Article

Not peer-reviewed version

Evolving HPC Data Centers with Kubernetes, Performance Analysis, and Dynamic Workload Placement Based on ML Scheduling

Vedran Dakić *, Mario Kovač *, Jurica Slovinac

Posted Date: 6 June 2024

doi: 10.20944/preprints202406.0074.v1

Keywords: high-performance computing; data center architecture; hardware and software integration; Kubernetes; dynamic performance evaluation

Article

Evolving HPC Data Centers with Kubernetes, Performance Analysis, and Dynamic Workload Placement Based on ML Scheduling

Vedran Dakić [1,*], Mario Kovač [2,*] and Jurica Slovinac [1]

[1] Algebra University, University of Zagreb, 10000 Zagreb, Croatia
[2] University of Zagreb, Faculty of Electrical Engineering and Computing; mario.kovac@fer.hr
* Correspondence: vedran.dakic@algebra.hr (V.D.); mario.kovac@fer.hr (M.K.)

Abstract: In the past twenty years, the IT industry has moved away from using physical servers for workload management to workloads consolidated via virtualization and, in the next iteration, further consolidated into containers. In the next step, container workloads based on Docker and Podman as underlying container technologies were orchestrated/automated via Kubernetes or OpenShift. On the other hand, high-performance computing (HPC) environments have been lagging in that process, as there's still much work to figure out how to apply containerization platforms for HPC in real-life scenarios. Kubernetes and OpenShift have many advantages – generally speaking, container technologies use quite a bit less overhead from the computing perspective while providing many benefits in flexibility, modularity, and maintenance. Therefore, they are ideal for tasks requiring a lot of computing power. There are also some tradeoffs regarding the complexity of these two platforms - they're just not that user-friendly when used by people without years of experience managing them. In this paper, we propose a different architecture based on seamless hardware integration and user-friendly, dynamic workload placement based on real-time performance analysis and prediction coupled with Machine Learning-based scheduling.

Keywords: high-performance computing; data center architecture; hardware and software integration; Kubernetes; dynamic performance evaluation

1. Introduction

The shift from conventional data center designs to more dynamic and efficient models is becoming increasingly critical in modern IT. Traditional data centers, which typically rely on fixed resource allocation and design patterns, are not well-suited to meet the variable demands of contemporary workloads, especially when dealing with HPC (High-Performance Computing). This often leads to inefficient resource use, higher operational costs, and elevated energy consumption. Adopting a dynamic approach involves leveraging orchestration technology such as Kubernetes to enhance workload management and execution. Kubernetes, a prominent container orchestration platform, supports automated deployment, scaling, and management of containerized applications. With substantial additional development and configuration effort, Kubernetes allows workloads to be assigned to the most efficient nodes instead of just the fastest ones, optimizing energy efficiency and cost.

Dynamic workload placement, guided by real-time metrics, resource availability, and energy efficiency, enables significant operational efficiency and sustainability improvements. This approach provides a more agile and responsive infrastructure that can adapt to diverse workload requirements, ensuring optimal performance while reducing energy usage and costs.

Implementing such dynamic strategies within data center operations marks a significant advancement, promising better resource utilization, a reduced carbon footprint, and alignment with

 doi:10.20944/preprints202406.0074.v1

modern business needs. This paper examines the technologies and strategies driving this transformation, focusing on the pivotal role of hardware and software integration orchestrated by Kubernetes in fostering a more efficient and sustainable HPC infrastructure. As we researched this topic, we built a fully functional platform that can do everything stated in this paper - install or add nodes to the Kubernetes cluster, do a dynamic evaluation, manage Kubernetes clusters, and place workloads based on Machine Learning (ML) input, as we will show later in this paper.

The goal of this paper is to propose a new software platform to apply the idea of containers and Kubernetes to HPC efficiently. The hardware part of that architecture can be designed as it has always been designed. The big difference between our approach and the traditional approach is how hardware and software interact in scheduling HPC workloads. Our ML-driven (Machine Learning) approach to workload placement offers many advantages, especially when integrated with an easy-to-use UI (User Interface).

The paper is organized as follows. The next sections provide relevant information about all technology aspects of the paper - containerization, container orchestration, virtualization technologies, and a general overview of HPC. Then we'll discuss real-life problems when applying Kubernetes to HPC workloads and offer our research and insight into how to solve these problems. This necessitated the creation of an entirely new software platform that aims to solve these problems in a way that's user-friendly and as automated as possible. One of our primary goals was to simplify how we create container workloads for our HPC environment and then schedule these workloads to be placed on the best possible node as calculated and recommended by a built-in AI engine. In our software platform, we can also override these recommendations and manually place workloads if we wish to do so. We discussed the way we designed our application in a separate section. The last sections of our paper are reserved for preliminary results, discussion about future research directions, limitations, and our conclusions.

2. Technology Overview

Virtualization, containers, orchestrators, and high-performance computing (HPC) are fundamental technologies in today's IT landscape, each enhancing the efficiency and scalability of computing environments in unique ways. Virtualization enables the creation of multiple virtual machines (VMs) from a single physical hardware resource, improving resource utilization and isolation. Containers, managed by tools like Docker and Podman, provide a lightweight alternative to VMs by encapsulating applications and their dependencies in isolated environments, thus ensuring portability and consistency across different deployment platforms. Orchestrators such as Kubernetes and OpenShift automate containerized applications' deployment, scaling, and management, ensuring high availability and streamlined operations in complex cloud infrastructures. HPC utilizes advanced architectures and parallel processing to handle demanding computational tasks, facilitating significant advancements in scientific research, engineering, and large-scale data analysis. Collectively, these technologies form the backbone of the modern digital ecosystem, driving innovation and enhancing operational efficiency across various industries. In this section, we'll go through a technology overview of all technologies related to our paper.

A. Virtualization

Virtualization abstracts physical hardware to create multiple simulated environments called virtual machines (VMs) from a single physical system, by creating a virtual framework that uses code that mimics the functionality of equipment [69]. Each VM operates its operating system (OS) and applications, isolated from other VMs, using a hypervisor. Hypervisors are categorized into Type 1 (bare metal) and Type 2 (hosted).

Type 1 Hypervisors are installed directly on physical hardware, acting as an intermediary layer between the hardware and the VMs. Examples include VMware ESXi, Microsoft Hyper-V, and Xen. Due to direct hardware access, these hypervisors are known for their high performance and low overhead. They are more efficient with much better performance since bare-metal hypervisors operate in the most privileged CPU level [1].

 doi:10.20944/preprints202406.0074.v1

Type-2 or hosted hypervisor is executed on the host operating system as a typical application [1]. The most prominent examples include Oracle VirtualBox and VMware Workstation. These hypervisors are typically easier to set up and are used primarily for development and testing. However, they generally do not match the performance levels of Type 1 hypervisors due to the extra layer of the host OS.

A primary advantage of virtualization is the strong isolation it offers. Each VM runs in a self-contained environment, enhancing security and stability. If one VM fails or is compromised, the others remain unaffected. This isolation allows diverse workloads and operating systems to run concurrently on a single physical server, maximizing resource utilization and flexibility.

However, this strong isolation comes at the cost of overhead. Each VM includes a full OS, leading to higher memory and storage consumption. This can make virtualization less efficient regarding resource usage than containerization. Despite this, virtualization is crucial for many scenarios, including running legacy applications, developing and testing various OS environments, and consolidating server hardware.

Another significant advantage of virtualization is support for live migration, which allows a running VM to be moved from one physical host to another with minimal downtime. This capability is essential for maintaining high availability and performing hardware maintenance without service interruption. VMware vMotion is the key feature of virtualization, and it allows a working virtual machine to move to another physical host without disrupting its service [2].

Virtualization also plays a critical role in disaster recovery. Virtual machine snapshots and backups enable administrators to restore systems to a previous state quickly, minimizing downtime and data loss in the event of a failure.

Additionally, virtualization facilitates better resource allocation and load balancing. Virtual machines can dynamically allocate CPUs, memory, and storage resources based on current needs, ensuring optimal performance and efficiency. Various policy-based approaches exist, such as resource pooling, reservations, limits, etc. Tools like VMware's Distributed Resource Scheduler (DRS) can also automatically balance workloads across multiple hosts, enhancing resource utilization. DRS takes resource management decisions according to metrics related to VMs, hosts, and clusters for memory and CPU (network and storage not considered) [3].

B. Containerization Technology

Containerization technologies have transformed application deployment and management by offering lightweight, portable, and consistent environments. It has been a component of the Linux kernel since 2008 [67]. This innovation began with BSD Jails, which introduced isolated user spaces. Docker later popularized containerization by streamlining container creation and management. Linux Containers (LXC) advanced the field by providing OS-level virtualization for Linux. Recently, Podman has emerged as a Docker-compatible alternative, offering daemonless container management and enhancing security and flexibility in container operations.

Containers offer significant advantages when compared to virtual machines. Some of the most important advantages include:

1. Efficient resource usage: Containers share the host system's kernel, eliminating the need for a complete operating system for each instance. This results in lower overhead than virtual machines (VMs), each requiring their own OS, leading to more efficient CPU and memory use. The container's average performance is generally better than the VM's and is comparable to that of the physical machine regarding many features [4].
2. Faster startup time: Containers can start almost instantly because they do not require booting an entire operating system. In contrast, VMs take significantly longer to start as they need to initialize a whole OS, and the startup time difference from server power-on can be up to 50% [5]. This speed advantage makes containers ideal for applications that require quick scaling and deployment.
3. Density: A single host can run many more containers than VMs due to their lightweight nature. For certain types of workloads, an application container's start-up time is 16x lower than that of a VM, and its memory footprint is 60x lower than that of a VM [6]. This higher density allows

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 June 2024 doi:10.20944/preprints202406.0074.v1

for better utilization of hardware resources, enabling more applications to run on the same infrastructure.

4. Consistency: Containers encapsulate the application and its dependencies, ensuring consistent behavior across different environments. This consistency across development, testing, and production reduces bugs related to environmental differences.
5. Better dependency management: Containers package all the necessary dependencies for an application, eliminating conflicts that arise when different applications require different versions of the same dependencies. This encapsulation simplifies dependency management and ensures reliable application performance across various environments.

Let's delve into some of the architectural details of these containerization methodologies.

C. BSD Jails

BSD jails are an efficient and secure virtualization mechanism in FreeBSD, introduced with FreeBSD 4.0 in 2000. They allow administrators to partition a FreeBSD system into multiple isolated mini systems called jails. In BSD jails, the operating system should be the point of greatest gain in jails-systems since the trapped systems are based on the sharing of the running kernel by the base system, thus eliminating the need for additional resources related to the implementation of a new system instance operating [7].

As shown in Figure 1., jails build on the concept of chroot, which restricts a process's view to a portion of the filesystem and enhances this by isolating processes, users, and the networking stack. This comprehensive isolation significantly improves security. BSD jails come in various forms, such as thick jails replicating a complete FreeBSD system and thin jails sharing more resources with the host system to save space. They can be assigned unique IP addresses and configured with specific network settings, enabling different software versions to run simultaneously, making them particularly advantageous in testing and development settings, and allowing safe testing of updates or new configurations without risking the host system's stability.

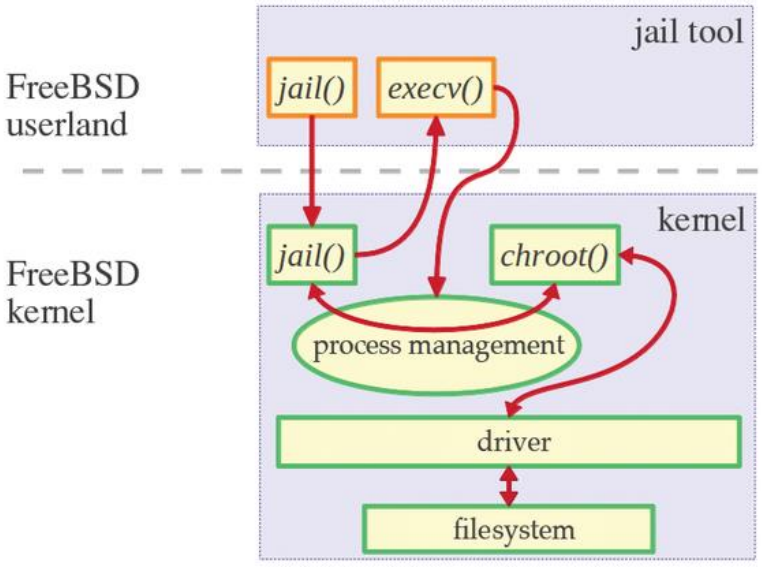

Figure 1. BSD jails architecture, https://www.admin-magazine.com/Archive/2013/13/How-to-configure-and-use-jailed-processes-in-FreeBSD, accessed on: 29 May 2024.

D. Docker

Docker, a platform for containerization, revolutionizes virtualization by packaging applications and their dependencies into containers, ensuring uniformity across different environments. This method contrasts sharply with traditional virtualization, where each virtual machine (VM) contains

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 June 2024 doi:10.20944/preprints202406.0074.v1

a complete guest operating system, leading to significant resource and performance overhead. Docker employs containerization to isolate applications from the host system, utilizing Linux kernel features such as namespaces and control groups (cgroups) to create lightweight containers.

As we can see in Figure 2., the Docker architecture comprises several vital components. The Docker Engine, the core of Docker, includes a daemon (dockerd) that manages container lifecycles, images, networks, and storage. This daemon interacts with the host's kernel to create and manage containers. The Docker Client is a command-line interface (CLI) used to interact with the Docker daemon, providing commands to build, run, and manage containers. Docker Hub, a cloud-based registry service, hosts Docker images, facilitating easy sharing and distribution.

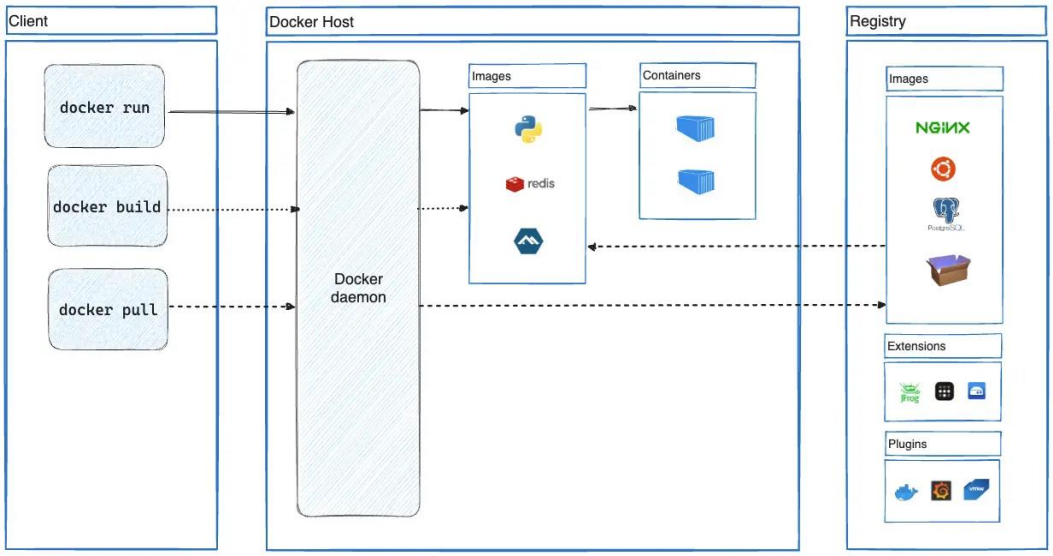

Figure 2. Docker architecture, https://docs.docker.com/get-started/overview, accessed on: 29 May 2024.

Docker containers share the host OS kernel, resulting in lower overhead than VMs. This shared kernel model allows for higher density and efficiency, as containers require fewer resources and start faster than VMs. A Felter et al. (2015) study demonstrated that Docker containers achieve near-native performance for CPU, memory, and I/O operations, often outperforming traditional VMs.

Managing many containers in production environments necessitates orchestration tools. Kubernetes is the most prominent tool for orchestrating Docker containers, providing automated deployment, scaling, and management of containerized applications. Kubernetes manages container lifecycles - automates their deployment, scaling, and management. With additional configuration, it can also manage network access, security, load balancing, and resource allocation across a cluster of nodes, providing high availability. Docker transforms the deployment and management of applications through efficient containerization, leveraging kernel features to offer lightweight virtualization. Security considerations must be addressed to ensure robust isolation despite its significant performance and resource utilization benefits. This is why it is necessary to protect the accuracy of the data, and the data's integrity must be preserved at all times [68]. Integrating orchestration tools like Kubernetes further enhances Docker's capabilities, making it an indispensable technology in modern DevOps practices. The term "DevOps" is a combination of the terms "development" and "operations [74], which refers to the unification of teams, transforming the existing silos into a set of teams that focus work on the organization and not on the activity within it, thus linking all the steps that already exist in software development (including delivery) [8]. For these mechanisms to work, the Dev and Ops teams need to collaborate and support each other [75].Kubernetes is a de-facto core component in implementing DevOps.

Although Docker as a project didn't invent containerization, it brought containers to the foreground in IT. It made them what they are today – the most prominent technology to package applications worldwide. The concept wasn't all that new– in UNIX systems, we could use *chroot*

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 June 2024 doi:10.20944/preprints202406.0074.v1

environments that could be traced as the beginning of the containerization idea since the end of the 1970s. We might consider the chroot idea the first idea that very crudely separated processes from the disk space, and out of that idea, in the 2000s, BSD jails were introduced.

E. LXC

After BSD jails, the LXC (Linux Containers) project was introduced to the market in 2008. Linux Containers (LXC) offer a lightweight virtualization method that provides an environment like a complete Linux system but without the overhead of a separate kernel. To isolate processes, LXC containers utilize kernel features such as IPC, UTS, mount, PID, network, and user namespaces. These namespaces ensure that processes within a container cannot interact with those outside, enhancing security.

One of the significant advantages of LXC is its simplicity, flexibility, and low overhead, as shown in Figure 3. Unlike traditional virtual machines, which require separate kernels and virtualized hardware, LXC containers share the host system's kernel, making them more efficient regarding resource usage [9]. LXC also offers comprehensive tools and templates for creating and managing containers. For instance, users can utilize lxc-create to create a container and lxc-start to initiate it. Containers can be configured to autostart, and their configurations can be managed through various commands and configuration files [10].

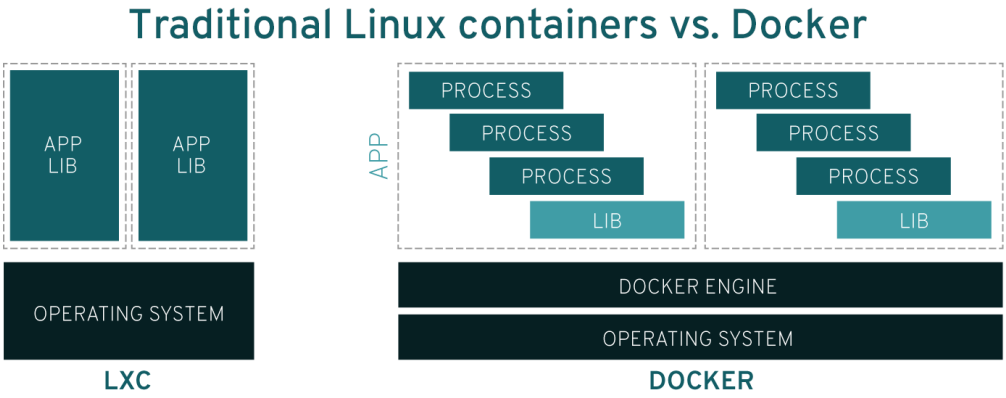

Figure 3. LXC vs Docker architecture, https://www.redhat.com/rhdc/managed-files/traditional-linux-containers-vs-docker_0.png, accessed on: 29 May 2024.

Moreover, LXC's architecture allows for high-density deployment of applications, which is particularly beneficial for environments that require rapid scaling and efficient resource management. This makes LXC an ideal solution for development, testing, and production environments where multiple isolated applications must run simultaneously on the same hardware. The ability to fine-tune resource allocation using cgroups further enhances LXC's efficiency, enabling precise control over CPU, memory, and I/O resources. As a result, LXC containers are widely used in enterprise and cloud computing environments to optimize performance and reduce costs [11,12].

F. Podman

Podman, short for "Pod Manager," is an open-source tool designed for developing, managing, and running Linux container systems. Developed by Red Hat engineers, Podman is part of the libpod library and is notable for its daemonless architecture, which enhances security and reduces overhead compared to Docker. Unlike Docker, Podman does not require a continuous background service (daemon) to manage containers. This makes it a more secure option by allowing containers to run without root privileges and thus minimizing potential security risks associated with root-owned processes [13].

Podman supports Open Container Initiative (OCI) standards and ensures compatibility with various container images and runtimes. Its command-line interface (CLI) is like Docker's, facilitating

 doi:10.20944/preprints202406.0074.v1

an easier transition for users. Podman integrates with other container management tools, such as Buildah for image building and Skopeo for image transfer between different registries [14].

Podman also manages pods, groups of containers that share the same resources, like Kubernetes pods, as shown in Figure 4. This feature benefits complex applications requiring multiple containers to work together seamlessly. Additionally, Podman offers a RESTful API for remote management and supports various Linux distributions, including Fedora, CentOS, and Ubuntu [15].

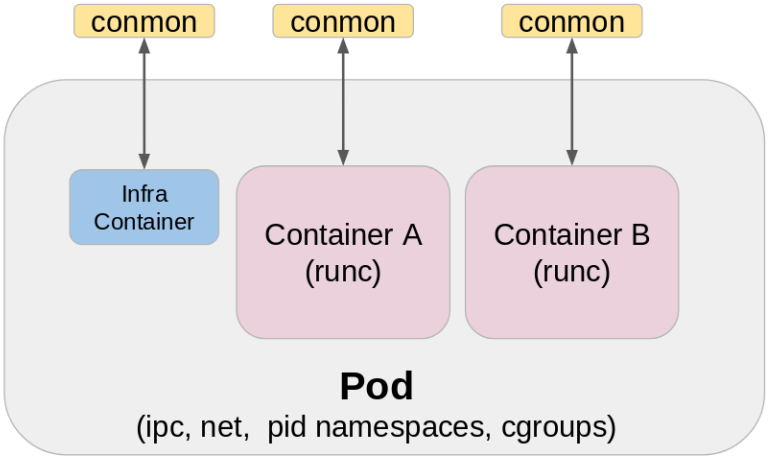

Figure 4. Podman architecture, https://developers.redhat.com/sites/default/files/blog/2019/01/podman-pod-architecture.png, accessed on: 29 May 2024.

The introduction of Podman in 2019 marked a shift away from Docker's dominance in the market. Podman addressed a significant issue with Docker: its monolithic architecture, which was seen as overly complex for large-scale designs, particularly concerning the high availability of the Kubernetes control plane [16]. While these platforms were not initially designed for high-performance computing (HPC) environments and large heterogeneous clusters, the idea has gradually gained traction. It is now becoming mainstream in the HPC world. HPC is an ideal use case for applications running close to the hardware to avoid costly indirection layers such as virtualization [17].

Consolidating IT infrastructure through containers and virtualization has transformed how IT deploys and manages applications. Leveraging these technologies can achieve greater efficiency, scalability, and resource optimization. The following section will discuss a detailed comparison of leading containerization and virtualization technologies. This comparison will highlight their unique features and suitability for different use cases, guiding us in selecting the best approach for our specific use cases.

G. Kubernetes

Docker and Podman's popularity led to the development of the next set of products for efficient management and scaling (Kubernetes and OpenShift). We're deliberately skipping the idea of Docker Swarm as it's obsolete and not applicable to standard, let alone HPC use cases [41]. These two products (Kubernetes and OpenShift) are, in essence, container management and orchestration platforms. Container orchestration technology was initially developed and applied inside large IT companies like Google and Yandex to deploy scalable high-load services [26]. Figure 5. illustrates the standard Kubernetes architecture orchestrating containers looks like:

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 June 2024 doi:10.20944/preprints202406.0074.v1

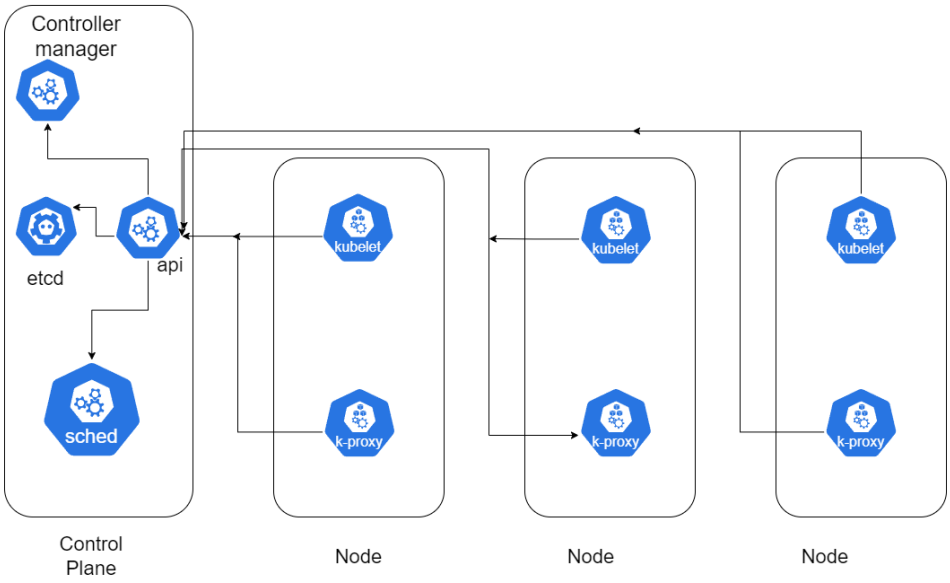

Figure 5. Standard Kubernetes architecture, https://kubernetes.io/docs/concepts/overview/components, accessed on: 29 May 2024.

Kubernetes is an open-source platform for container orchestration. It is essential in today's cloud computing landscape because it can automate containerized applications' deployment, scaling, and management. Its architecture offers a robust, scalable, and flexible management system for containerized applications across a cluster of nodes.

Kubernetes employs a master-worker architecture at its foundation. The master node is pivotal in maintaining the cluster's desired state, orchestrating tasks, and managing the overall lifecycle of the cluster. Key components of the master node include the API Server, which acts as the control plane's front end, handling all RESTful API requests. The etcd service is a key-value store that persists cluster state and configuration data. The Controller Manager manages controllers for routine tasks like node health monitoring and replication, and allows the cluster to be linked to the cloud platform [70]. Lastly, the Scheduler assigns workloads to specific nodes based on resource availability and predefined constraints [42].

Worker nodes, which host the containerized applications, include several critical components. The Kubelet is an agent running on each node, ensuring containers within a Pod run. It communicates with the master node to receive commands and reports the node's status. The Kube-Proxy manages network communication within the cluster, implementing network rules for Pod connectivity. The Container Runtime (e.g., Docker, containerd) is also responsible for running the containers [43].

Kubernetes organizes applications into Pods, the most minor deployable units and can contain one or more containers. Pods share the same network namespace, allowing communication via localhost. They are designed to be ephemeral, meaning they can be replaced or rescheduled as needed. Kubernetes uses ReplicaSets and Deployments to manage Pods, ensuring the desired number of Pod replicas run at any time [44].

A key feature of Kubernetes is its ability to perform service discovery and load balancing. The Service abstraction provides a stable IP address and DNS name to a set of Pods, ensuring reliable access. Services can be exposed internally within the cluster and externally to the internet. Kubernetes also supports Ingress resources, which manage external access to services, typically via HTTP/HTTPS [45].

Kubernetes offers Horizontal Pod Autoscaling (HPA) for scaling and resource management. HPA adjusts the number of Pod replicas based on observed CPU utilization or other selected metrics. This ensures efficient resource utilization and allows the system to respond to workload demands dynamically [46].

 doi:10.20944/preprints202406.0074.v1

Kubernetes emphasizes declarative configuration and automation. Users define desired states using YAML or JSON manifests, and Kubernetes ensures that the actual state matches the desired state. This approach simplifies management and enhances the reproducibility and reliability of deployments [47]. It helps deliver a scalable, flexible, resilient platform for managing containerized applications. Its master-worker model and robust abstractions for managing workloads and services make it a critical tool in modern cloud-native environments.

Now, if we expand our view of the problems we mentioned and add high availability of applications into the discussion, it becomes obvious why Kubernetes and OpenShift became so popular. Orchestrators reduce costs and allow mechanisms to be put in place that contribute to the application's resilience, providing adaptiveness to different operation environments and even fault tolerance mechanisms [27]. They are a developer's dream, as the hundreds or thousands of containers can run web applications to our heart's content. We can easily load-balance access to them via Kubernetes/OpenShift load-balancing objects (Ingress controllers, various load-balancers like Traefik, etc.) and achieve high availability via workload balancing. Stateless application architectures like the ones usually used by Kubernetes and OpenShift are, therefore, much easier to code. Furthermore, if we create such applications, we don't have to deal with clustering complexity, as the concept of high availability via clustering becomes unnecessary. This is a perfect case for real-life Kubernetes and one of the most common design patterns.

H. OpenShift vs Kubernetes

Kubernetes is an open-source container orchestration platform that automates deploying, scaling, and managing containerized applications. It uses a master-worker architecture, where the master node manages the cluster's state, and worker nodes run the containers. Critical components of Kubernetes include the API Server, etcd, Controller Manager, Scheduler, and Kubelet [48]. Kubernetes is known for its modular and extensible nature, allowing integration with various plugins and tools to enhance functionality [49]. It offers powerful features for service discovery, load balancing, horizontal scaling, and declarative configuration management [50].

OpenShift, developed by Red Hat, is a Kubernetes distribution that extends Kubernetes with several enterprise-grade features and tools, making it easier to manage containerized applications. OpenShift includes all Kubernetes features and adds capabilities like integrated CI/CD pipelines, developer-friendly tools, enhanced security features, and multi-tenancy support [51]. Its architecture is designed to provide a comprehensive and secure platform for running containerized applications in enterprise settings [52].

One of the main differences between Kubernetes and OpenShift is the level of built-in support and integration. OpenShift includes a built-in image registry, integrated logging and monitoring tools, and a web-based management console, which is not provided by default in Kubernetes. This makes OpenShift a more turnkey solution, especially for organizations that need these additional features without manual integration [53], as we can see from Figure 6.:

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 June 2024 doi:10.20944/preprints202406.0074.v1

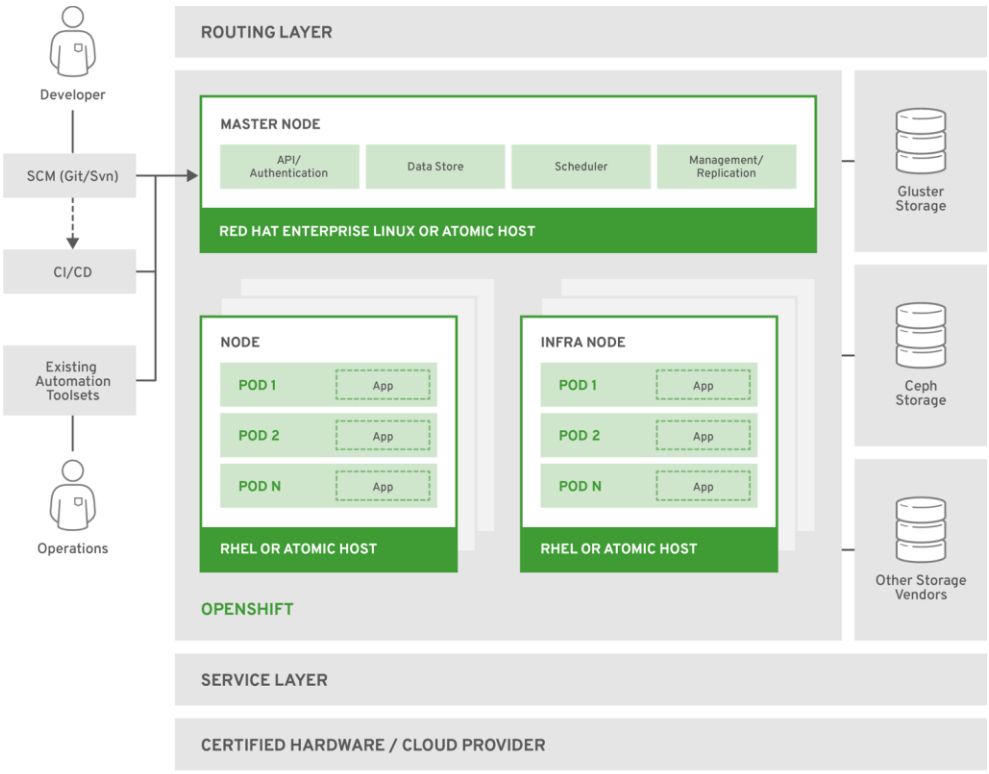

Figure 6. OpenShift architecture, https://docs.openshift.com/container-platform/3.11/architecture/index.html, accessed on: 29 May 2024.

Moreover, Ope nShift enforces stricter security policies by default. For instance, it includes Security Context Constraints (SCCs) that define a pod's actions, enhancing the platform's overall security. While Kubernetes is highly customizable, achieving a similar security level requires more manual configuration [54].

Both Kubernetes and OpenShift offer robust platforms for managing containerized applications, but OpenShift provides a more comprehensive and enterprise-ready solution with additional features and integrated tools. Kubernetes remains a powerful option for users who prefer to customize their environment to specific needs. Our choice of orchestration platform for this paper will be Kubernetes, as we don't want to impose even more cost to our environment related to OpenShift licensing.

I. HPC

High-performance computing (HPC) is essential for solving complex computational problems across various scientific and engineering fields. These systems are built to provide maximum computational power, utilizing advanced architectures and innovative technologies to handle large datasets and execute extensive simulations efficiently.

HPC systems have evolved to meet increasing computational demands. Although the HPC systems range from a desktop computing with an accelerator, through clusters of servers and data centers, up to high-end custom supercomputers [71], modern HPC architectures typically involve multi-core processors, parallel computing techniques, and high-speed interconnects. They are categorized into various types, including vector supercomputers, high-performance RISC processors, and parallel computing architectures like symmetric multiprocessors (SMP), workstation clusters, and massively parallel processors (MPP) [55]. Key components include processors, memory subsystems, interconnects, and storage systems. Advanced systems use heterogeneous architectures, incorporating accelerators such as GPUs, FPGAs, and specialized processing units to enhance performance and energy efficiency [56].

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 June 2024 doi:10.20944/preprints202406.0074.v1

A crucial aspect of HPC architecture is the interconnect network, which links multiple compute nodes into a cohesive system. Efficient communication between nodes is vital for high performance, especially in massively parallel systems. Technologies like InfiniBand and high-speed Ethernet achieve low-latency and high-bandwidth communication [57]. For years and years, all of these moving parts have mainly been using virtualization, similar to the architecture shown in Figure 7.:

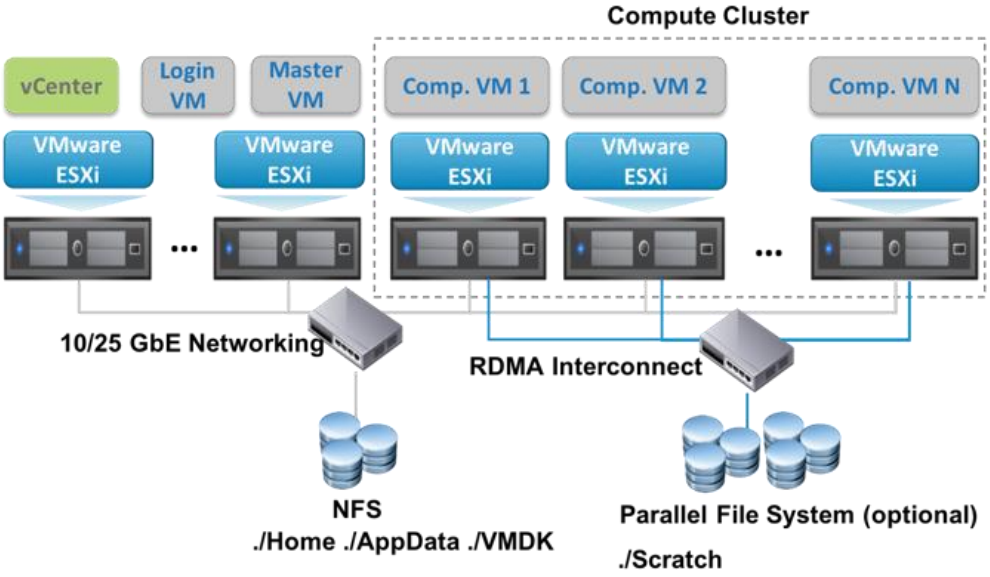

Figure 7. Virtualized HPC architecture, https://blogs.vmware.com/apps/2018/09/vhpc-ra-part1.html, accessed on: 29 May 2024.

HPC is applied in many fields requiring substantial computational resources. Everyday use cases include weather forecasting and climate modeling, molecular dynamics and bioinformatics, engineering and manufacturing, AI/ML training, oil and gas exploration, genomics, financial modeling, risk analysis, etc. [58–62].

HPC systems combine parallel computing techniques, multi-core processors, and specialized hardware accelerators to achieve high performance. Key components and architectural paradigms include:

1. Parallel Computing: HPC systems rely on parallel computing, where multiple processors perform computations simultaneously. This includes fine-grained parallelism, where tasks are divided into smaller subtasks, and coarse-grained parallelism, where larger independent tasks run concurrently [55].
2. Multi-Core and Many-Core Processors: These processors have multiple processing cores on a single chip, enhancing computational throughput, and are widely used in HPC systems [56].
3. Accelerators and Heterogeneous Computing: HPC systems use accelerators like GPUs, FPGAs, and specialized processing units to handle specific calculations more efficiently than general-purpose CPUs. Heterogeneous computing combines CPUs with accelerators, leveraging the strengths of different processing units to optimize performance [57].
4. Distributed and Cluster Computing: HPC systems are often organized as clusters of interconnected computers (nodes), each with its own processors, memory, and storage. These clusters can scale to thousands of nodes, handling large datasets and complex simulations. Interconnects like InfiniBand and high-speed Ethernet enable fast node communication [60].
5. Memory Hierarchy and Storage: Due to the large data volumes processed, efficient memory management is critical in HPC systems. HPC architectures use multi-level memory hierarchies, including cache, main memory, and high-speed storage solutions, to ensure quick data access and minimize latency [61].

Traditionally, HPC systems are typically equipped with workload managers that excel in specific tasks tailored to traditional use cases rather than new and emerging workloads such as AI. A workload manager comprises a resource manager and a job scheduler [29].

 doi:10.20944/preprints202406.0074.v1

This concludes our technology overview section of all pertinent technologies relevant to our paper. Let's now discuss the differences between virtualization and containers (the standard technology in HPC versus its up-and-coming competitor), so we can describe real-life problems when using containers for HPC workloads.

3. Virtualization vs Containers

Virtualization and containerization are two fundamental technologies in modern computing, each with unique advantages and specific use cases. Virtualization involves creating multiple simulated environments (VMs) on a single physical system using hypervisors, which provides strong isolation but with higher overhead. A container engine enables lightweight and efficient application deployment using OS-level features to create isolated user spaces. Here are ten key differences between virtualization and Podman, used here as an example of the currently recommended container engine:

1. Architecture: Virtualization employs a hypervisor to create and manage VMs, each with its own OS. In contrast, Podman runs containers on a shared OS kernel, which reduces overhead and boosts efficiency.
2. Isolation: VMs achieve strong isolation by running separate OS instances. Podman containers provide process and filesystem isolation through namespaces and control groups, which, while less robust, are adequate for many applications.
3. Resource Overhead: Virtualization demands more resources because an entire OS instance is needed per VM. Podman containers are lightweight, sharing the host OS kernel and minimizing resource usage.
4. Performance: VMs generally have higher overhead and can be less efficient. Podman containers deliver better performance and faster start-up times since they do not require a full OS boot.
5. Deployment Speed: Deploying VMs takes longer due to OS initialization. Podman containers can be deployed rapidly, making them ideal for quick development and scaling.
6. Resource Allocation: Resource allocation in virtualization is managed through the hypervisor and can be static or dynamic. Podman allows for flexible, real-time resource allocation for containers.
7. Use Cases: Virtualization is well-suited for running diverse OS environments, legacy applications, and high-security workloads. Podman excels in microservices, CI/CD pipelines, and modern application development.
8. Security: VMs offer strong security with robust isolation, making them suitable for untrusted workloads. Podman containers are secure but rely on the host OS kernel, which can present vulnerabilities.
9. Maintenance: Managing VMs involves handling multiple OS instances, leading to higher maintenance overhead. Podman containers simplify maintenance by sharing the host OS and dependencies.
10. Compatibility: VMs can run different OS types and versions on a single host. Podman containers are limited to the host OS kernel but can quickly move across compatible environments.

Virtualization and containers have unique strengths and serve different purposes in modern IT infrastructure. Understanding these differences is crucial for selecting the appropriate technology for specific use cases and balancing isolation, performance, and resource efficiency. Therefore, there will be situations where using virtual machines will be much more convenient for us, even if that means more overhead and larger compute objects (virtual machines).

Let's now discuss the problems in terms of how to apply the idea of Kubernetes to HPC systems with workloads that can hugely benefit from hardware-specific acceleration.

4. Challenges using Containers for HPC Workloads

HPC workloads have traditionally run on physical and virtual machines for many years and, for the most part - stayed away from containers as a delivery mechanism. There are good reasons why virtualization technologies are often much more convenient than containerization technologies when discussing HPC use cases. One of the most important is how they allocate hardware. Advanced

 doi:10.20944/preprints202406.0074.v1

hardware capabilities such as PCIe (PCI Express) passthrough for dedicated hardware access from a virtual machine and SR-IOV (Single Root I/O Virtualization) for hardware partitioning are well established and familiar when using virtualization technologies while being sparsely available on containerization technologies. The primary reasons for this aren't hardware-based but driver- and support-based. A lot of PCI Express cards we'd like to use in HPC environments aren't supported by container technologies. This might have a significant impact on how we accelerate HPC workloads, as we often need a GPU (Graphics Processing Unit), ASIC (Application-Specific Integrated Circuit), or FPGA (Field-Programmable Gate Array) to both accelerate workloads and make them much more efficient from compute/energy efficiency standpoint. But we also have in mind that HPC is a perfect use case for applications running as close to the hardware as possible to avoid costly indirection layers such as virtualization [20]. That's why we'd like to use containers for HPC workloads, as soon as possible.

Containers are almost exclusively used to create and deploy modern **web** applications, not HPC applications. Typically, single-threaded applications (like most web applications) need a horizontal (scale-out) scaling approach, as the traditional way of vertical scaling (scale-up) doesn't increase performance. On the other hand, HPC applications are not designed to be run on a single core - they're usually massively parallel and have traditionally been associated with algebraic solvers [72]. Here are some common reasons for such a design:

1. HPC application problem size and complexity: HPC apps have large datasets and complex calculations that a single processor cannot efficiently manage. Because of this, the problem is divided into many small tasks that can be processed concurrently using multiple processors or nodes [21].
2. Speed and efficiency: Massively parallel design enables HPC apps to be quickly executed on HPC systems much faster than traditional computing methods. HPC applications can achieve significant speedups by utilizing hundreds, thousands, or more processors in parallel, solving problems much quicker than with a single processor [22].
3. Scalability: HPC application scalability is essential to HPC application design, especially as datasets grow larger and more complex problems are found, necessitating more computation power. Different parallel algorithms are developed to improve performance as we scale to more processors [23].

Multiple parallel computing models have been developed over the years to enable us to achieve these massively parallel speedups:

1. MPI (Message Passing Interface): Allows processors to communicate by sending and receiving MPI messages.
2. OpenMP (Open Multi-Processing): This enables us to do parallel application programming for shared memory architecture.
3. CUDA (Compute Unified Device Architecture): Enables parallel processing on GPUs, often used for HPC because they're massively parallel hardware devices by design [24]. In GPU parallel programming and CUDA framework, "kernels" and "threads" are foundational concepts for programming [73].

With this in mind, there are three significant operational problems when deciding to use containers running HPC workloads:

1. HPC applications are usually deployed as standard applications, meaning they must be re-packaged into layers and container images if we want to run them in containers. This is a complex process and poses a real challenge. It's like the regular DevOps story of re-architecting a monolith application to be a microservices-based application, only much worse because HPC application libraries tend to be gigabytes and terabytes in size [8]. And that's even before we start discussing all potential security issues (image vulnerabilities, malware, clear text secrets, configuration issues, untrusted images, etc.) [63] or potential performance degradation [64].
2. Even if we manage to package them into containers—which is not a given—we need to be able to run them on a scale, which means running them via Kubernetes. Again, this is not a small task, as understanding Kubernetes architecture, commands, intricacies, and the YAML files we need to create to run applications manually is also very complex.

 doi:10.20944/preprints202406.0074.v1

Regarding workload placement, we must also be extra careful when running HPC applications on a Kubernetes cluster. The way Kubernetes scheduler distributes workloads leaves much room for improvement. It is well-suited for microservice-based web applications, where each component can be run in its container [65]. Still, it's not as efficient as it could be when discussing real-time applications with specific Quality of Service (QoS) requirements [66]. This becomes even more relevant with the size and scale of HPC data centers and their design. For example, suppose the only envelope we're pursuing in our HPC data center is the compute power. In that case, we want to have as many many-core processors and accelerators (GPU, FPGA, ASIC) as possible. From the perspective of operating systems used in these environments, there are also some long-forgotten problems with that approach.

For example, let's say that we want to use AMD Epyc Rome or Milan processors in our HPC data center. After we deploy our servers and install operating systems on top of them, we will notice some of the underlying problems we ignored for quite a while re-emerged with a vengeance. As an illustration, if we were to run AMD EPYC 7763 128-thread CPUs on today's operating systems, we'd soon notice that both Linux and Windows are pretty confused about how to use them – they won't recognize their correct core numbers, sometimes it will require BIOS re-configuration, many BIOS firmware updates and OS updates, etc.

Then, we have other design problems, as our current platforms have a real problem with memory, which is the component that introduces most of the latency into the computing problems (easily two orders of magnitude more significant than what CPU can handle internally). This also leads to the latest CPU designs with integrated memory, which seems to be a move in the right direction.

In that sense, an excellent explanation of why containers are so crucial for HPC is that the demise of Dennard scaling in the early 2000s spurred High-Performance Computing (HPC) to fully embrace parallelism and rely on Moore's law for continued performance improvement [28]. This is both good and bad simultaneously – as we can't stretch out power envelopes to infinity. We mostly thought of this problem as a hardware design problem, but that time is gone now – it's a hardware-software co-design problem. This is especially evident and felt in HPC workloads as these types expose the wrong design patterns even more because of how extreme the HPC workloads are as use cases. This is why we must move forward with our data center design principles to manage our Kubernetes-based HPC workloads better.

That means our hardware-software co-design problem becomes a hardware-software codesign/integration problem. We're no longer in an era where the efficiency of data center design took second place in overall data center performance. That means we must redefine the integration level between our hardware and software to use available resources efficiently, considering hardware heterogeneity and workload diversity in HPC environments.

We have already mentioned some problems related to the Kubernetes scheduler, so let's discuss them. The main issue is the very **robotic** nature of the default Kubernetes scheduler, which isn't suited to latency and bandwidth-sensitive environments such as HPC. This is why HPC-based environments will need a different architecture for the Kubernetes scheduler to use HPC data centers' capabilities fully. In that sense, we need more advanced context-aware scheduling algorithms that are not only based on regular resource utilization metrics such as CPU, RAM, disk, and network; we need to explore sustainable scheduling techniques in depth to reduce energy consumption and ensure a lower carbon footprint [65]. This is exactly what our software platform does with its hardware-software integration, as it monitors energy usage and feeds that data to multiple AI engines to predict more optimal placement of workloads when using HPC applications. In section 6., we'll discuss the architecture of our solution to these problems in detail.

5. Experimental Setup and Study Methodology

The experimental setup consisted of four nodes and a Kubernetes control node. Every node has 4 CPU cores and 8 GB of memory and uses SSD for storage. We also reserved a separate node with Intel Core i5-10400F CPU and 16GB of memory with NVIDIA GTX 1660 SUPER GPU to train our

 doi:10.20944/preprints202406.0074.v1

neural network model. We deliberately chose a simple and light (on hardware) setup as it should indicate whether our methodology works and if our hypothesis stands - the hypothesis being that the Kubernetes scheduler could do better if we fully customized it and if it gets help from the ML engine.

Our research started with a general use case—we wanted to see how Kubernetes first schedules regular web applications via its default scheduler. Then, we embarked on a journey to create our custom Kubernetes scheduler that should do a better job. One of the best examples of how simple Kubernetes scheduler is that it thinks of a CPU core as a CPU core, no matter the CPU generation, manufacturer, or frequency - it divides every single core into 1000 CPU units.

The next step was the development of a custom scheduler to iron out the situations where we weren't happy with the workload scheduling of the custom Kubernetes scheduler. For that purpose, we created a test suite that measures web application response time across multiple scenarios - a scenario in which the default scheduler does the placement and a scenario where our custom scheduler does the placement. Then, we used the data gathered to train a neural network to get an estimated response time for future web application scheduling via our custom scheduler. That also enabled us to calculate the difference in performance of the custom scheduler vs the estimated response time, as visible in Table 1.:

Table 1. Response times for our test scenario with a web application.

Test scenario	Estimated neural network response time	Custom scheduler response time	Mean response time error	Default scheduler response time
No workloads placed	6,45	5,89	0,56	5,94
One node is used for workloads	6,4	6,09	0,31	6,15
Two nodes are used for workloads	6,53	6,4	0,13	6,95
Three nodes are used for workloads	6,6	6,69	0,09	7,88
Four nodes are used for workloads	7,58	10,76	3,18	12,72

There's no question that our methodology works and that our hypothesis stands, and results bear these conclusions out. Our scheduler was always faster than the default one, and the difference reached staggering levels once we reached four nodes, the difference being more than 15%.

Once we did these tests, we scaled our environment to include more nodes using a set of four HP ProLiant DL380 Gen10 nodes with NVIDIA A100 GPUs. Each node has 24 CPU cores and 384GB of memory. This time, we used a container image with a set of NVIDIA HPC-Benchmarks to check our performance numbers for a use case much more tailored to HPC. We modified the container to start HPCG-NVIDIA after it boots to have reliable data regarding how much the scheduler took before HPCG-NVIDIA began running. We used the same general approach to testing as we did in the web application scenario - of course, the web application was replaced by HPCG-NVIDIA. Table 2 gives us more information about that scenario:

Table 2. Response times for our test scenario with HPCG-NVIDIA.

Test scenario	Estimated neural network response time	Custom scheduler response time	Mean response time error	Default scheduler response time
No workloads placed	9,32	9,08	0,24	9,28

 doi:10.20944/preprints202406.0074.v1

One node is used for workloads	10,13	9,76	0,37	10,03
Two nodes are used for workloads	12,41	11,78	0,63	12,13
Three nodes are used for workloads	13,78	13,37	0,41	13,59
Four nodes are used for workloads	14,96	14,41	0,55	14,88

These results are an even stronger indicator that our approach works well and merits further research and development. Based on our results, we concluded that there are two ways in which we could proceed:

1. We could develop a custom scheduler per HPC application to place applications on Kubernetes pods. This would require a lot of time in terms of coding the Kubernetes schedulers, which might or might not work. Furthermore, there's no guarantee that this approach would work across multiple HPC data centers, making the idea wrong.
2. We could develop something that acts as if a set of custom schedulers is present, without writing a set of custom schedulers, that could modify workload placement as it gathers information from the environment, learns about it, and then explicitly places workloads (or better yet, offers to place workloads) on a more suitable node or set of nodes. This is why we went with an ML-based idea that completely replaces the concept of writing multiple schedulers for multiple applications, as this process becomes unnecessary.

For these workload placement decisions to be as correct as possible, we had to create a system that would use a set of parameters to determine which node gets picked based on a score, and this score must be completely independent of the scores already available in Kubernetes. Also, these parameters need to be much more nuanced than splitting any CPU core into 1000 CPU units - parameters need to reflect actual performance for a given resource. They needed to represent various hardware devices that contribute to server performance - CPU performance, memory performance, storage performance, networking performance, GPU performance, and FPGA/ASIC performance. Power usage (efficiency) needs to be a part of the scoring system, as well as the health state of the server, as we might want to exclude a server from workload placement if there's a hardware failure status attached to it. That meant that all newly added nodes needed to go through multiple phases as they were deployed to the HPC environment - they needed to go through a scanning phase (to determine the hardware content of the server), a performance test phase, and a permanent power and health monitoring phase. These parameters represent a good selection on which to base our scores so we can have enough data to feed to ML engines to learn from. Our research has led us in this direction as it's impossible to do proper workload scheduling by counting on the Kubernetes scheduler only, and it's also impossible to create a single scheduler that would work well for all the design ideas we have nowadays.

6. Proposed Platform Architecture for Kubernetes Integration with HPC

Dedicated computing systems, like HPC clusters, are preconfigured by systems administrators with specialized software libraries and frameworks. As a result, workloads for these dedicated resources must be built to be hardware-specific and tailored to the system to which it is meant to be deployed to extract the maximum performance [30]. This hinders flexibility if, for example, we want to introduce manageable energy efficiency into the scheduling process. Still, at the same time, we don't want to do that at the expense of environmental complexity and lack of user-friendliness. We also want to have the capability to schedule workloads manually if such a scenario occurs, and we want to do all of these scenarios without having to spend a year learning YAML structure and

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 June 2024 doi:10.20944/preprints202406.0074.v1

Infrastructure-as-a-code, in general, to be able to schedule HPC workloads. For this purpose, it is not sufficient to assign each user a fixed priority (or a static share) and keep it intact [31]. But we can see a tremendous value in using Kubernetes and container platforms to package our applications as – if done correctly – these scenarios will yield better performance than virtual machines and be less wasteful on other resources, like disk space.

Our proposal extends the traditional approach of virtual machines for HPC workloads by providing deeper hardware-software integration. This will involve power usage tracking, deep infrastructure monitoring and scanning, PDU (Power Distribution Unit) management, and using ML engines to learn about the environment and efficiently schedule workloads.

This article proposes a new, multi-layer architecture based on the Kubernetes orchestration platform to deploy, configure, manage, and operate a large-scale HPC infrastructure. The only standard architecture and design principle we use is hardware, which is the same method we always use when designing large-scale HPC centers. We need servers, CPUs, memory, FPGAs, ASICs, and GPUs to handle the workload (what we refer to as the Hardware layer), but how we want that hardware to interact with HPC applications via Kubernetes and how workloads get placed on top of hardware is entirely different and based on our custom scoring system. Figure 8. illustrates the high-level overview of the architecture of our software platform:

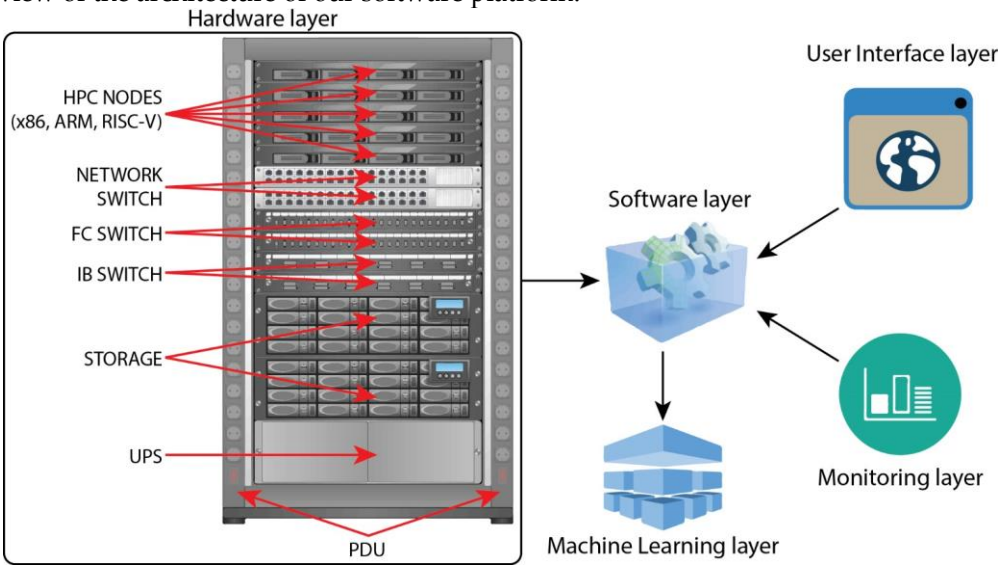

Figure 8. Proposed high-level architecture for using Kubernetes with HPC applications.

Let's now describe this new architecture and the layers it consists of.

A. Hardware Stack/Layer

The hardware stack consists of the regular components – PDUs, rack servers, network, InfiniBand and storage switches, storage, and the necessary hardware accelerators extensively used nowadays in modern HPC environments – ASICs, FPGAs, NVIDIA GPUs, etc. No change is needed in the hardware aspect from the regular design patterns, but we will significantly change how we use them via the software layer. 2U and 4U servers offer the best hardware upgradeability via various PCI Express cards, unlike blade servers or most HCI (Hyperconverged Infrastructure) servers, as they're often too limited in expandability and especially hardware compatibility.

B. Software Stack/Layer

The critical component of the proposed architecture is a software stack - a combination of provisioning virtual machine and container-based web application that we use to manage our HPC environment. This software stack is deeply intertwined with hardware in the HPC data center from the ground up, as this is one of the topics where we find that Kubernetes is severely lacking. Kubernetes's forte is the orchestration of containers without paying much attention to what's

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 June 2024 doi:10.20944/preprints202406.0074.v1

happening below, which is insufficient for what we need in HPC environments. We integrate PDU socket-to-server, storage, network, and network topology mapping into its hardware layer functions. We can see node monitoring in our platform in Figure 9.:

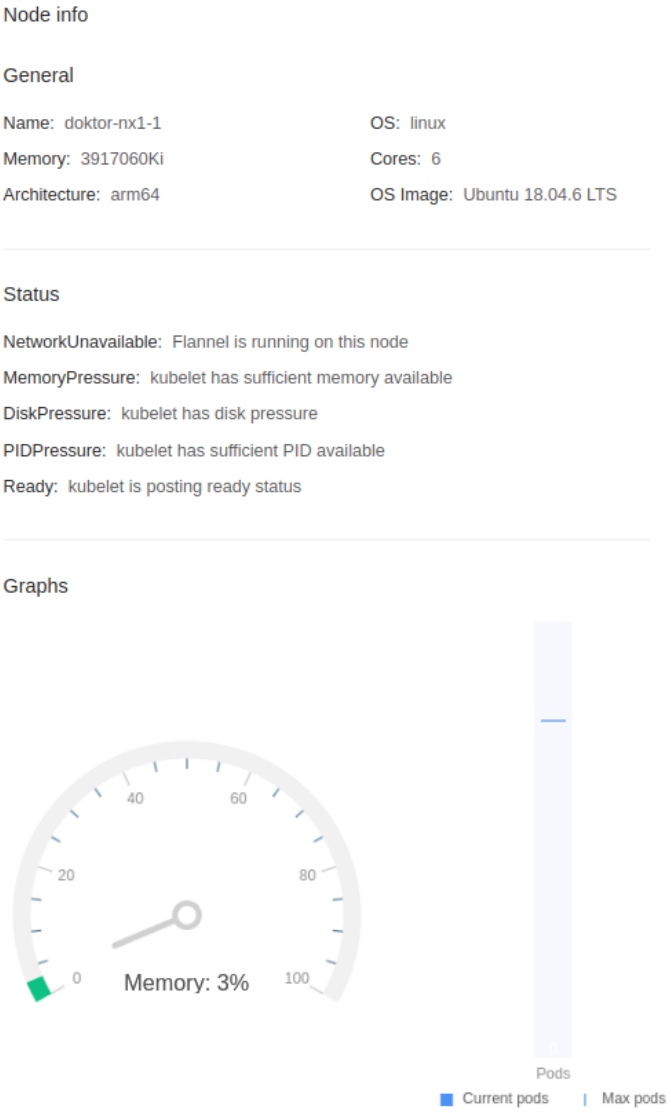

Figure 9. Node monitoring in our platform.

Our platform can use simple wizards to integrate existing Docker/Kubernetes environments, whether hosted locally or in the cloud. It can also do a complete deployment process for supported architectures (x86_64, ARM, and RISC-V) extended with heterogeneous compute components and do initial pre-tests to gauge synthetic performance as one of the factors for scheduling decisions. The reason for including this feature is that deploying container technologies like Docker directly for use in an HPC environment is neither trivial nor practical [32], especially on non-x86 platforms. Future iterations of the scheduling process use this information and the information about HPC applications we run in our containers to learn about our environment by inserting ML into scheduling decisions.

The software layer can also scan the underlying hardware to map out its features – CPU models, amount of memory, power supplies and usage, network connections, PCI Express accelerator devices (FPGAs, ASICs, GPUs, controllers), sensors, coolers, etc. It can also monitor the health state of available server components, a capability provided by server remote management options. The software platform then uses this data to learn about the environment via its automated backend and feeds it to ML algorithms. This is necessary as hardware capabilities and health states should be important factors when assessing where to place the HPC workload via operational risk analysis. This data must be fed to the ML engine to refine its scheduling recommendation to the Kubernetes

 doi:10.20944/preprints202406.0074.v1

orchestration platform, resulting in better workload placement. This deep hardware and software integration is needed because it yields much better long-term workload placement options that must be easily assigned via user-friendly UI. As a result, using this deeply integrated architecture, we solve the default Kubernetes scheduler's lack of finesse and programmability without explicitly developing sets of Kubernetes schedulers for different HPC scenarios and then attach them to Kubernetes pods, which is what we would be forced to do otherwise. There's an excellent example available on the Kubernetes documentation site for multiple scheduler configuration, available at https://kubernetes.io/docs/tasks/extend-kubernetes/configure-multiple-schedulers. It's a fantastic resource to fully grasp the complexity and level of Kubernetes knowledge needed to create it. Of course, we would have to make these custom schedulers per HPC app, which would also translate to statically assigning an HPC app to a specific set of nodes in a cluster if we don't have servers with the same capabilities (same GPU, same FPGA, same ASIC...). On top of the code complexity, this also becomes difficult and complex to manage operationally.

Our platform allows AI engines to suggest where to deploy HPC workloads. Still, we can also override that and explicitly place workloads on any of the Kubernetes nodes managed by our platform. In the workload deployment wizard, we can give our workload a custom name, assign it an image from the registry, and input/output storage location in case we need input and output data storage, as can be seen in Figure 10.:

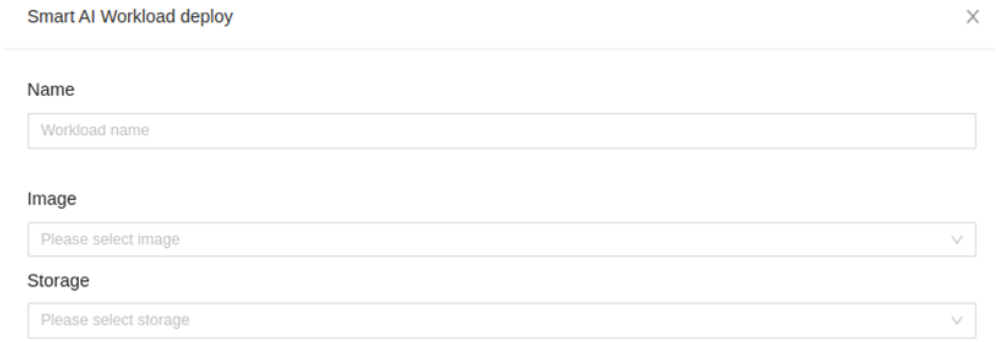

Figure 10. Easy deployment for any docker image in our Harbor registry.

As a part of that process, we also have an approval process when somebody requests manual placement, which can bring about necessary disputes about platform usage. A person in charge of the environment then gets to approve such a request or queue the workload in a fully automated, ML-backed way. Our software layer can also use the built-in QoS mechanisms in Kubernetes to solve some common Kubernetes issues, like poor Resource isolation between containers in Kubernetes [33]. One of these mechanisms is Kubernetes Memory Manager, which is NUMA-aware and will help latency and performance. We can use resource limits on the Kubernetes level (for CPU cores, for example). As mentioned in the previous section, we currently cannot solve direct hardware presentation in containers for all hardware devices, depending on the acceleration technology used (GPU, FPGA, ASIC), but also for more regular hardware, like network cards and storage controllers. Docker, Podman, and Kubernetes are still developing in this direction as this is where the market is pushing them. Until these problems are resolved, there will be workloads that are better suited to virtual machines than containers. That stems from Docker, Podman, and Kubernetes's original intent– to use CPU and memory as the primary computing fabric, not focusing on additional hardware or accelerators. Consequently, Kubernetes sets two resources for each pod: the central processing unit (CPU) and memory [34].

C. Machine Learning Layer

As a new and integral part of our software stack, we integrated multiple ML algorithms to do real-life research on which algorithm yields the most optimal scheduling and placement results. We deliberately chose to incorporate multiple algorithms to track the progress of these algorithms in terms of their learning and precision on the path of yielding the best scheduling recommendations.

 doi:10.20944/preprints202406.0074.v1

The best recommendations in our use case don't necessarily mean only the fastest computing task – this can be brought into context with the amount of power used to do that computing task (efficiency). From the get-go, our platform to manage HPC workloads on Kubernetes needs to be as efficient and *green* as possible while meeting the computational requirements in terms of performance, as wasting resources should never be the design goal of any IT system. This is where deep integration between software and hardware comes into play, making scheduling workloads and monitoring environments much more effortless. Furthermore, with health information always available via remote management interfaces, it also makes it easier to calculate scores to determine optimum placement and have alarms available when something on the hardware layer goes wrong. A scoring system we will discuss in this paper is entirely independent of the Kubernetes scheduler's scoring system, as it lacks finesse for all the different use cases we might encounter in the HPC space. We can see how we implemented ML into our UI in Figure 11.:

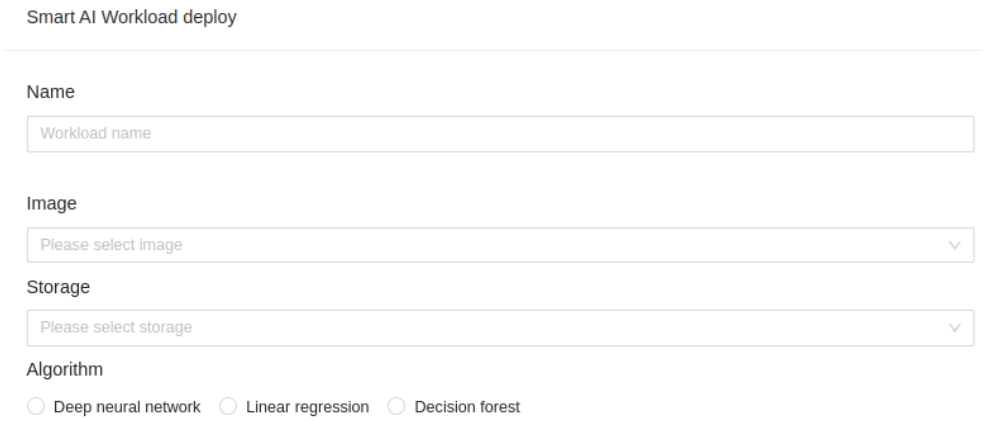

Figure 11. Multiple options are available for ML analysis for workload placement.

Furthermore, our software stack needs to be able to override ML suggestions as we can't start using ML scheduling from the start – we need to give ML time to learn about the environment and applications first. After all, overriding ML suggestions controls our impulses to always put some workloads first at the expense of others. ML algorithms are a valuable tool here as there are QoS parameters that we need to keep in mind when working with various HPC applications – especially on the latency and bandwidth level, depending on the application. We cannot approach this topic from the perspective of just using something simple, such as using response time from the system as the main QoS to be maintained [35].

We used three distinctive ML types to compare workload scheduling and placement recommendations thoroughly without bias. That way, we had multiple data sets to compare placement decisions as we developed our platform. Specifically, we used DNN (Deep Neural Network), DF (Decision Forest), and LR (Linear Regression). We integrated them into the UI of our platform so that we could quickly get to the corresponding scores.

D. User Interface Layer

The UI-based part of the application needs to be usable by a regular user, not an IT expert. Therefore, it needs to combine the simplicity of drag-and-drop wizards with the complexity of Docker and Podman in the backend. We also created a prompt-based interface (custom Discord chatbot application) to interact with the backend ML engine to check the state of our environment or schedule workloads, even if a scientist who wants to start an HPC workload isn't sitting at the computer. This makes it very straightforward to schedule/manage workloads without spending hours defining how and where these tasks should be started. In Figure 12., we can see information about the current state of our environment provided by the chatbot:

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 June 2024 doi:10.20944/preprints202406.0074.v1

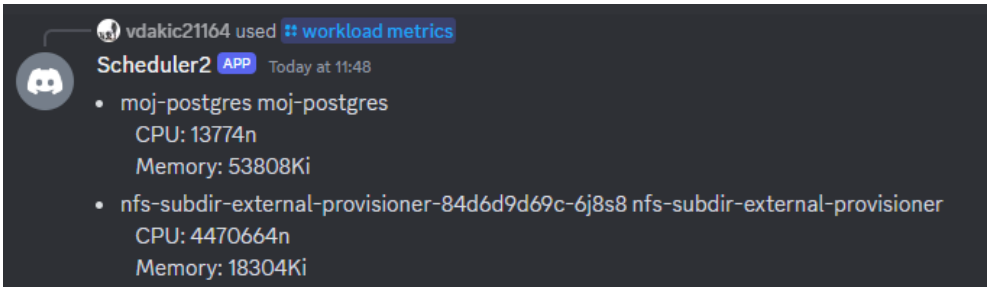

Figure 12. Discord chatbot provides information about workload metrics, starts workloads, stops workloads, etc.

One of the biggest challenges in making such a user-friendly interface is that we need to quickly package various HPC applications in containers without writing many YAML configuration files. Our platform implemented these features via multi-arch base containers and multi-arch applications, presented as Docker image layers, that can be imported as files or by working with Harbor Registry. Harbor is an open-source registry that can be used to store, distribute, sign, and scan container images. Multi-arch base containers (Ubuntu, CentOS, or Debian for x86, ARM, or RISC-V) are a good choice as they are small enough not to waste resources while still supported by most HPC applications. In HPC environments based on containers, it is essential to have the capability to run the HPC application instead of insisting on using the most common small container Linux distributions (for example, Alpine Linux), as these require additional Linux packages [36]. That also means we had to pre-build our base containers with specific requirements tailored to HPC environments. For example, they must support high-performance libraries like MPI and CUDA for NVIDIA GPU-based applications, etc. The HPC application container will include the application and its dependencies, such as the MPI library, accelerator libraries and tools, and parallel math libraries [37]. Then, we have a drag-and-drop system to automate container building with additional layers for HPC applications. This way, we avoid writing YAML files and forcing users to build container images manually. We should use the same design principle for storage and network integration – we integrate storage and network at the platform level and use storage/networking as input and output variables in the wizard so that they can be easily re-used by Kubernetes API when making API calls. This also enables us to integrate principles of micro-segmentation to reduce attack surfaces on our HPC applications and secure them as much as possible.

We can see the wizard-driven approach that we took for adding storage in Figure 13., as well as the storage types supported:

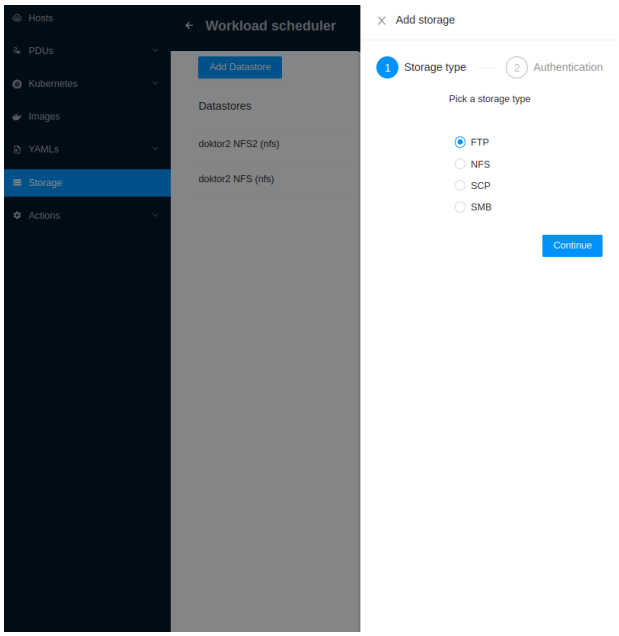

Figure 13. Add storage wizard in our platform.

 doi:10.20944/preprints202406.0074.v1

This simplifies adding storage so users can focus on actual work, i.e., placing workloads on Kubernetes clusters. Furthermore, there's no need to use many commands and additional parameters to do the same thing or waste time learning them.

E. Monitoring Layer

Apart from hardware monitoring for the scoring system, we also need to efficiently monitor what's happening in our environment from the management perspective, i.e., via dashboards and visual data representation. This is where integration with the backend Clickhouse database makes a lot of sense, as it's an extensive, column-sorted analytics database. The increasing complexity of future computing systems requires monitoring and management tools to provide a unified view of the center, perfect for understanding the hardware and general performance aspects of an extreme cluster or high-performance system [38]. As an example, we can see how we integrated PDU monitoring in our platform in Figure 14.:

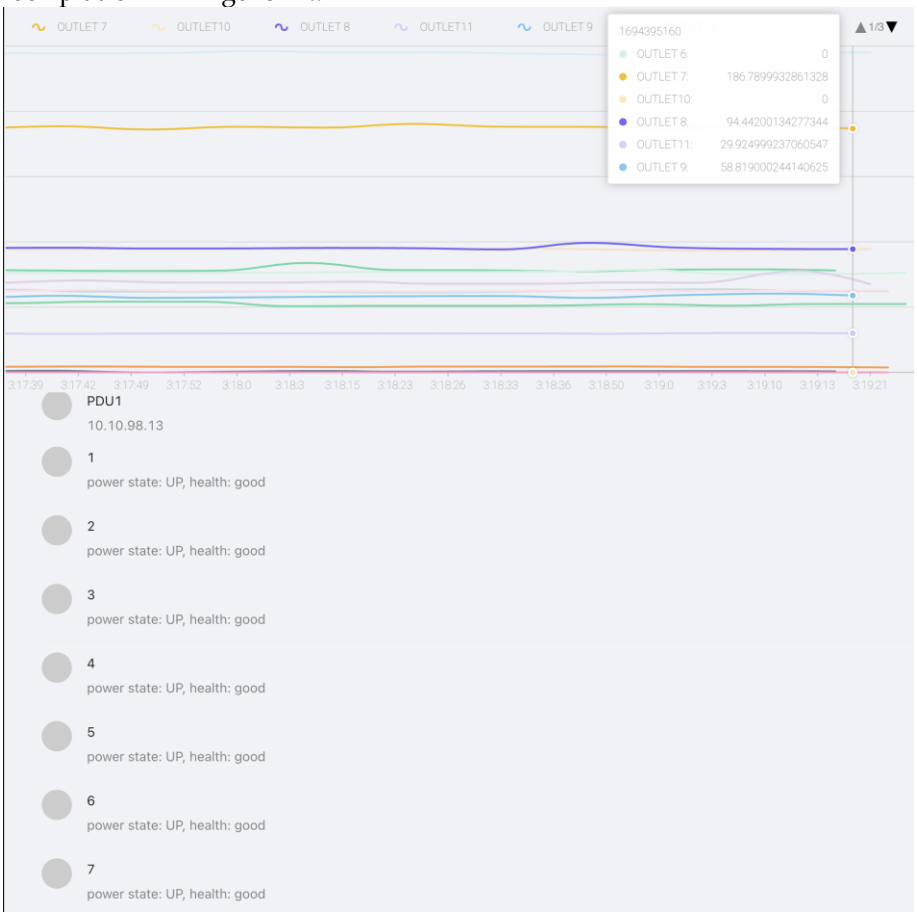

Figure 14. PDU monitoring in our platform.

This also allows us to prototype and embed additional graphs quickly, should the need arise. We can easily collect a large amount of data that needs to be processed [39].

7. Scheduling of HPC Workloads on our Platform via ML or Manual Placement

In the previous section, we mentioned problems with the Kubernetes scheduler, the complexity of creating a custom one, and how our platform avoids that process altogether. We achieve this by allowing the AI engine to explicitly start an HPC application packaged as a container on the node that it deems the most suitable for it if we choose to do so. The complex process creates a container image and its corresponding layers to package the application. We wanted to do away with the complexity of creating images, as this will significantly impact how we resolve software dependencies in HPC applications, as resolving lifecycle dependencies at the container orchestration

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 June 2024 doi:10.20944/preprints202406.0074.v1

level adds additional overhead [40]. The way we solve these problems is by using a wizard-driven process based on a drag-and-drop canvas where we can drop storage locations, base containers, HPC applications, and networks – generally speaking, all of the necessary details to create a workload with all of the settings required to run on a Kubernetes cluster. That means that, during this process, we go through four phases:

These four phases in our wizard ensure that we have all the necessary details to start the workload and all the prerequisites for it to run successfully.

There are subtle differences in manual workload placement compared to ML-based placement, so let's delve deeper into that.

7.1. Manual Workload Placement

Manual workload placement is simple - from the list of available nodes, we ignore everything that ML provides to us and click on the "Deploy" button on the right side of the wizard for the node that we want to select for workload deployment, as we can see in Figure 15.:

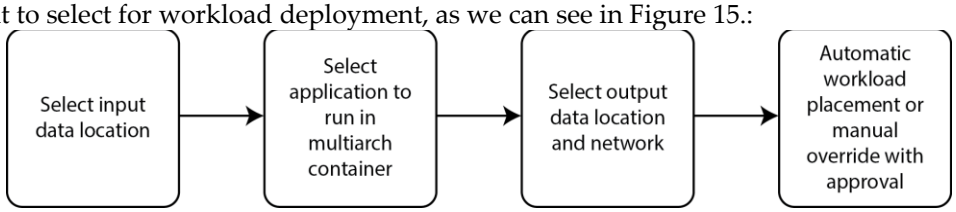

Figure 15. Workload placement in Kubernetes when using our proposed platform.

This means that if we - for whatever reason - want to manually select a node chosen by the ML and presented via scores (to be discussed in the following sub-section), we can easily do it. We can also see that scores are visible in the UI no matter which deployment model we choose. Manual workload placement might be beneficial if we have a set of HPC nodes reserved for some emergency workload or if we're starting to use some new HPC application and want to gauge its performance.

7.2. ML-Based Workload Placement

ML-based workload placement in our platform is based on a set of parameters that we detected as crucial for placement decisions to be as accurate and reliable as possible. Whichever workload placement methodology we choose, our application consults with the ML algorithms in the backend before making scheduling decisions. ML engine considers the parameter history on a per-app, per-server basis. There are five sets of parameters:

- **POWER usage** - A set of timestamped power readings from the PDU socket and server remote management, sampled at configurable intervals. These readings estimate HPC application power requirements, significantly improving the platform's energy efficiency, especially given time and many workload executions to become even more accurate.
- **HEALTH information**—A set of parameters taken from server remote management handling the health states of components, specifically fans, memory health state, power supply health state, power state (redundant or not), processor health state, storage health state, network, and remote management health state and temperatures.
- **CPU, memory, storage, and networking testing results**—As the server is provisioned from our platform in multiple passes, pre-determined synthetic and real-life benchmarks are automatically performed and averaged across the configurable number of runs. These bare-metal and containerized benchmarks determine the baseline hardware performance level for all servers added to the system. We use sysbench, stress-ng, hdparm, HPL, HPCC, and HPL with various parameters to gauge performance in single and multi-tasking scenarios.
- **NVIDIA GPU results**—A set of pre-determined synthetic and real-life benchmarks is automatically performed. The server with the installed NVIDIA GPU is provisioned from our platform in multiple takes, averaged across the configurable number of runs. We use NVIDIA HPC-Benchmarks for this purpose.

 doi:10.20944/preprints202406.0074.v1

- **FPGA/ASIC availability** - For supported FPGA/ASIC controllers, platform users can manually add additional scores per app. Ideally, this would be automated, but it's currently impossible because of the different software stacks used by FPGAs and ASICs.

All scores are taken per application, not as an aggregate, and our scoring system takes current usage as a parameter into consideration, as well. For example, to avoid operational risk, if two servers have the same NVIDIA GPU, one of those GPUs is being used less at a certain point in time, and our ML scheduler automatically picks the server with less GPU used by other apps. This enables us to have individualized, per-app scores that are much more relevant as we develop the platform for general use cases that support even more hardware. The numbered scoring system considers all of these parameters but also dispatches negative grades if there are different types of failures. For example, if one of the power supplies on the server fails, that server gets a score downgrade. Simplified, the algorithm works like this:

All of the mentioned parameters have weights that can be used as preset values (provided by the app itself, as fixed values) or changed by the platform manager by editing values assigned to any of these parameters. That, in turn, changes the server's overall per-application score and the relative score related to the set of parameters we changed manually. This seems like a reasonable compromise between automation and the capability to override the system, as that option always needs to exist.

Hardware tests are executed post-deployment before any workloads can be placed via our platform. Ansible playbooks in our deployment process automatically install and perform all necessary tests (multiple runs). Scores are saved per node in a CSV file, which is then added to the Clickhouse database and uploaded to ML. ML engines always have up-to-date data when we expand our environment.

All workload placement processes via our platform are also "captured" by ML using the same method as the initial one, enabling us to track performance and load on our servers constantly. By correlating power usage and health info data, ML engines also have up-to-date scores for every server added to our platform. As we saw in Figure 16., scores are automatically calculated and added next to each node's name, making it easier for users to track the most recent scores. If we don't manually select any specific node, these scores are used as an a priori recommendation system that automatically places the workload on a node with the highest score.

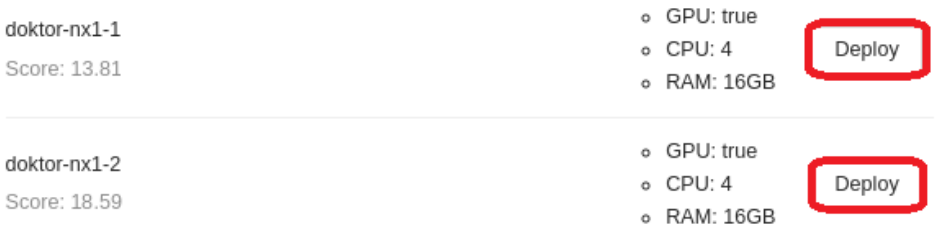

Figure 16. Manual workload placement in our platform.

 doi:10.20944/preprints202406.0074.v1

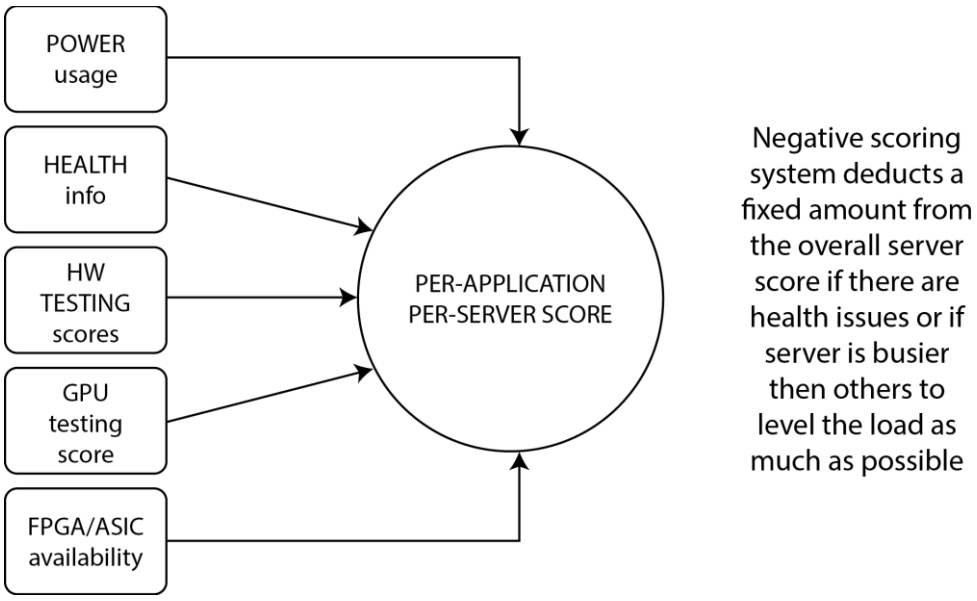

Figure 17. Explanation of the scoring system that influences ML decision-making process for workload scheduling.

8. Future Work

This paper presents our first steps towards exploiting virtualization and containerization technology in specifics of heterogeneous HPC environments. Multiple routes exist for developing our proposed platform to handle HPC workloads via Kubernetes. First and foremost, we mentioned that there are and likely will be workloads for which virtual machines are much better suited than containers, at least for now. That means that some integration with a purely hypervisor-based virtualized platform is a good development path for the future. Specifically, it needs to be as automated and easy to use as the platform we described and integrate various technologies for various operating systems and solutions. For example, cloud-init and cloudbase-init could customize virtual machines post-deployment from a template. After that, PowerShell and Ansible could be used to deploy and configure the necessary software components. With the current state of technology, it wouldn't be all that difficult to integrate that type of platform with some more advanced settings that are currently not viable solutions with Kubernetes, like direct hardware presentation via PCIe passthrough or something like SR-IOV, to partition PCIe cards into multiple virtual functions and provide them automatically to virtual machines. That would make it much easier to use FPGAs and ASICs that need to be fixed with Kubernetes (unfortunately, that list is quite long). We could then develop an ML-based scheduler that's application-aware to the point of spooling up either containers or virtual machines, depending on the available hardware, accelerators, applications, and their requirements. Furthermore, even more developed LLM-based (Large Language Model) management via Discord bot could become even more helpful for migrating workloads or application states from one node to another.

9. Conclusions

This paper proposes a novel platform to manage Kubernetes-based environments for HPC. It also covers a broad range of topics related to containerization, virtualization, orchestration, and HPC. When discussing the specifics of HPC, we concluded that using Kubernetes and containers for HPC workloads is complex and needs to be simplified as much as possible. This is why we developed a new platform with proactive and reactive components. It can proactively assign specific performance numbers to our HPC data center servers and reactively help place HPC workloads on them. We concluded that developing custom Kubernetes schedulers on a per-HPC application basis isn't the way to go. Instead, we created a platform that uses ML to load workloads automatically and manually. This approach took a lot of integration of various components - PDUs to track energy

 doi:10.20944/preprints202406.0074.v1

usage, remote management interfaces to track servers' health state, etc. All that data is fed to the ML engine to predict and suggest which Kubernetes node in our cluster is the optimal node to place that workload. Our study shows a lot of room for improvement in how we place our workloads in Kubernetes (in general), especially in HPC environments. Our platform offers a simple opportunity to go down that path without much additional effort, which was what we set up to do when we started this research.

References

1. 1. Đorđević, B.; Kraljević, N.; Davidović, N. Performance Comparison of CPU Hardware-Assisted Features for the Type-2 Hypervisors. 2024 23rd International Symposium INFOTEH-JAHORINA (INFOTEH) 2024. doi: 10.1109/INFOTEH60418.2024.10495932.
2. Chen, Y.-R.; Liu, I.-H.; Chou, C.-W.; Li, J.-S.; Liu, C.-G. Multiple Virtual Machines Live Migration Scheduling Method Study on VMware vMotion. 2018 3rd International Conference on Computer and Communication Systems (ICCCS) 2018. doi: 10.1109/CCOMS.2018.8463330.
3. Shirinbab, S.; Lundberg, L.; Hakansson, J. Comparing Automatic Load Balancing Using VMware DRS with a Human Expert. 2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW) 2016. doi: 10.1109/IC2EW.2016.14.
4. Li, Z.; Kihl, M.; Lu, Q.; Andersson, J.A. Performance Overhead Comparison between Hypervisor and Container Based Virtualization. 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA) 2017. doi: 10.1109/AINA.2017.79.
5. Lingayat, A.; Badre, R.R.; Kumar Gupta, A. Performance Evaluation for Deploying Docker Containers On Baremetal and Virtual Machine. 2018 3rd International Conference on Communication and Electronics Systems (ICCES) 2018. doi:10.1109/CESYS.2018.8723998.
6. Agarwal, K.; Jain, B.; Porter, D.E. Containing the Hype. Proceedings of the 6th Asia-Pacific Workshop on Systems 2015. doi:10.1145/2797022.2797029.
7. Antunes, C.; Vardasca, R. Performance of Jails versus Virtualization for Cloud Computing Solutions. Procedia Technology 2014, 16, 649–658. doi: 10.1016/j.protcy.2014.10.013.
8. Trigo, A.; Varajão, J.; Sousa, L. DevOps Adoption: Insights from a Large European Telco. Cogent Engineering 2022, 9. doi: 10.1080/23311916.2022.2083474.
9. Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., & Peterson, L. (2007). Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. ACM SIGOPS Operating Systems Review, 41(3), 275-287. DOI: 10.1145/1272996.1273025.
10. Xuefei Li, & Jingyue Jiang. (2016). Performance Analysis of PaaS Cloud Resources Management Model Based on LXC. In Proceedings of the 2016 International Conference on Cloud Computing and Internet of Things (CCIOT), 118-130. DOI: 10.1007/978-981-10-3966-9_13.
11. Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2015). An Updated Performance Comparison of Virtual Machines and Linux Containers. 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 171-172. DOI: 10.1109/ISPASS.2015.7095802.
12. Burns, B.; Grant, B.; Oppenheimer, D.; Brewer, E.; Wilkes, J. Borg, Omega, and Kubernetes. Queue 2016, 14, 70–93. doi: 10.1145/2898442.2898444.
13. Dordevic, B.; Timcenko, V.; Lazic, M.; Davidovic, N. Performance Comparison of Docker and Podman Container-Based Virtualization. 2022 21st International Symposium INFOTEH-JAHORINA (INFOTEH) 2022. doi: 10.1109/INFOTEH53737.2022.9751277.
14. Gantikow, H.; Walter, S.; Reich, C. Rootless Containers with Podman for HPC. Lecture Notes in Computer Science 2020, 343–354. doi: 10.1007/978-3-030-59851-8_23.
15. Sheka, A.; Bersenev, A.; Samun, V. Containerization in Scientific Calculations. 2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON) 2019. doi: 10.1109/SIBIRCON48586.2019.8958324.Voulgaris, K.;
16. Kiourtis, A.; Karabetian, A.; Karamolegkos, P.; Poulakis, Y.; Mavrogiorgou, A.; Kyriazis, D. A Comparison of Container Systems for Machine Learning Scenarios: Docker and Podman. 2022 2nd International Conference on Computers and Automation (CompAuto) 2022. doi: 10.1109/CompAuto55930.2022.00029.
17. Stephey, L.; Canon, S.; Gaur, A.; Fulton, D.; Younge, A.J. Scaling Podman on Perlmutter: Embracing a Community-Supported Container Ecosystem. 2022 IEEE/ACM 4th International Workshop on Containers

and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC) 2022. doi: 10.1109/CANOPIE-HPC56864.2022.00008

18. Greneche, N.; Cerin, C. Autoscaling of Containerized HPC Clusters in the Cloud. 2022 IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies (SuperCompCloud) 2022; pp. 1–7. doi: 10.1109/SuperCompCloud56703.2022.00006.
19. Liu, P.; Guitart, J. Fine-Grained Scheduling for Containerized HPC Workloads in Kubernetes Clusters. Available online: http://arxiv.org/abs/2211.11487 (accessed on 22.05.2024).
20. Beltre, A.M.; Saha, P.; Govindaraju, M.; Younge, A.; Grant, R.E. Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms. 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC) 2019; pp. 11–20. doi: 10.1109/CANOPIE-HPC49598.2019.00007.
21. Rathmayer, S.; Lenke, M. A Tool for On-Line Visualization and Interactive Steering of Parallel HPC Applications. Proceedings 11th International Parallel Processing Symposium. doi: 10.1109/IPPS.1997.580882
22. Gomez, C.; Martinez, F.; Armejach, A.; Moreto, M.; Mantovani, F.; Casas, M. Design Space Exploration of Next-Generation HPC Machines. 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2019. doi: 10.1109/IPDPS.2019.00017
23. Wang, P.; Posey, S. GPU Best Practices for HPC Applications at Industry Scale. Lecture Notes in Earth System Sciences 2013, 163–172. doi: 10.1007/978-3-642-16405-7_9
24. Nonaka, J.; Ono, K.; Fujita, M. 234Compositor: A Flexible Parallel Image Compositing Framework for Massively Parallel Visualization Environments. Future Generation Computer Systems 2018, 82, 647–655. doi: 10.1016/j.future.2017.02.011
25. Vu, D.-D.; Tran, M.-N.; Kim, Y. Predictive Hybrid Autoscaling for Containerized Applications. IEEE Access 2022, 10, 109768–109778. doi: 10.1109/ACCESS.2022.3214985.
26. Tesliuk, A.; Bobkov, S.; Ilyin, V.; Novikov, A.; Poyda, A.; Velikhov, V. Kubernetes Container Orchestration as a Framework for Flexible and Effective Scientific Data Analysis. 2019 Ivannikov Ispras Open Conference (ISPRAS) 2019; pp. 67–71. doi: 10.1109/ISPRAS47671.2019.00016.
27. Flora, J.; Goncalves, P.; Teixeira, M.; Antunes, N. A Study on the Aging and Fault Tolerance of Microservices in Kubernetes. IEEE Access 2022, 10, 132786–132799. doi: 10.1109/ACCESS.2022.3231191.
28. Milroy, D.J.; Misale, C.; Georgakoudis, G.; Elengikal, T.; Sarkar, A.; Drocco, M.; Patki, T.; Yeom, J.-S.; Gutierrez, C.E.A.; Ahn, D.H.; et al. One Step Closer to Converged Computing: Achieving Scalability with Cloud-Native HPC. 2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC) 2022; pp. 57–70. doi: 10.1109/CANOPIE-HPC56864.2022.00011.
29. Zhou, N.; Georgiou, Y.; Zhong, L.; Zhou, H.; Pospieszny, M. Container Orchestration on HPC Systems. 2020 IEEE 13th International Conference on Cloud Computing (CLOUD) 2020; pp. 34–36. doi: 10.1109/CLOUD49709.2020.00017.
30. Lublinsky, B.; Jennings, E.; Spišaková, V. A Kubernetes 'Bridge' Operator between Cloud and External Resources. 2023 8th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA) 2023; pp. 263–269. doi: 10.1109/ICCCBDA56900.2023.10154770.
31. Spišaková, V.; Klusáček, D.; Hejtmánek, L. Using Kubernetes in Academic Environment: Problems and Approaches (Open Scheduling Problem). Available online: https://jsspp.org/papers22/6.pdf (accessed on 22.05.2024).
32. Younge, A.J.; Pedretti, K.; Grant, R.E.; Brightwell, R. A Tale of Two Systems: Using Containers to Deploy HPC Applications on Supercomputers and Clouds. 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom) 2017; pp. 74–81. doi: 10.1109/CloudCom.2017.40.
33. Zhang, X.; Li, L.; Wang, Y.; Chen, E.; Shou, L. Zeus: Improving Resource Efficiency via Workload Colocation for Massive Kubernetes Clusters. IEEE Access 2021, 9, 105192–105204. doi: 10.1109/ACCESS.2021.3100082.
34. Ding, Z.; Wang, S.; Jiang, C. Kubernetes-Oriented Microservice Placement With Dynamic Resource Allocation. IEEE Trans. Cloud Comput. 2023, 11, 1777–1793. doi: 10.1109/tcc.2022.3161900.
35. Khaleq, A.A.; Ra, I. Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications. IEEE Access 2021, 9, 35464–35476. doi: 10.1109/ACCESS.2021.3061890.

36. Ruiz, L.M.; Pueyo, P.P.; Mateo-Fornes, J.; Mayoral, J.V.; Tehas, F.S. Autoscaling Pods on an On-Premise Kubernetes Infrastructure QoS-Aware. IEEE Access 2022, 10, 33083–33094. doi: 10.1109/ACCESS.2022.3158743.
37. Hursey, J. Design Considerations for Building and Running Containerized MPI Applications. 2020 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC) 2020; doi: 10.1109/CANOPIEHPC51917.2020.00010.
38. Sukhija, N.; Bautista, E. Towards a Framework for Monitoring and Analyzing High Performance Computing Environments Using Kubernetes and Prometheus. 2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI) 2019; pp. 257–262. doi: 10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00087.
39. Kosinska, J.; Tobiasz, M. Detection of Cluster Anomalies With ML Techniques. IEEE Access 2022, 10, 110742–110753, doi: 10.1109/ACCESS.2022.3216080.
40. Sebrechts, M.; Borny, S.; Wauters, T.; Volckaert, B.; De Turck, F. Service Relationship Orchestration: Lessons Learned From Running Large Scale Smart City Platforms on Kubernetes. IEEE Access 2021, 9, 133387–133401. doi: 10.1109/ACCESS.2021.3115438.
41. Vasireddy, I.; Ramya, G.; Kandi, P. Kubernetes and Docker Load Balancing: State-of-the-Art Techniques and Challenges. IJIREM 2023, 10, 49–54. doi: 10.55524/ijirem.2023.10.6.7.
42. Vohra, D. Installing Kubernetes Using Docker. Kubernetes Microservices with Docker 2016, 3–38. doi: 10.1007/978-1-4842-1907-2_1.
43. Bernstein, D. Containers and Cloud: From LXC to Docker to Kubernetes. IEEE Cloud Comput. 2014, 1, 81–84. doi: 10.1109/MCC.2014.51.
44. Liu, B.; Li, J.; Lin, W.; Bai, W.; Li, P.; Gao, Q. K-PSO: An Improved PSO-based Container Scheduling Algorithm for Big Data Applications. Int J Network Mgmt 2020, 31. doi: 10.1002/nem.2092.
45. Vasireddy, I.; Ramya, G.; Kandi, P. Kubernetes and Docker Load Balancing: State-of-the-Art Techniques and Challenges. IJIREM 2023, 10, 49–54. doi: 10.55524/ijirem.2023.10.6.7.
46. Pan, Y.; Chen, I.; Brasileiro, F.; Jayaputera, G.; Sinnott, R. A Performance Comparison of Cloud-Based Container Orchestration Tools. 2019 IEEE International Conference on Big Knowledge (ICBK) 2019. doi: 10.1109/ICBK.2019.00033.
47. Beltre, A.M.; Saha, P.; Govindaraju, M.; Younge, A.; Grant, R.E. Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms. 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC) 2019. doi: 10.1109/CANOPIE-HPC49598.2019.00007.
48. Vohra, D. Installing Kubernetes Using Docker. Kubernetes Microservices with Docker 2016, 3–38. doi: 10.1007/978-1-4842-1907-2_1.
49. Bernstein, D. Containers and Cloud: From LXC to Docker to Kubernetes. IEEE Cloud Comput. 2014, 1, 81–84. doi: 10.1109/MCC.2014.51.
50. Vasireddy, I.; Ramya, G.; Kandi, P. Kubernetes and Docker Load Balancing: State-of-the-Art Techniques and Challenges. IJIREM 2023, 10, 49–54. doi: 10.55524/ijirem.2023.10.6.7.
51. Lossent, A.; Rodriguez Peon, A.; Wagner, A. PaaS for Web Applications with OpenShift Origin. J. Phys.: Conf. Ser. 2017, 898, 082037. doi: 10.1088/1742-6596/898/8/082037.
52. Aly, M.; Khomh, F.; Yacout, S. Kubernetes or OpenShift? Which Technology Best Suits Eclipse Hono IoT Deployments. 2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA) 2018. doi: 10.1109/SOCA.2018.00024.
53. Linzel, B.; Zhu, E.; Flores, G.; Liu, J.; Dikaleh, S. How can OpenShift accelerate your Kubernetes adoption: a workshop exploring OpenShift features. CASCON '19: Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, 380-381. doi: 10.5555/3370272.3370326.
54. Vohra, D. Using an HA Master with OpenShift. Kubernetes Management Design Patterns 2017, 335–353. doi: 10.1007/978-1-4842-2598-1_15.
55. Marksteiner, P. High-Performance Computing — an Overview. Computer Physics Communications 1996, 97, 16–35. doi: 10.1016/0010-4655(96)00018-5.
56. Cardoso, J.M.P.; Coutinho, J.G.F.; Diniz, P.C. High-Performance Embedded Computing. Embedded Computing for High Performance 2017, 17–56. doi: 10.1016/B978-0-12-804189-5.00002-8.

57. Feng, W.; Manocha, D. High-Performance Computing Using Accelerators. Parallel Computing 2007, 33, 645–647. doi: 10.1016/j.parco.2007.10.001.
58. Kindratenko, V.; Thiruvathukal, G.K.; Gottlieb, S. High-Performance Computing Applications on Novel Architectures. Comput. Sci. Eng. 2008, 10, 13–15. doi: 10.1109/MCSE.2008.149.
59. Feng, W.; Manocha, D. High-Performance Computing Using Accelerators. Parallel Computing 2007, 33, 645–647. doi: 10.1016/j.parco.2007.10.001.
60. Lee, G. High-Performance Computing Networks. Cloud Networking 2014, 179–189. doi: 10.1016/B978-0-12-800728-0.00010-2.
61. Smith, M.C.; Drager, S.L.; Pochet, L.; Peterson, G.D. High Performance Reconfigurable Computing Systems. Proceedings of the 44th IEEE 2001 Midwest Symposium on Circuits and Systems. MWSCAS 2001 (Cat. No.01CH37257). doi: 10.1109/MWSCAS.2001.986212.
62. Levesque, J.; Wagenbreth, G. High Performance Computing; Chapman and Hall/CRC, 2010. doi: 10.1201/b10442.
63. Souppaya, M.; Morello, J.; Scarfone, K. Application Container Security Guide; National Institute of Standards and Technology, 2017. doi:10.6028/NIST.SP.800-190.
64. Ambrosino, G.; Fioccola, G.B.; Canonico, R.; Ventre, G. Container Mapping and Its Impact on Performance in Containerized Cloud Environments. 2020 IEEE International Conference on Service Oriented Systems Engineering (SOSE) 2020. doi:10.1109/SOSE49046.2020.00014.
65. Senjab, K.; Abbas, S.; Ahmed, N.; Khan, A. ur R. A Survey of Kubernetes Scheduling Algorithms. J Cloud Comp 2023, 12. doi:10.1186/s13677-023-00471-1
66. Carrión, C. Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges. ACM Comput. Surv. 2022, 55, 1–37. doi: 10.1145/3539606.
67. Rodriguez, G.; Yannibelli, V.; Rocha, F.G.; Barbara, D.; Azevedo, I.M.; Menezes, P.M. Understanding and Addressing the Allocation of Microservices into Containers: A Review. IETE Journal of Research 2023, 1–14. doi: 10.1080/03772063.2023.2205864.
68. Roslin Dayana, K.; Shobha Rani, P. Secure Cloud Data Storage Solution with Better Data Accessibility and Time Efficiency. Automatika 2023, 64, 756–763. doi: 10.1080/00051144.2023.2213564.
69. Jagadeeswari, N.; Mohanraj, V.; Suresh, Y.; Senthilkumar, J. Optimization of Virtual Machines Performance Using Fuzzy Hashing and Genetic Algorithm-Based Memory Deduplication of Static Pages. Automatika 2023, 64, 868–877. doi: 10.1080/00051144.2023.2223479.
70. Yang, H.; Ong, S.K.; Nee, A.Y.C.; Jiang, G.; Mei, X. Microservices-Based Cloud-Edge Collaborative Condition Monitoring Platform for Smart Manufacturing Systems. International Journal of Production Research 2022, 60, 7492–7501. doi: 10.1080/00207543.2022.2098075.
71. Holmes, V.; Newall, M. HPC and the Big Data Challenge. Safety and Reliability 2016, 36, 213–224. doi:10.1080/09617353.2016.1252085.
72. Houzeaux, G.; Garcia-Gasulla, M. High Performance Computing Techniques in CFD. International Journal of Computational Fluid Dynamics 2020, 34, 457–457. doi: 10.1080/10618562.2020.1833151.
73. Örmecioğlu, T.O.; Aydoğdu, İ.; Örmecioğlu, H.T. GPU-Based Parallel Programming for FEM Analysis in the Optimization of Steel Frames. Journal of Asian Architecture and Building Engineering 2024, 1–22. doi: 10.1080/13467581.2024.2345310.
74. Jha, A.V.; Teri, R.; Verma, S.; Tarafder, S.; Bhowmik, W.; Kumar Mishra, S.; Appasani, B.; Srinivasulu, A.; Philibert, N. From Theory to Practice: Understanding DevOps Culture and Mindset. Cogent Engineering 2023, 10. doi: 10.1080/23311916.2023.2251758.
75. Li, H.; Kettinger, W.J.; Yoo, S. Dark Clouds on the Horizon? Effects of Cloud Storage on Security Breaches. Journal of Management Information Systems 2024, 41, 206–235. doi: 10.1080/23311916.2022.2083474.