Enabling Kubernetes Workload Execution on Rootless HPC Systems with KSI: A Slurm Integration Framework

Jonathan Decker*, Mojtaba Akbari†, Ali Doosthosseini*,
Sören Metje*, Aasish Kumar Sharma† and Julian Kunkel*
*Institute for Computer Science, University of Göttingen,
Goldschmidtstraße 7, 37077 Göttingen, Germany
e-mail: jonathan.decker@uni-goettingen.de | adoosth@gwdg.de
soerenmetje@yahoo.de | julian.kunkel@gwdg.de
†Working Group Computing, GWDG,
Burckhardtweg 4, 37077 Göttingen, Germany
e-mail: mojtaba.akbari@gwdg.de | aasish.sharma@uni-goettingen.de

***Abstract*—Kubernetes has become a widespread orchestrator for cloud workloads but with increasing demand for compute the need arises to also access HPC environments that are operated via batch schedulers such as Slurm. A number of solutions for combining Slurm and Kubernetes are available, which can be categorized further based on the interaction between Slurm and Kubernetes that they provide. We consider the use case of utilizing an existing Slurm cluster to run Kubernetes workloads. In a previous publication we had introduced Kind Slurm Integration (KSI) based on Kind and rootless Podman and compared its performance, usability and maintainability to the existing solutions Bridge Operator and High-Performance Kubernetes (HPK). We found that Bridge Operator provides native performance as it effectively submits Slurm jobs through a Kubernetes interface and that HPK provides good performance by creating almost feature complete Kubernetes clusters on top of Apptainer. KSI on the other hand was able to provide fully functional Kubernetes clusters inside Slurm jobs but lacked behind in network performance and did not support multi-node clusters. In this work we present a new version of KSI with improved network performance through bypass4netns and support for multi-node clusters via Liqo. Overall, we conclude that running Kubernetes workloads under Slurm is possible with acceptable overhead in terms of performance and without missing out on features.**

***Keywords*-Kubernetes; HPC; Container; Slurm; Cloud.**

I. Introduction

This work discusses an extension of our previous publication [1], which had introduced the first version of KSI. We have since then improved on it and addressed several shortcomings. The motivation for the development of the first version and the improved version of KSI are the same and are given in the following.

Kubernetes has established itself as a widespread solution for orchestration of cloud workloads [2][3] and is used for various workloads including service computing, running large amounts of micro services, as well as batch jobs, such as data analytics or machine learning. However, batch jobs would fit better into HPC environments where powerful high-performance compute and networking resources are available. HPC workloads are commonly scheduled using a batch scheduler such as Slurm [4] but Kubernetes itself can also be used for scheduling HPC jobs using a batch scheduler such as Volcano [5] and have already been scaled to large clusters using appropriate workarounds [6].

Nevertheless, while Kubernetes brings a large array of features, its virtualization layers incur a performance overhead compared to the bare metal performance [7] that could be achieved with Slurm.

Users might want to bring their Kubernetes workloads into Slurm-based HPC environments to benefit from the reduced overhead compared to a regular Kubernetes cluster or to gain access to additional compute hardware, which could also include specialized hardware only available in HPC environments. However, rewriting Kubernetes workloads to be executable in Slurm might require significant effort and expertise with the scheduling systems. Nevertheless, various approaches and implementations exist for combining Slurm and Kubernetes enabling users to dynamically move workloads between cloud and HPC environments.

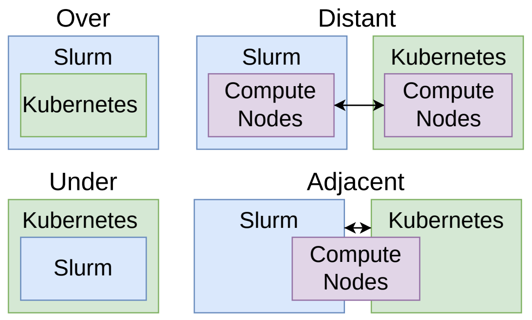

Figure 1. Overview of the four integration models based on the definition by Wickberg [8]. In the **Distant** model, the arrows represent a possible system for negotiating resources between Slurm and Kubernetes such that nodes could be moved as needed. In the **Adjacent** model, the arrows represent bridging tools that enable the execution of workloads under the other scheduling system, respectively.

As there have been various efforts to combine Kubernetes and Slurm, we consider the definition by Wickberg of Schedmd [8] who defines four categories from the perspective of Slurm for such approaches.

- **Over**: The entire Kubernetes environment exists within a Slurm job and is therefore temporary as it is fully removed once the job completes.

- **Distant**: Compute nodes are part of either a Kubernetes or a Slurm cluster and may be moved between the clusters.
- **Adjacent**: Slurm and Kubernetes utilize some form of plugins or bridging tools to cooperate but can still be used individually.
- **Under**: Kubernetes runs a Slurm cluster within its own environment across one or more pods.

These four models are visualized in Figure 1.

Given the above use case of running Kubernetes jobs in an existing Slurm environment, this fits the **Over** or **Adjacent** model. We have investigated existing solutions that implement either of these models and found various approaches that provide the **Adjacent** model but no system for having a Kubernetes cluster running within a Slurm job as described in the **Over** model. Therefore, we present **Kind Slurm Integration (KSI)** [9], an implementation of the **Over** model based on Kubernetes in Docker (Kind) [10]. We have systematically evaluated and compared KSI to existing solutions including **Bridge Operator** by IBM [11], **WLM-Operator** by Sylabs [12], **kube-slurm** by Kalen Peterson [13] and **High-Performance Kubernetes (HPK)** [14].

Our evaluation consists of a review of the state of the respective projects with regard to features and maintainability as well as a performance analysis to determine the overhead incurred by the respective approach. For this purpose, we benchmarked the solutions based on workload startup time, CPU compute performance, memory throughput, storage throughput, network latency as well as network throughput, and compared the results to bare metal. We found that not all of the implementations listed above were able to pass a minimal functionality test. For those that passed, no significant differences in CPU compute performance, memory throughput and storage throughput were found. However, our original implementation of KSI, as presented in [1], was outperformed in terms of startup time and network performance.

Furthermore, we had investigated whether a given approach supports all Kubernetes features including controlling workloads through `kubectl` and providing Kubernetes network abstractions such as services. We found that KSI provided the most complete support for Kubernetes features compared to the others. Bridge Operator, for instance, submits Slurm jobs through Kubernetes such that workloads are executed as scripts outside of Kubernetes. In the case of HPK, it treats Slurm worker nodes as part of its Kubernetes cluster but does not support `kubectl exec` and Kubernetes services.

In this paper we present an improved version of KSI with two additions that enable it to overcome its most significant shortcomings. To ensure it is clear what version of KSI we are referring to, we will note the original version of KSI as introduced in [1] as KSI 1 and the new version presented in this work as KSI 2 for the rest of the paper. If we refer to KSI without specifying which version, then a given statement applies to both versions of KSI unless specified otherwise in the direct context.

KSI 1 presented in [1] did not support multi-node clusters and fell behind in network performance. KSI 2 utilizes bypass4netns [15][16] instead of slirp4netns for rootless container networking, which significantly improves the network performance. Moreover, we have also integrated Liqo [17] such that KSI 2 can now be deployed across all nodes in a given Slurm job and assemble itself into a single Kubernetes cluster. Both versions of KSI implement the **Over** model of combining Slurm and Kubernetes such that it can be executed in a Slurm job without requiring an existing control plane.

Overall, this paper contributes a systematic evaluation of existing approaches that implement the **Adjacent** or **Over** model to combine Slurm and Kubernetes as well as the design and implementation of KSI. We cover both the original implementation in KSI 1 as given in [1] and the improvements introduced in this work for KSI 2. Finally, we provide an overview of the features and limitations of the evaluated approaches. It should be noted that KSI 1 is based on the master's thesis of one of the authors [18].

The remainder of the paper is organized as follows: In Section II, the various implementations for integrating Slurm and Kubernetes are discussed. The methods for benchmarking and comparing the solutions as well as the design of KSI are presented in Section III. The results of the evaluation are given in Section IV. Finally, Section V provides the conclusion and outlook for future work.

II. Related Work

To properly distinguish various approaches for combining Slurm and Kubernetes, we discuss the four models along with notable examples. We also cover related approaches that do not use either Slurm or Kubernetes and then provide a more in-depth look at the implementations, which we evaluated in this paper.

A. Models for Integrating Slurm and Kubernetes

The four categories for combining Slurm and Kubernetes defined by Wickberg of Schedmd [8] are **Over**, **Distant**, **Adjacent** and **Under** as defined in Section I.

a) ***Distant** model:* Notable implementations include [19] and [20], which both implement systems for dynamically changing the partitioning of a node pool between a Kubernetes and Slurm cluster.

b) ***Under** model:* Contributions have been made in [21], [22] and [23], in which Slurm is being run as a set of Kubernetes pods. A significant project in this category is Slinky [24] by Schedmd who had created a Slurm Kubernetes bridge implementation as a proof-of-concept before creating Slinky. The proof-of-concept implementation followed the **Adjacent** model, was not fully functional and has since then been removed from public access.

c) ***Adjacent** model:* Approaches in this category are relatively diverse in their methods including Bridge Operator [11], WLM-Operator [12] and HPK [14]. Each of these approaches is discussed in more detail in Subsection II-C.

d) ***Over** model:* There are no notable implementations of this model except for KSI 1 [25] and KSI 2 [9], which are presented in detail in Subsection III-B.

B. Other Approaches for Integrating HPC and Cloud

While this work focuses on combining Slurm and Kubernetes, it should be noted that there are alternative approaches to running HPC workloads through a cloud interface. For example, as mentioned in Section I, Volcano [5] is an extension for the Kubernetes scheduler, which implements features such as batch and gang scheduling. This enables the execution of batch workloads through Kubernetes without Slurm as shown in [26][27].

Another notable approach is hpc-connector [28] presented in [20], which enables the submission of jobs through an arbitrary cloud interface to be executed via Slurm. This approach can be considered similar to Bridge Operator but is not bound to Kubernetes but also lacks deeper integration with any specific cloud platform to enable advanced features.

Finally, there is [29], which introduces an integration of TORQUE [30], another HPC batch scheduler, with Kubernetes. This enabled scheduling of HPC workloads through Kubernetes to TORQUE similar to Bridge Operator, which would therefore match the **Adjacent** model.

C. Implementations for **Adjacent** Slurm and Kubernetes

1) WLM-Operator: Sylabs Inc. had developed the WLM-Operator [31] and Singularity-CRI [32] with Singularity-CRI providing a Kubernetes-compatible implementation of the Container Runtime Interface (CRI) for Singularity [12]. The WLM-Operator implements a Kubernetes operator that is able to interface with Slurm such that Slurm nodes become visible in Kubernetes as virtual nodes.

Moreover, it provides a Custom Resource Definition (CRD) in Kubernetes called *SlurmJob*, which enables the submission of Slurm jobs through Kubernetes. When submitting a SlurmJob, a dummy pod is created in Kubernetes and the actual job is submitted to Slurm to be run in a Singularity container. The results are then collected through another pod via a shared storage before closing the dummy pod once the job completes.

However, on December 30th 2020, both WLM-Operator and Singularity-CRI projects have been archived on Github with no further development planned.

2) Bridge Operator: IBM had developed Bridge Operator [33] for a Kubernetes cluster to be able to access external compute resources including Slurm clusters [11]. Bridge Operator implements a Kubernetes operator and provides the *BridgeJob* CRD, which accepts all the details required to launch a Slurm job including the remote URL of a Slurm cluster, what resources to request and a remote storage configuration.

For each BridgeJob, the Bridge Operator starts a monitoring pod and submits the job to Slurm. The monitoring pod regularly updates a Kubernetes ConfigMap with the current status and fetches the job output. The creators of Bridge Operator have also demonstrated how to run Kubeflow workloads through BridgeJobs [34], however, as these jobs are converted to Slurm jobs, no Kubernetes pods are directly being run in Slurm.

3) HPK: HPK [35] is presented in [14] and [36] as a way to run Kubernetes workloads on Slurm through Apptainer [37]. It is deployed as a single Apptainer container that runs the Kubernetes control plane and a custom implementation of virtual Kubelet [38], which presents an entire Slurm cluster as a single node in the cluster. Every time a new pod is to be scheduled, it submits a job through Slurm for the pod to be started as a container using Apptainer.

For the container networking to function, it relies on Flanneld service [39] to be installed on the nodes and the Flannel-CNI plugin [40] to be installed for Apptainer. However, only headless services without cluster IPs are supported as the additional layer of load balancing is not possible with the employed networking stack. Moreover, the command `kubectl exec`, which serves to execute commands inside Kubernetes pods, is not supported.

4) Kube-Slurm: The kube-slurm project [13] provides a tool for controlling Kubernetes resources using Slurm jobs. When deploying, Slurm and Kubernetes must both be installed on the same set of nodes with `kubectl` available on all nodes. Once deployed, users can submit Slurm jobs, which get scheduled by the tool as Kubernetes pods onto the nodes selected by the Slurm scheduler.

The deployment can also be completed with the **Under** model by having Slurm run within Kubernetes but still using Slurm to schedule the pods. Nevertheless, due to the way the access is provided to the Slurm scheduler, all users receive the same access to the Kubernetes cluster making this approach unfit for multi-user setups with potentially malicious users.

III. Methodology

This work focuses on approaches for combining Kubernetes and Slurm that allow for running workloads on an existing Slurm cluster following the **Over** or **Adjacent** model and investigates the suitability of the existing solutions. For that purpose we define the following research questions:

RQ1 Can workloads be submitted using Kubernetes tooling, e.g., `kubectl`?
RQ2 Can workloads be scheduled and executed on machines managed by an existing Slurm cluster without root access?
RQ3 Can workloads be executed across multiple machines in parallel?
RQ4 What is the performance overhead imposed by the tool?
RQ5 Is the tool easy to operate for the end user?
RQ6 Is the tool well maintained?

RQ1, **RQ2** and **RQ3** define the functional requirements. For a solution to be a valid approach for utilizing a Slurm cluster through Kubernetes, it should answer yes to at least **RQ1** and **RQ2** with a yes to **RQ3** being desirable but not strictly required. Notably, these requirements do not include whether a solution must be able to run Kubernetes workloads or if it may run Slurm workloads through a Kubernetes interface. For example, Bridge Operator accepts Slurm workloads submitted through Kubernetes while HPK processes Kubernetes workloads submitted through a Kubernetes interface. Still, both solutions execute the workloads on a Slurm cluster.

Moreover, the distinction between the **Over** and **Adjacent** model breaks down to whether the deployment requires an existing component, such as a Kubernetes control plane. For example, Bridge Operator and HPK, which both follow the **Adjacent** model, require a running Kubernetes control plane, to which a user submits their workloads. The respective tool then submits the workload to Slurm. KSI, on the other hand, implements the **Over** model such that a user would instead submit their workload by calling Slurm.

RQ4 is concerned with the performance cost of a given solution. Depending on the architecture and optimization of a given implementation, it may cost additional compute power or delay the start of workloads, which should be minimal for an application to fully harvest the power of HPC machines.

RQ5 and **RQ6** cover the usability and maintainability of a given software providing an indication for the viability in productive use.

While **RQ1**, **RQ2** and **RQ3** can be answered as yes or no questions, **RQ4**, **RQ5** and **RQ6** require a graded answer. We use a three point scoring from + (good) over ○ (fair) to − (bad) to be able to quickly compare the results for multiple implementations. + is the best score, which is given if the implementation fulfills the requirements without any significant drawbacks. ○ is the middle score, which indicates that some limitations apply and − is the lowest score, which applies if significant shortcomings exist.

A. Selection of Implementations to Evaluate

In Section II-C, we had introduced the WLM-Operator, Bridge Operator, HPK and Kube-Slurm. Before starting our evaluation we performed a minimal functionality test and found that the latest version of WLM-Operator is no longer functional on recent operating systems. Despite our best efforts and reaching out to Sylabs, we were unable to reproduce the minimal examples in the repository. Therefore, WLM-Operator can be considered retired and we will not further consider it.

Kube-Slurm requires the installation of a Kubernetes cluster on all nodes as part of its deployment, which violates **RQ2** that it must be able to operate without root access. Therefore, we will not further consider Kube-Slurm.

This only leaves HPK and Bridge Operator as viable targets for further evaluation along with KSI, which is introduced in the next section. However, when testing Bridge Operator we ran into a number of issues, which we reported on Github and created a pull request [41] with our code adjustments.

B. Kind Slurm Integration (KSI) Design

Our main objectives of designing another approach for combining Slurm and Kubernetes were that it should follow the **Over** model and support all Kubernetes features. Following the **Over** model, KSI can be run strictly inside Slurm jobs without relying on external components. This was important to us, as our use case involved a multi-user HPC system in which the users of KSI would not be able to deploy a control plane outside of their Slurm jobs as it is required by Bridge Operator or HPK. Moreover, as we could not find any existing projects employing the **Over** model, we consider this a research gap.

We utilized rootless Kind [10] to create a script that receives a Kubernetes workload, initializes a cluster inside a Slurm job, executes the workload and then closes the cluster as the Slurm job ends. Before settling on rootless Kind, we also considered Minikube [42], K3D [43] and Usernetes [44] but found rootless Kind to be the most suitable.

Kind [45] was developed with local development and automatic testing of Kubernetes in mind. It can deploy a fully functional Kubernetes cluster on a single node by deploying a "node" image, which internally runs all containers belonging to the cluster as nested containers. From the perspective of the host system, only a single container is running for the control plane node of the cluster. Moreover, by deploying multiple "node" images on the same host, Kind can simulate a multi-node cluster.

For operating KSI inside of Slurm jobs without access to root permissions, which are required for regular container operation, we employ rootless Kind [10]. Rootless Kind relies on a container runtime that supports rootless container deployment, which typically relies on Cgroups v2 features in the Linux kernel. In KSI 1 [1] we relied on rootless Podman, which in turn relies on the shadow-utils package to provide subuids and subgids for user namespaces. Since then we have switched to rootless Containerd for KSI 2 as it has better support for rootless container networking. Nevertheless, the dependencies employed by Podman and Containerd to enable rootless mode, are similar if not identical. For example, a central component for both versions of KSI is RootlessKit [46], a fakeroot implementation specifically for rootless containers.

In order to deploy KSI, the nodes must provide a recent Linux operating system with support for Cgroups v2. Rootless Podman or rootless Containerd must be set up as well as slirp4netns [47], to provide networking for rootless containers. Slirp4netns is the default network driver for rootless Containerd and used to be the default for rootless Podman until Podman 5.0 [48] was released, which uses pasta [49].

However, the Podman documentation [50] states that employing slirp4netns can lead to degraded network performance. We confirmed this in our publication [1] and noted it as a significant shortcoming of KSI 1. In order to overcome this shortcoming we replaced slirp4netns with bypass4netns [15][16] for KSI 2.

Bypass4netns is an experimental project that employs `SECCOMP_IOCTL_NOTIF_ADDFD`, a Seccomp filter introduced into the Linux kernel with version 5.9. It builds on top of slirp4netns by capturing socket syscalls made by slirp4netns and executing them in the host network namespace. By doing so, the authors of bypass4netns effectively achieved the same network throughput for rootless containers as for rootful containers that set the `--net=host` flag. This enables the deployment of rootless containers with the same network performance as rootful containers without compromising on security by exposing the host network.

Bypass4netns can be used with rootless Docker, Podman and Nerdctl, which is the Containerd CLI. It has been integrated with rootless Containerd and Nerdctl such that it can be utilized together more easily compared to Podman and Docker. Due to this integration we have switched from rootless Podman to rootless Containerd as noted previously. It should also be noted that Usernetes [44] also added support for bypass4netns.

When employing bypass4netns via Nerdctl, bypass4netns expects an annotation to be set via Nerdctl, which is checked by the bypass4netns daemon. When the daemon finds that annotation on a container, it enables accelerated networking via bypass4netns. In KSI 2, Kind is responsible for calling Nerdctl to deploy containers, however, Kind does not support passing through annotations. We therefore had to employ a workaround by wrapping the Nerdctl binary on the nodes with a script that injects the annotations.

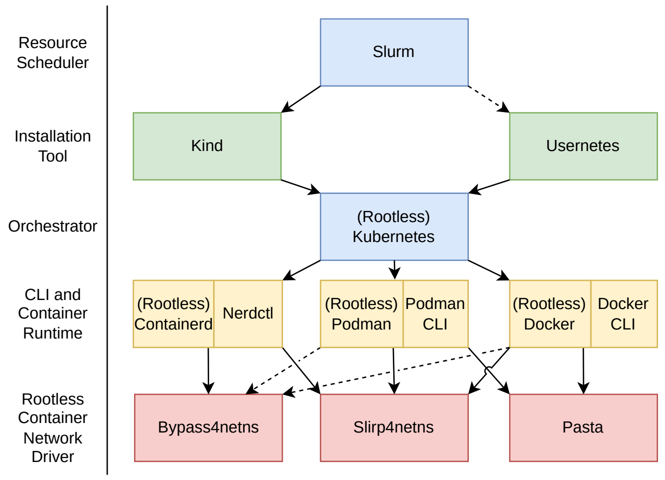

Figure 2. Overview of software options for setting up rootless Kubernetes on Slurm. From each row one option has to be used to assemble a viable software stack.

Figure 2 provides an overview of the components and alternatives in the software stack that can be employed for running rootless Kubernetes in Slurm jobs. Slurm is shown at the top as for the **Over** model the entire deployment is encapsulated in Slurm jobs. To deploy rootless Kubernetes, KSI employs Kind but we mentioned alternatives that also support rootless Kubernetes such as Usernetes, as shown in the figure, as well as K3D. In both cases, the goal of the tool is to deploy a rootless Kubernetes cluster.

The next layer in Figure 2 is the container runtime where the options are Containerd, Podman and Docker as each of them supports rootless containers. For Podman and Docker both the container runtime as well as the CLI tool share the same name, for Containerd, the associated CLI tool is called Nerdctl. KSI 1 utilized Podman but as discussed above, we switched to Containerd and Nerdctl for KSI 2. To complete the software stack a rootless container network driver is required, here the default in many cases is slirp4netns with bypass4netns being an experimental alternative.

Kind itself is not designed for multi-node clusters across multiple physical machines or VMs. Therefore, KSI 1 [1] was not able to achieve **RQ3**. However, in [1] we noted that a tool such as Kilo [51] or Liqo [17] could be used to enable mutli-node support. We have since then added Liqo to KSI 2 and enabled multi-node clusters such that **RQ3** can be satisfied.

Liqo operates by aggregating multiple Kubernetes clusters into a single cluster by representing each cluster as a virtual Kubelet in the main cluster. With this KSI 2 can be deployed across a number of nodes in a multi-node Slurm job and all the worker nodes each deploy a single node cluster. Then Liqo is installed and configured such that all single node clusters register themselves as virtual Kubelets on the main node. Liqo deploys a gateway pod in each cluster, which in our case means on each node that enables the clusters to automatically peer between each other. This allows for pod-to-pod communication between any of the nodes. Once the deployment and registration of all nodes through the main node has been completed, the main node is also no longer a single-point-of-failure for the cluster.

However, it should be noted that with this setup, every worker node runs a Kubernetes control plane instead of only Kubelet, as would be the case in a regular Kubernetes cluster. This adds overhead in terms of CPU and memory consumption on every node for running the Kubernetes control plane components.

The interface to run a Kubernetes workload via KSI is to submit it via Slurm, for example via `srun` in the following form:

```
srun -NX /bin/bash run-workload.sh
example-workload.sh /path/to/shared/folder
```

`X` in `-NX` specifies the number of nodes that should be requested from Slurm and used by the cluster. `run-workload.sh` is the main KSI script, which handles the provisioning of the rootless Kubernetes clusters via Kind and connecting them via Liqo. In order for Liqo to set up the multi-node cluster, it requires a shared folder across all nodes, for example an NFS share, which should be specified in place of `/path/to/shared/folder`. Finally, the actual workload should be specified in place of `example-workload.sh`. The workload script should use `kubectl` to create all components in Kubernetes that are needed and then wait for the workload to finish. Once the workload script returns, `run-workload.sh` will close the Kubernetes cluster and perform cleanup before stopping, which also ends the Slurm job.

The code for the KSI 1 implementation along with its documentation was released under the GPL-3.0 license on Github [25]. The code for KSI 2 with Liqo and bypass4netns integration was released under the same license on Github [9].

C. Performance Evaluation

To assess the performance overhead to answer **RQ4** we have broken down our benchmarking into the following factors:

- Startup time: Measured with a dummy workload
- CPU compute performance: Measured with Sysbench [52]
- Memory throughput: Measured with Stream [53]
- Storage throughput: Measured with Fio [54]
- Network latency: Measured with Netperf [55]
- Network bandwidth: Measured with iPerf3 [56]

We consider these as representative factors for user workloads that might be run through any of the tools under study.

We performed two rounds of benchmarks. First, the comparative study of the combinations of Kubernetes and Slurm following the **Adjacent** model against our KSI 1 implementation. Second, the comparison of network drivers for the improved implementation of KSI 2. In both cases we measured bare metal baselines by running the benchmarks without Kubernetes.

The first set of benchmarks were run on two machines with hardware specifications as shown in Table I.

TABLE I. HARDWARE SPECIFICATIONS OF THE MACHINES IN THE FIRST SET OF BENCHMARKS.

CPU	Intel(R) Xeon(R) CPU E5-2695 v3
CPU Sockets	2
Cores per socket	14
Threads per core	2
Total threads	56
RAM	24 DIMMs DDR4 16 GB 1866 MHz
Total RAM	384.00 GB
Storage	1 Verbatim Vi550 S3 SATA Revision 3.2 SSD
Total storage	128.00 GB
Network interface	QLogic BRCM 10G/GbE 2+2P 57800-t rNDC

On these nodes we used software versions as shown in Table II. The Kubernetes version v1.27.3 was the most recent version at the time of the experiments and was used for the external cluster for the Bridge Operator as well as by KSI 1. HPK, however, is pinned to v1.25.0 in its code base. Furthermore, we disabled SELinux, swap and write caching to measure the respective factors more clearly.

TABLE II. SOFTWARE VERSIONS OF THE MACHINES IN THE FIRST SET OF BENCHMARKS.

Linux OS	CentOS Stream 9
Slurm	23.02.5
Podman	4.6.1
slirp4netns	1.2.2-1
Kind	0.20.0
Kubectl	v1.28.2
Kubernetes	v1.27.3
HPK Kubernetes	v1.25.0
shadow-utils	2:4.9-8

For the second benchmark to validate the improved network performance of bypass4netns for KSI 2 compared to other network drivers, we used two machines with hardware specifications as shown in Table III.

TABLE III. HARDWARE SPECIFICATIONS OF THE MACHINES IN THE SECOND SET OF BENCHMARKS.

CPU	Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz
CPU Sockets	2
Cores per socket	20
Threads per core	2
Total threads	80
RAM	DIMM DDR4 Synchronous 2666 MHz
Total RAM	192 GB
Storage	Served from RAM due to Warewulf
Total storage	94 GB
Network interface	Intel 82599ES 10-Gigabit SFI/SFP+

On these we used software versions as listed in Table IV. Most significantly we switched to Rocky Linux from CentOS, replaced Podman with Nerdctl and Containerd and added bypass4netns. Also we used newer versions of Kind and Kubernetes.

TABLE IV. SOFTWARE VERSIONS OF THE MACHINES IN THE SECOND SET OF BENCHMARKS.

Linux OS	Rocky Linux 9 (Kernel 5.14)
Slurm	23.02.5
Nerdctl	v2.1.3
Containerd	v1.7.27
RootlessKit	v2.3.5
slirp4netns	v1.3.2
bypass4netns	v0.4.2
Kind	0.29.0
Kubectl	Client Version: v1.33.2
Kubernetes	Server Version: v1.33.2
Libseccomp	2.5.2
Liqoctl	v1.0.1

D. *Project State Evaluation*

Evaluating the maintainability and usability of software has been studied extensively [57][58] with many tools and methods having been proposed. For this work, in order to answer **RQ5** and **RQ6**, we have to consider what methods to employ.

In order to grade usability we consider the state of the available documentation as well as the difficulty of setting up and operating the respective tools for an assumed non-expert user based on our own experience of working with the tools during this study. For grading maintainability we reviewed the state of the code repositories based on their complexity, whether they have been kept up-to-date and how well issues had been addressed. Moreover, we consider that a code base that does a comparatively simple job while relying on more well maintained dependencies is itself more maintainable than a larger code that has more moving parts that may require maintenance.

We acknowledge that more sophisticated methods are available but consider our approach sufficient to compare the three projects under study on a three point grading schema.

IV. RESULTS

The commit hashes of the implementation versions used in our tests are as follows:

- Bridge Operator: `56334fa57caf2de28df6ff76df8a6e6232021421`
- HPK: `a902acbf2436e8a85a4620fddfa5745523f443d4`
- KSI 1: `780ef3a0562ad4bb12611f9ef43fa743fe0277d0` [25]
- KSI 2: `ea0457a97d056d32947d6e99538aac6e174e9213` [9]

A. *Functional Requirements*

For Bridge Operator and HPK the answer to **RQ1**, **RQ2** and **RQ3** is yes. For KSI 1 as well except for **RQ3**. This is solved in KSI 2 via the integration with Liqo.

B. *Project State*

a) Bridge Operator: When submitting a workload through Bridge Operator, it requires the user to create an instance of the CRD BridgeJob. With that no understanding of Slurm by the user is required. However, the available documentation for Bridge Operator is limited with some examples not working such that a patch was required to make it work [41]. While the project depends only on the Slurm REST API, giving it a stable foundation, the project itself seems abandoned with no activity after late 2022. Due to this, we rate it ○ in both usability and maintainability. We would have rated + for usability if the examples were all functional and for maintainability if the project was actively being maintained.

b) HPK: Similar to Bridge Operator, HPK can be controlled directly through `kubectl` without additional understanding of Slurm by the user. Nevertheless, while its documentation is also limited, after we had completed our experiments [35] was released along with `v0.1.2` of HPK containing a number of bug fixes. This shows that the project is being actively developed, moreover, when we ran into issues, we quickly received community support from the maintainers. With this we rate the maintainability as + and the usability as ○ because in addition to the points mentioned above, HPK does not support certain Kubernetes features, most notably `kubectl exec` and services, such that users need to work around these limitations.

c) KSI: Unlike the other two tools, KSI is started via Slurm as it has no active component outside of the Slurm job. Its usage is documented with several examples and it depends on Kind and either Podman for the original implementation and Containerd and Nerdctl for the improved version. Kind, Podman, Containerd and Nerdctl are well maintained projects. However, KSI 2 also depends on experimental features of Nerdctl as well as bypass4netns, which is also not in a stable state yet. Nevertheless, KSI delivers a feature complete Kubernetes cluster via a set of scripts making it both usable and maintainable so we rate KSI + for both factors. However, as KSI is our own creation, we cannot claim that this evaluation is unbiased and should be regarded as such.

C. Performance

The benchmarking scripts, as well as the raw test data, are available on Github [59]. Each benchmark was repeated 10 times to minimize random error with the standard deviation shown in the graphs. All benchmarks, except for those covered in Section IV-C6, were done as part of the first set of benchmarks with the implementation of KSI 1. Section IV-C6 covers the investigation of the network performance of KSI 2 with bypass4netns compared to slirp4netns, which KSI 1 used, and forms the second set of benchmarks.

TABLE V. WORKLOAD STARTUP TIME, LOWER IS BETTER.

Integration Approach	Startup Time in s
Bare Metal	0.141
Bridge Operator	2.725
HPK	2.497
KSI 1	53.921

1) Startup Time: The startup delays given in Table V have negligible standard deviation and show that Bridge Operator and HPK start a workload in 2 to 3 seconds while KSI 1 requires almost one minute. This result is as expected since Bridge Operator and HPK already have an active Kubernetes cluster running before they submit their Slurm job while KSI has to set up a Kubernetes cluster from scratch. Considering that HPC workloads often run for multiple hours, one minute extra start up time is not great but acceptable. Due to this we rate Bridge Operator and HPK with + and KSI with ○.

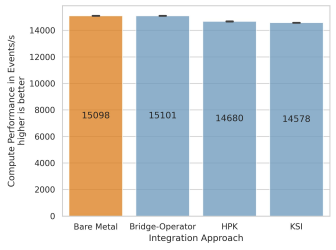

Figure 3. CPU compute performance results using Sysbench. Data was collected with KSI 1.

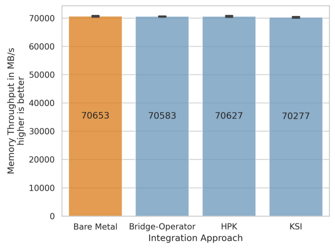

Figure 4. Memory throughput results using Stream. Data was collected with KSI 1.

2) Compute Performance: Figure 3 shows that Bridge Operator and bare metal are effectively on the same performance level while HPK and KSI 1 are slightly lower than bare metal (2.7% and 3.4%, respectively). The difference arises due to the virtualization overhead and additional active components running for HPK and KSI but is overall negligible so we rate all approaches with +.

3) Memory Performance: Figure 4 shows similar to Figure 3 only minor differences for HPK and KSI 1 due to the additional active components and virtualization such that we also rate all with + here.

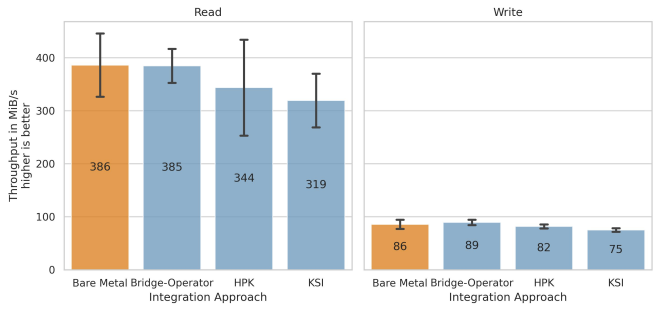

Figure 5. Storage throughput results using Fio and sequential operations. Data was collected with KSI 1.

4) Storage Performance: The sequential read and write shown in Figure 5 shows a similar pattern as the random read and write shown in Figure 6 with Bridge Operator being on the same level as bare metal and HPK and KSI 1 lacking behind. More specifically HPK is about 11% slower in sequential and 5% slower in random reading and KSI 1 is overall 17% slower in reading and 13% slower in writing than bare metal. These differences can also be attributed to the additional virtualization and overall resource consumption. While 17% slower reading is not good we rate it still as acceptable so HPK and KSI are rated as ○ and Bridge Operator as +.

5) Network Performance: For these benchmarks, the respective tool executed a workload containing a test client that

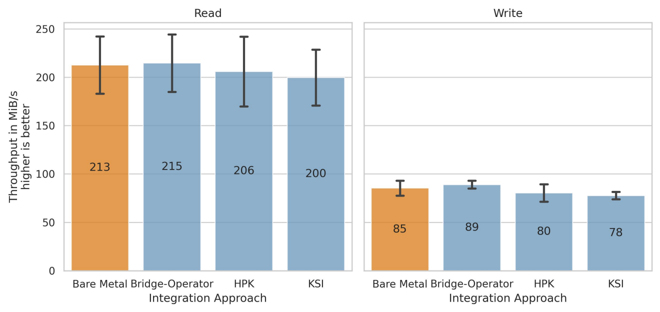

Figure 6. Storage throughput results using Fio and randomized operations. Data was collected with KSI 1.

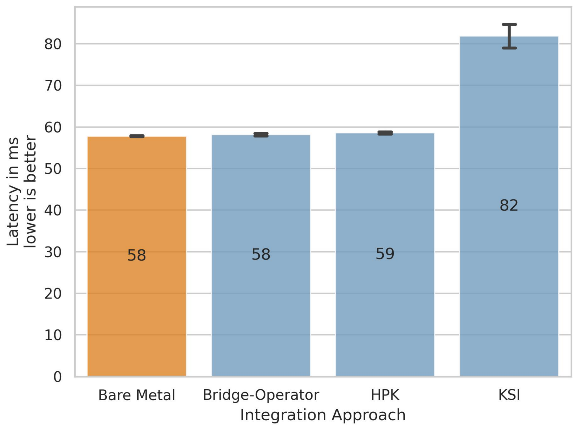

Figure 7. Network latency results using Netperf for the Kubernetes-Slurm integration solutions under study. Data was collected with KSI 1, which used slirp4netns.

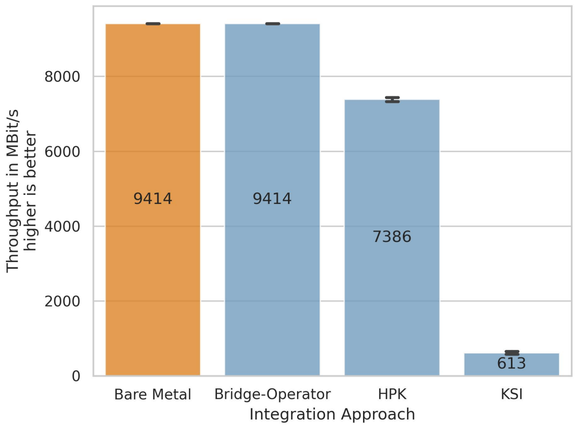

Figure 8. Network bandwidth results using iPerf3 for the Kubernetes-Slurm integration solutions under study. Data was collected with KSI 1, which used slirp4netns.

executed the network benchmark against a server running on the other of the two nodes in our test setup. What is shown as the bare metal latency and throughput are therefore the latency and peak throughput between the two nodes. Figure 7 shows network latency with all solutions on the same level except for KSI 1, which is 42% slower than bare metal. In Figure 8 the network throughput is even worse for KSI 1 with HPK already being 21% slower than bare metal, KSI 1 is 93.5% slower. As KSI 1 operates via rootlesss Podman, it uses slirp4netns as its driver, which according to the Podman documentation [50] results in degraded performance compared to rootful Podman networking.

Since our experiments concluded, pasta had replaced slirp4netns as the default network driver for rootless Podman, which promises better performance but initial tests could not show a significant overall improvement [60]. Our ratings are + for Bridge Operator, ○ for HPK and − for KSI 1.

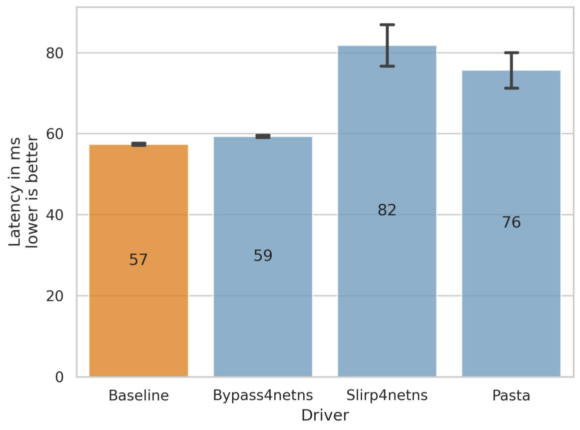

Figure 9. Network latency results using Netperf for KSI 2 using various rootless container network drivers. All variants were tested with Nerdctl except for pasta, which was tested with Podman 5.x.

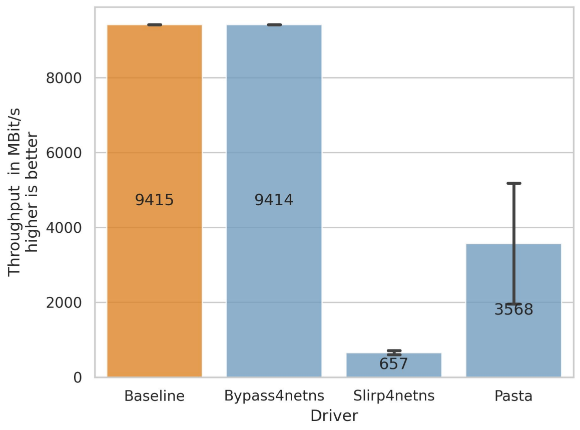

Figure 10. Network bandwidth results using iPerf3 for KSI 2 using various rootless container network drivers. All variants were tested with Nerdctl except for Pasta, which was tested with Podman 5.x.

6) Network Performance with Bypass4netns: In order to evaluate whether adding bypass4netns to KSI 2 improved the network performance, we compared its performance to slirp4netns and no containerization as a baseline. For all of these we used Nerdctl with rootless Containerd. Moreover, we also investigated pasta with Podman as pasta had replaced slirp4netns in Podman 5.0 in order to improve network performance of rootless containers. These measurements were conducted in the second set of benchmarks utilizing the hardware and software as listed in Tables III and IV.

The network latency shown in Figure 9 closely mirrors the results from the first round of benchmarks in Figure 7. Most notably, bypass4netns achieves effectively identical latency to the baseline. Pasta on the other hand shows slight improvements over slirp4netns but does not provide a silver bullet solution.

For network throughput Figure 10 also closely mirrors Figure 8 from the first round of benchmarks. Bypass4netns has also in this area caught up to the baseline, which shows that by employing bypass4netns, the network limitations of the KSI 1 implementation can be overcome.

However, as the hardware used in our experiments was limited to 10 Gbps, it is possible that bypass4netns would not be able to keep up with the baseline on more powerful network hardware. Nevertheless, based on the benchmarks done by the authors of bypass4netns [16], we can estimate that bypass4netns will be close to the baseline within a few percentage points even when using more powerful hardware. Our rating for the network performance of KSI 2 is +.

D. Evaluation

TABLE VI. Project assessment regarding quality requirements. *KSI 1* represents the original KSI implementation and *KSI 2* the improved KSI implementation with Liqo and bypass4netns. * Self-evaluation of KSI is not unbiased.

Project	RQ4 Startup	RQ4 Comp.	RQ4 Storage	RQ4 Net.	RQ5 Usab.	RQ6 Maintainab.
B-O	+	+	+	+	○	○
HPK	+	+	○	○	○	+
KSI 1	○	+	○	−	+*	+*
KSI 2	○	+	○	+	+*	+*

Table VI summarizes the ratings we have assigned throughout this section with Startup, Compute, Storage and Network performance all aiming at **RQ4** and Usability and Maintainability aiming at **RQ5** and **RQ6**, respectively.

Bridge Operator has shown performance close or identical to bare metal, which is as expected since it effectively submits a Slurm job through Kubernetes and does not start additional software in that Slurm job. This brings some limitations as it is not actually running a given workload using Kubernetes. Nevertheless, it has presented itself as a valid approach for extending a Kubernetes cluster via access to a Slurm cluster.

HPK provides a good middle ground for running Kubernetes jobs on Slurm with some performance deficiencies compared to bare metal. If WLM-Operator would be functional, we would have probably seen similar performance to HPK as WLM-Operator is based on Singularity and HPK is based on Apptainer and both projects still share the majority of their implementation. While HPK does not support all Kubernetes features, e.g., services and `kubectl exec` are not supported, it provides a solid choice for natively running Kubernetes workloads through Slurm.

KSI is functionally the most complete Kubernetes environment within a Slurm job and requires no external parts to be started and kept running outside of it. For the implementation of KSI 1 this comes with performance costs, as KSI 1 shows the weakest performance in all benchmarks, especially in startup time and networking. The slow startup time is understandable as KSI 1 has to bootstrap the Kubernetes control plane and cannot rely on an existing Kubernetes cluster. For network performance, KSI 1 relies on slirp4netns, which is known for causing performance degradation [50].

The KSI 2 implementation utilizes bypass4netns, which overcame the limited network performance of the original implementation. This puts the KSI 2 implementation on the same level as Bridge Operator as both were effectively on the same level as the baseline or bare metal measurements.

Nevertheless, in order for our KSI 2 implementation to achieve these performance levels, bypass4netns must be set up on each node. Bypass4netns requires access to features only available on Linux kernel version 5.9 and newer, which might not be available in all HPC environments. For instance, only RHEL-like 9 and newer operating systems, such as Rocky Linux 9, include a recent enough kernel version for bypass4netns. Moreover, security consideration as mentioned by the authors of bypass4netns [16] apply.

All the projects suffer from being either only proof-of-concept implementations, not being maintained or not being properly documented such that none of them provide a production ready solution for running Kubernetes inside of Slurm jobs.

V. Conclusion and Future Work

In this paper, we analyzed the state of solutions for combining Slurm and Kubernetes with the goal to enable dynamic computation between either environment. We focused on a subset of the available solutions to support our use case of running Kubernetes workloads on an existing Slurm cluster.

For this purpose, we improved upon our own solution KSI 1, which was originally based on Kind and rootless Podman and was able to deploy a fully functional Kubernetes cluster inside a Slurm job. We improved KSI 1 by switching from rootless Podman to rootless Nerdctl and employing bypass4netns for container networking instead of Podman's default network driver slirp4netns. Moreover, we added Liqo to enable multi-node clusters for KSI 2.

From the available solutions we took a closer look at Bridge Operator, HPK and our own solution, KSI, and found that they fulfill our functional requirements, except for KSI 1, for which multi-node support had not been implemented yet. KSI 2, our improved implementation, supports multi-node clusters. We further evaluated the performance of each solution and reviewed the state of their respective implementations.

We found that Bridge Operator delivers effectively bare metal performance equal to directly running a job through Slurm as this is effectively what Bridge Operator does. HPK established itself as a middle ground solution, providing an almost fully functional Kubernetes cluster inside a Slurm job with minor performance overhead. Our solution, KSI 1, showed slightly higher overhead compared to HPK and significantly less network throughput in its original implementation. Via the improvements presented in this work, we were able to achieve the same level of network performance as Bridge Operator. KSI 2 provides the most feature complete Kubernetes clusters compared to the other solutions. However, to achieve its fast network throughput it relies on recent Linux kernel features that might not be available in every HPC center. Moreover, it requires carefully preparing the system environments on every node that should run KSI 2 by setting up the respective software stack.

We conclude that by overcoming the two significant shortcomings of KSI 1, it is now able to deliver both feature complete Kubernetes environments as well as excellent performance. The next steps for KSI 2 include improving its overall usability and stability as well as comparing it to other solutions such as Usernetes.

Acknowledgments

This work was supported by the Federal Ministry of Education and Research (BMBF), Germany under the AI service center KISSKI (grant no. 01IS22093A and 01IS22093B). Moreover, additional work was funded by the European Union under the Horizon Europe initiative via grant no. 101092582 for the DECICE project.

References

[1] J. Decker, S. Metje, and J. Kunkel, "Running Kubernetes Workloads on Rootless HPC Systems using Slurm", presented at the CLOUD COMPUTING 2025, The Sixteenth International Conference on Cloud Computing, GRIDs, and Virtualization, Apr. 6, 2025, pp. 100–107, ISBN: 978-1-68558-258-6.

[2] "CNCF Annual Survey 2023", CNCF, Apr. 9, 2024, [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2023/ (visited on 2024.12.30).

[3] "9 Insights on Real-World Container Use | Datadog", [Online]. Available: https://web.archive.org/web/20230318234844/https://www.datadoghq.com/container-report/ (visited on 2024.12.30).

[4] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management", in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., Berlin, Heidelberg: Springer, 2003, pp. 44–60, ISBN: 978-3-540-39727-4. DOI: 10.1007/10968987_3.

[5] "Volcano-sh/volcano", Volcano, Dec. 30, 2024, [Online]. Available: https://github.com/volcano-sh/volcano (visited on 2024.12.30).

[6] "Scaling Kubernetes to 7,500 Nodes", Jan. 2021, [Online]. Available: https://openai.com/blog/scaling-kubernetes-to-7500-nodes/ (visited on 2021.02.12).

[7] A. M. Beltre, P. Saha, M. Govindaraju, A. Younge, and R. E. Grant, "Enabling hpc workloads on cloud infrastructure using kubernetes container orchestration mechanisms", in *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, Nov. 2019, pp. 11–20. DOI: 10.1109/CANOPIE-HPC49598.2019.00007.

[8] T. Wickberg, "Slurm and/or/vs Kubernetes", Schedmd, [Online]. Available: https://slurm.schedmd.com/SC23/Slurm-and-or-vs-Kubernetes.pdf (visited on 2025.08.06).

[9] "Gwdg/pub-2025-ksi", Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen, Aug. 1, 2025, [Online]. Available: https://github.com/gwdg/pub-2025-ksi (visited on 2025.08.01).

[10] "Kind – Rootless", [Online]. Available: https://kind.sigs.k8s.io/docs/user/rootless/ (visited on 2024.12.30).

[11] B. Lublinsky, E. Jennings, and V. Spišaková, "A kubernetes 'bridge' operator between cloud and external resources", in *2023 8th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, 2023, pp. 263–269. DOI: 10.1109/ICCCBDA56900.2023.10154770.

[12] Staff, "Introducing HPC Affinities to the Enterprise: A New Open Source Project Integrates Singularity and Slurm via Kubernetes", Sylabs, May 7, 2019, [Online]. Available: https://sylabs.io/2019/05/introducing-hpc-affinities-to-the-enterprise-a-new-open-source-project-integrates-singularity-and-slurm-via-kubernetes/ (visited on 2024.12.30).

[13] K. Peterson, "Kalenpeterson/kube-slurm", Aug. 17, 2024, [Online]. Available: https://github.com/kalenpeterson/kube-slurm (visited on 2024.12.30).

[14] A. Chazapis, F. Nikolaidis, M. Marazakis, and A. Bilas, "Running kubernetes workloads on HPC", in *High Performance Computing*, A. Bienz, M. Weiland, M. Baboulin, and C. Kruse, Eds., ser. Lecture Notes in Computer Science, Cham: Springer Nature Switzerland, 2023, pp. 181–192, ISBN: 978-3-031-40843-4. DOI: 10.1007/978-3-031-40843-4_14.

[15] N. Matsumoto and A. Suda, "Accelerating TCP/IP Communications in Rootless Containers by Socket Switching", 2022.

[16] N. Matsumoto and A. Suda, "Bypass4netns: Accelerating TCP/IP Communications in Rootless Containers", Feb. 1, 2024. DOI: 10.48550/arXiv.2402.00365. arXiv: 2402.00365 [cs], pre-published.

[17] "Liqotech/liqo", LiqoTech, Jul. 22, 2025, [Online]. Available: https://github.com/liqotech/liqo (visited on 2025.07.24).

[18] S. Metje, *Running Kubernetes Workloads on Rootless HPC Systems using Slurm*, GRO.data, Jan. 9, 2024. DOI: 10.25625/GDFCFP.

[19] F. Liu, K. Keahey, P. Riteau, and J. Weissman, "Dynamically negotiating capacity between on-demand and batch clusters", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18, Dallas, Texas: IEEE Press, Nov. 11, 2018, pp. 1–11.

[20] B. Wu, M. Hu, S. Qin, and J. Jiang, "Research on fusion scheduling based on Slurm and Kubernetes", in *International Conference on Algorithms, High Performance Computing, and Artificial Intelligence (AHPCAI 2024)*, vol. 13403, SPIE, Nov. 18, 2024, pp. 476–485. DOI: 10.1117/12.3051639.

[21] G. Zervas, A. Chazapis, Y. Sfakianakis, C. Kozanitis, and A. Bilas, "Virtual clusters: Isolated, containerized HPC environments in kubernetes", in *High Performance Computing. ISC High Performance 2022 International Workshops*, H. Anzt, A. Bienz, P. Luszczek, and M. Baboulin, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2022, pp. 347–357, ISBN: 978-3-031-23220-6. DOI: 10.1007/978-3-031-23220-6_24.

[22] T. Menouer, N. Greneche, C. Cérin, and P. Darmon, "Towards an Optimized Containerization of HPC Job Schedulers Based on Namespaces", in *Network and Parallel Computing*, C. Cérin, D. Qian, J.-L. Gaudiot, G. Tan, and S. Zuckerman, Eds., Cham: Springer International Publishing, 2022, pp. 144–156, ISBN: 978-3-030-93571-9. DOI: 10.1007/978-3-030-93571-9_12.

[23] C. Cérin, N. Greneche, and T. Menouer, "Towards Pervasive Containerization of HPC Job Schedulers", in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Sep. 2020, pp. 281–288. DOI: 10.1109/SBAC-PAD49847.2020.00046.

[24] "SlinkyProject/slurm-operator", SlinkyProject, Dec. 26, 2024, [Online]. Available: https://github.com/SlinkyProject/slurm-operator (visited on 2024.12.30).

[25] S. Metje, "Kubernetes Slurm Integration based on Kind", 2023, [Online]. Available: https://github.com/soerenmetje/kind-slurm-integration.

[26] P. Liu and J. Guitart, *Fine-grained scheduling for containerized HPC workloads in kubernetes clusters*, Nov. 21, 2022. DOI: 10.48550/arXiv.2211.11487. arXiv: 2211.11487[cs].

[27] D. Medeiros, J. Wahlgren, G. Schieffer, and I. Peng, "Kub: Enabling elastic HPC workloads on containerized environments", in *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, ISSN: 2643-3001, Oct. 2023, pp. 219–229. DOI: 10.1109/SBAC-PAD59825.2023.00031.

[28] "PRIMAGE / hpc-connector · GitLab", GitLab, Feb. 22, 2023, [Online]. Available: https://gitlab.com/primageproject/hpc-connector (visited on 2025.01.02).

[29] N. Zhou *et al.*, "Container orchestration on HPC systems through Kubernetes", *Journal of Cloud Computing*, vol. 10, no. 1, p. 16, Feb. 22, 2021, ISSN: 2192-113X. DOI: 10.1186/s13677-021-00231-z.

[30] G. Staples, "TORQUE resource manager", in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06, New York, NY, USA: Association for Computing Machinery, Nov. 11, 2006, 8–es, ISBN: 978-0-7695-2700-0. DOI: 10.1145/1188455.1188464.

[31] "Sylabs/wlm-operator", Sylabs Inc., Nov. 5, 2024, [Online]. Available: https://github.com/sylabs/wlm-operator (visited on 2025.01.02).

[32] "Sylabs/singularity-cri", Sylabs Inc., Mar. 1, 2024, [Online]. Available: https://github.com/sylabs/singularity-cri (visited on 2025.01.02).

[33] B. Lublinsky, E. Jennings, and V. Spišaková, "A Kubernetes 'Bridge' operator between cloud and external resources", Jul. 6, 2022. DOI: 10.48550/arXiv.2207.02531. arXiv: 2207.02531 [cs], pre-published.

[34] "Bridge-Operator/kubeflow at main · IBM/Bridge-Operator", [Online]. Available: https://github.com/IBM/Bridge-Operator/tree/main/kubeflow (visited on 2025.01.02).

[35] "CARV-ICS-FORTH/HPK", Computer Architecture and VLSI Systems (CARV) Laboratory, Dec. 26, 2024, [Online]. Available: https://github.com/CARV-ICS-FORTH/HPK (visited on 2025.01.02).

[36] A. Chazapis, E. Maliaroudakis, F. Nikolaidis, M. Marazakis, and A. Bilas, "Running Cloud-native Workloads on HPC with High-Performance Kubernetes", Sep. 25, 2024. DOI: 10.48550/arXiv.2409.16919. arXiv: 2409.16919 [cs], pre-published.

[37] "Apptainer/apptainer", The Apptainer Container Project, Dec. 30, 2024, [Online]. Available: https://github.com/apptainer/apptainer (visited on 2025.01.02).

[38] "Virtual-kubelet/virtual-kubelet", virtual kubelet, Feb. 24, 2025, [Online]. Available: https://github.com/virtual-kubelet/virtual-kubelet (visited on 2025.02.24).

[39] "Flannel-io/flannel", flannel-io, Feb. 25, 2025, [Online]. Available: https://github.com/flannel-io/flannel (visited on 2025.02.25).

[40] "Flannel-io/cni-plugin", flannel-io, Jan. 31, 2025, [Online]. Available: https://github.com/flannel-io/cni-plugin (visited on 2025.02.25).

[41] "Fix #2 #3 #6 by soerenmetje · Pull Request #4 · IBM/Bridge-Operator", [Online]. Available: https://github.com/IBM/Bridge-Operator/pull/4 (visited on 2025.01.02).

[42] "Kubernetes/minikube", Kubernetes, Jan. 2, 2025, [Online]. Available: https://github.com/kubernetes/minikube (visited on 2025.01.02).

[43] "K3d-io/k3d", k3d, Jan. 2, 2025, [Online]. Available: https://github.com/k3d-io/k3d (visited on 2025.01.02).

[44] "Rootless-containers/usernetes", rootless-containers, Dec. 30, 2024, [Online]. Available: https://github.com/rootless-containers/usernetes (visited on 2025.01.02).

[45] "Kind", [Online]. Available: https://kind.sigs.k8s.io/ (visited on 2025.02.21).

[46] "Rootless-containers/rootlesskit", rootless-containers, Jul. 25, 2025, [Online]. Available: https://github.com/rootless-containers/rootlesskit (visited on 2025.07.25).

[47] "Rootless-containers/slirp4netns", rootless-containers, Feb. 23, 2025, [Online]. Available: https://github.com/rootless-containers/slirp4netns (visited on 2025.02.25).

[48] "Releases · containers/podman", GitHub, [Online]. Available: https://github.com/containers/podman/releases (visited on 2025.02.25).

[49] "Passt - Plug A Simple Socket Transport", [Online]. Available: https://passt.top/passt/about/ (visited on 2025.02.25).

[50] "Podman/docs/tutorials/rootless_tutorial.md at main · containers/podman", GitHub, [Online]. Available: https://github.com/containers/podman/blob/main/docs/tutorials/rootless_tutorial.md (visited on 2025.02.21).

[51] L. S. Marín, "Squat/kilo", Dec. 24, 2024, [Online]. Available: https://github.com/squat/kilo (visited on 2025.01.02).

[52] A. Kopytov, "Akopytov/sysbench", Jan. 2, 2025, [Online]. Available: https://github.com/akopytov/sysbench (visited on 2025.01.02).

[53] J. Hammond, "Jeffhammond/STREAM", Dec. 24, 2024, [Online]. Available: https://github.com/jeffhammond/STREAM (visited on 2025.01.02).

[54] J. Axboe, "Flexible I/O Tester", 2022, [Online]. Available: https://github.com/axboe/fio (visited on 2025.01.02).

[55] "HewlettPackard/netperf", Hewlett Packard Enterprise, Dec. 10, 2024, [Online]. Available: https://github.com/HewlettPackard/netperf (visited on 2025.01.02).

[56] "Esnet/iperf", ESnet: Energy Sciences Network, Jan. 2, 2025, [Online]. Available: https://github.com/esnet/iperf (visited on 2025.01.02).

[57] L. Ardito, R. Coppola, L. Barbato, and D. Verga, "A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review", *Scientific Programming*, vol. 2020, no. 1, p. 8 840 389, 2020, ISSN: 1875-919X. DOI: 10.1155/2020/8840389.

[58] K. A. Dawood *et al.*, "Towards a unified criteria model for usability evaluation in the context of open source software based on a fuzzy Delphi method", *Information and Software Technology*, vol. 130, p. 106 453, Feb. 1, 2021, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2020.106453.

[59] "Gwdg/pub-2025-ksi-evaluation", Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen, Aug. 1, 2025, [Online]. Available: https://github.com/gwdg/pub-2025-ksi-evaluation (visited on 2025.08.01).

[60] "Rootless network performance (pasta vs slirp4netns) · containers/podman · Discussion #22559", GitHub, [Online]. Available: https://github.com/containers/podman/discussions/22559 (visited on 2025.01.03).