Secure Multi-Tenant Application Deployments in Kubernetes

Anil Kumar Manukonda

anil30494@gmail.com

Abstract

Multi-tenant deployments in Kubernetes make it simple for businesses to allocate cluster resources between teams or customers, while offering safety and avoidance of interference. We review important architectures and secure measures for running multi-tenant applications on Kubernetes used in the financial industry and within the scope of PCI-DSS. We explore the issue of shared clusters, identifying crossover threats and teaching you how to prevent them. After that, we discuss how clusters can be organized, comparing those shared by many users with those used by only one, along with how clusters, namespaces, virtual clusters and node isolation are used. Full coverage is given on security arrangements, for example, namespace organization, Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC), restricting access via Kyverno, OPA/Gatekeeper, network separation by Kubernetes Network Policies and service mesh and protection for pods (Pod Security Standards, seccomp, SELinux/AppArmor). We also show how to use Kubernetes on a multi-tenant platform by presenting a case study of a fictitious bank with example configurations (YAML code) for namespaces, RBAC roles, network policies and admission policies. The paper shows images of a secure shared architecture, different service mesh designs for each tenant and how CI/CD pipelines take care of supporting multiple tenants. We also include diagrams that show the process an attacker could take on a shared cluster and the way new tenants are introduced. We discuss what is coming next such as different multi-tenancy options, uses of sandboxing and changing compliance rules and how these will affect the way financial institutions manage Kubernetes multi-tenancy. This information, though sometimes personally viewed, is guided by practical DevSecOps skills designed to support both Kubernetes platform engineers and cloud security architects in building safe, compliance-based multi-tenant Kubernetes environments.

Keywords: Kubernetes, Multi-Tenancy, Secure Deployment, Namespace Isolation, Role-Based Access Control (RBAC), Attribute-Based Access Control (ABAC), Kyverno, OPA Gatekeeper, Network Policies, Service Mesh, Pod Security Standards, Seccomp, SELinux, AppArmor, PCI-DSS Compliance, Node Isolation, Virtual Clusters, Resource Quotas, Cluster Management, Policy Engines, Admission Controllers, Least Privilege, Network Segmentation, Container Security, CI/CD Pipelines, DevSecOps, Monitoring, Audit Logging, GitOps, Capsule, Hierarchical Namespaces, Istio, Mutual TLS (mTLS), Financial Services, Compliance Automation, Confidential Computing, Zero Trust, Tenant Onboarding, Cluster Hardening, Policy as Code, Security Context Enforcement, Pod Anti-Affinity, Node Pools, Taints and Tolerations, Runtime Security, Kubeaudit, Kube-bench, Cloud Security, Future Trends, Sandboxing, WebAssembly, Storage Isolation, Multi-Cluster Management, Regulatory Compliance, Cluster Scalability, Cluster Automation, Cluster Security Tools

Introduction

While Kubernetes was built for running one tenant, many organizations now handle multiple requirements by placing different applications, teams or customers on the same cluster. Kubernetes uses multi-tenancy in ways such as letting several internal groups use the same cluster or allowing one SaaS company to run its application for many customers all using the same cluster. Using multi-tenancy in the financial services industry is helpful for controlling costs and managing in one place, but brings up major security and compliance issues. Processing sensitive information (like credit cards under PCI-DSS) means financial institutions must ensure tenants are well segregated, because a failure to isolate may cause data exposure and violation of the rules [3].

Because of PCI-DSS compliance, access between card data handling systems (CDE) and other systems must be separated. In other words, workloads requiring PCI compliance need to be cut off from other tasks in the cluster. Lacking isolation, tenants and the provider would have to be PCI compliant, an almost impossible state to manage. According to the PCI Security Standards Council, without separate segmentation, every tenant using a cloud service and the service provider must prove they are PCI compliant. That's why it's important for financial institutions to have strong segmentation in any multi-tenant approach [3].

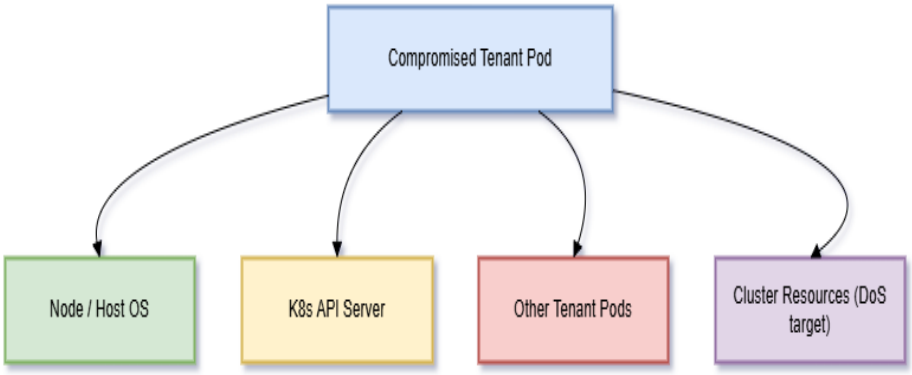
Besides, financial organizations use shared Kubernetes platforms both to save costs and operate more quickly in the future. Using multi-tenancy helps cut down on the work overwhelmed from operating multiple single-tenant clusters. The plan is to use security isolation without making it too costly. Kubernetes gives fundamental ways to isolate multi-tenancy (namespaces, role-based access control, network policies and so on), but these require additional tools and practices to make the system fit for use by high-security industries.

The structure of this paper is explained below. Our first task is to create a threat model for multi-tenant Kubernetes which highlights which parts of a cluster workloads are sharing. We then examine how architectures are designed: we study both types of multi-tenancy isolation (soft vs. hard), including the use of isolated clusters, clusters divided using namespaces and the use of virtual clusters. After that, we explain central security measures: how to organize namespaces correctly, set up RBAC/ABAC so users only get what they need, enforce security policies with policy engines, isolate networks at pod or service level and securely configure pods. We show how to use a Kubernetes multi-tenant approach in PCI compliance with financial data, using real YAML examples. We have made use of tables, diagrams (architecture, service mesh, CI/CD process) and flowcharts (attack surface, tenant access process) as examples to illustrate what we discuss in the paper. Finally, we explore upcoming trends and share suggestions on the rise of sandboxed containers, better multi-tenant tooling and recent security suggestions for Kubernetes in cloud-native financial environments.

Threat Model for Multi-Tenant Kubernetes

The assumption in multi-tenant clusters is that any tenant's workload might be compromised, perhaps due to a vulnerable application being used by an attacker. For the cluster to be safe, containment is key, so that a problem in one tenant doesn't affect any other tenant or cluster control plane. If an attacker gets into a pod in the normal way, they will meet different isolation boundaries – container, pod, namespace, node and so on – that they must get through to further their activity or become more powerful. When attacking, the primary goals of an adversary are: taking another tenant's data, draining or using up other tenants' resources or making the cluster unavailable through a Denial of Service (DoS) [4].

Attack Surfaces in a Shared Kubernetes Cluster: An explanation of how compromising a tenant workload could be carried out is provided in the table below. An attacker in an affected pod may execute several tactics at once: gaining access to the host system, using the Kubernetes API, spreading within other pods or flooding resources to cause a DoS attack.



Flowchart: Major attack surfaces when a tenant's pod is compromised in a shared cluster.

As can be seen, the primary isolation boundaries and attack areas are those listed above.

- **Container/Pod Boundary:** Kubernetes handles applications by putting them in pods which each can hold one or just a few containers. Because of isolation (namespaces, cgroups), the container runtime and Linux kernel keep processes running on different system resources. Attackers can use a kernel security problem or incorrectly configured container system to break free of the container into the host OS. When successful, they can go to other pods across that node or use those pods' credentials. To mitigate, update your runtimes, harden the Linux kernel and keep all host ports as few as possible (covered later).
- **Namespace & RBAC Boundary:** By using namespaces, Kubernetes objectives are separated and by adding RBAC, accounts have fewer things they can access. An attacker could use poorly designed RBAC policies or leak details of an account token in a pod to access more resources than allowed in their namespace. In such a case, an attacker might use the Kubernetes API to read or modify resources that are not part of the specific tenant, thanks to the overly wide permissions of the pod's service account. For this reason, workloads must use strict RBAC and few permissions in service accounts to prevent attackers from taking over more by way of the control plane [4].
- **Network/Service Boundary:** By standard, all Kubernetes pods are able to connect to each other unless access is blocked. Someone launching an attack may look for services in other workspaces and try to join them. Should NetworkPolicies not be present, pods or databases within another tenant can be exposed and accessed directly. That's why users have to block cross-tenant traffic through

policies by default: every pod can speak to any other pod in the cluster unless you designate a policy for blocking. Even under L3/L4 policies, a smart attacker could still use the L7 elements with the cluster (including the cluster DNS service) to pick up information. An example use case is finding services which you can do by looking up the *.svc.cluster.local records in the cluster's DNS. If clusters use one DNS routing and domain, proper segmentation might need unique DNS or by blocking such queries in a secure multi-tenant system.

- **Node/Workload Boundary:** Should multiple pods share a node, an escape from a compromised container offers the attacker access to different tenant's processes on that node (or even the ability to monitor its resources). If the attacker is not fully escaped, it might instead use up a victim's CPU, memory and disk capacity which makes the victim unavailable to others. When working with multiple tenants, it's a good idea to keep them separate by node pools or taints to avoid major problems if a node is compromised. If an attacker gets into a pod, they will only impact other pods in the same tenant group if those pods are using the same nodes. Really, isolating each node at the tenant level can mean nothing but making certain nodes work for a single tenant (taking into accountscheduling), though this leads to less pooling of resources.
- **Shared Infrastructure/Control Plane:** Each tenant shines some control plane components (APIs, controller) onto the other tenants, along with other secondary services like DNS, network configuration (CNI) and ingress control plugins. Because APIs and etcd control a lot of access, attackers could exploit them or unprotected etcd and walk freely through all your data in the system. Though the control plane is usually tough and not used by tenants, issues still exist: a webhook with cluster-admin privileges running for a tenant could harm all other tenants. Being sure that tenants are unable to change most cluster scoped objects or use resources like ClusterRoles and increasing the security of admission system security, are both needed for the threat model. Because of this, protection against attacks on the control plane constantly needs to be considered – if a tenant combines requests into large volumes, it could generate overload or overuse resources (so rate limits and resources quotas are a must).

All things considered, the threat model sees the cluster as a shared fate system, because one weak point in any layer can result in a cross-tenant attack. An attacker's purpose is to discover the weak part of the security boundaries. The goal is for us cluster operators is to secure each cluster layer – container runtime, pod security context, namespace/RBAC policies, network segmentation and control-plane protections – so that a single failure can't risk all the tenants. Next, we explain how to handle these threats with practical architecture and configuration approaches.

Architectural Considerations

Before designing multi-tenancy in Kubernetes, you should pick an isolation model that keeps your tenants' applications safe without making your job too tough or costly. Mostly, we make a difference between cases where tenants are logically separated (soft multi-tenancy) and cases where they are fully separated by physical infrastructure (hard multi-tenancy). Models vary according to the level of trust in tenants such as between internal teams and external customers or less significant and highly significant workloads. Financial institutions tend to separate work that needs to be trust or compliance-compliant from workloads not belonging to the same category, using hard multi-tenancy [1].

Single vs. Multi-Cluster Strategies:

The primary choice made in most cases is if one cluster can be shared between multiple tenants or each tenant has their own cluster. Every strategy has benefits as well as weaknesses:

- **Single Cluster (Shared):**User applications are all in a single cluster and generally restricted to their assigned namespaces. Resource efficiency rises and it's necessary to provision and run fewer clusters for management. Tellingly, it also lowers the work needed for same policies and procedures across different tenants and homes (one management system overseeing all). But, because many systems are running from one cluster, a single mistake could risk the entire tenant community. It takes great attention to set up RBAC, network policies and other measures to get close to the isolation typical of independent clusters. Although soft multi-tenancy may be fine within an organization, insider matters or mistakes can and should be prevented by clear policies. Where the compliance rules touch all workloads, having both in-scope and out-of-scope workloads housed together means the security of the entire cluster must be better managed or it risks failing the audit.
- **Multiple Clusters (Isolated):**There is one dedicated cluster for every type of tenant (or every sensitivity level). This is the quickest way to be sure your infrastructure is isolated – the cluster becomes your security boundary. Because tenants don't share any Kubernetes components, they do not influence each other. To comply with PCI, you may keep CDE applications on their own cluster, separate from all the other business applications, so the scope is reduced. More cost comes from using more controllers and complicated operations take place when a single cluster cannot cover both deployment and usage. As a result, the same policy must often be put into place on each of the clusters. For example, multi-cluster managers and GitOps help synchronize cluster configurations and so reduce unnecessary work.
- **Hybrid Approaches:**Many organizations decide to group customers by seeing how much risk they present or what environment they represent. The bank can set up clusters so that PCI-sensitive workloads all share a cluster and non-sensitive internal apps stay on a different cluster – grouping by trust levels for limited multi-tenancy within a cluster and keeping all high-risk work on entire separate clusters. A separate strategy uses a multi-tenant shared cluster for lower test and development purposes and single-tenant clusters for higher production isolation of essential tasks.

The strategies outlined above are detailed below in this table:

Comparison of Single Shared Cluster, Multiple Clusters and Hybrid

Strategy	Pros	Cons	Use Cases
Single Shared Cluster	- High resource efficiency (all resources pooled) - Easier centralized management (one control plane) - Consistent policy enforcement in one place	- Weaker isolation: risk of cross-tenant impact if isolation mechanisms fail - Harder PCI compliance (all tenants in scope unless segmented) - One cluster = single point of failure (control plane outage affects all tenants)	Internal multi-team deployments with some trust SaaS with many low-risk tenants (with strong policies)
Multiple Clusters (per tenant or group)	- Strong isolation by design (cluster as boundary) - Separate control planes (no noisy neighbor on API) - Flexible customization per tenant (different cluster configs)	- Higher costs (duplicated infra, idle resources) - Operational overhead (upgrades, monitoring, etc. multiplied) - Harder to enforce global policies uniformly (needs federation or automation)	Different security levels (e.g., separate PCI cluster) External untrusted tenants (managed K8s as a service) Cases where strict compliance isolation is required
Hybrid (Few clusters by trust level)	- Balances security and cost (isolates only where needed) - Groups similar tenants together (some sharing) - Limits blast radius more than single cluster	- Still some risk within shared groups - More clusters to manage than single, though fewer than per-tenant - Might need complex network setup if those clusters interconnect	Enterprises segmenting prod vs non-prod or by department Combining tenants with equal data sensitivity

Table 1: Comparison of Single Shared Cluster, Multiple Clusters and Hybrid

You often see hybrid strategies being used in financial services. By way of example, although all development and QA environments may use the same clusters to keep prices low, payment systems are put onto dedicated clusters to comply with regulations. Should a cluster serve both challenging and non-challenging workloads, all PCI controls must be applied to the entire cluster which is very demanding.

Because of this, making sure PCI-regulated workloads run only on their cluster or node pool is something employees commonly choose to keep workloads in compliance [2].

Namespace Isolation and Hierarchy

Within clusters, namespaces are what defines each tenant's resources. With namespaces, teams can give their apps the same name and resources are gathered logically. Multi-tenancy requires them and it's best practice to have "each tenant's resources in a separate namespace" as they are used. Roles from the RBAC namespace and policies from NetworkPolicies can be used then to isolate that Kubernetes tenant. Each client using SaaS is offered their own set of namespaces for their applications. On its own, Kubernetes does nothing to isolate one namespace's pods from another which means we have to design our namespaces with appropriate policies to prevent interference [6].

A challenge is finding a way to map your company's structure to namespaces. Do teams within the same business share a namespace or do they get both development (dev) and production (prod) namespaces separated from others? For external multi-tenancy, one customer might be assigned just one namespace. Things can be flexible and it's fine if some hierarchies apply: often, one business unit can include many project namespaces. You can't use Kubernetes by itself to make hierarchy in namespaces, but the Hierarchical Namespace Controller and other tools help you set up hierarchies to delegate and group your namespaces.

- If you go for one namespace for each tenant application, it gives complete app-level isolation: every instance is able to have policies that won't clash with others. You find this situation very often in SaaS services that have multiple customers. Yet, if a single tenant is spread across several apps, it may require either multiple namespaces or good role control within a single namespace.
- If you give every tenant a separate namespace for all their apps, administration is simple for that tenant, but you don't have detailed control. Everything a tenant does is based on the same shared policies. This is acceptable for small users, but things become problematic if different tenants' applications have different risk levels or administrators.

Using a hierarchical namespace, a tenant can get a main namespace and additional namespaces created for every app or environment. What Capsule (from Clastix) does is to create a Tenant custom resource which manages namespaces and sets up restrictions between different tenants. With Capsule, the cluster is divided by placing namespaces in different "areas" and setting policies, so different tenants do not interfere, including at the cluster-wide resource area (for example, it limits who can create CRDs and who can use ingress for each tenant).

Example: We'll take the example that "Tenant A" is a payment processing team that has to use dev, test and prod areas. It's possible to set up namespaces by using tenant-a-dev, tenant-a-test and tenant-a-prod. All the features are labeled to show they represent Tenant A. At this stage, we would use an operator or automation to provide certain default protections (network policies and resource quotas) for any new tenant: A namespace. Kyverno can make these kinds of tasks automated by acting when namespaces are marked with a tenant label (e.g., it can automatically generate RoleBindings and NetworkPolicies) [6].

It's important to also consider multi-tenancy when setting up resources for a cluster. It's not possible for namespaces to naturally separate items like CustomResourceDefinitions (CRDs), cluster roles or node resources across all devices in the cluster. Lack of ability on the part of tenants to change CRDs or Node

settings makes shared clusters hard. In this mode, central control over cluster-wide operations is typical – tenants only access very limited options in terms of resources. If customers require customization, this work must be done for them by cluster administrators or by upgrading to a cluster that runs entirely its own set of control plane resources.

Virtual Clusters / Control Plane Isolation

The solution sets up a (virtualized) API server and control plane for every tenant, with pods on the same set of shared worker nodes. Kubefed is a new model sometimes called Kubernetes on Kubernetes. One example is Loft's vCluster, in which Red Hat's HyperShift and Kamaji which all place the Kubernetes control plane in a pod inside the supercluster and share objects between the supercluster and cluster below it. One plus of tenants is that their API servers and etcd are isolated from each other so no one tenant can view the resources of others. The control plane is designed such that problems caused by misconfigurations or noisy neighbors are isolated to each tenant. The problem is that it is more complex to set up and use networking than classic containers and all of Kubernetes' functions might not be fully available (yet) in the virtual environment. Virtual clusters make it simple for multi-tenant users to take on admin-like tasks in their own cluster, while you still protect the main host cluster you manage. When it comes to financial services, virtual clusters allow each app team to work independently (even running modified Kubernetes versions or individual admissions within their control plane) as the platform team looks after the security of all worker nodes.

Many in the CNCF community have spotted the benefit of having standard multi-tenancy patterns. Indeed, a working group from Kubernetes has put out guidance and created a reference architecture that divides multi-tenancy solutions into two types (based on namespace or using the control plane). When selecting an architecture, a tenants trust analysis is needed: For instance, are each tenant inside the same security zone? Do you run any unknown or externally developed code in your software? If you find grouping those workloads into a dedicated or virtual cluster is not possible, then make sure each of those applications has dedicated nodes to itself.

Resource Quotas and Fairness

Using multi-tenancy adds both security and helps to isolate the noise between residents. We need to stop one tenant from controlling too many resources in a compute cluster, regardless of why it happens. We rely on Kubernetes ResourceQuota and LimitRange objects to do this properly. A ResourceQuota allows you to define the maximum CPU, memory and so on, a namespace can consume. In some cases, you may decide to set each tenant's quota based on what they pay for and the size of their organization. When a tenant exhauses their resources, it can block other tenants in much the same way as a DoS attack. No matter what a tenant does, quotas ensure they cannot take away more resources from the cluster than placed in their special account.

Also, allowing Kubernetes Priority Classes and setting the PodPriority and Fairness for the scheduler (using PriorityLevelConfiguration for API QPS) can make it so that only requests in line are sent at any one time by each tenant. If a business applies this principle to finance, key applications such as fraud detection pods, are given the highest priority, while other, less critical applications are put last, avoiding any eviction or slowness caused by other tenants.

It's also a good idea to add multi-tenancy to build/CI pipelines as we'll discuss in the CI/CD diagram. Tenants may also divide the work by using shared CI/CD infrastructure and deploying to the cluster. Splitting up things like credentials and where the deployment happens is key to the complete multi-tenant design.

In short, secure multi-tenant Kubernetes is designed by stacking different isolation mechanisms. The sections that follow will explain the actual security steps used to build these layers in a cluster.

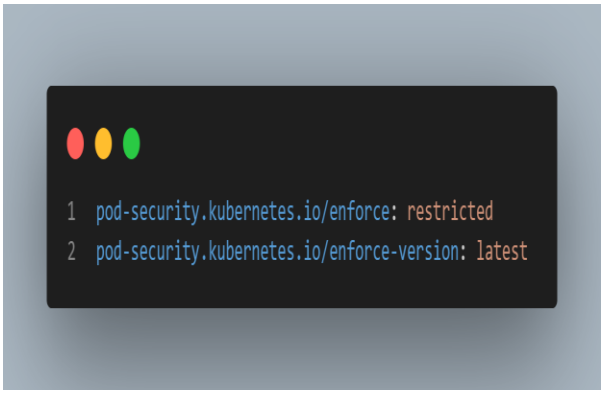
Security Mechanisms

Having decided how multi-tenant architecture will be built, we look at particular security measures that ensure each tenant operates independently. Part of this is namespace design (already introduced), the implementation of RBAC/ABAC for managing who can access resources, policy engines (Kyverno or OPA Gatekeeper) for rules, configuring network isolation and lastly, tightening access at the pod level with Pod Security, seccomp and Linux security modules. All mechanisms target one or more points in the threat model and together they add depth to the security for multi-tenant clusters [8].

Namespace Design and Baseline Configuration

It is important to put some core security in place on namespaces as the tenant boundary right from the beginning. It makes sense to standardize the setup of namespaces, so each new tenant namespace always gets the right labels, annotations and other child resources:

- **Labels identifying the tenant** – e.g., tenant: X – are useful for policy engines and network policies to select all pods of a tenant. If you use a naming pattern that puts the tenant name first for namespaces, that helps manage and control your resources.
- **ResourceQuota and LimitRange** – Determine at namespace creation which standard resources should apply for your pods. By way of illustration, when you make the tenant-a-prod namespace, give a ResourceQuota that controls CPU/Memory as well as counts for objects (such as pods and services). As a result, a tenant won't be able to generate a huge number of pods or exhaust all the memory. Quotas in financial orgs are sometimes given based on how the services will be used or what level of support employees need.
- **Default NetworkPolicy** – a crucial step: By default, Kubernetes namespaces let all traffic move in and out. People in multi-tenant clusters should always start with a "deny by default" NetworkPolicy in all namespaces as standard practice. Most of the time, all pods within the namespace are selected by this policy and all traffic going to or from them is denied unless whitelisted. Subsequent policies may be applied to permit convenient traffic such as between employer containers or with public services. The next section, Network Isolation, shows us an example.
- **Pod Security standard (via labels)** – Under Kubernetes Pod Security Admission, you can enforce a high or low level (such as "baseline" or "restricted") for pods in a specific namespace just by labeling that namespace. For instance:



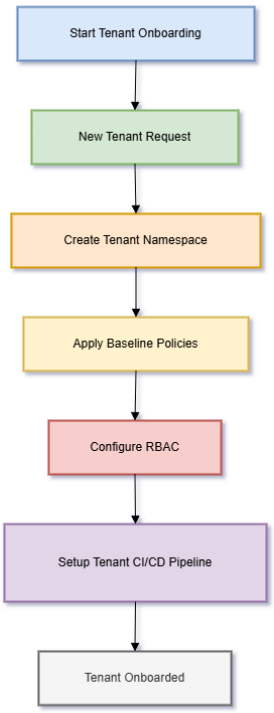
Code snippet: Kubernetes Pod Security Admission Labels (Restricted Mode)

naming a namespace this way guarantees that only well-behaved Restricted Pod Security Pods will function in it (privileged containers are not allowed – see Pod Security section for details). Many multi-tenant clusters choose enforce: restricted for all tenant namespaces, with exceptions only where absolutely needed (and those can be managed via an exemption or separate namespace) [4].

- **Audit and Logging hooks** – Naturally, namespace should be integrated with logging. Setting up a Fluentd/Logstash DaemonSet on every node can give logs the details of the namespace and tenant and send them all to a central place. It doesn't happen when making namespaces, but during setup you ensure each tenant's actions will leave an audit trail. Namespace information should appear in both Kubernetes events and audit logs, making it handy to use in filtering tenants during incident work.

By automating the above, you reduce the chance of human error when onboarding a new tenant. As explained, you can use Kyverno policies to help by noticing new Namespace objects and then starting additional ones. For example, Kyverno can have a ClusterPolicy that says: "when a Namespace is created with label tenant, generate a default NetworkPolicy and ResourceQuota in it". This kind of policy-based onboarding ensures consistency (see flowchart of tenant onboarding below).

Tenant Onboarding Workflow: The process for bringing on a secure tenant can be seen in this diagram:



Flowchart: Tenant onboarding lifecycle in a multi-tenant Kubernetes platform, showing creation of namespace and application of policies.

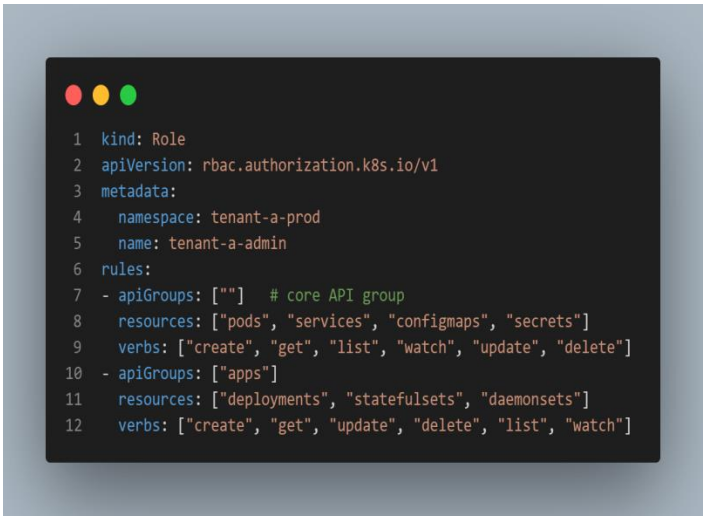
From a regulation perspective, this means the same cubed; compliance auditors can see that every namespace with sensitive data is given the same controls straight away, so mistakes are less likely.

RBAC and ABAC: Access Control in a Multi-Tenant Cluster

With Kubernetes, Role-Based Access Control (RBAC) is the main way to prevent users and service accounts from doing any more than they should on any given resource. Ensuring the correct design of RBAC rules is a major part of keeping a multi-tenant cluster safe.

With RBAC, you normally develop Roles (for specific namespaces) or ClusterRoles (for the entire cluster) and tie subjects (users, groups or service accounts) to the roles. Best ways to achieve multi-tenancy are:

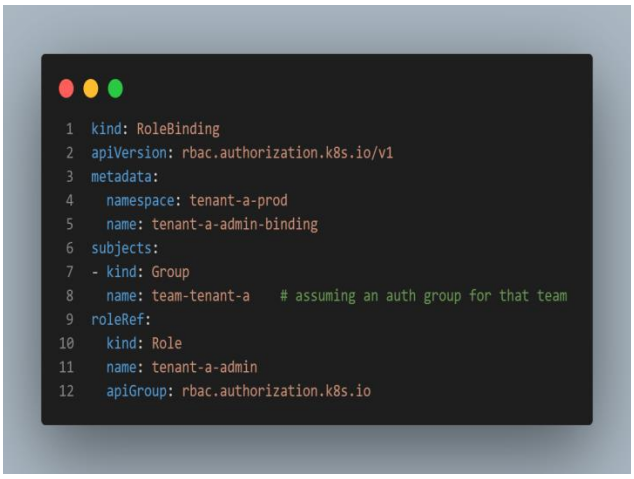
- **Least Privilege per Namespace:** Every tenant deserves to have a Role (or set of Roles) that gives them only the privileges they require in their own space. In some cases, a tenant's developers may create and manage Deployments, Services and similar things in their namespace, though not in others. An easy example of a Role is:



```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: tenant-a-prod
  name: tenant-a-admin
rules:
- apiGroups: [""]   # core API group
  resources: ["pods", "services", "configmaps", "secrets"]
  verbs: ["create", "get", "list", "watch", "update", "delete"]
- apiGroups: ["apps"]
  resources: ["deployments", "statefulsets", "daemonsets"]
  verbs: ["create", "get", "update", "delete", "list", "watch"]
```

Code snippet: Kubernetes RBAC Role for Tenant Namespace Administration

The hypothetical Role can use shared resources widely, though only in tenant-a-prod. You would attach it with RoleMembership to the tenant's user group.



```yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: tenant-a-prod
  name: tenant-a-admin-binding
subjects:
- kind: Group
  name: team-tenant-a    # assuming an auth group for that team
roleRef:
  kind: Role
  name: tenant-a-admin
  apiGroup: rbac.authorization.k8s.io
```

Code snippet: Kubernetes RoleBinding for Team-Based Access Control

For this reason, if any user from Tenant A attempts access to Tenant B's namespace, RBAC will block it.

- **No privilege escalation:**Stop anyone from being able to handle the central RBAC management unless it is intended. If you let tenants make RoleBindings in their area, they could use this privilege to tie their Role to a better ClusterRole and take on higher permissions. In many cases, operators prevent DCOPs from altering RBAC by giving these changes only to cluster admins or by using Gatekeeper preventing binding to cluster-admin roles. Because default permissions don't let users in the Kubernetes API to create roles outside their own namespace, any issues with RBAC rules could accidentally grant that access. Be sure that roles you provide don't have sensitive words for sensitive things (like making new roles or accessing secrets that aren't needed for the role).
- **Avoid ClusterRole where possible:**Any ClusterRole is valid in every namespace which is why assigning a tenant one must be done cautiously. Usually, users on their own won't have cause to open clusters. One situation in which ClusterRole helps is giving unrestricted view of specific global resources such as nodes, if you want to let users monitor metrics. Most of the time, use namespacedRoles for tenants. Should the use of ClusterRoles allow for sensible reuse, RoleBinding can set ClusterRoles to apply just to a namespace. For example, editClusterRole comes with Kubernetes and you can restrict it to one namespace so only the right people can edit in that area.
- **Service Accounts least privilege:**Workloads inside tenants' namespaces all use service accounts. A default service account is included in each namespace by default, still you could make extra service accounts for individual applications. Place limitations on these service accounts as well. Take a look at example since some applications need access to ConfigMaps in their namespace. Here, assign the Role and bind the SA together for this purpose. Also, turn off the automatic insertion of service account tokens for all pods that won't make API calls. It's accomplished by changing the pod spec to set automountServiceAccountToken: false. As a result, if the pod is attacked, the attacker does not possess Kubernetes API credentials.
- **ABAC (Attribute-Based Access Control):**Kubernetes in older versions worked by using ABAC mode, set with a policy file. RBAC has largely taken the place of RBAC, so it is rare to find it in current cluster deployment. For RBAC, a few ABAC features may be added on using layering. An approach could be to use claims from OIDC external tokens to create groups or create Gatekeeper policies that function just like ABAC. A method of using both is by setting "conditions" in Gatekeeper or through 3rd party webhooks: for example, only let users with a particular attribute visit specific namespaces. For purposes of this paper, we concentrate on using RBAC since it is the preferred and suggested approach. Since the user wondered, ABAC is explained here, even though it's rarely enabled in a multi-tenant cluster because it's not dynamic and gets hard to maintain at scale.
- **External AUTH and Project APIs:**Platforms like OpenShift go further by placing containers in Projects and adding a special UI for tenant project administrators to grant permissions within their namespaces. You can manage this in vanilla Kubernetes using tools or with a controller and with Capsule, tenants can be given permission to add their own roles. Do not outsource your security to just anyone.

To verify RBAC isolation, Kubernetes's auth can-i tool is invaluable (e.g., run kubectl auth can-i --as <tenant-user> --list to see what they can do). Additionally, audit logs should be monitored to flag any denied requests or suspicious access patterns (like a user from one tenant trying to list another's secrets).

Both least privilege (PCI Requirement 7) and separation of duties are supported by the use of RBAC. You should show that only approved personnel (or service accounts) can get to cardholder data workloads.

As a case in point, non-PCI app operators ought not be granted access to pods or the ability to run commands in the PCI namespace. Both roles and namespaces are kept completely distinct with RBAC.

Policy Engines: Kyverno and OPA/Gatekeeper

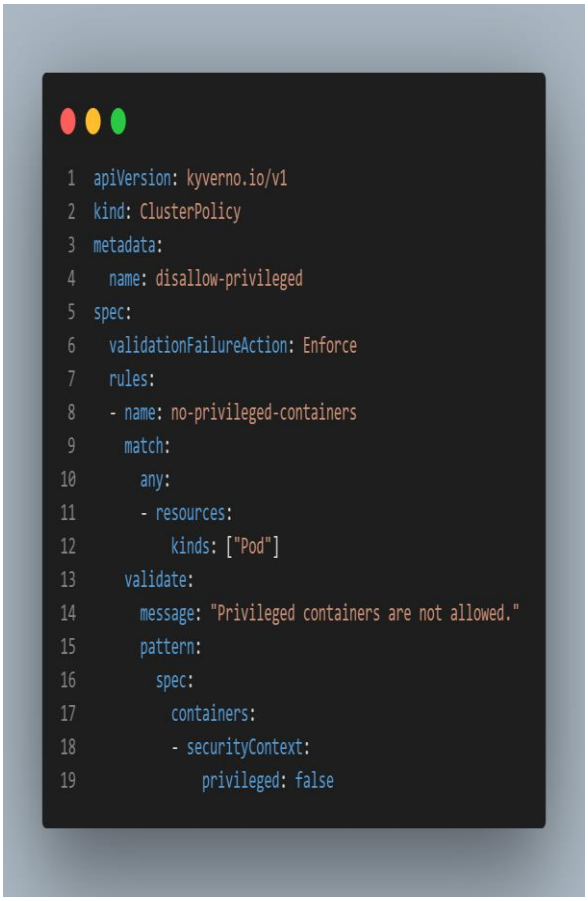
Although standard Kubernetes rules help, more complex multi-tenant situations usually call for creating your own rules. This could include needing certain containers and the workloads not to have the root access, using only a given list of ingress classes, assigning devices to workloads through specific labels and so on. That's when Kyverno and OPA Gatekeeper enter the picture. They examine and apply (or even edit) Kubernetes resources by checking that their structure follows the rules [7].

Kyverno is designed for Kubernetes with YAML polcies (Custom Resources) and straightforward pattern syntax. It's easy for platform teams to work with. OPA Gatekeeper employs OPA engine, policies created with Rego and set up via ConstraintTemplates and Constraints.

Both solutions can review files for compliance at login and as well as uncovering any non-compliant variables in existing resources during oversight mode.

Many companies use policy engines for different multi-tenant challenges.

- **Security Context Enforcement:**Make sure that no pod is running with privileges and that allowPrivilegeEscalation=false is set in all containers or that you add at least AllowPrivilegeEscalation: false to the node's kubelet flag. In Kyverno, there's a policy already added that prevents the use of privileged containers. Below I have listed an example of what a Kyverno validate policy could be like:



```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: disallow-privileged
spec:
  validationFailureAction: Enforce
  rules:
  - name: no-privileged-containers
    match:
      any:
      - resources:
          kinds: ["Pod"]
    validate:
      message: "Privileged containers are not allowed."
      pattern:
        spec:
          containers:
          - securityContext:
              privileged: false
```

Code snippet: KyvernoClusterPolicy to Disallow Privileged Containers

Any new Pod (or those created by workload controllers) will be checked to ensure that the privileged flag in the securityContext is false (or not present which Kyverno treats as false through pattern matching).

We can also use runAsNonRoot: true, specify certain user IDs and so on. Note that some of these are available through the "restricted" profile in Pod Security Standards and customers can add even greater controls using custom policies.

- **Multi-Tenancy Guardrails:**For this reason, tenants should not be permitted to use specific host namespaces or hostPath volumes that risk disrupting their own implementation. All policies in Gatekeeper's library address issues by, for example, forbidding access to hostPID, hostNetwork and hostPath (with a few possible exceptions). It is also possible to stop all users from using LoadBalancers, except for certain tenants to keep the costs down.

- **Namespacing and Labeling:**Require resources to use labels that identify who is using them. All the pods should include a label tenant=<ns-name> for making logs more consistent when using. Instead, make it necessary for all Namespaces to include assigned labels which might be of use to others or organizing processes.

- **Network Policy existence:**is able to implement this by looking for NetworkPolicy objects related to the namespace and if none are found, either stop deployment or alert as an audit detail. This means tenants don't accidentally leave their spaces vulnerable.

- **Resource usage policies:**E.g., no Deployment should be allowed to use more than X CPU, so others don't get left with less. For example, may set arule so that just cluster admins can generate NodePort services or hostNetwork features.

Kyverno and Gatekeeper are somewhat overlapping but each has strengths:

- You can use Kyverno to both modify and develop the resources in your cluster. I have also discussed that it can create default resource rules automatically, for example a default NetworkPolicy in every namespace. You will find this helpful in automating the process for bringing people on board. Many Kubernetes users will find it convenient that Kyverno policies are specified in YAML. The technology can also confirm (verify) images (signatures) and enjoy various features useful for ensuring security.
- Strong logic can be written with Gatekeeper (OPA) because Rego gives full programming capabilities for policy rules. We saw Kubernetes policy arrive earlier with Open Policy Agent. A variety of curated policies (one example being the new Pod Security Policies replacement library) are available to use now. You cannot perform mutations in Gatekeeper—it's not designed for them—to push for best immutable practices. When results from security reviews are expressed as Kubernetes objects, we can then feed them directly into monitoring.

You usually wouldn't use both at the same time for the same policies, as it could be an overlap or similarity. It's possible for you to find one of your own. If an organization has expertise with Open Policy Agent (OPA) anywhere such as for microservices authorization or Terraform policy, OPA Gatekeeper may be their first choice. An altered view is that Kyverno, with its Kubernetes focus, makes rules easier to define for some users. Gatekeeper is a CNCF project being driven by adopters, while the CNCF is incubating Kyverno. Admission control uses both plugins, so having your API server support them should not be a challenge in managed Kubernetes environments.

Example Gatekeeper Policy: As you saw with Kyverno, in Gatekeeper, you have a ConstraintTemplate for privileged containers called K8sPSPPrivilegedContainer, making the Constraint something like [7]:

```yaml
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sPSPPrivilegedContainer  # This kind comes from the template
metadata:
  name: deny-privileged-containers
spec:
  enforcementAction: deny
  match:
    any:
    - kinds:
        - apiGroups: [""]
          kinds: ["Pod"]
      namespaces: ["tenant-a-dev", "tenant-a-prod", ...]  # or all, depending on scope
```

Code snippet: Gatekeeper Constraint to Deny Privileged Pods in Specific Namespaces

Any pod setting its securityContext.privileged= true through its container is denied by this Gatekeeper constraint because the template logic sets up the Deny condition. In practice, the OPA Rego behind the scenes fails the policy whenever a container uses the .securityContext.privileged setting.

In the beginning, it's possible to have Kyverno or Gatekeeper set up to only check (not enforce) policies. In your PCI and critical environments, you want to enforce all requirements (so attacks are caught and denied). Yet, many corporations go through an audit to test where weaknesses lie, then start with the enforcement.

Compliance teams find these types of policy engines very helpful when explaining their controls. Encryption use and changing default passwords are two of several PCI requirements that can be turned into security policies. Even if some of these tasks aren't covered by Kubernetes, you can ensure that all images are drawn from approved and scanned repositories through policy rules. It meets the PCI rule for secure system development by using checked, authorized images in the deployment. Another way: insist that containers are run with a read-only root file system (which adds security) – a policy can set this up across the whole cluster. Using these engines, you have just one set of cluster rules which is ideal for showing your audit.

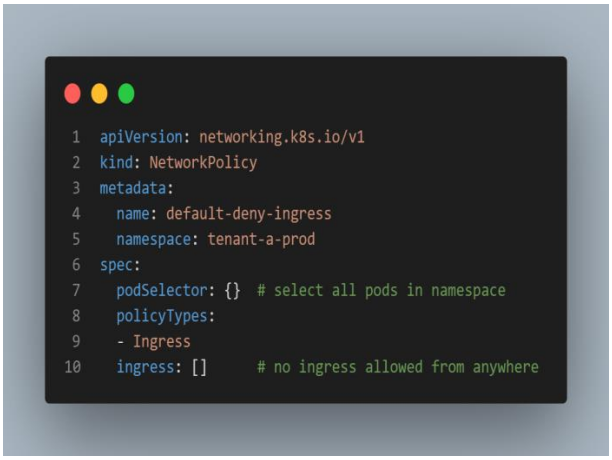
Network Isolation (Network Policies and Service Mesh)

We observed earlier that Kubernetes networks link any pod to any other by default, so they can interact, provided network rules are applied. A multi-tenant system cannot permit this – it would create an opportunity for tenants to cherry-pick their own sanctums. For this reason, Network Policies should always be used. NetworkPolicy works as a native tool that can manage pod traffic using rules by labeling, naming or choosing ports. They perform their tasks at layers 3 and 4 (and the CNI plugin handles this; best methods such as Calico and Cilium, accept NetworkPolicy semantics) [9].

Default Deny and Whitelisting:A common pattern is:

- In every namespace, configure so that all pods are set to "default deny" for ingress, meaning no ingress traffic from external namespaces is allowed. As a result, the namespace can only receive traffic if you manually grant permission (for example, communication with other pods within the same namespace is still allowed). An egress policy can also enforce all egress except to a specific group of external services or, if needed, block all egress and make all egress go through proxies.

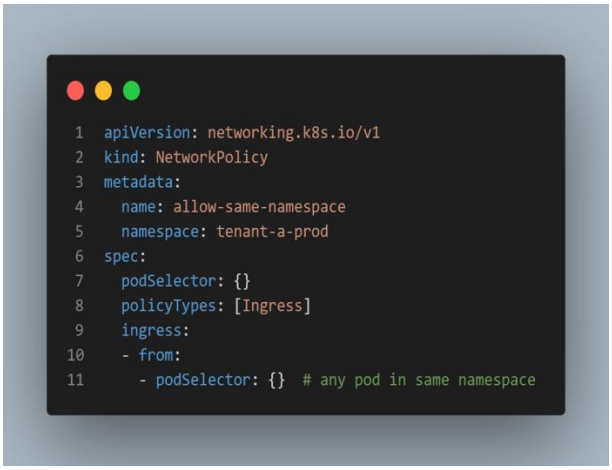
In that situation, a likely default deny ingress rule for a namespace could read:



```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
  namespace: tenant-a-prod
spec:
  podSelector: {}  # select all pods in namespace
  policyTypes:
  - Ingress
  ingress: []      # no ingress allowed from anywhere
```

Code snippet: Kubernetes NetworkPolicy for Default Deny Ingress Traffic

Applying the above means no pod outside tenant-a-prod (or even within it, in this case, since we gave no exceptions) can initiate connections to pods in tenant-a-prod. We could also use an allow policy inside the same namespace:



```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-same-namespace
  namespace: tenant-a-prod
spec:
  podSelector: {}
  policyTypes: [Ingress]
  ingress:
  - from:
    - podSelector: {}  # any pod in same namespace
```

Code snippet: Kubernetes NetworkPolicy to Allow Ingress from Same Namespace

As a result, tenant A's pods could talk to each other, though nothing from the other namespaces would get through. To make sure certain common or grouped services traffic flows, extra explicit policies are needed. To illustrate, perhaps a standard logging option is set up in logging namespace; tenant pods would be allowed to log there if an egress policy permitted access only if app=logging in namespace logging.

Namespace selectors: Another method NetworkPolicy can follow is to look at namespace labels to permit traffic from some other pods. It's possible to implement this by referring to each namespace as tenant=<name>, then setting no-almost-access policies for pods in namespaces. However, one nuance: if you want a rule like "Tenant A's namespace allows from itself but not from Tenant B's," you could label Tenant A's namespace with tenant=A and in Tenant A's policies allow from namespaceSelector: tenant=A. As a result, only the same tenant is allowed inside. With a specific label for each tenant, each isolated unit is set apart from all others. Another approach: use the built-in matchLabels on namespace. E.g.:

Code snippet: Kubernetes NetworkPolicy Ingress Rule with NamespaceSelector

However, that requires you to pass in the namespace name as the label (Kubernetes won't do it automatically). Better to set a label tenant: a and use that.

The fact is, NetworkPolicies are like all other namespaced objects in Kubernetes; they are unable to enforce rules everywhere in the cluster. That's why each namespace must have its own policies, even though selectors allow them to point to other namespaces. It's possible to automate this too: you can have Kyverno insert a default-deny policy each time a namespace is created.

NetworkPolicy is the best way to restrict traffic moving from one system to another. Should the attacker get out of their app container, if restrictions prevent them from talking to the pods in other namespaces or the database, the threat is well contained. Though they could attempt to escalate using those APIs or through the node, at least they can't use automatic scanning to find the cluster's open ports.

Service Mesh for Multi-Tenancy: Specifically, service meshes (such as Istio, Linkerd and Kuma) ensure that communication between services is restricted and encrypted. Because of the service mesh, tenants in a multi-tenant system can have more isolation.

- Even with incorrect network policies, a mesh can check that every pod presents a proper digital certificate prior to allowing traffic into the system. Istio is capable of requiring services to include mysterious digital signatures that identify them with the relevant tenant or workload. So the services owned by Tenant A could not properly trust an identity presented by Tenant B, even after it got through the network layer.
- Mesh also allows authorization policies at layer7: you can say "service X in tenant A can only be called by service Y in tenant A" using Istio's AuthorizationPolicy resource. This is like an application-level firewall. It complements NetworkPolicy (which stops unauthorized connections at L3/L4) by allowing you to also drop unauthorized requests at L7.
- Also, services inside a service mesh can be separated by having each tenant have its own dedicated virtual mesh or set of routing rules. In other words, you can use Istio to connect one control plane to more than one cluster or a single control plane to multiple meshes, allowing you to put up Istio ingress gateways for each of your tenants. As a result, every tenant can enter through their own space.

A diagram is provided below that outlines how to deploy a tenant-specific service mesh (Istio being an example):

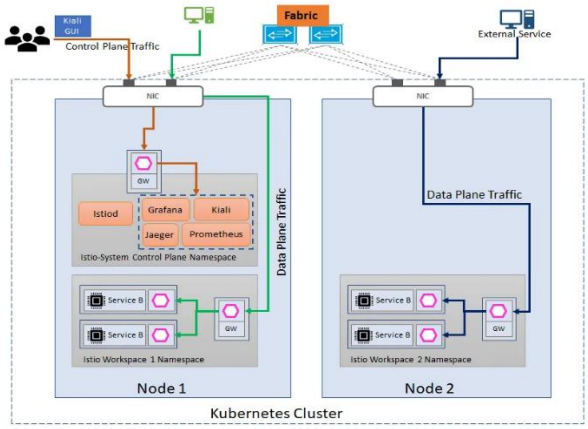


Diagram: Single Istio control plane with separate ingress gateways per tenant workspace (tenants on Node1 and Node2), ensuring isolated data plane traffic [14].

As illustrated in this figure (similar to an Istio multitenancy situation), we find that:

- A single Istio control plane (Istiod) is running in the infrastructure Istio-System namespace on Node1 and takes care of all config, but stays out of the data traffic.
- Each tenant gets two routing locations (called Namespace1 on Node1 and Namespace2 on Node2) with their own Istio Ingress Gateway (GW) inside each namespace. Such gateways serve only those applications running in that tenant's environment.
- With the green and blue arrows pointing in the same direction, we see data plane traffic. Each tenant's services communicate within the environment through the mesh and incoming traffic from outside comes through that tenant's own ingress gateway. All control plane data (orange line) to sidecars is handled by Istiod and is logically divided by the proper Istio configuration.
- If total isolation is important, running a separate Istio control plane per tenant is possible, although things become more complicated and at this time do require careful setup (but Red Hat changes that by letting users run multiple control planes that are limited to separate projects).

You can use multi-tenancy without a service mesh, but including it makes it much easier in places that rely on zero trust. Financial services that require secure encryption of internal traffic (since PCI includes encryption for sensitive data while on untrusted networks and inside a cluster could be trusted, applying mTLS using a mesh is considered a common practice for additional security) can rely on a mesh to apply mTLS everywhere. Observability by tenant is also improved, as metrics and traces can be filtered out separately.

Since it costs resources to use a mesh (sidecar proxies being one of them), some may prefer network policies plus adding auth natively to their applications. As service mesh evolves, it's now easier to use it with multiple tenants.

Pod Security: Pod Security Standards, Seccomp, SELinux/AppArmor

Taking these tenants apart makes it possible to shield the pods from damage when one is attacked. The PSS now replaces the PodSecurityPolicy and sets out baselines, baseline and restricted profiles for pods. For all workloads used by tenants, the Restricted profile is the best setting. Restricted PSS needs the use of [7]:

- No privileged containers (securityContext.privileged=false).

- Process identities, network interfaces and IPC identifiers come from the container; no links to host namespaces are used.
- HostPath volumes won't run (unless the team approves).
- Performing all tasks as a non-root user and using a root file system that can't be modified.
- App allows only specific Linux capabilities (forbids adding everything and adds only those you say), including ones you shouldn't have.

The PodSecurity Admission plugin which Kubernetes includes, verifies these. By labeling a namespace with pod-security.kubernetes.io/enforce: restricted, any pod that doesn't meet these criteria will be rejected. Getting a lot of security is quite simple with this technique. You can think of it as having pod security settings built into its policy engine. If you can't set up restricted security, ensure baseline is active and then place any exceptions and other policies just for the unique case.

Beyond PSS, additional controls:

- **Seccomp Profiles:**The Linux kernel includes Seccomp which blocks certain system calls that a process requests. Kubernetes lets you set a seccomp profile per pod or container (either the RuntimeDefault or a custom profile) [7]. The Restricted PSS expects seccomp to be enabled (RuntimeDefault or a specific profile). If the system is multitenant, it works well to use Docker/default seccomp (which restricts some risky syscalls). The CIS (Community Interest Security) benchmarks say that it should be used. Making tough seccomp profiles requires understanding your application's syscall usage, but it's rarely done and hard to support. To use an alternate way, PCI helps by promoting the use of gVisor or Kata Containers as they sandbox syscalls. GVisor handles system calls which greatly reduces the risk from untrusted code when run on your system. You can apply it by pod by setting the RuntimeClass for you application. Kata gluepods run on lightweight VMs which gives them excellent isolation but affects performance. Protected workloads are best served by choosing them in a PCI environment so they can act like separate VMs on a shared node.

- **Linux Capabilities:**As the default, containers get limited capabilities in Linux (DROP all) and have a few added ones back such as NET_BIND_SERVICE. Maintain that containers aren't started with the maximum authority. Pod Security Standards does not allow adding anything to a workload's capabilities not on the allowlist. You may tweak security by using custom policy if it's required (for example, no support for caps).

- **AppArmor/SELinux:**The modules manage processes by using different policies. AppArmor is easier to use on vanilla Kubernetes; one can load an AppArmor profile on nodes and then specify via an annotation that pods must use it (e.g., annotation container.apparmor.security.beta.kubernetes.io/<container_name>: localhost/<profile>). AppArmor can manage who or what, is able to access files or the network in the container. Multi-tenant clusters on Ubuntu or Debian bring more protection because AppArmor can place limits on a container so that hacking it can't cause extreme damage, for example preventing it from accessing some host files. More often, SELinux is found on platforms based on Red Hat and OpenShift also sets it up by default in containers. Because of SELinux, containers are unable to read or change each other's files without permission. If you have SELinux set up with container_runtime_t, you should definitely keep it on – it aids in keeping containers separate on the same node.

- **Pod Anti-Affinity:**A further option: if you don't want tenants to share a node, even in soft multi-tenancy mode, set up pod anti-affinity using the tenant field. One way to do it would be to add a

tenant label and then require tenant pods to repel pods from other tenants. Keeping it working like this is not easy, so it's usually best to use node pools or taints/tolerations to assign nodes to each group of tenants. Still, scheduling can raise the level of isolation, as long as you remember it's not perfect: if a pod is put in the wrong place at the wrong time, it could break the pattern; it's primarily there to say the pods and nodes you place together won't touch.

Example Pod Security Enforcement (Namespace Label method):

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: tenant-a-prod
  labels:
    pod-security.kubernetes.io/enforce: "restricted"
    pod-security.kubernetes.io/enforce-version: "latest"
```

Code snippet: Kubernetes Namespace with Restricted Pod Security Enforcement

As a result, if anybody tries to run a pod in tenant-a-prod that uses root permissions or a hostPath, the API server will reject it before any creation takes place [7].

Example Seccomp and User ID in Pod Spec:

```yaml
apiVersion: v1
kind: Pod
metadata: { name: secure-app, namespace: tenant-a-prod }
spec:
  securityContext:
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: app
    image: examplecorp/app:1.0
    securityContext:
      runAsUser: 1000
      allowPrivilegeEscalation: false
      capabilities:
        drop: ["ALL"]
        add: []  # no extra caps
      seccompProfile:
        type: RuntimeDefault
```

Code snippet: Kubernetes Pod Spec with Secure Defaults and RuntimeSeccomp Profile

The new kernel settings disable all caps locking, deny privilege escalation and related setuid tricks and rely on seccomp default mode. It doesn't handle AppArmor because we do that with annotations, yet if we had an AppArmor profile, we could apply it with annotations.

Kubelet and Node Security: That's not mentioned in detail, but it helps pod security – secure the kubelet's permissions (anonymous auth should be disabled and TLS auth should be enabled for the kubelet). When there are many clusters running together, an incorrectly configured node can give an attacker access to the kubelet API and the ability to run special or private containers. By default, kubelet runs on the cluster private network and requires authentication in cloud settings and you should check these. kube-bench can verify many of these settings and give you a report on your cluster's security (some evaluations in kube-bench line up with PCI and CIS rules that relate to system security).

You can use kubeaudit (made by Shopify) which scans your cluster for dangerous configurations, including running as root, specific permissions, lack of resource controls and so on – it inspects the manifests of your deployment. It helps to run these from time to time, to find the errors that broke through [12].

For financial reasons, proof of pod security will help comply with rules for protecting systems such as PCI's Req 2 on secure settings and Req 10 on monitoring – if a pod misbehaves and is stopped by a security mechanism, that could be counted as a security event. It is also important to tighten pod security to try to keep containers from creating unexpected network listeners and editing permitted files, as this matches the standard for only allowing intended programs to run and for data to flow at the network level (Requirements 1 and 2 from PCI address these issues at the main network).

Case Study Example: Secure Multi-Tenant Kubernetes in a FinTech Org

Let's strengthen these ideas by considering a real case study situation. FinBank is an example of a fintech company planning to use a Kubernetes platform shared by their teams and some partner applications. A few of them handle credit card payments subject to PCI-DSS, but others are just general business applications that do not fall under PCI-DSS requirements. FinBank seeks to help all teams use the same Kubernetes infrastructure, butguarantee that sensitive cardholder data is kept isolated according to the requirements by PCI. Here, I'll show how the methods we studied join together in this situation.

Context: FinBank decides to use a hybrid isolation model:

- **Two clusters:**One group is fully devoted to PCI jobs (CDE cluster) and the other cluster serves non-PCI jobs. Doing this allows us to identify the apps that are riskiest right away.
- Several teams use each cluster within the same network. Within the PCI cluster, each team operates payment API, fraud detection and similar services, but all these services are deemed part of PCI. Those teams are trusted more than others (since they're internal employees), but the least privilege rule is still important.
- On the non-PCI network, tenants are mainly company applications and a few analytics services from third parties (which are seen as less trusted code).

They implement the following:

Namespace Strategy: All teams and applications receive their own defined namespaces in their assigned cluster. The Payments group's production workloads are found in pay-prod on the PCI cluster (and

identified with the label tenant=pay). The Fraud service is starting in fraud-prod and similar environments. For those apps outside the PCI scope, the namespaces are named research-prod and marketing-prod.

Baseline Policy Deployment: Every namespace that gets deployed through FinBank's platform is configured to have policies inserted by Helm charts and Kyverno by the platform engineering team.

- A ResourceQuota is provided (maximum of 10 pods and 20 CPU allowed in dev, with more in prod).
- The default choice is to deny traffic both to and from different namespaces, by using a default-deny NetworkPolicy.
- Each namespace now has a label showing if it is PCI-sensitive (for example, com.finbank/pci= true/false). As a result, this label appears in network policy code to ensure that no traffic ever passes between a PCI namespace and any non-PCI namespace near the cluster's edge.

RBAC:

- Only members of the PCI project groups and a handful of platform admins can use the PCI cluster. Through RBAC, PCI-Developers in Active Directory is now mapped to have edit access to PCI namespaces within Kubernetes. This cluster grants non-PCI developers no rights of any kind.
- With PCI, service accounts are not permitted to do anything. As an example, the app's pods function as a service account, named pay-app-sa and are allowed only to read configmaps from within their namespace and list their own pods (for leading). It can't include private secrets (you have to explicitly add the needed secret and confirm the reference and permission).
- In the non-PCI scenario, Role-Based Access Control allows a wider team profile, with every team still given its own RoleBindings. FinBank connected their LDAP groups so that only people in the marketing group are allowed access to marketing-* areas.

Network Segmentation:

- **VPC-level:**Each PCI cluster node in the solution is isolated from the main corporate network by strict firewall security. Just a few predetermined passageways can take someone to it. Each namespace in a cluster is automatically given default isolation. In addition, they arrange for all namespaces in PCI clusters to prohibit talking to each other, except in rare cases. When namespaces needed to exchange data such as payment service in pay-prod talking to fraud service in fraud-prod, separate NetworkPolicies were added to approve the traffic only for specific ports.
- **Cross-Cluster isolation:**They make it an absolute rule that the PCI and non-PCI clusters are not directly connected on the pod network. Services in the PCI cluster cannot be reached directly by app partners in non-PCI cluster. Instead, the necessary information changes hands through an API gateway in front of the PCI cluster which cleans the data and monitors it for compliance.
- In the non-PCI case, network policy is relied on to assemble specific tenants. For example, the third-party analytics app is protected by placing it in a namespace that won't let it contact anything inside the network except a single message broker.

Service Mesh:

- FinBank deployed Istio in the PCI cluster to enforce mutual TLS between services. If someone was able to place a sniffer in a single pod, they'd still see just encrypted data exchanged between

microservices. Istio's AuthorizationPolicies made it possible for them to manage which services could call others (so only the web front-end pod connects to the payment-auth service) [5].

- Initially, they chose not to set up a mesh in the non-PCI cluster (to make things easier for applications that aren't crucial), but they did turn on NetworkPolicies and simple TLS for every traffic entry and exit [5].

Pod Security & Hardening:

- Both clusters have Pod Security Admission set to restricted on all namespaces (except one or two system namespaces like where they run Datadog agent which needed an elevated permission; those namespaces are tightly controlled and not multi-tenant).
- They use a custom seccomp profile (based on Docker's default but removing a few additional syscalls) which they apply to all pods via a MutatingAdmissionWebhook (using Kyverno's mutate feature). Essentially, any pod coming in without a seccomp profile gets one injected to ensure nobody runs unconfined. They also set readOnlyRootFilesystem: true on most deployments via a similar mechanism.
- All images designed for containers need to run as ordinary users. This is enforced by both Pod Security (runAsNonRoot) and a separate Gatekeeper policy that checks runAsUser is not 0. Due to heritage, the old app couldn't be easily switched to a new user and was put in its own namespace with an exception, along with instructions for how to deal with it (no secrets).

Isolation on Nodes:

- In the PCI cluster, they took advantage of the cluster's node pool concept: nodes are labeled with either role=pci-sensitive or role=common. All PCI workloads are tainted so that only those pods land on pci-sensitive labeled nodes. The system components or less-trusted monitoring agents run on the common nodes. This way, even if a third-party agent has a vulnerability, it's not on the same node as a card data processing pod. This also helps when applying host-level controls like file integrity monitoring or antivirus, focusing on those nodes.
- In the non-PCI cluster, each environment dev and prod, including apps created internally is allowed variation, even if it's between different nodes, but not always between tenants.

CI/CD Pipeline Integration:

- Deployments for FinBank are performed with Jenkins and Argo CD. For each application pipeline, only credentials are issued that allow deployment to the app's own namespace (by using a Kubernetes service account token with a RoleBinding that applies just there). The deep dive job for Payment Service, Jenkins, uses a restricted kubeconfig in the pay-dev and pay-prod namespaces. As a result, if someone compromises your CI or pipeline, they won't be able to use it to deploy an unsafe app in another developer's system.
- As part of GitOps, every manifest is reviewed and if a manifest contains a container as root or requests hostPath, the pipeline will fail the build. By doing this, we aim to catch problems in the beginning.

Monitoring and Auditing:

- All alerts are sent directly to the SIEM. Audit logs for Kubernetes are on – this is especially important on the PCI cluster – and any action you take such as an update or exec, on pods in PCI namespaces causes an alert to be sent (because this is something automation accounts normally do; if manually triggered by a developer, the alert helps review any special action).
- Prometheus is checking how much resources are used in each namespace and alerts when one tenant's consumption shoots up unexpectedly.
- On PCI nodes, Falco is set up as a runtime security tool to spot if a container attempts to break out (by opening a shell or contacting specific data). This could go outside of standard Kubernetes, but it's important for defense in depth.

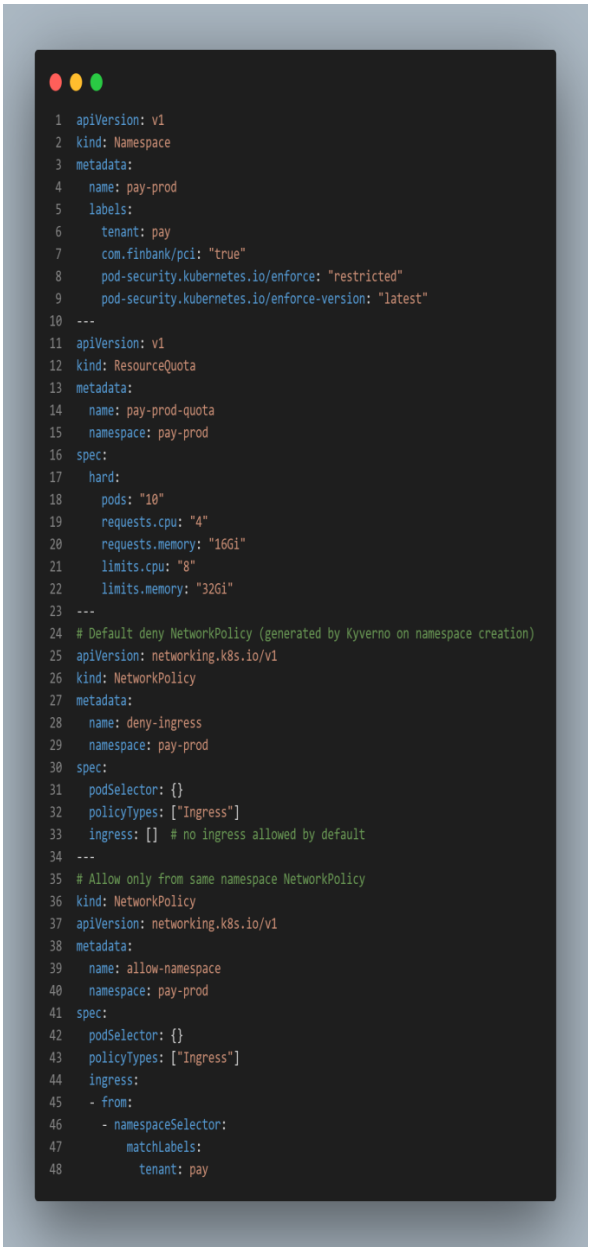
With this setup, FinBank's multi-tenant Kubernetes platform achieved the following security outcomes:

- **Strong network segmentation:**Network traffic can only move between tenants when allowed, thanks to privacy requirements between CDE and non-CDE (Req. 1).
- **Least privilege access:**At both the Kubernetes API level (RBAC) and the pod level (no additional permissions), the use of least privilege is supported (Req. 7 and 8).
- **Monitoring & logging:**Extensive tracking of access and what containers do which follows PCI Req. 10 to watch all use of network resources and cardholder data.
- **Vulnerability management:**They also added image scanners (following PCI Req. 6) to test and block any images with urgent security issues (Gatekeeper can act on this with a custom admission getting labels or using an external scanner's attestations).

Setting all these rules presented a problem for developers: they often missed their objectives if they included or used images that ran as root (making deployment fail). The answer was clear documentation and including the rules in standard Helm charts so teams wouldn't make changes easily. Over the years, teams discovered that security checks were mostly automatic in the platform, avoiding any chance of human error.

Summary of Configurations (YAML): Below are snippets of some key configurations from this case study:

Namespace Creation for PCI app (Payment Service):



```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: pay-prod
  labels:
    tenant: pay
    com.finbank/pci: "true"
    pod-security.kubernetes.io/enforce: "restricted"
    pod-security.kubernetes.io/enforce-version: "latest"
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pay-prod-quota
  namespace: pay-prod
spec:
  hard:
    pods: "10"
    requests.cpu: "4"
    requests.memory: "16Gi"
    limits.cpu: "8"
    limits.memory: "32Gi"
---
# Default deny NetworkPolicy (generated by Kyverno on namespace creation)
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-ingress
  namespace: pay-prod
spec:
  podSelector: {}
  policyTypes: ["Ingress"]
  ingress: []  # no ingress allowed by default
---
# Allow only from same namespace NetworkPolicy
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-namespace
  namespace: pay-prod
spec:
  podSelector: {}
  policyTypes: ["Ingress"]
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          tenant: pay
```

Code snippet: Kubernetes Namespace Configuration with Resource Quota, Pod Security, and NetworkPolicies

- In practice, pay-prod only admits traffic from namespaces labeled “pay” which is either itself or pay-staging, if they opened up traffic between environments. Often they move on to making that more precise or even allow for separate pay-staging to pay-prod steps as required.

RBAC for CI service account (deploying Payment app):

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: pay-prod
  name: deployer
rules:
- apiGroups: ["apps", ""]
  resources: ["deployments", "services", "pods"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
- apiGroups: [""]
  resources: ["pods/exec"]
  verbs: ["create"]
  # if CI needs to exec into a pod for a job (generally not, but example)
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: pay-prod
  name: jenkins-deployer-bind
subjects:
- kind: ServiceAccount
  name: jenkins-deployer
  namespace: cicd  # assume a cicd ns where Jenkins SA lives
roleRef:
  kind: Role
  name: deployer
  apiGroup: rbac.authorization.k8s.io
```

Code snippet: Kubernetes RBAC Role and RoleBinding for Jenkins CI Deployment in pay-prod Namespace

- **Kyverno Policy Example (from FinBank):**auto-generate NetworkPolicy on new Namespace (illustrative):

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: gen-netpol-default
spec:
  rules:
  - name: gen-default-netpol
    match:
      any:
      - resources:
          kinds: ["Namespace"]
    generate:
      kind: NetworkPolicy
      apiVersion: networking.k8s.io/v1
      metadata:
        name: default-deny
        namespace: "{{request.object.metadata.name}}"
      spec:
        podSelector: {}
        policyTypes: ["Ingress"]
        ingress: []
```

Code snippet: KyvernoClusterPolicy to Auto-Generate Default Deny NetworkPolicy per Namespace

With this Kyverno policy, a default deny network policy is automatically created in new namespaces. It's common to filter the namespaces to just some or even exclude kube-system.

OPA Gatekeeper Policy Example: Configure clusters so anyone in a certain group (this is advanced) can act as an exec in PCI namespaces, but others can't (for pedagogic reasons):

Using a ConstraintTemplate that audits kubectl exec requests (an AdmissionReview for subresource exec), you could deny those unless the user is in secops group. Although setup may be complex, this approach proves that it's possible to control API usage for each tenant in a multi-tenant cluster (since a developer having access to their namespace doesn't require us to allow them to get into running containers). I believe debugging needs it, but for PCI you may only add it on non-production or with monitoring).

The outcome for FinBank was a secure multi-tenant Kubernetes platform where teams could self-service deployments, but behind the scenes the platform enforced a compliance-aware guardrail framework. During a PCI audit, they were able to show:

- Network diagrams and policies proving the CDE (card data env) was isolated from other environments except through controlled points.
- Kubernetes config and Open Policy Agent rules showing that no unauthorized container config could run (no privilege escalation, etc.).

- Audit logs and monitoring demonstrating active oversight (they even simulated an attack in a pre-production environment – a penetration test – and showed that the attacker could not move from a compromised pod to any other tenant's pod or the control plane, thanks to these controls).

This case demonstrates how running multi-tenancy with high security on Kubernetes is possible.

Below is a summary table comparing various isolation techniques and security controls discussed, for easy reference.

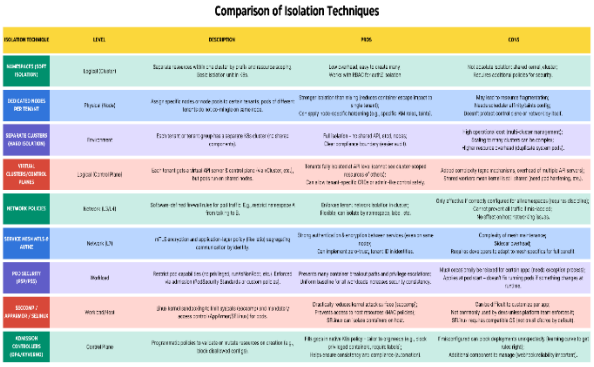


Table 2: Comparison of Isolation Techniques

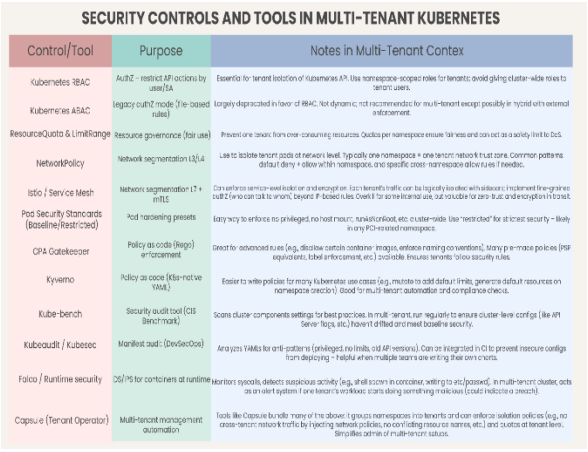


Table 3: Security Controls and Tools in Multi-Tenant Kubernetes

Future Trends in Multi-Tenant Kubernetes

Since cloud-native adoption is rising, the multi-tenancy area is also continuing to develop new ideas. Certain future trends and the latest best practices that apply are [1]:

- **First-class Multi-Tenancy Support in Kubernetes:** Multi-tenancy could be made a standard part of the Kubernetes project. Some talks have addressed secure container runtimes and newly introduced virtual clusters. Even though it's not part of the most recent release yet, wing SIG's projects (Hierarchical Namespaces and better Pod Security Admission) suggest Kubernetes may soon make managing tenants and setting policies in namespaces easier. In addition, the new ValidatingAdmissionPolicy (stable since 1.25) lets you write some admission policies using CEL directly. Doing that means simple policies won't need Gatekeeper or Kyverno, helping improve both the speed and dependability of enforcement [1].

- **Kubernetes as a Service:** A lot of organizations, in order to offer multi-tenancy, select Kubernetes and spin up new clusters whenever teams want them. Experts are starting to support the use of clusters that disappear as soon as a job is done. There's a chance that future improvements will make it look like you can manage 100 clusters as simply as you can manage a single one. As a result, using hard multi-tenancy at a big scale could be encouraged, as it further limits the need for advanced in-cluster shielding. They want to tie different clusters into a single management framework which in return allows you to manage policies similarly across all of them (resolving the problem of policy sprawl in multi-cluster) [1].
- **Container Security Sandboxes:** We currently have gVisor and Kata Containers available to us. In the future, we could rely on WebAssembly modules to enhance the sandboxing of certain code because WASM achieves very secure isolation. Hosting WebAssembly workloads within Kubernetes is getting popular for dealing with untrusted code (such as things customers upload for safe runtime) [1].
- **Better Tenant-aware Tooling:** Because Capsule was selected for the CNCF Sandbox, it's likely to mature and help make it possible for Kubernetes clusters to become multi-tenant platforms with just a few mouse clicks. Anticipate even better solutions that provide namespace control, automate networking and offer tenant-aware protection in only one operator or software. With isolation in mind, service meshes are available with multi-tenant support which encourages each customer to manage their own settings for traffic control (so each customer's traffic control filters are separated from the rest) [1].
- **Zero Trust and East-West Security:** As a result of this way of thinking, Kubernetes designs are now intended to treat everything inside the cluster (including other containers) as possibly compromised. It seems that future multi-tenant clusters will prefer strong encryption and authentication during all service communications, letting meshes or native mTLS do the job or they might make use of SPIFFE/SPIRE identities. In financial orgs, this is all the more important since, even within your team, always make sure every phone call is verified [1].
- **Compliance Automation:** There will be more examples of Kubernetes controls being lined up with compliance standards. As an example, operators who can inspect a cluster for PCI DSS 4.0 requirements and report their findings. As Security as Code and Policy as Code gain popularity, an artifact such as an OPA policy rule or Gatekeeper examination summary may become acceptable formal proof of compliance controls. Maybe through open source (like OpenSCAP for containers or Bank of America's KubeSafe), a hardened multi-tenant blueprint for financial organizations in Kubernetes can be created.
- **Multi-tenant Persistent Storage:** We haven't delved much into storage isolation in the section above. With multi-tenancy, several tenants may each claim storage volumes from the same storage source. It is important that each tenant cannot see the data of the other (look at machine learning recommendations based on this). More and more organizations could start using storage encryption that is separated by tenant using keys. Now, Kubernetes allows volumes from PVCs and cloning and in future, it may stop anyone (except those with permission) from cloning a volume across different namespaces.
- **Evolving Pod Security:** The removal of PodSecurityPolicy (PSP) in favor of Pod Security Admission is complete, but the ecosystem should provide other flexible choices. For example, Kubewarden (which uses WebAssembly policies) could get well known; it has features much like

Gatekeeper but is faster and easier to use. In the coming years, we could find Kubewarden or its peers used more regularly to control access in high-traffic, common areas, because it runs WASM ahead of time to be faster than OPA. This is especially significant in big clusters full of many tenants operating continuously, since small delays from admission control can be very important.

- **Regulatory Changes:** Version 4.0 of PCI-DSS released in 2022 says that controls should always be in use and checked regularly. Operators in the finance sector may have to make sure multi-tenant Kubernetes is monitored more regularly and use tools that quickly spot changes, like a NetworkPolicy being removed (to get it back in place again within seconds). Since immutable infrastructure works so well, immutable compliance could emerge: attach security config to GitOps and fix any deviance it finds. It agrees with the declarative design in Kubernetes.
- **Confidential Computing:** There are now Confidential VMs being offered by cloud providers which encrypt virtual machine memory. It could become possible in the future to place Kubernetes worker nodes on confidential VMs, so that the cloud admin never gets access to tenant memory. That may need to happen with some financial data. Working Kubernetes with Hardware Security Modules (HSMs) or secure enclaves for keys could result (for example) in encrypting each etcd node and creating unique sensors for each user's secrets. Because of these features, multi-tenant clusters benefit from enhanced security against problems such as a bad cloud provider inside or a broken host operating system [1].

In summary, the future is aimed at making it easier and safer to implement multi-tenancy – whether by making the shared model stronger (with more effective isolation and policy) or by letting operators manage many ISO clusters with automation and remote tools. Still, financial services businesses will watch closely, until these new advancements allow them to prove they are equally or more secure than working in separate silos. The challenge of making development easy yet keeping everything secure does not disappear, but choosing the right policies and automation in Kubernetes slowly inches the two towards each other where having many tenants is concerned.

Conclusion

Making a multi-tenant Kubernetes environment safe in finance or other top-priority industries means relying on smartly chosen architecture, proper policy and automated systems. Given that Kubernetes namespaces serve as tenancy boundaries, by also making sure these spaces are backed up by reliable RBAC, network segmentation and individual pod restrictions, organizations enjoy the gains of multi-tenancy without risking shared resources. We find that protection cannot rely on just one setup; we need NetworkPolicies, Pod Security Standards and admission policies which support each other and provide extra protection. A case study of a PCI-compliant deployment reveals that a shared Kubernetes platform is able to meet all the compliance rules, but only after careful configuration and regular use of extra tools such as policy engines and monitoring software.

Multi-tenant Kubernetes will keep changing as the practice advances. Those responsible for managing a cluster should follow new developments (such as escape exploits in containers or cross-namespace assaults) and the updates in Kubernetes that prevent them (such as updates adding security features to new releases). They ought to help tenants value security: transparent explanations for policies (like the reason not to run as root) support the company's efforts and make its implementation smoother.

Every control in Kubernetes should be described according to the common language found in the accepted security standards. As part of PCI-DSS requirements, network segmentation, least privilege, secure configurations, monitoring and incident processes are carried out using Kubernetes. Auditors may

not be proficient Kubernetes users which means it's up to the organization to explain how their cluster satisfies every requirement (since several Kubernetes capabilities can satisfy a single requirement you discussed). If cloud engineering teams start collaborating with compliance early, it's possible for the platform to automatically notice and stop non-compliant setups in development processes.

As we conclude, secure multi-tenancy on Kubernetes is possible and shouldn't be confused with an impossible concept. Because it works closely with the cloud-native view: control everything with code, apply rules the same for all and handle risk by adding layers. Security-focused organizations are recommended to embrace Kubernetes multi-tenancy, using the advice in this paper: divide aggressively, think all the time about being breached, set safeguards automatically and regularly inspect your defenses. The outcome is a cloud system that supports many applications and guarantees sensitive data and workloads are separated, follow regulations and are secure against risks related to being near other things. When we have increased tooling and expertise in the Kubernetes community, applying isolation should be relatively easy, so teams have more time to deploy and build innovations. While guidance is needed to secure the flexibility of Kubernetes, when complete, it becomes a natural choice for cloud strategies requiring the highest standards of security and compliance.

References

1. Kubernetes.io. (2022, December 10). Multi-Tenancy Best Practices – Kubernetes.io Documentation. Retrieved from https://web.archive.org/web/20221230232536/https://kubernetes.io/docs/concepts/security/multi-tenancy/
2. Microsoft. (2022). Kubernetes Multi-Tenancy Models (Soft vs. Hard Isolation) – Azure Architecture Center. Retrieved from https://web.archive.org/web/20220930202755/https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/service/aks
3. PCI Security Standards Council. (2018, April 17). Cloud Computing Guidelines v3.0. Retrieved from https://www.pcisecuritystandards.org/pdfs/PCI_SSC_Cloud_Guidelines_v3.pdf
4. Raesene. (2022, December 20). PCI Compliance for Kubernetes in Detail – Part 16: Segmentation. Retrieved from https://raesene.github.io/blog/2022/12/20/PCI-Kubernetes-Section16-Segmentation/
5. Batra, S. (2020, July 18). Service Mesh (Istio) Patterns for Multitenancy (Layer5 Blog). Retrieved from https://layer5.io/blog/service-mesh/service-mesh-istio-patterns-for-multitenancy
6. Clastix. (2022). Capsule – Kubernetes multi-tenancy operator. Retrieved from https://web.archive.org/web/20220921124922/https://capsule.clastix.io/
7. Open Policy Agent | Gatekeeper. (2023). PSP Privileged Container Policy – ConstraintTemplate example. Retrieved from https://github.com/open-policy-agent/gatekeeper-library
8. Kyverno. (2022). Disallow Privileged Containers Policy (v1.6). Retrieved from https://web.archive.org/web/20220727040122/https://kyverno.io/policies/
9. Kubernetes.io. (October 24, 2022). Network Policies – Concepts and Default Deny Best Practice. Retrieved from https://web.archive.org/web/20221205135735/https://kubernetes.io/docs/concepts/services-networking/network-policies/
10. CNCF TAG Security. (2020). Cloud Native Security Whitepaper. Retrieved from CNCF TAG Security GitHub retrieved from https://tag-security.cncf.io/community/resources/security-whitepaper/v2/CNCF_cloud-native-security-whitepaper-May2022-v2.pdf
11. Aqua Security. (2020). Kube-bench: CIS Benchmark Tool. Retrieved from https://github.com/aquasecurity/kube-bench

12. Shopify. (2021). kubeaudit – Tool to Detect Insecure Configurations. Retrieved from https://github.com/Shopify/kubeaudit
13. Kubernetes.io. (2022). Multi-Tenancy Checklist (v1.26). Retrieved from https://web.archive.org/web/20221230232536/https://kubernetes.io/docs/concepts/security/multi-tenancy/
14. Medium (Jun 26, 2020) Service Mesh (Istio) patterns for Multitenancy retrieved from https://medium.com/@sudeep.batra/service-mesh-istio-patterns-for-multitenancy-2462568636f7