A Comparative Analysis of Kubernetes and SLURM for Deploying Interactive Applications

Sanele Dlamini
University of Cape Town
South Africa

ABSTRACT

This research compares the deployment of interactive applications using Kubernetes as an orchestration framework and SLURM as a resource manager and scheduler, using the Cube Analysis and Rendering Tool for Astronomy (CARTA) as a case study. CARTA is a next-generation image visualization and analysis tool designed for ALMA, VLA, and SKA pathfinders. Deploying CARTA on Kubernetes presents challenges because backend processes must run under the end user's identity and access large shared file systems while complying with access control policies. Additionally, attaching shared file systems is less common in enterprise Kubernetes environments compared to HPC clusters. This study focuses on evaluating the deployment complexity and customization requirements of orchestration frameworks, such as Kubernetes and SLURM, as resource managers for HPC clusters. The analysis aims to identify how reusable and adaptable deployment configurations can be developed for each platform to support scalable, multi-user workflows.

1 INTRODUCTION

The Cube Analysis and Rendering Tool for Astronomy (CARTA) [1] is a remote visualization software designed to work with large radio astronomy image cubes produced by the latest generation of telescopes. CARTA employs a client-server architecture to visualize large image files, ranging from gigabytes to terabytes, commonly designed for ALMA, VLA, and SKA pathfinder observations. Due to their size, such datasets are impractical to process on standard personal computers or laptops. In this architecture, intensive computation and data storage tasks are offloaded to high-performance remote servers or clusters, while only the visualization-ready outputs are transmitted to the client. This enables users to leverage modern web technologies, including GPU-accelerated rendering, to enhance their online experiences. This setup also allows user interaction with ALMA and VLA science archives, utilizing CARTA as an interface.

The CARTA backend is currently deployed on High-performance computing (HPC) clusters. A resource manager and scheduler, such as SLURM [7], is used to manage applications on HPC clusters. To run jobs, users select from multiple preconfigured templates that describe how their application should run, specifying the number of nodes to use, the number of CPUs, memory, and GPUs on each node, and cluster wide scheduling rules and policies (e.g, fair share, priority) that apply to all jobs. An alternative approach would be to deploy interactive applications using orchestration frameworks, such as Kubernetes (K8s) [5]. In Kubernetes, you define desired states using YAML files. It dynamically discovers and manages resources, providing ease of updates and reconfiguration. Kubernetes (K8s) is the most widely used container orchestration framework, providing built-in mechanisms that enhance system fault tolerance [2]. Fault tolerance refers to a system's ability to detect failures and continue operating without disruption, thereby protecting users and data. While SLURM includes mechanisms to enhance fault tolerance in HPC environments, this work compares the deployment of CARTA using Kubernetes as an orchestration framework and SLURM as a resource manager and scheduler, with a focus on their respective deployment complexities.

In the following two subsections, we elaborate on using containers, which allow packaging the dependencies for individual executables into portable images. While container orchestration frameworks such as Kubernetes require containers, in HPC environments, applications like CARTA may run without containerization or with tools like Singularity [12] to package specific components. We then discuss persistent storage solutions capable of handling the large image files that are visualised using CARTA.

1.1 Containerization

Containerization is a method of packaging software applications and their dependencies into lightweight, portable containers running reliably across different computing environments. In this work, we use two container engines: Docker [9] and Singularity. Docker is the most widely adopted container engine, with the broader ecosystem standardizing around containerd and alternatives like CoreOS rkt. Both Docker and Singularity can be used in conjunction with orchestration frameworks, such as Kubernetes. However, Docker is designed for cloud-native workloads and requires root privileges to build and run containers, posing security risks in multi-user HPC environments [10]. In contrast, HPC systems typically adopt Singularity, which allows unprivileged execution and integrates more securely with resource managers [13]. The following subsection discusses the persistent storage used in both deployment approaches.

1.2 Permanent Storage with a Distributed File System

This subsection discusses a large, shared, and permanent file system that stores user data and is accessible across all nodes. The file system stores the large image cubes that CARTA reads for visualization and analysis. To manage user data reliably, we utilize CephFS [11], a distributed network file system designed for petabyte-scale storage. CephFS supports data sharing across all cluster nodes and individual containers within those nodes. While this setup is standard in HPC and research environments, attaching a shared file system is less prevalent in enterprise Kubernetes deployments. CephFS enables large cubes to be accessed across the entire cluster. In the following subsection, we present the problem

statement that we aim to address by utilizing Kubernetes as an alternative to SLURM-managed clusters for deploying CARTA.

1.3 Problem Statement

Deploying interactive applications is relatively straightforward when the number of users and memory requirements are low, as demonstrated by using CARTA as a case study. However, deployment becomes significantly more challenging when there is a need to serve such applications on large, scalable systems, such as high-performance computing (HPC) clusters or self-hosted cloud environments. In these settings, it is necessary to run backend processes under the end user's identity and to access large, shared file systems while adhering to strict access control policies. These challenges manifest differently across deployment methods, such as orchestration frameworks like Kubernetes and resource managers like SLURM, in the management of HPC clusters. Kubernetes, while widely used in cloud-native environments, requires substantial customization to support persistent storage, user authentication, and multi-user workflows. Conversely, SLURM, commonly used in HPC environments, provides native support for user-based job execution but requires manual integration of services, data paths, and reverse proxy configurations.

The demand for robust deployment strategies increases in research centers, where groups of researchers need to analyze large volumes of data, requiring substantial computing power. In this work, we compare two deployment approaches for deploying CARTA: one using Kubernetes as an orchestration framework and the other using SLURM as a resource manager within HPC clusters. The focus will be on comparing the deployment complexities of interactive applications by evaluating Kubernetes and SLURM. The findings will guide system administrators and infrastructure designers in selecting the most appropriate framework based on deployment complexity rather than raw performance. The following subsection outlines the research questions that will inform and direct this study.

1.4 Research Questions

The following research questions will inform the methodological approach of this study.

- How can interactive applications be effectively deployed on a Kubernetes cluster compared to applications managed by SLURM?
- How generic and reusable are Kubernetes and SLURM deployment approaches, and what level of customization is required to transform a baseline installation into a production-ready environment?
- Can Kubernetes (K8s) provide greater fault tolerance and resilience than when SLURM manages applications?
- How do deployment practices and configuration workflows influence operational resilience and the ease of maintaining service availability during failures?

The rest of the proposal is structured as follows. Section 2 provides a Literature review, Section 3 outlines the methodology we will use to determine the best deployment and metrics, and Section 4 presents our timeline and conclusion.

2 LITERATURE REVIEW

This section will cover the literature review on application deployment, especially interactive applications.

Zonca and Sinkovitch [14] explore various approaches to deploying interactive applications in High-Performance Computing (HPC) environments, with a focus on containerization and resource management. Several studies have compared Docker and Singularity, emphasizing Singularity's security advantages in HPC environments where Docker's root privileges pose security risks. Researchers have demonstrated JupyterHub deployment on HPC using SLURM and Singularity, highlighting the need for seamless interactive computing and GPU utilization in AI and scientific workloads. Other studies have explored adaptive containerization architectures to enhance deployment efficiency in high-performance computing (HPC) environments. Additionally, research has shown that Kubernetes can be leveraged for interactive application deployment, offering scalability and fault tolerance compared to resource managers like SLURM. Kubernetes dynamically allocates resources, unlike SLURM, which requires job scripts with predefined configurations. Studies have also examined the efficiency of batch schedulers, such as SLURM, in scaling machine learning frameworks, including TensorFlow and MATLAB/Octave, to tens of thousands of cores. These findings highlight the strengths and limitations of Kubernetes and SLURM in managing large-scale workloads, providing a basis for evaluating their effectiveness in deploying CARTA and similar interactive applications.

Kumar and Kaur [3] studied container-based Jupyter Lab and an AI framework on HPC with GPU usage. They used SLURM as their workload manager to run Singularity on HPC with GPU usage. Singularity addresses the weaknesses of Docker, which lack root access, security-related issues, and native support for HPC workload managers, making it unsuitable for secure HPC systems [4]. They transformed the Docker image to a Singularity image using the Singularity pull and build commands. To configure the container environment to support NVIDIA GPU utilization for running containerized applications migrated from Docker to Singularity on an HPC system, they used the' singularity run' or' exec' command with the' –nv' option. They assessed JupyterLab and an AI framework for their performance evaluation, transitioning them from Docker to Singularity to address increasingly intricate real-world challenges by utilizing GPU capabilities within an HPC environment alongside native solutions.

In [9], Shah and Dubaria presented the deployment of WordPress on the Google Cloud Platform using a Docker container and Kubernetes as their orchestration framework. The application was scalable, fault-tolerant, and efficiently managed the Docker containers.

Reuther et al. [8] presented how they used SLURM to launch 32,000 TensorFlow processes in 4 seconds and 262,000 Octave processes in 40 seconds in the MIT Lincoln Laboratory Supercomputing Center (LLSC). Their task was to scale interactive machine learning frameworks, such as TensorFlow, and data analysis environments, like MATLAB and Octave, to tens of thousands of cores. They chose SLURM because it can handle synchronously parallel jobs and job arrays, scaling up to manage over 100,000 jobs. The results of their experiment were promising, as many researchers at MIT

often utilize MATLAB and Octave for rapid prototyping, algorithm development, and data analysis through the use of SLURM.

Müller et al. [6] proposed an adaptive containerization approach to accelerate the deployment of applications and workflows on HPC systems using containers. To achieve this, they examine the specific HPC requirements for container tools and conduct an in-depth analysis of the containerization stack, including container engines and registries.

This work was inspired by the deployment of JupyterHub, where a single notebook server process is started for each user to support many students working on scientific collaborations. Similarly, CARTA operates in a way that initiates a dedicated process for each user. This work focuses on scaling CARTA to support large collaborative teams, addressing the challenges of resource allocation and efficient deployment to meet the demands of such large-scale usage.

This work makes significant contributions to various aspects of existing literature. We present the deployment of an interactive application using Kubernetes on ilifu and deploying the latter on HPC using SLURM as our resource manager with CARTA as our example. The following section will discuss the methodology of the two deployments.

3 METHODOLOGY

This section outlines a detailed methodology for comparing how CARTA can be deployed using Kubernetes and SLURM to run on cloud-based Kubernetes (K8s) and SLURM-based high-performance computing (HPC) systems. This section will have three parts: the setup of the Kubernetes and SLURM testbeds, followed by the configuration of CARTA in both the Kubernetes cluster and the SLURM cluster. We will then present the architecture of how the components are structured in each deployment. We will also measure some metrics of both deployments to compare their deployment complexities.

3.1 Testbed Setup

The testbed setup includes both Kubernetes and SLURM clusters. We first describe the Kubernetes cluster configuration and the SLURM cluster setup. On OpenStack, we create a Virtual machine (VM) with 25 GB disk space, running Ubuntu 22.04. We will install Minikube to create a Kubernetes cluster along with kubectl to interact with the cluster.

For the SLURM setup testbed on OpenStack, we will create four virtual machines (VMs). We will install Ubuntu 22.04 on each VM and install MariaDB, which is required to support SLURM accounting and store records of job and resource usage. One VM/node will be used as a login node, and three additional VMs/nodes will be used as compute nodes. We install Munge for authentication on all four nodes and ensure all nodes are configured within the same subnet. The login node will have the **slurmctld** daemon, which manages the cluster. The **slurmctld** and **slurmd** daemons communicate over TCP, where the munge is used for security and authentication. Then, the other three nodes will be compute nodes on which we will install **slurmd** to execute jobs. We will then mount the CephFS file system on all nodes. In the following subsection, we outline the configurations of CARTA for both Kubernetes clusters and SLURM clusters.

3.2 Configuration of CARTA on Kubernetes

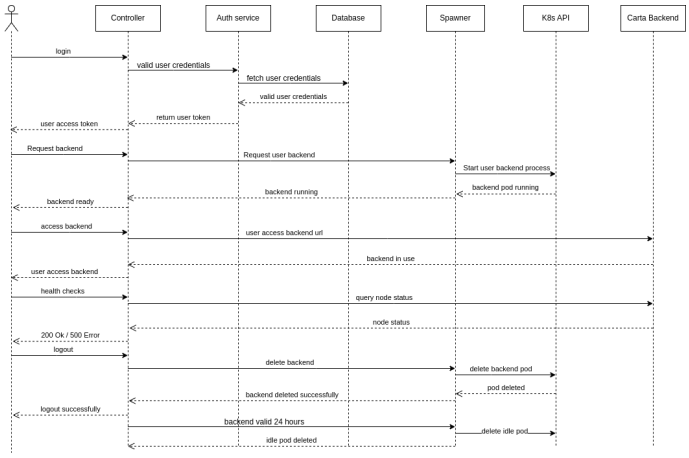

Figure 1: Kubernetes Based System

This section describes how CARTA is deployed on Kubernetes using the Kubernetes/client-node library. This library allows programmatic interaction with the Kubernetes API from Node.js or TypeScript applications. This deployment model follows the approach used by systems such as JupyterHub [14] with KubeSpawner, where a dedicated backend pod is provisioned for each user upon request.

The process begins when a user logs in through the Controller, which forwards authentication requests to the Auth Service. The Auth Service validates the credentials against external identity providers such as PAM, LDAP, or OIDC, and returns an access token upon successful authentication. The user can then request the initiation of a CARTA backend instance. The Controller forwards this request to the Spawner, which manages the lifecycle of the backend pods [9]. The Spawner first queries the Kubernetes API to determine whether a backend pod for the user already exists. If no pod is running, the Spawner instructs the Kubernetes API to create a new pod. Once the pod reaches the running state, the Spawner notifies the Controller, which updates its internal state and provides the user with the backend access URL. The user connects directly to the CARTA backend via this URL.

During backend usage, the Controller periodically performs health checks by querying the Kubernetes API for the status of the pod and node. Based on the health status received, the controller may return successful or failed responses to the user. When the user logs out, the Controller instructs the Spawner to delete the backend pod, which the Kubernetes API removes. The Spawner confirms the successful deletion, and the Controller updates its internal state accordingly.

Additionally, the system includes an idle timeout policy, which automatically deletes inactive backend pods after 24 hours to conserve cluster resources. This behavior is explicitly configured within the deployment system and is not a default Kubernetes feature.

3.3 Configuration of CARTA on HPC using SLURM as a resource manager

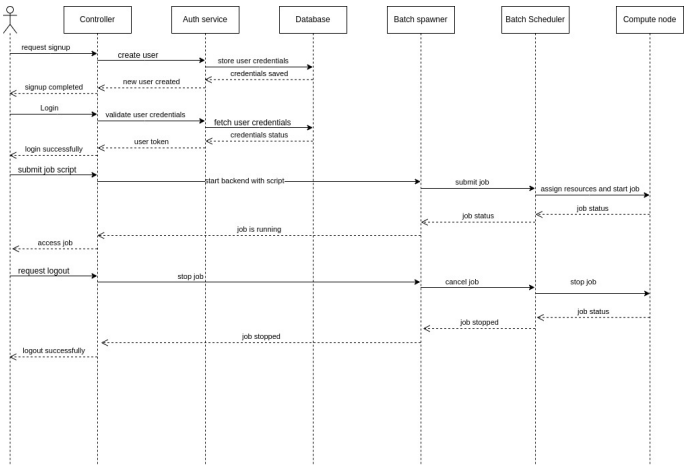

Figure 2: Using a Batch Scheduler

This section describes how CARTA is deployed using SLURM as a resource manager. Figure 2 shows the architecture, where each user interacts with CARTA through a personal backend process scheduled and managed by SLURM on an HPC cluster.

The process begins when the user logs into the Controller, which forwards authentication requests to the Auth Service. The Auth Service validates the provided credentials against external identity providers such as LDAP, OIDC, or PAM, and returns an access token upon successful authentication. After a successful login, the Controller invokes the start() function on the BatchSpawner, which manages the lifecycle of user backends on the cluster.

Upon receiving the start request, the Batch-spawner internally generates a SLURM job submission script using pre-configured templates and resource specifications. This submission script defines job parameters, including resource requests (CPU, memory, GPUs), execution time, and the CARTA backend launch command. Batch Spawner then submits the job to the SLURM Batch Scheduler using the SLURM `sbatch` command or an equivalent submission interface.

The Batch Scheduler processes the job request, assigns the required resources, and launches the job on an available compute node. Batch Spawner monitors the job status by periodically polling the scheduler. Once the job runs, Batch Spawner reports its active status to the Controller, allowing the user to connect to the running CARTA backend and start their session.

When the user initiates a logout, the Controller calls the stop() function on the Batch Spawner, which sends a cancellation request to the Batch Scheduler. The Batch Scheduler cancels the running job, releases the allocated resources, and notifies the Batch Spawner of the job termination. Upon confirmation that the job has stopped, the Controller updates its state and logs the user out successfully. The following section will discuss the metrics we use to compare the two deployment systems. The metrics will be on the deployment complexities of Kubernetes and SLURM.

3.4 Slurm Architecture

The diagram below shows a cluster managed by SLURM. The user (Actor) interacts through their web browser to access the frontend, which connects to the CARTA controller. Controller queries the database (session/state info). The controller schedules backend tasks via SLURM (slurmd). SLURM starts the CARTA backend on a compute node. The backend accesses the shared file system to load data. The frontend provides a user interface to view the image cubes from the backend.

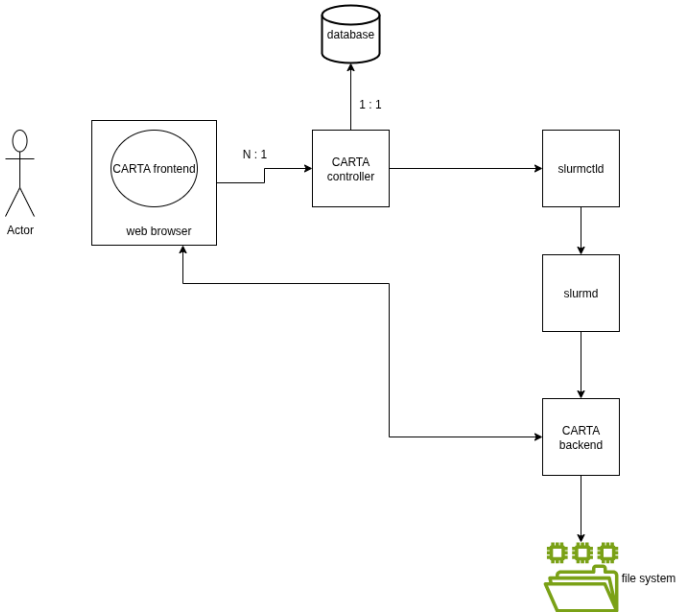

Figure 3: SLURM ARCHITECTURE

3.5 Kubernetes Cluster architecture

This diagram illustrates the architecture of how CARTA will be deployed on a Kubernetes cluster. The CARTA controller pods can potentially be deployed on low-computational nodes (Node 1), while the CARTA backend pods run on high-computational nodes (Node 2) to handle intensive processing tasks. The controller and backend pods communicate via Kubernetes services, which expose an application running in the cluster behind a single outward-facing endpoint, even when the workload is split across multiple backends. The control plane manages the cluster operations. Its components are the **kube-API-server**, which validates and configures data for API objects, including pods, services, replication controllers, and others, and the **etcd** stores all cluster configuration and state data. The **scheduler** is a control plane process that assigns Pods to Nodes. The **controller manager** is a daemon that embeds the core control loops shipped with Kubernetes. The **cloud control manager** enables you to link your cluster to your cloud provider's API, separating the components that interact with the cloud platform from those that only interact with your cluster.

On each node, the **kubelet** runs containers defined in pods, and the kube-proxy manages networking to route traffic between

pods and services. This architecture enables CARTA's frontend and backend to be flexibly distributed across nodes, scaling resources while maintaining consistent communication and access to shared storage.

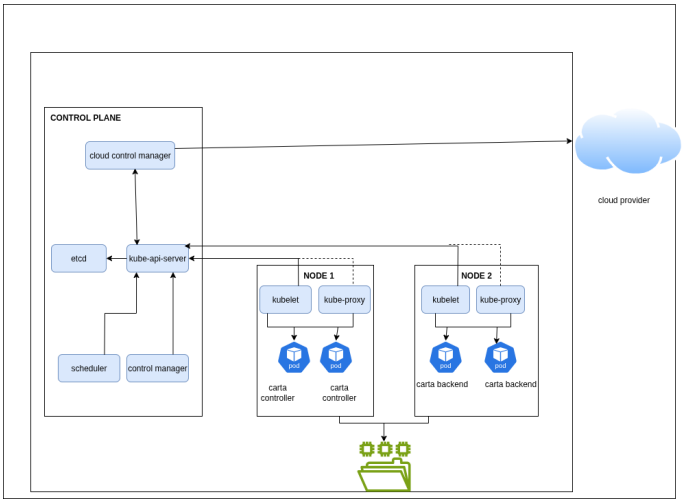

Figure 4: Kubernetes Cluster Architecture

3.6 Metrics to be measured

To compare deployment complexity between Kubernetes and SLURM, we define a set of qualitative and functional metrics. These metrics focus on the effort, flexibility, and operational requirements of deploying interactive applications, such as CARTA. The evaluation is based on the following key functional areas:

- Configuration Overhead: an effort to customize a generic installation configuration to work in a production cluster.
- Support for incorporating external systems such as authentication services (e.g., LDAP, OIDC) and distributed file systems (e.g, CephFS).
- Network and Service Exposure: Mechanisms available to expose interactive services to users, including ingress and access methods.
- Recovery Complexity: Behavior of each system when a service fails (e.g., pod crash or job termination) and the required recovery steps.
- Customization Effort: The extent to which a generic installation must be adapted to function within a production HPC or cloud-native environment.
- Documentation and Community Support: Availability of usage examples, templates, and community troubleshooting resources.

Next, we provide a plan for the work breakdown.

4 TIMELINE

This section provides our work breakdown, as shown in the figure below. The first part of the project is the research proposal, which includes the introduction, problem statement, significance of the study, literature review, methodology, and timeline. Next, we will deploy CARTA on both SLURM and Kubernetes. We then record the results and document which deployment is best for interactive applications.

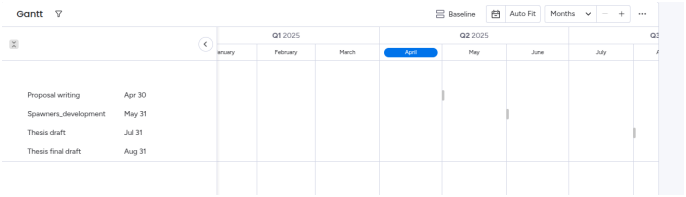

Figure 5: Timeline for Research

We utilize Monday.com for project management, Slack for collaboration, Overleaf for documenting our reports, and GitHub for storing code and scripts.

5 CONCLUSION

This proposal explores the deployment of interactive applications by comparing two distinct approaches: Kubernetes (K8s) and a conventional High-Performance Computing (HPC) environment using SLURM as the workload manager. The research will focus on understanding the configuration workflows, customization requirements, and operational considerations involved in deploying CARTA at scale in each environment.

By analyzing aspects such as user authentication, persistent storage integration, service orchestration, and the adaptability of deployment templates and tools, this study aims to evaluate how each platform supports reusable, production-ready configurations. The comparative analysis will provide insights into the relative deployment complexity of Kubernetes and SLURM and the effort required to transition applications from development to fully functional HPC environments.

The findings are expected to offer valuable guidance for system administrators, infrastructure designers, and researchers in selecting deployment strategies that align with their operational requirements, available resources, and organizational practices.

REFERENCES

[1] A Comrie, R Simmonds, A Pińska, and AR Taylor. 2020. Efficient Data Processing for Large Image Cube Visualisation. *Astronomical Data Analysis Software and Systems XXIX* 527 (2020), 217.

[2] Gor Mack Diouf, Halima Elbiaze, and Wael Jaafar. 2020. On Byzantine fault tolerance in multi-master Kubernetes clusters. *Future Generation Computer Systems* 109 (2020), 407–419.

[3] Mandeep Kumar and Gagandeep Kaur. 2022. Study of container-based JupyterLab and AI Framework on HPC with GPU usage. In *2022 International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON)*. IEEE, 1–5.

[4] Dirk Merkel et al. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux j* 239, 2 (2014), 2.

[5] Ruchika Muddinagiri, Shubham Ambavane, and Simran Bayas. 2019. Self-hosted kubernetes: Deploying docker containers locally with minikube. In *2019 international conference on innovative trends and advances in engineering and technology (ICITAET)*. IEEE, 239–243.

[6] Tiziano Müller, Nina Mujkanovic, Juan J Durillo, and Nicolay Hammer. 2023. Survey of adaptive containerization architectures for HPC. *arXiv preprint arXiv:2308.12147* (2023).

[7] Tatiana Ozerova, Elena Aksenova, Oleg Borisenko, Matvey Kraposhin, and Eshsou Khashba. 2021. Slurm PaaS in a cloud environment for development and debug purposes. In *2021 Ivannikov Ispras Open Conference (ISPRAS)*. IEEE, 127–133.

[8] Albert Reuther, Jeremy Kepner, Chansup Byun, Siddharth Samsi, William Arcand, David Bestor, Bill Bergeron, Vijay Gadepally, Michael Houle, Matthew Hubbell, et al. 2018. Interactive supercomputing on 40,000 cores for machine learning and data analysis. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–6.

[9] Jay Shah and Dushyant Dubaria. 2019. Building modern clouds: using docker, kubernetes & Google cloud platform. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 0184–0189.

[10] Joe Stubbs, Julia Looney, Marjo Poindexter, Elias Chalhoub, Gregory J Zynda, Erik S Ferlanti, Matthew Vaughn, John M Fonner, and Maytal Dahan. 2020. Integrating jupyter into research computing ecosystems: Challenges and successes in architecting jupyterhub for collaborative research computing ecosystems. In *Practice and Experience in Advanced Research Computing 2020: Catch the Wave*. 91–98.

[11] Sage Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*. 307–320.

[12] Andrew J Younge, Kevin Pedretti, Ryan E Grant, and Ron Brightwell. 2017. A tale of two systems: Using containers to deploy HPC applications on supercomputers and clouds. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 74–81.

[13] Naweiluo Zhou, Huan Zhou, and Dennis Hoppe. 2022. Containerization for high performance computing systems: Survey and prospects. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2722–2740.

[14] Andrea Zonca and Robert S Sinkovits. 2018. Deploying Jupyter Notebooks at scale on XSEDE resources for Science Gateways and workshops. In *Proceedings of the Practice and Experience on Advanced Research Computing*. 1–7.