GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Security Analysis of Ephemeral Kubernetes in HPC Environments

Shrinath Madde

MatrNr: 25235677

Supervisor: Jonathan Decker

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2026

Abstract

Kubernetes is a widely used container orchestration platform that automates the deployment, scaling, and management of containerised applications. Ephemeral Kubernetes [1] is a novel framework that integrates container orchestration directly into High-Performance Computing (HPC) environments by leveraging Warewulf, a stateless node provisioning system, to dynamically create and destroy Kubernetes clusters through simple node reboots, achieving sub-90-second cluster deployment times. While this approach offers rapid and flexible cluster provisioning, its original architecture inherently prioritises speed over security. By relying on shared Network File System (NFS) storage for cluster bootstrapping, the system exposes critical cryptographic materials, most notably full cluster-admin credentials, creating severe risks of persistent credential exposure and privilege escalation. This paper analyses these fundamental security flaws and proposes two architectural enhancements to secure the cluster lifecycle. First, we introduce a key-based credential distribution system that eliminates persistent storage risks through authenticated, transient file delivery. Second, we implement a token-based access control model that enforces the principle of least privilege by replacing unrestricted admin credentials with scoped join tokens. Together, these methodologies ensure that Ephemeral Kubernetes can satisfy the rigorous security demands of multi-tenant HPC facilities without sacrificing its signature agility and performance.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☑ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Abbreviations

API Application Programming Interface

CA Certificate Authority

CN Common Name

GPU Graphics Processing Unit

HPC High-Performance Computing

HTTP Hypertext Transfer Protocol

KSI Kind Slurm Integration

MPI Message Passing Interface

NFS Network File System

PKI Public Key Infrastructure

SHA Secure Hash Algorithm

VM Virtual Machine

WW Warewulf

1 Introduction

While Kubernetes is not as popular in production HPC environments as it is in cloud VMs due to complexity in adaptability, maintainability, performance bottlenecks, and security, several genuine approaches attempt to fill this gap, but they suffered in one or two key areas.

Zhou et al. [2] use Torque-Operator to bridge Kubernetes and HPC clusters managed by TORQUE, enabling container orchestration while keeping the HPC environment intact. However, this requires persistent external components outside Slurm jobs, adding infrastructure complexity and security concerns in multi-user environments. Chazapis et al. [3] run Slurm under Kubernetes, converting user services into Slurm scripts that execute as Singularity containers on HPC compute nodes. This approach does not support Kubernetes ClusterIP services, lacks GPU mapping and port forwarding, and relies on static Slurm allocations, limiting scheduling flexibility. Decker et al. [4] introduced KSI (Kind Slurm Integration) to deploy full Kubernetes clusters within Slurm jobs. While the original version suffered from poor network performance and single-node limits, the updated KSI 2 implementation fixes these issues using bypass4netns, achieving network speeds comparable to bare metal. However, this fix requires Linux kernel 5.9+, e.g., Rocky Linux 9, which is not available in all HPC environments, and introduces specific security considerations.

Ephemeral Kubernetes [1] addresses challenges in adaptability, maintainability and performance bottlenecks by integrating container orchestration directly into existing HPC stacks via Warewulf with minimal architectural changes. This framework provides native Kubernetes performance and services, and could also be used to dynamically negotiate HPC cluster capacity between on-demand and batch clusters [5]. The current Ephemeral Kubernetes implementation prioritises deployment speed and operational simplicity over security, relying on shared Network File System (NFS) storage to distribute sensitive cryptographic materials. Specifically, the Kubernetes cluster-administrator configuration file (`Kube.config`) containing full cluster-admin credentials is stored on NFS accessible to all cluster nodes. This architectural decision creates severe security vulnerabilities:

- **Credential Exposure**: Any compromised node or malicious container with NFS access can steal cluster-admin credentials.
- **Privilege Escalation**: Stolen credentials enable deployment of privileged containers, leading to root access on any cluster node.
- **Lateral Movement**: Compromising a single worker node provides a path to compromise the entire cluster.

These vulnerabilities are unacceptable for production HPC environments, particularly those serving multiple tenants, processing sensitive data, or subject to compliance requirements.

2 Ephemeral Kubernetes Overview

2.1 Architecture Overview

Ephemeral Kubernetes is built upon Warewulf (WW), a stateless node provisioning system commonly deployed in HPC environments. The architecture consists of three primary components: A Warewulf host server that provisions operating system images, control plane nodes that manage the Kubernetes cluster, and worker nodes that execute containerised workloads.

Unlike traditional Kubernetes deployments on cloud infrastructure, where nodes maintain persistent state, nodes in this HPC-based setup boot from network-provisioned images through Warewulf and lose all local state upon reboot. The system leverages two distinct Warewulf images:

- A **control image** containing Kubernetes control plane components [6] (`kube-apiserver`, `kube-scheduler`, `kube-controller-manager`, `etcd`), HAProxy, and Keepalived for high availability.
- A **base image** for worker nodes containing only the `kubelet` and container runtime.

Both images include pre-loaded container images to enable offline deployment without internet connectivity.

A critical architectural component is the shared Network File System (NFS) storage mounted at `/share` on all nodes. This shared storage serves multiple purposes: Synchronising configuration between nodes, storing pre-loaded container images, coordinating leader election during cluster bootstrap, and distributing cryptographic credentials required for cluster formation, as illustrated in Figure 1.

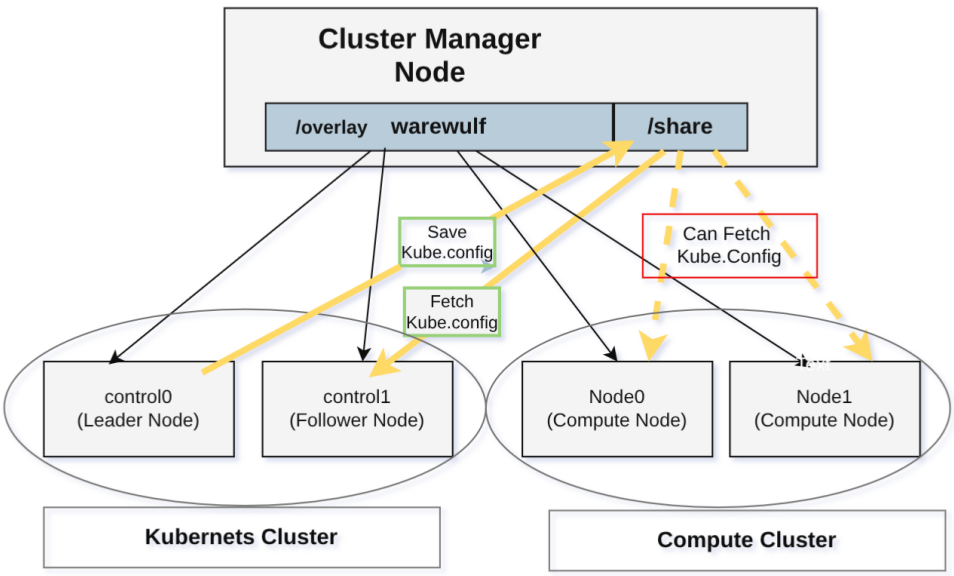

Figure 1: Unsecured ephemeral Kubernetes setup: The (`Kube.config`) (used to join the cluster) is directly accessible through the /share folder; anyone can use it and deploy privileged pods and get root access on the node. Figure made with DrawIO.

2.2 Cluster Formation and Node Join Process

The cluster formation process in Ephemeral Kubernetes follows a coordinated sequence that relies heavily on the shared NFS storage for synchronisation.

2.2.1 Leader Election & Cluster Initialization

When multiple control plane nodes boot simultaneously, a leader election mechanism determines which node initialises the cluster. The leader node creates a `/share/leader` folder on the NFS share. Other control nodes see that the `/share/leader` folder already exists, recognise themselves as followers and wait for cluster initialisation to complete.

The elected leader node performs several security-critical operations:

- **Certificate Generation**: The leader runs `kubeadm init`, which generates the cluster's Public Key Infrastructure (PKI) [7], including the Certificate Authority (CA) certificates, API server certificates, and etcd certificates.
- **Credential Creation**: `Kubeadm` generates the administrator configuration file (`Kube.config`) [8] containing client certificates signed by the cluster CA. These certificates include the Common Name (CN) `kubernetes-admin` and Organisation (O) `system:masters`, granting unrestricted cluster access.
- **Credential Distribution**: The leader copies all generated certificates and the admin configuration file to the NFS share: `/share/pki/ca.crt`, `/share/pki/ca.key`, `/share/pki/etcd/ca.crt`, `/share/pki/etcd/ca.key`, and `/share/Kube.config`.
- **Ready Signal**: After completing initialisation, the leader creates the `/share/leader_ready` file to signal other nodes.

2.2.2 Follower and Worker Node Join Process

Non-leader nodes poll for the `leader_ready` file every 5 seconds. Once detected, nodes retrieve credentials from NFS and join the cluster. All joining nodes authenticate with the API server using the client certificate embedded in the `Kube.config` file, which is validated against the cluster CA (`ca.crt`).

- **Control Plane Follower Nodes**: Copy CA certificates from `/share/pki/` to `/etc/kubernetes/pki/` as they need to run control plane components such as the API server, etcd, and scheduler. They then start the Phylactery service for HAProxy configuration management and join as control plane members using `kubeadm join -discovery-file /root/.kube/config -control-plane -v=5`.
- **Worker Nodes**: Do not require any PKI certificates as they only run workloads and do not host any control plane components. They simply read the credentials from `/share/Kube.config` and join the cluster using `kubeadm join -discovery-file /root/.kube/config -v=5`.

Both node types use the same administrator credentials for cluster joining.

2.3 Phylactery Service and Node Rejoin Mechanism

The Phylactery service is a Python-based daemon (`phylactery.py`) deployed exclusively on control plane nodes as a systemd service. It serves two primary purposes within the Ephemeral Kubernetes architecture. First, it manages the High-Availability (HA) setup by maintaining up-to-date HAProxy configurations across all control plane nodes. When a new control plane node joins the cluster, every existing HAProxy instance must be updated with the new node's endpoint. Phylactery automates this by writing its node's hostname and IP address to the NFS share, reading the corresponding files from other nodes to reconstruct the full HAProxy backend list, and then notifying all other Phylactery instances via a lightweight HTTP server (port 9999) to reload their configurations. Together with Keepalived, which manages the Virtual IP (VIP) address for the control plane endpoint, Phylactery ensures that API requests are continuously load-balanced across all active control plane nodes, providing automatic failover.

The second and equally critical function of Phylactery is handling the rejoin process for rebooted nodes. Since Warewulf provisions nodes statelessly, a rebooted control plane node loses all local state and reappears to the existing cluster as a new entity attempting to join with the same hostname and IP address. Both Kubernetes and etcd refuse such a join, as they already have an active member registered under those identifiers. Before the node can successfully rejoin, Phylactery reads the `Kube.config` file from the NFS share to authenticate with the cluster API, then uses `kubectl` to drain and delete the stale node entry from the Kubernetes node list, and `etcdctl` to remove the corresponding stale member from the etcd cluster. Only after completing this cleanup does Phylactery signal readiness, allowing the installation script to proceed with the `kubeadm join` command. This cleanup mechanism is the precise reason the original architecture requires the `Kube.config` file to be persistently available on the NFS share without the cluster-admin credentials it contains, Phylactery cannot authenticate with the API server to purge these stale entries.

2.4 PKI Certificates and Their Usage

Kubernetes uses certificates and keys to secure the cluster and verify identities. Each certificate serves a specific purpose in the cluster's security model.

- **Cluster CA (`ca.crt/ca.key`)**: The master certificate for the entire cluster. The `ca.crt` file lets nodes verify they're talking to the real API server. The `ca.key` file is used to create certificates for all cluster components. This is the most critical key - anyone with `ca.key` can create admin credentials and take over the cluster.

- **Service Account Keys (`sa.key/sa.pub`)**: These keys create and verify tokens for pods running in the cluster. When a pod needs to talk to the API server, it gets a token signed with `sa.key`. The API server checks this token using `sa.pub`; all control plane nodes must use identical keys so tokens work no matter which API server receives the request.

- **Front Proxy CA (`front-proxy-ca.crt/front-proxy-ca.key`)**: Used for API aggregation, which lets additional API servers connect to the main Kubernetes API. All control plane nodes must share these certificates for extension APIs to work correctly.

- **etcd CA** (`etcd/ca.crt/etcd/ca.key`): Protects communication between etcd database instances. When a new control plane node joins, it uses `etcd/ca.key` to create certificates that let it communicate securely with existing members.
- **Admin Kubeconfig File**: The `Kube.config` file contains the cluster CA certificate, a client certificate proving admin identity, and a private key. Possession of this file grants complete administrative control over the cluster.

2.5 Security Challenges in the Ephemeral Kubernetes Model

As mentioned in Section 2.3, all the nodes in the Kubernetes cluster plane need to have persistent access to cluster config files and certificates, which exist on `/share` folder on the cluster manager, which can give admin-level access to the Kubernetes cluster to any user on any node.

- **Privilege Escalation**: Stolen credentials enable deployment of privileged containers [9], leading to root access on any cluster node. A user with admin access to a Kubernetes cluster can escalate this to root access on any node of the cluster, for example, by deploying a privileged container that mounts `/etc/shadow` and replaces the root password.
- **No Revocation Mechanism**: Once credentials are copied, they cannot be revoked or expired. If an attacker copies the `Kube.config` file, there is no way to invalidate those credentials without recreating the entire cluster.
- **Lateral Movement**: Compromising a single worker node provides a path to compromise the entire cluster. The shared credential model means that breaching any single node's security gives an attacker the keys to the entire infrastructure.

The current setup stores `ca.key` and `Kube.config` in the shared `/share` directory, which creates a security risk. Anyone accessing this folder can use `ca.key` or `Kube.config` to generate admin certificates and gain full cluster access. A better approach would use a secured folder for storing these sensitive files, which is accessible only with some authentication, or share only limited-permission join tokens and service account tokens that only allow specific operations like node management.

3 Proposed Security Enhancement: Key-Based Credential Distribution

To address the critical security vulnerabilities inherent in the NFS-based credential distribution model, we propose a key-based authentication system that eliminates persistent credential exposure while maintaining the operational benefits of ephemeral cluster deployment.

3.1 Architecture Overview

The enhanced architecture replaces the shared NFS storage of sensitive credentials with a secure file server running on the Warewulf cluster manager. This server implements a key-based authentication mechanism where each node receives a unique key that grants access to the credentials and config file.

The secure file server operates as an end-to-end encrypted HTTPS service listening on the cluster manager's internal network interface (typically `10.0.0.3:8000`). Unlike the previous NFS-based approach, where credentials remained indefinitely accessible at `/share/pki/` and `/share/Kube.config`, the new system stores all sensitive materials in a private directory (`/var/lib/warewulf/private-secrets`) with strict filesystem permissions (mode 700), preventing access by any user or process except root, and the server running on the cluster manager will serve those files to anyone with a valid key, as illustrated in Figure 2.

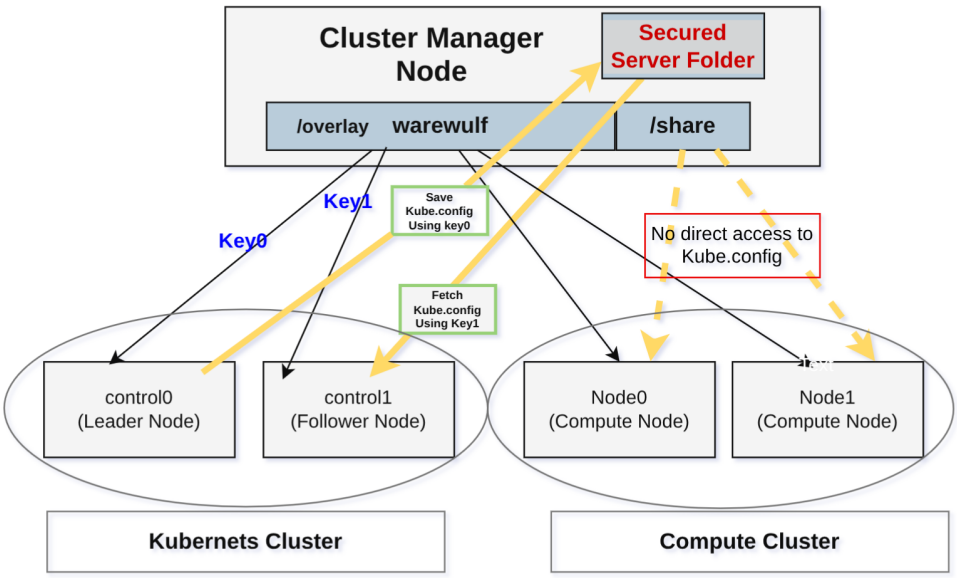

Figure 2: Kube.config will be saved in a secure server folder on the cluster manager, which is accessible using key (separate for each node, provided through overlay) authentication. Figure made with DrawIO.

3.2 Key Generation and Distribution

Prior to cluster deployment, the cluster administrator executes a key generation script (`generate-Keys.sh`) that creates unique SHA-256 Keys for each node registered in Warewulf. Each key is stored as a JSON file in `/var/lib/warewulf/Keys/` containing metadata including the associated node name, creation timestamp, and expiration status.

The Keys are distributed to nodes through Warewulf's system overlay, alongside other Kubernetes configuration files inside the `k8s-overlay`, and each node receives its unique key via a node-specific file (`/etc/k8s-key.{NODE_NAME}`) during node provisioning. Once a node joins the cluster successfully, the key can be deleted from the node as they are of no use afterwards, eliminating residual credential exposure on running nodes.

3.3 Secure Credential Upload Mechanism

The leader node, after completing `kubeadm init` and generating the cluster PKI, uploads sensitive materials to the secure file server rather than copying them to NFS storage. The upload process uses HTTPS POST requests with binary payloads, as shown in Listing 1.

```
curl -f -s -X POST --data-binary @/etc/kubernetes/admin.conf \
  "${SECURE_SERVER}/upload/kube.config/${LEADER_KEY}" \
  || { echo "ERROR: Failed to upload kube.config"; exit 1; }

curl -f -s -X POST --data-binary @/tmp/pki.tar.gz \
  "${SECURE_SERVER}/upload/pki.tar.gz/${LEADER_KEY}" \
  || { echo "ERROR: Failed to upload pki.tar.gz"; exit 1; }
```

Listing 1: Secure credential upload from the leader node (`k8s-install-final.sh`).

The server validates the key before accepting the upload and sanitizes filenames to prevent directory traversal attacks. Only two files are uploaded: The cluster administrator configuration (`Kube.config`) and a compressed archive of CA certificates (`pki.tar.gz`) containing exclusively:

- Cluster CA certificate and key (`ca.crt`, `ca.key`)
- Service account Keys (`sa.key`, `sa.pub`)
- Front proxy CA certificate and key (`front-proxy-ca.crt`, `front-proxy-ca.key`)
- etcd CA certificate and key (`etcd/ca.crt`, `etcd/ca.key`)

Critically, the archive excludes node-specific certificates (API server certificates, etcd peer certificates) that should be uniquely generated per node, preventing certificate reuse vulnerabilities.

3.4 Key-Authenticated Download and Self-Destruction

Follower nodes and worker nodes retrieve credentials by presenting their unique Keys to the secure file server. Listing 2 shows the full download and self-destruction sequence from `k8s-install-final.sh`.

The server validates the key's existence and ensures it has not been marked as expired. Upon successful credential download and cluster join completion, the node immediately deletes its key from local storage, removing the file `/etc/k8s-key.{NODE_NAME}`. This key self-destruction ensures that once a node successfully joins the cluster, the authentication credential no longer persists in the node's memory or filesystem, eliminating the risk of key theft from a running node.

3.5 Key Lifecycle and Node Rejoin Mechanism

The key lifecycle accommodates the ephemeral nature of Warewulf-provisioned nodes. When a node reboots, it loses all state including its key. However, Warewulf re-provisions the node from the base image during each boot, delivering the same key through the overlay system. The node uses this key to re-authenticate, download credentials, rejoin the

```
until curl -f -s "${SECURE_SERVER}/config/${NODE_KEY}" \
     -o /root/.kube/config; do
  echo "Waiting for kube.config on secure server...
    (${DOWNLOAD_ATTEMPTS}/30)"
  sleep 5
done
until curl -f -s "${SECURE_SERVER}/certs/${NODE_KEY}" \
     -o /tmp/pki.tar.gz; do
  echo "Waiting for pki.tar.gz on secure server...
    (${DOWNLOAD_ATTEMPTS}/30)"
  sleep 5
done
# ... join cluster via kubeadm join ...
rm -f /tmp/pki.tar.gz
# Self-destruct key -- attack window closed
rm -f "$NODE_KEY_FILE"
```

Listing 2: Key-authenticated credential download and key self-destruction on a follower or worker node (`k8s-install-final.sh`).

cluster (with Phylactery handling cleanup of the previous node instance), and subsequently deletes the key. This cycle repeats indefinitely across reboots without accumulating credential exposure. For operational deployments, or in the event of a suspected cluster compromise where an attacker might have stolen a node's key, administrators must regenerate Keys using `generate-Keys.sh` and rebuild Warewulf overlays with `wwctl overlay build` before rebooting the nodes. This ensures that previously compromised Keys cannot be reused to access the secure server.

3.6 Security Improvements

The key-based system delivers several critical security enhancements over the NFS-based approach:

- **Elimination of Persistent Exposure**: Sensitive credentials exist only in a private directory on the cluster manager after cluster formation, no longer stored indefinitely on a network-accessible filesystem.
- **Node-Specific Authentication**: Each node authenticates using a unique key tied to its identity, enabling audit logging of credential access for forensic investigation.
- **Reduced Lateral Movement Risk**: Compromising a worker node no longer grants access to cluster-admin credentials as the key is deleted immediately after joining.
- **Time-Bounded Attack Window**: The attack window for key theft is limited to the brief period between node boot and successful cluster join, typically under 90 seconds, drastically reducing the temporal attack surface compared to indefinite NFS exposure.

4 Alternative Approach: Token-Based Access Control

Instead of sharing the `Kube.config` file and sensitive certificate keys in the `/share` directory, another secure approach we implemented uses limited-permission tokens and carefully selected certificates that follow the principle of least privilege.

4.1 Files in the Shared Storage

To secure the cluster while maintaining the resilience required for HPC environments, this architecture shares only the specific credentials necessary for node lifecycle operations. Unlike the original configuration which exposed full administrative control, these files follow the principle of least privilege.

4.1.1 Bootstrap Join Token (`join-command.txt`)

This file contains the `kubeadm join` command equipped with a bootstrap token [10]. This token is required for all nodes (workers and control plane) to join the cluster.

- **Scope**: The token's permissions are strictly limited to the node joining process.
- **Security**: It cannot be used to access cluster resources, read secrets, or perform administrative operations. Even if a malicious actor obtains this token, they cannot query the API server or modify the cluster state.

4.1.2 Phylactery Service Account Token (`phylactery-token.txt`)

This token allows the Phylactery service to clean up stale node entries from the Kubernetes node list managed by the API server and from the etcd member database upon reboot and rejoin (unlike the original approach, where the service relied on the unrestricted `Kube.config` to clear these entries). It carries minimal permissions restricted to node management (list, delete, drain), ensuring it cannot access cluster secrets or perform broader administrative tasks.

- **Scope**: Strictly limited to node lifecycle management operations only. The token can list, drain, and delete node objects from the Kubernetes node list but has no permissions beyond these specific operations.
- **Security**: Even if a malicious actor obtains this token, they cannot access cluster secrets, deploy or modify workloads, generate new credentials, or perform any administrative operations. The token provides no path to privilege escalation unlike the original `Kube.config` which granted full cluster-admin access.

4.1.3 Certificate Key (`certificate-key.txt`)

This key is required exclusively for control plane follower nodes to fetch the certificates and private keys (`ca.key`, `etcd/ca.key`, `sa.key`, `front-proxy-ca.key`, `ca.crt`, `etcd/ca.crt`, `etcd/server.crt`, `etcd/server.key`) necessary for running control plane services such as the API server, etcd, and scheduler. In the original architecture, these

were stored in plain text on NFS, directly accessible to any node or user with share access. The token-based approach eliminates this by having the leader node encrypt all these sensitive files into a secure bundle during `kubeadm init`. The `certificate-key.txt` acts as the decryption password for this bundle. When a control plane follower joins or rejoins after a reboot, it presents this key to the API server, receives the encrypted bundle, and decrypts it locally to obtain the certificates it needs, ensuring raw private keys never reside on NFS at any point.

- **Scope**: Strictly limited to control plane nodes only. Worker nodes do not require this key as they never need to obtain or decrypt any certificate bundle.
- **Security**: Even if a malicious actor obtains this key, they must still actively interact with the API server to retrieve the encrypted bundle, generating auditable logs. Critically, raw private keys such as `ca.key` are never directly accessible on NFS, eliminating the primary attack vector of the original architecture.

4.2 Cluster Formation in Token-Based Architecture

Similar to the original architecture, a leader election mechanism determines which control plane node initialises the cluster. Once elected, the leader runs `kubeadm init` to generate the cluster PKI and bootstrap the control plane. Unlike the original architecture where all sensitive certificates and `Kube.config` were copied directly to the `/share` folder, the leader instead places only three limited-access credentials on NFS: `join-command.txt`, `phylactery-token.txt`, and `certificate-key.txt`. Once follower control plane and worker nodes detect that the cluster is ready, they first use `phylactery-token.txt` to clean up any stale node entries from the previous session, then use `join-command.txt` to register with the cluster. Additionally, control plane follower nodes use `certificate-key.txt` to fetch the required certificates and private keys from the API server before joining as control plane members. A comprehensive files usage comparison between the original architecture and the new token-based model is outlined in Table 1.

Component / Function	**Original Architecture (Insecure)**	**Updated Secured Approach (Token-Based)**
Phylactery service node cleanup	`Kube.config`	`phylactery-token.txt`
Cluster Joining	`Kube.config`	`join-command.txt`
Control Plane Authentication	Raw keys, e.g., `ca.key`, `etcd/ca.key`	`certificate-key.txt`

Table 1: Side-by-side comparison of credential distribution and access control between the original NFS-based architecture and the updated token-based security model.

5 Discussion

While the proposed architectural enhancements significantly improve the security posture of Ephemeral Kubernetes, a fundamental limitation applies to both the key-based and

token-based approaches. The cluster CA private key (`ca.key`) must remain persistently available on all control plane nodes, as Kubernetes requires it to sign certificates for new nodes joining the cluster. If an attacker compromises any active control plane node, they can use `ca.key` to generate arbitrary administrative credentials, escalating to full cluster access. This risk cannot be mitigated through credential distribution improvements alone, as it is a strict requirement of the Kubernetes PKI architecture. The protection of running control plane nodes therefore remains entirely dependent on the underlying infrastructure security and network isolation of the HPC environment.

6 Future Work

With the fundamental security challenges of Ephemeral Kubernetes addressed, the next logical step is the deployment of a fully integrated HPC platform where an Ephemeral Kubernetes cluster and a Slurm batch cluster operate side by side. In such a setup, Ephemeral Kubernetes would handle inference and service workloads, while Slurm continues its traditional role of managing compute-intensive batch jobs. The central challenge of this integration lies in the dynamic management of nodes between the two clusters: determining when and how nodes are migrated from Slurm compute duties to Kubernetes inference workloads and back, ideally driven by real-time demand signals from both schedulers. Building an optimal system that balances this flexibility with the energy and time costs of frequent node reboots remains an important open challenge.

Furthermore, this dual-cluster deployment introduces new security considerations regarding the Warewulf provisioning infrastructure. In a side-by-side setup, Warewulf serves distinct overlays to each group of nodes based on their registered identity. An attacker on a Slurm compute node requesting an overlay from the Warewulf server would normally only receive the Slurm-specific overlay, which contains no Kubernetes credentials. However, if the attacker obtains the MAC address and IP address of a node in the Kubernetes cluster (e.g., via standard unprivileged commands like `arp` or `ip neigh`), they could potentially impersonate that node and download its Kubernetes overlay containing the secret key for the secured credential server. Mitigating this impersonation vulnerability will require either removing secrets from overlays entirely or introducing strict network segmentation between the Kubernetes and Slurm clusters in future iterations.

7 Conclusion

Ephemeral Kubernetes offers a promising solution for integrating dynamic container orchestration into traditional HPC environments, though its original design left significant security challenges unaddressed. The reliance on unsecured NFS storage for credential distribution creates a single point of failure where the compromise of any node can lead to total cluster takeover. This report has implemented two architectural approaches to address these vulnerabilities.

The key-based credential distribution system (Section 3) provides a simpler and more intuitive solution by removing all sensitive materials from the shared filesystem entirely and placing them on a secured server with key-based authentication, where each node receives its unique key through the Warewulf overlay system. The leader node uploads

credentials using authenticated HTTPS POST requests (Listing 1), and each follower or worker node downloads them with its unique key and immediately destroys it upon joining (Listing 2). This approach eliminates direct access to credential files from the shared storage, significantly reducing the attack surface with minimal architectural overhead.

For environments that require an additional layer of security, the token-based access control model (Section 4) can be applied on top, replacing full administrative credentials with minimum access tokens that follow the principle of least privilege. Rather than sharing the `Kube.config` file containing unrestricted cluster admin access, this approach distributes only scoped bootstrap join tokens, a limited permission Phylactery service account token, and an encrypted certificate key, ensuring that even if an attacker breaks into the secured storage, they would need to perform the extra step of authenticating with the API server and downloading the admin credential files. By adopting these enhancements, HPC centres can substantially reduce the attack surface of Ephemeral Kubernetes for multi-tenant production use while preserving the rapid, stateless provisioning that defines the platform.

References

[1] Jonathan Decker and Julian Kunkel. Ephemeral Kubernetes: dynamically deleting and recreating clusters using Warewulf. *The Journal of Supercomputing*, 81:1491, 2025.

[2] Neng Zhou, Yiannis Georgiou, Maciej Pospieszny, et al. Container orchestration on HPC systems through Kubernetes. *Journal of Cloud Computing*, 10(16), 2021.

[3] Antony Chazapis, Fotis Nikolaidis, Manolis Marazakis, and Angelos Bilas. Running Kubernetes workloads on HPC. In *High Performance Computing. ISC 2023*, 2023.

[4] Jonathan Decker, Simon Metje, and Julian Kunkel. Running Kubernetes workloads on rootless HPC systems using Slurm. *Proceedings of the International Conference on Cloud Computing (CLOUD COMPUTING 2025)*, 2025.

[5] Feng Liu, Kate Keahey, Pierre Riteau, and Jon Weissman. Dynamically negotiating capacity between on-demand and batch clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*, 2018.

[6] Kubernetes Authors. Kubernetes components. `https://kubernetes.io/docs/concepts/overview/components/`, 2024. Accessed: 2025.

[7] Kubernetes Authors. PKI certificates and requirements. `https://kubernetes.io/docs/setup/best-practices/certificates/`, 2024. Accessed: 2025.

[8] Kubernetes Authors. Organizing cluster access using kubeconfig files. `https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/`, 2024. Accessed: 2025.

[9] Kubernetes Authors. Pod security standards. `https://kubernetes.io/docs/concepts/security/pod-security-standards/`, 2024. Accessed: 2025.

[10] Kubernetes Authors. Authenticating with bootstrap tokens. https://kubernetes.io/docs/reference/access-authn-authz/bootstrap-tokens/, 2024. Accessed: 2025.