Towards Enabling Hostile Multi-tenancy in Kubernetes

Ali Kanso
Microsoft Corporation
Redmond, WA, USA
alikanso@gmail.com

Slava Oks
Microsoft Corporation
Redmond, Washington, USA
slavaoks@microsoft.com

Mostafa Elzeiny
Microsoft Corporation
Redmond, WA, USA
moelzein@microsoft.com

Gurpreet Virdi
Microsoft Corporation
Redmond, WA, USA
gurvir@microsoft.com

Abstract

Kubernetes has become the de facto standard for container orchestration, yet its core design does not address the requirements of hostile multi-tenancy. Native constructs such as namespaces, role-based access control, and admission controllers offer logical separation but fall short of delivering the strong isolation guarantees needed in untrusted, adversarial environments. This paper introduces a Kubernetes-compatible architecture that combines per-tenant virtual control planes, hypervisor-backed virtual machines as sandboxes for containers, and automated policy enforcement to achieve secure, isolation-centric multi-tenancy. Each tenant is provisioned with a dedicated virtual control plane (via *vCluster*) connected to a virtual node that schedules workloads into VM-based sandboxes (with Azure Container Instances) while maintaining the familiar Kubernetes API. A policy engine (*Kyverno*) automatically hardens namespaces by enforcing network segmentation, resource governance, and strict security contexts at admission time. Evaluation results show that the proposed model significantly improves inter-tenant isolation with negligible runtime performance overhead, offering a practical pathway to zero-trust container orchestration in hostile cloud and edge environments.

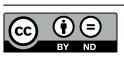
CCS Concepts

• **Software and its engineering** → Software creation and management; Software development techniques.

Keywords

Containers, security, multi-tenancy, workload isolation, Kubernetes

ACM Reference Format:
Ali Kanso, Slava Oks, Mostafa Elzeiny, and Gurpreet Virdi. 2025. Towards Enabling Hostile Multi-tenancy in Kubernetes. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3731599.3767357



SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/2025/11
https://doi.org/10.1145/3731599.3767357

1 Introduction

Recent advances in cloud-native high-performance computing (HPC) have demonstrated the feasibility of running large-scale, parallel workloads on *Kubernetes*, leveraging container orchestration for elasticity, reproducibility, and resource efficiency [1] [2]. In such increasingly diverse deployment environments, ranging from public clouds to edge computing, organizations face the challenge of securely hosting multiple tenants without compromising performance or usability.

Kubernetes was not originally designed with hostile multi-tenancy in mind [3]. By design, Linux containers that are orchestrated by *Kubernetes* rely on operating-system level virtualization. I.e., processes share the same host kernel, and isolation is achieved primarily through Linux namespaces and control groups (cgroups). Linux Namespaces partition the visibility of system resources such as process IDs (PID), file systems, network stacks, user and IPC spaces providing each container with its own "bubble" despite running on the same kernel [4, 5]. Furthermore, cgroups enforce resource constraints (CPU, memory, I/O) and enable process grouping and accounting, critical for multi-tenant management [5]. While these mechanisms offer efficient, lightweight isolation, they remain vulnerable to host-level compromise: shared system-call interfaces can be exploited to "escape" from containers into the host environment. Notable examples include the Dirty COW exploit (CVE-2016-5195) [6] and more recent vulnerabilities like CVE-2022-0185, which enabled *Kubernetes* container breakouts [7, 8].

The *Kubernetes* API server constitutes a critical control-plane component in a shared multi-tenant *Kubernetes* Cluster. Role-Based Access Control (RBAC) mechanisms can restrict user actions to defined namespaces, but RBAC alone cannot protect against attacks targeting the API server itself. A successful compromise of the API server, whether through software vulnerabilities, misconfiguration, or privilege escalation may enable an adversary to manipulate resources across tenant boundaries. Recent proposals such as *KubeFence* [9] attempt to mitigate this risk via fine-grained API filtering, but such solutions also highlight the limitations of namespace-level logical separation in hostile multi-tenancy scenarios.

In a multi-tenant *Kubernetes* cluster, achieving robust isolation necessitates not only sandboxing containers and the control plane but also systematically leveraging all available native security mechanisms to supplement basic namespace separation [3]. Beyond RBAC, *Kubernetes* offers constructs such as *NetworkPolicy*, *PodSecurity contexts*, *LimitRange*, and *ResourceQuota*, which, when

Table 1: Multi-tenancy classification matrix

	Customer Code Execution?	
	no	yes
Friendly	Least stringent isolation	Moderate isolation
Hostile	Strong isolation	Most stringent isolation

consistently applied, can significantly reduce the attack surface between tenants. Admission controllers play a pivotal role in this process by ensuring that these controls are enforced automatically and uniformly across *Kubernetes* namespaces. By intercepting and validating API requests before they are committed to the cluster state, admission controllers can reject workloads that violate isolation requirements (e.g., *privileged* mode, *hostPath* mounts) and mutate resources to inject protective configurations such as default network segmentation, resource limits, and restrictive security contexts [9, 10]. Automating this enforcement is critical in hostile multi-tenancy scenarios, as it eliminates reliance on manual configuration and ensures that every tenant environment is hardened by default.

In this work, we address these gaps by proposing a *Kubernetes* architecture targeting hostile multi-tenancy that leverages per-tenant virtual control planes, virtual nodes backed by VM-based sandboxes, and automated policy enforcement through admission controllers. Our design aims to achieve isolation levels that make it safer to run untrusted workloads while preserving *Kubernetes* API compatibility.

2 Background

Multi-tenancy refers to the co-location of workloads from multiple organizational or administrative domains on shared infrastructure. The degree of isolation required in a multi-tenant environment is determined by two primary factors: (i) whether the platform executes customer-provided code, and (ii) whether tenants are assumed to be cooperative or potentially hostile. These two dimensions yield four distinct categories of multi-tenancy (Table 1), each demanding different levels of security rigor.

No customer code execution/friendly-tenants is the least stringent case, where the service provider fully controls all executed code and tenants are trusted not to act maliciously. Isolation primarily serves to prevent accidental interference rather than deliberate attacks.

Customer code execution/friendly-tenants entails that arbitrary code is executed, but tenants are assumed to be cooperative, isolation must account for unintentional vulnerabilities or bugs in the code that could impact other tenants. Security boundaries are more relaxed compared to hostile scenarios, focusing on fault containment rather than malicious prevention.

No customer code execution/hostile-tenants implies that tenants cannot directly execute arbitrary code, but their actions or data may still attempt to trigger harmful behaviors in the platform. Stronger isolation is needed to mitigate indirect exploitation (e.g., resource starvation, API abuse).

Customer code execution/hostile-tenants represents the most demanding scenario. Arbitrary code from potentially malicious actors is executed within the shared environment, requiring strong security boundaries, often equivalent to VM-level isolation, to prevent privilege escalation, container breakout, and API exploitation (CVE-2022-0185 [7], CVE-2022-0492 [8]).

As such, we distinguish between four conceptual levels of isolation that progressively offer more stringent protection according to the environment we aim to secure.

2.1 Motivating Example

Consider a Spark-based big data analytics platform offered as a service, where multiple tenants submit arbitrary computation tasks to process tera-bytes of data in parallel to a shared *Kubernetes* cluster. Each job submitted by a customer is processed as follows:

1. Driver Container
   - ◯ The driver container receives the customer's code (e.g., *PySpark*, *Scala*, or *SQL*).
   - ◯ It parses and schedules tasks across the cluster, maintaining execution metadata and orchestrating distributed computation.
2. Worker Containers
   - ◯ Worker containers execute the scheduled tasks on the assigned data partitions.
   - ◯ These containers often operate in a multi-tenant environment, sharing network, storage, or computational resources with other tenants' workloads.

In a hostile multi-tenancy scenario, this setup exposes several risks:

- Arbitrary Code Execution: Malicious tenants can submit code designed to access or exfiltrate data belonging to other tenants if isolation is insufficient.
- Resource Interference: Malicious tasks may overconsume CPU, memory, or I/O, leading to denial-of-service effects for other tenants.
- Privilege Escalation and Container Breakout: If container boundaries or orchestration policies are misconfigured, attackers may attempt to escape the container or escalate privileges on the host node.
- API Exploitation: Malicious code could abuse exposed APIs, such as shared storage endpoints, metadata services, or orchestration interfaces, to gain unauthorized access or disrupt cluster operations.

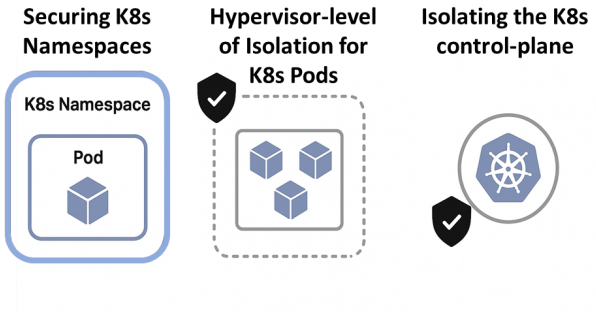


Figure 1: Multi-layer Solution

This example illustrates the most demanding security scenario in multi-tenant analytics services, where arbitrary, potentially malicious code must be executed safely, necessitating strong isolation guarantees and careful control over resource access, API exposure, and inter-container communication.

Kubernetes, as designed, primarily addresses cases where tenants are assumed to be cooperative, leveraging namespaces, RBAC, and network policies for separation rather than strict isolation. In scenarios where hostile tenants and customer code execution coincide, these mechanisms alone are still necessary, but insufficient to provide the security guarantees required for high-assurance workloads.

3 Multi-tenancy Solution

Our *Kubernetes* multi-tenancy solution combines three layers of defense (Figure 1) to achieve stronger isolation in hostile environments.

- **Securing the namespaces** ensures each tenant's scope is locked down with enforced policies via policy engines like *Kyverno* [12].
- **Isolating the pods** using *virtual-kubelet* [13] to schedule workloads into Azure Container Instances (*ACI*) [14]. This eliminates shared-kernel risks by running each tenant's pods in dedicated VM-based sandboxes.
- **Isolating the control plane** through per-tenant virtual control-plane (*vCluster* [15]) providing each tenant with its own virtual API server and control components, preventing interference or unauthorized access to the host cluster and other tenants.

Together, these measures deliver stronger end-to-end workload, network, and control-plane isolation while preserving the *Kubernetes* developer experience. Our Solution integrates the above mechanisms/products coupled with an automated tenant on-boarding process

3.1 Virtual-kubelets Creating Hypervisor-Isolated Containers

A regular *Kubernetes* node runs pods locally, while a *virtual-kubelet* [13] registers as a node with the *Kubernetes* control-plane, but schedules pods to an external provider for isolated execution. At the foundation of our design, each tenant's workloads are scheduled via a *virtual-kubelet* into hypervisor-isolated containers, such as those provided by Azure Container Instances. *ACI* containers are by design sandboxed in isolated utility virtual machines with dedicated kernel and a shared-nothing design where two pods on the same physical machine would not be sharing any OS-kernel, storage or networking resources. Unlike traditional *Kubernetes* nodes, *virtual-kubelets* act as lightweight proxies to external compute resources (a *virtual-kubelet* is a single pod in a *Kubernetes* cluster that masquerades as a node). By associating each *virtual-kubelet* to a dedicated Azure Virtual Network (*VNet*) [17], tenants are strongly isolated at the network level: pods from different tenants cannot directly communicate unless explicitly allowed through *VNet* peering. Even pods in the same *VNet* are subdivided across subnets where firewall rules can govern their communication patterns. This approach ensures that even workloads running on the same physical infrastructure remain logically and network-wise separated [18]. Another major advantage of this solution is that it enables a *serverless* architecture in which it eliminates the need to pre-provision large *Kubernetes* clusters. A core cluster with few nodes, hosting the *virtual-kubelets* allows for the creation of thousands of highly isolated pods. Moreover, Azure Container Instances (*ACI*) Confidential Containers can run workloads inside hardware-based trusted execution environments (TEEs), such as AMD SEV-SNP, which protect data in use by encrypting memory and isolating it from the host OS, hypervisor, and other tenants [19]. This ensures containerized applications can process sensitive data securely in the cloud while maintaining the simplicity of *ACI*'s serverless container model.

Leveraging ACI for pod scheduling and hosting introduces a serverless dimension to the Kubernetes cluster. When the Kubernetes scheduler designates the virtual node for pod hosting, the responsibility for pod creation is transferred to the ACI control plane, which utilizes its own scheduler to identify an appropriate physical machine and instantiate the virtual machine that will run the pod containers.

3.2 Namespace-Level Isolation

For each tenant, a dedicated *Kubernetes* namespace is provisioned as part of the tenant onboarding [20]. This namespace is bound to its own virtual node, which represents the tenant's allocated *virtual kubelet* resources [13]. Within the namespace, isolation is enforced through *Kubernetes*-native mechanisms, including:

- *Role-Based Access Control* (RBAC) rules that limit tenant permissions strictly to their namespace.
- *NetworkPolicies* that control ingress and egress traffic at the pod level.
- *Resource quotas* and *LimitRanges* to prevent resource starvation and enforce fair usage.
- *Security contexts* that constrain container privileges and enforce runtime security boundaries.

These controls are enforced automatically through *Kyverno*, which intercepts all API server requests and ensures that all resources comply with organizational and tenant-specific security policies before creation [12]. By combining namespace scoping

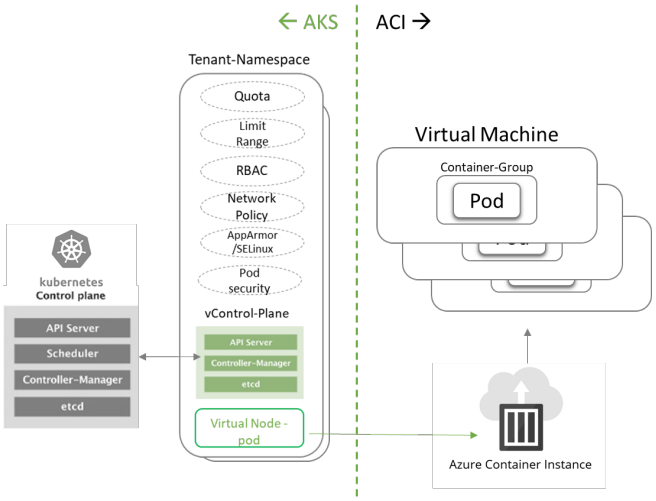


Figure 2: Overview of the proposed multi-tenancy solution, combining virtual-kubelets, vClusters, and namespace hardening for layered isolation

with admission control, the architecture prevents misconfiguration or privilege escalation that could compromise other tenants.

3.3 vCluster Injection for Tenant Transparency

To provide tenants with the illusion of a dedicated *Kubernetes* environment, each tenant's namespace hosts a *vCluster* [15]. *vClusters* are virtualized *Kubernetes* control planes running on top of the shared cluster infrastructure but fully confined to the tenant's namespace. From the tenant's perspective, they interact with an independent *Kubernetes* API server, complete with custom RBAC, resource quotas, and pod scheduling, while in reality, all workloads remain scheduled through the isolated *virtual-kubelet* and controlled namespace. This is needed since the tenant's application that is offered as a service is often managed by a *Kubernetes* operator [21] that is closely integrated with the *Kubernetes* API-server, and therefore the operators need permission to read/write/modify resources using the API-server. For instance, the Spark Operator [22] that manages the creation of Spark Application within the *Kubernetes* cluster requires permission to create/delete pods, service, configmaps, etc. and it is not unusual for developers to give their Operators excessive permissions to *all* operations [22] on the API server. By allocating a virtual control-plane per namespace per tenant, we avoid the implications of excessive permissions. And if the tenant manages to break free of their pod, and attacks the API-server, they will only be attacking their own *Kubernetes* control-plane without impacting other tenants.

The combination of *virtual-kubelets*, dedicated *VNets*, hardened namespaces, and *vCluster* injections create a more stringent multi-layered tenant isolation. Together, these mechanisms minimize attack surfaces and prevent cross-tenant interference, offering a more secure, scalable, and transparent multi-tenant *Kubernetes* environment.

4 The Tenant Onboarding Process

The tenant on-boarding process consists of several steps:

4.1 Namespace Creation and Policy Enforcement

When a new tenant is onboarded, the system first creates a dedicated *Kubernetes* namespace for that tenant. This namespace serves as the fundamental isolation boundary for all tenant resources. The namespace creation event triggers *Kyverno* admission controllers, which automatically inject and enforce all required policies.

4.2 vCluster Deployment

Following namespace setup, a vCluster instance is provisioned within the tenant's namespace (as a group of pods created by the *vCluster* controllers). The *vCluster* is fully scoped to the tenant's namespace, ensuring that all operations remain confined and isolated from other tenants. Each tenant has a unique access configuration for their *vCluster*, preserving tenant autonomy while maintaining overall system security.

4.3 Virtual Kubelet Initialization

A *virtual-kubelet* pod is instantiated in the tenant namespace. This *virtual-kubelet* is responsible for scheduling tenant workloads onto hypervisor-isolated containers, such as Azure Container Instances (*ACI*), providing hardware-level isolation for tenant workloads. In addition, a new *VNet* is created to make sure the tenant's pods have limited communication reach.

4.4 Operator Deployment and Application Scheduling

Finally, a dedicated operator instance is created in *vCluster* to manage the tenant's application. The operator is scheduled on the *virtual-kubelet*, thereby inheriting hypervisor-level isolation. For example, if the tenant requires a Spark cluster, the Spark operator will:

- Create/configure the necessary Spark pods.
- Schedule all application pods on the *virtual-kubelet*.
- Delete/clean-up the resource when the Spark cluster is deleted.

This design ensures that both the operator and the tenant workloads are further isolated from the host cluster and other tenants. By leveraging *virtual-kubelets*, each tenant effectively receives a sandboxed execution environment, while the *vCluster* provides a familiar and fully-featured *Kubernetes* interface.

While our implementation leverages Azure Container Instances (*ACI*) as the sandbox for virtual nodes, the underlying architecture is agnostic to the specific serverless container provider. The *virtual-kubelet* abstraction allows pods to be scheduled onto any compatible serverless backend, enabling integration with alternatives such as AWS *Fargate* [24] or HashiCorp Nomad. Albeit these providers differ in OS support (*ACI* supports both Windows and Linux), and hardware passthrough capabilities (*ACI* supports GPUs and other direct device assignment), our design can adapt to their execution models, ensuring that tenants benefit from secure, scalable,

Table 2: Security risk mitigation across namespace, pod, and control-plane isolation layers.

Attack Type	Namespace Security (Policies)	Pod Isolation (VM-based via Virtual Kubelet)	Control-Plane Isolation (*vCluster*)
DDoS / Resource Exhaustion	Quotas, limits, and NetworkPolicies restrict resource usage within namespace scope.	Abuse is contained within tenant's sandbox VM; cannot starve other tenants.	Each tenant has its own API server; flooding impacts only the tenant's *vCluster*, not the host cluster.
Privilege Escalation	RBAC enforcement and pod security contexts prevent privilege misconfigurations.	Sandbox VM prevents container breakout from escalating to host kernel privileges.	Compromise of tenant API server is isolated; cannot escalate to host control plane or other tenants.
API Abuse	Namespace scoping restricts unauthorized access to resources across tenants.	VM sandbox isolates workloads; API misuse within sandbox cannot impact others.	Each tenant interacts only with its own *vCluster* API server; no access to host cluster APIs or other tenants' resources.
Container Escape	Pod security policies enforce hardened contexts (e.g., no privileged pods, seccomp, AppArmor).	Even if container escape occurs, attacker is confined within sandbox VM, not shared host kernel.	Not applicable directly but prevents escaped workloads from accessing shared cluster-level APIs or impacting other tenants.
Lateral Movement	NetworkPolicies and RBAC block cross-namespace communication and unauthorized access.	Tenant VM acts as a boundary, containing any compromised workload within tenant sandbox.	*vCluster* prevents compromised workloads from pivoting into control-plane components of other tenants or host cluster.

and serverless workloads regardless of the underlying container provider.

5 Evaluation

Security and strong tenant isolation comes at a price; in this section we start by analyzing the resiliency of our solution against six common attack types then we examine the overhead imposed by our solution from multiple facets.

5.1 Security Risk Evaluation

We assess our multi-tenancy solution by examining its resilience to major security threats in *Kubernetes* clusters, including denial-of-service, privilege escalation, API abuse, container escape, and lateral movement (Table 2). These attacks are key risks for tenant isolation and workload integrity.

5.2 Resource Overhead

Each tenant's *vCluster* is a lightweight *Kubernetes* control plane running inside the host cluster in the tenant's namespace. This involves an extra API server and associated components like etcd and controllers per tenant. While minimal, these control planes still consume CPU and memory, requiring a few hundred MB of memory and a fraction of a core per *vCluster*. As the number of tenants increases, the overhead scales linearly. This overhead is mostly constant per tenant and considered "lightweight" for isolation benefits. However, collectively, many control plane instances will use resources that could otherwise be available for application workloads.

Virtual kubelet is deployed as a pod per namespace and hence presents an overhead per tenant. Its average consumption is *half a CPU core* and *500 MB of RAM* managing *100 pods*. The resource consumption can vary depending on the pod configuration (e.g. health-probe frequency, log extraction frequency, metrics extraction etc.).

While *Kyverno's* resource footprint is small, it requires necessary *CPU* cycles, especially under high load or with complex policies. It can be horizontally scaled to avoid bottlenecks. Overall, *Kyverno* adds manageable *Kubernetes* control-plane latency while ensuring consistent enforcement of security and isolation rules. Note that *Kyverno* is used per *Kubernetes* Cluster, not per tenant, and this reduces its overhead.

Dedicated Namespaces are a logical partitioning and by themselves do not add any resource cost, they are considered metadata.

5.3 Latency and Performance Impact

Although resources such as CPU and memory may be adequate, the introduction of additional layers can nonetheless result in increased latency across various operations, including pod startup times, network communications, and API calls.

In a standard cluster, pod scheduling and startup typically take a few seconds. In our multi-layered design, tenant pod creation involves extra steps: the *vCluster* syncs the pod spec to the host cluster, which schedules it to a *virtual-kubelet*, triggering *ACI* to provision the container group (inside a dedicated virtual machine). These steps introduce delays, especially due to *ACI's* asynchronous API and cold-start times. As a result, pod readiness may take *tens of seconds* (depending on the image size, operating system type, network configuration, storage options, and a variety of factors), significantly longer than on real nodes with basic OS level isolation, primarily due to control-plane indirection and *ACI* provisioning latency. There are two ways to mitigate this extra startup latency. (i) use *image caching* in *ACI* which caches the image to be used locally,

thus significantly reducing startup time. (ii) use *ACI* standby pools [23], which creates a pool of pods on standby and enables them when the *virtual-kubelet* creates the pod, this reduces the startup latency to *less than 1 second*.

Once running, a tenant's application performance is *near-native* in many respects, but there are subtleties:

- **Network latency:** Because each tenant's pod network is isolated at the *VNet* level, if a tenant service communicates with another tenant's service, by default it must go through external routes due to security boundaries.
- **Disk I/O and CPU performance:** Running in an *ACI* hypervisor-based container means a slight drop in raw *CPU* and I/O performance compared to running on the host's bare metal. Azure's hypervisor is quite optimized with Hyper-V VMBus devices and Direct Device Assignment (DDA), but any virtualization layer adds some overhead.

As for the API latency, tenants interact with their *vCluster* API server, which runs as a pod in the host cluster. Read operations are fast and served from the *vCluster's* etcd, often faster than a shared cluster under load. Write operations (e.g., creating a Deployment) trigger the *syncer* to call the host API, adding latency to realizing the desired state, though the initial API response is quick.

Kyverno Policy Check also incurs some Latency when creating resources in the host cluster, *Kyverno* validates them via admission webhooks. This adds a *few milliseconds per request*, but bursts (e.g., 50 pods at once) can cause queuing and increase latency. *Kyverno* is scalable and typically introduces minimal overhead, affecting only control-plane interactions and not pod runtime. This is something that can be changed since *Kyverno* can run with multiple replicas.

5.4 Operational Complexity and Maintenance

One significant downside of this multi-layer approach is its *complexity*. The design introduces many moving parts: multiple *Kubernetes* virtual control planes, cross-network integrations, policy engines, and a virtualized node layer.

5.4.1 Multiple Security Layers to Manage. While security isolation is the goal, having multiple layers means multiple policies to manage. You have Azure NSGs (Network Security Groups) or *VNet* rules at the network layer, *Kubernetes* NetworkPolicies, *Kyverno* policies for ensuring things like "no privileged containers", and RBAC rules in both the host and *vCluster*. Maintaining a coherent security policy across these layers is complex.

5.4.2 Troubleshooting Complexity. When something goes wrong, the multi-layered nature makes debugging harder. Consider a scenario where a tenant's pod is not starting. In this setup, you should consider: Is the issue in the tenant's *vCluster* (e.g., their API server didn't create the pod)? Is it the *syncer* failing to create the object in the host cluster? Is it *Kyverno* rejecting the pod due to a policy? Or did the pod get created in *ACI* but failed to start due to an application error? The ops team needs visibility into all layers. Traditional *Kubernetes* debugging tools would need adaptation.

5.4.3 Scaling Complexity. With our design, scaling in terms of number of tenants is not just about adding more instances. Each new tenant adds load to the control and also adds complexity to manage. If we had, say, 100 tenants, we'd have 100 *vCluster* deployments to track, 100 *VNets*, etc. We need automation for rotating credentials in *vClusters* and regenerating certificates, etc.

Other aspects of complexity include learning curve and team expertise, software compatibility and feature gap since we are integrating multiple solutions from different vendors/sources.

6 Related Work

Both gVisor [25] and Kata Containers [26] improve isolation over standard container runtimes, they still operate within a shared host or hypervisor context. Moreover, they do not offer the serverless aspect that *virtual-kubelet* combined with *ACI* offer. A more suitable comparison would be with AWS *Fargate* [24] combined with Amazon Elastic Kubernetes Service. *ACI* uses lightweight utility VMs that support PCI devices, while *Fargate* uses lightweight Firecracker microVMs. Firecracker has its own limitations in terms of limited operating system support (only a subset of Linux Kernels), and Firecracker does not expose arbitrary PCI devices to the guest. This means no direct GPU passthrough, no *SR-IOV* network interface passthrough for ultra-low-latency networking and no *NVMe* device passthrough for native block storage speeds.

The closest related work that surveys a list of alternative solutions for *Kubernetes* multi-tenancy is the work by Attilio [27] where the author gives an overview of the different approaches to harden multi-tenancy in *Kubernetes* over multiple layers. The work is more of a survey of the available techniques than a coherent multi-layer solution that is deployed, tested, and evaluated, and therefore it lacks the rigor and empirical evaluation we present in our solution.

Oksana Baranova's master's thesis [28] focuses on network-layer isolation within multi-tenant Kubernetes clusters, with particular emphasis on the implementation of *service-mesh* technology. The objective is to prevent any network traffic from one tenant reaching another tenant's services, even in complex microservice architectures where dozens of services communicate internally. Baranova recognized that while namespaces and network policies provide basic isolation, a misconfiguration or sophisticated attack might still allow cross-tenant calls. The work mainly targets the networking layer and not the other aspects of isolation.

Zheng, Zhuang, and Guo [29] present a multi-tenant framework designed to improve security. The framework integrates namespace-based resource partitioning, network isolation, and fine-grained access control, while leveraging a multi-layer scheduling architecture. Nevertheless, it does not present a comprehensive multi-layer solution.

7 Conclusion

In conclusion, this multi-layer isolation solution provides robust security for hostile multi-tenancy, at the cost of increased complexity and some performance overhead. We can compare it to the extreme ends: at one end, a single shared cluster (low overhead, low isolation), and at the other, completely separate clusters per tenant (very high isolation, high overhead). This design attempts to approach the isolation of separate clusters while still sharing an underlying platform.

In summary, a *Kubernetes* cluster does not inherently support multi-tenancy. To date, there is no single solution/product that

can remedy this concern. The proposed mechanisms and tools are designed to improve tenant isolation and help mitigate security risks. Any cloud user can use our proposed solution to improve their multi-tenant isolation.

Acknowledgments

Special thanks to the Azure *ACI* team for their insightful input and collaboration. Their expertise and feedback were instrumental in shaping key aspects of this work. The authors acknowledge the use of large language models (LLMs), specifically, the Microsoft Word Copilot's "make formal" feature, as writing assistants to enhance the overall quality and style of the manuscript.

References

[1] Friese, P. A., Eleliemy, A., Haus, U.-U., & Schulz, M. (2025). Closing the HPC-Cloud Convergence Gap: Multi-Tenant Slingshot RDMA for Kubernetes. arXiv preprint arXiv:2508.09663

[2] Liu, P., & Guitart, J. (2022). Fine-Grained Scheduling for Containerized HPC Workloads in Kubernetes Clusters. In Proceedings of the IEEE International Conference on High Performance Computing & Communications; Smart City; Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys), Chengdu, China, December 2022.

[3] Kubernetes Multi-tenancy, Retrieved August 13, 2025, from https://kubernetes.io/docs/concepts/security/multi-tenancy/

[4] S. Nimmagadda, Linux Namespaces and cgroups as OS Primitives for Lightweight Virtualization: Architecture, Isolation Mechanisms, and Performance Evaluation. Turkish Journal of Computer and Mathematics Education (TURCOMAT), 9(2):811–822, 2018.

[5] Gao, X., *et al.* "Breaking the Resource Rein of Linux Control Groups." CCS '19: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019

[6] Dirty COW. 2016. CVE-2016-5195. Retrieved August 13, 2025, from https://dirtycow.ninja

[7] BleepingComputer. 2022. Linux Kernel Bug Can Let Hackers Escape Kubernetes Containers. Retrieved August 13, 2025. https://www.bleepingcomputer.com/news/security/linux-kernel-bug-can-let-hackers-escape-kubernetes-containers/

[8] The Hacker News, New Linux Kernel cgroups Vulnerability Could Let Attackers Escape Container. Retrieved August 13, 2025. https://thehackernews.com/2022/03/new-linux-kernel-cgroups-vulnerability.html

[9] Hussein Younes *et al.* 2025. KubeFence: API Call-Level Filtering to Enhance Multi-Tenancy Security in Kubernetes. arXiv:2504.11126 [cs.CR].

[10] Cloud Security Alliance. 2022. Kubernetes Security Best Practices: Definitive Guide. Retrieved August 13, 2025, from: https://cloudsecurityalliance.org/blog/2022/03/03/kubernetes-security-best-practices-definitive-guide

[11] Kubernetes Documentation. 2025. Admission Control in Kubernetes. Retrieved September 11, 2025. https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/ .

[12] Kyverno. 2023. Admission Controllers and Policy Enforcement in Kubernetes. Retrieved August 14, 2025. https://kyverno.io/docs/introduction/admission-controllers/

[13] Virtual Kubelet. 2025. Virtual Kubelet: Kubernetes Node Implementation Masquerading as a Kubelet. Retrieved August 14, 2025. https://virtual-kubelet.io/

[14] Microsoft. 2023. Azure Container Instances. Retrieved August 14, 2025. https://azure.microsoft.com/en-us/products/container-instances

[15] Loft Labs. (2023). Solving Kubernetes Multi-Tenancy Challenges with vCluster. Retrieved from https://www.loft.sh/blog/kubernetes-multi-tenancy-vcluster

[16] Microsoft.Virtual nodes on Azure Container Instances (AKS). Retrieved August 14, 2025. https://learn.microsoft.com/en-us/azure/container-instances/container-instances-virtual-nodes

[17] Azure Virtual Network, Retrieved August 14, 2025. https://learn.microsoft.com/en-us/azure/virtual-network/virtual-networks-overview

[18] Network isolation in Azure Kubernetes Service. Retrieved August 14, 2025. https://learn.microsoft.com/en-us/azure/aks/concepts-network

[19] ACI Confidential Containers, Retrieved August 14, 2025. https://learn.microsoft.com/en-us/azure/container-instances/container-instances-confidential-overview

[20] Kubernetes Namespaces. Retrieved August 14, 2025. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

[21] Operator Pattern in Kubernetes. Retrieved August 14, 2025. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

[22] Spark Operator RBAC permissions: build-tools/helm/spark-kubernetes-operator/templates/operator-rbac.yaml, Retrieved August 14, 2025. https://github.com/apache/spark-kubernetes-operator/

[23] ACI standby pools, Retrieved August 14, 2025. https://learn.microsoft.com/en-us/azure/container-instances/container-instances-standby-pool-overview

[24] Amazon Web Services. 2024. AWS Fargate. Retrieved August 14, 2025, https://aws.amazon.com/fargate/

[25] The Container Security Platform gVisor, Retrieved August 14, 2025. https://gvisor.dev

[26] Open Infrastructure Foundation. 2024. Kata Containers. Available at: https://katacontainers.io

[27] Attilio Oliva, 2024. Multi-Tenancy in Kubernetes Clusters. Master's thesis, Politecnico di Torino. Retrieved August 14, 2025: https://webthesis.biblio.polito.it/secure/33340/1/tesi.pdf .

[28] Baranova, O. 2021. Multi-Tenant Isolation in a Service Mesh. Master's thesis, Aalto University. Available at: https://aaltodoc.aalto.fi/items/58a0bd81-9d46-4606-b665-d2623ead3f41

[29] Zheng, C., Zhuang, Q., and Guo, F. 2021. A Multi-Tenant Framework for Cloud Container Services. arXiv preprint arXiv:2103.13333. Available at: https://arxiv.org/abs/2103.13333