A Comparative Analysis of Kubernetes and SLURM for Deploying Interactive Applications

Sanele Dlamini
University of Cape Town
South Africa

ABSTRACT

This study evaluates Kubernetes (an orchestration platform) and Slurm (an HPC workload manager) as deployment substrates, 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) [2] 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 the execution of applications on HPC clusters. To run jobs, users select from multiple preconfigured templates that describe how their application should run, specifying the number of CPU cores, the amount of memory, and GPUs on each node, as well as cluster-wide scheduling rules and policies (e.g., fair share priorities) that apply to all jobs. An alternative approach would be to deploy interactive applications using orchestration frameworks, such as Kubernetes (K8s) [6]. 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 in enterprise computing environments, providing built-in mechanisms that enhance system fault tolerance [3]. 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.

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 [12], a distributed network file system designed for petabyte-scale storage. CephFS provides shared, POSIX-compliant storage accessible from all nodes and pods. 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.

- To what extent are Kubernetes deployment configurations reusable across different environments, and what additional customization effort (e.g., configuration changes and added files) is required to transition from a basic to a production-ready CARTA deployment, as measured by the number of steps and lines of code?
- To what extent are SLURM deployment configurations reusable across different environments, and what additional customization effort (e.g., configuration changes and added files) is required to transition from a basic to a production-ready CARTA deployment, as measured by the number of steps and lines of code?
- How do Kubernetes and SLURM differ in handling system failures during CARTA deployment, measured by the number of automated recovery actions versus manual ones? Interventions required to restore functionality?

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.

Talwar et al. [11] compare four deployment models—manual, script-based, language-based, and model-based—along a scale of complexity. Manual deployment is easy to start, but does not scale. Script-based approaches add imperative automation, yet struggle with dependency management and fault tolerance. Language-based methods utilize declarative specifications to encode dependencies and lifecycles, thereby enhancing reuse and robustness. Model-based deployment takes it a step further, orchestrating services from high-level models with dynamic adaptation and self-healing capabilities. Using metrics such as deployment steps, lines of code for initial setup and changes, and time to develop/deploy, Talwar et al. demonstrate that manual/script-based approaches have lower upfront costs but degrade rapidly as systems grow. In contrast, language- or model-based approaches invest early and reduce long-term human effort. (Figure 1) captures this trade-off: more automation shifts effort earlier but pays off with scale and complexity.)

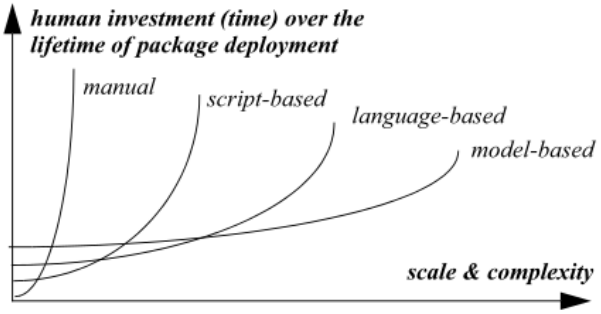

Figure 1: Hypothesis: [11] The level of automation of tools pushes the cost earlier in the development cycle to benefit repeated deployment. Developing, learning, and creating templates comes at an initial cost. The cost pays off in complex or scaled (repeated) deployment

Zonca and Sinkovitch [14] describe how to deploy JupyterHub at scale on XSEDE to support classes, workshops, and science gateways. Serving notebooks configurations, controllers, and desired-state management; SLURM aligns with the script-based deployment model ion, per-user isolation, persistent storage, and elastic capacity. They implement and document three strategies on XSEDE/Jetstream: An HPC batch-scheduler mode (SLURM/PBS) in which JupyterHub runs on a Jetstream VM, authenticates users (e.g., via CILogon), and uses BatchSpawner to submit one job per user to a supercomputer (e.g., Comet); once a job starts on a compute node, the single-user notebook is tunneled back to the Hub; A Docker Swarm deployment where the Hub and NGINX run as Swarm services and user notebooks are scheduled as containers across VMs, with persistent home directories provided by an NFS share on an XFS volume with quotas (SwarmSpawner mounts the user's folder on any node); and a Kubernetes deployment on Jetstream using Zero-to-JupyterHub on a private K8s cluster, where Rook/Ceph supplies distributed, redundant storage and KubeSpawner creates per-user pods with resource limits—capacity scales by adding or removing nodes. Results show that all three approaches are viable:

Docker Swarm works well for workshop-scale environments, the HPC batch-scheduler path enables access to supercomputing resources, and Kubernetes offers fault tolerance and demonstrated scalability to thousands of users.

Kumar and Kaur [4] 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 [5]. 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.

Packard et al. [8] investigate the operational challenges and practical lessons derived from deploying and managing self-hosted Kubernetes clusters at the Texas Advanced Computing Center (TACC). Their study addresses the persistent gap between the conceptual advantages of container orchestration and the real-world complexity of administering Kubernetes in research computing environments. Over two years of implementation using kubeadm, the authors explored cluster setup, storage integration with Ceph and RBD, network isolation via Flannel and Calico, and user management via namespaces and quotas. They evaluated Kubernetes as a flexible alternative to traditional HPC workload managers such as SLURM, highlighting its ability to support diverse workloads—including long-running Jupyter notebooks and large-scale data pipelines—while offering reproducibility, scalability, and self-service configuration. A case study deploying InfluxDB for the DARPA-WASH project demonstrated that Kubernetes could sustain high-performance data ingestion (2.6 million values/s using fifteen processors) and enable rapid prototyping for researchers with limited system-administration experience. Nevertheless, the authors emphasize significant challenges: steep learning curves for users, complicated ingress and networking setup, difficulties configuring persistent storage with CephFS/RBD, and high administrative overhead during upgrades and maintenance. The paper concludes that while Kubernetes can effectively serve as a production platform for scientific workloads, its successful adoption in HPC contexts depends on substantial operational expertise and well-planned automation strategies.

Aydin et al. [1] note that HPC infrastructures are growing in scale and complexity, making reliability in the face of failures (e.g., node crashes, network partitions, resource exhaustion) critical. They evaluate how two dominant systems—Kubernetes (container orchestration) and SLURM (HPC workload manager)—deliver fault tolerance in real HPC contexts and what trade-offs they impose on reliability and operations. Their research asks how Kubernetes and SLURM detect, withstand, and recover from common HPC failures; what their relative strengths and limitations are (e.g., MTTR, detection accuracy, overhead); and what guidance emerges for improving resilience across HPC, cloud, and edge environments. The results indicate that Kubernetes is better suited to dynamic, highly available, self-healing services, while SLURM remains strong for traditional HPC scheduling with tight resource control. The authors recommend hybrid strategies and AI-driven predictive maintenance to improve resilience in both systems.

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 it. For the SLURM setup testbed on OpenStack, we will create four virtual machines (VMs): three VMs will have 25 GB of disk space and 16 GB of RAM (Compute nodes), and one VM will have 15 GB of hard disk space and 8 GB of RAM (login node). 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 serve as a login node, and three additional VMs/nodes will serve 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 CARTA configurations for both Kubernetes and SLURM clusters.

3.2 Configuration of CARTA on Kubernetes

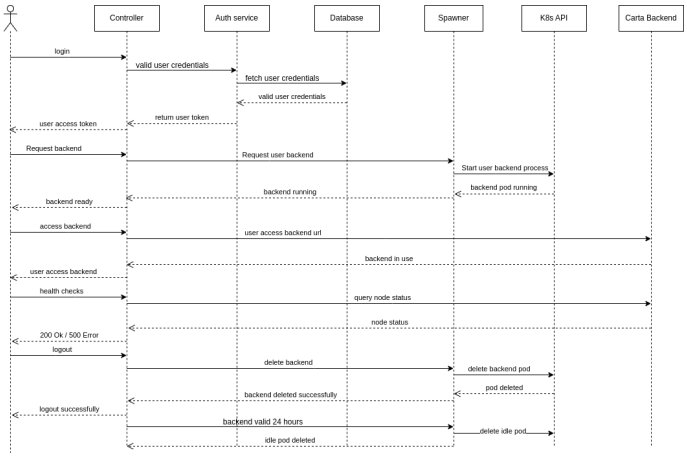

Figure 2: Kubernetes-Based System.

Figure 2 shows the sequence of events of 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 Controller is the entry point for users, responsible for managing authentication and spawning the Carta backend. The Controller forwards a 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 deletes inactive backend pods after 24 hours; this is configured in our system, not a Kubernetes default. This behavior is explicitly configured within the deployment system and is not a default Kubernetes feature.

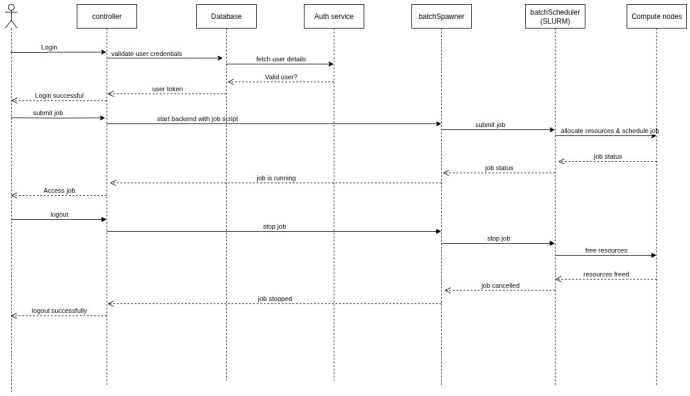

Figure 3: Using a Batch Scheduler

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

This section describes how CARTA is deployed using SLURM as a resource manager. Figure 3 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 BatchSpawner generates an SLURM job submission script using preconfigured templates and resource specifications. This submission script defines job parameters, including resource requests (CPU, memory, GPUs), execution time, and the CARTA backend launch command. BatchSpawner 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. BatchSpawner monitors the job status by periodically polling the scheduler. Once the job runs, BatchSpawner 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 BatchSpawner, which sends a cancellation request to the Batch Scheduler. The Batch Scheduler cancels the running job, releases the allocated resources, and notifies the BatchSpawner 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

Figure 3 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 submits jobs to Slurm, which are scheduled by slurmctld; the jobs execute on compute nodes via slurmd. The backend accesses the shared file system to load data. The frontend provides a user interface to view the image cubes from the backend.

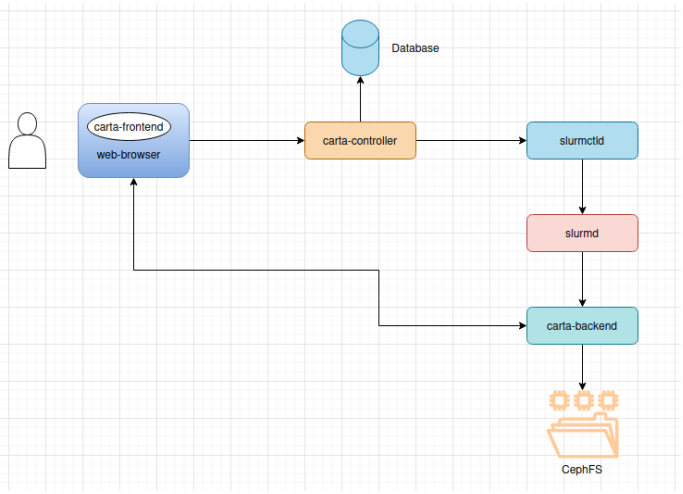

Figure 4: SLURM Architecture

3.5 Kubernetes Cluster architecture

Figure 4 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.

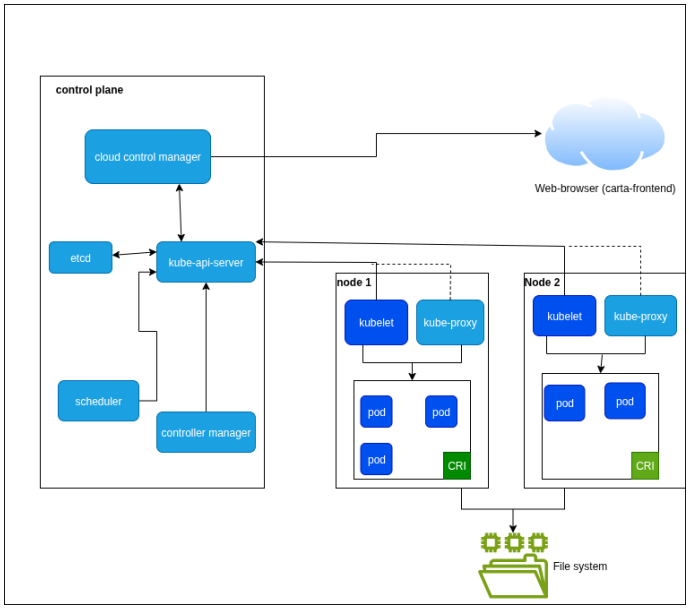

Figure 5: Kubernetes Cluster Architecture

3.6 Metrics to be measured

To compare deployment complexity between Kubernetes and SLURM, we define a set of qualitative and quantitative 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:

- Quantitative metrics
  - Number of lines of code (LOC) written for deployment
  - number of steps involved to deploy
  - LOC to express configuration changes
- Qualitative metrics
  - Ability to automate the management process, including adaptability to changes (e.g., failures, load).
  - Robustness, expressed in terms of misconfigurations.
  - Expressiveness of management (e.g., ability to express constraints, dependencies, and models).
  - Barrier to first use of the deployment tool.

Next, we present a work breakdown plan.

4 TIMELINE

Figure 5 provides our work breakdown. 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, the final stage involves integrating authentication into the CARTA deployments on both SLURM and Kubernetes to enable secure, user-specific access. We then record the results and document which deployment is best for interactive applications.

Figure 6 shows the timeline of this work. We utilize Slack for collaboration, Overleaf for documenting our reports, and GitHub for storing code and scripts.

Task	Sub-tasks	Estimated time frame	Status	Due date
Carta-controller	• Adding functionality to the controller to automatically start carta-backend when a user logs in. Exposing port 3002 through the socket protocol. Both K8s and Slurm • Adding carta-backend health checks on the controller to detect if carta-backend is deployed and running	Two months	In progress(done with K8s part)	30 November 2025
Carta-backend	• Mount CephFS on carta-backend pods • Mount CephFS to the cluster VMs	Two weeks	In progress	15 Dec 2025
Metrics collection and results analysis	Collection of metrics Results analysis	Two weeks	Not started	5 Jan 2025
Thesis first draft	First write-up	One month	Not started	05 Feb 2025
Thesis final draft	Final draft	Two months	Not started	10 April 2025

Figure 6: Timeline for Research

4.1 Risks and mitigation plan

Figure 7 provides the risk list, their impacts, and how we will control them.

Risk	Impact	Likelihood	Mitigation Strategy
Cluster configuration or networking failures (Kubernetes or SLURM testbeds)	Delays in deployment setup and metric collection	Medium	Maintain version-controlled configuration files; perform incremental testing; use snapshots on OpenStack to restore environments quickly.
CephFS integration or storage mount issues	Inability to access shared datasets, affecting CARTA tests	Medium	Test CephFS integration early; document mount procedures; use smaller datasets for functional verification before scaling.
Authentication or user-mapping errors during multi-user deployment	Blocking user sessions and inaccurate comparison results	Low	Use test users and mock identity providers (LDAP/OIDC) before live integration; log all access attempts for quick debugging.
insufficient compute resources or quota limits on OpenStack	Incomplete experimental runs or constrained scaling tests	Medium	Request temporary quota extensions in advance; use smaller virtual clusters for functional trials.
Timeline slippage due to unforeseen technical hurdles	Delay in data collection and documentation	Medium	Apply agile iteration: document partial results weekly; prioritize critical experiments first.
Software version incompatibility or dependency updates	Broken environments or inconsistent results	Low	Pin versions in Dockerfiles and SLURM scripts; maintain a reproducible environment

Figure 7: Risk lists and mitigation plan

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 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, as well as the effort required to transition applications from development to full-scale HPC production environments.

The findings will provide valuable guidance to system administrators, infrastructure designers, and researchers on selecting deployment strategies that align with their operational requirements, available resources, and organizational practices.

REFERENCES

[1] Mirac Aydin, Michael Bidollahkhani, and Julian M Kunkel. [n. d.]. Comparing Fault-tolerance in Kubernetes and Slurm in HPC Infrastructure. ([n. d.]).

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[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] Michael Packard, Joe Stubbs, Justin Drake, and Christian Garcia. 2021. Real-world, self-hosted Kubernetes experience. In *Practice and Experience in Advanced Research Computing 2021: Evolution Across All Dimensions*. 1–5.

[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] V. Talwar, Qinyi Wu, C. Pu, Wenchang Yan, Gueyoung Jung, and D. Milojicic. 2005. Comparison of Approaches to Service Deployment. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*. 543–552. https://doi.org/10.1109/ICDCS.2005.18

[12] 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.

[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.