DRAFT

Industrial Control via Application Containers: Maintaining determinism in IAAS

Florian Hofer[1,3*] | **Martin Sehr**[2] | **Alberto Sangiovanni-Vincentelli**[3] | **Barbara Russo**[1]

[1]Faculty of Computer Science, Free University of Bolzano-Bozen, Bolzano, 39100, Italy

[2]Corporate Technology, Siemens Corporation, Berkeley, CA, 94704, USA

[3]EECS, University of California, Berkeley, CA, 94720, USA

Correspondence
Florian Hofer, Faculty of Computer Science, Free University of Bolzano-Bozen, Bolzano, 39100, Italy
Email: florian.hofer@stud-inf.unibz.it

Funding information
iCyPhy, through Siemens Corporation, Corporate Technology

Industry 4.0 is changing data collection, storage, and analysis in industrial processes fundamentally, enabling novel applications such as flexible manufacturing of highly customized products. However, real-time control of these processes has not yet realized its full potential in using the collected data to drive further development. Indeed, typical industrial control systems are tailored to the plant they need to control, reusing, and adapting to challenge. In the past, the need to solve plant-specific problems overshadowed the benefits of physically isolating a control system from its plant. We believe that modern virtualization techniques, specifically *application containers*, present a unique opportunity to decouple control from plants. This separation permits us to fully realize the potential for highly distributed and transferable industrial processes even with real-time constraints arising from time-critical subprocesses. This paper explores the challenges and opportunities of shifting industrial control software from dedicated hardware to bare-metal servers or (edge) cloud computing platforms using *off-the-shelf* technology, i.e., technologies commercially available. We present a migration architecture and show, using a specifically developed orchestration tool, that containerized applications can run on shared resources without compromising scheduled execution within given time constraints. Through latency and computational performance experiments, we explore three system setups' limits and summarize lessons learned.

KEYWORDS
Industrial Control Systems, Real-Time, IAAS, Container orchestration, Determinism

1 | INTRODUCTION

Emerging technologies such as the Internet of Things and Cloud Computing offer the chance to innovate structure and control of industrial processes. These new technologies allow the creation of flexible production systems, for

which control is performed through distributed sensing, big-data analysis, and cloud storage. Such systems may also take advantage of new computing paradigms like the Edge Networking paradigm or the Fog Computing, which brings data storage and computation as close as possible to the point of need (e.g., a gateway) [1, 2]. In this cloud-to-things continuum [3], new technologies and a new paradigm of computation and data transmission and storage offer an opportunity for innovation of the industrial system and processes like never before. This opportunity is the essence of the fourth industrial revolution, i.e., Industry 4.0 (I4.0) [2]. However, the control of industrial processes has not yet fully embraced such a revolution, and there are reasons for it. Firstly, control systems in industrial processes require timeliness and robustness imposed from the production chain, making maintaining formal guarantees or implementing changes difficult. For instance, control software for industrial processes must respond within specific time limits set by the physical systems they control. New technologies and paradigms need to comply with such requirements. Secondly, moving control tasks to Edge or Fog networks may require dealing with heterogeneous delay sensitivities of the task and network delays when connectivity is not appropriate to meet real-time deadlines [3]. Thus, such systems may need specific care in task scheduling.

On the other hand, modern virtualization techniques can efficiently use cloud computing resources in industrial control systems and reduce operational cost and production downtime [4]. These include virtualization techniques that create virtual instances of a computer system, containerization techniques that create virtual instances of individual processes and serverless applications that run as Functions as a Service [5, 6, 7]. Using such technologies allow time-machines (i.e., engines that perform snapshots of control software and machine state), control redundancy (i.e., parallel operation of containers and virtual server instances [8]), and online system reconfiguration (i.e., reprogramming of control algorithms and product specifications with little or no downtime [9]). In particular, containers simplify the parallel execution of control software on devices such as PLCs and, to a lesser extent, on sensing and actuating field devices. They allow applications such as performance and distributed health monitoring to run on a shared end node [10, 11] and can host digital twins (i.e., digital replicas of physical components) to predict malfunction, maintenance intervals, and tool lifespan [12]. Virtualization, containers, and serverless applications yield to the creation of light and easily distributed control applications able to run on any system, and that are, at the same time, easy to maintain and update [13]. These technologies can increase the reliability and robustness of control systems while enabling self-* properties, through which systems can automatically maintain themselves throughout different scenarios (i.e., Self-aware, Self-predict, Self-compare, Self-configure, Self-maintain, Self-organize) [8].

All these advantages do not come at zero cost. Deploying an efficient infrastructure for control systems that considers real-time constraints and exploits the new technologies and principles of Industry 4.0 is not an easy task for control engineers and developers. Being able to select and master all these technologies in the implementation, deployment, and maintenance of such novel control infrastructures may appear too demanding for control engineers and developers of small and medium-size companies [14]. Thus, migrating to such a light and distributed architecture may not be a viable solution for many companies.

This work aims to *propose a novel control systems architecture that exploits virtualization and containerization of bare-metal host or cloud environments to build light and easily distributed control applications that can easily follow the principles of Industry 4.0 and use off-the-shelf technologies.* To illustrate the viability of such novel architecture, we have performed two series of experiments with different architecture instances that make use of off-the-shelf technologies: a set of latency tests (Section 6.2.1) and a set of performance tests (Section 6.3). The former compares bare-metal and hardware virtualization solutions and chooses the most suitable for implementing control systems' virtualization—the latter tests for resource optimization using container orchestration. Thus, the contributions of this paper are the following:

- An architecture for virtualization of control systems;
- A scheduling schema to statically allocate and monitor containers and their resources;
- A set of tests to trade-off resource efficiency with real-time task scheduling for containerization and validate the architecture.

The rest of this paper is structured as follows. Sections 2 and 3 analyze related work and background to motivate our analysis. Section 4 describes the research questions and introduces a preview of our approach. In Section 6, we illustrate the design of the sets of experiments and present their results. Section 5 proposes an architectural solution based on reference models for Industry 4.0, implemented through the orchestration of containers and validated by the results of the tests. Finally, we discuss lessons learned and conclude in the last two Sections.

2 | CONTROL PROGRAM EXECUTION

Our work pertains to two major research areas that focus on different aspects of control program execution: cloud and high-performance computing and control containerization. The former focuses on lowering its latency and gives less importance to its execution determinism. The latter tries to reshape its run-time environment and, thus, to create a level of independence to its underlying hardware. During this redesign, determinism stays in focus, leaving system virtualization in the background. The resulting combination of *containers executed on cloud resources* and *strictly time-dependent control application containerization* constitutes a new challenge that can be coped with applying insights from both fields. Such a combination requires an operating system kernel that supports and *exceeds* soft real-time guarantees secured by low latency kernel flavors in use on HPC installations while keeping only limited environmental control. In this paper, we assess the feasibility of this approach using off-the-shelf technology.

In 2014, Garcia-Vallas et al. [15] analyzed challenges for predictable and deterministic cloud computing. Even though they focus on soft real-time applications, specific aspects and limits apply to any real-time system. Merging cloud computing with real-time requirements is challenging: the guest OS has only limited access to physical hardware and thus suffers from the unpredictability of non-hierarchical scheduling and thick stack communications. While real-time enabled hypervisors such as the para-virtualized RT-Xen manage virtual instances with direct access to hardware, the shared resources still suffer from latency that may make real-time execution impossible. Hallmans et al. [16] draw similar observations, but they reach different conclusions. They conclude that it is possible to move a complete soft real-time system into the cloud; the authors see an upcoming development that further allows for real-time systems. Many latency performance evaluations confirm this possibility. Nonetheless, to our knowledge, no one has verified the proper execution of real-time tasks within deadlines.

Containerizing of control applications is a recent concern. Moga et al. [5] presented the concept of containerization of full control applications to decouple the hardware and software life-cycles of an industrial automation system. The authors propose OS-level virtualization as a suitable technique to cope with automation system timing demands. They propose two approaches to migrate a control application into containers on top of a patched real-time Linux-based operating system: a) a given system is decomposed into subsystems, where a set of sub-units performs a localized computation, which then is actuated through a global decision-maker, or b) Devices are defined as a set of processes, where each process is an isolated standalone solution with a shared communication stack, and based on this, systems are divided into specialized modules, allowing a granular development and update strategy. The authors demonstrate the feasibility of real-time applications with containerization, even though they express concern about the technical solution's maturity. Goldschmidt and Hauk-Stattelmann in [7] perform benchmark tests on modularized

industrial Programmable Logic Controller (PLC) applications. This analysis examines the impact of container-based virtualization on real-time constraints. As there is no solution for PLCs' legacy code migration, the migration to application containers could extend a system's lifetime beyond the physical device's limits. Even though tests showed a worst-case latency in the order of $15ms$ on Intel-based hosts, the authors argue that the container engines may be stripped down and optimized for real-time execution. In follow-up work, Goldschmidt et al. [17], a possible multi-purpose architecture, was described and tested in a real-world use case. The results show the worst-case latency of about $1ms$ for a Raspberry PI single-board computer, making the solution viable for cycle times of about $100ms$ to $1s$. The authors state that memory overhead, containers' restricted access, and problems due to technology immaturity are still to be investigated. Tasci et al. [6] address architectural details not discussed in [7] and [17]. These additions include the definite run-time environment and how to achieve deterministic communication of containers and field devices in a novel container-based architecture. They proposed a Linux-based solution as a host operating system, including both single kernel preemption-focused PREEMPT-RT patch and co-kernel oriented Xenomai. With this patch, the approach exhibits better predictability, although it suffers from security concerns introduced by sensitive system files required by Xenomai. For this reason, they suggested limiting its application for safety-critical code execution. They analyzed and discussed inter-process messaging in detail, focusing. Finally, they implemented an orchestration run-time, managing intra-container communication, and showed that task times as low as $500\mu s$ are possible.

The three solutions discussed above *are all based on a bare-metal solution*. Although these solutions represent the first essential step for reallocating an embedded control software onto a dedicated infrastructure (i.e., virtualizing the control units), showing that containerization of real-time applications is viable, they remain limited to physical hardware execution.

3 | BENCHMARKS IN HARDWARE VIRTUALIZATION AND OS CONTAINERIZATION

Felter et al. in [18] study the performance of instances based on hardware virtualization via Kernel-based Virtual Machines (KVMs) and container OS-virtualization using the cross-platform capable Docker. The authors state that Docker results in equal or better performance than KVMs in almost all cases. Arango et al. [19] analyze three containerization techniques for use in cloud computing. The paper compares Canonical's Linux Containers (LXC), Docker, and Singularity, an engine developed by Lawrence Berkeley National Laboratory, to a bare-metal application. In many aspects, the Singularity containers performed better, sometimes even better than the bare-metal implementation. However, this is due primarily to the engine's blended approach; namely, Singularity is a partial virtualization solution since it grants access to I/O operations without context changes. We will discuss the containerization techniques and selection of candidates in Section 6.2. A recent work by Telschig et al. [1] explores a platform-independent container architecture for real-time systems. The authors identify mixed-criticality, cross-platform operation, and third-party software as the main reason for developing new architectures. Their proposal manages communication between this dependent distributed software through an architecture. This architecture focuses on the isolation of critical from non-critical tasks and portability. The presentation concludes with the introduction of a prototype agent. Abeni et al. [20] attempt to extend the Linux standard scheduler to respond better. Their work details the Complete Fair Scheduler hierarchical extension with a deadline-based algorithm optimizing latency results for containerized software. The modified scheduler successfully manages a more considerable amount of time-critical tasks, performing better than the default deadline-based scheduler.

Overall, containerization has shown powerful enough to be a resource economic replacement for traditional vir-

tualization techniques. However, a performance investigation with real-time applications remains due; scheduling techniques like those presented in [20] further prove that there is still room for improvement.

4 | RESEARCH QUESTIONS AND APPROACH PREVIEW

While flexibility and efficiency are big advantages, smart systems display increased running costs. New architectures suggested for Industry 4.0 mostly provide distributed and decentralized control. On the other hand, control layers have to carry out more complex tasks resulting in a high amount of small distributed supervision loops on the hardware. In turn, this would increase maintenance and operation costs.

Virtualization of control units, i.e., abstracting function from hardware[1], allows up-scaling the installed computation appliances. Such a unit can run on shared hardware exploiting cost reduction advantages. Besides, the virtualization of control units can be further extended by virtualizing the computing machines and distributing the function over containers. This up-scaling has to use standard hardware and software to keep a low maintenance profile. The market offers an increasing number of readily off-the-shelf software solutions. Unfortunately, identifying the solution that fits the specific real-time constraint is often a challenging problem for practitioners. Therefore, one of the goals of this work is to study whether, and to which extent, off-the-shelf technology can be used for hard real-time applications migrated to a virtualized computing resource. Specifically, we propose an *architecture based on off-the-shelf technologies for the containerization and virtualization of control systems*, Fig. 3 in Section 5. Its design can be used as a template to ease transitions to a containerized setup and shared instances, enabling advanced features for novel industrial control systems.

Control software is usually characterized by one or more real-time tasks with periodic executions and computation deadlines. In literature, three types of periodic real-time applications have been studied: Soft, where computation value decreases with a deadline overshoot; Firm, when exceeding the maximum delivery time nulls the computation value; or Hard, where a missed deadline may have catastrophic consequences [21]. Missing deadlines is a collective matter as a task that exceeds its timing limits may further impede other tasks' execution. The delayed scheduler yield consumes additional resources that may cause a bottleneck, and follow-up tasks may not maintain their deadlines. Eq. 1 shows the estimation of a single run of the total required computation time c_i of a periodic real-time task i.

$$c_i = f_i + r_i = f_i + t_i + \sum_{j=1}^{N} n_{ij} + \sum_{k=1}^{M} m_{ik} \tag{1}$$

f_i is the wake-up or firing time (latency), and r_i is the total run-time. The former captures the time spent between the period start and the execution start of a task. Its measurement includes task switching times and delays due to higher priority tasks and interrupts served. The latter expresses the actual used computation time t_i and task interaction (n_{ik}) and environment-induced noise (m_{ij}). Tasks interaction noise includes interruptions by higher priority tasks, task inter-process communication (IPC), and I/O waits and latency due to missed pages. Environment noise includes hypervisor delays and hardware or (kernel) software interrupts and network delays. As the characteristics of the tasks and execution environment that may determine a noise (n or m in Eq. 1) are not always known in advance, the estimation of such noise in some cases may be difficult if not impossible. This restriction particularly impacts deterministic (hard) real-time systems that must know in advance that every task i will complete within its deadline d_i, [22, 23]. In this work, we aim to assess alternative architectural configurations for a control system migrated to a

[1] Worth noticing that virtualization of control systems does not imply the use of virtual machines or containers.

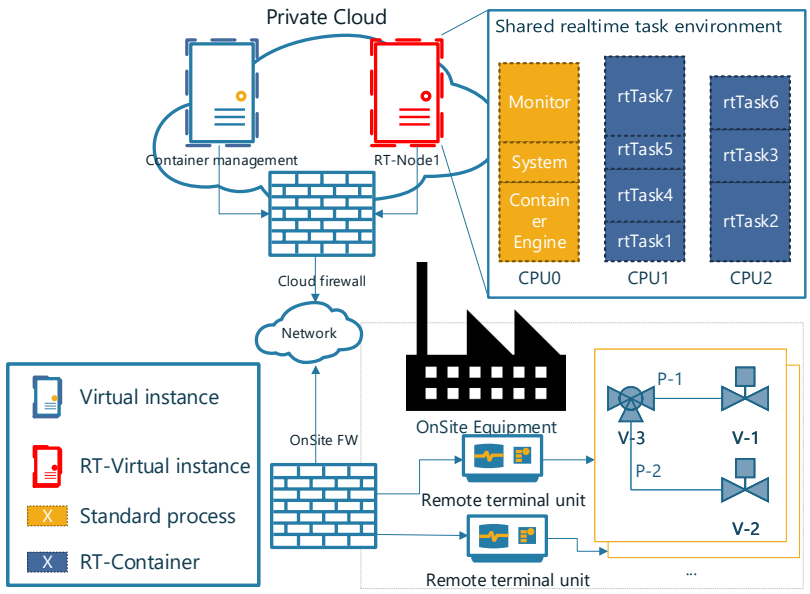

FIGURE 1 Application example: A cooling and auxiliary regulation system configuration for a gas turbine migrated with the real-time enabled cloud. RT-Node1 monitors and handles the on-premises installation.

virtualized and containerized environment while maintaining the original level of the system's determinism[2] running in a bare-metal environment.

As an illustrative example, Fig. 1 depicts a distributed facility that controls the auxiliary and cooling systems of a group of thermo-electric gas-turbines. The control system has been migrated to a software system with shared resources and application containerization. The control components have been separated from the on-site remote terminal unit and run in a shared, two- instances virtualized private cloud environment, [16]. The real-time capable virtualization instance (RT-node1 on the right in the private cloud) acts as an intermediary between Monitoring and Management and the on-premises end-terminals. Control software is divided into multiple independent binaries, isolated and adapted to run on a standard Linux system [5]. In these settings, the application refresh rate, or periodicity p, must not exceed the expected maximum round-trip time between the remote terminal unit and the cloud of $100ms$. For instance, if the system software exhibits the worst-case computing time of $10ms$ and each instance uses assigned CPU and memory exclusively, the remaining CPU time of $90ms$ is spent in idle, which is a high resource waste. Thus, *sharing the spare CPU-time can improve resource utilization, and placing multiple containers on the same resources can reduce the required system size and running costs.* In principle, such an approach enables flexible resource management and may reduce infrastructural cost [16]. Thus, to evaluate the level of task-optimization required for the proposed architecture to meet the control unit's deadlines (Eq. 1), we performed a series of experiments. Specifically, we will answer the following research questions:

[2] For "maintaining" we mean that we experimentally observe the same determinism

RQ1: *What are the viable off-the-shelf system configurations that make resource sharing through containers viable?*
We have seen in Eq. 1 that the achievable amount of resource sharing depends on the system and concurrent running tasks. This research question intends to investigate the responsiveness of possible candidate off-the-shelf alternative solutions. Systems that prove a low and stable firing time, f_i, fulfill a vital prerequisite to achieve determinism in a shared context.

RQ2: *What is the achievable level of CPU sharing with a standard real-time enabled kernel?*
While a constant f_i describes the strength of reaction, the actual CPU load shows the variability of task run-times, r_i. Thus, monitoring software run-time on shared CPUs displays the impact of task interaction and operating system, I/O and virtualization delays on the programmed run-time t_i.

To answer the questions, we performed two sets of experiments as illustrated in Fig. 2. Firstly, we study latency

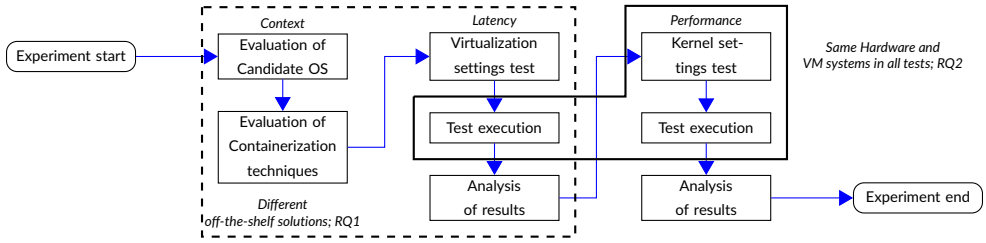

FIGURE 2 Overview of the approach for the evaluation of alternative architectural configurations and performance optimization of the chosen solution

behavior with a shared resource on multiple hardware and system setups. Operating systems and container engines and their configurations are selected with latency stress tests. Such tests show how task reaction times change with varying configurations and system load, RQ1. Secondly, we run a set of experiments on the best solution found in RQ1 to analyze performance and determinism with different loads. We further explore computation time stability, delays, and occurring deadline misses if more than one task runs on the same resource through resource allocation. Isolated CPU performance tests allow us to remove confounding factors and have a recommendation on the upper sharing boundary, RQ2. With such sets of experiments, we aim at identifying the parameters of Eq. 1. The latency tests focus on grasping a task's firing time, f_i, using a low footprint capturing software and logs. First, we test in idle, second using stress on CPU, I/O, and random memory access. The performance tests try to identify the noise the OS, the hypervisor, and concurrently running system tasks cause, n_ij in Eq. 1. Section 6 details the approach and answers the research questions.

5 | REAL-TIME SMART SYSTEM ARCHITECTURE

Our architecture aims at contributing to the current debate within the systems engineering research community on strategies for designing flexible systems' architecture ([24]) where flexibility is defined as how easily a system's capabilities can be modified in response to external change, [25]. In this respect, our work proposes a flexible architecture for control systems that exploit virtualization and containerization capabilities over the cloud, designs it as a leveled model for the real-time system in the cloud ([16, 26]), and finally, is deployed exploiting off-the-shelf technologies. The way we decoupled control from the machine and move it to the cloud makes the overall system highly modular

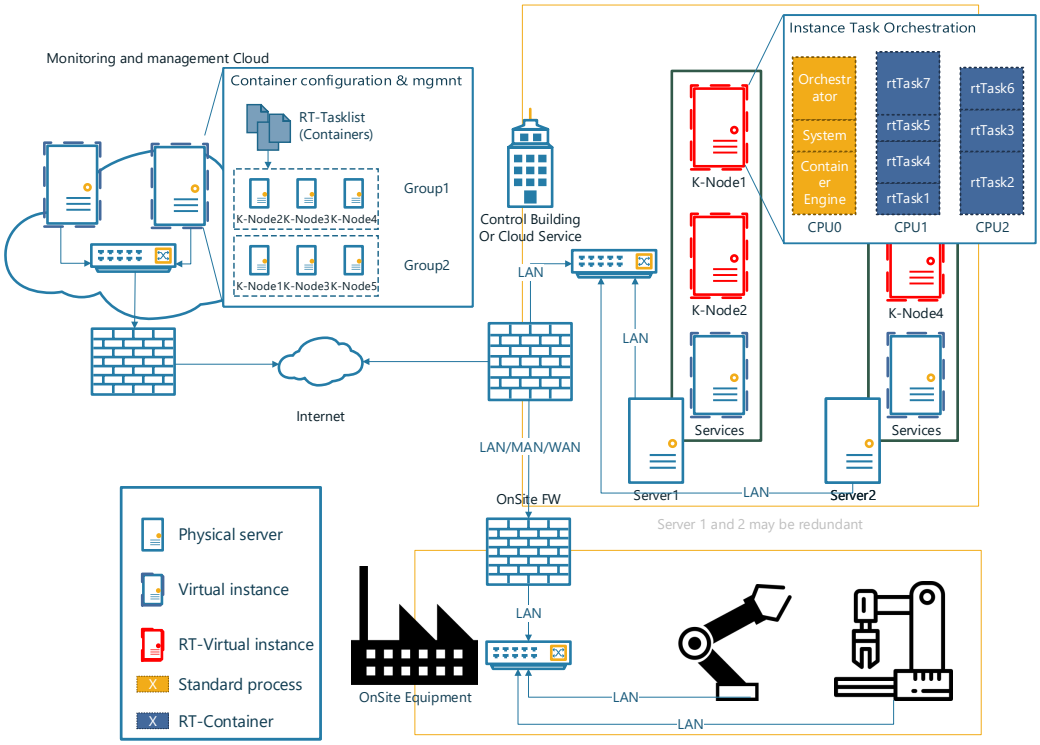

FIGURE 3 The proposed architecture for Container-based Virtualized Industrial Control Systems. The Monitoring and Management Cloud monitors the system and deploys containers in the Control Cluster. The physical servers dedicated for control tasks operate multiple virtual machines, each hosting several real-time application containers. The Orchestrator is responsible to organize and monitor the run-time load.

as it combines the decoupling mechanism of the leveled model for real-time systems of Hallman *et al.* ([16]) and the containerization decomposition in a virtualized cloud system. Furthermore, our approach's deployment is grounded on the concept of flexibility as the incorporation of options or design alternatives into a system ([25]). Specifically, our proposed architecture is deployed and tested under different off-the-shelf technologies. We used latency and performance as Tradespace variables ([27]) to test the deployed architectural alternatives.

5.1 | Overview and Layering

Our architecture extends the concept of the *leveled* model of Hallmans et al. [16] to off-the-shelf technology and managed real-time systems fitting the requirements of virtualized control software. We divide the architecture into three layers (Fig. 3):

- The **Monitoring and Management Cloud** or cloud in Hallmans et al., i.e., services hosted on a cloud or private virtualized dedicated infrastructure;
- The **Control Cluster** or $\mathrm{Real\text{-}time\ Cloud}$, for process control and control-related services;

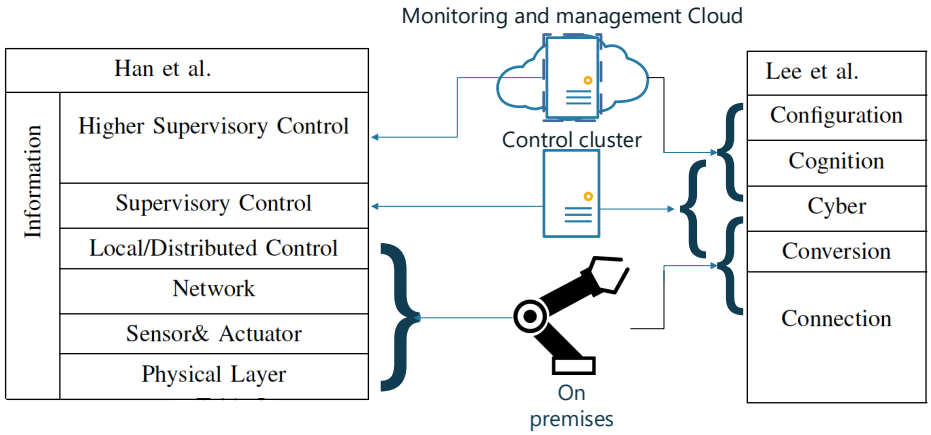

FIGURE 4 Comparison and mapping of layers to the architecture proposal of the two used references, Han et al. [29] and Lee et al. [8].

- The **On-premises Installation** or process, connecting a multitude of heterogeneous devices that interact with the physical world.

The three layers may overlap such that, for example, the control cluster can be part of the cloud or on-premises installation as an internal IAAS infrastructure.

We then use the layer classification and analysis technique of recent work [28] to integrate the model with two other styles. The first integration is the 5C model of Lee et al. [8] on the right of Fig. 3. This model represents a hierarchical distribution of functions, divided into five layers, each representing Industry 4.0 smart property, [8]. The second style is based on the three control levels (Local Control, Supervisory Control, and Higher Supervisory Control) that act as an incentive for control loop division and ease vulnerability investigations [29]. This model is represented on the left of Fig. 3. This separation reflects the division by criticality and timing requirements recommended by Hallmans et al. ([16]). Fig. 4 illustrates the mapping of layers and competencies.

As we illustrate in the rest of this paper, this architecture allows for a better implementation of virtualized control. Through a layered approach, we ease problem identification and handling. It enables detailed assessments such as gradual detection of security issues and adaptation of architectures as carried out within similar heterogeneous environments [30, 28]. In the following sections, we detail the three layers and their mapping to the two reference architecture styles while pointing out the connection to the functions and attributes of Industry 4.0.

5.2 | Management and Monitoring Cloud

The first component of the architecture consists of cloud-based monitoring and management infrastructure and services, top left in Fig. 3. Many of the architectural approaches introduced after the publication of the Industry 4.0 vision include this component as a hub. In this layer, data is globally collected and analyzed, and data-dependent supervision decisions are taken. It performs data acquisition and aggregation from the on-premises devices and analyzes them, for instance, through artificial intelligence tools. The integration in the layer of distributed diagnosis and prognosis frameworks, as proposed in Wu et al. [10], allows for host machine learning processes based on collected and aggregated

plant data. Techniques such as Preventive Health Management (PHM) as a Service, which reduces the maintenance effort for the plant operator by relying on Platform as a Service (PaaS), and Software as a Service (SaaS), can be implemented [11]. Such frameworks and services ultimately enable self-adjustment and self-optimization techniques to reduce production waste and automatically adapt to variations such as mechanical wear (Configuration level, Fig. 4). This cloud layer also hosts a service for container management, providing instruments for appropriate planning, positioning, and execution of real-time containers. Real-time tasks and containers are arranged according to their function and interdependence and deployed on available real-time capable nodes, called K-nodes in Fig. 3. Through the help of client-side agents, the service can update and replace distributed applications during run-time seamlessly. Again, this component enables self-configuration by automatically taking care of the software replacements based on a reconfiguration plan [9]. A system monitoring tool can be paired with a container management tool to verify the container execution. Unlike the above-cited PHM and diagnosis and prognosis frameworks, this monitoring tool aims not to produce surveillance but to monitor K-nodes and virtual servers' status (by monitoring server metrics like CPU usage, memory consumption, I/O, network, disk usage, process, etc.). Extension of the monitoring service with a time series database further allows tracking changes in time, performing data analysis, and applying data-based techniques such as deep learning.

A further service placed at this architecture level is an interface to the human operator. Plant operators in an Industry 4.0 context interact with the system more in decision-making than in an operative role. As such, information is displayed to enable informed operators to make decisions and interventions in production processes. To this aim, features such as simulation and synthesis may optionally be available [31] (for the $\mathrm{Cognition}$ level in the Lee et al. style in Fig. 4). Finally, replicas in the form of a digital twin test reliability and the overall monitoring and supervision of the cloud environment (Cyber level in the Lee et al. style in Fig. 4).

5.3 | Control Cluster

The central element of the architecture is an IAAS infrastructure to host services and processes that have to interact with on-premises devices. Fig. 3 shows an example of a hardware configuration for container-based virtualized industrial control systems. Depending on the system's needs, the represented components are virtual or physical servers that run either in a cloud (virtual instance) or operate in a private (edge) cloud. In the latter case, each server can run more than one virtual instance, obtaining the same resource sharing advantages of a computing cloud infrastructure again. In both cases, the hosted virtual instances can be real-time capable, running control software, or non-real-time capable for further services. The real-time instances, called K-nodes in Fig., run multiple containers managed by the cloud service. A dedicated tool orchestrates system resources at run-time (See Section 5.5). In this environment, each binary of an application can be managed within one container, including constraints and boundaries to ensure operations. For instance, via a time-machine [8], a machine has access to its saved PHM states, consequently its performance indicators and reports. It can compare them with itself and others of a fleet, enabling $\mathrm{self\text{-}comparison}$.

As noted by Telschig et al. [1], the continuous growing demand for extension of control loops with cloud-based analytics (see Section 5.2) requires mixed-critical software components to run on the same system. Thus, in the Control Cluster, a time-critical component runs on fix assigned resources to guarantee timeliness. It is isolated from others that run on a best-effort CPU scheduling policy while still sharing the system's resources. In this setting, non-real-time instances or separately allocated resources can handle such best-effort tasks. The scheduler may reclaim co-located best-effort resources to buffer real-time task resource shortage within the time-frame of a task migration. Non-real-time instances can address other, less critical services. For example, they can run a time-machine or carry the edge computing portion of the health monitoring framework detailed in Wu et al. [10] ($\mathrm{Conversion}$ level, Fig. 4).

The former collects snapshots of real-time control applications to enable peer comparison and similarity analysis, thus promoting self-awareness [8]. The latter operates with redundancy on multiple copies of containers (Cyber level, Fig 4), or the virtual instances themselves can have replicas to increase the system's robustness. Server 2 in Fig. 3 could be a replica of Server 1, ready to take control when the latter fails. As we can have replicas in digital twins, the real-time application can be extended by a model representing the physical device and its environment. The model, fed with sensory input coming from on-premises and interfaced with the running process and human operators, then allows self-comparison and diagnosis, [12].

5.4 | On-premises Installation

The control software connects with the sensing and actuation devices placed on or near the factory's equipment ($\mathrm{Connection}$ level, Fig 4). Depending on the timing and determinism requirements, this connection might need to follow more restrictive protocols. An example of such protocols can be found in the Time-Sensitive Networking (TSN) standards family [32]. However, application-specific needs and physical location set the need for such protocols. Depending on control requirements, including device distance and cycle times, popular COTS Ethernet-enabled protocols may suffice. Traditional choices such as isochronous ProfiNet and EtherCAT manage hundreds of devices in a time-critical manner for local networks [33]. The proposal leaves thus the choice of the connection type to each application case.

Although the on-premises computation has been moved to the Control Cluster component, the proposed style foresees additional control software installed on on-premises devices. For redundancy purposes, the cluster may indeed operate a redundant copy of the on-premises controller (Section 5.3). Some devices may operate as Remote Terminal Units (RTU), serving as an interface to the containerized software or even execute some minor local control function. Such local control loops [29] would have the advantage of reducing latency while exploiting the computing power of a (private) cloud.

Morabito [34] shows that control applications can run inside a container on typical ARM single board computers with a minimal performance impact. Replication and stateful container snapshots further enable Industry 4.0 features also for such on-site devices. As part of data evaluation and sharing, they can now independently and automatically calculate health, estimated remaining useful life and other parameters, supporting machines' self-awareness. For instance, via a time-machine [8], a machine has access to its saved PHM states, consequently its performance indicators and reports. It can compare them with itself and others of a fleet, enabling $\mathrm{self\text{-}comparison}$. Thus, through containerization, we ease maintenance and reduce cost while increasing resilience and robustness.

5.5 | The Orchestrator

The heart of this proposed architecture style is the orchestration software running on each real-time capable node of the Control Cluster. An $\mathrm{orchestrator}$, in this context, is a tool developed to increase resource utilization without significantly impacting determinism. It monitors containers and resources and assigns the latter according to algorithms, rules, or predetermined configurations.

There are two ways to manage resources: static and $\mathrm{dynamic}$. If statically configured, the level of latency and determinism that is achievable can be defined up-front. A static resource schedule is created offline and passed to the orchestrator for execution. Although such a configuration would be the safest, the amount of resource sharing gained is limited. For such a fixed schedule, the configuration must be pessimistic, taking the given worst-case execution time (WCET) as regular and granted and reserving the corresponding CPU-slice for every application. For

higher resource savings, a dynamic reallocation strategy is attractive. A dynamic scheduling strategy yet reallocates containers during run-time to guarantee timeliness when unforeseen delays occur. It allows higher resource sharing as it can adapt to current needs. Complete dynamic rescheduling of containers would be non-deterministic as it depends on the feasibility/admission test [21]. However, with given constraints, determinism can be managed within a certain probability of success that depends on the spare and available time-frame and is inversely proportional to the optimism in reprogramming worst-case times.

Instead of allocating resources based on worst-case parameters, a dynamic orchestrator uses probabilities to assess the situation. It considers typical run times, contemporaneity factors, and probabilities of exceeding a given WCET. It samples run-times and generates cumulative distribution functions or performs curve-fitting to predict distribution models and probabilities. The combined probabilities then tell the rate of success of a schedule and trigger resource organization as needed. This approach resembles the $\mathrm{vertical\ scaling\ techniques}$ used in cloud-hosted applications [35]. Similar approaches in cloud computing environments increase resource efficiency through oversubscription, where the reserved resources may exceed the actual requirements acting as buffers for worst-case situations [36]. In our case, it can be assessed to which probability a system-wide malfunction may occur, allowing a system administrator to set a maximum acceptable boundary of risk. This boundary then defines the probability of success of dynamic scheduling related to the achieved resource savings: the higher the risk, the more savings may be achieved.

6 | EXPERIMENTS

Innovating a traditional control system to align it with the Industry 4.0 principles is a process that comprises different stages with an increasing level of innovation. A traditional control system is typically embedded (i.e., the control is native to the hardware). Thus, the most straightforward migration strategy externalizes the control to a computer system equipped with OS and appropriate software programs. Such a system is referred to as a bare-metal server, and it is typically single-tenant (i.e., dedicated to a single customer). A control system on a bare-metal server can be further containerized. Containers execute individual threads or services in isolation and can support parallel multi-tenant computations (i.e., multiple customers execute their threads in isolation). A bare-metal server can also be virtualized to increase portability. Virtual machines are emulators of a computer system, and a bare-metal server can be migrated to one or more virtual machines (VM)s. VMs can fully virtualize the whole computer system and run on shared hardware or a host OS. In the former case, the VM manager (i.e., hypervisor) is Type 1 or native and has direct access to hardware; in the latter case, it is Type 2 and depends on the host OS layer's services. Containers and virtualization can be combined to increase portability, thread isolation, and multi-tenancy. Finally, a control system can be migrated to the cloud as IaaS. Cloud services can host a control system as VMs equipped with Type 1 hypervisor and can additionally be containerized.

In this work, we propose an approach to evaluate and compare the performance of the following types of virtualized control systems:

- A containerized bare-metal server, which we use as baseline;
- A containerized Type 1 hypervisor controlled virtual generic instance on the cloud designed according to the architecture described in Section 5;
- A containerized Type 1 hypervisor controlled virtual compute-optimized instance on the cloud designed according to the architecture described in Section 5.

Each of the three systems runs the same Real-Time enabled OS and test real-time applications that log measurement data during run-time. To be as faithful as possible, we extracted the parameters of a running multi-tenant industrial control system from specifications and data provided by Siemens Germany (example of Fig. 1). We instantiated it as an example for performance test case 4, Section 6.3. To add variability and test generalizability of the approach, we extended the test cases to cover scenarios with a smaller and mixed-period executable (test case 1-3). These should help to identify the challenges of interfering tasks and the effect they have on responsiveness and determinism on each other in a virtualized and containerized environment.

Fig. 2 illustrates our method. To set up the testing infrastructure, we base our choices on our previous work on the evaluation of containerization and virtualization solutions through offline and latency tests [37]. We briefly summarize this selection and the resulting solutions in Sections 6.2 and 6.2.1 respectively.

6.1 | Experiment Setup

We have chosen the following systems for all our tests (including Hofer et al. [37]). The bare-metal server features two Intel Xeon X5560 (Q1'09) processors on eight cores, 16 threads, limited to two cores for our experiments. For hypervisor-based tests, we selected Amazon Web Services (AWS) to host cloud-based environments. Their recent virtual instances use a new hypervisor based on KVM, called `HVM`, which allows direct assignment and control of hardware and resources, reducing the virtualization overhead. The new instances offer comparable HPC performance but greater flexibility and scalability [38]. We selected an AWS HVM Type 1 hypervisor-based T3.xlarge generic and a C5.xlarge computation optimized instance. The T3 instances feature an Intel Xeon Platinum 8100 or 8200 (Q3'17/Q2'19) series CPU, while C5 runs a custom Intel Xeon Gold 6200 series (Q3'17) CPU. A typical T3 instance is further limited to a 40% CPU baseline. If an instance exceeds that level of CPU, it will eventually be throttled down to 40%. A T3-Unlimited enabled variant allows for CPU bursts up-to 100% at an additional price. Both AWS instances run on four virtual CPUs, 8 or 16 GB of RAM, shared resources, and use a custom configured kernel set up to support their proprietary hardware in Ubuntu.

6.2 | Selection of the off-the-shelf OS and containerization tools

In Hofer et al. [37], we reviewed the state-of-the-art operating systems that can provide both (hard) real-time and container framework support. Table 1 summarizes the operating systems we reviewed. We identified two promising operating systems, Xenomai and PREEMPT-RT patched Linux, and tested task latencies in various configurations, with the system both in idle and stress scenarios. For such latency tests, we chose Linux kernel version 4.9.51, the latest release available that features both patches at test time. We require the EDF scheduler and Greedy Reclamation of Unused Bandwidth (GRUB) algorithm for this work's performance test, available only in kernel versions 4.13 or higher. These operate on the latest Ubuntu LTS release and patch, i.e., Ubuntu Server 18.04.2 and `PREEMPT-RT` 4.19.50-rt22. Thus, in the end, we choose Ubuntu Server LTS with the PREEMPT_RT patch for the optimization tests.

We further studied the three containerization platforms introduced by [19]. While the solutions seem similar, they differ in their goals. LXD focuses on (stateful) system containers, also called infrastructure containers, whereas Docker focuses on ephemeral, stateless, minimal containers. While the former is better suited for long-running software based on clean distribution images, the latter is indicated for instances that are replaced entirely when a new version is available. The two models are not mutually exclusive; for instance, LXD can provide a full Linux system in a Docker instance. Singularity tries to offer a hybrid solution that combines direct access performance with containers' isolation. Singularity is an incomplete isolation solution since it grants access to I/O operations without context changes. After

TABLE 1 Summary of tested operating systems

Name	Pros	Cons	Evaluation
resinOS + patch(X3/PRT)	Small, automatic updates, Balena preinstalled	manual patch, possible efficiency issues	Maintenance high, may not work at max efficiency
Ubuntu Core + patch (X3/PRT)	Small, automatic updates	manual patch, possible efficiency issues	Maintenance high, may not work at max efficiency
Ubuntu LTS + Xenomai 3	Ubuntu network and technologies, Separation of RT and nRT	Recompiling of applications needed, many RT cores drops performace	Expect high performance, medium maintenance
Ubuntu LTS + PREEMT_RT	Ubuntu network and technologies, up-to date	Keep awareness of limits, driver and hardware management needed	Med-high performance and low maintenance expected

manual tryouts and specification review, we selected Balena[3] and Docker[4]. With these technologies, we configured a real-time container engine and built and ran a containerized real-time application. Our experiments showed that the containers could be grouped and orchestrated using the Linux kernel's CGroups feature to optimize resource utilization. This feature will be used in optimization tests. More details are available in Hofer et al. [37]. The selected configuration is used for the new performance and optimization tests in Section 6.3.

6.2.1 | Latency tests

The latency tests verify the suitability of specific hardware or virtualization solutions in Table 1. By applying computational and I/O stress to a task's shared resources, we can examine its real-time parameters' latency effects. We focused on the interaction of virtualized control tasks with the shared environment, as described in the following. The performance tests execute in container batches with varying system load and timing constraints. We use the Earliest Deadline First (EDF) scheduling to reach high theoretical utilization rates of 100% [21]. Then, we partition resources via *CGroups* to virtually address every resource slice as if it were a separate computation unit. The orchestration software of Section 5.5 helps us manage interrupts, create CGroups and assigning its slices, and manage system resources to isolate them from test containers during run-time. First, we observe performance variation by changing kernel boot parameters of the off-the-shelf OS chosen in the latency tests. We apply boot time kernel settings during multiple reboots such as scheduler tick timing, scheduler isolation, and RCU back-off CPUs. The goal is to find settings that promise steady execution on the three hardware instances. A stable median, low average, and standard deviation indicate ideal kernel configurations for each machine type. Next, we compare the performance of the three test architectures with the most stable setup. We increase and mix task configurations and verify the testing run-time determinism in long-term execution. Dropping of performance and the amount of missed deadlines ascertain the absolute upper sharing bound.

Our test model relies on calibration loops to reproduce similar load scenarios for all three system instances, fixing

[3] Balena project homepage: https://www.balena.io

[4] Docker homepage: https://www.docker.com

the task time t_i. We remove the control-task related noise and delays ($n_i k$) by reducing the test tool to computation only. The test software locks its memory in a page, avoids IPC or I/O, and reduces involuntary task switches where possible. For all data, we then produce statistics containing minimum, maximum, average, and standard deviation. Further statistics such as median skew and test group maximum values are added to the performance tests to assess distribution and uniformity.

We use *cyclictest* [39] Version 1.0 to measure the latency of cyclic firing behavior of a real-time application and stress [40] to simulate load in the system. The offline preliminary tests run on a dual-core, four-thread, i7 Skylake (U) system, while the main hardware comparison tests run on the three systems detailed in Section 6. During the progressive isolation of CPU resources, we measure the idle firing time and firing time change with every CPU runs stressing threads. Once found the best setup, we perform one idle and one stressed test for each configuration. All variants, i.e., Standard Ubuntu, Xenomai patch, and PREEMPT-RT patch, run the tests for at least one million firing loops. The logged results deliver the data for the long term test evaluation.

6.2.2 | Execution and Results

In summary, the latency tests give the following main results. The first preliminary latency tests determined that guest-host CPU isolation with a load balancer is the best setting for our purposes. Table I in [37] displays test results for the preliminary test. Fig. 5 then shows the comparison test results with our found best setting. Ideally, the maximum firing delay of the threads should stay below $\frac{1}{10}^{th}$ of the cycle time, which we assumed to be $100ms$ for the sake of comparison in this study. Therefore, Fig. 5 features two reference lines visualizing the boundaries for typical thresholds, one at $10ms$ (for a $100ms$ cycle) and $100\mu s$ (for a $1ms$ cycle). A total of ten million loops over multiple hours have been executed for each configuration. All results obtained have been gathered under *stress* and should be considered the worst-case scenarios. Among all standard kernel configurations, the reference bare-metal solution equipped with any of the three patches (BM, left box-plots in Fig. 5) performs best in the mean. If we consider the PREEMPT_RT configurations (Prt) across all machine types, the bare-metal setup performs best in mean but not spread as the box-plot whisker spans higher, almost reaching the $100\mu s$ threshold. With only 96 occurrences out of 10 million (0.00096%) exceeding the upper limit, a general T3 instance with PREEMPT_RT can be an economical solution for a bare-metal replacement where strict determinism is not needed or cycle times are higher than the peak value measured $49ms$. It shows the lowest spread and peak ($114\mu s$) among the measured instances that only a PREEMPT_RT T3-Unlimited enabled unit outperforms.

6.2.3 | Latency results discussion

RQ1: *What are possible off-the-shelf system configurations that make resource sharing through containers viable?*
We identified Ubuntu 16.04 LTS with the PREEMPT_RT real-time patch, and Docker containers as the best fit among the examined low maintenance options. We came up with four different solutions suitable for migration to application containerization by observing systems under stress and analyzing task latency across different configurations. These solutions maintain wake-up determinism at different levels as follows:

1. The bare-metal solution (BM) ensures the most deterministic behavior for real-time requirements. Even though it is the weakest among all configurations in terms of CPU resources, the strict bond between hardware and software boosts its responsiveness.
2. The virtualized instance C5 with PREEMPT_RT patch is the best non-hardware solution for real-time requirements that trades-off good average latency and deterministic behavior. While it still suffers from some Hypervisor

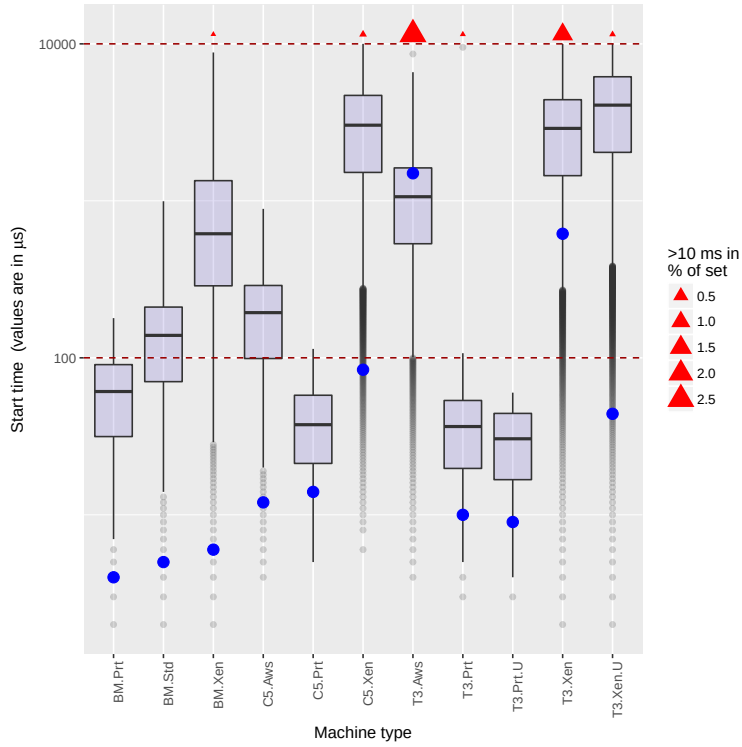

FIGURE 5 Boxplot of latencies, with mean (blue dot) and overshoot size (red triangles).

BM.Prt	Bare Metal with PREEMPT-RT	T3.Aws	AWS HVM type 1 with T3
BM.Std	Bare Metal standard	T3.Prt	AWS HVM type 1 with T3 and PREEMPT-RT
BM.Xen	Bare Metal with Xenomai	T3.Prt.U	AWS HVM type 1 with T3 and PREEMPT-RT Unlimited
C5.Aws	AWS HVM type 1 with C5.xlarge	T3.Xen	AWS HVM type 1 with T3 and Xenomai
C5.Prt	AWS HVM type 1 with C5.xlarge and PREEMPT-RT	T3.Xen.U	AWS HVM type 1 with T3 and Xenomai Unlimited
C5.Xen	AWS HVM type 1 with C5.xlarge and Xenomai		

latency, the exclusiveness of CPU access and the ability to control C-states allow reducing non-I/O induced noise and plot better value consistency.

3. The T3 unlimited instance with PREEMPT_RT is a cheap solution with good average latency. As there is no guarantee of extra CPU power availability or responsiveness, these configurations can be chosen as an intermediate solution between T3 and C5 instances.
4. The T3 instance with PREEMPT_RT is a viable solution with good average latency that might not qualify for hard real-time requirements. Also, this T3 instance may not ensure the physical CPU exclusiveness. For this reason, the C5 PREEMPT_RT instance may be a better choice for stricter timing requirements.

In conclusion, the results are promising and confirm the feasibility of migration to IAAS solutions.

6.3 | Performance tests for resource optimization with container orchestration

With the chosen test infrastructure described in Section 6.2.1, we designed and executed a set of performance tests of the three types of deployed architectures for virtualized control systems (Section 6) while executing real-time tasks in a shared and containerized environment. The new performance tests further evaluate system configurations and assess CPU sharing limits while the orchestrator we define in Section 6.3 manages tasks and resources.

6.3.1 | Test design

We perform the following resource efficiency tests by placing a set of real-time applications on shared resources. For this purpose, we use the real-time test software `rt-app` [41] to create configurable dummy applications. We place them into separate containers and configure them with running periods and computation times. For simplicity, we match relative computation deadline d_i and period p_i of a container i in all tests (See Eq. 1). Beyond the worst-case value expected for r_i, WCET w_i, the app configuration requires a simulated run-time parameter t_i. The latter defines the amount of time the application spends performing dummy loop calculations to approximate the simulated task's supposed run-time. As the number of loops to perform depends on a constant value set at a startup (calibration) and isolated task-caused noise, the resulting execution time c_i depends solely on interaction with the system. This calibration further ensures identical task execution models for each hardware, respecting the same system load and run-time parameters on all tested systems. During test execution, the orchestrator and $\mathrm{rt\text{-}app}$ monitor the system's run-time behavior. With these results, we can detect startup latency, execution jitter, and deadline misses.

Our experiments run with the following performance parameters. We observe the actual total computation time ci as we want to minimize fluctuation while keeping zero deadline overshoots during run-time. Consequently, the smaller the standard deviation and the skew of ci, the more stable is the solution.

Each test batch consists of the following four configurations:

Test Case 1 – *lower bound:* homogeneous period and run-time among all containers executing on the same resources with a WCET w_i smaller than the best case scheduler's wake-up granularity ($1000\mu s$). We force high-resolution granularity scheduling with this test, causing more scheduler calls, and consequently, context changes and latency, than planned for the highest scheduler tick rate. The test setup consists of ten containers with a WCET w_i of $900\mu s$ each. With a period and deadline of $10ms$ each, this results in a resource utilization factor 9% for each container.

Test Case 2 – *upper bound:* homogeneous period and run-time among containers executing on the same resources, with a run-time to period ratio ($\frac{t_i}{p_i}$) close or equal to 0.5. The kernel limits the scheduler's refresh rate to $1000/4000\mu s$ for 1000/250Hz systems, making this configuration a scheduling challenge. The configured test case includes two containers with $2.5ms$ WCET and $5ms$ period.

Test Case 3 – *diversity*: mixed periods and run-times for each container executing on the same resources. Irregular run-times should challenge the possibility of execution alignment where containers always run in the same order. Moreover, the deadline priority continues to rotate, helping to determine stability in mixed scenarios. This test case consists of a diverse set of containers: one container set to $2.5/5ms$, one $900\mu s/10ms$, and one configured as $3/9ms$ for worst-case computation time and deadline/period, respectively.

Test Case 4 – *Simulation test:* As a conclusive test, we emulate the scenario of our example application extracted from the use case specifications provided by Siemens Germany. We verify the boundaries for this use case through the parallel operation of multiple instances of the flow control software. The test configuration includes ten

containers with the period and run-time homogeneous among all tasks and running on the same resources. The timing is based on the values of the example in Section 4 (10ms run-time, 100ms deadline and period).

The test scripts and complete test results can be found in the project repository [42].

6.3.2 | Execution and results

The first test batch for kernel configuration shows the full dynamic tick configuration with mandatory back-off as performing best. The setup scored the best results in most runs for average and median stability. Test case 1's values gave standard deviations of 31μs on bare-metal, 7μs on type C5, and 15μs on type T3 instances running ten containers in parallel. Second by performance and stability is the fixed-tick kernel configuration. This second configuration turns out useful if more than one task is available to run or next in line simultaneously, mostly when mixed with non-deadline-oriented schedules. For the long-run tests of test batch two, we choose, therefore, a PREEMPT_RT kernel with full dynamic ticks and RCU back-off, with a run-time of 15 minutes each.

In Test Batch Two, we repeated the same tests on all three systems. To test the repeatability, we re-calibrated and repeated the tests multiple times. Additional results and diagrams can be found in the project archive [43]. Tables 2 to 5 report the results of the four test cases. We display numbers for test cases 1 and 4 with loads from close to 50 up-to 100% of CPU time only.

Test Case 1: Even though the system load never reaches 100% in these tests, the table shows continuity among all instances with minor variations of about one to five percent. The AWS C5 instance performed best among the three candidate systems. With a steady average run-time, close to null skew, and an unvarying standard deviation, the three's most resourceful keeps a steady and deterministic run. The other virtual instance is slightly slower but keeps a small variation bound. Interestingly, in this configuration, the bare-metal system shows the most jitter. Unlike T3 and C5, the skew and standard deviation values halve and double from test to test, having a similar, varying impact on the average run-time. In the highest configuration, systems reach loads of 90 to 93%. No runs across all configurations show any overshoot, confirming the feasibility of handling multiple real-time containers while suffering from relatively small system noise.

Test Case 2: This test case contains only two configurations: one or two containers. Unfortunately, the system noise is already high enough to make a single container exceed 50% of CPU load. A consequent phenomenon is that configuration two produces only one run time log on all candidate systems. All performed tests do not deliver enough values to get minimum and maximum for the skew and deviation, marked by an asterisk in Table 3, leaving doubt on possible performance change. However, the visible data does suggest a small skew of the distribution, but the standard deviation tends to remain low. The high amount of deadline misses on configuration two, around 20 or more for all systems, nonetheless upholds that system overload causes the expected misbehavior.

Test Case 3: The average run-time in this mixed container spans all running containers and serves as a variation monitor rather than a proper average. Indeed, the high number of deadline misses in this third configuration causes task preemption and run-time values to boost. Like in the previous test cases, AWS C5 keeps a steady and centered distribution displaying no skew. The bare-metal configuration behaves similarly to test case 1 with a slightly trembling skew around a tenth of a microsecond while maintaining a relatively constant standard deviation. However, the general-purpose instance suffers from the high load of the last test and drifts into an unusually high skew and standard deviation of 37μs. Test 1 and 2 already use close to 90% of CPU time, causing first deadline overshoots for bare-metal (20157) and AWS T3 (25) in configuration two. The AWS C5 instance instead does not

TABLE 2 Test batch 2, test case 1

Configuration	Bare-Metal			AWS T3			AWS C5		
	AVG	SKW	SD_MX	AVG	SKW	SD_MX	AVG	SKW	SD_MX
4 units <50%	935	0/14	22.83	903	3/11	16.21	914	2/2	5.05
5 units <60%	950	0/12	23.91	904	4/11	16.33	913	0/0	6.53
6 units <70%	969	0/6	11.34	905	5/11	15.73	913	0/4	5.97
7 units <80%	930	0/19	22.37	904	1/11	16.46	913	0/4	5.48
8 units <90%	926	0/6	12.28	904	2/10	15.43	913	1/3	5.43
9 units <100%	920	0/13	23.33	904	4/10	15.14	914	1/3	5.12
10 units ≈100%	933	0/6	11.66	904	4/9	14.81	914	1/4	5.50

TABLE 3 Test batch 2, test case 2

Configuration	Bare-Metal			AWS T3			AWS C5		
	AVG	SKW	SD_MX	AVG	SKW	SD_MX	AVG	SKW	SD_MX
1 unit <60%	2584	4	21.28	2510	12	27.34	2538	1	6.48
2 units ≈100%	2521	13*	23.23	2506	14*	26.03	2535	0*	5.75

TABLE 4 Test batch 2, test case 3.

Configuration	bare-metal			AWS T3			AWS C5		
	AVG	SKW	SD_MX	AVG	SKW	SD_MX	AVG	SKW	SD_MX
1 unit <60%	2579	10	22.16	2507	9	23.23	2534	1	7.70
2 units <90%	2589	4/14	22.65	2569	4/7	19.01	2587	0/1	5.40
3 units ≈100%	2269	1/7	26.54	2179	7/37	37.25	2183	0/1	11.13

TABLE 5 Test batch 2, test case 4.

Configuration	Bare-Metal			AWS T3			AWS C5		
	AVG	SKW	SD_MX	AVG	SKW	SD_MX	AVG	SKW	SD_MX
4 units <50%	10712	0/8	31.78	10072	14/16	45.69	10139	2/2	12.03
5 units <60%	10614	8/10	35.66	10056	14/16	35.46	10310	1/3	68.98
6 units <70%	10132	3/10	34.87	10038	4/7	23.30	10136	1/2	10.55
7 units <80%	10115	0/11	31.25	10166	49/57	123.36	10138	1/2	11.08
8 units <90%	10104	3/12	32.20	10052	10/16	41.55	10137	1/2	9.04
9 units <100%	10356	3/8	29.83	10059	5/16	126.17	10138	1/2	9.94
10 units ≈100%	10089	4/10	46.29	10027	1/3	12.75	10136	1/2	9.91

Performance variations with an increasing number of containers. Values in μs.

AVG - average run-time of container set overall measured run-times

SKW - absolute min and max distance between average run-time value and median for the configuration

SD_MX - standard deviation of the container with the highest skew

* The starred values indicate runs where at least one thread did not produce log output

fail any run in these first two tests. For test three yet, all configurations show misses in the order of thousands in this 15-minute test.

Test Case 4: In this fourth test case, the AWS C5 keeps a median centered distribution again and, except for the second test, stays with distributions close to $10\mu s$. Similarly, the bare-metal system shows the same small distribution skew and slight jitter in its standard deviation as it did for case 1. However, the AWS T3 in this configuration seems to suffer more from system latency and noise, showing higher skew and standard deviations than in any test before. As for maintaining deadlines, the results keep stable for all tests except full-load, where they show 3, 244, and 3 overshoots for bare-metal, T3, and C5, respectively.

6.3.3 | Performance results discussion

RQ2: *What is the achievable level of CPU sharing with a standard real-time enabled kernel?* The static orchestration tests highlight some additional facts beyond the best performing configurations. One thing that arose is the performance increase of generic instances (T3/T3U) when moving from 10 to 50% of CPU load. This improvement is like those observed for latency tests, probably due to the hypervisor's CPU management. It may move the vCPU thread or fill the remainder with other instance's vCPU threads. Unlike computation-intensive instances (C5), a type T3 does not require locking onto a specific physical CPU, adding volatility and latency. The latency witnessed during low system load may thus be a product of virtual thread shifting. As a T3 instance has no hardware access, this CPU could continuously transition to a lower C state, adding more latency for the thread to be back in execution. Thus, similarly to the latency tests' conclusions in Section 6.2.1, a higher CPU load reduces system-induced latency and noise. Another observation is that with a high CPU load, the real-time tasks take over minor threads. This overtaking causes misbehavior such as incomplete or empty log files, tasks not terminating at predefined times, or sometimes unresponsiveness of the system. Furthermore, in some cases, one container in the test configuration kept a continuous run-time deviation from the preset run-time. Occasionally, the average and median run-time kept around $12ms$ instead of $10ms$ while displaying the same jitter and deviation behavior. Despite this deviation, the run-time values' present stability or determinism does not hinder real-time operation. During preliminary testing, we noted that the restart of a virtual instance on the AWS cluster causes it to move to a different system rack. Given that hardware across system racks may not be equal, e.g., Xeon 8100 vs. 8200 series CPU, this change after shutdown is a variable to be considered. While this influenced the calibration for AWS-based tests, it does not influence the comparison among the same virtual instance results. The resulting run-time data from both test batches shows that resource sharing for real-time containers is feasible. Properly configured, a system can reach a utilization limit of 0.9 or 90%. Our tests have shown that although under stress, both latency and determinism reach desired values. Among all, the AWS C5 shows the most stable run-time values. It is the most resourceful of all systems and thus likely suffering the least from background noise. Being a virtual instance, it does not respond directly to hardware interrupts like the bare-metal system, softening the amount and duration of interrupts. However, this does not mean it is not influenced by system noise. As seen in test configuration two of Table 5, the C5 system can still be subject to significant variations. Nevertheless, the bare-metal instance shows a higher fluctuation in skew and standard deviation but still stays steady in a specific range. In all tests, the results for skew and deviation remained within 20 or $30\mu s$. While this jitter may seem a problem, it can be isolated to this range, making it predictable. Lastly, the generic AWS T3 shows the worst but still relatively stable run-time behavior. The highest fluctuations are shown in test case 4, where idle times between cycle repetition are the longest. Indeed, this confirms that during idling, the hypervisor may change the physical CPU reservation. If we consider these constraints, also an economic generic AWS T3 instance may suffice our computational needs. In the end, all systems show adequate stability for the sample loads we created. The worst variation of system run-time stays

within 126μs, a value that has to be considered when dealing with hard deadlines in the order of a few milliseconds or less. However, this confirms that all setups allow shared computational loads up-to and exceeding 90%. Only close to full load, the systems start to suffer from deadline overshoots. Starting from these results, we can now investigate if off-the-shelf technology keeps the process viable once task I/O ($n_i k$) and network latency are taken into account.

The generalization of the results is made based on the premise that the observed real-time tasks are calibrated to their system, representing similar behavior on all three candidates. The task combination and duration have been chosen rigorously to consider system utilization factors, system scheduler tick rate, programmed interruptions, task period mismatches with consequent priority shifting, and their combination. Task combinations with longer periods will suffer less from the system and environment-induced noise. Tasks with shorter periods, e.g., a 1 ms control loop for motion control systems, enter a domain where a commercial scheduler tick of 1000Hz would lose its utility and require different scheduling schemes or system setups. A local control loop (on-premises RTU, Section 5.4) managing the quick feedback is one thinkable solution.

6.4 | Scripts and tools developed for the tests

Both kernel versions with patches described in Section 6.2 can be built and restored for all three architectures using our automated script, available online [43]. Further details, the script executing all the tests, the installation scripts, the experiment data, and technical details and results are available in [37].

7 | LESSONS LEARNED

Migrating control applications from hardware to bare-metal and finally to an IAAS infrastructure has two significant advantages. First, application containerization allows managing and monitoring execution easily. It eases parallel operation, redundancy, quick updates, and upgrades for container-confined code. We have seen that replacing a set of running containers at run-time is feasible and it allows distributed updates and life-fixes of critical problems. A container may execute on-site on the device while keeping a copy on the IAAS for backup or redundancy. Second, physical hosts, in the cloud and private cloud, serve multiple virtual instances. The computing power available to individual instances is often flexible. Such computing power usually exceeds the original hardware's performance, permitting us to use more complex and demanding control algorithms. On the other hand, application containerization requires some software adaptation, the generalization of I/O and interfaces, and results in hardware abstraction. This extra effort must be taken into account when evaluating a migration strategy. The introduced distance between control cloud and on-site devices may also require more time-critical communication protocols respecting standards such as the TSN family, which may add some overhead. On the other side, a recent systematic mapping study [2] highlights the limits of available architectures, for instance, based on the 5C attributes. Our architecture style has been designed to ease application migration and support the migrated application's self-* properties with the management and monitoring layer. In summary, the proposed architecture gives support to a complete control solution for both research and industry.

Thanks to the results of our tests, we have drawn a few lessons we learned that could be beneficial for practitioners aiming to use application containers for industrial control as in the following:

Real-time requirements, and consequently, architecture, are application-specific. While a generic architecture, as presented in Section 5 covers most situations, the current layout changes for each case. Like in the motivating exam-

ple, Fig. 1, levels may merge where the environment requires it and give the final architecture a different, reduced shape. However, responsibilities, functions, and goals remain unchanged.

Picking a real-time capable OS does not guarantee determinism. Different OSs have distinct trade-offs. While the Linux Xenomai patch outperforms the PREEMT_RT patch, its induced kernel overhead limits system scalability. When choosing the OS, we have to match hardware and constraints for the best results closely.

Modern virtualization techniques perform well enough to maintain determinism. Both the latency and performance tests showed satisfying results confirming the viability of application migration. Depending on task configuration, we can reach subscription rates exceeding 90% of CPU resources. The next and last constraint to tackle will be the network and I/O latency. However, this constraint depends on the applications' timing requirements and, thus, needs further investigation.

Direct hardware access decreases latency and improves responsiveness. Despite the less powerful hardware, the Bare-Metal server still outperforms newer hypervisor-based instances for task responsiveness. Similarly, limited access to CPU resources improves virtualized performances, i.e., AWS C5 vs. T3-Unlimited. Thus, although possible, virtualized instances require newer and better hardware to reach similar performance. A practitioner might thus need to consider resource sharing beyond control containers to reduce hardware installation costs. The architecture of Section 5 helps to address this job.

Economic virtual instances may suffice for less strict determinism requirements. Generic AWS T3 shared instances show comparable results for task firing latency but add variability when under stress. While this variability discourages their use in environments with strict timing requirements, i.e., task periods of few milliseconds or less, it enables them, however, for less critical operations, e.g., periods in 100's of milliseconds like in the motivating example, Section 4.

8 | CONCLUSIONS AND FUTURE WORK

This paper explored the limits and feasibility of migrating real-time applications from bare-metal servers to virtualized IAAS configurations, up-scaling the installed computation appliances. We showed that containerization offers a novel paradigm for control applications exploiting cost reduction advantages. Unfortunately, identifying the off-the-shelf solution that fits the specific real-time constraint is often challenging for practitioners. We suggest, therefore, an architecture to help the migration and placement of these new applications. It serves as a template for transition and enables advanced features for novel industrial control systems. Next, we experimentally verified migration viability, analyzing environment noise and latency. We introduced an orchestration tool to manage environment setup and schedule the real-time containers based on pre-configured capacities. With system calibrated testing tools and targeted test cases, we address worst-case scenarios to identify upper limits for externally induced noise and latency. With a worst run-time variation of $126\mu s$ on a generalized T3 instance, we have confirmed that environment-induced noise is low enough to permit migration. As the testing scenarios targeted worst-cases, except for some particular use cases, the result is transferable to other task setups and combinations. Finally, we concluded with a summary of the pros and cons and listed lessons learned.

In future work, I/O and system latency will be investigated, and dynamic allocation strategies will be exploited to improve system performance. A dynamic orchestration algorithm will help tackle issues that arise when tasks do not respect their designed parameters. This new configuration will also increase a system's robustness and detect a deviation of task behavior due to cyber-attacks or externally induced overloads. New latency and performance tests on industrial use cases will help further analyze limits and possibilities for shared-resource real-time systems, including

robustness and behavior when under attack.

references

[1] Telschig K, Schönberger A, Knapp A. A Real-Time Container Architecture for Dependable Distributed Embedded Applications. In: 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE) IEEE; 2018. p. 1367–1374.

[2] Hofer F. Architecture, technologies and challenges for cyber-physical systems in Industry 4.0 - A systematic mapping study. In: 12th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM); 2018. p. 1–10.

[3] Jiang Y, Tsang DHK. Delay-Aware Task Offloading in Shared Fog Networks. IEEE Internet of Things Journal 2018;5(6):4945–4956.

[4] Roy R, Stark R, Tracht K, Takata S, Mori M. Continuous maintenance and the future – Foundations and technological challenges. CIRP Annals 2016;65(2):667–688.

[5] Moga A, Sivanthi T, Franke C. OS-level virtualization for industrial automation systems: Are We There Yet? In: Proceedings of the 31st Annual ACM Symposium on Applied Computing - SAC '16 ACM Press; 2016. p. 1838–1843.

[6] Tasci T, Melcher J, Verl A. A Container-based Architecture for Real-Time Control Applications. In: 2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC) IEEE; 2018. p. 1–9.

[7] Goldschmidt T, Hauck-Stattelmann S. Software Containers for Industrial Control. In: 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) IEEE; 2016. p. 258–265.

[8] Lee J, Bagheri B, Kao HA. A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems. Manufacturing Letters 2015 jan;3:18 – 23.

[9] Telschig K, Knapp A. Towards Safe Dynamic Updates of Distributed Embedded Applications in Factory Automation. In: 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA) No. reconfiguration., IEEE; 2017. p. 1–4.

[10] Wu D, Liu S, Zhang L, Terpenny J, Gao RX, Kurfess T, et al. A fog computing-based framework for process monitoring and prognosis in cyber-manufacturing. Journal of Manufacturing Systems 2017 apr;43:25–34.

[11] Terrissa LS, Meraghni S, Bouzidi Z, Zerhouni N. A new approach of PHM as a service in cloud computing. In: NA, editor. 2016 4th IEEE International Colloquium on Information Science and Technology (CiSt) IEEE, IEEE; 2016. p. 610–614.

[12] Schroeder G, Steinmetz C, Pereira CE, Muller I, Garcia N, Espindola D, et al. Visualising the digital twin using web services and augmented reality. In: NA, editor. 2016 IEEE 14th International Conference on Industrial Informatics (INDIN) IEEE; 2016. p. 522–527.

[13] Fazio M, Celesti A, Ranjan R, Liu C, Chen L, Villari M. Open Issues in Scheduling Microservices in the Cloud. IEEE Cloud Computing 2016 sep;3(5):81–88.

[14] Hofer F, Russo B. IEC 61131-3 Software Testing: A Portable Solution for Native Applications. IEEE Transactions on Industrial Informatics 2020 jun;16(6):3942–3951.

[15] García-Valls M, Cucinotta T, Lu C. Challenges in real-time virtualization and predictable cloud computing. Journal of Systems Architecture 2014 oct;60(9):726–740.

[16] Hallmans D, Sandström K, Nolte T, Larsson S. Challenges and Opportunities when Introducing Cloud Computing into Embedded Systems. In: 2015 IEEE 13th International Conference on Industrial Informatics (INDIN); 2015. p. 454–459.

[17] Goldschmidt T, Hauck-Stattelmann S, Malakuti S, Grüner S. Container-based architecture for flexible industrial control applications. Journal of Systems Architecture 2018;84:28–36.

[18] Felter W, Ferreira A, Rajamony R, Rubio J. An updated performance comparison of virtual machines and Linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) IEEE; 2015. p. 171–172.

[19] Arango C, Dernat R, Sanabria J. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments; 2017, arXiv preprint arXiv:1709.10140.

[20] Abeni L, Balsini A, Cucinotta T. Container-based real-time scheduling in the Linux kernel. SIGBED Rev 2019;16(3):33–38. https://doi.org/10.1145/3373400.3373405.

[21] Buttazzo GC. Hard real-time computing systems: predictable scheduling algorithms and applications, vol. 24. Springer Science & Business Media; 2011.

[22] Oshana R. 2 - Overview of Embedded Systems and Real-Time Systems. In: Oshana R, editor. DSP Software Development Techniques for Embedded and Real-Time Systems Embedded Technology, Burlington: Newnes; 2006.p. 19–34.

[23] Osborne S, Anderson J. Simultaneous Multithreading and Hard Real Time: Can It Be Safe? In: ECRTS; 2020. p. 1:23.

[24] Ryan ET, Jacques D, Colombi J. An ontological framework for clarifying flexibility-related terminology via literature survey. Syst Eng 2013;16:99–110.

[25] Broniatowski DA, Moses J. Measuring Flexibility, Descriptive Complexity, and Rework Potential in Generic System Architectures. Systems Engineering 2016;19(3):207–221.

[26] Hofer F, Sehr MA, Russo B, Sangiovanni-Vincentelli A. ODRE Workshop: Probabilistic Dynamic Hard Real-Time Scheduling in HPC. In: 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC); 2020. p. 207–212.

[27] Ross AM, Hastings DE. 11.4.3 The Tradespace Exploration Paradigm. INCOSE International Symposium 2005;15(1):1706–1718.

[28] Hofer F, Russo B. Architecture and its vulnerabilities in Smart-Lighting systems;, https://www.florianhofer.it/papers/?id=6c0b6c41c039423788a965e7cb9547155b2a33bd, under submission.

[29] Han S, Xie M, Chen HH, Ling Y. Intrusion Detection in Cyber-Physical Systems: Techniques and Challenges. IEEE Systems Journal 2014 dec;8(4):1052–1062.

[30] Hofer F. Enhancing Security and Reliability for Smart-* Systems' Architectures. In: 2018 IEEE International Symposium on Software Reliability Engineering Workshops; 2018. p. 150–153.

[31] Gorecky D, Schmitt M, Loskyll M, Zuhlke D. Human-machine-interaction in the industry 4.0 era. In: NA, editor. 2014 12th IEEE International Conference on Industrial Informatics (INDIN) IEEE; 2014. p. 289–294.

[32] Time-Sensitive Networking (TSN) Task Group; 2019. https://1.ieee802.org/tsn/.

[33] Robert J, Georges JP, Rondeau E, Divoux T. Minimum Cycle Time Analysis of Ethernet-Based Real-Time Protocols. International Journal of Computers Communications & Control 2012 sep;7(4):744.

[34] Morabito R. Virtualization on internet of things edge devices with container technologies: a performance evaluation. IEEE Access 2017;5:8835–8850.

[35] Shekhar S, Abdel-Aziz H, Bhattacharjee A, Gokhale A, Koutsoukos X. Performance Interference-Aware Vertical Elasticity for Cloud-Hosted Latency-Sensitive Applications. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD) IEEE; 2018. p. 82–89.

[36] Chen J, Cao C, Zhang Y, Ma X, Zhou H, Yang C. Improving Cluster Resource Efficiency with Oversubscription. In: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC) IEEE; 2018. p. 144–153.

[37] Hofer F, Sehr M, Iannopollo A, Ugalde I, Sangiovanni-Vincentelli A, Russo B. Industrial Control via Application Containers: Migrating from Bare-Metal to IAAS. In: 2019 IEEE 11th International Conference on Cloud Computing Technology and Science (CloudCom) IEEE; 2019. p. 62–69. ArXiv:1908.04465 [cs.DC].

[38] Amazon AWS User guide - The C5 instance; 2018. https://aws.amazon.com/ec2/instance-types/c5/.

[39] rt-tests - test programs for real-time kernels; 2018. https://directory.fsf.org/wiki/Rt-tests.

[40] Stress-ng - the next generation stress testing tool; 2018. http://kernel.ubuntu.com/~cking/stress-ng/.

[41] rt-app - scheduler tools GitHub home; 2019. https://github.com/scheduler-tools/rt-app.

[42] Hofer F, Test archive Orchestration; 2019. http://bit.ly/2PQqnJN.

[43] Hofer F, Test archive; 2019. http://bit.ly/2XdoYPn.