Enabling Containerisation of Distributed Applications with Real-Time Constraints

Nasim Samimi
Eindhoven University of Technology, The Netherlands

Luca Abeni
Scuola Superiore Sant'Anna, Pisa, Italy

Daniel Casini
Scuola Superiore Sant'Anna, Pisa, Italy

Mauro Marinoni
Scuola Superiore Sant'Anna, Pisa, Italy

Twan Basten
Eindhoven University of Technology, The Netherlands

Mitra Nasri
Eindhoven University of Technology, The Netherlands

Marc Geilen
Eindhoven University of Technology, The Netherlands

Alessandro Biondi
Scuola Superiore Sant'Anna, Pisa, Italy

Abstract

Containerisation is becoming a cornerstone of modern distributed systems, thanks to their lightweight virtualisation, high portability, and seamless integration with orchestration tools such as Kubernetes. The usage of containers has also gained traction in real-time cyber-physical systems, such as software-defined vehicles, which are characterised by strict timing requirements to ensure safety and performance. Nevertheless, ensuring real-time execution of co-located containers is challenging because of mutual interference due to the sharing of the same processing hardware. Existing parallel computing frameworks such as Ray and its Kubernetes-enabled variant, KubeRay, excel in distributed computation but lack support for scheduling policies that allow guaranteeing real-time timing constraints and CPU resource isolation between containers, such as the `SCHED_DEADLINE` policy of Linux. To fill this gap, this paper extends Ray to support real-time containers that leverage `SCHED_DEADLINE`. To this end, we propose KubeDeadline, a novel, modular Kubernetes extension to support `SCHED_DEADLINE`. We evaluate our approach through extensive experiments, using synthetic workloads and a case study based on the MobileNet and EfficientNet deep neural networks. Our evaluation shows that KubeDeadline ensures deadline compliance in all synthetic workloads, adds minimal deployment overhead (in the order of milliseconds), and achieves lower worst-case response times, up to 4 times lower, than vanilla Kubernetes under background interference.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering

Keywords and phrases Kubernetes, real-time containers, SCHED_DEADLINE, KubeRay

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2025.3

Supplementary Material *Software (Source Code)*: `https://gitlab.retis.santannapisa.it/dra-rt/dra-rt-driver.git`

Funding This work was partially supported by the EU ECSEL project TRANSACT grant agreement No. 101007260 and the EU Horizon Europe Framework Programme project NANCY grant agreement No. 101096456.

1 Introduction

Containers are increasingly used to deploy applications across distributed systems composed of microservices that need to work in a coordinated and scalable way [26, 56]. They provide an isolated environment that packages applications with all dependencies, enhancing portability and offering more lightweight virtualisation than traditional virtual machines [12].

Additionally, containers also seamlessly integrate with orchestration tools like Kubernetes [3], significantly simplifying the monitoring, management, and scaling of virtualised applications. Thanks to its advantages, several distributed computing frameworks such as Ray [48], Apache Spark [63], and Flyte [1] use Kubernetes to leverage its orchestration infrastructure.

As containerised applications become more prevalent across the edge-to-cloud continuum, they are increasingly integrated into real-time cyber-physical systems, where meeting strict timing requirements is essential to ensure safety [33, 54]. One example is the distributed inference of deep learning algorithms for image recognition or classification, which supports decision-making processes that actuate actions in the physical environment. Real-time performance over such AI-driven distributed infrastructures is especially sought in fields like automotive. For instance, autonomous vehicles often run deep-learning inference for object detection and lane recognition distributed across edge nodes or in-vehicle compute units. Several proposals for next-generation autonomous vehicles already underway [35, 49]. Meeting real-time constraints for containerised applications is becoming important also for Network Function Virtualisation (NFV) [10] and Software-Defined Networking (SDN) [39], which often employ highly-tuned Linux versions to minimise noise and latency [28].

Meeting real-time constraints in containerised distributed applications requires controlling the interferences and latencies at the network and processing levels. Network latencies can be managed by leveraging QoS control techniques such as DiffServ [21, 22] or by using edge-fog architectures to bring data processing and decision-making closer to the data origin. However, taming processor-level interference suffered by containerised applications is difficult since they share the same processing resources. Furthermore, containers co-located on the same hardware platforms often belong to different applications and tenants, and they do not trust each other. Therefore, the need arises for **(1)** partitioning the CPU processing resources, assigning a configurable CPU bandwidth share to each container with controllable CPU latency, and **(2)** enforcing that a potentially faulty or malicious application cannot harm the timing guarantees of another application.

Among the various frameworks for container-enabled distributed processing, the Ray framework [48] attracted particular interest: for example, it is used by industrial players such as OpenAI (ChatGPT), Netflix, ByteDance (TikTok), and Uber [9, 58, 59]. Ray is a popular open-source framework for distributed computations, supporting machine learning, data processing, simulation, and optimisation [48]. Ray is increasingly often applied to latency-sensitive workloads such as deep-learning inference and robotics, where predictable response times are often required, even if the applications are not strictly hard real-time [48]. With its native Kubernetes integration, KubeRay simplifies managing Ray clusters through Kubernetes orchestration. KubeRay is generally deployed in standard cloud servers, where its instances are often affected by interferences from other co-located applications on the same servers or physical CPUs (the so-called "noisy neighbour" problem in cloud computing). As a result, applications executing in Ray usually suffer from large runtime variations in their execution times, often preventing them from meeting the tight timing constraints of some classes of real-time virtualised applications (such as video processing and AI inference in cyber-physical systems).

The Constant Bandwidth Server (CBS) reservation algorithm [16] is an effective means to achieve timing performance and enforce temporal isolation of CPU resources. A CBS reserves a portion of the available bandwidth for each application, protecting it from interferences by other applications with hard-to-predict timing behaviour running on the same physical platform. CBS provides fine-grained control over container runtime and period, unlike the coarse-grained scheduling of `SCHED_OTHER`. CBS allows containers to be analysed in isolation, enabling modular scheduling, a key concept in compositional frameworks like CSF [53] and MPR [29]. This simplifies analysing system behaviour and supports incremental deployment.

Also, CBS avoids the need for over-provisioning CPU resources to meet latency requirements, allowing tighter control of system utilisation. In contrast to user-space throttling mechanisms previously used to approximate real-time behaviour, CBS delegates resource enforcement to the kernel, reducing complexity in container orchestration environments.

CBS is based on the Earliest Deadline First (EDF) scheduling algorithm and has been available in mainline Linux since version 3.14 through the `SCHED_DEADLINE` policy [41]. The mainline version of `SCHED_DEADLINE` works for general Linux threads, as well as virtualised workloads like QEMU/KVM virtual machines that create a Linux thread for each virtual CPU; hence, the host operating system (OS) can schedule the VM's virtual CPUs by scheduling the corresponding threads. An out-of-tree patch is instead required [13] to schedule containers with `SCHED_DEADLINE`. The resource reservation and timing isolation features of the CBS and `SCHED_DEADLINE` are suitable to satisfy the corresponding needs of co-located containers; nevertheless, frameworks such as Ray lack support for `SCHED_DEADLINE`.

The problem of employing `SCHED_DEADLINE` to schedule Ray workloads involves enabling Kubernetes (using KubeRay) to manage containers with `SCHED_DEADLINE` and integrating this real-time containerisation feature of Kubernetes with KubeRay. The real-time containerisation was proven to be practical by a state-of-the-art prototype [31] (called RT-Kubernetes). Nevertheless, this prototype introduced deep and invasive changes across multiple core components of the Kubernetes control plane (including the scheduler and kubelet), spanning many files. These modifications significantly complicate maintainability across different Kubernetes versions and require custom builds and non-standard deployment procedures, making the system difficult to maintain, debug, or update over time. Consequently, it only works with a single old version of Kubernetes (in conjunction with an old version of the Linux kernel) and does not address the integration with Ray.

This paper. We make the following three contributions:

- We design and implement a new Kubernetes extension, called KubeDeadline, which extends Kubernetes to support `SCHED_DEADLINE` reservations while ensuring its compatibility with future versions of Kubernetes. To support a modular, portable, and maintainable real-time containerisation feature, we employ the Dynamic Resource Allocation (DRA) [6] framework provided by recent versions of Kubernetes. Our goal is to create a sustainable, easy-to-install real-time containerisation feature across Kubernetes versions. We design a real-time resource driver for Kubernetes, referred to as the Real-Time Dynamic Resource Allocator (RT-DRA), that **(i)** performs admission control on the existing nodes in the Kubernetes cluster to select a suitable node with sufficient resources and that **(ii)** employs the `SCHED_DEADLINE` scheduler to deploy containers.
- We integrate KubeDeadline with Ray through KubeRay, allowing the execution of Ray-distributed applications leveraging `SCHED_DEADLINE` reservations. This contribution is achieved by adapting the interface and core functionality of KubeRay to employ our KubeDeadline resource driver RT-DRA. Our modifications are general and, therefore, allow for the integration of any Kubernetes resource driver with KubeRay.
- We present an experimental evaluation using synthetic task sets and a case study based on distributed machine learning inference to show the effectiveness of the proposed solution.

2 Background

Providing timing guarantees and CPU isolation of distributed applications using the Ray framework requires modifications across multiple levels, from the Linux kernel scheduler to the Ray framework itself, as well as the Kubernetes-based containerisation solution. The corresponding needed background is introduced in the following.

2.1 The SCHED_DEADLINE scheduler of Linux

The Linux kernel implements five different schedulers, which are queried in order, thus implicitly implementing a first level of priorities among them. The `stop_machine` scheduler is queried first, which is used for kernel facilities only, and it is not available to the user. Secondly, the kernel queries the `SCHED_DEADLINE` scheduler, which provides resource reservation based on the CBS and EDF algorithms (extensively discussed in the following). Then, the fixed-priority schedulers follow, `SCHED_FIFO` and `SCHED_RR`, which only differ by the fact that equal priority tasks are executed to completion in `SCHED_FIFO` while they are assigned a time quantum in `SCHED_RR` (similarly to weighted round robin). The last one is the general purpose scheduler, implemented by the `SCHED_OTHER` scheduling class.

Most important to this work is `SCHED_DEADLINE`. It is a scheduler that implements the CBS resource reservation algorithm [11, 16] and is based on EDF scheduling, providing the assigned workload with the required CPU bandwidth and the worst-case CPU latency. `SCHED_DEADLINE` reservations enable timing isolation through a budgeting mechanism that relies on two key parameters that need to be configured for each reservation r_i: the reservation budget Q_i (also called runtime) and the reservation period P_i. Essentially, `SCHED_DEADLINE` implements a *resource partitioning* mechanism by supplying Q_i time units to the workload running inside the reservation during each period of P_i time units. `SCHED_DEADLINE` functions as both a resource partitioning and a *resource enforcement* mechanism, ensuring that each workload receives no more than its guaranteed fraction of CPU capacity. This approach prevents a faulty or untrustworthy application within a reservation from impacting the timing behaviour of another application due to an overrun caused by a fault or a cyber-attack.

The theoretical properties of `SCHED_DEADLINE` ensure that, if the reservation servers are schedulable, i.e., if each reservation can actually receive Q_i time units every period P_i, the workload running inside each reservation r_i is provided with a fraction of the overall CPU bandwidth α_i and a bounded worst-case CPU delay Δ_i, which are a function of the budget Q_i and the period P_i. For example, when the reservation is scheduled by only a single core (as under partitioned multiprocessor scheduling), $\alpha_i = Q_i/P_i$ and $\Delta_i = 2 \cdot (P_i - Q_i)$ [53].

The workload running within the reservation can be general Linux processes, including QEMU/KVM virtual processors. A kernel patch to support hierarchical constant bandwidth servers [13] is instead required to manage containers. The patch implements a two-level scheduling hierarchy (named H-CBS [43] from now on). Hierarchical scheduling enables compositional analysis, the ability to analyse each container's schedulability in isolation. This allows us to leverage existing analyses from real-time literature, avoiding the need to develop new analytical models from scratch. At the root of the hierarchy, the CBS as implemented by `SCHED_DEADLINE` schedules groups of tasks (named control groups, or `cgroups`, in the Linux jargon, recalled in the following). Then, a second-level scheduler based on fixed priorities (`SCHED_FIFO` or `SCHED_RR`) selects the highest-priority task in the `cgroup`. In more detail, the root scheduler creates m_i identical CPU reservations (scheduled by `SCHED_DEADLINE` on m_i different CPU cores) all with runtime Q_i and reservation period P_i; this multi-core reservation is often indicated as (Q_i, P_i, m_i). Note that there is no need to change the applications to be able to use `SCHED_DEADLINE`. Hence, the source code of the applications is not required [13, 31] (see [29] for more detail on how to configure these parameters). The reserved CPU time is shared among all the tasks included in the `cgroup`, and since containers are generally based on `cgroups`, this mechanism provides timing guarantees to containers. In other words, H-CBS allows assigning multiple computational activities to the same CPU reservation, while the vanilla `SCHED_DEADLINE` maps only a single task to each reservation.

2.2 The SCHED_OTHER scheduler of Linux

The `SCHED_OTHER` [2] policy is the default scheduling policy used in Linux for general-purpose, non-real-time tasks, that is implemented through a fair-share scheduler (CFS or EEVDF [55], depending on the kernel version; note that CFS is a variant of the Start Fair Queuing (SFQ) [34] algorithm, designed to approximate fair queuing behaviour for general-purpose operating systems). `SCHED_OTHER` is designed to fairly divide CPU time among all runnable tasks by tracking how much processor time each has used and distributing time slices accordingly. The scheduler uses a red-black tree to track runnable processes, ensuring that the process with the least virtual runtime is always selected next. This design aims to give every task a fair share of CPU access over time, regardless of when it becomes runnable.

`SCHED_OTHER` does not offer guarantees about when or how quickly a task will execute. Instead, it ensures relative fairness, i.e., tasks are treated equally unless their nice value (a user-defined priority hint ranging from -20 to +19) suggests otherwise. A lower nice value increases a task's likelihood of being scheduled sooner and more frequently, while a higher nice value means a lower priority for a task.

The scheduler dynamically adjusts task priorities based on their CPU usage and nice levels. While `SCHED_OTHER` tasks can preempt one another, they are always preempted by higher-priority real-time tasks, such as those scheduled using `SCHED_DEADLINE` (see Section 2.1). By design, `SCHED_OTHER` tasks are scheduled only with the leftover CPU capacity, after all real-time tasks have been served [8]. This makes `SCHED_OTHER` unsuitable for real-time workloads, especially under high system load or interference.

2.3 Control group files (cgroup)

Control Groups (cgroups) is a Linux kernel feature that enables resource management by allocating, prioritising, and restricting system resources such as CPU, memory, I/O bandwidth, and network access among different processes or groups of processes.

Cgroups organise processes into hierarchical groups, applying resource constraints through control group files located in the cgroup filesystem. For real-time scheduling, cgroups expose several configuration files, including:

- **`cpu.rt_period_us`** – Defines the real-time scheduling period (in microseconds). For example, a value of `1000000` in the file, means the scheduling period is 1 second.
- **`cpu.rt_runtime_us`** – Specifies the maximum CPU time allocated for real-time tasks within a period. For example, the file contents can be a value of `500000`, which means the task can run for 0.5 seconds per scheduling period. This file is used to apply a single runtime allocation, meaning that it is valid for all CPUs.
- **`cpuset.cpus`** – Defines the specific CPU cores assigned to a group. For example, the file contents can be `0-3,5`, which means the process can run on CPU cores 0, 1, 2, 3, and 5.

In the new version of the real-time Linux kernel used in our system, another cgroup file related to real-time scheduling is introduced, aligning well with our resource allocation needs:

- **`cpu.rt_multi_runtime_us`** – Unlike `cpu.rt_runtime_us`, `cpu.rt_multi_runtime_us` specifies individual runtime values for each CPU. Different from `cpu.rt_runtime_us`, it allows to selectively allocate budgets on a subset of the physical cores.
  The file format for specifying `cpu.rt_multi_runtime_us` follows a space-separated list, where each value corresponds to the real-time runtime for a specific CPU core in the system. For example, on a 4-core system, if CPUs 0 and 2 are assigned 200 microseconds of runtime, while CPUs 1 and 3 have no real-time allocation, the file contents are: `200 0 200 0`. We use this cgroup file for allocating runtime to applications.

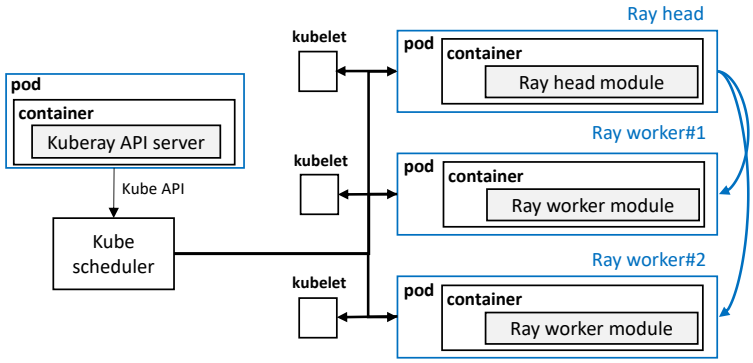


Figure 1 Ray architecture: The Ray head and Ray workers.

2.4 Kubernetes containerisation software stack

While Kubernetes leverages specific kernel functionalities and mechanisms to implement containers, other components of the software stack run in user space. Kubernetes [3] executes containerised applications over a cluster of nodes divided between control and worker nodes. The former kind host the Kubernetes control plane, while the latter host containers in charge of effectively running the applications. The *pod* represents the basic execution unit and is formed by one or more containers, network connections, and storage. The Kubernetes architecture includes multiple microservices allocated on different cluster nodes.

Kube-scheduler [4] is the default scheduling component of Kubernetes and plays a central role in determining where newly created pods are deployed within a cluster. It operates on the control plane and continuously monitors the Kubernetes API server for pods that do not yet have a node assignment. Once it detects such a pod, it initiates a scheduling cycle that involves two major phases: filtering and scoring.

During the filtering phase, the scheduler evaluates each available node in the cluster to determine whether it meets the pod requirements. These include checks for sufficient CPU, memory, or other resource availability (e.g., a pod requesting 2 CPUs and 4GiB of memory will not be scheduled on a node with only 1 CPU or 2GiB free); compatibility with node and pod affinity/anti-affinity rules (e.g., a pod may require to be co-located with other pods running a specific microservice, or explicitly separated from them); toleration of taints applied to nodes (e.g., a node tainted with `dedicated=gpu:NoSchedule` only accepts pods with a matching toleration, ensuring it's used exclusively for GPU workloads); and compliance with topology spread constraints (e.g., ensuring that pods of the same application are evenly distributed across availability zones to improve fault tolerance). Nodes that do not meet all required criteria are excluded from consideration.

In the scoring phase, the scheduler ranks the remaining eligible nodes by applying scoring functions. These are implemented as scoring plugins, each of which assigns a numeric score to every eligible node. The final decision is based on combined, weighted scores from multiple plugins. The node with the highest score is selected, and the scheduler communicates the decision to the API server by updating the pod specification to bind it to the chosen node.

The scheduler is highly extensible. Kubernetes allows users to define custom scheduling policies or even write custom schedulers. This extensibility makes it possible to support specialised workloads and deployment strategies that are not addressed by default heuristics.

Kubelet is the primary node-level agent and runs on every node in the Kubernetes cluster. Once a pod has been scheduled to a node, the kubelet is responsible for ensuring that the pod is actually created and maintained as specified. It retrieves the pod definition from the API server and communicates with the container runtime, typically via the Container Runtime Interface (CRI), to start and manage the container(s).

The kubelet performs several critical tasks beyond container creation. It monitors the health of running containers, restarting them if necessary according to the specified restart policy. It reports the node and pod status back to the API server, which is essential for observability and control decisions made by the control plane. Additionally, it enforces resource constraints, such as CPU and memory limits, using the CRI. Upon startup, the kubelet registers its node with the Kubernetes API server, advertising its available resources and any attached labels. This metadata is then used by the scheduler for pod placement.

Overall, the kube-scheduler and kubelet form a tightly integrated loop within the Kubernetes control and execution planes. The kube-scheduler makes placement decisions based on a global view of the cluster state, while the kubelet ensures local enforcement of those decisions and continuously manages the operational lifecycle of containers.

Container runtimes. As mentioned, the kubelet manages the lifecycle of pods on its worker node by interacting with a high-level CRI, such as `containerd` or `cri-o`, to manage container lifecycles. These, in turn, use a low-level container runtime, referred to as Open Container Initiative (OCI), like `runc` or `crun`, and monitor the health status of containers. `containerd` (considered in this work) is an industry-standard runtime that manages the lifecycle of containers. It is responsible for pulling container images, managing container storage, networking, and supervision. It acts as an abstraction layer between the higher-level container orchestration tools, such as Kubernetes, and the lower-level runtime components, such as `runc`. The latter is a lightweight, low-level container runtime that creates and runs containers according to the OCI specifications. It is responsible for starting containers and configuring their resource limits, namespaces, and other low-level settings. `runc` operates under the control of high-level container runtimes to run containers on a host.

Default resource allocation of Kubernetes. By default, Kubernetes schedules containers using the Linux `SCHED_OTHER` scheduler. CPU usage is controlled via Linux cgroups using two key parameters: `cpu.cfs_quota_us` and `cpu.cfs_period_us`. These define the maximum amount of CPU time a container is allowed to consume within a given period and are used to enforce upper bounds on CPU usage. However, this quota-based control does not provide any temporal guarantees or strict isolation.

To better manage resource allocation and prioritisation, Kubernetes classifies pods into three Quality of Service (QoS) classes: *BestEffort*, *Burstable*, and *Guaranteed* [7]. These classes are determined by how CPU and memory requests and limits are specified in the pod resource configuration. *BestEffort* pods declare no resource requests or limits and are scheduled only when spare capacity is available, offering no performance guarantee. *Burstable* pods specify both resource requests and limits, with the limit higher than the request. Kubernetes guarantees the requested amount, but the pod may use additional resources up to its limit when resources are available. *Guaranteed* pods define equal values for requests and limits, receive the highest scheduling priority, and may be assigned dedicated CPU cores. However. this setup does not provide formal real-time guarantees at the kernel level.

Memory is managed in a similar fashion, with upper bounds enforced via cgroups. However, specifying memory requests and limits does not inherently prevent memory overcommitment unless explicitly configured through eviction policies or node-level settings. For GPUs, Kubernetes relies on device plugins to advertise GPU resources and uses node labelling and resource requests to place pods on GPU-enabled nodes.

To support more advanced resource types and customisable scheduling policies, Kubernetes introduced the Device Resource Assignment (DRA) framework. DRA enables external plugins to influence scheduling and resource provisioning decisions, allowing the system to support heterogeneous resources like GPUs or FPGAs. We discuss DRA in more detail in Section 3.2.

Users interact with Kubernetes through control nodes using the `kubectl` command, which performs common cluster operations. Components like pods are typically defined in YAML manifests and deployed with `kubectl`.

2.5 Ray and KubeRay

Ray [48] is an open source framework for building and scaling distributed applications, which is also popularly used for distributed training and inference of AI algorithms (used by OpenAI, Netflix, ByteDance, and Uber [9, 58, 59], among others). It consists of two fundamental classes of nodes: the *head node*, which is responsible for the coordination, scheduling, and management of a cluster of nodes, and a set of *Ray worker nodes*, which are in charge of performing the actual execution and sending the results back to the head node. We consider the KubeRay variant of Ray – an open-source project that integrates Ray with Kubernetes and simplifies the management of Ray clusters using the Kubernetes orchestration features. In this context, the challenge of leveraging `SCHED_DEADLINE` to schedule Ray workloads involves making KubeRay compatible with Kubernetes, which we address in Section 4. The interaction between Ray and Kubernetes when using KubeRay is graphically shown in Figure 1. In the context of Kubernetes, the Ray head node and Ray worker nodes are, in fact, pods. We use the term node to refer to a node in a Kubernetes cluster, and Ray head and Ray worker to refer to the head node and worker nodes in a Ray cluster, respectively. The Ray Head manages and distributes the workload among the Ray workers. To create the Ray head and workers, a YAML manifest with the required configuration for the Ray head and workers must be submitted through `kubectl`. The API server of KubeRay translates this to pod configurations and sends it to the kube-scheduler, the Kubernetes component responsible for selecting the appropriate Kubernetes cluster node to deploy pods. Kube-scheduler creates pod scheduling contexts using the configuration of the Ray head and workers to deploy pods.

To leverage the capabilities of `SCHED_DEADLINE`, various components as well as configuration parameters, need to be modified, as discussed in the following section.

3 KubeDeadline: system architecture and real-time container deployment

Extending Kubernetes in a way that makes it compatible with `SCHED_DEADLINE` reservations involves addressing three key technical challenges:

C1. It must be possible to specify reservation capabilities (namely, `SCHED_DEADLINE`'s parameters budget and period, and a number of cores) related to each container.

C2. The kubelet, which runs on each worker node and is in charge of activating the containers, must be modified to interact with the Linux kernel API of `SCHED_DEADLINE` and control groups to set the budget and period of the container.

C3. The kube-scheduler must be extended to make the container-to-node placement aware of the `SCHED_DEADLINE` admission control logic (based on CBS's theoretical analysis).

3.1 Design overview

Design Alternatives. Each of these challenges can be addressed in different ways.

The easiest way to address **C1** is by requiring a non-backward compatible modification of the manifest file of the pod by adding new fields to specify the period, budget, and number of cores assigned to a pod. The manifest is used by the Kubernetes API server with the `kubectl` command line tool, which communicates with the Kubernetes scheduler to choose a worker node for the container and pass the content of the manifest. While the benefit of this solution is simplicity, it makes the approach not-retro-compatible because Kubernetes must be modified to parse additional fields of the (YAML) manifest file. To avoid retro-compatibility issues, it is possible to use annotations in the pod manifest file, which still would not provide any indication to Kubernetes on how to use the specified parameters, addressing **C1** only.

To address **C2** and **C3**, the easiest solution is to patch Kubernetes by directly modifying its codebase. However, this method has the drawback of binding the Kubernetes real-time extension to a single Kubernetes version, making forward compatibility bound to keeping the patches up to date. This approach was followed in a prototype developed in [31], for which no updated patches have been provided, making the prototype compatible only with old versions of Kubernetes and the Linux kernel [31]. Conversely, we are interested in a modular solution that makes the extension portable to future versions of Kubernetes. To address **C3**, it is possible to implement a scheduler plugin [5]. Nevertheless, this option would still not allow for a uniform solution to address all three technical challenges.

A holistic, DRA-based, solution. The three challenges can all be addressed by using Kubernetes' *Dynamic Resource Allocation* (DRA), a mechanism for requesting and sharing resources between pods and containers inside a pod. DRA generalises the persistent volumes API for generic resources and has been used, for example, to enable more flexible control over GPU sharing across Kubernetes pods [6].

In the first step, our goal is to create a resource driver using Kubernetes DRA that deploys containers compatible with the `SCHED_DEADLINE` scheduler in Linux by defining a budget, period, and number of cores of the reservation servers directly through the Kubernetes API server. We call this resource driver *RT-DRA*. The driver must first select a node within the Kubernetes cluster that has cores available and sufficient bandwidth to accommodate the container (challenge **C3**). Once a suitable node is identified, the required cores are assigned to the container using a placement algorithm, such as best-fit or worst-fit decreasing algorithms, which previous work found to work particularly well in this context [14]. Ultimately, we expect the scheduling requirements to be applied by creating `cgroup` files for the container on the selected node, specifying the reservation parameters accordingly.

Hierarchical scheduling and system model. Making Kubernetes compatible with `SCHED_DEADLINE` gives rise to a hierarchical scheduling framework, shown in Figure 2. A first scheduling layer involves containers, encapsulated in `SCHED_DEADLINE` reservations, which are scheduled by the `SCHED_DEADLINE` scheduler of Linux leveraging the CBS and EDF algorithms. On the other hand, `SCHED_DEADLINE`-enabled containers implement a *virtual processor* abstraction for the workloads running within each container, associated with a reservation r_i. Each reservation r_i is characterised by a budget, period, and assigned number of cores – the (Q_i, P_i, m_i) triplet, which translates to a guaranteed CPU bandwidth α_i and a maximum CPU delay Δ_i [53]. Each container implements a second level of scheduling, in

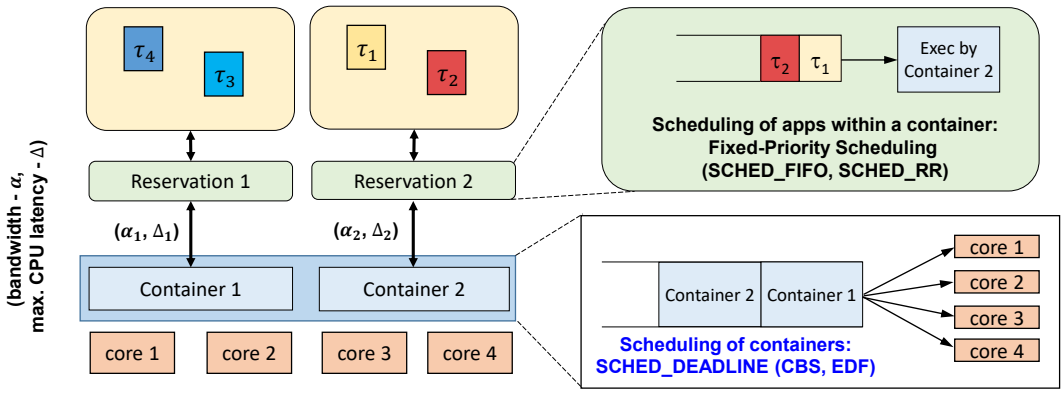


Figure 2 Overview of the hierarchical scheduling framework obtained with KubeDeadline on each node.

which each task τ_j is executed by leveraging a fixed-priority scheduling mechanisms (leveraging the `SCHED_FIFO` and `SCHED_RR` scheduling classes of Linux). The proposed framework does not constrain the activation pattern (e.g., periodic, sporadic) and deadline (arbitrary) of each task τ_i, but requires assigning each task a priority according to fixed-priority scheduling.

Schedulability in the hierarchical scheduling system. A set $\mathcal{R}_i$ of CBS reservations allocated to core c_i is guaranteed to be schedulable if it verifies the EDF utilisation test [11, 44]:

$$\sum_{r_i \in \mathcal{R}_i} \frac{Q_i}{P_i} \leq 1. \tag{1}$$

In practical Linux systems, this admission test is slightly modified to ensure the CPU core reserves a 5% capacity for housekeeping activities (i.e., the right-hand-side bound is set to 0.95). In this context, the schedulability of reservations means that they can correctly deliver Q_i time units of budget every P_i, guaranteeing the corresponding bandwidth α_i and CPU latency Δ_i. Schedulability of workloads running in the reservations must also be guaranteed. Given the H-CBS hierarchical scheduling framework considered in this work, fixed-priority schedulability analysis under limited resource supply can be used. State-of-the-art methods are available to select suitable values of parameters (Q_i, P_i, m_i) that guarantee schedulability [20, 29, 42] based on the timing parameters of the workloads running inside the reservations (execution times, periods etc.). Methods to estimate them are available in literature in case they are not known [15, 27], as it can happen for dynamic workloads.

KubeDeadline approach. To begin, we introduce basic concepts related to DRA. Then, we describe the development of RT-DRA and how it prepares resources using `SCHED_DEADLINE`. In the last step, we explain how container runtimes are adapted to deploy the real-time containers. The integration of RT-DRA and the adapted container runtimes within Kubernetes form our extended version of Kubernetes, which we refer to as *KubeDeadline*. The realisation of the approach is detailed next. Fig. 3 shows an overview of the extension.

3.2 Kubernetes DRA

The DRA framework was introduced to overcome limitations in how Kubernetes handles specialised hardware or non-standard resources. DRA provides a dynamic, extensible mechanism that enables Kubernetes to negotiate and allocate custom resources like FPGAs

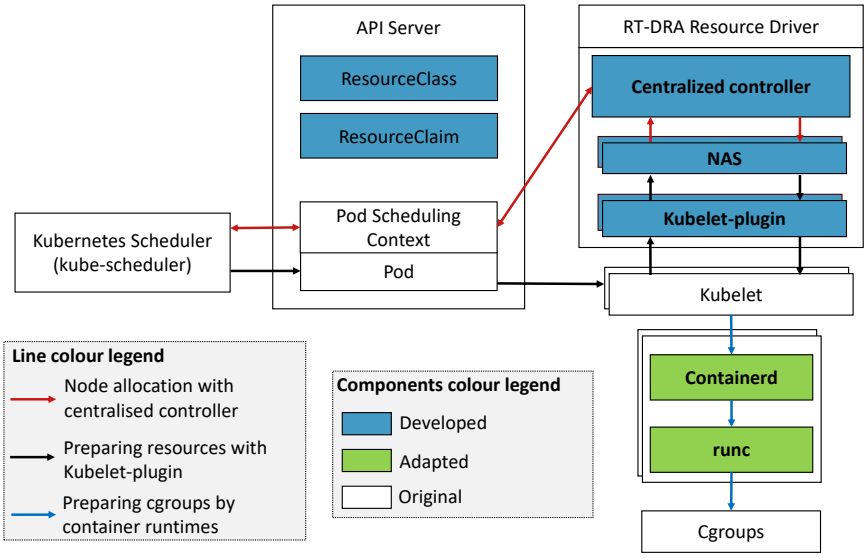


Figure 3 KubeDeadline: resource preparation flow, from pod specification to cgroups.

or GPUs during pod deployment. It is particularly useful for workloads that need fine-grained control over resource parameters that Kubernetes cannot natively express. That is, the Kubernetes DRA allows creating custom resource definitions (CRD) not originally supported by Kubernetes. To create a new resource type, we must define **two key concepts**:

- **ResourceClass** – Defines the custom resources that a container can request. It specifies how the parameters for a resource should be configured.
- **ResourceClaim** – Represents a request for a resource by a container. It refers to a specific ResourceClass and requests an instance of that resource for the container.

Kubernetes interacts with the resource driver through its two main components: the **centralised controller** and the **kubelet-plugin**. These components have specific predefined interfaces to maintain adaptability and compatibility with Kubernetes. These functions are designed to be general, meaning that their actual implementation heavily depends on the specific type of resource being integrated into Kubernetes.

The **centralised controller** selects the most suitable node within the Kubernetes cluster that can meet the container resource requirements and must allocate resources to *claims*.

The **kubelet-plugin** prepares the allocated resources by producing and forwarding the allocation information in the form of a Container Device Interface (CDI) – a standard mechanism that allows specifying device-specific parameters and configurations for containers – to the kubelet of the corresponding node. This component is specific to each node.

The centralised controller and kubelet-plugin communicate with each other through **Node Allocation State (NAS)** that maintains the information of the node, such as resources allocated to the claims. This component is specific to each node.

When a user submits a pod that includes a ResourceClaim, the kube-scheduler contacts the DRA centralised controller, as specified in the ResourceClaim, to find a suitable node. The controller checks each node's NAS to see if the resource is available. Once a node is selected, the controller updates the NAS to record the allocation and allow the kubelet-plugin to access the information. The kubelet-plugin on the selected node reads the allocation state from the NAS and generates a CDI file, which is also recorded in the NAS. The kubelet then reads the CDI and passes it (currently in the form of pod annotations) to the container runtime so that the container can be created and access the requested resource.

3.3 Implementing ResourceClass and ResourceClaim for RT-DRA

We defined a ResourceClass for RT-DRA called `rt.kubedeadline.io`, which specifies the parameters for real-time CPU reservation and is registered with API group `rt.resource.kubedeadline.io` following DNS-style naming conventions in Kubernetes. A real-time ResourceClaim is a request for a specific RT-DRA resource and contains three attributes: `count`, `runtime`, and `period`, which denote the number of cores m_i, the budget Q_i, and the period P_i, respectively.

The configurations in Figures 4a, 4b, and 4c illustrate how a real-time container is deployed using RT-DRA. To request a real-time resource, designers must define a `resourceClaim`, as shown in Figure 4a. In Figure 4b, the `ResourceClaimParameters` define the scheduling parameters, including `count`, `runtime`, and `period`. Each `ResourceClaimParameters` must have a unique name (line 11, Figure 4a), which allows the corresponding `ResourceClaim` to reference it (line 4, Figure 4b). Here, the name is `rtclaimparams`. In this example, two cores are requested, each with a `runtime` of 100 and a `period` of 1000 time units.

The `ResourceClaim` must have a name and specify its associated `ResourceClass` by referencing `resourceClassName`, which should be set to `rt.kubedeadline.io`, as previously mentioned. Additionally, it defines `parametersRef`, which includes a reference (`rtclaimparams`) to the claim parameters – containing the requested scheduling parameters defined before – along with `apiGroup`, which is set to `rt.resource.kubedeadline.io`.

As shown in Figure 4c, to deploy the real-time container, it must be encapsulated within a pod, as it is the smallest deployable unit in Kubernetes and containers cannot run outside the scope of a pod. Additionally, all resource claims must be referenced in the scope of the pod and then used by the containers. Here, the pod references the required `ResourceClaim` by name (`rtclaim`) and sets a name for this reference (`rtcpu`, line 9). The container inside the pod (`ctr0`) then associates itself with the claim by referencing the name `rtcpu`. Each container can have at most one claim, as shown in line 17.

Pods, resource claims, and parameters are defined in YAML format, stored together or separately, and deployed with the `kubectl` command.

3.4 Node Allocation State (NAS) of RT-DRA

The Node Allocation State (NAS) is in charge of maintaining the information state of each node and allowing the centralised controller and the kubelet-plugin to communicate. It stores the following data:

- `Allocated claims`: Maintains a mapping of CPU core IDs to resource claims. Each claim has a unique name and ID. The centralised controller writes this mapping to the NAS after an allocation, enabling the kubelet-plugin of the corresponding node to read it and generate the necessary CDI devices, which describes the necessary device-specify configurations of containers.
- `Allocated utilisation`: Tracks the current utilisation of each CPU core across all nodes. This value is updated by the centralised controller after every allocation or deallocation to reflect the real-time CPU usage across nodes. It stores essential data that is used to ensure a feasible allocation allowing to satisfy real-time constraints [41].
- `Prepared claims`: Stores the claim IDs that have been prepared by the kubelet-plugin and are ready for allocation, or removes the claim IDs that are unprepared.

```yaml
apiVersion: resource.k8s.io/v1alpha2
kind: ResourceClaim
metadata:
  name: rtclaim  # Name for the resource claim template
spec:
  spec:
    resourceClassName: rt.kubedeadline.io
    parametersRef:
      apiGroup: rt.resource.kubedeadline.io
      kind: RtClaimParameters
      name: rtclaimparams  # reference to resource claim parameters
```

(a) Defining resource claims.

```yaml
apiVersion: rt.resource.kubedeadline.io/v1alpha1
kind: RtClaimParameters
metadata:
  name: rtclaimparams  # Name of the claim parameters
spec:
  count: 2  # Number of CPUs
  runtime: 100
  period: 1000
```

(b) Resource claim parameters.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod0
  labels:
    app: pod
spec:
  resourceClaims:
    - name: rtcpu  # Name of the resource claim
      source:
        resourceClaimName: rtclaim  # Reference to resource claim
  containers:
    - name: ctr0
      image: example:latest
      resources:
        claims:
          - name: rtcpu  # Resource claim reference for this container
```

(c) Pod configuration.

Figure 4 Overview of deploying a pod with a container configured to use a resource claim.

3.5 Preparing resources with RT-DRA

Next, we describe the behaviour of RT-DRA during the deployment of a pod requesting real-time resources (e.g., the pod in Figure 4b), as described in Algorithm 1 and as shown in Figure 3. Each step of the algorithm describes the functionality of a main component in RT-DRA (centralised controller and kubelet-plugin). Components of RT-DRA communicate with different components in Kubernetes. Hence, the described procedure must comply with Kubernetes DRA specifications to ensure compatibility with Kubernetes without requiring any modifications to the core functionality of Kubernetes itself.

Input and output. The input consists of the scheduling context for a pod P with a resource claim C. This claim includes parameters such as the budget, period, and number of cores that can be allocated to the pod. These parameters serve as constraints that guide the

allocation process. The desired output is the CDI devices generated by the kubelet-plugin. For example, for parameters in Figure 4b, the CDI devices may be represented as a string of `rtcpu-runtime=100-period=1000-CPUSET=0,1`, where 0 and 1 are IDs of the allocated cores. However, if the requested resources by the pod are not available, i.e., there is no node with sufficient resources, no CDI devices are returned.

Algorithm 1 Preparing resources with RT-DRA using `SCHED_DEADLINE`.

Input : Scheduling context for pod P with resource claim C=(`runtime`, `period`, `count`), a vector CBW of consumed bandwidths of all cores, whose elements are denoted by $CBW[ID]$ where ID is the core identifier
Output : CDI devices

1 **Step 0: Input validation with centralised controller**
2 **Step 1: Node allocation with centralised controller**
3 Receive the set N of available nodes from the Kube scheduler.
4 **foreach** node $n \in N$ **do**
5 $BW \leftarrow CBW$
6 $bw \leftarrow$ `runtime`/`period`
7 selected_core_IDs $\leftarrow find\_by\_admission\_test(n, bw,$ `count`$)$
8 **if** selected_core_IDs *is empty* **then**
9 **continue**
10 **foreach** ID *in* selected_core_IDs **do**
11 $BW[ID] \leftarrow BW[ID] + bw$
12 Notify the Kube scheduler of the selected node n
13 $CBW \leftarrow BW$
14 Record allocation details in the Node Allocation State (NAS)
15 **Exit loop**
16 **Step 2: Preparing resources using kubelet-plugin**
17 Read allocation state for claim C from the NAS.
18 Generate CDI devices.
19 Forward CDI devices to the kubelet of the corresponding node.

Step 0: Input validation with centralised controller . First, we ensure that the requested resource class matches the name `rt.kubedeadline.io`, the requested parameter names are `count`, `runtime`, and `period`, and their values are valid for real-time resources. Specifying extra fields for parameters or missing one field generates an error (Line 1).

Step 1: Node allocation with centralised controller. In this step, the Kubernetes scheduler provides a list N of available nodes to the centralised controller of the RT-DRA at Line 4 (to avoid the interference of real-time containers on non-real-time containers, it is possible to schedule real-time containers on dedicated nodes using Kubernetes taints and tolerations). RT-DRA iterates on each node $n \in N$ to determine whether the node can accommodate all claims for the pod P. For each node, a search based on the admission test of Equation (1) is performed to find a set of cores to accommodate all requested reservations. Partitioning strategies such as worst-fit or best-fit are used. The partitioning strategy is specified at the RT-DRA startup and, once specified, it cannot be changed. These operations are implemented by the *find_by_admission_test* function in Algorithm 1.

Before initiating the admission test on a node, the consumed bandwidths of the cores are copied from the NAS. This ensures that the NAS is updated for deterministic allocations. The admission test searches for cores that can accommodate all reservation servers within a claim and returns a list of IDs from the selected cores. If the admission test fails, the list will be empty. At Line 8, if the list is empty, the loop will be skipped to check the next node.

If the search is successful, the copy of the cores' bandwidths BW is updated to include the new values of the allocated reservations, i.e., bw (Line 10). The centralised controller notifies the Kubernetes scheduler (Line 12) and records the allocation details in the NAS in Line 14. This step concludes by logically assigning the pod to the selected node, but no actual resource allocation takes place at this point. If none of the nodes meets the condition for scheduling a pod, the pod will not be scheduled, and the pod will stay in a pending state until the request to deploy the pod is deleted by Kubernetes or a node becomes available.

Step 2: Preparing resources using kubelet-plugin. The kubelet-plugin of the selected node receives the resource claim C from the Kubernetes scheduler, and the Kubelet-plugin on the designated node retrieves the allocation state, that is, IDs of the allocated cores, from the NAS (Line 17). The kubelet-plugin then generates CDI devices based on the claim parameters and IDs of the allocated cores (Line 18).

3.6 Deployment of real-time containers with container runtimes

After the CDI devices are prepared by the RT-DRA and sent to the kubelet, the kubelet forwards the CDI devices to `containerd` without making any changes. Until Kubernetes receives the CDI devices, the pod stays in the `pending` state.

`containerd` supports Container Device Interface (CDI) devices; however, it lacks the capability to decode CDI devices generated by RT-DRA. To address this, we modified `containerd` to properly parse these CDI devices and extract the real-time scheduling parameters, runtime, period, and number of cores. `containerd` already defines internal variables for the real-time scheduling parameters. Since these variables are natively supported, no modifications were necessary to their definitions. Instead, our modification enables `containerd` to extract the corresponding values from the CDI device metadata and assign them to these pre-existing variables.

We made a minimal modification to the `containerd` codebase, adding a total of 19 lines of code in a single file. These changes introduce a helper function that extracts the number of cores, runtime, and period values from the CDI device information embedded in the pod annotations by the kubelet. If `containerd` detects real-time scheduling parameters in the CDI, it invokes the helper function and applies the extracted values for real-time scheduling. Otherwise, it treats the container as a standard non-real-time workload and schedules it with `SCHED_OTHER` (the default, general purpose scheduler of Linux). By default, `containerd` transmits these parameters to `runc` during container deployment.

`runc` natively defines and accepts the real-time scheduling parameters, runtime, period, and number of cores from `containerd`. Additionally, `runc` supports the creation of `cpu.rt_period_us`, a cgroup file that specifies the real-time scheduling period.

However, `runc` does not currently support the creation of `cpu.rt_multi_runtime_us`, a feature available in real-time kernels that allows for more flexible real-time execution across multiple CPUs. To address this, we introduced a custom function in `runc` that uses the number of cores and runtime to generate and configure the `cpu.rt_multi_runtime_us`

file. This enhancement[1] in `runc` allows creating `cpu.rt_multi_runtime_us` instead of `cpu.rt_runtime_us` when real-time scheduling parameters are present. Note that `cpu.rt_runtime_us` is not currently used by Kubernetes.

We modified approximately 116 lines of code across two files in `runc`. In one file, we introduced two small helper functions to read from and write to the `cpu.rt_runtime_us` values in the parent cgroups when a container is being created. This step is necessary to enable nested real-time (RT) cgroups, as the Linux kernel requires that a child cgroup runtime does not exceed runtime of its parent.

These helpers were integrated into the existing function that configures real-time cgroup parameters for the container (e.g., `cpu.rt_period_us` and `cpu.rt_runtime_us`). We updated this logic to additionally first invoke the functions that update the runtime of parent directories and then handle the `cpu.rt_multi_runtime_us` for the container. In a second file, we added a function to handle destroying the container, which is called from within the `destroy` function. This cleanup step identifies the container runtime and recursively removes it from all parent cgroups to avoid accumulated values in the runtime of parent directories.

After `runc` could successfully create the `cgroup` files, the resource allocation process is finalised, the pod P is created and its state changes to `running` (see Figure 3 for the connection among the different components of KubeDeadline).

Maintenance of `containerd` and `runc`. The non-invasive and lightweight modifications in containerd and runc poses no maintainability issues [2]. No external libraries were introduced; all additions rely solely on standard Go libraries or existing internal libraries within each tool. The added code paths are only triggered when real-time annotations are explicitly present, ensuring that default container creation remains unaffected. Due to the minimal footprint, the changes are safe to remove or upstream if needed. Importantly, these modifications do not introduce any new dependencies or require additional build steps. Both `containerd` and `runc` continue to build using their standard pipelines, without the need for custom toolchains or specific build scripts. All modified components are made open source to support transparency, reproducibility, and development.

4 Using KubeDeadline in KubeRay

As shown in Figure 1 and discussed in Section 2.5, KubeRay consists of two key classes of pods: the Ray head, which is in charge of coordinating and distributing the workload to a cluster of Ray workers, and a set of Ray workers, in charge of executing the workload and sending back the results. By default, KubeRay provides a customised *API* for specifying resource and configuration values for the Ray head and Ray workers. The KubeRay *API server* is responsible for translating resource specifications (such as CPU and memory) provided via the KubeRay API into Kubernetes pod specifications, ensuring that the Ray head and worker nodes are properly scheduled within the cluster.

Although Kubernetes natively supports resource claims for custom-defined resources (i.e., resources not included by default, such as FPGA or specialised hardware accelerators), the KubeRay API currently accepts standard resource types only, including CPU (default Linux

[1] Modifications in `runc` extend beyond KubeDeadline, enabling support for general Linux cgroups (`cpu.rt_multi_runtime_us`); we plan to submit them for future `runc` versions.

[2] We forward-ported our changes to the latest versions of containerd and runc, in approximately 10 minutes.

scheduling policies), memory, and GPU. As a result, when attempting to schedule Ray head and Ray worker pods with custom resource claims defined in DRAs, KubeRay does not recognise these claims and cannot propagate them to the Kubernetes DRA.

We enabled KubeRay to deploy Ray head and worker pods using the `SCHED_DEADLINE` scheduler. This required modifying KubeRay to recognise and manage resource claims, ensuring they are correctly propagated to Kubernetes similar to standard resource requests.

```yaml
headGroupSpec:
  rayStartParams: {}
  template:    #Pod template
    spec:
      resourceClaims:
        - name: rt-head
          source:
            resourceClaimName:
                rtclaim
      containers:
        - name: ray-head
          resources:
            claims:
              - name: rt-head
```

(a) Head Group Specification.

```yaml
workerGroupSpecs:
    rayStartParams: {}
    template:    #Pod template
      spec:
        resourceClaims:
          - name: rt-worker
            source:
              resourceClaimName:
                  rtclaim
        containers:
          - name: ray-worker
            resources:
              claims:
                - name: rt-worker
```

(b) Worker Group Specification.

Figure 5 Ray configuration.

Hence, we modified the KubeRay API (YAML interfaces) by introducing `Claim` in the list of resource definitions. `Claim` consists of a `name` and a `resource`, allowing KubeRay to recognise and process resource claims. As a result, when a claim is specified for the Ray head and worker in YAML configurations, the KubeRay API can correctly interpret and accept it. API server is also modified to receive the claims from the API and propagate them to Kubernetes. Since Kubernetes already natively supports resource claims, no additional redefinition of these claims is required at the Kubernetes level. Finally, the claims will be processed by the Kubernetes DRA. These modifications are not specific to RT-DRA and allow KubeRay to be used with other types of resources defined within custom DRAs.

Deploying Ray head and worker with real-time resources. Figures 5a and 5b show the configurations for deploying Ray components, i.e., Ray head and Ray worker. Claiming reservation servers for Ray components is similar to the normal pod deployment shown in Figure 4c. As shown, Ray head is referencing the `resourceClaim` named `rt-head` and Ray worker is referencing a `resourceClaim` named `rt-worker`. Both `rt-head` and `rt-worker` are referencing a `resourceClaim` named `rtclaim` which is defined as in Figure 4a and Figure 4b.

5 Experimental evaluation

We conducted extensive empirical experiments to validate our KubeDeadline and evaluate its effectiveness. We performed three sets of experiments. First, we provide a proof of concept to validate KubeDeadline and the RT-DRA by showing that they can deploy containers using the `SCHED_DEADLINE` scheduler according to the given runtime, period, and number of cores. Second, we measure the overhead introduced by the RT-DRA. Third, we evaluate the modified KubeRay in a case study focusing on two deep-learning models for image classification: MobileNet [36] and EfficientNet [57]. We measure the response times of the image classification models under various configurations with real-time workloads in KubeDeadline containers.

Experimental setup. The experiments were conducted on a cluster composed of two identical servers equipped with an Intel Xeon E3-1240 V2 processor. This processor has 4 physical cores (no hyper-threading), each with a base clock speed of 3.40 GHz and a maximum frequency of 3.8 GHz. The system supports 32-bit and 64-bit modes. The server is configured with 32 GB of DDR3 RAM, running at 1600 MHz. The server runs a real-time Linux kernel version 5.15.105 equipped with the H-CBS patch [13].

5.1 KubeDeadline Evaluation

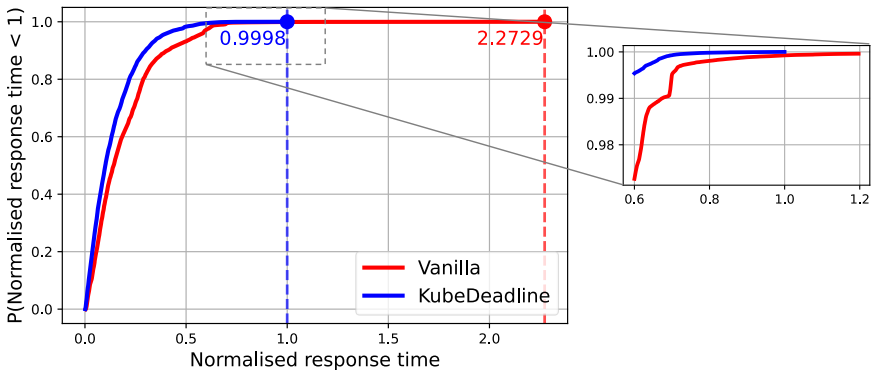


Figure 6 Experimental CDF of the response times of all the tasks of 150 task sets with utilisation ranging from $U = 0.6$ to $U = 2.0$ and a number of tasks ranging from $N = 4$ to $N = 12$.

In the first set of experiments, we evaluate the effectiveness of our approach for sets of synthetic periodic real-time tasks with implicit deadlines. We generated 150 task sets with utilisation ranging from $U = 0.6$ to 2.0 and a number of tasks ranging from 4 to 12 using the RandFixedSum algorithm [30]. We assigned the scheduling parameters (budget, period, and number of assigned cores) based on the Multiprocessor Periodic Resource (MPR) real-time analysis [29] using the CARTS tool [50].

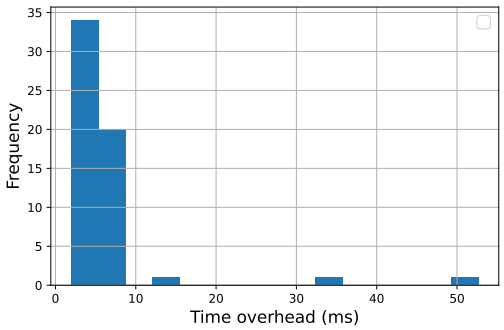
The MPR theory guarantees that these scheduling parameters allow respecting deadlines of all containerised tasks. Therefore, by showing that, under KubeDeadline, every task experiences response times always smaller than its period (implicit deadlines), we empirically verify the correctness of our solution. To summarise the response times of all tasks in a single figure, the normalised response time (response time divided by the task period) has been used: if the normalised response time is smaller than 1, then the deadline requirement is met; otherwise, it is violated.

Figure 6 displays the experimental Cumulative Distribution Function (CDF) derived by all the normalised response times of all the tasks of all the task sets (more than 275000 samples). The KubeDeadline line represents the CDF of the normalised response times experienced with KubeDeadline when properly-designed reservations are used, while the "Vanilla" line represents the CDF of the normalised response times experienced with a standard Kubernetes. The figure shows that the KubeDeadline CDF reaches 1 for a value of the normalised response time smaller than 1 (0.9998), indicating that no deadline has been missed, while the Vanilla Kubernetes CDF reaches 1 for a value of the normalised response time larger than 1 (about 2.27), indicating that a standard Kubernetes (using the standard `SCHED_OTHER` Linux scheduler) is not able to respect all the tasks' deadlines.

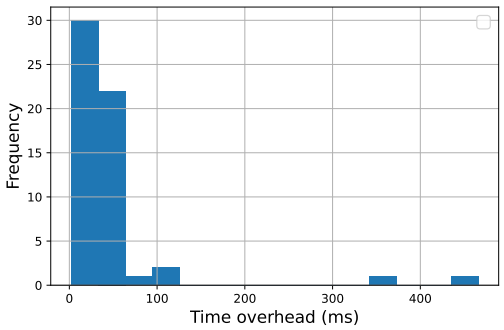
5.2 RT-DRA Overhead

As mentioned earlier, RT-DRA includes the two main components centralised controller and kubelet-plugin. Hence, we measured the timing overhead of each component in deploying a pod. We repeated this measurement over 20 pods, where each container claims one to four cores. The minimum, maximum, and average time overhead of centralised controller are 2 ms, 466 ms, and 48.5 ms, respectively, and for kubelet-plugin are 1.96 ms, 52.66 ms, and 6.2 ms. Histograms of the time overhead of the centralised controller and kubelet-plugin are shown in Figure 7a and Figure 7b, respectively. Note that overhead occurs only during container deployment, not during application execution. Thus, it is deemed acceptable.

The kubelet-plugin performs a constant-time operation involving CDI data generation, resulting in $\mathcal{O}(1)$ complexity. In contrast, the centralised controller evaluates resource allocation across all cluster nodes and performs admission tests on each node, yielding a complexity of $\mathcal{O}(n \cdot c)$, where n is the number of nodes and c is the number of cores per node.



(a) Time overhead of the centralised controller component of RT-DRA.



(b) Time overhead of the kubelet-plugin component of RT-DRA.

Figure 7 Time overhead of the centralised controller and kubelet-plugin components of RT-DRA.

5.3 KubeRay with real-time containers

In this experiment, we focus on comparing, in the context of a case study, the performance of KubeRay extended to support `SCHED_DEADLINE` with the vanilla KubeRay, which schedules threads with the default `SCHED_OTHER` scheduling class of Linux, designed for general-purpose applications. The purpose of this experiment is to demonstrate the advantage of real-time containers over non-real-time containers under background workload, highlighting the added value of real-time support in Kubernetes and KubeRay.

Evaluation metric. We aim to evaluate the impact of real-time scheduling on application response times, demonstrating how it enhances deadline guarantees and enables support for tighter deadlines. Instead of defining explicit deadlines, we assess the system capability by measuring the achievable frame rate, i.e., frames-per-second (FPS). In video inference workloads, each inference corresponds to one video frame and should complete within a soft deadline to ensure smooth streaming/processing. Since FPS = 1/deadline, higher FPS implies the system is able to meet shorter per-frame deadlines, which in turn implies improved responsiveness and service quality. **Minimum FPS** reflects the worst-case response time

observed during execution; a higher minimum FPS corresponds to a lower and more predictable worst-case response time. **Average FPS** represents the overall system performance under typical conditions, such as interference from background workload.

As a case study, we consider a machine-learning-based image classification application. We used a MobileNet model [36] and an EfficientNet model [57] pre-trained on the ImageNet dataset for image classification. We processed a dataset of 5000 individual images, sequentially running inference on images to emulate frame-by-frame processing in a video stream. This approach allowed us to measure the processing time per image per different CPU allocations.

To manage and execute this task, we used Ray jobs (a Ray job creates a Ray cluster by deploying a head node and worker nodes and runs the application, e.g., running MobileNet or EfficientNet inference on the ImageNet dataset), which enables creating and managing a Ray cluster and submitting individual tasks to the Ray cluster. We allocate a specific amount of CPU and memory to the Ray head and workers to observe how different resource configurations affect the response times of the inference tasks for each image.

In all experiments, whether using KubeDeadline or Vanilla Kubernetes, we introduce a CPU-intensive background workload, such as an additional Ray job, to create interference. This background workload is carefully configured to ensure that the main Ray job under evaluation receives its full requested CPU and memory resources. The background workload is scheduled using `SCHED_DEADLINE`. The background workload is distributed across CPU cores in a way that ensures each Ray component, both the head and the workers, receives its full requested CPU allocation from a single core, without requiring execution across multiple cores. This setup preserves core-locality and avoids migration overhead. For instance, in a system with two CPU cores, if the Ray head requests 80% of a core and the Ray worker requests 20%, we configure the background load such that one core remains fully available (100%) for the Ray head, while the other core is only 60% utilised by background tasks, leaving 40% free for the Ray worker.

In the first experiment, we processed images sequentially with MobileNet, running inference on one image at a time. This setup placed the entire workload on the Ray head, allowing us to observe how varying the CPU allocation of the head affects response times. In the second experiment, we processed the images in batches of 50 for both MobileNet and EfficientNet, distributing the workload primarily across the Ray workers. This configuration enabled us to examine how varying the CPU allocation of the workers impacts overall performance while keeping the Ray head configuration constant.

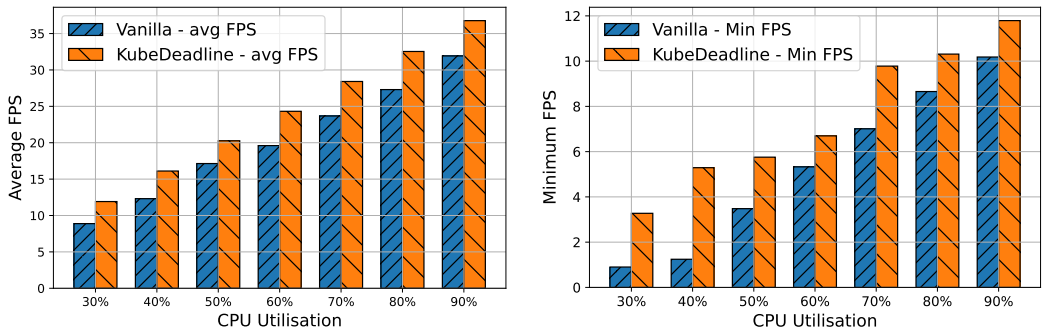
For both experiments, we consider a Ray cluster with one Ray head with 3GB of memory and four Ray workers, each with 4GB of memory. The chosen parameters for these experiments serve as illustrative examples to demonstrate system behaviour and are not derived from a specific use case. For applications with strict or arbitrary deadlines, the values of Q and P can be determined using state-of-the-art methods (e.g., [29]).

Both sets of experiments are done in two parts: (i) the Ray cluster is deployed using KubeDeadline and (ii) the Ray cluster is deployed using vanilla Kubernetes.

Experiment 1. The Ray head's CPU budget was varied from 30000 μs to 90000 μs in steps of 10000 μs, while keeping the period fixed at 100000 μs. The Ray workers' configurations remained constant, each with a 50000 μs budget and a 100000 μs period. We assigned a priority of 90 to all the threads in the Ray head and worker nodes.

In the vanilla Kubernetes setup, the Ray head CPU quota was varied from 300 to 900 $mCPU$[3] in increments of 100 $mCPU$, while the Ray workers remained at 500 $mCPU$ each.

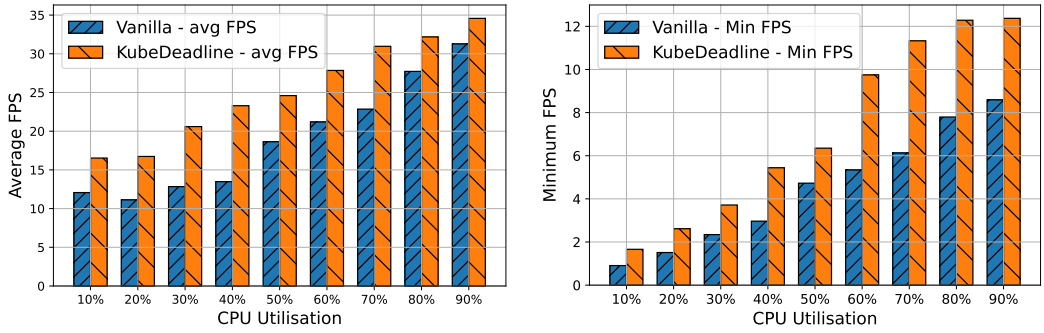
[3] In Kubernetes, $mCPU$ (millicpu) is a unit used to represent CPU resources. One $mCPU$ for a period



(a) Average achievable FPS. **(b)** Minimum achievable FPS.

Figure 8 Comparison of vanilla Kubernetes and KubeDeadline, KubeRay head, MobileNet.

We measured the average achievable frame-per-second (FPS) for both vanilla Kubernetes and KubeDeadline across different bandwidths allocated to the Ray head. As shown in Figure 8a, by decreasing the allocated bandwidth, the average FPS decreases for both vanilla Kubernetes and KubeDeadline. KubeDeadline provides an average FPS between 10 and 38, whereas vanilla Kubernetes provides an average FPS between 8 and 33. Note that KubeDeadline provides a higher FPS than vanilla Kubernetes in all the experiments. For example, when the Ray head has a reservation server with 80% utilisation, it provides 33 FPS, while vanilla Kubernetes can provide 33 FPS at 90% utilisation (900 $mCPU$). That is, KubeDeadline achieves higher throughput under background workloads by providing temporal isolation and enforcing per-container CPU budgets, ensuring consistent performance despite system interference.

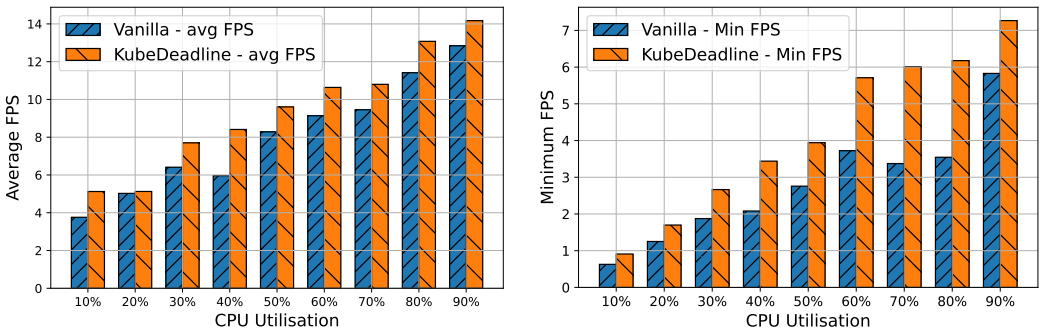


(a) Average achievable FPS. **(b)** Minimum achievable FPS.

Figure 9 Comparison of vanilla Kubernetes and KubeDeadline, KubeRay workers, MobileNet.

Figure 8b depicts the minimum achievable FPS for both vanilla Kubernetes and KubeDeadline according to the worst observed inference response time across different bandwidths allocated to Ray head. Also in this case, KubeDeadline provides a higher FPS than vanilla Kubernetes in all the experiments. KubeDeadline provides a minimum FPS between (almost) 2 and 13, where vanilla Kubernetes provides a minimum FPS between 1 and 9. For example, when Ray head has a reservation server with 70% utilisation, it provides an FPS of 10,

of 100 ms is equivalent to a quota of 50 ms.



(a) Average achievable FPS.

(b) Minimum achievable FPS.

Figure 10 Comparison of vanilla Kubernetes and KubeDeadline, Kuberay workers, EfficientNet.

while vanilla Kubernetes can provide 10 FPS at 80% utilisation (800 $mCPU$). Also, the FPS of vanilla Kubernetes suddenly decreases from 40% utilisation (less than 1 FPS), while KubeDeadline still provides 5 FPS. That is, KubeDeadline has a lower worst-case response time and can guarantee tighter deadlines, despite the interference from background workload. Furthermore, the worst observed FPS in KubeDeadline increases smoothly by increasing the bandwidth showing a robust and predictable behaviour, whereas vanilla Kubernetes shows abnormal behaviour illustrating its unpredictability and vulnerability to background noise.

Experiment 2. The Ray head configuration was kept constant with a budget of 90000 μs and a period of 100000 μs. The Ray workers budgets were varied from 10000 μs to 90000 μs in increments of 10000 μs, while maintaining the period at 100000 μs. For the vanilla Kubernetes setup, the Ray workers' CPU quotas were varied from 100 $mCPU$ to 900 $mCPU$ in increments of 100 $mCPU$, while the Ray head remained fixed at 900 $mCPU$.

As shown in Figure 9a, decreasing the allocated bandwidth for MobileNet, decreases the average FPS of both vanilla Kubernetes and KubeDeadline. KubeDeadline provides an average FPS between 17 and 35, where vanilla Kubernetes provides an average FPS between 12 and 32. Note that KubeDeadline provides higher FPS than vanilla Kubernetes in all experiments. Figure 9b depicts the minimum achievable FPS for MobileNet for both vanilla Kubernetes and KubeDeadline according to the worst observed inference response time across different bandwidths allocated to Ray head. KubeDeadline provides a higher FPS than vanilla Kubernetes in all the experiments. KubeDeadline provides a minimum FPS between almost 2 and 12, where vanilla Kubernetes provides a minimum FPS between 1 and 9. These results reinforce the earlier findings in Experiment 1, showing that KubeDeadline consistently achieves higher throughput and better worst-case response times across different workloads by enforcing temporal isolation and fine-grained CPU management.

Similarly in EfficientNet, as shown in Figure 10a, KubeDeadline provides an average FPS between 5 and 14, where vanilla Kubernetes provides an average FPS between 4 and 13. Figure 10b shows that KubeDeadline provides a higher FPS than vanilla Kubernetes in all experiments. KubeDeadline provides a minimum FPS between almost 1 and 7, where vanilla Kubernetes provides a minimum FPS between 0.5 and 6. Additionally, KubeDeadline maintains a minimum FPS of at least 4 from 50% CPU utilisation onward, whereas vanilla Kubernetes does not exceed a minimum FPS of 4 until CPU utilisation is 90%. Moreover, KubeDeadline achieves a minimum FPS of 6 at 80% utilisation, while vanilla Kubernetes never reaches 6 FPS at any utilisation level. Similar trends to MobileNet are observed

with EfficientNet, further validating the robustness of KubeDeadline in providing a lower worst-case response time and maintaining predictable inference performance across different models, despite the interference from background workload.

6 Related Work

Containerisation solutions have been widely adopted for their advantages in terms of efficient usage of resources, scalability, and simplified management. These benefits can be exploited to accelerate development, improve application reliability, and achieve greater operational efficiency. However, the use of such solutions in environments characterised by stringent timing constraints is curbed by the need for isolation among co-located containers, which can generate interference and jeopardise execution predictability. Several solutions have been proposed to mitigate this issue by enforcing isolation among containers running on the same platform. These solutions have been initially proposed for virtual machines (VMs) exploiting and extending features provided by hypervisors. For example, Checconi et al. [23] applied real-time scheduling techniques within the hypervisor to provide mechanisms for allocating computational resources while controlling interferences of co-located services. Lee et al. [40] proposed a technique to support real-time processing in the Xen bare-metal (Type-1) hypervisor by applying hierarchical real-time scheduling techniques assigning a precise share of the CPU time to each VM. The proposed mechanism has been incorporated within OpenStack [62] by enforcing timing isolation by introducing RT-based mechanisms. Cucinotta et al. [24] proposed, the integration of real-time scheduling of virtual machines deployed with KVM-hosted (Type-2) hypervisor within a complete cloud management solution for guaranteed resource allocation to multimedia and service-oriented real-time applications. The need for more time-predictable solutions as an enabling technology for Industry 4.0 has been highlighted in [19]. These solutions rely on machine virtualisation for flexible resource management and exploit real-time scheduling techniques within the hypervisor to enforce temporal isolation among co-located VMs. However, solutions based on machine virtualisation are affected by significant processing and memory overheads [17, 25] due to replicated features at the hypervisor and VM levels. Consequently, containerisation is often preferred to machine virtualisation due to a simpler architecture and lower overheads. Hence, solutions based on real-time scheduling have been proposed for containerisation frameworks. In this context, Kadusale et al. [37] explored WebAssembly (WASM) as an alternative to Docker containers for real-time serverless edge computing, analysing cold startup time and execution performance.

In the context of supporting latency-sensitive networking applications such as the deployment of network function virtualisation (NFV) services, a solution has been proposed [26] to exploit `SCHED_DEADLINE` to improve the isolation among co-located containers handled by OpenStack, allowing to define timing-related parameters in the MANO descriptors (structured templates that define the configuration, resource requirements, and lifecycle management of NFV). A similar solution for Kubernetes has been proposed [31]. Still, it was based on heavy modifications of the source code, making it only a proof of concept that already lacks the most recent features of Kubernetes. Furthermore, Ray is not supported in [31]. Differently, this work uses DRA to make Kubernetes compatible with `SCHED_DEADLINE`, providing a modular, portable, and maintainable solution.

While using the same approach at the kernel level based on `SCHED_DEADLINE`, Struhar et al. [56] focused on a hierarchical solution for the orchestration to increase flexibility while handling online variations of the workload. However, their work does not integrate the approach with Kubernetes and Ray.

Several studies have focused on monitoring and management frameworks for distributed and edge computing environments. Gala et al. [32] proposed a low-overhead monitoring framework for mixed-criticality applications on multicore platforms, ensuring real-time guarantees when sharing CPU, memory, and network resources, also introducing an edge-cloud hybrid architecture with a custom orchestration layer for Docker-based environments.

Similarly, Reis et al. [52] introduced LEM, a large-scale workflow control tool for managing and monitoring distributed applications across edge nodes. LEM focuses on workflow management rather than fine-grained hierarchical scheduling. Additionally, Wu et al. [61] addressed resource management in edge-cloud orchestrated vehicular networks, proposing a two-stage network slicing framework that uses reinforcement learning for resource allocation. Walser et al. [60] propose a fog and edge orchestration framework for real-time containers to improve scalability and deadline handling.

While these works provide valuable contributions in monitoring, workflow management, and resource allocation for distributed and edge environments, they do not fully address the requirements of real-time containerised applications with hierarchical scheduling. Specifically, they lack integration with Kubernetes native scheduling mechanisms and do not adopt real-time scheduling policies like `SCHED_DEADLINE`.

Monaco et al. [47] extended Kubernetes with shared resource orchestration to enforce memory bandwidth and last-level cache allocation, reducing performance interference among co-located real-time containers, but without focusing on timing isolation at the level of CPU processing time. Additionally, Barletta et al. [18] present k4.0s, a Kubernetes-based orchestration system for Industry 4.0, enabling criticality-aware monitoring and scheduling of real-time, mixed-criticality workloads. Moreover, Lumpp et al. extend Kubernetes to support mixed-criticality container orchestration for autonomous mobile robots across edge-cloud platforms [45], and later introduced RT-Kube which adds real-time capabilities to Kubernetes via custom resource definitions for timing constraints, a criticality-aware scheduler, and runtime monitoring with deadline-driven task migration [46]. However, none of these supports deploying real-time containers using `SCHED_DEADLINE`.

Ray is becoming widely used to deploy distributed parallel applications and is becoming typical for AI-based ones. However, the level of timing predictability is currently not satisfactory [51]. To exploit the advantages of Kubernetes in seamlessly parallelising distributed applications with Ray, Kanso et al. [38] proposed KubeRay, a suite of tools to create Ray clusters in Kubernetes with minimum effort. However, based on the standard implementation of Kubernetes, it is affected by the same issues in timing predictability when the application running in the deployed cluster presents timing constraints and other workloads can be assigned to the worker nodes.

To the best of our knowledge, this work proposes the first solution to make Ray compatible with `SCHED_DEADLINE` reservations, thus allowing fine-grained resource partitioning and offering timing isolation to distributed workloads developed in containers.

7 Conclusion

This paper addresses the problem of providing real-time support to containerised applications. To this end, we proposed KubeDeadline, which extends the Kubernetes framework to schedule Linux containers according to the `SCHED_DEADLINE` policy, which allows guaranteeing a portion of the overall CPU bandwidth with a bounded CPU latency, as well as providing timing isolation through a CPU resource enforcement mechanism. This has been implemented using the RT-DRA, which allows extending Kubernetes in a modular and maintainable way

without needing direct code patches that would make the extension hard to maintain for future Kubernetes versions. Then, we extended the Ray framework for distributed processing to make it compatible with the RT-DRA and hence with `SCHED_DEADLINE`. Extensive experimental results based on both synthetic workloads and a deep learning case study show that KubeDeadline allows providing timing guarantees for distributed applications using Ray and Kubernetes. Future work will consider extensions to jointly address CPU and GPU resource partitioning, e.g., by using `SCHED_DEADLINE` jointly with new scheduling techniques for NVIDIA GPUs. Furthermore, new algorithms will be designed to optimise the node selection of KubeDeadline. Additionally, we plan to enhance our DRA to inspect per-container reservations via the proc and cgroup filesystems, enabling it to detect existing non-real-time allocations and avoid conflicts at the CPU level.

References

1 Build production-grade data and ml workflows, hassle-free with flyte. Accessed: 2025-04-24. URL: `https://flyte.org/`.

2 CFS scheduler — the Linux kernel documentation. Accessed: 2025-04-28. URL: `https://docs.kernel.org/scheduler/sched-design-CFS.html`.

3 Kubernetes documentation. `https://kubernetes.io/`. Accessed: 2024-11-14.

4 Kubernetes scheduler | Kubernetes. Accessed: 2025-04-28. URL: `https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/`.

5 Kubernetes scheduling framework. `https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework`. Accessed: 2024-11-14.

6 Nvidia k8s-dra-driver. `https://github.com/NVIDIA/k8s-dra-driver`. Accessed: 2024-11-14.

7 Pod quality of service classes | Kubernetes. Accessed: 2025-04-28. URL: `https://kubernetes.io/docs/concepts/workloads/pods/pod-qos/`.

8 Real-time group scheduling — the Linux kernel documentation. Accessed: 2025-04-28. URL: `https://docs.kernel.org/scheduler/sched-rt-group.html`.

9 Scale machine learning & AI computing | Ray by anyscale. Accessed: 2025-04-29. URL: `https://www.ray.io/`.

10 Network Functions Virtualisation – Introductory White Paper – An Introduction, Benefits, Enablers, Challenges & Call for Action, 2012. SDN and OpenFlow World Congress. URL: `https://portal.etsi.org/NFV/NFV_White_Paper.pdf`.

11 L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS 1998)*, Madrid, Spain, december 2-4 1998. `doi:10.1109/REAL.1998.739726`.

12 Luca Abeni. Virtualized real-time workloads in containers and virtual machines. *Journal of Systems Architecture*, 154:103238, 2024. `doi:10.1016/J.SYSARC.2024.103238`.

13 Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the Linux kernel. *ACM SIGBED Review*, 16:33–38, November 2019. `doi:10.1145/3373400.3373405`.

14 Luca Abeni, Alessandro Biondi, and Enrico Bini. Partitioning real-time workloads on multi-core virtual machines. *Journal of Systems Architecture*, 131:102733, 2022. `doi:10.1016/j.sysarc.2022.102733`.

15 Luca Abeni, Tommaso Cucinotta, and Daniel Casini. Period estimation for Linux-based edge computing virtualization with strong temporal isolation. In *2024 IEEE 3rd Real-Time and Intelligent Edge Computing Workshop (RAGE)*, pages 1–6, 2024. `doi:10.1109/RAGE62451.2024.00013`.

16 Luca Abeni, Giuseppe Lipari, and Juri Lelli. Constant bandwidth server revisited. *ACM SIGBED Review*, 1291, January 2015. `doi:10.1145/2724942.2724945`.

17 Jason Anderson, Hongxin Hu, Udit Agarwal, Craig Lowery, Hongda Li, and Amy Apon. Performance considerations of network functions virtualization using containers. In *2016 International Conference on Computing, Networking and Communications (ICNC)*, pages 1–7, 2016. doi:10.1109/ICCNC.2016.7440668.

18 Marco Barletta, Marcello Cinque, Luigi De Simone, and Raffaele Della Corte. Criticality-aware monitoring and orchestration for containerized industry 4.0 environments. *ACM Transactions on Embedded Computing Systems*, 23(1):1–28, 2024. doi:10.1145/3604567.

19 Marco Barletta, Marcello Cinque, Luigi De Simone, Raffaele Della Corte, Giorgio Farina, and Daniele Ottaviano. Partitioned containers: Towards safe clouds for industrial applications. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, pages 84–88, 2023. doi:10.1109/DSN-S58398.2023.00029.

20 Enrico Bini, Marco Bertogna, and Sanjoy Baruah. Virtual multiprocessor platforms: Specification and use. *Proceedings - Real-Time Systems Symposium*, pages 437–446, 2009. doi:10.1109/RTSS.2009.35.

21 Steven Blake, David Black, Mark Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services. Technical report, The Internet Society, 1998. doi:10.17487/RFC2475.

22 Brian E Carpenter and Kathleen Nichols. Differentiated services in the internet. *Proceedings of the IEEE*, 90(9):1479–1494, 2002. doi:10.1109/JPROC.2002.802000.

23 Fabio Checconi, Tommaso Cucinotta, Dario Faggioli, Giuseppe Lipari, et al. Hierarchical multiprocessor CPU reservations for the Linux kernel. In *Proceedings of the 5th international workshop on operating systems platforms for embedded real-time applications (OSPERT 2009), Dublin, Ireland*, pages 15–22, 2009. URL: http://www.artist-embedded.org/docs/Events/2009/OSPERT/Proceedings-PreWorkshop.pdf#page=15.

24 T. Cucinotta, F. Checconi, Z. Zlatev, J. Papay, M. Boniface, G. Kousiouris, D. Kyriazis, T. Varvarigou, S. Berger, D. Lamp, A. Mazzetti, T. Voith, and M. Stein. Virtualised e-learning with real-time guarantees on the irmos platform. In *2010 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–8, 2010. doi:10.1109/SOCA.2010.5707166.

25 Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Alessio Balsini, and Carlo Vitucci. Reducing temporal interference in private clouds through real-time containers. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 124–131, 2019. doi:10.1109/EDGE.2019.00036.

26 Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Riccardo Mancini, and Carlo Vitucci. Strong temporal isolation among containers in openstack for NFV services. *IEEE Transactions on Cloud Computing*, 2021. doi:10.1109/TCC.2021.3116183.

27 Tommaso Cucinotta, Fabio Checconi, Luca Abeni, and Luigi Palopoli. Adaptive real-time scheduling for legacy multimedia applications. *ACM Trans. Embed. Comput. Syst.*, 11(4), January 2013. doi:10.1145/2362336.2362353.

28 Daniel Bristot de Oliveira, Daniel Casini, and Tommaso Cucinotta. Operating system noise in the Linux kernel. *IEEE Transactions on Computers*, 72(11):196–207, January 2023. doi:10.1109/TC.2023.3248921.

29 Arvind Easwaran, Insik Shin, and Insup Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, 43(1):25–59, September 2009. doi:10.1007/S11241-009-9073-X.

30 Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *Proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010. URL: https://retis.sssup.it/waters2010/waters2010.pdf#page=6.

31 Stefano Fiori, Luca Abeni, and Tommaso Cucinotta. RT-Kubernetes - containerized real-time cloud computing. *Proceedings of the ACM Symposium on Applied Computing*, pages 36–39, April 2022. doi:10.1145/3477314.3507216.

32 Gautam Gala, Carlos Rodriguez, Veaceslav Monaco, Javier Castillo, Gerhard Fohler, Veaceslav Falico, and Sergey Tverdyshev. Monitoring framework to support mixed-criticality applications on multicore platforms. *Proceedings - 2022 25th Euromicro Conference on Digital System Design, DSD 2022*, pages 649–656, 2022. doi:10.1109/DSD57027.2022.00092.

33 Pablo González-Nalda, Ismael Etxeberria-Agiriano, Isidro Calvo, and Mari Carmen Otero. A modular CPS architecture design based on ROS and Docker. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, 11(4):949–955, 2017. doi:10.1007/s12008-016-0313-8.

34 Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In Karin Petersen and Willy Zwaenepoel, editors, *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, Washington, USA, October 28-31, 1996*, pages 107–121. ACM, 1996. doi:10.1145/238721.238766.

35 Arne Hamann, Selma Saidi, David Ginthoer, Christian Wietfeld, and Dirk Ziegenbein. Building end-to-end IoT applications with QoS guarantees. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. doi:10.1109/DAC18072.2020.9218564.

36 Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, April 2017. arXiv:1704.04861.

37 Isser Kadusale, Gautam Gala, and Gerhard Fohler. WASM and containers for real-time serverless edge computing. *JRWRTC 2024*, page 11, 2024. URL: https://cister-labs.pt/rtns24/JRWRTC2024-proceedings.pdf#page=15.

38 Ali Kanso, Edi Palencia, Kinshuman Patra, Jiaxin Shan, Mengyuan Chao, Xu Wei, Tengwei Cai, Kang Chen, and Shuai Qiao. Designing a Kubernetes operator for machine learning applications. *Proceedings of the Seventh International Workshop on Container Technologies and Container Clouds*, 2021. doi:10.1145/3493649.3493654.

39 D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, January 2015. doi:10.1109/JPROC.2014.2371999.

40 Jaewoo Lee, Sisu Xi, Sanjian Chen, Linh T.X. Phan, Chris Gill, Insup Lee, Chenyang Lu, and Oleg Sokolsky. Realizing compositional scheduling through virtualization. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 13–22, 2012. doi:10.1109/RTAS.2012.20.

41 J. Lelli, C. Scordino, L. Abeni, and D. Faggioli. Deadline scheduling in the Linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016. doi:10.1002/SPE.2335.

42 Hennadiy Leontyev and James H. Anderson. A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. *Real-Time Systems*, 43(1):60–92, 2009. doi:10.1007/S11241-009-9076-7.

43 Giuseppe Lipari and Sanjoy Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings Seventh IEEE Real-Time Technology and Applications Symposium*, pages 26–35. IEEE, 2001. doi:10.1109/RTTAS.2001.929863.

44 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

45 Francesco Lumpp, Franco Fummi, Hiren D Patel, and Nicola Bombieri. Containerization and orchestration of software for autonomous mobile robots: A case study of mixed-criticality tasks across edge-cloud computing platforms. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9708–9713. IEEE, 2022. doi:10.1109/IROS47612.2022.9981581.

46 Francesco Lumpp, Franco Fummi, Hiren D Patel, and Nicola Bombieri. Enabling Kubernetes orchestration of mixed-criticality software for autonomous mobile robots. *IEEE Transactions on Robotics*, 40:540–553, 2023. doi:10.1109/TRO.2023.3334642.

47 Gabriele Monaco, Gautam Gala, and Gerhard Fohler. Shared resource orchestration extensions for Kubernetes to support real-time cloud containers. *Proceedings - 2023 IEEE 26th International Symposium on Real-Time Distributed Computing, ISORC 2023*, pages 97–106, 2023. doi:10.1109/ISORC58943.2023.00022.

48 Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/nishihara.

49 Philipp Mundhenk, Arne Hamann, Andreas Heyl, and Dirk Ziegenbein. Reliable distributed systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 287–291. IEEE, 2022. doi:10.23919/DATE54114.2022.9774734.

50 Linh T. X. Phan, Jaewoo Lee, Arvind Easwaran, Vinay Ramaswamy, Sanjian Chen, Insup Lee, and Oleg Sokolsky. CARTS: A tool for compositional analysis of real-time systems. *SIGBED Review*, 8(1):62–63, March 2011. doi:10.1145/1967021.1967029.

51 Kun Ran, Yingbo Cui, Zihang Wang, and Shaoliang Peng. Performance evaluation of Spark, Ray and MPI: A case study on long read alignment algorithm. In *Algorithms and Architectures for Parallel Processing: 23rd International Conference, ICA3PP 2023, Tianjin, China, October 20–22, 2023, Proceedings, Part III*, pages 57–76, Berlin, Heidelberg, 2024. Springer-Verlag. doi:10.1007/978-981-97-0798-0_4.

52 Rui Reis, Pedro M. Santos, Mario J. Sousa, Nuno Martins, Joana Sousa, and Luis Almeida. LEM: a tool for large-scale workflow control in edge-based industry 5.0 applications. *Proceedings - 19th International Conference on Distributed Computing in Smart Systems and the Internet of Things, DCOSS-IoT 2023*, pages 317–323, 2023. doi:10.1109/DCOSS-IOT58021.2023.00059.

53 Insik Shin and Insup Lee. Compositional real-time scheduling framework. *Real-Time Systems Symposium*, pages 57–67, 2004. doi:10.1109/REAL.2004.15.

54 Michael Sollfrank, Frieder Loch, Steef Denteneer, and Birgit Vogel-Heuser. Evaluating Docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation. *IEEE Transactions on Industrial Informatics*, 17(5):3566–3576, 2021. doi:10.1109/TII.2020.3022843.

55 I. Stoica, H. Abdel-Wahab, K. Jeffay, S.K. Baruah, J.E. Gehrke, and C.G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *17th IEEE Real-Time Systems Symposium*, pages 288–299, 1996. doi:10.1109/REAL.1996.563725.

56 Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V. Papadopoulos. Hierarchical resource orchestration framework for real-time containers. *ACM Trans. Embed. Comput. Syst.*, April 2023. doi:10.1145/3592856.

57 Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019. URL: http://proceedings.mlr.press/v97/tan19a.html.

58 The New Stack. How Ray, a distributed AI framework, helps power ChatGPT, 2023. Accessed: 2025-02-17. URL: https://thenewstack.io/how-ray-a-distributed-ai-framework-helps-power-chatgpt.

59 Uber Technologies Inc. How Uber uses Ray to optimize the rides business, 2023. Accessed: 2025-02-17. URL: https://www.uber.com/en-IT/blog/how-uber-uses-ray-to-optimize-the-rides-business/.

60 Stefan Walser, Jan Ruh, and Silviu S Craciunas. Real-time container orchestration based on time-utility functions. In *IEEE 20th International Conference on Factory Communication Systems (WFCS)*, pages 1–8. IEEE, 2024. doi:10.1109/WFCS60972.2024.10541041.

61 Wen Wu, Kaige Qu, Peng Yang, Ning Zhang, Xuemin Sherman Shen, and Weihua Zhuang. Cost-effective two-stage network slicing for edge-cloud orchestrated vehicular networks. *2022 IEEE/CIC International Conference on Communications in China, ICCC 2022*, pages 968–973, 2022. doi:10.1109/ICCC55456.2022.9880642.

62 Sisu Xi, Chong Li, Chenyang Lu, Christopher D. Gill, Meng Xu, Linh T.X. Phan, Insup Lee, and Oleg Sokolsky. RT-Open Stack: CPU resource management for real-time cloud computing. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 179–186, 2015. doi:10.1109/CLOUD.2015.33.

63 Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX workshop on hot topics in cloud computing (HotCloud 10)*, 2010. URL: https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets.