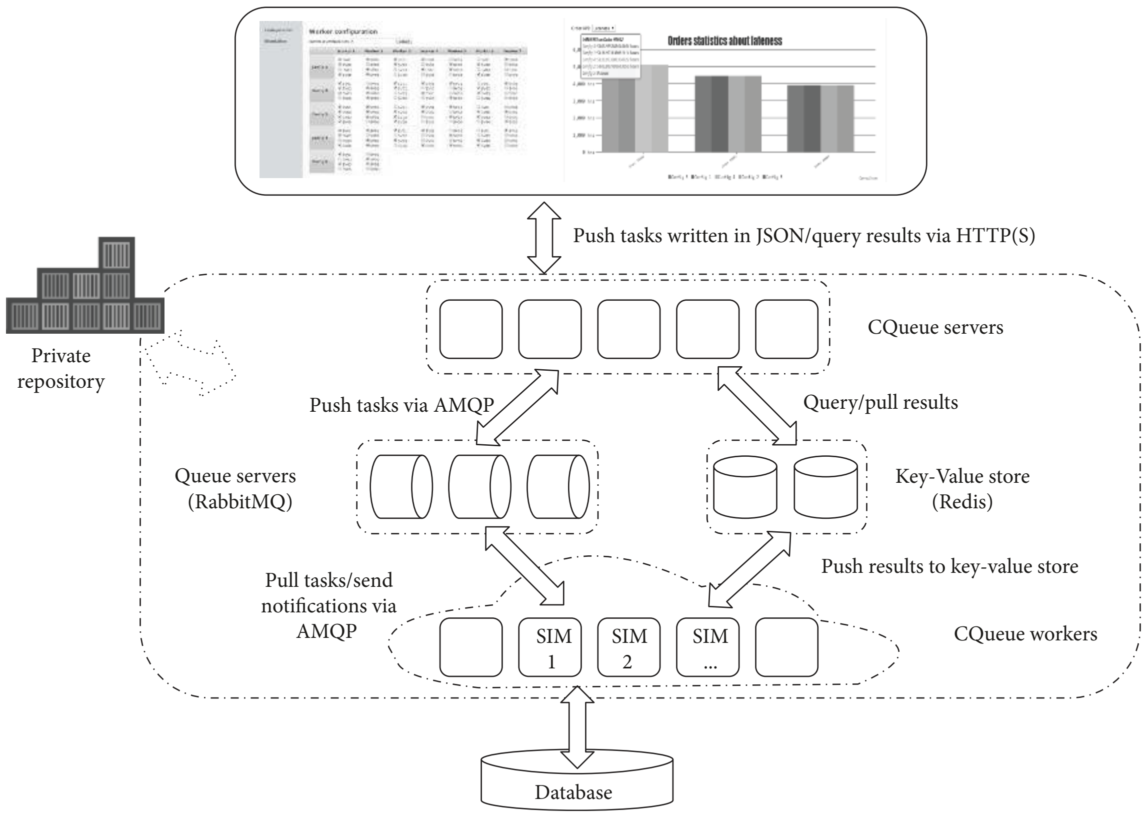


FIGURE 6: CQueue architecture in the context of simulation.

scenarios. All the data about the resources, the entities or workpieces traveling in the manufacturing system, the routings, the sequence of operations, the control logic, and the manufacturing orders to be completed are all stored in the database according to the CMSD specification. One of the nonfunctional requirements for selecting this solution, namely, the direct implementation of SQL database tables and relations, was the speed of building and updating simulation models instantly.

According to the nature of the data stored in the MySQL database, two types of tables can be distinguished. On the one hand, the implementation of the CMSD standard provides the information related to simulation. On the other hand, there are tables, which store specific information necessary for the application itself in this new environment.

*3.2.4. Data Access Layer.* Both the higher-level GUI that is responsible for setting up the input parameters and visualizing the results of simulation scenarios and the Docker Container Manager (currently CQueue) are connected to the main database with a data access layer (DAL) (see Figure 7).

The correspondence, the bidirectional match between classes in the system and data in the database, is assured by the data access layer which is implemented with the Entity Framework. This allows a higher level of abstraction when dealing with data and supports the creation and maintenance of data-oriented applications with less code than in traditional applications. The objects linked to relational data are defined just like normal objects and decorated with attributes to identify how properties correspond to columns in the database tables.

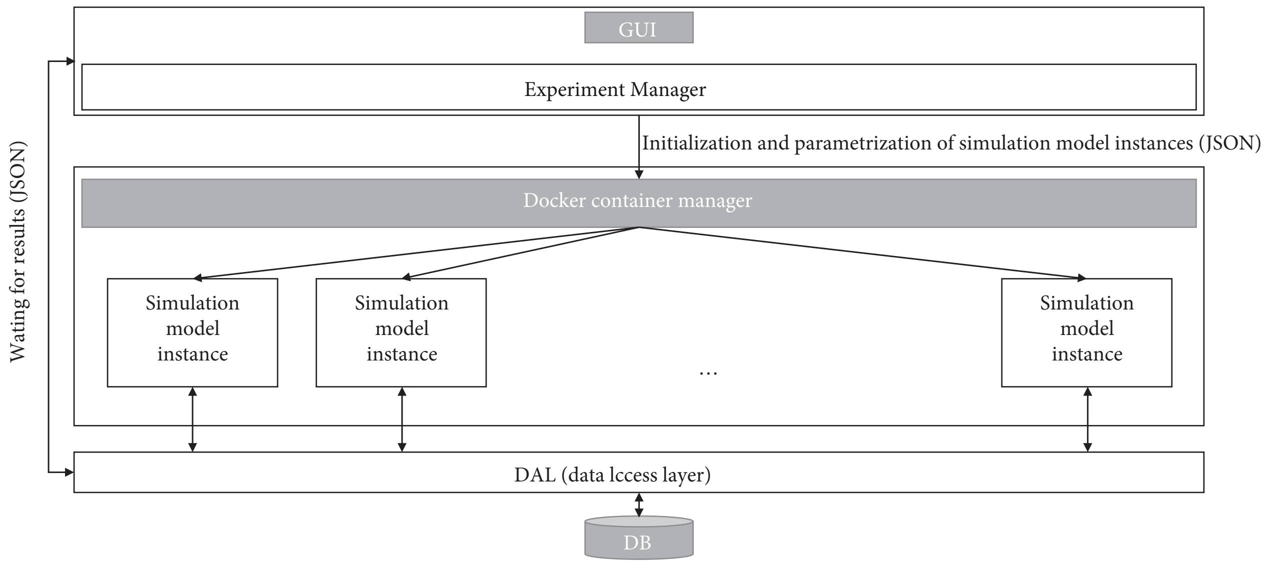

GUI
Experiment Manager
Initialization and parametrization of simulation model instances (JSON)
Docker container manager
Simulation model instance
Simulation model instance
...
Simulation model instance
Wating for results (JSON)
DAL (data lccess layer)
DB

Figure 7: Web application architecture for simulation management/visualization (extended by the Docker Container Manager).

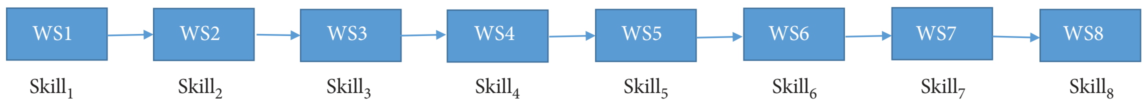


Figure 8: The layout of the flow-shop system.

## 4. Evaluation of CPS Use Cases

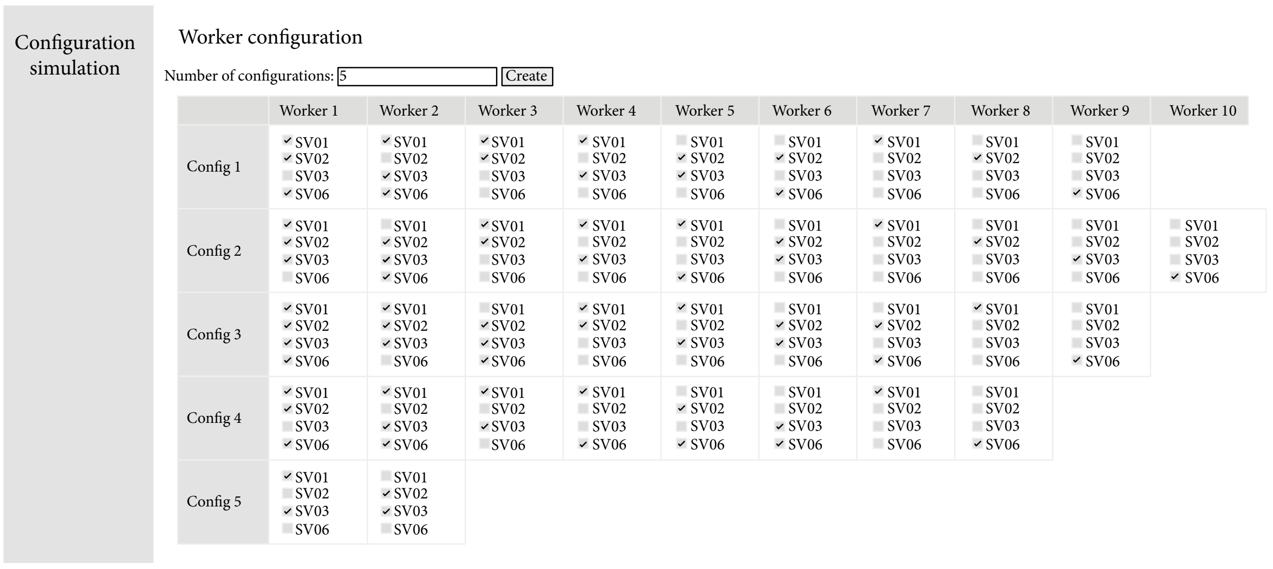
*4.1. EasySim: The Modeled System and Its Simulation Experiments.* As mentioned in the introductory part, the reason for using EasySim instead of any other existing DES tools is the difference in performance. EasySim is a simulation kernel providing only the core functionality of a DES tool. No graphical user interface had been developed for it, and we believe that one the most promising data structures had been selected to represent the event list because its properties (size, the speed of accessing its content) can highly influence the speed of the simulation. Furthermore, EasySim had been developed for building a DES model in the most direct way with programming. The overall intention to develop EasySim in such way was to achieve fast simulation runs, and because EasySim is our own implementation, we could be sure that the integration with the other tools in this paper would be in the most convenient way. Again, it is true that the simulation model presented below would have been implemented in any other simulation software. EasySim was selected because of its flexibility to be integrated in the back-end platform.

Regarding the simulation model, it implements a production line which contains eight workstations connected to each other in a linear way, called a flow-shop line. The modeled production line is part of a real manufacturing factory, and operation times were given for each workstation provided by the factory. Additionally, the model implements some kind of stochastic behavior such as failure of workstations which can be optionally used in the simulation. This capability of stochastic behavior has been realized by integrating a mathematical software package during the development of EasySim which ensures proper random number handling and different mathematical functions to approximate reality as much as it is possible. The operations on each workstation are different and may require the presence of one or more human workers who perform the manufacturing of the assembly task at the given station. As in a real production system, the workers are also different; for each specific task at a workstation, a worker needs to have a specific skill. Moreover, the operators are assigned to specific shifts meaning that shift by shift we can have different teams, grouping different workers with different skills. As Figure 8 illustrates, it is a linear, acyclic production line which contains eight workstations (WS1, WS2, etc.). Below each workstation, there is a required skill name which indicates that a worker can operate on the workstation only if the worker has the specific skill. A worker can have multiple different skills meaning that he can operate on different workstations. An evident solution is to have one worker for each workstation with the required skills of course, but in reality, factories have less workers available to allocate them, so the task is to find an optimal worker set which is able to carry the order out with a minimal number of workers.

The task of the planner is to find the right configuration of workers for each specific shift. Naturally, the problem can be formulated as a formalized mathematical problem, but as the nature of the operation times is stochastic (i.e., each operation varies and follows a distribution; additionally, failures may occur unexpectedly at each workstation), the

Table 1: The layout of the flow-shop system.

| | $Skill_1$ | $Skill_2$ | $Skill_3$ | $Skill_4$ | $Skill_5$ | $Skill_6$ | $Skill_7$ | $Skill_8$ |
|---|---|---|---|---|---|---|---|---|
| $Worker_1$ | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| $Worker_2$ | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| $Worker_3$ | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |



| | Worker 1 | Worker 2 | Worker 3 | Worker 4 | Worker 5 | Worker 6 | Worker 7 | Worker 8 | Worker 9 | Worker 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Config 1 | ✓ SV01 ✓ SV02 ☐ SV03 ✓ SV06 | ✓ SV01 ☐ SV02 ✓ SV03 ✓ SV06 | ✓ SV01 ✓ SV02 ☐ SV03 ☐ SV06 | ✓ SV01 ☐ SV02 ✓ SV03 ☐ SV06 | ☐ SV01 ✓ SV02 ✓ SV03 ☐ SV06 | ☐ SV01 ✓ SV02 ☐ SV03 ✓ SV06 | ✓ SV01 ☐ SV02 ☐ SV03 ☐ SV06 | ☐ SV01 ✓ SV02 ☐ SV03 ☐ SV06 | ☐ SV01 ☐ SV02 ☐ SV03 ✓ SV06 | |
| Config 2 | ✓ SV01 ✓ SV02 ✓ SV03 ☐ SV06 | ☐ SV01 ✓ SV02 ✓ SV03 ✓ SV06 | ✓ SV01 ✓ SV02 ☐ SV03 ☐ SV06 | ✓ SV01 ☐ SV02 ✓ SV03 ☐ SV06 | ✓ SV01 ☐ SV02 ☐ SV03 ✓ SV06 | ☐ SV01 ✓ SV02 ✓ SV03 ☐ SV06 | ✓ SV01 ☐ SV02 ☐ SV03 ☐ SV06 | ☐ SV01 ✓ SV02 ☐ SV03 ☐ SV06 | ☐ SV01 ☐ SV02 ✓ SV03 ☐ SV06 | ☐ SV01 ☐ SV02 ☐ SV03 ✓ SV06 |
| Config 3 | ✓ SV01 ✓ SV02 ✓ SV03 ✓ SV06 | ✓ SV01 ✓ SV02 ✓ SV03 ☐ SV06 | ☐ SV01 ✓ SV02 ✓ SV03 ✓ SV06 | ✓ SV01 ✓ SV02 ☐ SV03 ☐ SV06 | ✓ SV01 ☐ SV02 ✓ SV03 ☐ SV06 | ☐ SV01 ✓ SV02 ✓ SV03 ☐ SV06 | ☐ SV01 ✓ SV02 ☐ SV03 ✓ SV06 | ✓ SV01 ☐ SV02 ☐ SV03 ☐ SV06 | ☐ SV01 ☐ SV02 ☐ SV03 ✓ SV06 | |
| Config 4 | ✓ SV01 ✓ SV02 ☐ SV03 ✓ SV06 | ✓ SV01 ☐ SV02 ✓ SV03 ✓ SV06 | ✓ SV01 ☐ SV02 ✓ SV03 ☐ SV06 | ✓ SV01 ☐ SV02 ☐ SV03 ✓ SV06 | ☐ SV01 ✓ SV02 ☐ SV03 ✓ SV06 | ☐ SV01 ☐ SV02 ✓ SV03 ✓ SV06 | ✓ SV01 ☐ SV02 ☐ SV03 ☐ SV06 | ☐ SV01 ☐ SV02 ☐ SV03 ✓ SV06 | | |
| Config 5 | ✓ SV01 ☐ SV02 ✓ SV03 ☐ SV06 | ☐ SV01 ✓ SV02 ✓ SV03 ☐ SV06 | | | | | | | | |

Figure 9: User interface for parameterizing workers' skills and teams.

usage of a discrete-event simulation tool is more adequate to model the line in question. To provide input, control the run of the simulation model, and visualize the results of the simulation scenarios, a web application was developed and integrated with the orchestrated back-end platform described earlier. This is presented in the upper part of Figure 7 and includes the Experiment Manager and the GUI for visualization. The visualized functionality of the EasySim DES kernel introduced at the beginning of the paper is available through this web application, and some examples of the GUI of the system is presented in the figures of the following section.

*4.1.1. Parametrization of the System.* There are two main input sets for each simulation experiment. First, the model should be fed with production orders which are recorded giving the order IDs, due dates, and product quantities. All other data describing the products, their routing, and the operation times are given in advance and stored in the database. The second input, which is specific to the model, is the matrix of workers and their skills.

In the example presented in Table 1, $Worker_1$ can work on WS1, WS4, and WS6 workstations but cannot work on WS2, WS3, WS5, WS7, and WS8 workstations. A worker can have multiple different skills, so considering the example before, a worker can operate both on WS2 and on WS3 only if he has the $Skill_2$ and $Skill_3$ skills.

Figure 9 shows how the parametrization of the worker's skills can be completed with the help of the high-level GUI. As you can see, there are ten different workers provided as columns in the matrix while in each configuration—which will run in parallel on the orchestrated back-end platform—separate skill patterns can be defined for each worker. These are denoted by the names of the workstations, e.g. SV01 and SV02.

*4.1.2. Execution of Simulation Runs and the GUI.* Having the input parameter set, the simulation model instances are built dynamically for each configuration and the simulation runs start in parallel. One simulation run includes one worker configuration (a row with a Config *X* label on the figure of the worker configuration) with the selected orders. These instantiated simulations as configurations are detached from the GUI, and they are handled on the orchestrated back-end platform as separate workers. When a simulation run is completed (i.e., the worker with its simulation finishes), the output statistics are saved into the database for each configuration. After the successful execution of each worker, the Docker Container Manager notifies the GUI about the completion of the simulation run, and when all the running configurations were completed, the GUI can visualize the simulation results. Figure 10 provides the statistics about the utilization of the workers in configuration number 3. The blue part in the top region of the figure illustrates the percentage the operator was idle while the green part indicates the time where the operator was working. With the orders completed in this configuration, we can see that by

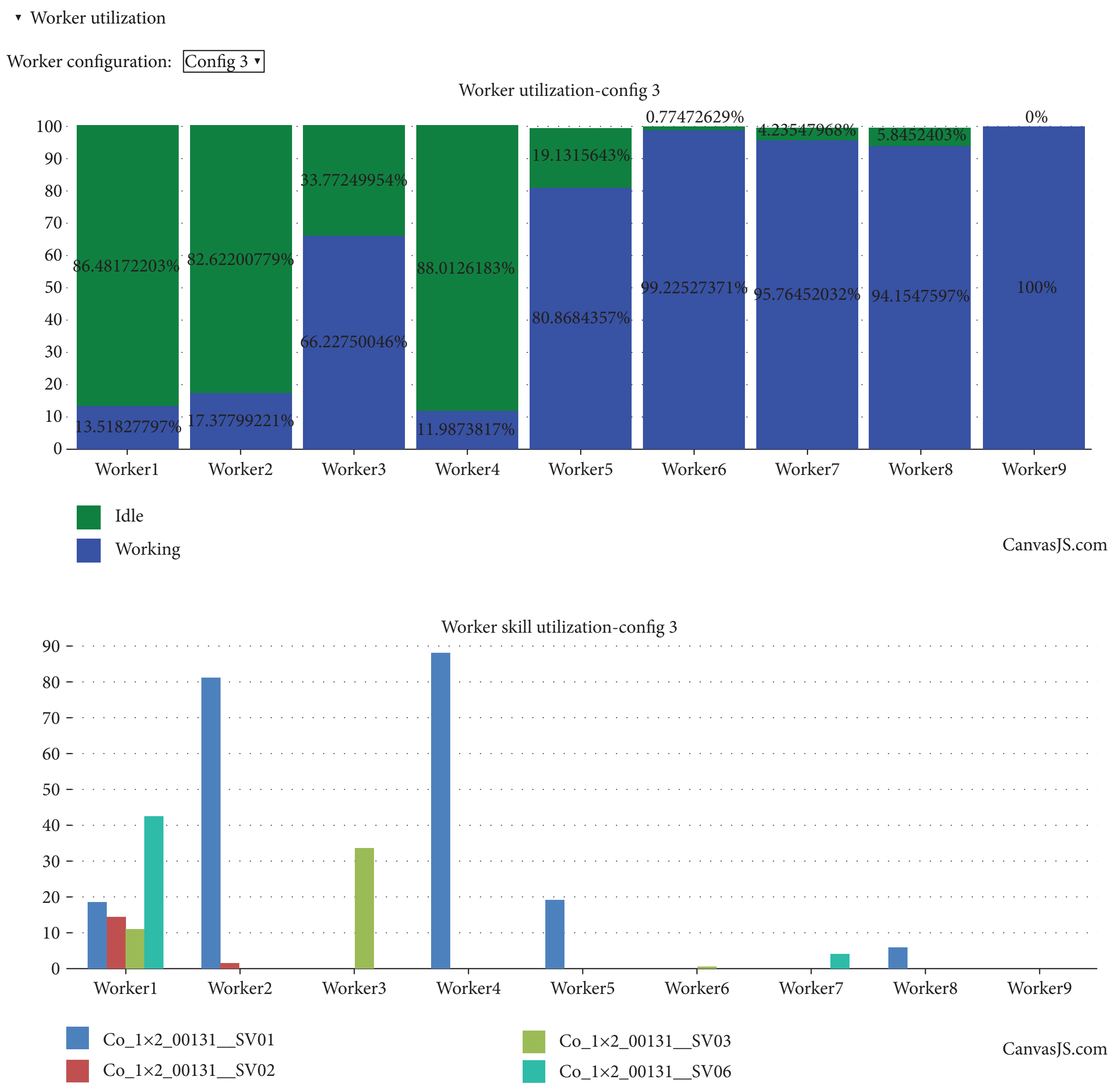
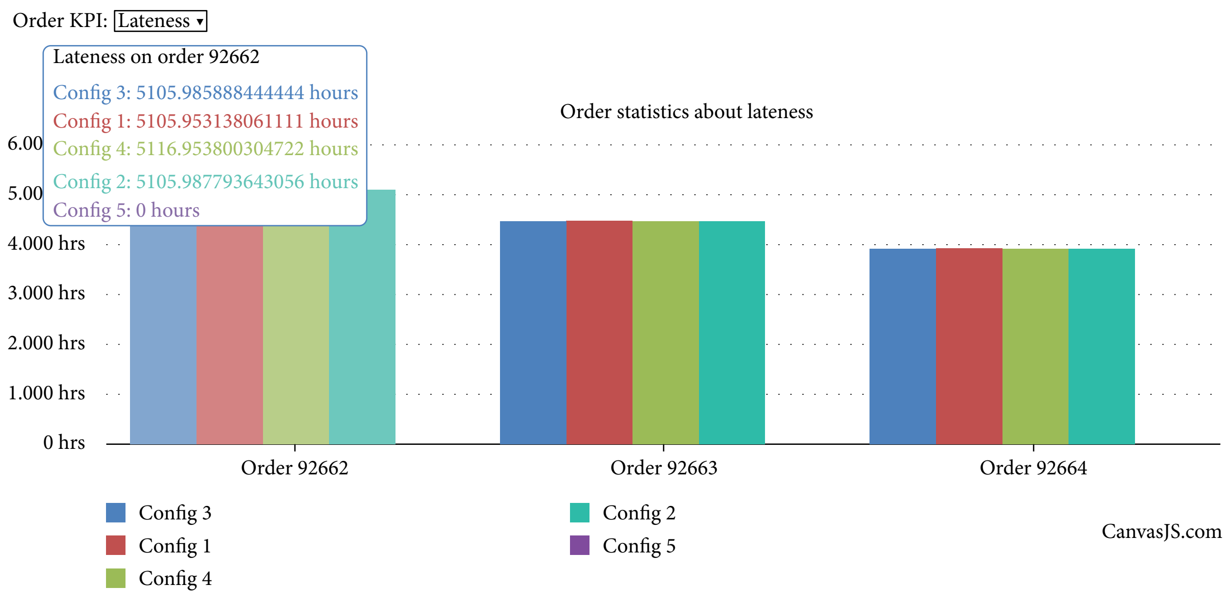


Figure 10: Chart representing the utilization of workers in a configuration.

applying seven operators, we will have a very underutilized environment. Figure 11 gives an overview of how the 3 distinct orders behaved in the system, mainly meaning that there were no significant differences between the five different configurations. As the main focus of the paper is the orchestrated back-end, we additionally included some explanatory charts, but many additional key performance indicators can be visualized within the GUI. Some of them visualize aggregated measures while others specific resource, buffer, or worker-related measures.

*4.2. Sensor Data Back-End Experiments: Multilevel Autoscaling in MiCADO.* The developed sensor data back-end has been successfully migrated under the MiCADO [43] (Microservices-based Cloud Application-level Dynamic Orchestrator) framework that attempts to unify and also to extend the previously described tools including Occopus and CQueue in a long term. It allowed us to evaluate the sensor data back-end in a more fine-grained way using multilevel scaling, that is, not only at the VM level but also at the container level. This approach utilized the two control loops of MiCADO that led to the presented results.

*4.2.1. MiCADO from a Scalability Point of View.* MiCADO [43] (developed by the EU H2020 COLA [44] project) is a framework performing the execution of compound microservice-based infrastructure on cloud resources. Beyond the execution, the most important functionality of MiCADO is the automatic scaling on two levels. The microservice-level autoscaling deals with keeping the optimal number of



Lateness on order 92662
Config 3: 5105.985888444444 hours
Config 1: 5105.953138061111 hours
Config 4: 5116.953800304722 hours
Config 2: 5105.987793643056 hours
Config 5: 0 hours

🟦 Config 3
🟥 Config 1
🟩 Config 4
🟩 Config 2
🟪 Config 5

Figure 11: Statistics of the orders pulled through the system.

container instances for a particular microservice. The cloud-level autoscaling deals with allocating the optimal amount of cloud resources for the entire microservice-based infrastructure.

MiCADO is developed by integrating various tools into a common framework. For executing microservice infrastructure, a dynamically extendable and resizable Swarm [45] cluster is used. Monitoring is performed by Prometheus [40] with agents on the nodes of the cluster. The communication with the cloud API and the orchestration of the Swarm cluster is performed by Occopus [35] mentioned in previous sections. Each of the components is integrated taking into account the replaceability in the future in case a better tool appears in its area. The scaling and optimization logic is built by the COLA project as well as the submission interface. For describing the microservice infrastructure, the project has chosen the TOSCA [31] specification language where the components, requirements, relations, and so on can be easily defined in a portable way. The way of describing scaling/optimization policies is developed by the COLA project as an extension of the TOSCA specification.

The conceptual overview of the two control loops implemented by the aforementioned components and tools is shown in Figure 12. In both control loops, Policy Keeper performs controlling and decision-making on scaling while Prometheus acts as a sensor to monitor the measured targets. In the microservice control loop, the targets are the microservice containers realizing that the infrastructure can be controlled. Containers are modified (number, location, etc.) by Swarm playing as an actuator in the loop. A similar control loop is realized for the cloud resources represented by virtual machines in our case. Here, Occopus acts as an actuator to scale up/down the virtual machines (targets). The microservice control loop controls the consumers while the cloud-level control loop controls the resources. As a consequence, the microservice loop affects the cloud loop since more consumers require more resources.

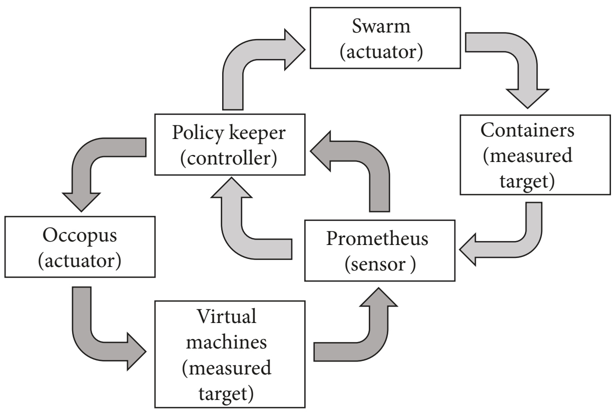
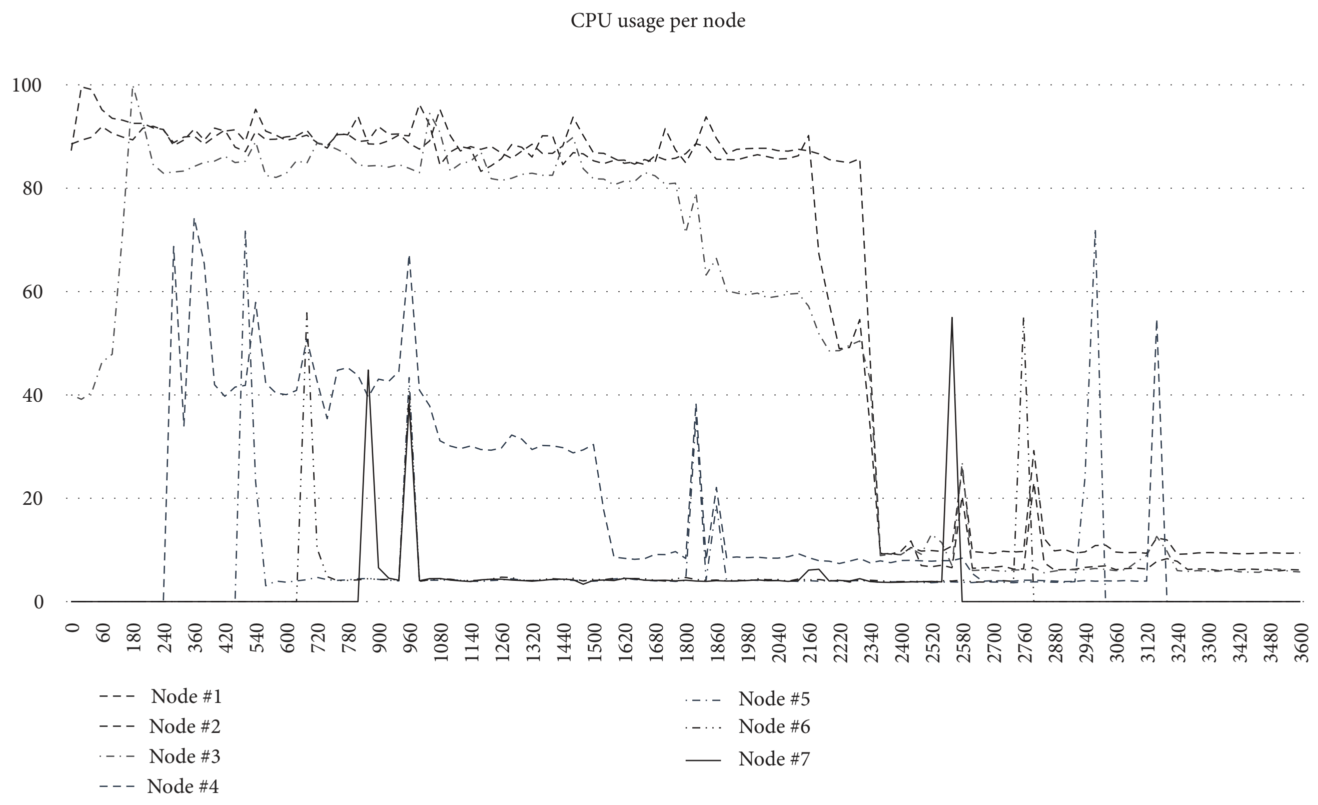

Swarm (actuator)
Policy keeper (controller)
Containers (measured target)
Occopus (actuator)
Prometheus (sensor )
Virtual machines (measured target)

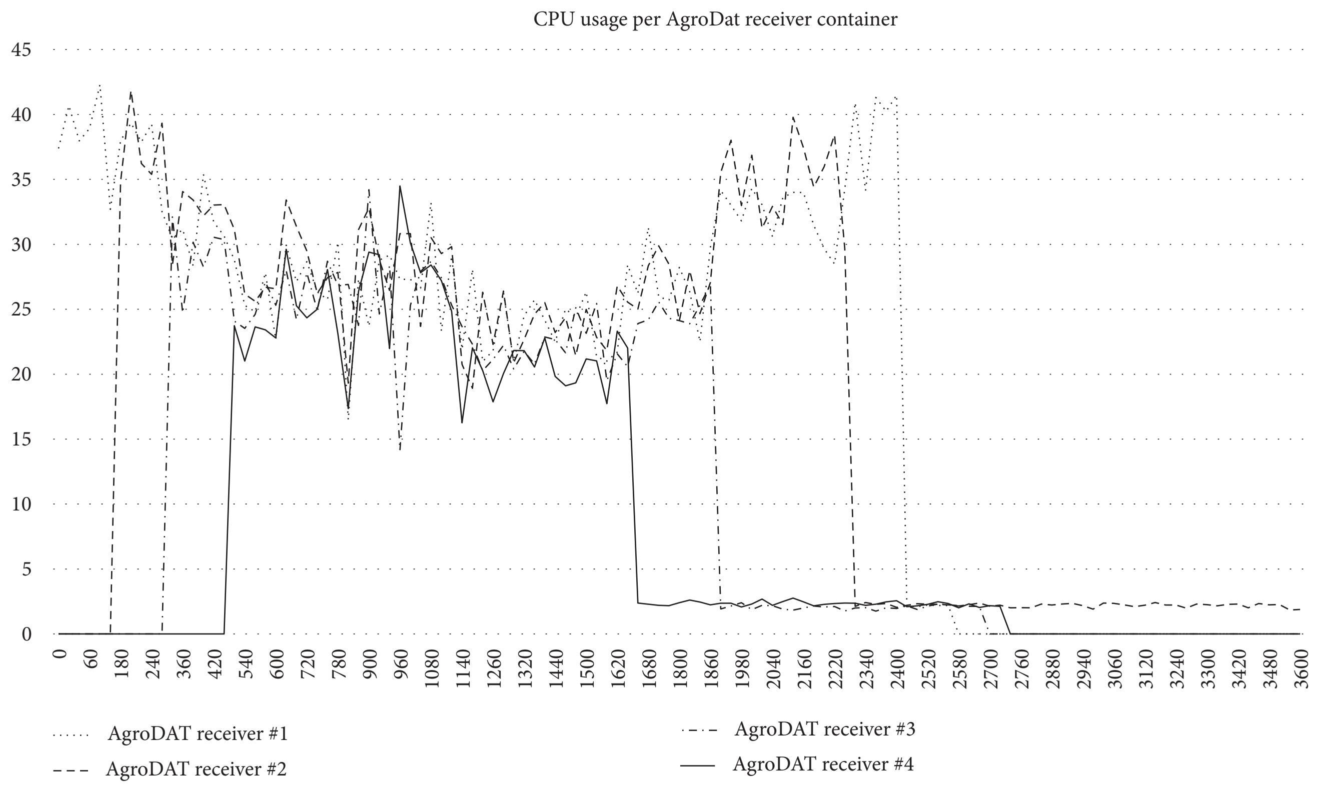
Figure 12: Control loops applied for multilevel autoscaling of virtual machines and containers in MiCADO.

The goal of MiCADO control loops is to provide an automatic scaling functionality for an infrastructure built by microservices. For automatic scaling, there are several different scenarios in which scaling can focus on optimizing the running infrastructure for various goals. The execution of the microservice infrastructure has different requirements and different measurable characteristics. For example, processing, memory, network bandwidth, and disk i/o are all resources MiCADO may reserve for the infrastructure while CPU load, memory usage, response time, or disk usage are measurable characteristics. Beyond optimizing for some of the characteristics, MiCADO is also being developed towards optimizing for costs generated by the usage of (commercial) cloud resources.

Beyond optimizing for easily measurable external characteristics, MiCADO is prepared to monitor some internal parameters of the microservice infrastructure. For example, monitoring the length of a queue enables MiCADO to perform optimization in different scenarios like keeping the

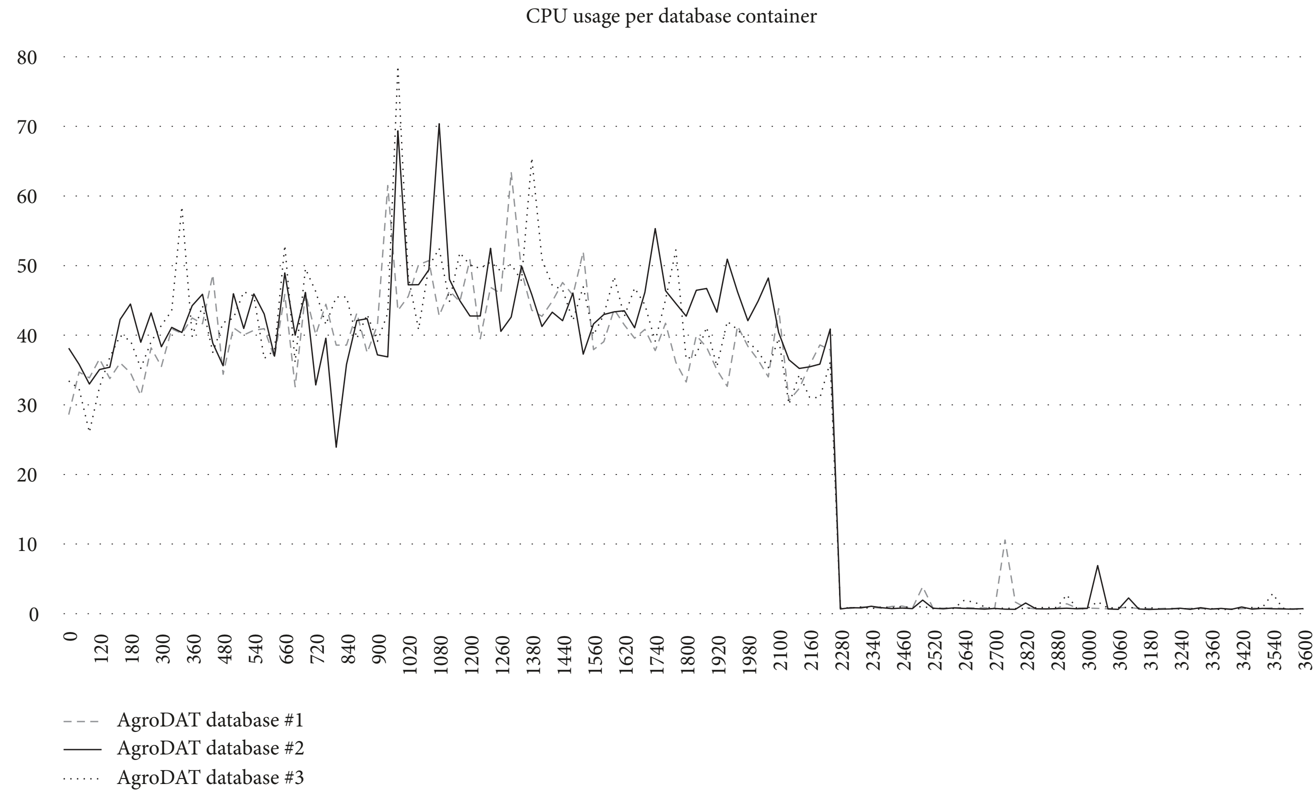


(a) CPU usage per node



(b) CPU usage per data receiver container

Figure 13: Continued.



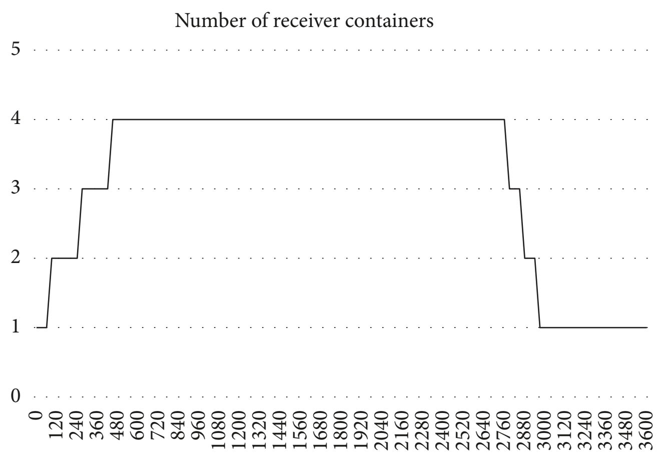
(c) CPU usage per database container

Figure 13: CPU usage per container.

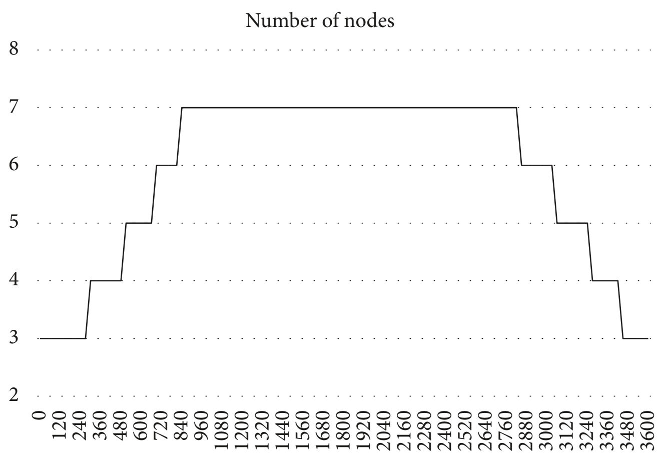
number of items on a certain level, keeping a predefined processing rate of items, or making the items consumed by a predefined deadline. The different scenarios and optimization strategies are continuously developed and added to the latest version of MiCADO. The current version of MiCADO (v3) supports the performance-based policy for containers and virtual machines.

*4.2.2. Results.* The multilevel scaling of the back-end is handled by MiCADO. With MiCADO's dual control loops, we can scale the container-based data collectors and the host virtual machines as well. The whole data collector infrastructure is deployed in MiCADO.

All measurements were performed on an OpenNebula-based cloud within MTA SZTAKI. All nodes ran as virtual machines on AMD Opteron 6376 CPUs with all virtual CPUs (VCPUs) mapped to actual physical cores and connected via a 10 Gbit/s network. The virtual machines had 1 VCPU and 2 GB RAM allocated. The measured data are exported from MiCADO's Prometheus monitoring component. MiCADO was configured to scale up or down the containers in every minute and the workers nodes in every third minute. We only scaled the collector components automatically within the architecture. The collector components are scaled up when the average CPU usage of the collectors reaches 30 percent. This allows the saturation of the worker nodes and distribution of the incoming data between additional collector containers. The collectors are scaled down at 10-percent average CPU usage. The MiCADO worker nodes are scaled up at 60-percent CPU usage and down at 20-percent CPU usage. Measurements were performed with test sensor data, and we generated and sent 800 data packages per second to the framework. We deployed the collector infrastructure initially with one collector and three database containers. As shown in Figure 13(c), as the collectors' average CPU usage reaches the threshold, MiCADO scaled up the collector containers, and the incoming data was distributed between the increased numbers of collector containers. The number of the collector components can be seen in Figure 14(a). The balanced CPU usage of the collectors' database components can be seen in Figure 13(c). As seen in Figure 13(a) and in Figure 14(b), when the MiCADO worker nodes' average CPU usage reached the threshold, MiCADO automatically scaled up the worker nodes. Only node #4 had a high CPU load, because the number of the collector containers was enough to process the incoming data. The remaining running nodes can be considered spares in the system, or alternatively we can manually reschedule the running collector components to balance the worker nodes' CPU usage within the cluster. As shown in Figure 13(c) and in Figure 14(a), when the stream of the incoming data finished, the worker components' CPU usage lowered and MiCADO scaled down the collector's container service to the minimum number of containers. This scaling down can be observed in Figures 13(a) and 14(b), with the MiCADO worker nodes as well.



(a) Total number of data receiver containers



(b) Total number of all containers (nodes)

Figure 14: Number of containers.

## 5. Future Works and Conclusions

This section summarizes further the targeted use cases: (i) connected cars (see Section 5.1) and (ii) precision agriculture (see Section 5.2), and contains conclusions for the paper. In both (connected cars and precision agriculture) CPS areas, a subset of the presented back-end framework has been already applied and integrated successfully with other system components particularly for research and evaluation purposes and also for forming the baseline of new production-level services.

As additional future work, we have started to study and elaborate the adaptation of different job-based policies, including deadline and throughput, in MiCADO over the CQueue microservice infrastructure. The integration will lead to an autoscalable CQueue job execution framework with different strategies on scaling. Furthermore, the adaptation of this sensor data ingestion architecture is already in progress in two further sectors, namely, connected cars and precision farming, with some positive preliminary results based on the outlined sectoral demands.

*5.1. Connected Cars.* Connected car technologies have been rapidly advancing with several new digital solutions and autonomous driving features. Connected cars collect and make interpretable massive amounts of data—mostly from digital sensors of IoT systems by exchanging useful data (e.g., alerts) between other vehicles, stoplights, and back-end services [33, 46]. Even though automobiles today are equipped with significant amount of processing and storage capacities, the rapidly growing amount of raw data and the higher-level functionalities require robust and scalable back-end technologies that can handle the underlaying sophisticated processing and analytical functions. Relying on the basic principles and some components of the presented sensor data back-end architecture, our current research focuses on CAN data collection, remote device flashing, Eco-driving, and weather report and forecast with some promising initial achievements (see details in [47]).

*5.2. Precision Agriculture.* The ultimate aim of the AgroDat.hu project [48] is to create a knowledge center for precision agriculture based on the local sensor data (and also integrating semi- or unstructured data from international repositories). Concerning the sensors, more than 1000 complex sensor pillars have been deployed at various selected locations covering more than 8000 hectares of 58 farmers. The sensor pillars have modular structure [49] with facilities to measure environmental factors (weather, soil moisture, etc.), phenotypes (sensor image data), and other parameters continuously for at least 3 years. The communication network is based on a 4G GSM network and M2M communication-enabled SIM cards. For processing and storing data and also for providing services for researchers, farmers, decision-makers, and so on, a new big data center is in operation based on the OpenStack [50] cloud. It is responsible for providing an elastic and flexible framework for the higher-level software services: (among others) back-end for data collection, processing, and decision support systems.

The back-end architecture of AgroDat.hu contains the two main functionalities: (i) data collectors for processing incoming sensor and sensor image data and (ii) a Backoffice system for additional functionalities and administrative functions. The obtained raw data is then made available for processing and analytics in the big data center.

(1) First, the data collectors are responsible for gathering and storing sensor messages in the cloud for further processing. They preprocess the data and store it in a structured format for the other cloud-based functions. Additionally, it is also stored directly in the input format to have backup for security reasons and to be available for future implemented functions and statistics. (2) The collected data can be visualized within a *Backoffice* application and is also available for further processing by analytical tools. The detailed description of these results can be found in [51].

*5.3. Conclusions.* In this paper, we presented different variants (based on orchestrated VMs and containers) and also the major implementation steps of a scalable sensor data back-end and predictive simulation architecture that can be adapted with low-entry barriers to other use case scenarios as well.

According to the evaluation of the orchestrated back-end framework, the solution is highly scalable and the back-end facilitates the transfer of the results achieved in the field of digital factory (DF), namely, the DES solution presented in the paper, by allowing much faster, parallelized behavior forecasting of manufacturing systems. We strongly believe that the orchestrated cloud and container-based back-end platform support industrial applications by providing the required up-to-date and cutting-edge ICT technologies.

The novelties of our cloud orchestration solution are mostly described in [35]. However, the combination of the key features such as (i) multilevel autoscaling including VMs and containers, (ii) cloud agnostic approach, and (iii) generic open-source solutions for such wide scope including various CPS problems makes our solution innovative.

We are just in the initiating phase of creating and operating the Centre of Excellence in Production Informatics and Control (EPIC CoE), and the integrated solution presented in the paper already allows us to offer complex, powerful, and affordable Industry 4.0 solutions to all stakeholders, especially to SMEs. Moreover, we recently started the innovation and knowledge transfer activities in the "Cloudification of Production Engineering for Predictive Digital Manufacturing" (CloudiFacturing) consortium [52] in order to adapt the presented sensor data back-end and predictive simulation architecture by the help of digital innovation hubs across Europe.

## Conflicts of Interest

The authors declare that they have no conflicts of interest.

## Acknowledgments

This work was partially funded by the European "COLA (Cloud Orchestration at the Level of Application)" project (Grant Agreement no. 731574 (H2020ICT-2016-1)), by the National Research, Development and Innovation Fund of Hungary (Grant no. VKSZ 12-1-2013-0024) (Agrodat.hu), and by the International Science and Technology Cooperation Program of China (Grant no. 2015DFE12860). On behalf of the project Occopus, the authors thank for the usage of the MTA Cloud [53] that significantly helped us achieve the results published in this paper. The research conducted with the scope of the discrete-event simulation was supported by the European Commission through the H2020 project EPIC (http://www.centre-epic.eu) (Grant no. 739592).

## References

[1] L. Monostori, B. Kádár, T. Bauernhansl et al., "Cyber-physical systems in manufacturing," *CIRP Annals*, vol. 65, no. 2, pp. 621–641, 2016.

[2] C. Kardos, G. Popovics, B. Kádár, and L. Monostori, "Methodology and data-structure for a Uniform System's Specification in Simulation Projects," *Procedia CIRP*, vol. 7, pp. 455–460, 2013.

[3] A. Gupta, M. Kumar, S. Hansel, and A. K. Saini, "Future of all technologies-the cloud and cyber physical systems," *International Journal Of Enhanced Research In Science Technology & Engineering*, vol. 2, p. 2, 2013.

[4] I. Mezgár and U. Rauschecker, "The challenge of networked enterprises for cloud computing interoperability," *Computers in Industry*, vol. 65, no. 4, pp. 657–674, 2014.

[5] R. Gao, L. Wang, R. Teti et al., "Cloud-enabled prognosis for manufacturing," *CIRP Annals*, vol. 64, no. 2, pp. 749–772, 2015.

[6] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (IoT): a vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[7] "AWS Internet of Things," October 2017, http://aws.amazon.com/iot.

[8] "Azure IoT Suite - IoT Cloud Solution," October 2017, http://www.microsoft.com/en-us/internet-of-things/azure-iot-suite.

[9] "Google IoT Core," October 2017, http://cloud.google.com/iot-core.

[10] "FIWARE architecture description: big data," October 2017, https://forge.fiware.org/plugins/mediawiki/wiki/fiware.

[11] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Real-time Data Systems*, Manning Publications Co., 2015.

[12] "Highly scalable blog. In-stream big data processing," October 2017, https://highlyscalable.wordpress.com/2013/08/20/in-stream-big-data-processing/.

[13] "FIWARE architecture description: big data," October 2017, https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE.ArchitectureDescription.Data.BigData.

[14] "FIWARE glossary," October 2017, https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE.Glossary.Global.

[15] "Hadoop: WebHDFS REST API," October 2017, http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/WebHDFS.html.

[16] M. Kornacker, A. Behm, V. Bittorf et al., "Impala: a modern, open-source SQL engine for Hadoop," in *7th Biennial Conference on Innovative Data Systems Research (CIDR'15)*, Asilomar, CA, USA, January 2015.

[17] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, Incline Village, NV, USA, May 2010.

[18] "The Apache Cassandra project," October 2017, http://cassandra.apache.org.

[19] "Cascading: application platform for enterprise big data," October 2017, http://www.cascading.org.

[20] G. Casale, D. Ardagna, M. Artac et al., "DICE: quality-driven development of data-intensive cloud applications," in *2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering*, pp. 78–83, Florence, Italy, May 2015.

[21] "The Apache Spark project," October 2017, http://spark.apache.org.

[22] "The Apache Storm project," October 2017, http://storm.apache.org.

[23] V. Abramova and J. Bernardino, "NoSQL databases: MongoDB vs Cassandra," in *C3S2E '13 Proceedings of the International C∗ Conference on Computer Science and Software Engineering*, pp. 14–22, Porto, Portugal, July 2013.

[24] "Docker-build, ship, and run any app, anywhere," October 2017, https://www.docker.com/.

[25] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[26] "OpenLambda," October 2017, https://open-lambda.org/.

[27] "OpenFaaS: Functions as a Service," October 2017, https://blog.alexellis.io/introducing-functions-as-a-service/.

[28] "A Kubernetes native serverless framework," October 2017, http://kubeless.io/.

[29] "Terraform by HashiCorp," October 2017, https://www.terraform.io/.

[30] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "TOSCA: portable automated deployment and management of cloud applications," *Advanced Web Services*, pp. 527–549, 2013.

[31] "OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA)," October 2017, https://www.oasis-open.org/committees/tosca.

[32] "OASIS - advancing open standards for the information society," October 2017, https://www.oasis-open.org/.

[33] T. Haberle, L. Charissis, C. Fehling, J. Nahm, and F. Leymann, "The connected car in the cloud: a platform for prototyping telematics services," *IEEE Software*, vol. 32, no. 6, pp. 11–17, 2015.

[34] "GE. Predix platform: the foundation for digital industrial applications," March 2018, https://www.ge.com/digital/predix-platform-foundation-digital-industrial-applications.

[35] J. Kovács and P. Kacsuk, "Occopus: a multi-cloud orchestrator to deploy and manage complex scientific infrastructures," *Journal of Grid Computing*, vol. 16, no. 1, pp. 19–37, 2017.

[36] "HAProxy: the reliable, high-performance TCP/HTTP load balancer," October 2017, http://www.haproxy.org/.

[37] "The uWSGI project. uWSGI 2.0 documentation," October 2017, https://uwsgi-docs.readthedocs.io.

[38] J. Gardner, "The Web Server Gateway Interface (WSGI)," *The Definitive Guide to Pylons*, pp. 369–388, 2009.

[39] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[40] "The Prometheus monitoring tool," November 2017, https://prometheus.io/.

[41] International Society of Automation, "ISA95, enterprise-control system integration," November 2017, https://www.isa.org/isa95/.

[42] R. Bloomfield, E. Mazhari, J. Hawkins, and Y.-J. Son, "Interoperability of manufacturing applications using the Core Manufacturing Simulation Data (CMSD) standard information model," *Computers & Industrial Engineering*, vol. 62, no. 4, pp. 1065–1079, 2012.

[43] T. Kiss, P. Kacsuk, J. Kovacs et al., "MiCADO—Microservice-based Cloud Application-level Dynamic Orchestrator," *Future Generation Computer Systems*, 2017, http://www.sciencedirect.com/science/article/pii/S0167739X17310506.

[44] "COLA: Cloud Orchestration at the Level of Application," November 2017, http://www.project-cola.eu.

[45] "Swarm mode of Docker," November 2017, https://docs.docker.com/engine/swarm/.

[46] W. He, G. Yan, and L. Da Xu, "Developing vehicular data cloud services in the IoT environment," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1587–1595, 2014.

[47] C. Marosi, R. L. Attila, A. Kisari, and S. Erno, "A novel IoT platform for the era of connected cars," in *2018 IEEE International Conference on Future IoT Technologies (Future IoT)*, pp. 1–11, Eger, Hungary, January 2018.

[48] "Agrodat.hu project website," October 2017, http://www.agrodat.hu.

[49] P. Gábor, S. Péter, and É. Gábor, "Power consumption considerations of GSM-connected sensors in the AgroDat.hu sensor network," *Sensors & Transducers*, vol. 189, no. 6, pp. 52–60, 2015.

[50] X. Wen, G. Gu, Q. Li, Y. Gao, and X. Zhang, "Comparison of open-source cloud management platforms: OpenStack and OpenNebula," in *2012 9th International Conference on Fuzzy Systems and Knowledge Discovery*, pp. 2457–2461, Sichuan, China, May 2012.

[51] C. Marosi, A. F. Attila, and R. Lovas, "An adaptive cloud-based IoT back-end architecture and its applications," in *Proceedings of The 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2018)*, Cambridge, UK, March 2018.

[52] "CloudiFacturing project website," May 2018, http://cloudifacturing.eu.

[53] "MTA Cloud," October 2017, https://cloud.mta.hu.