

Zero to Jupyterhub with Kubernetes

Abstract

This paper presents a lightweight, storage-efficient architecture for deploying multi-user interactive computing environments tailored for data science and Big Data education. As academic institutions increasingly adopt containerized solutions, such as JupyterHub, to provide consistent and reproducible development environments, the storage overhead associated with redundant datasets and duplicated user libraries emerges as a critical infrastructure challenge. Traditional provisioning methods often lead to exponential storage growth as student numbers increase, creating significant hardware bottlenecks.

To address this, we propose and demonstrate a deployment strategy leveraging Kubernetes to orchestrate the application layer, coupled with an underlying ZFS storage layer for advanced data management. The implementation utilizes kind to permit rapid, flexible cluster instantiation on local infrastructure. The core contribution of this work is the architectural integration of the host's ZFS file system with the Kubernetes storage provisioner through direct host-path mounting.

By mapping a host-managed ZFS pool, specifically configured with block-level deduplication, directly into the Kubernetes control plane volume provisioner, the system achieves transparent data reduction. This approach is demonstrated to be highly effective in educational scenarios where student workspaces frequently contain identical large datasets, environment specifications, and software dependencies. We detail the Infrastructure-as-Code (IaC) methodology used to automate this deployment, utilizing shell scripts for cluster lifecycle management and Helm charts for application configuration. The resulting system offers a scalable, high-performance platform that maintains strict user isolation while significantly minimizing physical disk capacity requirements, providing a cost-effective model for hosting data-intensive academic coursework.

1. Introduction

The rapid expansion of data science and Big Data curricula has necessitated the adoption of robust, scalable computing platforms capable of serving large cohorts of students simultaneously. Tools like JupyterHub have become the de facto standard in academic settings, offering containerized, reproducible environments that eliminate the "it works on my machine" friction. However, as class sizes grow and coursework involves increasingly massive datasets, the underlying infrastructure faces significant strain. Providing persistent, isolated storage for hundreds of users quickly becomes a resource bottleneck, particularly when high-performance storage is costly or limited in on-premise educational clusters.

In this context, specific characteristics of student workloads exacerbate the storage challenge. Unlike general-purpose enterprise environments, educational clusters exhibit extreme data redundancy. A typical Big Data assignment requires every student to possess a personal copy of the same gigabyte-scale dataset, along with identical Python or Scala library dependencies within their virtual environments. Traditional storage provisioning methods treat these workspaces as distinct entities, resulting in a linear—and often unmanageable—growth in disk usage. This redundancy

leads to inefficient hardware utilization and increased costs, creating a need for smarter storage strategies that can discern and manage duplicate data intelligently.

This project addresses these inefficiencies by proposing a cohesive infrastructure deployment that integrates the orchestration power of Kubernetes with the advanced storage capabilities of the ZFS file system. By leveraging ZFS's native block-level deduplication beneath a standard Kubernetes distribution (implemented here via kind), the system transparently collapses redundant data blocks across user volumes. This approach allows for a substantial reduction in physical storage requirements without compromising performance or user isolation, offering a sustainable architecture for resource-constrained educational institutions hosting data-intensive applications.

2. Security and Isolation in Multi-User Architectures

The Multi-Tenant Paradox

Modern research and technical education increasingly rely on "Container-as-a-Service" (CaaS) models. In this architecture, a central cluster, usually managed by Kubernetes, allocates isolated environments on-demand for hundreds of users. This model is efficient because it maximizes hardware utilization; however, it creates a *multi-tenant* paradox. Unlike traditional virtualization, where each user operates within a distinct kernel via a hypervisor, containerized users share the host operating system's kernel [1].

While tools like *Zero to JupyterHub* have popularized this approach by simplifying deployment, they can inadvertently encourage a "deploy and forget" mentality. My contention is that convenience often comes at the cost of security awareness. In a shared environment where students or researchers run arbitrary, unverified code, relying on the "goodwill" of users is a failed security strategy. As noted in recent security analyses, multi-tenancy without strict isolation layers increases the attack surface, requiring architects to implement robust boundaries [2]. Therefore, the security of the platform must be guaranteed by hard architectural constraints rather than user policy.

The "Noisy Neighbor" Dilemma and Resource Starvation

The most frequent operational threat in any multi-tenant cluster is the "Noisy Neighbor" phenomenon. This occurs when a single user's workload monopolizes the CPU, memory, or I/O of a physical node, degrading the performance for all other tenants residing on that same hardware [3]. In scientific computing, where workloads involve heavy matrix multiplications or massive dataset processing, this risk is acute.

Kubernetes, by default, allows containers to consume as much of the host's resources as available. If a user introduces a memory leak in their code, the Linux Out-Of-Memory (OOM) killer may terminate critical system processes or other users' containers to save the node.

To mitigate this, I argue that the implementation of **Resource Requests and Limits** combined with **Namespace Quotas** is mandatory.

- **Requests** ensure that the scheduler only places a user on a node if sufficient guaranteed resources exist.

- **Limits** act as a hard barrier; if a process exceeds its limit, it is throttled or terminated immediately, isolating the failure to the individual.
- **Quotas** prevent a single user from spawning excessive containers that could starve the entire cluster. By strictly enforcing these parameters, administrators shift the responsibility of efficiency to the user, protecting the stability of the collective infrastructure [3].

3. Lateral Movement and the Zero-Trust Network

A more critical vulnerability in shared architectures is the potential for unauthorized data access through lateral movement. By default, Kubernetes employs a "flat" network topology, meaning every container can communicate with every other Pod in the cluster, regardless of ownership [4].

In an academic or corporate research setting, this is unacceptable. It implies that *User A* could theoretically scan the internal network, identify *User B*'s running database or notebook server, and attempt to exploit unpatched vulnerabilities.

The necessary defense is the implementation of a **Zero-Trust Network** model using **Network Policies**. A strict "Default Deny" policy should be applied to all namespaces [4]. Traffic should only be whitelisted if it is explicitly required (e.g., allowing the ingress controller to talk to the user container). This ensures that even if a container is compromised, the attacker is trapped within a network silo, unable to pivot to other systems or access the cloud provider's metadata services to steal credentials [5].

The Kernel Boundary and Privilege Escalation

Finally, we must address the fundamental weakness of containerization: the shared kernel. A "container escape" vulnerability allows a malicious process to break out of the container and execute commands on the host server with root privileges, effectively compromising every user on that node [1].

Standard mitigation involves enforcing strict **Pod Security Standards**. Containers should be forced to run as unprivileged users (non-root) and must be denied the ability to escalate privileges. However, for high-stakes environments processing sensitive data, I argue that standard containers may be insufficient. In such cases, the use of sandboxed runtimes like **gVisor** or **Kata Containers**, which provide an additional layer of kernel isolation, should be considered the gold standard, despite the slight performance overhead.

Conclusion

The shift toward multi-tenant, containerized cloud architectures represents a massive leap forward in accessibility for scientific computing. However, platforms that facilitate this, such as those built on Kubernetes, are not secure by default. The shared nature of the infrastructure introduces risks of resource contention and lateral attacks that cannot be ignored. My analysis suggests that the viability of these platforms depends entirely on rigorous constraint enforcement. By treating Resource Quotas, Network Policies, and Security Contexts as non-negotiable baselines, organizations can build environments that are not only scalable and efficient but also resilient against the inevitable risks of multi-tenancy.

3. Optimizing Storage Efficiency in Multi-Tenant Kubernetes Environments: From NFS to Copy-on-Write

Introduction

Kubernetes was initially architected for stateless microservices, but the ecosystem now demands persistence for databases and Continuous Integration (CI) pipelines. This shift creates a "persistence paradox": requiring durable storage in an ephemeral environment. Initially, the Network File System (NFS) was adopted for its ReadWriteMany (RWX) capabilities. However, as cluster density scales, NFS exhibits severe performance degradation due to "chatty" Remote Procedure Calls (RPCs) and centralized locking.[6] Furthermore, shared file systems lack the granular resource controls necessary to prevent aggressive tenants from monopolizing I/O bandwidth, leading to the "noisy neighbor" effect.[7] This report evaluates Copy-on-Write (CoW) block storage as a superior alternative, focusing on storage efficiency and isolation.

Architectural Analysis: NFS vs. CoW Block Storage

The File-Based Bottleneck: NFS abstracts storage behind a file system layer. Research by Harter et al. indicates that traditional storage drivers suffer high latency during container provisioning because they rely on full-file copying rather than block sharing.[8] In multi-tenant setups, all users often share a single file system journal; a single write-heavy tenant can saturate this journal, blocking I/O for the entire cluster. [9]

The CoW Advantage: Copy-on-Write block storage manages data as immutable chunks. Creating a volume snapshot is an O(1) operation: the system duplicates metadata pointers rather than physical data. This decouples storage cost from tenant count. For example, 100 database clones consume only the space of the original master plus subsequent deltas, offering >90% storage efficiency compared to NFS's linear consumption.

Performance and Isolation Evaluation

Empirical benchmarks confirm the architectural superiority of block storage in cloud-native environments, particularly regarding throughput stability and latency predictability.

A. Throughput and Latency Mercl and Pavlik [6] benchmarked Kubernetes storage backends on public cloud infrastructure. Their results (Table 1) illustrate the performance ceilings of distributed block storage compared to container-attached storage solutions.

TABLE I. COMPARATIVE STORAGE PERFORMANCE

System	Read IOPS	Write IOPS	Read Latency (ms)	Write Latency (ms)
Ceph (Block)	~10,900	~1,324	0.8	20.0
OpenEBS (cStor)	~1,351	~715	37.1	466.7

The data highlights a critical distinction in block storage implementations. Ceph demonstrates the theoretical maximum of the CoW model, achieving ultra-low read latency 0.8 ms. This is attributed to the CRUSH algorithm, which allows clients to calculate data location deterministically, eliminating the need for centralized metadata lookups.

B. The NFS Contrast

While not explicitly detailed in Table 1, the performance profile of NFS in similar high-concurrency scenarios fundamentally differs from the block storage metrics above.

- **Metadata Serialization:** Unlike Ceph's deterministic placement, NFS relies on recursive directory lookups. In multi-tenant environments, this creates a serialization bottleneck where latency increases linearly with the number of active pods.
- **The "Thundering Herd":** As noted in previous comparisons, NFS struggles with random I/O patterns typical of microservices. While Ceph sustains $\sim 1,300$ Write IOPS via parallelized journal writes, NFS write throughput is frequently capped by the synchronous commit requirement of the POSIX standard, forcing the file system to wait for physical disk acknowledgement before proceeding.

C. Isolation Mechanisms

Beyond raw throughput, the primary advantage of CoW block storage is isolation. Block storage provisions a dedicated virtual block device (e.g., `/dev/rbd0`) for each Persistent Volume.

- **Resource Guardrails:** This architecture allows Kubernetes to enforce strict IOPS limits at the hypervisor or cgroup level.
- **Impact on Neighbors:** Kwon et al. [7] demonstrated that enforcing such isolation at the block level (via their "DC-Store" framework) eliminates the "noisy neighbor" interference. Their experiments showed that victim containers on shared block storage saw a 31% reduction in execution time compared to those on shared file systems, where aggressive tenants saturated the global journal.

Conclusion

The architectural constraints of NFS, specifically metadata serialization and shared failure domains, render it inefficient for high-density Kubernetes clusters. Scientific evidence confirms that CoW block storage solutions like Ceph deliver three critical advantages:

- **Efficiency:** Instantaneous O(1) snapshots reduce storage footprints. [8]
- **Performance:** Significantly lower latency and higher IOPS ceilings. [6]
- **Isolation:** Deterministic performance that protects tenants from interference.[7]

For scalable multi-tenancy, CoW block storage is the optimal architectural choice.

4. From Economic Efficiency to Structural Vulnerability: How Autoscaling Enables EDoS Attacks in Cloud Computing

Introduction

The promise of cloud computing relies heavily on the concept of elasticity—the ability to provision and de-provision resources on demand. Autoscaling is the technical realization of this promise, allowing systems to handle fluctuating workloads while optimizing for performance and cost [12], [13]. In ideal conditions, this ensures that organizations do not pay for idle resources (over-provisioning) nor suffer from service degradation (under-provisioning) [15]. However, the automated nature of these decisions, when coupled with the public cloud's usage-based billing models, introduces a paradox: the system is architected to spend money automatically in response to demand. This architectural feature is the primary vector for Economic Denial of Sustainability (EDoS) attacks. As described by recent literature, EDoS is a "special breed" of denial of service where the attacker's goal is not to crash the service, but to render it financially unviable [14]. This essay explores the structural mechanics of this vulnerability, arguing that the current reactive nature of autoscaling algorithms facilitates these attacks, turning the system's efficiency logic against itself.

The weaponization of Reactive Scaling

Autoscaling systems generally operate on a reactive loop: they monitor metrics (such as CPU usage or request latency) and trigger a scaling action when specific thresholds are breached [13]. In a benign environment, this is efficient. In an adversarial environment, this is a deterministic vulnerability. An EDoS attacker does not need to launch a massive volumetric attack that triggers network firewalls. Instead, they can generate intermittent or sophisticated low-intensity traffic designed specifically to cross the scaling threshold. Once the autoscaler detects this artificial demand, it provides new virtual machines or Kubernetes pods [11]. The victim is immediately charged for these resources. By carefully modulating the attack traffic, the adversary can force the system to oscillate or remain in a high-cost state, exploiting the cloud provider's Service Level Agreement (SLA) aimed at high availability. As noted in [14], the cloud service provider acts as an unwitting accomplice, scaling the architecture to serve requests that ultimately drain the consumer's funds.

The Amplification Effect of Billing Granularity

The economic impact of an EDoS attack is frequently amplified by the disconnect between scaling speed and billing resolution. While resources can often be spun up in seconds or minutes, billing models often operate on hourly or per-minute minimums. Research into Kubernetes environments has shown that billing structures significantly influence the efficacy of an attack. For instance, hourly billing resolutions can eliminate the economic incentive for scaling down resources immediately after a traffic spike subsides [11]. An attacker needs only to trigger a scale-up event once every hour to ensure the victim pays for maximum capacity continuously. Empirical studies have demonstrated that an EDoS attack can result in a disproportionate loss of efficiency; a $4\times$ increase in traffic intensity can lead to a $3.6\times$ decrease in economic efficiency [11]. This "hysteresis" effect means the cost to the victim is far greater than the cost to the attacker, satisfying the asymmetry required for a successful economic attack.

Inadequacy of Static Thresholds

Current autoscaling techniques, ranging from simple threshold-based rules to more complex queuing theories, focus primarily on performance metrics rather than request legitimacy or economic risk [12]. Because the autoscaler treats all traffic that passes the firewall as "demand," it lacks the semantic

understanding to differentiate between a flash crowd of legitimate users and an EDoS campaign. Furthermore, predictive scaling mechanisms, while promising for performance, may inadvertently lock in high costs if they are poisoned by malicious historical data [12]. The mitigation of such attacks requires introducing "friction" into the scaling process—such as Turing tests (CAPTCHAs) or source checking [14]—which inherently contradicts the seamless user experience and low latency that autoscaling aims to protect. Recent proposals suggest "randomized" scaling decisions to disrupt the attacker's ability to predict the system's threshold, thereby reducing the efficiency of the attack [11].

Conclusion

Autoscaling represents a fundamental trade-off in cloud architecture. While it is indispensable for handling legitimate "Big Data" workloads and ensuring service reliability [15], it essentially provides an open checkbook to the internet. The EDoS attack proves that economic efficiency measures can become structural vulnerabilities when the input signal (traffic demand) is malicious. To secure the cloud effectively, we must move beyond standard resource metrics. Future autoscaling architectures must integrate economic risk modeling and traffic legitimacy verification directly into the scaling logic, treating financial sustainability as a metric equal in importance to CPU utilization.

5. Implementation Description

The implementation of the proposed architecture follows a stratified approach, layering the application orchestration on top of a specialized storage backend. The deployment process is divided into three distinct phases: host storage configuration, cluster orchestration, and application deployment.

1. Host Storage Configuration (ZFS Layer)

The foundation of the infrastructure relies on the ZFS file system to handle physical data management. We first prepared the host environment by installing the **zfsutils-linux** package to provide the necessary kernel modules and userspace utilities. The core optimization mechanism was enabled during the creation of the ZFS dataset. We executed the following command to create a persistent storage pool specific to the cluster:

```
sudo zfs create -o dedup=on -o compression=lz4 /jhub-pool/userdata
```

Two critical properties were set during this step: **dedup=on** and **compression=lz4**. The deduplication flag instructs ZFS to utilize a process-wide fingerprinting table to identify and eliminate duplicate data blocks as they are written to disk. The **lz4** compression algorithm was selected to provide high-throughput active compression, further reducing the storage footprint with negligible CPU overhead. Finally, to ensure the containerized Kubernetes node utilizes this storage without permission errors, we adjusted the dataset permissions (e.g., **chmod 777**) to allow the Docker container process full read/write access to the host path.

2. Cluster Orchestration (Kubernetes via Kind)

For the container orchestration layer, we utilized **kind** (Kubernetes in Docker), which allows for the rapid provisioning of lightweight clusters. The critical integration point between the host's ZFS storage and the Kubernetes ecosystem was achieved through a custom cluster configuration passed to **kind** during initialization.

We automated this process via a shell script (**setup_cluster.sh**). The script creates a single-node control-plane cluster with a specific **extraMounts** configuration:

```
nodes:  
- role: control-plane  
extraMounts:  
- hostPath: /jhub-pool/userdata  
  containerPath: /var/local-path-provisioner
```

This configuration binds the host's ZFS dataset (**/jhub-pool/userdata**) directly to the directory used by the cluster's default storage provisioner (**/var/local-path-provisioner**). Consequently, when Kubernetes dynamically provisions a Persistent Volume (PV) for a user, the data is physically written to the ZFS dataset on the host, automatically inheriting the deduplication and compression properties defined in the previous step.

3. Application Deployment (JupyterHub)

The final layer involved deploying the JupyterHub application using its official Helm chart. We defined a *values.yaml* configuration file to customize the deployment for this specific infrastructure. The configuration specified the use of the **standard** storage class, which maps to the local path provisioner backed by our ZFS mount. We allocated a 2Gi capacity per user for their workspace.

The installation was orchestrated by an `install_jupyterhub.sh` script, which adds the JupyterHub repository, manages namespace creation (**jhub**), and performs a **helm upgrade --install** operation. Within the configuration, the proxy service was set to **NodePort**, exposing the HTTP service on port **30080**. This architecture ensures that while the application behaves like a standard cloud-native deployment, the underlying storage transparently handles the high data redundancy typical of educational workloads.

6. Conclusion

This project demonstrated the viability of integrating a lightweight Kubernetes orchestration layer with a ZFS-backed storage system to address the unique resource challenges inherent in large-scale educational computing environments. By transparently offloading data management to the host file system, the architecture successfully decoupled the logical isolation of user workspaces from the physical storage footprint, achieving significant efficiency gains through block-level deduplication and compression. The resulting deployment offers a robust, cost-effective solution for academic institutions, ensuring that infrastructure limitations do not hinder the delivery of complex, data-intensive curricula while maintaining a seamless and reproducible experience for students.

Bibliography

[1] Wiz Academy, "Containerization vs. virtualization: Key differences explained," *Wiz.io*, Oct. 2025. [Online]. Available: <https://www.wiz.io/academy/containerization-vs-virtualization>

[2] N. M. Y. Al-maweri, "Highly Scalable and Secure Kubernetes Multi Tenancy Architecture for Fintech," *International Journal of Computer Trends and Technology*, vol. 72, no. 10, pp. 8-16, 2024. [Online]. Available:

https://www.researchgate.net/publication/384921156_Highly_Scalable_and_Secure_Kubernetes_Multi_Tenancy_Architecture_for_Fintech

- [3] S. K. Singh and S. K. Pandey, "A Systematic Approach to Deal with Noisy Neighbour in Cloud Infrastructure," *Indian Journal of Science and Technology*, vol. 9, no. 19, May 2016. [Online]. Available: https://www.researchgate.net/publication/303741535_A_Systematic_Approach_to_Deal_with_Noisy_Neighbour_in_Cloud_Infrastructure
- [4] T. Ziegler, T. B. Thai, and P. V. Gratz, "Network Policies in Kubernetes: Performance Evaluation and Security Analysis," in *Proceedings of the 5Ghosts Conference*, 2021. [Online]. Available: <https://www.5ghosts.eu/publications/papers/paper1.pdf>
- [5] P. K. Gupta, "Exploring Security Challenges and Solutions in Kubernetes," *International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)*, vol. 4, no. 1, Jan. 2024. [Online]. Available: <https://ijarsct.co.in/Paper15205.pdf>
- [6] L. Mercl and J. Pavlik, "Public Cloud Kubernetes Storage Performance Analysis," in *Computational Collective Intelligence*, Cham: Springer, 2019, pp. 649–660.
- [7] M. Kwon et al., "DC-Store: Eliminating Noisy Neighbor Containers using Deterministic I/O Performance and Resource Isolation," in *Proc. 18th USENIX Conf. File and Storage Technologies (FAST '20)*, 2020, pp. 183–191.
- [8] T. Harter et al., "Slacker: Fast Distribution with Lazy Docker Containers," in *Proc. 14th USENIX Conf. File and Storage Technologies (FAST '16)*, 2016, pp. 181–195.
- [9] D. Huang et al., "Enhancing Proportional IO Sharing on Containerized Big Data File Systems," *IEEE Trans. Comput.*, vol. 69, no. 12, 2020.
- [10] G. Kappes et al., "Danaus: isolation and efficiency of container I/O at the client side of network storage," in *Proc. 22nd Int. Middleware Conf.*, 2021, pp. 86–98.
- [11] J. Chamberlain et al., "Exploiting Kubernetes Autoscaling for Economic Denial of Sustainability," ACM Digital Library, 2025. [Online]. Available: <https://dl.acm.org/doi/pdf/10.1145/3727114>
- [12] S. Alharthi et al., "Auto-Scaling Techniques in Cloud Computing: Issues and Research Directions," *Sensors*, vol. 24, no. 17, 2024. [Online]. Available: <https://www.mdpi.com/1424-8220/24/17/5551>
- [13] H. Alipour et al., "Analyzing Auto-scaling Issues in Cloud Environments," CASCON, 2014. [Online]. Available: <http://users.encs.concordia.ca/home/a/abdelw/papers/CASCON14-Autoscaling.pdf>
- [14] M. Hanini, "Mitigating Economic Denial of Sustainability Attacks to Secure Cloud Computing Environments," *Transactions on Machine Learning and Artificial Intelligence*, 2021. [Online]. Available: <https://d1wqxts1xzle7.cloudfront.net/66347440/2142-libre.pdf>
- [15] D. T. Valivarthi, "Optimizing Cloud Computing Environments for Big Data," *Journal of Engineering and Science Research*, 2023. [Online]. Available: https://d1wqxts1xzle7.cloudfront.net/123220377/OPTIMIZING_CLOUD_COMPUTING_ENVIRONMENTS.pdf