Kubernetes Scheduling with Checkpoint/Restore: Challenges and Open Problems

Viktória Spišaková[1,2(✉)], Radostin Stoyanov[3,7], Lukáš Hejtmánek[2], Dalibor Klusáček[5], Adrian Reber[4], and Rodrigo Bruno[6]

[1] Faculty of Informatics, Masaryk University, Brno, Czech Republic
spisakova@ics.muni.cz
[2] Institute of Computer Science, Masaryk University, Brno, Czech Republic
[3] University of Oxford, Oxford, UK
[4] Red Hat, Munich, Germany
[5] CESNET, Prague, Czech Republic
[6] INESC-ID, Instituto Superior Técnico, University of Lisbon, Lisbon, Portugal
[7] Red Hat, London, UK

Abstract. Efficient resource management and scheduling have been persistent challenges since the early days of computing and remain crucial today. The widespread adoption of containers managed by orchestrators like Kubernetes has introduced new dimensions to these challenges. Despite their lightweight nature, containers still suffer from inefficiencies due to over-provisioning and long-running workloads allocating resources potentially forever. Existing scheduling techniques are not enough to meet these demands, and there is a growing need for orchestration and scheduling policies that support advanced preemption, migration, and fault tolerance. Well-established container Checkpoint/Restore (C/R) mechanisms, implemented through tools like CRIU, offer a promising solution for improving resource scheduling efficiency. However, these mechanisms remain only partially integrated with Kubernetes. In this paper, we propose a novel Interruption-Aware Scheduling Strategy that incorporates transparent C/R with the Kubernetes scheduler. We discuss the current C/R mechanisms, outline the key design choices involved in integrating with the Kubernetes scheduler, and explore associated challenges and open problems. We further discuss potential solutions to these challenges, offering a path toward more efficient resource management to better meet the needs of today's computational landscape.

Keywords: Checkpoint and Restore · Kubernetes · Containers · Resource Management · Scheduling

1 Introduction

Achieving efficient resource management presents a fundamental challenge in large-scale systems, from high-performance computing (HPC) clusters to cloud

D. Klusáček et al. (Eds.): JSSPP 2025, LNCS 16210, pp. 41–62, 2026.
https://doi.org/10.1007/978-3-032-10507-3_3

computing platforms. The performance of such systems is driven predominantly by two key factors: *scheduling strategy* and *workload characteristics*.

While HPC environments focus on optimizing batch processing for specialized engineering and scientific workloads [37], modern cloud-based systems are designed to support a broad range of services [28]. Despite these differences, both domains share a common challenge: *How to achieve optimal resource utilization while maintaining fault tolerance and responsiveness in rapidly changing environments?* The emergence of container orchestration platforms like Kubernetes has transformed cloud computing by enabling the automated deployment, scaling, and management of containerized workloads at scale. However, the nature of these workloads has also become more dynamic and resource-intensive, and as a result, existing scheduling techniques fall short of fully leveraging the elastic capabilities of modern cloud infrastructures [12,36,55]. Therefore, there is a clear need for new, adaptive scheduling techniques and tools to support demands for scalability, performance, and fault tolerance at the same time.

HPC employs a classical scheduling model where workloads are submitted as jobs to a centralized queueing system (ordered by priority). A centralized scheduler effectively manages both job placement and resource allocations, which the job then uses throughout its entire run time. After termination, allocated resources are immediately freed and can be reused, which enables high efficiency in HPC environments. For instance, Fig. 1 demonstrates the efficiency of a batch scheduling strategy within the Czech national e-INFRA CZ [9] distributed infrastructure, which is based on Open PBS. This figure illustrates the ratio between the requested and actually used CPU years of the top 15 users' workloads over one year. For the majority of users, this ratio is more than ∼80%, indicating that most of the requested resources are consumed and are kept busy with high per-user utilization that translates to high cluster utilization. Idle time is minimized as a result of well-managed job life cycles and workload types that are generally better aligned with the requested resources.

Kubernetes is the most widely adopted orchestration framework for containerized environments. However, its default scheduling strategy and the characteristics of typical workloads differ significantly from those of traditional HPC systems. Workloads define resource *requests* and *limits* that are independent of the workload's lifetime, which can span from a few seconds to several months, or even continue effectively "forever". Every workload is scheduled upon creation by the Kubernetes scheduler, which plans according to the *requested resources*, but the workload may utilize resources up to its *defined limits*.

This scheduling strategy allows for flexibility and rapid startup times. However, resources are not reclaimed immediately when demand drops, resulting in resource fragmentation and diminished efficiency across many clusters. This is a fundamental difference between Kubernetes and HPC environments where resources are dedicated for long periods of time and not fully reclaimed. This difference also explains why Kubernetes clusters, particularly at large-scale, fail to achieve high resource utilization when compared to traditional batch systems.

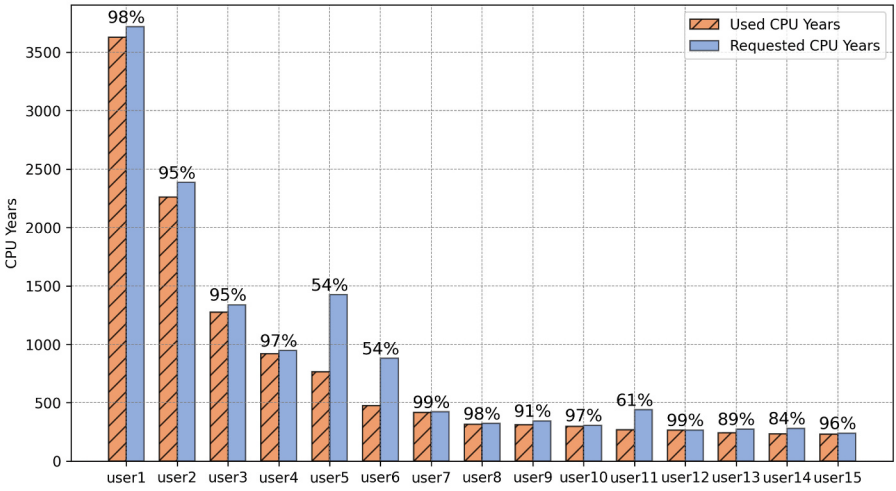


Fig. 1. A comparison between requested CPU time (blue bars) and actually consumed CPU time (orange bars). Percentage above each pair of bars represents the ratio between requested and real for the top 15 users of e-INFRA CZ HPC environment. (Color figure online)

In Fig. 2, we illustrate this problem by showing the ratio of requested CPUs to actual CPU usage for the e-INFRA CZ Kubernetes cluster nodes. This utilization data demonstrates how actual usage ratio can drop in cloud environments. The ratio of real node CPU utilization to total requested CPUs ranges from approximately ∼0.78 to 129%. Removing an outlier where utilization exceeds requests reduces the range to approximately ∼0.78 to 85%, with the average CPU utilization as a percentage of requests around ∼21%. The average real CPU utilization compared to the number of physical CPUs per node is approximately ∼6%.

This gap between requested resources and real usage demonstrates the fundamental problem we aim to address. Computing nodes often appear fully utilized because users overestimate resource needs (e.g., setting CPU and memory significantly higher than what workloads actually consume), while actual utilization remains low. As a result, clusters appear saturated but remain mostly idle, preventing new workloads from being scheduled and leading to unnecessary energy consumption and increased operational costs. In cloud environments, this inefficient resource utilization is primarily driven by over-provisioning and, to some extent, by the lack of resource elasticity [2]. While both HPC and cloud systems suffer from over-requesting, batch systems mitigate this by releasing resources once time-constrained jobs complete. In contrast, Kubernetes may leave resources reserved indefinitely for long-running services with excessive requests, even when those services are idle (see Fig. 3).

Our previous research [41] explored scavenger jobs—low-priority workloads designed to be easily terminated—to improve resource utilization by dynamically

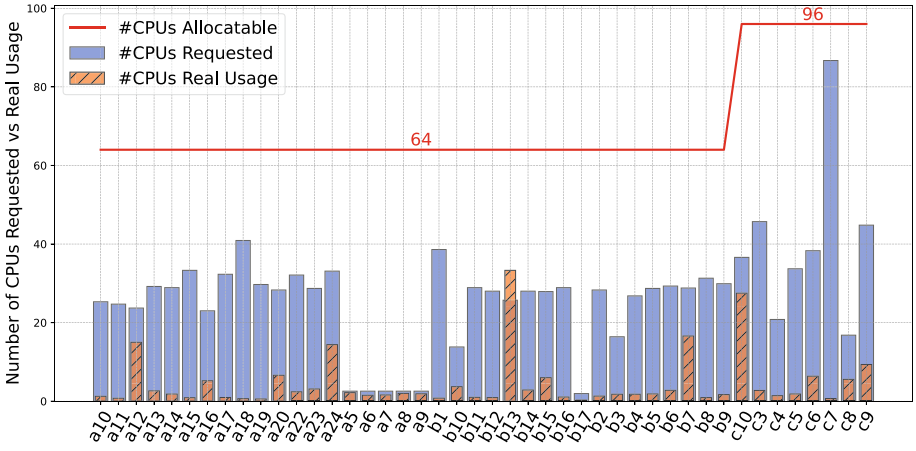
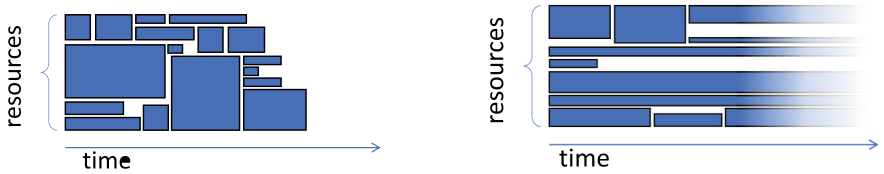


Fig. 2. A comparison of requested CPU resources (blue bars) and actual usage (orange bars) for individual nodes in the e-INFRA CZ Kubernetes cluster, with a red line indicating the number of physical CPUs per node. (Color figure online)

reclaiming idle resources. However, this approach has a crucial limitation: when a scavenger job is preempted, all of its computational progress is lost, wasting the energy and CPU time already consumed. This inefficiency highlights the need for a more advanced strategy. The challenge, therefore, is to develop a mechanism that can dynamically reallocate resources from running workloads without the penalty of discarding completed work, thereby improving overall efficiency and elasticity in cloud environments.

In this paper, we explore a novel approach of integrating transparent Checkpoint/Restore mechanisms as a core scheduling primitive within the Kubernetes container orchestration system. While C/R is an established concept that has been successfully leveraged for purposes such as virtual machine migration in traditional clouds, fault tolerance in HPC, or accelerating cold-starts in serverless or edge environments, our work pioneers its integration with Kubernetes scheduling. This initiative is particularly significant in current computational landscape as Kubernetes is the leading solution for container orchestration, while CRIU stands as the foremost solution for transparent container C/R. By combining both technologies, our research contributes to the field by providing a more efficient and flexible container orchestration solution that addresses the need for cloud elasticity and adaptability in diverse computing environments.

The rest of the paper is organized as follows: Sect. 2 introduces the problem statement that stems from the necessity of cloud elasticity combined with the diversity of cloud workload types and imperfect scheduling. In Sect. 3 we expand more on the reasons why we advocate for C/R as a new scheduling primitive and discuss these mechanisms. The integration of C/R into the scheduling strategy is described in Sect. 4, which delves deeper into the design and architecture of the proposed interruption-aware scheduling strategy. Section 5 explores the open



(a) Limited workload duration in HPC.

(b) Unlimited workload duration in Kubernetes.

Fig. 3. Visualization of workload lifetime characteristics in different computing environments. (a) HPC systems schedule jobs which have a defined start and end time. (b) Cloud-native platforms like Kubernetes manage a mix of workloads, including services designed to run indefinitely (represented by fading bars).

problem of robust C/R integration within Kubernetes ecosystem, providing an overview of the architectural challenges, networking, security considerations, cost awareness and C/R user transparency. We conclude the paper and discuss future work in Sect. 7.

2 The Need for Lossless Workload Preemption

Kubernetes clusters and cloud platforms in general run multi-class workloads all at once—from interactive applications and long-running batch jobs to short, bursty tasks as well as (micro)services that can run essentially forever. Interactive applications may require substantial resources intermittently but not continuously. Serverless workloads demand instant scaling capabilities. Some workload types also have placement limitations—for example, display-forwarding applications rely on specific GPU types that support video encoding. The bursty nature of HPC workloads requires sufficient resource slack and proper fault-tolerance.

Each workload type brings its own set of requirements, especially concerning performance and responsiveness. To support these requirements, cloud environments need elasticity—the ability to scale up or down so that the workload is allocated the capacity it needs when it needs it. Existing Kubernetes features for elasticity include autoscaling and evictions. However, these are not enough. Workloads are not equally elastic, as not all workload types can be automatically scaled with a positive effect, e.g., scaling an HPC-style workload by one more instance will only perform the same computation twice, effectively wasting more resources. Evicting (preempting) workloads is also limited to stateless workloads, since the preemption of stateful workloads would lead to losing all their progress, which is the opposite of efficient scalability.

In this context, it is particularly important to realize that HPC workloads, which are both stateful and hardly scalable by default, are no longer deployed only in batch environments. The situation has undoubtedly changed as HPC workloads are already being executed on the cloud and in containers, especially in the field of life sciences [4]. Thus, as the adoption of container technologies

continues to grow, more and more HPC workloads are inevitably being turned into containers managed by orchestrators like Kubernetes. This trend is further emphasized in the outcomes of Liu's work [21]:

> *"Our achievements demonstrated that containerization technologies can support the convergence of HPC and ML applications, not only keeping the well-known advantages of containerization regarding customization, portability, reproducibility, and fault isolation but providing also performance benefits thanks to fine-grain deployments and resource allocation."*

Consequently, there is a need for a more comprehensive approach that can effectively manage dynamic and heterogeneous workloads without compromising efficiency and scalability, and provide a lossless preemption mechanism. For that, we propose a new scheduling method—*Interruption-Aware Scheduling Strategy* (IASS) that combines transparent checkpoint/restore mechanisms with advanced workload placement and preemption policies.

3 Checkpoint/Restore as a New Scheduling Primitive

Checkpoint/Restore is a mechanism for capturing the execution state of an application at an arbitrary point in time (checkpoint), allowing it to be restored to that state later (restore), potentially on a different machine. *Transparent* C/R refers to a system-level approach that can be used without changing the application's source code or altering the flow of operations within the application.

By introducing C/R capabilities into the scheduler, it could be possible to checkpoint all workloads (thus saving progress) and reschedule them with an optimized schedule, with the definition of *optimized* varying depending on the environment or the application e.g., improving data locality for I/O-intensive workloads, workload consolidation to free up hardware resources for scaling down or maintenance, or catering to a custom priority system.

In practice, achieving a transparent checkpointing mechanism for all application types is not only difficult—it is also infeasible to checkpoint every workload. However, explicit C/R support for a set of workloads (e.g., machine learning training jobs, HPC simulations) has potential in multiple areas (e.g., fault-tolerance, auto-scaling, dynamic load-balancing) to improving resource utilization through a set of new or modified scheduler characteristics and abilities:

(i) **Backfilling**: In this context, backfilling means opportunistic capacity usage where low priority workloads are scheduled onto the free resources with no guarantee of resource availability. When resources are required by higher-priority workloads, these opportunistic workloads are immediately preempted (checkpointed). This strategy aims to maximize resource utilization by filling in the gaps that would otherwise remain idle.

(ii) **Lossless preemption**: Higher-priority or urgent workloads can be scheduled without losing the computational progress of preempted ones, enabled

by checkpoint-based rather than termination-based preemption. This approach allows to effectively generalize preemptibility across workloads, irrespective of their priority level or execution semantics.

(iii) **Dynamic workload migration**: Workload migration refers to the process of transferring a running workload from one node to another while preserving the runtime state of all processes within it. The transfer might be necessary due to fragmentation of available cluster resources, node maintenance or node CPU or memory pressure. The C/R mechanism minimizes disruption of workloads during the workload reallocation.

(iv) **Infrastructure-level fault tolerance**: Periodic checkpointing can be used to protect long-running and stateful workloads from unexpected hardware or node failures. In the case of such an event, the scheduler restarts the workload from the most recent checkpoint instead of resuming from scratch. This mechanism allows to begin execution close to the point of node failure and potentially preserve hours or days of computation.

Incorporating C/R capabilities into the scheduling system enables flexible workload management and advanced optimizations that are completely transparent to workloads and users. From the user's perspective, they can be offered another quality-of-service tier at a lower price and without wait times for resources, but the task might be interrupted during execution. It might be very profitable for users to adopt C/R computing, especially if the task is of a long-running, delay-insensitive nature.

An example of such dynamics are Amazon spot instances, which offer similar opportunistic resource usage without providing reservation guarantees. These instances (and their workloads) can be terminated at any time, typically to accommodate higher-tier reserved instances. However, user receives a short interruption notice (e.g., two minutes) before the instance is ultimately stopped[1]. This interruption notice can be used to trigger a checkpointing mechanism for the running workloads and to rescheduling them later [53]. To enable this level of reliability and improve resource utilization in container-based environments, we explore existing checkpointing mechanisms and evaluate their suitability as core primitives within the Kubernetes scheduling process in the following sections.

3.1 CRIU: Checkpoint/Restore In Userspace

CRIU [7] is a Linux utility that enables transparent system-level checkpointing in a wide range of use cases. It is developed as an open-source project and designed for checkpointing running applications or containers and later restoring them on the same or a different system. It achieves this by using kernel interfaces such as cgroups, ptrace, vmsplice that allow inspecting the target process tree (container) and saving its runtime state to persistent storage. This process is

[1] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-instance-termination-notices.html.

fully transparent to the application that is being checkpointed, and can be used with a wide variety of workloads. In addition, CRIU supports optimizations that reduce the downtime during live migration strategies. CRIU was originally developed to support container migration with OpenVZ and has been integrated into many popular container runtimes, including runc, crun, Docker, Podman, LXC/LXD, containerd, and CRI-O. However, the implementation across runtimes differs in some aspects. For example, Docker leverages containerd, which in turn uses runc to perform checkpointing via CRIU's RPC API. Podman, by default, utilizes crun, which implements the checkpoint/restore operations directly with the libcriu library. LXC, on the other hand, interacts with CRIU through its command-line interface. These implementation differences often lead to varying levels of performance and feature support.

Checkpointing workloads that utilize external devices, such as GPUs, requires saving and restoring the internal execution state of both the GPU and the driver. This functionality has been enabled with CRIU through plugins for AMD [31] and NVIDIA [13] GPUs. These plugins interact directly with the GPU driver via recently introduced checkpointing capabilities that enable transparent checkpointing. For NVIDIA GPUs, this functionality is exposed through a `cuda-checkpoint` utility [43], which allows for the execution of lock, checkpoint, restore, and unlock actions for tasks running on the GPU. For AMD GPUs, this functionality is exposed as Kernel Fusion Driver (KFD) `ioctl` operations [30], which can be used to perform similar actions. These plugins enable checkpoint/restore support with both stand-alone applications and containers, offering unified and fully-transparent CPU-GPU snapshots [52].

3.2 Alternative Checkpoint/Restore Mechanisms

The majority of existing C/R tooling originates in the HPC community, which has long been interested and involved in the utilization of the C/R mechanism because HPC jobs are usually long-running, and the sudden loss of all progress is undesirable. C/R in HPC was originally and is usually implemented on the individual application layer rather than generally due to the specific logic and goals of each application. For example, molecular dynamics tool Gromacs [3] can do periodic checkpoints and after the interruption, it can resume and continue, Nextflow [8] workflow manager can resume failed or interrupted computation from the last completed task. However, many scientific tools lack built-in progress-saving mechanisms because checkpointing was not originally seen as crucial and over time, integrating this functionality has become too difficult due to the applications' growing complexity.

Apart from application-level C/R, DMTCP [1] and BLCR [14] are other well-known C/R tools in the HPC community. However, these approaches rely on users to either write code that implements these mechanisms directly into their application or to use specific libraries that handle checkpointing. Furthermore, BLCR is a kernel module that cannot be used with today's kernel version. These approaches are suboptimal, as they burden application developers and researchers with the additional complexity of saving/restoring program state and

verifying that their application resumes correctly from a checkpoint. The use of CRIU for C/R does not require any cooperation from the user or the applications; it is generic, application-agnostic, lightweight, and can be integrated into fully automated preemption-based scheduling and into modern cloud environments.

Recent years show that the HPC community is increasingly interested in optimized resource utilization and is again pointing toward C/R. In a recent presentation [54], NERSC (National Energy Research Scientific Computing Center[2]) acknowledges that C/R is a crucial capability in HPC due to complex, time-consuming computations and can facilitate scheduler optimizations. In a recent study, Guitart [11] builds on the current trend in HPC to containerize and researches the integration of CRIU and its advanced configuration options with containerization to enable practical live migration of HPC workloads. A more recent paper by Hoefler et al. [15] explicitly argues for the need to converge HPC and cloud approaches and workloads through high-performance containers and presents a comprehensive description of such platform. This shows that the HPC community is not only actively pursuing cloud—container-based environments—but also looking for a C/R mechanism that would be aligned with these environments.

4 IASS: Interruption-Aware Scheduling Strategy

Integration of C/R happens as part of a scheduling strategy. The proposed scheduler is dynamic and treats interruptions and possible subsequent changes in the scheduling plan as a core functionality, not as a form of exception or error. IASS is built on top of four key characteristics:

(i) Dynamic filling-in and vacating the resources using C/R.
(ii) Providing preemption via checkpoint rather than termination-based preemption.
(iii) Inherent support for migration, survivable evictions and infrastructure-based fault-tolerance.

Integrating IASS with Kubernetes scheduling requires targeted changes on multiple levels. Firstly, integrating C/R functionality through CRIU into the Kubernetes stack, and secondly, integration of scheduling decisions and C/R related logic into the Kubernetes scheduler.

4.1 CRIU Integration in Kubernetes

Minimal support for creating container checkpoints from Kubernetes was added in July 2022 [35] as an experimental feature. The container checkpointing functionality was part of the Kubernetes Enhancement Proposal (KEP) "Forensic Container Checkpointing" [33] and as of the current Kubernetes version, it is enabled by default.

[2] https://www.nersc.gov.

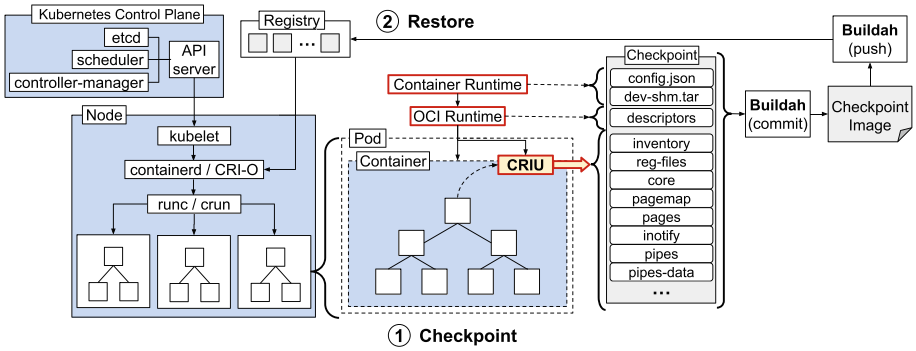


Fig. 4. An overview of the C/R workflow in Kubernetes. Checkpoint is requested through a call to an API server which is a part of the Kubernetes control plane. The call is subsequently passed to the specific node running a container that was requested to be checkpointed. Each node runs a container runtime and an OCI runtime that create and execute containers and support passing the checkpoint request to CRIU. CRIU is used to checkpoint the CPU/GPU state and memory of running containers, which is then packaged with into a checkpoint image and pushed to a container registry. This checkpoint image can later be used to restore the container.

The forensic container checkpointing use case opened the possibility of implementing only the container checkpointing functionality without the need to immediately provide the functionality to restore containers from a checkpoint. As container checkpointing is a completely new concept on the Kubernetes level, one important goal was to avoid disrupting existing functionality and introducing security risks (see Sect. 5.2). To achieve this, the design was limited to only include an interface for creating checkpoints of containers, without providing a corresponding interface for restoring them.

Figure 4 shows the placement of CRIU in the Kubernetes ecosystem. The tool itself is installed on the physical cluster node, and calls to it are integrated into other tools that comprise the whole stack, namely container runtimes, container engines, the kubelet[3], and the Kubernetes API. As of Kubernetes version 1.33, the C/R feature is a functional part of Kubernetes and can be tested.

4.2 IASS Integration with the Kubernetes Scheduler

Incorporating C/R-related scheduling logic into the default Kubernetes scheduler involves exploring two scheduler components: the scheduling framework and internal scheduling queues.

The Kubernetes scheduler is structured as a framework comprising two cycles—scheduling (selects feasible node for a workload) and binding (commits workload to its target node). Both cycles are implemented through multiple plugins (see [18]), with each plugin corresponding to a distinct, customizable

[3] Agent that runs on each node within a Kubernetes cluster.

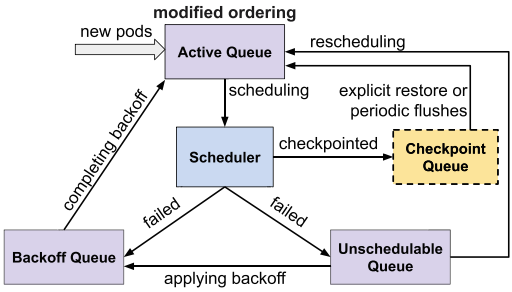


Fig. 5. Kubernetes scheduling queues with added Checkpoint Queue and modified Active Queue ordering. New pods enter the Active Queue for processing by the scheduler. Pods that have failed scheduling are added to the Unschedulable or Backoff queues. After completing backoff, these pods re-enter the Active Queue for another scheduling attempt. Checkpointed pods are added to the Checkpoint Queue, and explicit restore requests or periodic flushes trigger their transition to the Active Queue.

phase within the cycles. Before entering the scheduling cycle, each workload goes through *Pre-enqueue* plugins and subsequently the *QueueSort* plugin that together determine the most suitable workload for the next scheduling cycle.

Pre-enqueue plugins maintain internal queues for scheduling requests and decide which queue the workload enters. When a workload is submitted into the Kubernetes cluster, it is placed in one of the three existing queues. If all workload requirements are met, the workload enters `Active` queue. If the workload cannot be scheduled, it is moved to either `Unschedulable` or `Backoff` queues [19]. The scheduler periodically re-evaluates and moves workloads from these queues back to the Active queue.

At this queue level, we introduce of a dedicated `Checkpoint` queue for checkpointed workloads with its own internal ordering (see Fig. 5). This queue implements workload-specific logic for reallocating workloads and moving them from this queue back to the Active queue, e.g., to backfill free resources by restoring preempted workloads with long-running tasks such as machine learning training jobs. This queue transition can be triggered manually by issuing a *restore request* at the Kubernetes API level or automatically by *periodic flush request*. Periodic flushing prevents starvation of previously checkpointed workloads and ensures that once-started workloads can eventually complete. If a workload is flushed from the Checkpoint queue, it is likely to remain in either Active or Backoff queue as all required objects should already exist in the system.

The selection order of the highest-priority workload from the Active queue for the next scheduling cycle can be customized in the *QueueSort* plugin. By default, workloads are sorted primarily by priority and secondarily by arrival time. The IASS QueueSort extends this default behavior with the logic that accounts for the workloads originating from the Checkpoint queue, allowing for priority scheduling based on workload type, e.g., to prioritize interactive work-

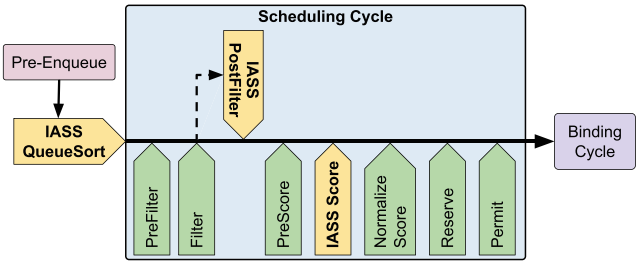


Fig. 6. Simplified Kubernetes scheduler architecture, highlighting the *QueueSort, PostFilter* and *Score* plugins implementing the Interruption-Aware Scheduling Strategy.

loads, or workloads whose checkpoint count exceeds a configurable threshold (to prevent starvation).

The key components of IASS are integrated into the *PostFilter* phase, which implements preemption, and the *Score* phase, which is responsible for ranking suitable nodes (see Fig. 6). The IASS PostFilter plugin enables checkpoint-based workload preemption for through C/R capabilities rather than simple deletion. This PostFilter plugin can be further extended to support more advanced preemption strategies that are specific to the type of workload, its duration, the estimated checkpoint overhead in terms of storage and performance, as well as the checkpoint frequency.

Our previous work explored a new priority class [29] that enables preemption of low-priority workloads using the C/R functionality. IASS integrates this preemption logic directly into the scheduler, which eliminates the requirement for workloads to use a particular priority class. In particular, the IASS Score phase considers C/R-enabled workloads when assessing the suitability of a node. The scheduler improves placement for the workload by ranking nodes higher if they meet certain criteria, such as hosting more checkpointable workloads or having larger local storage. These nodes are preferred because their existing tasks can be preempted efficiently to free up resources and ensure that there is sufficient capacity to store checkpoints.

A natural question that follows is: *What are the advantages of integrating IASS into the default scheduler over developing a specialized one like Volcano*[4] The primary advantage of integrating IASS into the default scheduler is a significantly lower barrier to adoption. Our approach leverages the core Kubernetes components and eliminates the overhead of deploying and managing a separate scheduler. We build IASS on the recently introduced native support for container checkpointing, and explore advanced preemption mechanisms that can be applied to a broad range of use cases within the existing ecosystem. This native integration significantly improves resource efficiency as it avoids the loss of computational progress for long-running workloads.

[4] https://volcano.sh/en/.

5 Open Problems: IASS Integration with Kubernetes

To demonstrate the practical feasibility of IASS, we have created several prototypes showcasing the complete C/R process covering various use-cases, including interactive Jupyter Notebooks [48,57], video streaming [47], an in-memory database [46], and a self-hosted AI platform [45]. We further demonstrated IASS mechanisms with GPU-accelerated workloads such as machine learning training [52] and large language model inference [40,49].

The integration of CRIU with Kubernetes provides a general solution that enables transparent checkpointing for many use cases. The initial support is integrated as container checkpointing for forensic analysis, focused only on the checkpointing functionality and leaving the restore functionality as a subsequent development phase. It is a foundational use case that introduced the C/R functionality as an alpha feature in Kubernetes [34] but it addresses only a specific goal. Moreover, it is currently possible to checkpoint and restore only individual containers (including GPU ones), not entire Pod objects[5], even though it is technically feasible[6]. More robust C/R support in Kubernetes requires addressing several key challenges which can be broadly categorized into five areas: *design and architecture, security, networking, cost-awareness, and policies*. The following subsections delve into each of these areas, discussing the specific problems and potential solutions.

5.1 Design and Architecture: Checkpointed/Restored Workload Specification and Restore Process

When considering a checkpointed and restored Pod, the key question is: *"Should this Pod be a new object, or is it the old one?"*. The problem is analogous to the *Ship of Theseus* thought experiment, yet it might be possible to choose an answer. Additionally, addressing container's state (checkpointed, restored) is a useful information not only for Kubernetes itself but also for other systems too, e.g., external workflow systems that periodically verify workload status.

Design Specification of Checkpointed/Restored Pod. According to the official documentation [26], a Pod's unique identifier (UID) is intended to be unique across the entire lifetime of a Kubernetes cluster as it is intended to distinguish between historical occurrences of similar entities. Therefore, it is crucial to answer whether Pod UID should be retained or new one assigned, as this answers the question about the identity and continuity of a Pod.

From the continuity point of view, retaining the same Pod UID is a reasonable choice as the restored Pod is essentially the same entity. Furthermore, the same

[5] The smallest deployable unit in Kubernetes clusters, representing a group of one or more containers and serving as a typical workload.

[6] In the initial phase of integrating checkpointing into Kubernetes, a sample implementation of full Pod live workload migration was implemented but was not accepted by the community [32].

UID preserves Pod's identity in relation to other objects (e.g., owner references) and avoids potential issues with updating references to the original Pod. On the other side, assigning a new UID for the restored Pod creates a clear distinction between historical occurrences and current state and enables a "fresh" start, allowing for transparent changes in Pod configuration (e.g., change in resource requests) and reducing the risk of conflicts with other components that may have cached references to the original object. There are arguments for both approaches but we argue that **retaining** UID would be the most logical approach. In essence, restoring the Pod from checkpoint does not create a new, distinct entity, it merely restores the state and resumes the operation. Retaining the same Pod ID also contributes to the sense of continuity and greatly simplifies interaction with other Kubernetes components and external systems that rely on Pod identity.

Despite preserving the Pod UID, the system must provide clear information about whether the object is in the checkpointed or already restored state because it affects other decisions. Additional metadata, labels or annotations could be added to the Pod to indicate state change but due to the wider implications C/R has on the system, a more "built-in" approach should be chosen. According to the official API conventions [17], using Pod phases is deprecated and conditions should be employed instead. Conditions provide a standard mechanism for higher-level status reporting and besides being a good way of passing information to users, they also serve as a way of communication between internal and external Kubernetes components. API conventions state that new condition types may be added; thus, we deem this status property a viable place for introducing a new state (condition) that clients need to monitor and the system acknowledges.

Restore Process. The current version of Kubernetes does not provide an interface to directly restore containers. Instead, this process bypasses the entire Kubernetes lifecycle management system and relies on the container engine's ability to recognize checkpoints stored within container images. In particular, to restore a container, a new image must be created from a checkpoint archive and then a completely new container running this image must be launched in Kubernetes, along with a new Pod specification. Such approach prevents Kubernetes from keeping track of the container's checkpoint/restore history.

To address this limitation, the restore process should be natively integrated within the Kubernetes API which calls lower-level restoration endpoints. Furthermore, Pod object definition should be retained as an empty shell, devoid of its containerized contents, to be used to restore the Pod. This way, upon restoration, the checkpointed container would be reinstated and linked back to the original Pod, effectively reviving the Pod's functionality. Once the container is started, Pod condition should be again updated to reflect restored and running (failed/pending) nature of the Pod. This approach allows to extend the Pod lifecycle with C/R states that remain under the purview of Kubernetes.

5.2 Security: Checkpoint Encryption

A container checkpoint contains a snapshot of the application memory, which might include sensitive data such as passwords, keys, and API tokens. With container checkpoints potentially holding sensitive data, security becomes a critical concern for all the use cases. Our recent work extending CRIU with built-in support for encryption creates a foundation for improving the security of the C/R process [51]. Since the technical implementation happens in CRIU, the broader integration in the Kubernetes environment means finding a set of best practices and security standards and define the integration design.

In systems design, security features should be enabled by default, ensuring all users benefit from protection without the need to take any action. Opting out of a security feature should be managed by a permission system that would allow authorized users or special cases to disable the security feature and would also serve as a activity record tool for audits. Disabling the feature should be transparent and provide the user with warnings or alerts, informing them about potential risks and consequences. Based on these principles, encryption should be enabled by default on a kubelet level, with the necessary configuration details located in a kubelet configuration file that is located on every cluster node and features node-specific kubelet configuration. Existing Kubernetes primitives such as Role-Based Access Control (RBAC), admission controllers and Audit API could be leveraged to implement a permission system.

5.3 Networking: IP Address Changes and Distributed C/R

The limitations of Kubernetes network implementation pose challenges for use cases that rely on seamless migration and network continuity, especially the workloads with zero-downtime requirements. The Kubernetes networking ecosystem is based on a Container Network Interface (CNI) with several implementations such as Calico, Cilium, Weave. These CNIs implement networking uniquely, but Kubernetes mandates that each cluster node has its own network prefix for Pods. As a result, when a Pod is migrated from one node to another, regardless of the CNI implementation used, the Pod's primary IP address changes to a new subnet, preventing it from being migrated with the Pod. This results in the Pod's network connection being irretrievably lost. Additionally, even if a Pod does not have an established connection and only listens for incoming connections at a specific Pod IP, the inability to migrate the specific IP prevents the restoration of the listening socket.

A potential solution to this challenge lies in the Multus CNI [25], which permits the allocation of multiple networks and IP addresses to a Pod. This additional network does not require subnets for nodes, thereby facilitating IP migration. We propose assigning multiple networks to Pods requiring stable network connections (such as those running MPI applications) and utilizing the secondary network for communication. This approach can ensure consistent network connectivity for such Pods, even during migration. Another option is to use a load balancer on the data plane that dynamically reroutes packets to the

correct IP address after container migration [10]. This approach can be further improved by offloading the load balancing functionality to a high-performance P4 programmable network device [16,23].

Addressing the stable IP address challenge would pave the way for the implementation of more advanced features such as distributed checkpointing. This involves checkpointing multiple containers or Pods to ensure that a consistent global snapshot has been created and distributed applications can be restored correctly. However, CRIU does not provide native support for checkpoint coordination. We recently proposed a coordinated checkpointing model [44] followed by a proof of concept conference presentations [42,50].

5.4 Cost Awareness: Provenance and Price of Checkpoint/Restore

While C/R has the potential to improve resource utilization, efficiency and fault-tolerance, this functionality requires additional storage to save checkpoints, network bandwidth for data transfers, and adds performance overheads. It is important to carefully assess which workloads are feasible for C/R and when it is worthy to do so. For example, frequent checkpointing or workloads generating very large checkpoints can quickly exhaust available storage capacity. In such cases, checkpointing not only imposes significant storage demands but also consumes additional compute and I/O resources, potentially interfering with the primary execution of the workload. Thus, the number of checkpoints, their frequency, and required storage capacity, depends on the specific workloads and use case. The *checkpoint-restore operator* [5] addresses this by limiting the number of checkpoints per selected object. This approach can be further optimized by integrating data compression mechanisms to decrease the size of checkpoints.

The cost of the restoration process can be particularly high for large checkpoints, primarily due to the time-consuming steps involved in creating container images and performing push/pull operations with the container image registry. To evaluate these overheads, we conducted a series of C/R experiments across a set of applications:

- A simple small container running a shell counter that prints numbers.
- A container that unpacks Linux kernel sources (6.8.0).
- A container that installs and runs `stressapptest`[7] application and allocates 8GB of memory. Purpose is to maximize randomized traffic to memory from processor and I/O, with the intent of creating a realistic high load situation to test the existing hardware devices in a computer.

As shown in Table 1, converting a checkpoint archive into container image and uploading it to a registry introduces significant delays, especially for large checkpoints. A promising optimization is to introduce support for direct restore from checkpoint archives, and avoid the image build stages. In addition, the results also show that collecting changes in almost 89,000 files in the overlay filesystem (Kernel untar case) takes a considerable amount of time (Chckpt.Time),

[7] (https://github.com/stressapptest.

Table 1. Table summarizing checkpoint/restore aspects in Kubernetes: size, time, image creation/upload, and restore time.

Scenario	Chckpt.Size	Chckpt.Time	Restore Image C/U	Restore Time
Simple counter	455 kB	0.93 s	2.1 s	1.0 s
Linux kernel untar	398 MB	68.85 s	16.6 s	8.0 s
`stressapptest`	8 GB	87.36 s	292.8 s	66.0 s

similar to saving approximately 8 GB of memory (`stressapptest`). Thus, both operations (saving large memory or tracking a lot of file changes) incur significant performance overhead. Furthermore, these results show that the duration of checkpoint creation is not directly related to the checkpoint size but rather to the complexity of the operation, which can vary for different workloads.

5.5 Policies: Checkpoint/Restore Transparency to Users

The integration of C/R mechanisms within systems, particularly in environments such as the Kubernetes scheduler, introduces a range of policy-related challenges that must be addressed. While C/R is designed to function transparently between the infrastructure and workloads, wider integration neglects the perspectives of users and external entities. As a result, the implications of C/R may not be readily apparent to these actors, necessitating the development of clear and comprehensive policies.

One key concern is how and if at all to communicate the existence and functionality of C/R to users and external systems. For instance, in workflow management scenarios, tasks are typically interdependent, wherein subsequent tasks rely on the successful completion of prior steps. If a preceding task is interrupted and enters a checkpointed state, should the subsequent tasks be paused, cancelled, or otherwise adjusted? Establishing robust policies for these scenarios is critical to ensure operational continuity and user awareness. Considerations must also be given to the trade-offs associated with checkpointing in broader cloud environments. If a user's workload is checkpointed, what advantages or compensations can be offered? This could include improved resource allocation, cost reductions, or guarantees of workload integrity and availability.

6 Related Work

Some of the first works on modern resource management with C/R have already been explored in a paper by Li et al. [20] that employs CRIU for checkpoint-based preemption of containers in Hadoop YARN. In one of the first discussions on this topic at the Linux Plumbers conference, Google described how they use CRIU-based task migration as a method for improving the efficiency and utilization within large-scale production clusters [56]. A study by Chaudhary et al. [6] further explored the use of CRIU and a custom Kubernetes scheduler to

create a fair scheduling using migration (based on C/R) instead of destructive, termination-based preemption. Mangkhangcharoen et al. [22] provides a comprehensive and comparative study of the applicability of existing C/R mechanisms (incl. CRIU) to checkpoint training states of deep learning (DL) applications among Kubernetes nodes at the edge and later resuming when resources are available (as edge is known for strict resource constraints). A paper by Shukla et al. [39] presents a scheduling service for DL training and inference workloads that is based on the ability to C/R these workloads in Microsoft's distributed infrastructure.

To the best of our knowledge, no prior work has addressed the domain-specific challenges of integrating transparent C/R into the default Kubernetes scheduler. Existing studies typically apply C/R within Kubernetes to improve fault tolerance or enable live migration, but they do not directly incorporate these mechanisms into the default scheduler.

A wider conceptual architectural proposal on C/R of stateful containers prototyped in Kubernetes, albeit only to achieve fault-tolerance, was presented in paper by Müller et al. [24]. A practical and particularly liked approach to realize C/R is a Kubernetes *Operator* pattern. Operators use custom resource definitions to introduce new object types to Kubernetes that are managed by a set of dedicated controllers in an infinite loop. This pattern is often used when introducing new functionality into the ecosystem since it only extends environment's functionality and fully customizable resource definitions and controller allow for any operational logic. A paper by Schmidt [38] proposes a Kubernetes operator that enables to perform checkpoint/restore for a failover use case—monitored applications are regularly checkpointed and in case of need, the application is restored from the latest checkpoint. In a paper Zhang et al. [58] introduce a measure for stateful pod migration using CRIU and Operator pattern. The MemVerge Transparent Checkpoint Operator[8] automatically creates a snapshot for Kubernetes Pods when needed and subsequently restarts the Pod from that snapshot. GRIT[9] is a project that prototypes automated GPU workload migration in a Kubernetes and implements custom resource `Checkpoint`. Slightly different idea was proposed by Onuş at KubeCon Europe 2024 [27] where he proposed a CRIU command wrapper that executes given command, checkpoints it when container is terminated (and restores from checkpoint); the tool is mainly designed for node shutdown and restart scenarios.

7 Conclusion

We propose IASS, an *Interruption-Aware Scheduling Strategy* for the Kubernetes container orchestrator that leverages transparent C/R as a core scheduling primitive. Motivated by the evolving cloud computing architectures and real-world workload characteristics, IASS brings a new solution that provides automated mechanisms to dynamically preempt, migrate, and reschedule running workloads

[8] https://docs.memverge.com/KubernetesTransparentCheckpointOperator/latest/.
[9] https://github.com/kaito-project/grit.

to improve resource utilization. These mechanisms are infrastructure optimizations that are completely transparent to the user.

Future work will focus on the gradual adoption of IASS concepts tailored for a set of workloads within our Kubernetes environment by targeting those with small checkpoint sizes to minimize application downtime and the associated overhead. To further understand the implications of IASS across different workload types, future work will conduct a comprehensive cost-benefit analysis that considers multiple scenarios. We also recognize the importance of fostering collaboration with the Kubernetes community in order to achieve consensus across different working groups and address the broader community objectives. Resource underutilization remains a major challenge in cloud environments, and C/R can enhance the efficiency of scheduling and help solve the problem.

Acknowledgments. Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Ansel, J., Arya, K., Cooperman, G.: DMTCP: transparent checkpointing for cluster computations and the desktop. In: 2009 IEEE International Symposium on Parallel & Distributed Processing, pp. 1–12. IEEE (2009)
2. Baset, S.A., Wang, L., Tang, C.: Towards an understanding of oversubscription in cloud. In: 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12). USENIX Association, San Jose, CA (2012). https://www.usenix.org/conference/hot-ice12/workshop-program/presentation/baset
3. Bekker, H., et al.: GROMACS - a parallel computer for molecular-dynamics simulations. In: DeGroot, R., Nadrchal, J. (eds.) PHYSICS COMPUTING '92, pp. 252–256. World Scientific Publishing (1993)
4. Brum, R., Teylo, L., Arantes, L., Sens, P.: Ensuring application continuity with fault tolerance techniques. In: Borin, E., Drummond, L.M.A., Gaudiot, J.L., Melo, A., Melo Alves, M., Navaux, P.O.A. (eds.) High Performance Computing in Clouds : Moving HPC Applications to a Scalable and Cost-Effective Environment, pp. 191–212. Springer International Publishing, Cham (2023). https://doi.org/10.1007/978-3-031-29769-4_10
5. Checkpoint-Restore operator (2025). https://github.com/checkpoint-restore/checkpoint-restore-operator
6. Chaudhary, S., Ramjee, R., Sivathanu, M., Kwatra, N., Viswanatha, S.: Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In: Proceedings of the Fifteenth European Conference on Computer Systems. EuroSys '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3342195.3387555
7. CRIU: Checkpoint/restore in Userspace. https://criu.org/ (2025). Accessed 11 Feb 2025

8. Di Tommaso, P., Chatzou, M., Floden, E.W., Barja, P.P., Palumbo, E., Notredame, C.: Nextflow enables reproducible computational workflows. Nat. Biotechnol. **35**(4), 316–319 (Apr2017)
9. e-INFRA CZ: e-infrastructure for research and development in the Czech Republic (2025). https://www.e-infra.cz/en
10. Estes, P., Murakami, S.: Live container migration on OpenStack. https://www.openstack.org/summit/barcelona-2016/summit-schedule/events/15091/live-container-migration-on-openstack (2016). openStack Summit Barcelona
11. Guitart, J.: Practicable live container migrations in high performance computing clouds: diskless, iterative, and connection-persistent. J. Syst. Architect. **152**, 103157 (2024)
12. Guo, J., et al.: Who limits the resource efficiency of my datacenter: an analysis of Alibaba datacenter traces. In: Proceedings of the International Symposium on Quality of Service. IWQoS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3326285.3329074
13. Gurfinkel, S.: Checkpointing CUDA Applications with CRIU. https://developer.nvidia.com/blog/checkpointing-cuda-applications-with-criu/ (2024)
14. Hargrove, P.H., Duell, J.C.: Berkeley lab checkpoint/restart (BLCR) for Linux clusters. In: Journal of Physics: Conference Series. vol. 46, p. 494. IOP Publishing (2006)
15. Hoefler, T., et al.: XaaS: acceleration as a service to enable productive high-performance cloud computing. Comput. Sci. Eng. **26**(3), 40–51 (2024). https://doi.org/10.1109/MCSE.2024.3382154
16. Kosorin, S.: P4-enabled container migration in Kubernetes. https://summerofcode.withgoogle.com/programs/2024/projects/sYbpOJhD (2024)
17. Kubernetes API conventions. https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md (2023)
18. Kubernetes scheduling framework. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/ (2023)
19. Kubernetes scheduler queues. https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scheduling/scheduler_queues.md (2021)
20. Li, J., Pu, C., Chen, Y., Talwar, V., Milojicic, D.: Improving preemptive scheduling with application-transparent checkpointing in shared clusters. In: Proceedings of the 16th Annual Middleware Conference, p. 222–234. Middleware '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2814576.2814807
21. Liu, P.: Convergence of high performance computing, big data, and machine learning applications on containerized infrastructures. Ph.D. thesis, Universitat Politècnica de Catalunya (2023)
22. Mangkhangcharoen, S., Haga, J., Rattanatamrong, P.: Migrating deep learning data and applications among Kubernetes edge nodes. In: 2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCCDSSSmartCityDependSys), pp. 2004–2010 (2021). https://doi.org/10.1109/HPCC-DSS-SmartCity-DependSys53884.2021.00299
23. Miao, R., Zeng, H., Kim, C., Lee, J., Yu, M.: SilkRoad: making stateful layer-4 load balancing fast and cheap using switching ASICs. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, pp. 15–28. SIGCOMM '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3098822.3098824

24. Müller, R.H., Meinhardt, C., Mendizabal, O.M.: An architecture proposal for checkpoint/restore on stateful containers. In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, pp. 267–270. SAC '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3477314.3507221
25. Multus CNI. https://github.com/k8snetworkplumbingwg/multus-cni
26. Object names and IDs. https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#uids
27. Onuş, M.: The party must go on - resume pods after spot instance shut down. https://sched.co/1YeP3 (2024)
28. Parayil, A., et al.: Towards workload-aware cloud efficiency: a large-scale empirical study of cloud workload characteristics. In: Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering, pp. 136–146. ICPE '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3676151.3722008
29. Radostin Stoyanov, A.R.: Preemptive scheduling of stateful GPU Intensive HPC applications in Kubernetes. https://sc23.conference-program.com/presentation/?id=misc217&sess=sess448 (2023)
30. Rajneesh Bhardwaj: drm/amdkfd: CRIU Introduce Checkpoint-Restore APIs (2021). linux Commit ID: 3698807094ecae945436921325f5c309d1123f11
31. Rajneesh Bhardwaj, Felix Kuehling, D.Y.S.: Fast checkpoint restore for GPUs. https://lpc.events/event/11/contributions/891 (2021). Accessed 15 Aug 2023
32. Reber, A.: Add –checkpoint to drain. https://github.com/kubernetes/kubernetes/pull/97194 (2020)
33. Reber, A.: Forensic container checkpointing. https://github.com/kubernetes/enhancements/issues/2008 (2020)
34. Reber, A.: Forensic container checkpointing in Kubernetes. https://kubernetes.io/blog/2022/12/05/forensic-container-checkpointing-alpha/ (2022)
35. Reber, A.: Minimal checkpointing support. https://github.com/kubernetes/kubernetes/pull/104907 (2022)
36. Reiss, C., Tumanov, A., Ganger, G.R., Katz, R.H., Kozuch, M.A.: Heterogeneity and dynamicity of clouds at scale: google trace analysis. In: Proceedings of the Third ACM Symposium on Cloud Computing. SoCC '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2391229.2391236
37. Rodrigo, G.P., Östberg, P.O., Elmroth, E., Antypas, K., Gerber, R., Ramakrishnan, L.: Towards understanding HPC users and systems: a NERSC case study. J. Parallel Distrib. Comput. **111**, 206–221 (2018)
38. Schmidt, H., Rejiba, Z., Eidenbenz, R., Förster, K.T.: Transparent fault tolerance for stateful applications in Kubernetes with checkpoint/restore. In: 2023 42nd International Symposium on Reliable Distributed Systems (SRDS), pp. 129–139 (2023). https://doi.org/10.1109/SRDS60354.2023.00022
39. Shukla, D., et al.: Singularity: Planet-scale, preemptive and elastic scheduling of ai workloads (2022). https://arxiv.org/abs/2202.07848
40. Spišáková, V., Stoyanov, R., Reber, A.: Efficient transparent checkpointing of AI/ML workloads in Kubernetes. In: Proceedings of KubeCon + CloudNativeCon Europe 2025. Cloud Native Computing Foundation (2025)
41. Spišaková, V., Klusáček, D., Hejtmánek, L.: Using Kubernetes in academic environment: problems and approaches. In: Klusáček, D., Julita, C., Rodrigo, G.P. (eds.) Job Scheduling Strategies for Parallel Processing, pp. 235–253. Springer Nature Switzerland, Cham (2023)

42. Spišaková, V., Stoyanov, R., Reber, A.: Checkpoint coordination for distributed containerized applications. https://lpc.events/event/18/contributions/1803/ (2024)
43. Steven Gurfinkel: CUDA Checkpoint and Restore Utility. https://github.com/NVIDIA/cuda-checkpoint
44. Stoyanov, R.: Checkpointing and rollback-recovery of distributed applications in Kubernetes. https://radostin.io/files/Red-Hat-RIG-13-04-2023.pdf (2023)
45. Stoyanov, R.: Container Checkpoint/Restore of Open-WebUI + Ollama (2024). https://youtu.be/SObC5tZ-MbM?si=_BjtnxEkM9T-Dqhw
46. Stoyanov, R.: Container Checkpoint/Restore of Redis (2024). https://youtu.be/-7lC7caiYTI?si=svZPdY-b-Rauah4t
47. Stoyanov, R.: Container Checkpoint/Restore of Restreamer (2024). https://youtu.be/ITWi15X7j78?si=6xBJ075kEuz5mqaj
48. Stoyanov, R.: Container checkpoint/restore of a Jupyter Notebook (2025). https://youtu.be/WQNmKMCZ5wk?si=WgxmW54FXCooawrq
49. Stoyanov, R.: Optimizing resource utilization for interactive GPU workloads (2025). https://youtu.be/40qKIU1pj88?si=aB-MJ_8Wa_5TdG_u
50. Stoyanov, R., Reber, A.: Enabling coordinated checkpointing for distributed HPC applications. https://sched.co/1YeT4 (2024)
51. Stoyanov, R., Reber, A., Ueno, D., Clapiński, M., Vagin, A., Bruno, R.: Towards efficient end-to-end encryption for container checkpointing systems. In: Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems, pp. 60–66. APSys '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3678015.3680477
52. Stoyanov, R., et al.: CRIUgpu: Transparent Checkpointing of GPU-Accelerated Workloads (2025). https://arxiv.org/abs/2502.16631
53. Teylo, L., Brum, R.C., Arantes, L., Sens, P., Drummond, L.M.d.A.: Developing checkpointing and recovery procedures with the storage services of amazon web services. In: Workshop Proceedings of the 49th International Conference on Parallel Processing. ICPP Workshops '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3409390.3409407
54. Timalsina, M.: Checkpointing and restarting jobs with DMTCP inside the container. https://www.nersc.gov/assets/DataDay2024/Checkpoint-Restart_DataDay_24.pdf (2024)
55. Tirmazi, M., et al.: Borg: the next generation. In: Proceedings of the Fifteenth European Conference on Computer Systems. EuroSys '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3342195.3387517
56. Victor Marmol, A.T.: Task migration at scale using CRIU. https://lpc.events/event/2/contributions/69/ (2018)
57. Viktória Spišaková, R.S.: Optimizing resource utilization for interactive GPU workloads with transparent container checkpointing (2025)
58. Zhang, H., et al.: KubeSPT: stateful pod teleportation for service resilience with live migration. IEEE Trans. Serv. Comput. **18**(3), 1500–1514 (2025). https://doi.org/10.1109/TSC.2025.3564888