Enabling Kubernetes Orchestration of Mixed-Criticality Software for Autonomous Mobile Robots

Francesco Lumpp, Franco Fummi, *Member, IEEE*, Hiren D. Patel, *Member, IEEE*, and Nicola Bombieri, *Member, IEEE*

***Abstract*—Containerization and orchestration have become two key requirements in software development best practices. Containerization allows for better resource utilization, platform-independent development, and secure deployment of software. Orchestration automates the deployment, networking, scaling, and availability of containerized workloads and services. While containerization is increasingly being adopted in the robotic community, the use of task orchestration platforms (e.g., Kubernetes) is still an open challenge. The biggest limitation is due to the fact that state-of-the-art orchestrators do not support real-time (RT) containers, while advanced robotic software often consists of a mix of heterogeneous tasks (i.e., ROS nodes) with different levels of temporal constraints (i.e., mixed-criticality systems). This work addresses this challenge by presenting RT-Kube, a platform that extends the de-facto reference standard for container orchestration, Kubernetes, to schedule tasks with mixed-criticality requirements. It implements monitoring of tasks and detects missed deadlines for those with RT constraints. It selects low-priority tasks to be migrated at runtime to different units of the computing cluster to free resources and recover from temporal violations. We present quantitative experimental results on the software implementing the mission of a Robotnik RB-Kairos mobile robot to demonstrate the effectiveness of the proposed approach. The source code is publicly available on GitHub.**

***Index Terms*—Autonomous mobile robots, containers, edge cloud, Kubernetes, mixed-criticality systems (MCSs), orchestration, real time (RT).**

I. Introduction

ENSURING the correctness of robotic software is crucial, especially considering the involvement of robots in safety-critical tasks [1]. In addition to functional requirements, these standards encompass various nonfunctional constraints, such as real time (RT), quality of service (QoS), reliability, scalability, and energy efficiency [2]. Meeting these constraints requires configuring robotic application software to operate effectively across diverse computing architectures, including edge-cloud computing clusters [3].

In this context, *containerization* has emerged as a viable solution [4]. It offers advantages, such as improved resource utilization, platform-independent development, and secure software deployment. However, as software for autonomous and intelligent robots becomes more complex, traditional containerization approaches may no longer suffice as they lack the means to scale to the more complex computing architectures. To address this complexity, it becomes necessary to partition services and tasks into distinct containers. This approach helps manage the increasing size of container images, adapt container mapping to different cluster nodes, and enhance system resilience against node failures [5].

Within the context of *multicontainer* deployments, a significant challenge is ensuring continuous robot functionality even in the face of disruptions. As a result, many robotics companies are exploring platforms like Kubernetes, which is the de-facto standard for container *orchestration*, for automatic software deployment to address this issue [6], [7].

However, there is also a growing need for software standards that support *mixed-criticality* applications, which can be found in various domains, such as industrial automation [8], automotive [9], and avionics [10]. A mixed-criticality system (MCS) combines software components (e.g., ROS nodes) with different levels of criticality within a shared computing platform [11]. One of the primary research challenges in MCSs is ensuring the correct execution of high-criticality tasks while sharing computing resources with lower- or noncritical tasks [12] in a user-transparent manner.[1] Within this context, the introduction of software components and layers through containerization can complicate meeting *RT* requirements [13]. Although some research efforts have explored integrating RT properties into container-based virtualization [14], [15], [16], [17], supporting

Manuscript received 8 July 2023; revised 24 October 2023; accepted 8 November 2023. Date of publication 20 November 2023; date of current version 15 December 2023. This paper was recommended for publication by Associate Editor J. Le Ny and Editor N. Amato upon evaluation of the reviewers' comments. This work was supported by the European Union Next-GenerationEU (Piano Nazionale di Ripresa e Resilienza (PNRR) – Missione 4 Componente 2, Investimento 1.5 – D.D. 1058 23/06/2022 and was carried out within the PNRR research activities of the consortium iNEST (Interconnected North-Est Innovation Ecosystem), ECS_00000043). *(Corresponding author: Francesco Lumpp.)*

Francesco Lumpp, Franco Fummi, and Nicola Bombieri are with the Department of Engineering for Innovation Medicine, University of Verona, 37129 Verona, Italy (e-mail: francesco.lumpp@univr.it; franco.fummi@univr.it; nicola.bombieri@univr.it).

Hiren D. Patel is with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON N2L 3G1, Canada (e-mail: hiren.patel@uwaterloo.ca).

Digital Object Identifier 10.1109/TRO.2023.3334642

[1]For the sake of clarity, we will refer to ROS nodes as "software tasks" (or simply "tasks"), and real-time ROS nodes as "RT tasks." We will use "computing nodes" or "nodes" to refer to the physical devices within the edge-cloud computing cluster.

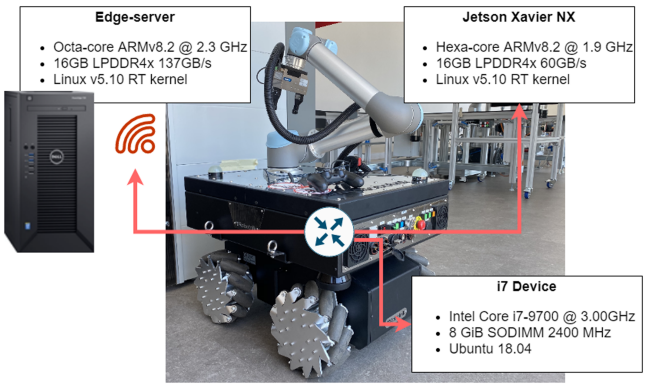


Fig. 1. RB-Kairos compute architecture.

tasks with mixed-criticality requirements remains an open challenge.

Another emerging challenge for MCSs is *integrating* RT containers with orchestration. Orchestration has demonstrated its effectiveness in automating the deployment, networking, scaling, and availability of containerized workloads and services in cloud-native applications [18]. Nevertheless, current state-of-the-art orchestrators do not yet support *mixed-criticality* containers, limiting their widespread adoption for robotic software.

Our work addresses the challenges posed by software containerization and orchestration in the context of autonomous mobile robots with mixed-criticality tasks deployed across an edge-cloud computing architecture (see Fig. 1). Our prior effort in [19] presented an analysis where we verified, theoretically and through a large set of experimental results, whether the overhead introduced by containerization can influence RT tasks. This work extends and completes such preliminary analysis by presenting *RT-Kube*, an ROS and Kubernetes compliant orchestration platform for MCS. RT-Kube includes a new RT scheduler that does not require modifications to the Kubernetes source code. It also incorporates a monitoring plugin consisting of multiple software layers designed to detect worst-case execution time (WCET) overruns, temporal violations, and missed deadlines for RT tasks.

Furthermore, RT-Kube implements a customizable algorithm for selecting eviction victims (lower-priority ROS tasks) and migrating them in RT to different computing nodes within the cluster, thereby freeing up resources when too many temporal violations occur. Building on our preliminary work, RT-Kube can be deployed in various environments without the need for custom-compiled executables. This includes provisioned environments, which are commonly encountered in cloud-based computing (i.e., automatically managed and installed from official sources). The main novel contributions of this work are as follows.

1) An orchestration platform for MCS, that extends the standard Kubernetes scheduling with container sorting, node filtering, scoring, and container-node binding through criticality-aware algorithms and policies.
2) A monitoring mechanism that checks the status of each RT container across the edge-cloud platform and efficiently notifies violations of temporal constraints (e.g., missed deadlines).
3) An RT scheduler that implements runtime migration of *state-less* RT containers across the cluster nodes to avoid system performance degradation.
4) *RT-Kube*, which is released as an ROS-compliant Kubernetes plugin that extends the standard release with the RT support (available at https://github.com/UNIVR-RT-Kube).

The rest of this article is organized as follows. It first presents a performance model and the corresponding experimental results to measure the upperbound latency spikes introduced by RT-Kube. It presents a quantitative analysis of the orchestrator efficiency, by applying RT-Kube to orchestrate containerized mixed-criticality software benchmarks. It evaluates a real case study implementing the mission of a Robotnik RB-Kairos mobile robot for navigation and transportation of goods. Section II presents the background necessary to easily understand the solution, while Section III presents the related works. Section IV describes RT-Kube, while Section V presents the experimental results. Finally, Section VI concludes this article.

II. Background

This section provides background information on various topics related to RT software, Linux scheduling, and container orchestration.

A. Real-Time Software

RT software applications are often crucial for mission-critical tasks, and their timing requirements must be precise and reliable. These applications can have varying degrees of requirements on their timing, typically falling into two categories: 1) *soft* real time and 2) *hard* real time.

In the case of a *soft* RT application, consider a scenario like simultaneous localization and mapping (SLAM) software. This software periodically processes sensor data to create a map. While it is essential that this computation is timely, some degree of delay may be tolerable. In this context, a delayed result could lead to a temporarily less accurate map, which might result in navigation errors. However, such errors are acceptable as long as they are within certain limits.

In contrast, a hard RT application has no tolerance for delayed results. For example, in an antilock braking system in cars, any delay in applying the braking correction can be detrimental. A late correction could adversely affect the handling of the car, rather than improving it. In such cases, a task is considered late if it surpasses its *deadline*, which represents the absolute time by which its execution must be completed.

Both hard and soft RT tasks typically consist of repetitive computation phases that are triggered periodically or sporadically. In *periodic* tasks, these computation phases are activated at fixed and regular intervals. Conversely, *sporadic* tasks are activated with a minimum time interval between each activation, but the actual intervals may vary, allowing for more flexibility in their timing.

During execution, tasks can vary in the amount of time they take to complete, but this variation must have an upper limit. This upper limit is known as the WCET, which represents the maximum amount of time a task can take to finish under any possible system condition. These conditions are influenced by the state of the system, including factors, such as resource availability (e.g., CPU, memory, network) and resource characteristics (e.g., CPU/memory clock frequency, network bandwidth).

A RT task is defined as follows:

$$\text{RT\_task} = (P,\ D,\ \text{WCET}) \tag{1}$$

where P represents the task's period, indicating how often new instances of the task arrive. D is the deadline, specifying the time by which the task must be completed. WCET is the worst-case execution time. These times need to abide the following:

$$\text{WCET} \leq D \leq P. \tag{2}$$

Determining deadlines and periods is a crucial aspect of system design and involves specifying system requirements and task characteristics during the design phase. Estimating the WCET is accomplished through various techniques. These methods include static analysis, where the code is carefully examined to identify the longest possible execution path, and measurement-based approaches, where RT tasks are executed on the target platform to measure their actual execution times. There are advanced tools available for conducting this static analysis, which are considered state of the art in the field.

B. Linux Scheduling

The Linux kernel has several scheduling policies available, some for RT tasks and other for standard processes. RT sporadic and periodic tasks can be scheduled using the `SCHED_DEADLINE` policy. This policy is an implementation of the earliest deadline first (EDF) scheduling algorithm, augmented with a constant bandwidth server to allow for better timing control (see [20] to dive deeper into the topic).

The `SCHED_DEADLINE` policy requires an admission test to check if there is room to guarantee the task WCET within the deadline, every period. This means that the task (t_{new}) has to pass the multiprocessor global EDF *task admission test* to be placed in the scheduling pool of RT tasks

$$\sum_{i}^{RT \cup \{t_{\text{new}}\}} \frac{\text{WCET}_i}{P_i} \leq M * \frac{\text{sched\_rt\_runtime\_us}}{\text{sched\_rt\_period\_us}} \tag{3}$$

where RT is the set of running RT tasks, WCET_i and P_i are the WCET and period of the tasks, respectively, M is the number of CPU cores, and *sched_rt_runtime_us/sched_rt_period_us* represents the maximum allowed utilization of the CPU for RT tasks (user-defined Linux kernel variable equal to 95% by default).[2]

[2]In (3), runtime is used instead of WCET due to Linux documentation terminology.

C. Container Orchestration

Kubernetes is an open-source container orchestration platform that simplifies the management of containerized applications. It works by coordinating and distributing workloads across a cluster of devices, i.e., the *computing nodes*, ensuring efficient resource utilization and high availability.

The standard Kubernetes architecture is composed of a single *master* and one *kubelet* unit per cluster node. Each kubelet contains one or more containers of ROS tasks. The master serves as the central control plane, overseeing the state of the cluster through a controller manager, a database of the cluster information (ETCD), and a scheduler unit. The scheduler manages the container deployments across the cluster nodes. The functional units (i.e., master and kubelets) communicate through the HTTP REST protocol. The master manages the kubelets requests through an API server [21].

III. Related Work

Container-based edge-fog-cloud systems have been investigated in several works (e.g., [22], [23], [24], [25]). The experimental findings demonstrate that the *edge-cloud computing continuum* and container-based virtualization, when combined together, improve scalability, resource usage, and performance. Containers have minimal to no performance overhead, but caution is required when many containers access the same shared resources [26]. Recently, various architectures for cyber-physical systems (CPSs) based on containers (e.g., Docker) and using ROS as middleware have been investigated [27]. These works have shown the potential to improve information flow among various network levels and increase software modularity.

Containerization combined with orchestration has been investigated in fog and edge computing [28], [29], [30], [31]. Different strategies of microservice deployment can be adopted to improve the performance, energy efficiency, and carbon footprint ([28], [32]), as well as the QoS ([29], [31]).

Several research works have been done to improve the performance of robotic applications. In this direction, SeART [33] is a framework that can intelligently schedule RT tasks by taking into account the current context to activate the minimum-cost tasks. Other RT robotic applications also show the need for improving performance to guarantee RT constraints [34]. The issues tied to containerization in a fog-based system for robotic applications have also been explored. In [6], the authors highlight the lack of fog-based frameworks to satisfy the RT demands of robotic applications.

Containers with RT constraints (i.e., RT containers) have been the focus of several recent studies due to the increasing adoption of container-based virtualization. The review in [35] explores existing solutions that guarantee RT constraints when working with containerized applications. The authors underline the lack of tools for RT container management and analysis on communication between RT containers. Legacy applications with RT constraints have been successfully emulated using containers [13] and the performance overhead is low enough

to run containerized RT applications for industrial automation applications [8]. RT technologies (RT OS ResinOs, Ubuntu Core, Co-kernel Xenomai 3, and Ubuntu with Preempt_rt software patch) have been tested together with containerization to demonstrate that container isolation is a new, competitive paradigm that allows for better resource usage when combined with RT [36]. The Linux scheduler `SCHED_DEADLINE` was compared to the *cgroups* policy (i.e., the main technology used to enable containers) to analyze which technology better handles RT application resources [16]. The scheduler is consistently more reliable and achieves better results than *cgroups*. This is also true in resource-constrained situations caused by high system load. Extensions of the Linux kernel have been proposed to improve the efficiency of RT scheduling of *cgroups* [15]. The same authors also proposed a modification of the Kubernetes source code to include some RT constraints [37].

High-performance computing (HPC) makes wide use of containers. A framework has been proposed to efficiently schedule RT containers in many-CPU systems [38], but additional research is needed to determine the effects of the operating system and I/O on deadlines. The feasibility of migrating RT applications from bare-metal servers to virtualized IaaS configurations for Industry 4.0 has also been explored [17].

A framework for applications in MCSs has been proposed in [14] to reduce interference between tasks when an application exceeds its WCET. The framework shows low efficiency due to the adopted static priority scheduler and it supports only one computing node. In subsequent work [39], it has been extended to use dynamic priority scheduling with a bandwidth server to improve performance.

To fulfill the need of safety-critical RT systems and streamline the certification process, the authors in [40] examined the benefits of utilizing both SGX isolation and unikernel features. Another proposed framework for RT orchestration introduces extensive modifications of the Kubernetes code-base and uses a unique patch for the Linux kernel to deploy best-effort and RT tasks [41].

Several works [42], [43], [44] show how, in edge-cloud computing architectures, live container migration can improve performance when resource usage can be monitored and utilization spikes mitigated by migrating services to a different computing node.

Unlike previous research, such as [37] and [41], this work proposes a platform for container orchestration onto edge-cloud architectures for MCSs based on off-the-shelf technologies (i.e., Linux OS + Preempt-RT, Kubernetes-K3S). The platform is ROS- and Kubernetes-compliant and does not require any custom software patch to be used; thus, it is supported in provisioned installation of Kubernetes, as well as normal environments. It supports per-node WCET, different levels of criticality (e.g., A, B, and C for RT containers), RT monitoring of all resources and overrun deadlines, and stateless migration of ROS tasks based on customizable policies. This framework allows to better meet the functional and extra-functional constraints of advanced multidomain software which are typical of autonomous mobile robots.

IV. RT-Kube Platform

The standard Kubernetes architecture does not support any notion of RT containers, as it does not have the data structures or the modules required to handle the additional requirements of such containerized RT tasks. This means that these RT tasks will be treated equally, even though their requirements are different.

Fig. 2 presents an overview of the proposed extended architecture. RT-Kube evolves the standard platform with the following components (highlighted in bold in Fig. 2).

1) *Custom Resource Definition Module for RT Containers (RT CRD)*. It allows for the specification of RT parameters, such as deadline, period, WCET, and criticality level of containers (Section IV-A).
2) *Secondary RT Scheduler*. It implements a scheduler with RT plugins that uses the additional data (RT CRD) to perform the schedulability test of RT containers (Section IV-B).
3) *Container-Level and Cluster-Level Monitors of RT Tasks*. These implement the monitoring of RT containers at two levels. At the container-level, the monitors collect information at runtime about temporal violations (i.e., overrun WCET and missed deadlines). At the cluster-level, one main monitor combines such information to the cluster status to implement container migration and recover from temporal violations (Section IV-C).

We implemented the nonisolation of containers to support communication among containerized ROS tasks. In standard ROS environments, nodes communicate through IP addresses and port numbers, where the IP corresponds to the device IP in the network and ports are assigned randomly. This allows communication and synchronization of ROS tasks to be easily implemented also when they are distributed on different devices of the computing cluster. In contrast, standard containers require the association to private (isolated) subnet IPs [45]. To tackle such a communication issue among *containerized ROS tasks* distributed across different cluster devices, the proposed solution implements the *nonisolation of containers*. All containers are launched with access to the networking interfaces of the host (through the option "hostNetwork: true" under Kubernetes). This eliminates any network overhead introduced by the containers [46]. It also removes the network address translation (NAT), which is not required in our target applications.

A. CRD Module for RT Containers

In standard orchestration platforms, the user provides a set of specifications for each container (e.g., memory, CPU, storage requirements). To schedule RT-containers, we consider four additional specifications: 1) criticality level, 2) deadline, 3) period, and 4) the WCET of the corresponding containerized RT task. The platform takes advantage of this information to calculate the utilization of all containers in each node, and assesses the impact of a new RT container deployment on the system performance.

Fig. 3 shows an example of CRD module (lines 1–13) with the extended specifications for the deployment of the Kubernetes *nginx* use case [47]. The first 13 lines create the CRD object "`example-realtime-data`," where lines 6 to 13 contain

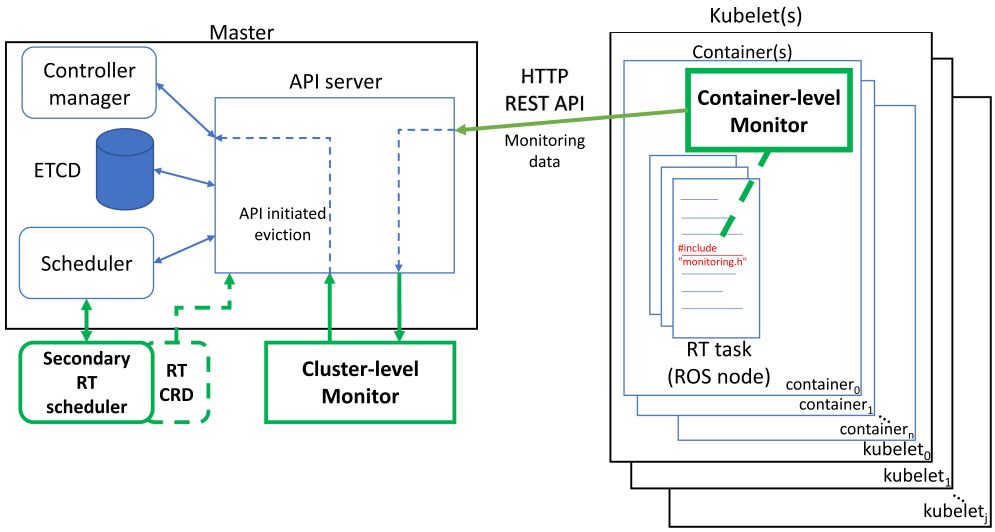


Fig. 2. RT-Kube overview.

the RT CRD with the four additional values. *Criticality (A, B, C, and D)* is used for the new sort and score phases, as well as in the monitoring (see Fig. 4 and Section IV-C). We borrowed such a criticality classification from the automotive safety integrity levels (ASIL) standard, with C being the highest the platform supports as of now. Note that it would be possible to support ASIL-D tasks, but it would require a compute platform that honors ASIL-D requirements as well. Further, criticality standards from other domains could also be adopted in a similar way.

The platform uses the deadline, period, and WCET information for the sorting, filtering, and scoring phases (see Section IV-B).

The CRD object is linked to the *deployment* object (*Kind* fields, lines 2 and 16) through a label that matches the RT specification at lines 4 and 30. These labels are compared at scheduling time for each deployment to find the matching RT CRD.

B. Secondary RT Scheduler

Fig. 4 shows an overview of the secondary RT-scheduler. The orchestration begins with the queue of containers (i.e., standard and RT), and the list of nodes representing the compute platforms (i.e., the *cluster* of computing devices).

We assume that each RT ROS task is mapped to one RT-container. Our evaluation shows that this one-to-one configuration, when compared to other solutions, is the most flexible as it incurs negligible overhead, and it experiences minimal performance penalties for RT tasks (see Section V-A). The scheduler implements the following four steps. First, *sorting* of containers using a hierarchical scheduling policy to generate a priority queue ordered by criticality. Then, for each container in the queue, the *filtering* phase selects the nodes of the cluster that satisfy the container specifications (Section IV-B1). The *scoring* phase creates a node ranking by considering user-defined policies, and schedules the container on the highest ranking node (Section IV-B2). Finally, the *binding* phase maps the container to the cluster node by allocating resources on the identified node for the container. To deploy RT-containers, the platform relies on the *taint* and *tolerations* features of Kubernetes [47] to identify which nodes of the cluster run an RT operating system (Fig. 3, line 38). Nodes with no RT operating system or RT capabilities are automatically excluded from the pool of schedulable nodes for RT tasks.

1) Filtering of Nodes: The platform applies (3) extended to the containerized version of tasks to implement the *container admission test*. Algorithm 1 depicts the filtering phase, which relies on such a container admission test. The algorithm takes as input the list of nodes N of the cluster, and the container of task t_{new} that has to be scheduled, x. For each node of the cluster (line 3), the algorithm considers all the containers currently running (i.e., already deployed) in the node and sums up the utilization of each container (line 4). The result (currentUT$_n$) represents the left-hand side of (3) extended to containers. The algorithm calculates the projected total utilization of the node (newUT$_n$) by considering the additional resources of container x under deployment (line 5). If the resulting projected utilization is greater than the threshold $n.\mathsf{thresholdUT}$ [which represents the right-hand side of (3) extended for containers], the algorithm filters the current node from the pool of schedulable nodes. The algorithm also uses an XOR operator (line 8) to check whether both the node and task criticality are C. If only one of the two is C, the node n is marked as not schedulable.[3]

2) Scoring of Nodes: To implement the scoring phase, the platform relies on the following equation to obtain a normalized ranking (nRank) for each node:

$$\forall n \in N: \quad \text{nRank} = \begin{cases} 1 - \frac{n.\mathsf{thresholdUT} - \text{newUT}_n}{n.\mathsf{thresholdUT}} & \text{if crit.} = A \\ \frac{n.\mathsf{thresholdUT} - \text{newUT}_n}{n.\mathsf{thresholdUT}} & \text{if crit.} = C \end{cases} \tag{4}$$

[3]In the Kubernetes terminology, a *nonschedulable* node is a cluster node that does not satisfy the requirements of the container that has to be scheduled.

```yaml
1   apiVersion: rt.scheduling/v1
2   kind: RealTime
3   metadata:
4     name: example-realtime-data
5   spec:
6     criticality: C
7     rtPeriod: 100
8     rtDeadline: 100
9     rtWcets:
10    - node: nodeA
11      rtWcet: 50
12    - node: nodeB
13      rtWcet: 60
14  ---
15  apiVersion: apps/v1
16  kind: Deployment
17  metadata:
18    name: nginx-deployment
19    labels:
20      app: nginx
21  spec:
22    replicas: 1
23    selector:
24      matchLabels:
25        app: nginx
26    template:
27      metadata:
28        labels:
29          app: nginx
30          rt-scheduling: example-realtime-data
31      spec:
32        schedulerName: RT-scheduler
33        containers:
34        - name: nginx
35          image: nginx:1.14.2
36          ports:
37          - containerPort: 80
38        tolerations:
39        - key: "RealTime"
40          operator: "Equal"
41          value: "RT"
42          effect: "NoSchedule"
```

Fig. 3. Example of RT CRD (lines 1-13) and the corresponding Kubernetes configuration module for the container deployment configured with the real-time parameters.

Algorithm 1: Real-Time Filtering Extension for the Kubernetes Scheduler.

input : A list of nodes N, an RT-container x
output: A list of schedulable nodes M

1 **Function** RealTimeFilterPlugin (N, x)**:**
2 $M \leftarrow N$
3 **for** $\forall n \in N$ **do**
4 $\text{currentUT}_n \leftarrow \sum_{c \in n} \frac{\text{WCET}_c}{P_c}$
5 $\text{newUT}_n \leftarrow \text{currentUT}_n + \frac{\text{WCET}_x}{P_x}$
6 **if** $\text{newUT}_n > n.\text{thresholdUT}$ **then**
7 $\text{M} \leftarrow \text{M} - \{n\}$
8 **else if** $(x.criticality = \text{“C”} \oplus n.criticality = \text{“C”})$ **then**
9 $\text{M} \leftarrow \text{M} - \{n\}$
10 **end**
11 **end**
12 **return** M

where $n.\text{thresholdUT}$ represents the threshold RT utilization (M∗*sched_rt_runtime_us*/*sched_rt_period_us*), and newUT_n represents the projected RT utilization after the deployment of x in n.

With a normalized utilization value [0,1] for each node, independent of the total runtime and CPU cores available on the node, the platform applies a custom policy for scoring based on the

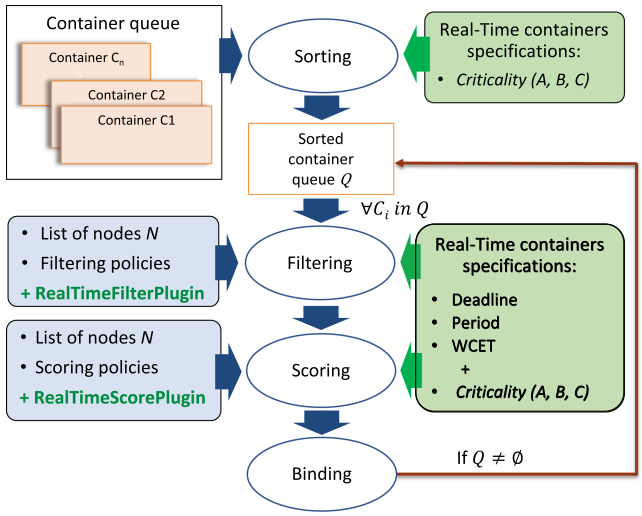


Fig. 4. Secondary RT-scheduler.

value of the *criticality* field in the RT specification extension (see Algorithm 2). If a task has a criticality level of A, the algorithm assigns the task to the node with the highest RT load (i.e., the node with the highest normalized utilization), line 3. In contrast, for level C, the algorithm gives the highest rank to the node with the lowest normalized utilization (line 11). For the criticality level B (line 5), the algorithm maps the utilization to a function that gives the highest rank to the nodes with a utilization level closest to $1/K$, as follows:

$$\forall n \in N: \ \text{nRank} = \begin{cases} h\left(\frac{n.\text{thresholdUT} - \text{newUT}_n}{n.\text{thresholdUT}}\right) & \frac{n.\text{thresholdUT} - \text{newUT}_n}{n.\text{thresholdUT}} \leq \frac{1}{K} \\ g\left(\frac{n.\text{thresholdUT} - \text{newUT}_n}{n.\text{thresholdUT}}\right) & \text{otherwise.} \end{cases} \tag{5}$$

Scoring with this function allows us to modify the deployment behavior to best fit the needs and requirements of the tasks. For example, we can make a criticality B task resemble the behavior of a criticality A task with $K \to 1^-$ and a linearly increasing $h(\cdot)$, or closer to C with $K \to 0^+$ and a linearly decreasing $g(\cdot)$. In our experiments, we applied (5) with $K = 2$, a linear function $h(\cdot) = a(\cdot) + b$, and a quadratic function for $g(\cdot) = a'(\cdot)^2 + b'(\cdot) + c'$. This allows us to linearly increase rank for nodes that have newUT_n lower than $1/2$, but greater than 0, and then a sharp decrease once that the threshold utilization value is reached. Fig. 5 shows the corresponding mapping, whereby the nodes with average RT load are classified as nodes with the highest ranking.

C. Container-Level and Cluster-Level Monitoring of RT-Containers

RT-Kube implements the monitoring of RT containers at two levels. At container-level, one monitor per container collects temporal information of the corresponding RT task and reports, at runtime, any temporal violation (i.e., overrun WCET

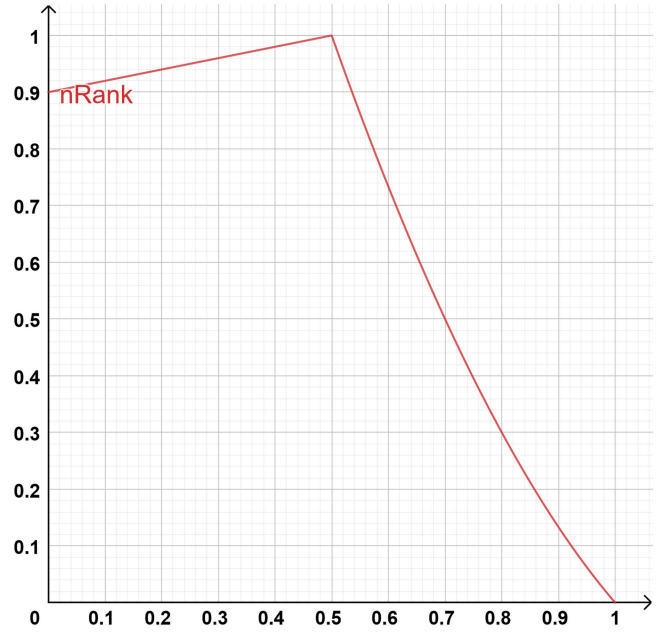


Fig. 5. Mapping results with (5) for the criticality level B with the coefficients adopted in the experimental results ($a = \frac{1}{5}$, $b = \frac{9}{10}$, $a' = \frac{5}{3}$, $b' = -\frac{9}{2}$, $c' = \frac{17}{6}$).

Algorithm 2: Real-Time Scoring Extension for the Kubernetes Scheduler.

input : A node n, the RT container x, the projected utilization newUT_n
output: The score s for RT container x on node n

Function `RealTimeScorePlugin` (n, x, newUT_n):
  $\mathsf{nRank} \leftarrow \frac{n.\mathsf{thresholdUT} - \mathsf{newUT}_n}{n.\mathsf{thresholdUT}}$
  if $x.criticality$ = *"A"* **then**
    $s \leftarrow 1 - \mathsf{nRank}$
  else if $x.criticality$ = *"B"* **then**
    if $\mathsf{nRank} \leq \frac{1}{K}$ **then**
      $s \leftarrow h(\mathsf{nRank})$
    else
      $s \leftarrow g(\mathsf{nRank})$
    end
  else if $x.criticality$ = *"C"* **then**
    $s \leftarrow \mathsf{nRank}$
  end
return s

or missed deadline) to the cluster-level monitor. This last implements the *reconcile phase*, which consists of checking the temporal constraints (i.e., threshold of missed deadlines) and eventually migrating containers to free resource and recover the system. To guarantee portability, each container-level monitor takes advantage of the `SIGXCPU` signal [48] that any Linux operating system can raise when a temporal violation occurs. Once the monitor receives such a signal, it communicates the updated counter of missed deadlines to the cluster-level monitor, which in turn implements the corresponding orchestration countermeasures. The injection of monitors in the SW application does not require any modification to the source code. Fig. 6

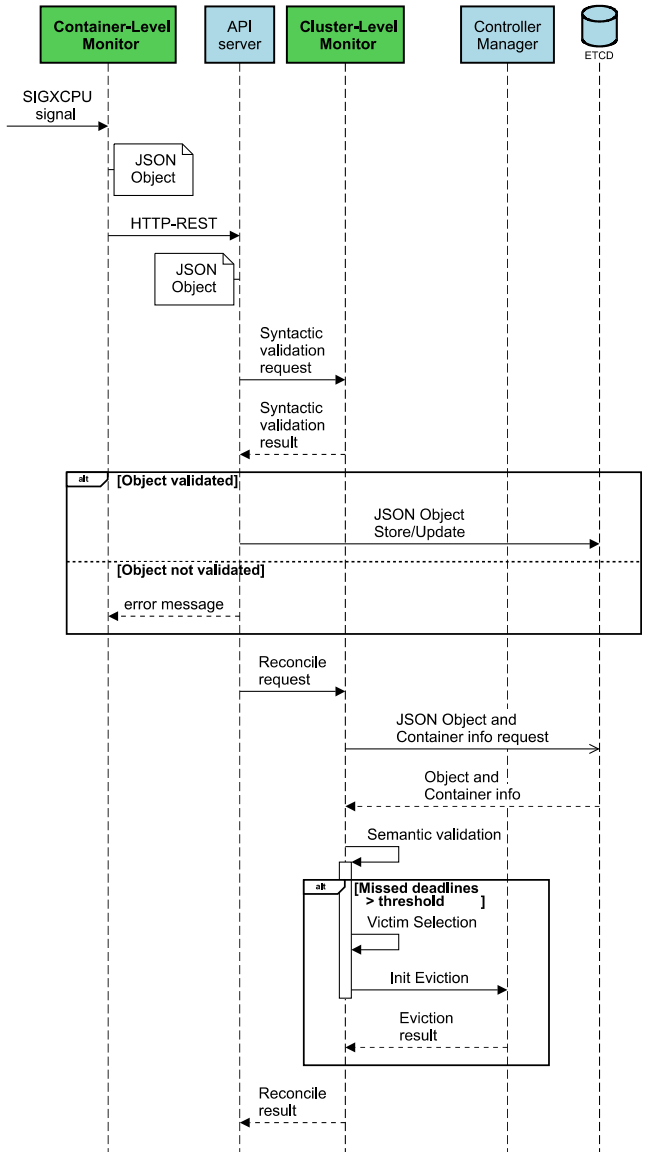


Fig. 6. Sequence diagram of the whole dynamic SW orchestration, starting from the container-level monitoring units.

depicts the sequence diagram of the whole SW orchestration, starting from the container-level monitoring units. The figure reports the components of the standard Kubernetes release and the extension. The extension components are highlighted in bold. To implement a continuous runtime monitoring while saving computational resources, we implemented each container-level monitor unit through two threads. The first receives the `SIGXCPU` signal and updates the counters for the missed deadlines. The second communicates the RT container status (i.e., the updated number of missed deadlines) to the API server periodically through the HTTP-REST protocol. The container status is encoded into a Kubernetes compliant JSON object (see Fig. 7) to guarantee modularity and scalability of the system. The API server validates the object syntactically (validation request in Fig. 6) and updates the RT container status in the ETCD database. It then requests for the system-level check to the cluster-level monitor (i.e., reconcile request).

```
1 apiVersion: rt.monitoring/v1alpha1
2 kind: Monitoring
3 metadata:
4   name: monitoring-test
5 spec:
6   node: nodeA
7   containerName: nginx
8   missedDeadlinesPeriod: 0
9   missedDeadlinesTotal: 0
```

Fig. 7. Example of monitoring object created by the container-level monitor of RT-tasks.

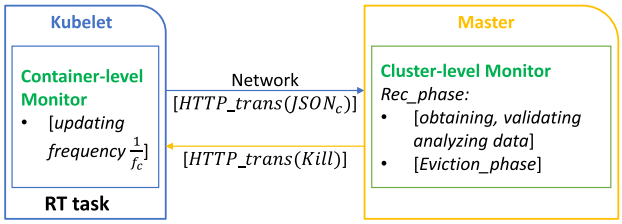


Fig. 8. RT-Kube architecture considered for the analysis and formalization of the response time.

The cluster-level monitor implements the reconcile phase. It collects the updated status of missed deadlines from each container-level monitor of the cluster. The updated status consists of the number of deadlines missed since the last update and the total number of missed deadlines in the RT container lifetime. If the number of missed deadlines, either in the period or total, is higher than the user-defined threshold (obtained from the centralized ETCD database), the cluster-level monitor selects a container to be immediately evicted and killed. The choice relies on the policy statically defined by the user. In our implementation, aside from RT containers with criticality C, any container with a lower criticality than the one with temporal violations and that has a stateless task can be selected. The cluster-level monitor notifies the eviction target to the controller manager (initialize eviction in Fig. 6), which implements the *container eviction* and automatic redeployment as for the standard protocol of Kubernetes' controllers. The monitor also implements *tainting* of the node that hosted the evicted container, which consists of marking the node as not available to host new containers for a period of time. This allows us to avoid immediate deployments of new containers in the node where the missed deadlines were observed. As a consequence, the evicted container is redeployed on a different cluster node, thus implementing a de-facto stateless migration.

D. Response Time and Predictability of the System Recovery From Temporal Violations

We define *response time* (RT_i^c) of the system recovery from temporal violations of the RT container c on the cluster node i, as the time elapsed from the first deadline missed by c that leads c to exceed the threshold of missed deadlines to a container eviction on i. It is characterized by three components (see Fig. 8)

$$RT_i^c \leq \frac{1}{f_c} + \text{HTTP\_trans}(\text{JSON}_c) + \text{Rec\_phase} \qquad (6)$$

where f_c is the frequency the monitor in container c updates the master with the total number of missed deadlines, HTTP_trans(JSON$_c$) is the time spent for transferring the JSON object containing the counter information from node i to the master; Rec_phase is the time spent by the master for the reconcile phase.

Each cluster-level monitor updates the counter of missed deadlines at every SIGXCPU signal locally, while it updates the master periodically to save system resources. The period between two counter updating defines the worst-case delay [i.e., first component of (6)].

We assume that the cluster-level monitor and the Kubernetes master are hosted on the same cluster node. As a consequence, we consider the communication latency between monitor and master being negligible. In contrast, we consider the time spent for transferring the updated data from the container-level monitor to the master over the network. This latency strongly depends on the static and dynamic characteristics of the communication network (i.e., bandwidth, traffic, etc.). Predictable networks for RT applications have been extensively studied in literature [49], and could be considered to increase the predictability of such a response time component. Nevertheless, we generalize the definition and consider HTTP_trans(JSON$_c$) as the worst-case time spent for transferring the JSON object containing the counter information (e.g., ≈ 180 B in our case study) to the master over HTTP.

The third component represents the time spent by the master for the reconcile phase. The master accesses the ETCD database for the semantic validation of the updating message (i.e., the JSON object), the request for specifications of each container (i.e., memory, CPU, storage requirements) and the additional specifications for RT containers (i.e., CRD module with deadline, period, WCET, criticality). Using this information, the master implements the victim selection and eviction through a message (i.e., standard killing Kubernetes message) over the network. We model the latency of the reconcile phase as follows:

$$\begin{aligned}\text{Rec\_phase} \leq\ & 2 \cdot n \cdot t_{\text{SPEC}} + n_{RT} \cdot t_{\text{CRD}} \\ & + \text{Eviction\_phase} \\ & + \text{HTTP\_trans}(\text{Kill})\end{aligned} \qquad (7)$$

where n is the total number of containers, n_{RT} is the number of RT containers ($n_{RT} \leq n$), t_{SPEC} and t_{CRD} are the latencies spent for retrieving the container and RT container specifications from the database. HTTP_trans(Kill) represents the latency spent for notifying a killing procedure by the master to the cluster node operating system. For this component, the considerations formulated before on the transfer time of a Kubernetes message over HTTP apply. The time for the eviction phase strictly depends on the complexity of the algorithm implementing the victim selection. We implemented three policies with different complexity. The first relies on an iterative linear search over the list of containers deployed on node i to find the container with the highest use of a single system resource (i.e., either CPU or memory). It starts from the lowest priority class of containers (i.e., nonreal-time) and, in case none of them is deployed in i, it iterates on the lists of the higher priority containers.

The second policy implements a similar search, by considering the combination of two system resources. Being based on reordering and search phases, its complexity is linearithmic on the number of containers (i.e., $n \cdot \log(n)$, with n the number of containers). The third policy considers intercontainer communication dependencies and task semantic, and has quadratic complexity (n^2). We present a comparison among the three policies in terms of response time in Section V-D.

V. Results

We evaluated the RT-Kube efficiency on the software that implements the mission of a Robotnik RB-Kairos, which is a skid-steering mobile platform equipped with a Universal Robots UR5 and a Schunk WSG50 gripper. The software is distributed on computing cluster that consists of three nodes (see Fig. 1): two on-board programmable devices, i.e., an NVIDIA Jetson Xavier with the RT operating system (Linux kernel $v.5.10$, and the $preempt\_rt$ patch) and a desktop with an i7 9700 locked at 3.0 GHz with 8 GB RAM and a standard *nonreal-time* Linux-based operating system. The onboard nodes communicate through a Gigabit Ethernet switch (802.3ab). The third node consists of an off-board desktop with an octa-core CPU and 16 GB of RAM, with Linux kernel $v.5.10$, and the $preempt\_rt$ patch. It communicates with the other (on-board) cluster nodes through WiFi (802.11ac).

We first evaluated empirically the impact of containerization on RT tasks in terms of WCET overruns and missed deadlines (Section V-A). We then compared the efficiency of the proposed Kubernetes extension to support RT-containers in terms of task rejections and missed deadlines with a synthetic software benchmark (Section V-B) and with the real software implementing the robot mission (Section V-C). Then, we analyzed the response time of the system recovery from temporal violations achieved by RT-Kube (Sections V-D and V-E).

A. Impact of Containerization on RT-Tasks

Workload Configuration: We evaluated the impact of containerization on RT tasks by using a large set of standard software benchmarks for RT tasks. For brevity, we only report the results obtained with the $cyclicdeadline$ benchmark from *rt-tests* [50] (the results with the other benchmarks show similar behavior). The benchmark is configured to run with either 1, 2, 4, or 8 identical tasks, maximum utilization of 95% per task, WCET of 3.8 ms, period of 4.0 ms, and deadline equal to the period. Note that we set these parameters to experience a full workload capacity on the system, which allows us to assess the performance impact of containerization. We evaluated three scenarios: 1) all RT tasks running natively on the edge platform to establish a baseline performance metric; 2) all RT tasks running in one container to analyze whether an application composed of multiple RT tasks is affected by the overhead; 3) all RT tasks running, where each RT task is mapped into its own container to evaluate how the overhead of containerization scales with multiple isolated tasks and if the overhead caused by a high number of containers can interfere with RT deadlines. We also run these three scenarios with additional tasks that overload the system with memory accesses to assess the behavior of the RT tasks under stress.

Key Results: Table I shows the average actual runtime among n tasks, observed WCET among all the n tasks, total WCET overruns, and total of missed deadlines for all n tasks. Columns with "stress" identify those scenarios where the system was overloaded with memory accesses.

Our results indicate that containerization does introduce overhead on observed WCET, and that such an overhead is evident only in the higher CPU-load configurations, i.e., four or more tasks on the 8-core CPU. However, containerization does not impact the average-case runtime of tasks with or without stress. When compared to tasks running natively on the device, containerization impacts the WCET overruns and missed deadlines only when the RT utilization is close to 100% (last two rows of the table). This is due to the fact that when the task overruns the deadline and all CPU cores are allocated to RT tasks, the system cannot remap the task to a different available core. Interestingly, the number of containers created to group the RT tasks does not influence the observed WCET as, both when using one container and n containers, the observed times are similar. This is highlighted by the increased value of observed WCET (both with or without stress) by around 9.5% when the tasks are grouped into one container w.r.t. the native configuration, while the value is comparable between 1 container and n containers with any number of tasks. We observed the same behavior for the WCET overruns and missed deadlines. The task WCET was overrun 28 times with eight tasks in one Docker container, and this led to 24 missed deadlines over 8 millions (i.e., 0.00035%). The values do not increase by increasing the number of containers.

Summary: We noticed a negligible overhead when containerizing applications in general. Only at maximum utilization (e.g., eight tasks over 8-CPU cores with 95% utilization) does the overhead lead to missed deadlines. With appropriate provisioning, we contend that containerization with the proposed off-the-shelf technology can support RT applications. In our experimental analysis, we obtained no missed deadlines for the containerized configurations also with eight tasks by slightly relaxing the deadline w.r.t. to the period (e.g., increasing the period from 4.0 to 4.5 ms, and the WCET from 3.8 to 4.3 ms, still with 95% maximum utilization per task).

B. Benchmarking the Orchestration With RT Support

Orchestration Platform Configurations: We used the following three configurations for the orchestration platform.

1) *Native K3S-Standard:* The standard Kubernetes configuration for orchestration.
2) *Native K3S-Best Configuration:* The *best* configuration for orchestration with the native Kubernetes scheduler. We statically and manually assign an optimal task-to-node orchestration solution.
3) *RT-Kube:* Our proposed orchestration platform with the extended RT specifications and secondary RT scheduler.

Workload and Deployment Setups: The workload consists of 73 tasks. 72 of them have been containerized into 72 RT-containers. The remaining task has been containerized into a

TABLE I
COMPARISON OF RT-TASKS RUNNING NATIVELY, CONTAINERIZED IN A SINGLE AND MULTIPLE DOCKER CONTAINERS

# tasks	config.	average runtime (ms)	average runtime with stress (ms)	Observed WCET (ms)	Observed WCET with stress (ms)	WCET overruns (#)	WCET overruns with stress (#)	missed deadlines (#)	missed deadlines with stress (#)	total deadlines per config (#)
1	native	1.219	1.256	3.143	3.655	0	0	0	0	30k
	1 container	1.164	1.172	3.055	3.119	0	0	0	0	
2	native	1.234	1.251	3.118	3.755	0	0	0	0	60k
	1 container	1.154	1.169	3.092	3.052	0	0	0	0	
	2 containers	1.144	1.162	3.091	3.090	0	0	0	0	
4	native	1.225	1.262	3.100	3.845	0	0	0	0	120k
	1 container	1.155	1.160	3.522	4.014	0	1	0	1	
	4 containers	1.134	1.154	3.783	3.704	0	0	0	0	
8	native	1.187	1.204	3.565	4.120	0	9	0	5	8M
	1 container	1.155	1.146	4.337	4.511	28	23	24	21	
	8 containers	1.140	1.141	4.405	4.322	26	18	10	14	

TABLE II
RT CONTAINERS DISTRIBUTION FOR THE RT ORCHESTRATION

	1st deployment (#)	2nd deployment (#)
Criticality A	32	16
Criticality B	24	24
Criticality C	16	32
Stress	1	1

non-RT container and implements stress activity through memory accesses on the platform.

All containers have one instance of "cyclicdeadline," the WCET has been set to 2.9 ms for all tasks, and the deadlines at 24.2, 18.1, and 14.5 ms for levels A, B, and C, respectively. The resulting RT utilizations are: 12% for A, 16% for B, and 20% for C. We tested two deployment setups, shown in Table II. With these configurations and deployments, an improper orchestration would result in higher than 100% RT utilization on a single cluster node, and, thus, rejected tasks.

Key Results: Table III summarizes the workload and deployment setup in the first three columns. The fourth column shows the number of RT tasks that have been mapped into the corresponding RT nodes by the orchestrator, but that have been rejected by the Linux kernel admission test [see (3)]. The last column shows, for all criticality levels, the number of missed deadlines. This column reports the deadlines missed by the running tasks (not rejected) and the total missed deadlines (those missed by the running + those missed because of the task rejection).

We found that the orchestration of the standard Kubernetes produces a task-to-node mapping that is inadequate for RT tasks. This is underlined by the amount of RT tasks that, after mapping, have been rejected by the Linux Kernel RT admission test and by the number of missed deadlines w.r.t. the best (manually configured) orchestration configuration.

Table IV reports the efficiency comparison between the proposed solution when compared with the native Kubernetes, with a dynamic workload. The benchmark implements a sequential number of RT container deployments. We set up a first scenario

TABLE III
COMPARISON AMONG KUBERNETES STANDARD ORCHESTRATION, BEST ORCHESTRATION, AND THE PROPOSED RT-KUBE

De-ploy-ment	Config.	Critical tasks/ Nodes	Tasks rejected	Deadlines missed (running tasks - running+rejected)
1st	Native K3S-standard	A: J(32) B: J(24) C: J(16) oth.: J(1)	A: 8 B: 7 C: 6	A: 3.3% - 27.5% B: 9.1% - 35.6% C: 6.5% - 41.5%
	Native K3S-best config.	A: I(32) B: I(24) C: J(16) oth.: R(1)	A: 0 B: 0 C: 0	A: 0.0% - 0.0% B: 0.0% - 0.0% C: 0.0% - 0.0%
	RT-Kube	A: I(32) B: I(24) C: J(16) oth.: R(1)	A: 0 B: 0 C: 0	A: 0.0% - 0.0% B: 0.0% - 0.0% C: 0.0% - 0.0%
2nd	Native K3S-standard	A: J(16) B: J(24) C: J(32) oth.: J(1)	A: 8 B: 10 C: 10	A: 0.0% - 50.0% B: 2.9% - 44.6% C: 4.3% - 34.2%
	Native K3S-best config.	A: I(16) B: I(24) C: J(32) oth.: R(1)	A: 0 B: 0 C: 0	A: 0.0% - 0.0% B: 0.0% - 0.0% C: 0.0% - 0.0%
	RT-Kube	A: I(16) B: I(24) C: J(32) oth.: R(1)	A: 0 B: 0 C: 0	A: 0.0% - 0.0% B: 0.0% - 0.0% C: 0.0% - 0.0%

TABLE IV
EXPERIMENTAL RESULTS FOR DYNAMIC ORCHESTRATION

Configuration	T_0		$T_0 + t_\alpha$	
	Deployed	Pending - Rejected	Deployed	Pending - Rejected
Native-K3S	N_1: 8 N_2: 8	1 - 0	N_1: 9 N_2: 8	0 - 1
RT-Kube	N_1: 8 N_2: 8	1 - 0	N_1: 8 N_2: 8	1 - 0
Native-K3S	N_1: 8 N_2: 0	1 - 0	N_1: 9 N_2: 0	0 - 1
RT-Kube	N_1: 8 N_2: 0	1 - 0	N_1: 8 N_2: 1	0 - 0

(first two rows of Table IV) in which 16 RT containers are deployed and run correctly onto the cluster for 100% RT utilization. In this context, there is CPU available for the (standard) orchestrator, while the CPU is fully loaded for the RT admission test [see (3)]. When a new RT container has to be deployed (instant $T_0 + t_\alpha$), the orchestrator of the native Kubernetes immediately deploys the container on the cluster. This leads to an overload of the cluster RT utilization (i.e., more than 100%) and, as a consequence, to a failed admission test by the Linux kernel. This *failing deployment* keeps getting restarted by Kubernetes. The admission test succeeds and the container is deployed only when RT resources of the same node become available and the new RT utilization is within 100%. Nevertheless, if any other different RT node becomes available, the issue persists as Kubernetes has no way of knowing the cause of the failure. In contrast, with the proposed solution, the new container remains pending *at orchestration level* until any RT resources in the cluster become available.

We set up a second scenario (last two rows of Table IV), in which the first RT node has 100% utilization, with eight containers, while the second RT node has 0% utilization. When a new RT container has to be deployed, with the native Kubernetes, there is only a 50% chance for the correct node to be picked, as the orchestrator has no knowledge of the RT utilization. In contrast, the proposed orchestration platform guarantees that the correct node (one with 0% RT utilization) is always selected.

C. Orchestration With RT Support of the Robot's Mission Software

Software Configuration: The robot's mission is implemented through an SW application composed of 80 ROS tasks. It performs pick and place operations, delivering goods from a conveyor belt to a storage area and vice-versa. The most critical part of the software is the driver controlling the arm operations. This task needs to maintain a control loop that communicates directly with the arm hardware at 120 Hz (8-ms deadline, 7.6-ms WCET, 90% utilization). Exceeding the deadline and delaying such a communication beyond a certain threshold causes the arm to go into safe mode and halting operations. This translates into a measured maximum lateness of 0.7 ms, after which the connection is dropped.

Key Results: As expected, we found that, with the native K3S orchestration, all tasks were mapped onto the three devices randomly. This resulted in missed deadlines to exceed the threshold and to the safety stop of the robot. We then evaluated two alternative scenarios with the proposed RT orchestrator. The first supports multiple criticality levels (i.e., non RT tasks with A, B, and C RT tasks), while the second only supports non RT with RT tasks (e.g., [37] and [41]). In the first solution, we classified three tasks implementing the robot localization and mapping (SLAM) as RT criticality B. Then, the arm driver as an RT task with criticality C. We defined the edge server as an RT node for criticalities A and B, while the on-board Jetson for criticality C. In the second scenario, we classified the SLAM, as well as the arm driver, as RT. Both are deployed on the RT-capable Jetson onboard. In both scenarios the rest of the software was configured as non RT. These two test cases allow us to evaluate the improvement for the automatic separation of different classes of criticality as well as the benefits of RT orchestration.

TABLE V
EXPERIMENTAL RESULTS WITH THE RB-KAIROS

Config.	Avg runtime (ms)	Std. dev.	Observed WCET (ms)	Max. lateness (ms)	Missed deadlines: arm driver	Missed deadlines: SLAM
Native	7.15	3.58	32.32	-	24.50%	-
non-RT + RT [37], [41]	0.20	0.11	5.72	0.95	1.28%	$\approx 0\%$
non-RT + A + B + C	0.15	0.07	2.06	0.49	0.17%	$\approx 0\%$

Table V shows the results. Our solution based on multilevel criticality (third row in the table) manages to take advantage of criticality-aware load distribution, which allows the arm driver to perform substantially better than native and other alternatives, with lower observed WCET and reduced deadline misses. The much less restrictive control loop of the SLAM nodes allows for performance improvement in both solutions.

Summary: Unlike other alternatives, the proposed solution did not suffer from any safety-related stop as the maximum lateness bound was never exceeded. This is due to the isolation of the highest criticality tasks from the lower criticality ones implemented by the orchestrator. We also observed very different average runtime and WCET when comparing the native Kubernetes approach and RT-Kube. With the native Kubernetes, our case study caused an order of magnitude more missed deadlines w.r.t. RT-Kube, which make the native solution practically unusable.

D. Analysis of Response Time for Container Migration

Software Configuration: We first measured the response time of the system recovery from temporal violations implemented by RT-Kube [(6) in Section IV-D] through a large set of benchmarks, which consist of different numbers and distributions of non- and RT-containers on the cluster. We deployed the container-level monitors on each device of the cluster and, the cluster-level monitor on the K3S master node (i.e., the external server). In all tests, we set the updating frequency of the container-level monitors to 10 Hz. We measured, on average, 53 ms as the time elapsed from the missed deadline to the update instant [the first component of (6)].

Network Impact: We analyzed the time taken to transfer the updating messages from the container-level monitors to the master across the Ethernet+WiFi network, with and without network congestion. Without network congestion, we measured an average latency of ≈3 ms for both transmissions (HTTP_trans(JSON$_c$) and HTTP_trans(Kill)). With a congested network, we measured, on average, a communication latency of 9.9 ms for the two transmissions. We measured the worst-case transfer time to send each update message to the server as 115.2 ms (i.e., HTTP_trans(JSON$_c$)) and 84.99 ms to send the $HTTP\_trans(Kill)$ message to the node.

TABLE VI
CLUSTER-LEVEL MONITOR RECONCILE RESPONSE TIMES FOR VARIOUS COMBINATIONS OF RT SCHEDULING OBJECTS AND CONTAINERS

Contain-ers (#)	Cont. Distribution non-RT	–	RT	Reconcile phase (ms) Linear	Linearithmic	Quadratic
1	1	–	0	41.07	40.81	40.83
1	0	–	1	41.75	41.49	41.51
8	8	–	0	40.26	39.27	57.27
8	4	–	4	42.98	41.99	59.99
8	0	–	8	45.70	44.71	62.71
16	16	–	0	51.32	51.62	157.94
16	8	–	8	56.76	57.06	163.38
16	0	–	16	62.20	62.50	168.82
32	32	–	0	65.43	106.07	173.57
32	16	–	16	70.87	111.51	179.01
32	0	–	32	76.31	116.95	184.45
64	64	–	0	115.23	152.39	687.02
64	32	–	32	126.11	163.27	697.90
64	0	–	64	136.99	174.15	708.08

The congestion was obtained through 1 Gb/s of traffic from the master to the container-level monitor, and vice versa (obtained with the *netcat* and *pv* emulation software).

Key Results: Table VI reports the average time spent for the reconcile phase in the tested configurations. The first column indicates the total number of containers deployed in the cluster node. The second column reports the distribution of standard and RT containers. The last three columns report the time spent in the reconcile phase with the linear, linearithmic, and quadratic policies.

We found that the time required by the three policies is comparable with a low number of containers deployed in the cluster node (i.e., eight in our experimental setup). Linear and linearithmic still achieve comparable performance for up to 16 containers. With more containers, the different impacts of the three policies on the response time become evident. When analyzing how each policy is affected by the number of standard and RT container, we observed that, when moving from 1 to 32 *standard* containers, the latency of the reconcile phase increases by 59.3%, 159.9%, and 325.1% with the linear, linearithmic, and quadratic policies, respectively. When moving from 1 to 32 *RT* containers, the time increases by 82.3%, 181.9%, and 344.5%.

Summary: In general, the response time with both linear and linearithimc policies is less than 200 ms, in which, as expected, there is a negligible contribution of the transfer latency. The contribution of the monitor updating to the overall latency could become negligible at higher updating frequencies. The trade-off comes at the cost of more computational resources spent on operations, particularly on the master node. The *smarter* quadratic policy leads to a response time of around 750 ms. In general, RT containers impact the reconcile time slightly more than standard containers. Such an additional overhead is negligible and becomes even less relevant as the complexity of the algorithm grows. This is due to the fact that the overhead for each container is constant.

E. Analysis of Response Time for System Recovery With the Robot's Mission Software

Software Configuration: We measured the response time of the system recovery with the robot's mission software. The 80 ROS tasks implementing the robot's mission were organized into 13 containers, for a total of 18 containers by including the RT containers of the SLAM software and ARM drivers. The first two columns of Table VII show the configuration and distribution. We deployed all the non-RT containers on the robot's on-board i7 and all the RT containers on the on-board Jetson.

TABLE VII
CLUSTER-LEVEL MONITOR RESPONSE TIME ON THE AUTONOMOUS MOBILE ROBOT CASE STUDY

Cont. (#)	Container Distribution non-RT – RT	6 (# element)	AVG Time (ms)	WCET (ms)
18	14 – 4	(1) updating freq.	50.00	100.00
		(2) trans ($JSON_c$)	9.89	115.21
		(3) Rec_phase	23.47	108.09
		(3.1) trans (Kill)	8.67	84.99
		Total	92.03	408.29

Key Results: Table VII shows the results. The fourth column reports the average time and the last column reports the worst-case time for each component, with the total time in the last row. On average, the delay caused by the updating frequency is 50 ms, the communication both ways takes ≈20 ms, and the computation is ≈20 ms, with a total time of ≈90 ms. In the worst-case scenario, communication consumes half of the total response time, with one fourth being computation for the reconcile phase. The last one fourth is the configurable updating frequency, where the same considerations of Section V-D apply. Nonetheless, the observed worst-case response time, from detecting a missed deadline to the Kubelet on the target node receiving a command is below half a second (≈408 ms).

We also analyzed the CPU usage and the number of missed deadlines through Prometheus and Grafana (i.e., two popular monitoring and visualizing tools for Kubernetes). Fig. 9 shows the collected data. To showcase the cluster-level monitor eviction, we deployed an RT container with a CPU-intensive task to stress the system, bypassing the schedulability check on the Jetson and forcing the system into a particularly unsustainable scenario. At instant 36, Fig. 9(a) underlines the start of a sequence of CPU overload, while Fig. 9(b) reports the corresponding increase of missed deadlines. At instant 71, the missed deadlines cross the threshold and the cluster-level monitor starts the eviction process. The RT container of the CPU-intensive task was selected for eviction as it had the highest CPU utilization. In the following time instants, Fig. 9(a) and (b), depict the decrease of both CPU usage and missed deadlines. Meanwhile, the RT container of the CPU-intensive task was sent back to the scheduling queue and restarted on the server, as it is the only other node of the cluster with RT support.

During the eviction phase, the victim container is immediately killed and restarted on a different node. This results in the evicted task experiencing downtime due to the required cold-starting on the new node. RT-Kube migrates noncritical or less-critical tasks that steal resources to higher criticality tasks in devices where temporal violations are observed. In our system, all container images are already downloaded onto permanent storage for all nodes to avoid unpredictable network latency. This is possible thanks to the proposed multi, smaller, and modular container

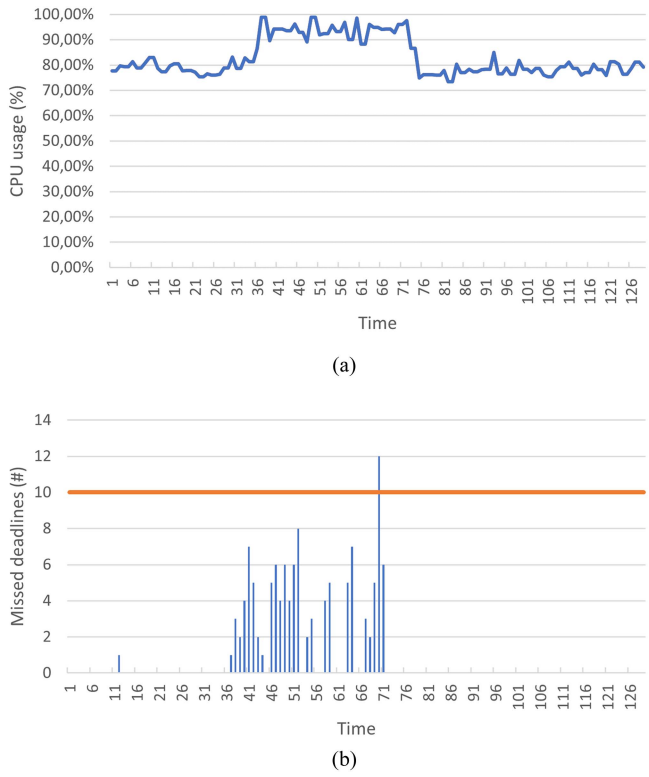


Fig. 9. Real-time monitoring on the Robotnik RB-Kairos. (a) CPU usage with the RT tasks for the Jetson Xavier on the Robotnik RB-Kairos. (b) Missed deadlines for the RT tasks for the Jetson Xavier on the Robotnik RB-Kairos.

organization of the software. With this configuration, the downtime is in average less than one second (less than 3 s in the worst case). Since it involves only "less-critical tasks," it is acceptable and does not involve any disruption.

VI. Conclusion

In recent years, the adoption of containerization and Kubernetes orchestration in resource-constrained edge computing environments has gained significant attention. This trend is not limited to RT computing systems but extends to domains, such as robotics and CPSs, where the complexity of AI-based software on modern autonomous platforms continues to grow. This article delved into this subject within the context of autonomous mobile robots, characterized by the deployment of mixed-criticality tasks across an edge-cloud computing architecture. It introduced an orchestration platform called RT-Kube designed for MCSs, which extends the capabilities of standard Kubernetes to seamlessly support RT containers. Through experimental results on synthetic benchmarks and a real case of study, the article shows that the container overhead, when deploying RT tasks on off-the-shelf embedded systems, is negligible. It also showed that RT-Kube can reduce the number of missed deadlines when deploying RT tasks by upto 50% compared to the standard Kubernetes scheduler, achieving improved reliability. It implements monitoring of potential deadline overruns with a 90-ms response time on average (400-ms worst case) and reduces the total number of missed deadlines by an order of magnitude, ensuring robust RT performance within Kubernetes-based edge-cloud environments.

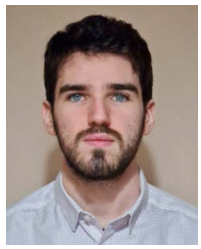
Acknowledgment

This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

References

[1] T. Iqbal, S. Rack, and L. D. Riek, "Movement coordination in human-robot teams: A dynamical systems approach," *IEEE Trans. Robot.*, vol. 32, no. 4, pp. 909–919, Aug. 2016.
[2] M. Piaggio, A. Sgorbissa, and R. Zaccaria, "A programming environment for real-time control of distributed multiple robotic systems," *Adv. Robot.*, vol. 14, no. 1, pp. 75–86, 2000.
[3] P. Thakur and V. Kumar Sehgal, "Emerging architecture for heterogeneous smart cyber-physical systems for industry 5.0," *Comput. Ind. Eng.*, vol. 162, 2021, Art. no. 107750.
[4] D Merkel et al., "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, 2014, Art. no. 2.
[5] P. Melo, R. Arrais, and G. Veiga, "Development and deployment of complex robotic applications using containerized infrastructures," in *Proc. IEEE 19th Int. Conf. Ind. Inform.*, 2021, pp. 1–8.
[6] M. Shaik et al., "Enabling fog-based industrial robotics systems," in *Proc. IEEE Symp. Emerg. Technol. Factory Automat.*, 2020, pp. 61–68.
[7] N. Nikolakis, R. Senington, K. Sipsas, A. Syberfeldt, and S. Makris, "On a containerized approach for the dynamic planning and control of a cyber-physical production system," *Robot. Comput.-Integr. Manuf.*, vol. 64, 2020, Art. no. 101919.
[8] M. Sollfrank, F. Loch, S. Denteneer, and B. Vogel-Heuser, "Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation," *IEEE Trans. Ind. Inform.*, vol. 17, no. 5, pp. 3566–3576, May 2021.
[9] AUTOSAR: AUTomotive Open System ARchitecture, Hörgertshausen, Germany, "The standardized software framework for intelligent mobility," 2021. [Online]. Available: www.autosar.org
[10] Collins Aerospace, "Connectivity and network services." Accessed: Oct. 1, 2023. [Online]. Available: www.arinc.com
[11] A. Burns and R. Davis, "Mixed criticality systems-a review," Dept. Comput. Sci., Univ. York, Tech. Rep. 2013, 2013, pp. 1–69.
[12] A. Burns and R. Davis, "A survey of research into mixed criticality systems," *ACM Comput. Surveys*, vol. 50, no. 6, pp. 1–37, 2017.
[13] T. Tasci, J. Melcher, and A. Verl, "A container-based architecture for real-time control applications," in *Proc. IEEE Int. Conf. Eng., Technol. Innov.*, 2018, pp. 1–9.
[14] M. Cinque, R. Corte, A. Eliso, and A. Pecchia, "Rt-cases: Container-based virtualization for temporally separated mixed-criticality task sets," in *Proc. Leibniz Int. Proc. Inform.*, 2019, Art. no. 5.
[15] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the Linux kernel," *SIGBED Rev.*, vol. 16, no. 3, pp. 33–38, Nov. 2019.
[16] M. Thiyyakat, S. Kalambur, and D. Sitaram, "Improving resource isolation of critical tasks in a workload," in *Proc. 23rd Int. Workshop Job Scheduling Strategies Parallel Process.*, vol. 12326, 2020, pp. 45–67.
[17] F. Hofer, M. Sehr, A. Sangiovanni-Vincentelli, and B. Russo, "Industrial control via application containers: Maintaining determinism in IaaS," *Syst. Eng.*, vol. 24, no. 5, pp. 352–368, 2021.
[18] D. Bernstein, "Containers and cloud: From LXC to docker to Kubernetes," *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, Sep. 2014.
[19] F. Lumpp, F. Fummi, H. Patel, and N. Bombieri, "Containerization and orchestration of software for autonomous mobile robots: A case study of mixed-criticality tasks across edge-cloud computing platforms," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2022, pp. 1–6.
[20] "Linux scheduler." Accessed: Oct. 1, 2023. [Online]. Available: docs.kernel.org/scheduler/index.html
[21] "Architecture." Accessed: Oct. 1, 2023. [Online]. Available: kubernetes.io/docs/concepts/architecture/cloud-controller/
[22] F. Carpio, M. Delgado, and A. Jukan, "Engineering and experimentally benchmarking a container-based edge computing system," in *Proc. IEEE Int. Conf. Commun.*, 2020, pp. 1–6.

[23] V. Ibarra-Junquera, A. González, C. M. Paredes, D. Martínez-Castro, and R. A. Nuñez-Vizcaino, "Component-based microservices for flexible and scalable automation of industrial bioprocesses," *IEEE Access*, vol. 9, pp. 58192–58207, 2021.
[24] S. Aldegheri, N. Bombieri, S. Germiniani, F. Moschin, and G. Pravadelli, "A containerized ROS-compliant verification environment for robotic systems," in *Proc. Des., Automat. Test Europe Conf. Exhibit.*, 2021, pp. 222–225.
[25] G. Kurtzer, V. Sochat, and M. Bauer, "Singularity: Scientific containers for mobility of compute," *PLoS One*, vol. 12, 2017, Art. no. e0177459.
[26] A. Moga, T. Sivanthi, and C. Franke, "Os-level virtualization for industrial automation systems: Are we there yet?," in *Proc. ACM Symp. Appl. Comput.*, 2016, pp. 1838–1843.
[27] P. González-Nalda, I. Etxeberria-Agiriano, I. Calvo, and M. Otero, "A modular CPS architecture design based on ROS and docker," *Int. J. Interactive Des. Manuf.*, vol. 11, no. 4, pp. 949–955, 2017.
[28] K. Kaur, S. Garg, G. Kaddoum, S. H. Ahmed, and M. Atiquzzaman, "KEIDS: Kubernetes-based energy and interference driven scheduler for industrial IoT in edge-cloud ecosystem," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4228–4237, May 2020.
[29] H. Sami and A. Mourad, "Dynamic on-demand fog formation offering on-the-fly IoT service deployment," *IEEE Trans. Netw. Service Manag.*, vol. 17, no. 2, pp. 1026–1039, Jun. 2020.
[30] F. Lumpp, M. Panato, F. Fummi, and N. Bombieri, "A container-based design methodology for robotic applications on Kubernetes edge-cloud architectures," in *Proc. Forum Specification Des. Lang.*, 2021, pp. 01–08.
[31] H. A. Ozmen, S. Işık, and C. Ersoy, "A hardware and environment-agnostic smart home architecture with containerized on-the-fly service offloading," *Comput. Elect. Eng.*, vol. 92, 2021, Art. no. 107090.
[32] S. Aldegheri, N. Bombieri, F. Fummi, S. Girardi, R. Muradore, and N. Piccinelli, "Late breaking results: Enabling containerized computing and orchestration of ROS-based robotic sw applications on cloud-server-edge architectures," in *Proc. 57th ACM/IEEE Des. Automat. Conf.*, 2020, pp. 1–2.
[33] F. Mastrogiovanni, A. Paikan, and A. Sgorbissa, "Semantic-aware real-time scheduling in robotics," *IEEE Trans. Robot.*, vol. 29, no. 1, pp. 118–135, Feb. 2013.
[34] L. Han, L. Xu, D. Bobkov, E. Steinbach, and L. Fang, "Real-time global registration for globally consistent RGB-D SLAM," *IEEE Trans. Robot.*, vol. 35, no. 2, pp. 498–508, Apr. 2019.
[35] V. Struhár, M. Behnam, M. Ashjaei, and A. V. Papadopoulos, "Real-time containers: A survey," in *Proc. Workshop Fog Comput. IoT*, 2020, pp. 7:1–7:9.
[36] F. Hofer, M. A. Sehr, A. Iannopollo, I. Ugalde, A. Sangiovanni-Vincentelli, and B. Russo, "Industrial control via application containers: Migrating from bare-metal to IAAS," in *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci.*, 2019, pp. 62–69.
[37] S. Fiori, L. Abeni, and T. Cucinotta, "Rt-Kubernetes: Containerized real-time cloud computing," in *Proc. 37th ACM/SIGAPP Symp. Appl. Comput.*, 2022, pp. 36–39.
[38] F. Hofer, M. A. Sehr, A. Sangiovanni-Vincentelli, and B. Russo, "ODRE workshop: Probabilistic dynamic hard real-time scheduling in HPC," in *Proc. IEEE 23rd Int. Symp. Real-Time Distrib. Comput.*, 2020, pp. 207–212.
[39] M. Cinque, R. Della Corte, and R. Ruggiero, "Preventing timing failures in mixed-criticality clouds with dynamic real-time containers," in *Proc. 17th Eur. Dependable Comput. Conf.*, 2021, pp. 17–24.
[40] L. De Simone and G. Mazzeo, "Isolating real-time safety-critical embedded systems via SGX-based lightweight virtualization," in *Proc. Int. Symp. Softw. Rel. Eng. Workshops*, 2019, pp. 308–313.
[41] V. Struhár, S. S. Craciunas, M. Ashjaei, M. Behnam, and A. V. Papadopoulos, "React: Enabling real-time container orchestration," in *Proc. 26th IEEE Int. Conf. Emerg. Technol. Factory Automat.*, 2021, pp. 1–8.
[42] S. Choudhury, S. Maheshwari, I. Seskar, and D. Raychaudhuri, "Shareon: Shared resource dynamic container migration framework for real-time support in mobile edge clouds," *IEEE Access*, vol. 10, pp. 66045–66060, 2022.
[43] A. E. González and E. Arzuaga, "Herdmonitor: Monitoring live migrating containers in cloud environments," in *Proc. IEEE Int. Conf. Big Data*, 2020, pp. 2180–2189.
[44] S. Zheng, F. Huang, C. Li, and H. Wang, "A cloud resource prediction and migration method for container scheduling," in *Proc. IEEE Conf. Telecommun., Opt. Comput. Sci.*, 2021, pp. 76–80.
[45] Docker, "Configure networking." Accessed: Oct. 1, 2023. [Online]. Available: docs.docker.com/network
[46] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2015, pp. 171–172.
[47] "Deployments." Accessed: Oct. 1, 2023. [Online]. Available: kubernetes.io/docs/concepts/workloads/controllers/deployment
[48] L. Torvalds, "sched_deadline: Implement runtime overrun signal support." Accessed: Oct. 1, 2023. [Online]. Available: git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=34be39305a77b8b1ec9f279163c7cdb6cc719b91
[49] M. Felser, "Real-time ethernet - industry prospective," *Proc. IEEE*, vol. 93, no. 6, pp. 1118–1129, Jun. 2005.
[50] "RT-tests." Accessed: Oct. 1, 2023. [Online]. Available: git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git



Francesco Lumpp received the bachelor's and master's degrees in computer science and engineering from the University of Verona, Verona, Italy, where he is currently working toward the Ph.D. degree in computer science with the Department of Engineering for Innovation Medicine.

His research focuses on the development and optimization of software for the edge-cloud computing continuum and mixed-criticality systems.

Franco Fummi (Member, IEEE) received the Laurea degree in electronic engineering and the Ph.D. degree in electronic and communication engineering at the Polytechnic of Milan, Milan, Italy, in 1990 and 1994, respectively.

Since 2001, he has been a Full Professor in computer architecture with the University of Verona, Verona, Italy, where he is leading the Cyber-physical and IoT Systems Design (CISD) group, currently composed of more than 20 people, and working on hardware description languages and electronic design automation methodologies for modeling, verification, testing, and optimization of cyber-physical systems. He is also a Co-Founder of two spin-off companies: EDALab, focused on networked embedded systems design, and the automation control software company FACTORYAL.

Hiren D Patel (Member, IEEE) received the bachelor's and Ph.D. degrees in computer engineering from Blacksburg, VA, USA, in 2001 and 2017, respectively.

He is currently a Professor with Electrical and Computer Engineering Department, University of Waterloo, ON, Canada. His research interests include the design, analysis, and implementation of computer hardware and software systems, real-time embedded systems, computer architecture, hardware architectures for machine learning and artificial intelligence, and security.

Nicola Bombieri (Member, IEEE) received the Ph.D. degree in computer science from the University of Verona, Verona, Italy, in 2008.

He is currently a Professor with the Department of Engineering for Innovation Medicine, University of Verona. His research interests include parallel and heterogeneous architectures, artificial intelligence at the edge, and parallel programming languages. He develops embedded and efficient Software applications, with a particular emphasis on addressing multi- and extra-functional constraints, such as performance, power, and energy efficiency.