International Journal of Emerging Research in Engineering and Technology
Pearl Blue Research Group | Volume 7 Issue 2 PP 87-99, 2026
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET-V7I2P113

Original Article

Kubernetes RBAC and Network Policy Enforcement in Multi-Tenant Enterprise Environments

Bharat Singh Chaudhary
Independent Researcher, Cloud Security & DevSecOps Architect, Quorum Information Technology, Calgary, Alberta, Canada.

Received On: 08/03/2026 ***Revised On: 07/04/2026*** ***Accepted On: 15/04/2026*** ***Published On: 21/04/2026***

Abstract - *Multi-tenant Kubernetes clusters present a unique set of security challenges that single-tenant deployments simply do not face. When multiple development teams, business units, or even external customers share the same underlying cluster infrastructure, the blast radius of any misconfiguration or privilege escalation grows dramatically. A compromised pod in one tenant's namespace can, without proper controls, enumerate services in every other namespace, access secrets belonging to other teams, and even escape to the host node if pod security settings are not enforced. These are not theoretical risks they are the exact attack patterns documented in the MITRE ATT&CK Container Matrix and exploited in real-world Kubernetes breaches. This paper examines the design and enforcement of Role-Based Access Control (RBAC) policies and Kubernetes Network Policies in multi-tenant environments, drawing on operational experience managing enterprise telecommunications infrastructure with over 200 namespaces across development, staging, and production tiers. We propose a layered access-control framework that combines namespace-scoped RBAC bindings, default-deny network policy segmentation, Pod Security Standards enforcement, and Kyverno admission controller automation to achieve tenant isolation without sacrificing developer productivity. The framework is validated against six common attack scenarios from the MITRE ATT&CK Container Matrix, including lateral pod-to-pod movement, privilege escalation through service account token abuse, privileged container escape, cross-namespace kubectl access, egress to external command-and-control servers, and unauthorized image deployment from untrusted registries. Results indicate that the proposed layered approach blocks 100 percent of tested cross-tenant access attempts while adding only 12 milliseconds of admission webhook latency per request. The paper also discusses the operational trade-offs between namespace-level soft multi-tenancy and hard multi-tenancy approaches using virtual cluster solutions.*

Keywords *- Kubernetes, RBAC, Network Policy, Multi-Tenancy, Container Security, Namespace Isolation, Admission Controller, Pod Security Standards, Kyverno, Calico, MITRE ATT&CK.*

1. Introduction

Kubernetes has become the default orchestration platform for deploying containerized workloads across industries ranging from financial services to telecommunications. The economic pressure to consolidate infrastructure drives many organizations toward multi-tenant clusters, where workloads from separate teams or customers run on shared node pools. This approach reduces infrastructure cost running five separate clusters for five teams costs significantly more than running one shared cluster but it introduces security boundaries that Kubernetes does not enforce by default.

I have spent the better part of three years managing Kubernetes clusters in telecommunications environments, and one pattern emerges repeatedly: organizations adopt Kubernetes for its orchestration capabilities but underestimate the security configuration work required to make shared clusters safe. Out of the box, a Kubernetes cluster permits unrestricted pod-to-pod communication across all namespaces, grants broad default service account permissions, and does not enforce pod security baselines. In a single-tenant development cluster, these defaults are inconvenient. In a multi-tenant production cluster, they are dangerous.

The consequences of this default-open posture are not hypothetical. In 2023, a widely reported Kubernetes breach at a European financial institution began with a compromised web application pod that was able to reach the Kubernetes API server using the default service account token mounted in every pod. From there, the attacker enumerated services across namespaces, discovered a database credential stored in a ConfigMap rather than a Secret, and exfiltrated customer records belonging to an entirely separate business unit sharing the same cluster. The root cause was not a sophisticated zero-day exploit it was the absence of basic RBAC restrictions and network segmentation that would have contained the breach to the originally compromised namespace.

Building on our earlier work on zero-trust security architecture for containerized microservices [1], this paper focuses specifically on the RBAC and network-layer controls needed to achieve meaningful tenant isolation in shared Kubernetes clusters. We present practical configurations tested in production environments and evaluate them against documented attack scenarios.

1.1. Research Questions

This paper addresses four specific research questions:

- How should RBAC policies be structured to prevent cross-tenant access in multi-namespace Kubernetes clusters?
- What NetworkPolicy configurations are required to enforce default-deny segmentation between tenant workloads?
- How can admission controllers automate the enforcement of security baselines across hundreds of namespaces?
- What is the performance overhead of layered security enforcement in production multi-tenant clusters?

1.2. Scope and Limitations

This paper addresses soft multi-tenancy the scenario where tenants are internal teams within the same organization who are not mutually hostile but should be isolated to limit blast radius. Hard multi-tenancy, where tenants are external customers who are mutually untrusted, requires additional controls (virtual clusters, dedicated node pools, separate API server endpoints) that are discussed briefly but are not the primary focus.

2. Background And Related Work2.1 Kubernetes RBAC Model

Kubernetes RBAC operates through four primary API objects: Role, ClusterRole, RoleBinding, and ClusterRoleBinding. Understanding the distinction between these objects and particularly the scoping implications of each is essential for multi-tenant security design.

A Role defines a set of permissions (verbs on resources) scoped to a single namespace. For example, a Role in namespace 'team-alpha' might grant get, list, and watch permissions on pods and services within that namespace only. A ClusterRole defines permissions that can apply cluster-wide or can be referenced by RoleBindings in individual namespaces. The critical design decision in multi-tenant environments is whether to use ClusterRoles with namespace-scoped RoleBindings sharing permission definitions while constraining scope or fully namespace-scoped Roles.

In practice, we strongly recommend the ClusterRole-plus-RoleBinding pattern. Defining common persona roles as ClusterRoles (developer, deployer, admin, viewer) and then binding them per namespace via RoleBindings provides consistency across hundreds of namespaces without requiring duplicate Role definitions in each one. The key safety property is that a RoleBinding can only grant permissions within its own namespace, regardless of whether the referenced ClusterRole contains broader permission definitions.

The most dangerous misconfigurations involve Cluster Role Bindings bindings that grant permissions across all namespaces simultaneously. A ClusterRoleBinding granting the 'edit' ClusterRole to a development team effectively gives them write access to every namespace in the cluster, including kube-system. In multi-tenant environments, ClusterRoleBindings should be restricted to platform administrators only and subject to rigorous review.

2.1. Service Account Token Risks

Every pod in Kubernetes is automatically assigned a service account, and by default, the default service account token is mounted at /var/run/secrets/kubernetes.io/serviceaccount/token in every pod. This token grants the pod an identity that can make API calls to the Kubernetes API server. The permissions associated with the default service account vary by cluster configuration, but in many default installations, the default service account has read access to secrets, services, and endpoints within its namespace.

This is a common attack vector. An attacker who gains code execution inside a pod through a vulnerable application, a supply chain compromise, or a container escape can read the service account token and use it to query the Kubernetes API. If the default service account has excessive permissions (or if the cluster does not enforce RBAC at all, which was the default before Kubernetes 1.6), the attacker inherits those permissions.

The mitigation is straightforward but frequently overlooked: set automountServiceAccountToken: false on all pods that do not need API access (which is the vast majority), and create dedicated service accounts with minimal permissions for pods that genuinely require API interaction.

2.2. Network Policy Fundamentals

Kubernetes NetworkPolicy resources control traffic flow between pods based on namespace labels, pod labels, IP blocks, and port numbers. The fundamental property that makes NetworkPolicy tricky in multi-tenant environments is that it is an opt-in mechanism in the absence of any NetworkPolicy, all ingress and egress traffic is permitted across all namespaces. This means that multi-tenant isolation requires explicit default-deny policies in every tenant namespace, with specific allow rules layered on top.

A subtlety that catches many administrators is that NetworkPolicy enforcement depends entirely on the Container Network Interface (CNI) plugin. Not all CNI plugins support NetworkPolicy. Flannel, one of the most commonly used CNI plugins, does not enforce NetworkPolicy at all the resources can be created without error, but they have no effect on traffic. This creates a dangerous false sense of security. Organizations must verify that their CNI plugin (Calico, Cilium, Weave Net) actually enforces the policies they define.

In our environments, we use Project Calico as the CNI plugin specifically because of its robust NetworkPolicy enforcement, including support for GlobalNetworkPolicy (a Calico extension that applies policies across all namespaces)

and host endpoint policies that protect the node network interfaces themselves.

2.3. Pod Security Standards

Kubernetes v1.25 replaced the deprecated PodSecurityPolicy (PSP) with Pod Security Standards (PSS) enforced via the built-in PodSecurity admission controller. PSS defines three profiles: Privileged (unrestricted), Baseline (prevents known privilege escalations like hostNetwork, hostPID, and privileged containers), and Restricted (fully hardened, requiring a non-root user, read-only root filesystem, and no capability escalation).

In multi-tenant clusters, tenant namespaces should enforce the Restricted profile, while system namespaces (kube-system, monitoring, cert-manager) may require Baseline due to their operational needs. The transition from PSP to PSS is well-documented, but many clusters still run without any pod security enforcement a gap that admission controllers like Kyverno can fill with more granular and customizable policies.

2.4. MITRE ATT&CK Container Matrix

The MITRE ATT&CK Container Matrix provides a structured taxonomy of adversary techniques targeting containerized environments. Understanding this matrix is essential for designing Kubernetes security controls that address real attack patterns rather than theoretical risks. The matrix covers the full attack lifecycle from initial access through execution, persistence, privilege escalation, defense evasion, credential access, discovery, lateral movement, and impact.

Several techniques in the matrix are directly relevant to multi-tenant isolation failures. T1610 (Deploy Container) describes attackers deploying new containers into a compromised namespace. T1613 (Container and Resource Discovery) covers enumeration of cluster-wide resources via the Kubernetes API. T1611 (Escape to Host) describes container escape techniques using privileged containers, hostPath mounts, or kernel exploits. T1552.007 (Container API Credentials) describes theft of service account tokens for API access. Each of these techniques has specific mitigations that map to the RBAC, NetworkPolicy, and admission controller controls presented in this paper.

Table 1: Kubernetes Threat Mapping with MITRE ATT&CK and Corresponding Mitigation Controls

MITRE Technique	ID	Description	Mitigation Control
Deploy Container	T1610	Adversary creates a new container in the cluster	Kyverno image allow-list + RBAC create restriction
Container & Resource Discovery	T1613	Enumeration of services, pods, secrets across namespaces	RBAC namespace scoping + disable SA token auto-mount
Escape to Host	T1611	Break out from container to underlying node	PSS Restricted profile + disallow privileged containers
Container API Credentials	T1552.007	Steal service account token from mounted filesystem	automountServiceAccountToken: false + bound tokens
Lateral Movement via Exec	T1609	kubectl exec into pods in other namespaces	RBAC: no exec verb in tenant roles
Network Service Discovery	T1046	Scan internal services across namespaces via DNS	Default-deny NetworkPolicy + DNS egress restriction

2.5. Prior Work and Industry Guidance

The CIS Kubernetes Benchmark provides over 200 security recommendations covering RBAC, network policies, pod security, API server configuration, and etcd encryption. The NSA/CISA Kubernetes Hardening Guide (2022) offers prescriptive guidance specifically for multi-tenant and classified environments. Our framework synthesizes recommendations from these sources along with the MITRE ATT&CK Container Matrix into a cohesive, implementable configuration.

Several academic studies have examined Kubernetes security. Shamim et al. (2020) analyzed misconfiguration patterns in public Kubernetes deployments and found that 73% had at least one critical RBAC misconfiguration. Lei et al. (2023) evaluated network policy effectiveness using formal verification methods and identified edge cases where overlapping policies could create unintended traffic paths. Rahman et al. (2021) studied IaC security defects in Kubernetes manifest files and classified them into seven categories. Our work extends these studies by providing an integrated, production-validated framework that addresses RBAC, network, and admission control simultaneously.

From an industry perspective, several Kubernetes security platforms have emerged including Fairwinds Polaris (open-source best-practice scanner), Kubescape (NSA/CISA benchmark implementation), and commercial platforms like Prisma Cloud, Aqua, and Sysdig. These tools validate configurations against predefined and custom policies but do not prescribe an integrated multi-tenant architecture. Our contribution complements these tools by providing the architectural framework that determines what policies should be enforced.

2.6. Multi-Tenancy Models in Kubernetes

Kubernetes multi-tenancy exists on a spectrum. At one end, namespace-as-a-service provides teams with isolated namespaces within a shared cluster. At the other end, cluster-per-tenant provides complete infrastructure isolation but at significantly higher cost. Between these extremes, virtual cluster solutions like vCluster create lightweight Kubernetes

clusters running inside namespaces of a host cluster, providing stronger isolation than namespaces alone without the full cost of separate clusters.

The choice of tenancy model depends on the trust relationship between tenants. When tenants are internal teams within the same organization (the most common scenario), namespace-level isolation with strong RBAC and NetworkPolicy is typically sufficient. The security controls protect against accidental cross-tenant access and limit the blast radius of a compromise, but they do not need to withstand a determined insider attack from a tenant with administrative intent to breach the isolation.

When tenants are external customers or mutually untrusted parties, namespace isolation is insufficient. Kernel-level attack surfaces (container escape exploits targeting the shared kernel), kubelet API exposure, and etcd data co-mingling create risks that namespace boundaries cannot address. These scenarios require virtual clusters, dedicated node pools with taints and tolerations, or fully separate clusters.

Table 2: Comparison of Kubernetes Multi-Tenancy Models: Isolation, Cost, and Complexity

Tenancy Model	Isolation Level	Cost	Complexity	Suitable For
Shared Namespace (no isolation)	None	Lowest	Lowest	Single-team development only
Namespace per Tenant (this paper)	Medium-High	Low	Medium	Internal teams, same organization
Virtual Cluster per Tenant	High	Medium	Medium-High	Semi-trusted tenants, ISV platforms
Cluster per Tenant	Highest	Highest	Highest	Mutually untrusted external customers

3. Proposed Multi-Tenant Security Framework

3.1. Namespace Design and Labeling Strategy

The namespace is the fundamental isolation boundary in Kubernetes. Each tenant receives a dedicated namespace with standardized labels that drive both RBAC bindings and NetworkPolicy selectors. Getting the labeling scheme right at the beginning is critical hanging it after hundreds of namespaces are in production is extremely painful.

We define three mandatory label categories for every tenant namespace:

- Ownership: tenant.company.com/owner — identifies the business unit or customer that owns the namespace (e.g., 'network-ops', 'billing-team', 'customer-acme')
- Environment: tenant.company.com/environment — distinguishes dev, staging, and production workloads; production namespaces receive stricter policies
- Sensitivity: tenant.company.com/sensitivity — classifies data-handling requirements: public, internal, confidential, or restricted; drives encryption and network policy stringency

These labels serve double duty. RBAC RoleBindings reference them to determine which users and groups have access to which namespaces. NetworkPolicies use them as selectors to permit or deny cross-namespace traffic. And Kyverno policies validate them to ensure every namespace meets the organization's labeling requirements before workloads can be deployed.

We also apply a set of resource quotas and limit ranges to every tenant namespace. Resource quotas prevent a single tenant from consuming all cluster resources (the 'noisy neighbor' problem), while limit ranges ensure every container specifies CPU and memory requests and limits. Without these, a memory-leaking application in one namespace can trigger OOM kills across the entire node, affecting workloads from all tenants sharing that node.

```
apiVersion: v1
kind: Namespace
metadata:
  name: team-billing-prod
  labels:
    tenant.company.com/owner: billing-team
    tenant.company.com/environment: production
    tenant.company.com/sensitivity: confidential
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/warn: restricted
```

3.2. RBAC Configuration Pattern

After experimenting with several RBAC models over the past three years including per-namespace custom Roles, group-based ClusterRoleBindings (a mistake we corrected quickly), and Kubernetes aggregated ClusterRoles we settled on a four-persona model that covers the access patterns we see across all tenant teams.

Table 3: Kubernetes Tenant RBAC Roles and Permission Model

ClusterRole Name	Target Persona	Key Permissions	Rationale
tenant-developer	Application developer	get/list/watch pods, logs, events; create/update deployments, services, configmaps; port-forward	Day-to-day development and debugging without ability to modify security-critical resources
tenant-deployer	CI/CD service account	create/update/delete deployments, rollouts, jobs; read secrets (scoped to specific names)	Automated pipeline deployment with minimal credential access
tenant-admin	Team lead / namespace owner	Full CRUD within namespace; manage RoleBindings within namespace; create/read	Self-service team management without cluster-level privileges

		secrets	
tenant-viewer	Auditor / read-only stakeholder	get/list/watch all resources within namespace; no create/update/delete	Compliance review and visibility without modification capability

Each ClusterRole is bound via a RoleBinding not a ClusterRoleBinding within the tenant's namespace. This is the critical security property. A RoleBinding in namespace 'team-billing-prod' that references the 'tenant-developer' ClusterRole grants those permissions only within 'team-billing-prod'. The same user has zero access to 'team-network-prod' unless a separate RoleBinding exists in that namespace.

We bind these roles to Azure Active Directory groups (via OIDC integration) rather than individual users. This means that when a new developer joins the billing team, they are added to the 'billing-developers' AD group, and they automatically inherit the tenant-developer role in the billing team's namespaces. No Kubernetes-side configuration change is needed.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: billing-developers
  namespace: team-billing-prod
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: tenant-developer
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: ad-group-billing-developers      # Azure AD group
```

3.2.1. Complete ClusterRole Definition

For clarity and reproducibility, below is the complete definition of our tenant-developer ClusterRole. Notice the specific exclusion of secrets from the core API group developers can view configmaps but not read secrets directly. They access secrets indirectly through pods that mount them as environment variables or volumes.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: tenant-developer
  labels:
    tenant.company.com/persona: developer
rules:
# Core workload access
- apiGroups: ['']
  resources: [pods, pods/log, pods/portforward, services, endpoints,
              configmaps, events, persistentvolumeclaims]
  verbs: [get, list, watch]
# Deployment management (no delete — handled by deployer persona)
- apiGroups: [apps]
  resources: [deployments, replicasets, statefulsets, daemonsets]
  verbs: [get, list, watch, create, update, patch]
# Job and CronJob access for batch workloads
- apiGroups: [batch]
  resources: [jobs, cronjobs]
  verbs: [get, list, watch, create, delete]
# HPA and scaling visibility
- apiGroups: [autoscaling]
  resources: [horizontalpodautoscalers]
  verbs: [get, list, watch]
# NetworkPolicy visibility (but not modification)
- apiGroups: [networking.k8s.io]
  resources: [networkpolicies, ingresses]
  verbs: [get, list, watch]
# EXPLICITLY EXCLUDED: secrets, serviceaccounts, roles, rolebindings
```

The explicit exclusion list at the bottom is a commentary reminder, not syntax Kubernetes RBAC is deny-by-default, so resources not listed in the rules are automatically denied. The comment is there for human reviewers who need to quickly verify what the role does not include.

3.2.2. Aggregated ClusterRoles for Extensibility

Kubernetes supports aggregated ClusterRoles using label selectors. This mechanism allows platform extensions (CRDs from operators, custom resource types from Helm chart installations) to automatically extend existing personas without modifying the persona ClusterRole definitions.

For example, when we install the Argo Rollouts operator, it creates CRDs for Rollout and AnalysisTemplate resources. Without aggregation, tenant-developer would not have permissions on these new resource types. With aggregation, the Argo Rollouts installation includes a ClusterRole fragment labeled with tenant.company.com/persona: developer that grants get/list/watch on Rollouts, and Kubernetes automatically merges it into the tenant-developer aggregated role. This keeps the persona model extensible without requiring manual updates for every new CRD.

3.3. Service Account Hardening

Beyond user-facing RBAC, service accounts represent the most commonly exploited access vector in Kubernetes breaches. Our framework enforces three service account controls:

First, we disable automatic service account token mounting on all pods by default. The Kyverno policy described in Section 3.5 automatically injects automountServiceAccountToken: false into every pod specification unless the pod is explicitly annotated to require API access. This single control eliminates the most common post-exploitation technique in container breaches.

Second, we create dedicated service accounts for every workload that genuinely needs Kubernetes API access (operators, controllers, CI/CD agents), each with a Role scoped to the minimum required permissions. A monitoring agent that needs to list pods gets get/list/watch on pods only not the broad permissions of the default service account.

Third, we enable Kubernetes bound service account token volumes (a feature stable since Kubernetes 1.22) which issues short-lived, audience-scoped tokens instead of the long-lived static tokens used by legacy service account configurations. These tokens expire after one hour and are automatically rotated, limiting the window of exploitation if a token is stolen.

3.4. Network Policy Architecture

Network Policy is where multi-tenant isolation becomes tangible at the network layer. Our architecture follows three principles: deny everything by default, allow only what is documented, and enforce at the CNI level using Calico.

3.4.1. Default Deny-All Policy

Every tenant namespace receives a deny-all policy applied automatically at namespace creation via a Kyverno generate rule. This policy blocks all ingress and egress traffic to and from every pod in the namespace. Nothing works until explicit allow rules are added.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
  namespace: team-billing-prod
spec:
  podSelector: {}              # matches ALL pods in namespace
  policyTypes:
  - Ingress
  - Egress
  # No ingress or egress rules = deny all
```

Some administrators worry that deny-all will break everything and prefer to start with allow-all and gradually restrict. In my experience, this approach never succeeds — the gradual restriction never happens because every restriction risks breaking something, and no one wants to be responsible for a production outage caused by a network policy change. Starting with deny-all and adding explicit allows is harder initially but results in a clean, documented, and auditable network posture.

3.4.2 Baseline Allow Rules

With deny-all in place, we layer on baseline allow rules that every namespace needs:

- DNS egress: Allow TCP/UDP port 53 to kube-dns (without DNS, nothing resolves this is always the first allow rule)
- Intra-namespace communication: Allow all traffic between pods within the same namespace (tenants trust their own pods)
- Ingress from load balancer: Allow traffic from the ingress controller namespace to pods serving external traffic
- Monitoring: Allow scrape traffic from the prometheus namespace to all pods exposing metrics endpoints

3.4.3. Cross-Namespace Rules

Cross-namespace traffic is the most sensitive category. When the billing service needs to call the authentication service in a different namespace, we create a targeted allow rule that permits traffic only between the specific pods involved, on the specific ports required, with namespace selector matching the target namespace labels.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-billing-to-auth
  namespace: team-auth-prod
spec:
  podSelector:
    matchLabels:
      app: auth-service
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          tenant.company.com/owner: billing-team
      podSelector:
        matchLabels:
          app: billing-api
    ports:
    - port: 8443
      protocol: TCP
```

This rule says: allow ingress to the auth-service pods, but only from pods labeled 'billing-api' in namespaces labeled 'billing-team', and only on port 8443. Any other pod, from any other namespace, on any other port, is still blocked. This is microsegmentation at the Kubernetes level.

3.4.4. Egress Control for Data Exfiltration Prevention

Ingress policies receive most of the attention in multi-tenant discussions, but egress control is equally important for preventing data exfiltration. A compromised pod that cannot communicate outbound to an attacker-controlled server cannot exfiltrate data, cannot download additional tools, and cannot establish a reverse shell. Our egress control strategy is particularly strict.

Beyond the baseline DNS allow rule, we permit egress only to explicitly documented endpoints. Production pods that need to reach external APIs (payment gateways, email services, cloud provider APIs) have specific egress rules allowing traffic to the documented IP ranges or CIDR blocks of those services. All other egress is blocked.

This is admittedly one of the most restrictive aspects of our framework, and it creates the most friction with

development teams. Developers are accustomed to pods being able to reach any internet endpoint to download packages, update dependencies, or call third-party APIs. In our model, each external dependency must be documented and an egress rule created. We maintain a shared Calico GlobalNetworkPolicy that permits egress to common cloud provider API endpoints (AWS API, Azure Management, internal artifact registries) so that individual teams do not need to duplicate these rules.

```
# Calico GlobalNetworkPolicy — common egress allowances
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: allow-common-egress
spec:
  selector: tenant.company.com/owner != ''
  types: [Egress]
  egress:
  # AWS S3, STS, KMS API endpoints (ca-central-1)
  - action: Allow
    protocol: TCP
    destination:
      nets: ['52.95.144.0/24', '52.95.145.0/24']
      ports: [443]
  # Internal artifact registry
  - action: Allow
    protocol: TCP
    destination:
      nets: ['10.100.0.0/16']
      ports: [443, 8443]
```

3.4.5. Network Policy Testing and Validation

A NetworkPolicy that is syntactically valid but semantically incorrect can create the illusion of security. We validate every NetworkPolicy through two mechanisms: automated policy testing using the Kubernetes NetworkPolicy simulator tool netpol (which evaluates whether a given labeled pod can reach another given labeled pod under the defined policies), and monthly penetration testing where we deploy a test pod in each namespace and attempt to reach pods in every other namespace.

The penetration test is simple but effective: deploy a curl-enabled pod in namespace-alpha and attempt HTTP connections to every known service endpoint in namespace-beta, namespace-gamma, and namespace-delta. Any successful connection indicates a policy gap. We automate this using a Kubernetes CronJob that runs monthly and reports results to the security team's Slack channel. In 12 months of operation, the test has uncovered two policy gaps both caused by newly created services that matched existing allow rules more broadly than intended.

3.5. Admission Controller Enforcement with Kyverno

RBAC and NetworkPolicy cover API-level access and network-level traffic. But there are security properties that neither can enforce container image sources, pod security context settings, resource limits, and label compliance. This is where admission controllers fill the gap.

We chose Kyverno over OPA/Gatekeeper for two reasons. First, Kyverno policies are written in YAML, which our operations teams already understand, rather than Rego, which requires learning a new policy language. Second, Kyverno supports generate rules that can automatically create resources (like default-deny NetworkPolicies) when a new namespace is created, reducing the manual onboarding steps for new tenants.

Our Kyverno policy set includes the following enforcement rules:

Table 4: Kubernetes Security Policies for Multi-Tenant Enforcement and Hardening

Policy Name	Action	Rationale
require-labels	Validate: reject namespaces missing mandatory labels	Ensures every namespace is properly classified for RBAC and NetworkPolicy targeting
require-resource-limits	Validate: reject pods without CPU/memory limits	Prevents noisy-neighbor resource exhaustion across tenants
disallow-privileged	Validate: reject privileged containers, hostNetwork, hostPID	Blocks the most common container escape techniques
restrict-image-registries	Validate: only allow images from internal registry	Prevents supply chain attacks via untrusted public images
generate-deny-all-netpol	Generate: create default-deny NetworkPolicy in new namespaces	Automates network isolation without human intervention
disable-automount-sa-token	Mutate: inject automountServiceAccountToken: false	Removes default API access from pods that do not need it
require-readonly-rootfs	Validate: reject pods without readOnlyRootFilesystem: true	Prevents attackers from writing tools or malware into the container filesystem

The generate-deny-all-netpol policy deserves special attention because it solves a real operational problem. Without it, every new namespace must be manually configured with a deny-all NetworkPolicy before workloads are deployed. In a fast-moving development environment where teams create namespaces frequently, this manual step is inevitably missed. The Kyverno generate rule makes it automatic the moment a namespace is created with tenant labels, the deny-all policy appears within seconds.

3.6. Resource Quota and LimitRange Enforcement

Multi-tenant isolation is not just about preventing unauthorized access it is also about preventing resource abuse. A single tenant running a memory-leaking application or a crypto mining container (either maliciously or through a compromised image) can destabilize an entire node, affecting workloads from all tenants sharing that node. Resource quotas and LimitRanges address this 'noisy neighbor' problem.

We configure ResourceQuota objects in every tenant namespace to cap the total CPU, memory, storage, and object count available to the tenant. The specific limits are negotiated with each team based on their workload requirements and adjusted quarterly based on actual usage data. Starting too restrictive creates friction; starting too permissive defeats the purpose.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: tenant-quota
  namespace: team-billing-prod
spec:
  hard:
    requests.cpu: '16'
    requests.memory: 32Gi
    limits.cpu: '32'
    limits.memory: 64Gi
    persistentvolumeclaims: '10'
    pods: '50'
    services: '20'
    secrets: '30'
    configmaps: '30'
```

LimitRange objects complement quotas by setting per-container defaults and maximums. Without a LimitRange, developers must specify resource requests and limits on every container and many do not. The LimitRange provides sensible defaults while preventing any single container from requesting excessive resources.

```
apiVersion: v1
kind: LimitRange
metadata:
  name: tenant-limits
  namespace: team-billing-prod
spec:
  limits:
  - type: Container
    default:              # applied if no limits specified
      cpu: '500m'
      memory: 512Mi
    defaultRequest:              # applied if no requests
specified
      cpu: '100m'
      memory: 128Mi
    max:
      cpu: '4'
      memory: 8Gi
    min:
      cpu: '50m'
      memory: 64Mi
```

3.7. Audit Logging and Monitoring

Security controls are only effective if you can verify they are working and detect when they are bypassed. Kubernetes audit logging records every API request to the API server, including the user identity, the resource accessed, the action performed, and the response code. In multi-tenant environments, audit logging is essential for forensic investigation, compliance reporting, and anomaly detection.

We configure audit logging with the following policy levels: RequestResponse for all write operations on secrets, RBAC bindings, and namespace modifications; Request for all other write operations; Metadata for read operations on sensitive resources; and None for health check endpoints and kube-probe user agents to reduce log volume.

Audit logs are forwarded to a centralized SIEM (Splunk in our environment) where we maintain detection rules for the following anomalies:

- Privilege Escalation: Any ClusterRoleBinding creation or modification this is the highest-privilege operation in Kubernetes and should be rare
- Unusual Secret Access: Secret access from pods that have not historically accessed secrets indicates possible lateral movement
- Token Abuse: API calls from service accounts that should have automount Service Account Token: false indicates a policy bypass
- Access Enumeration: Repeated 403 Forbidden responses from the same source indicates credential probing or RBAC enumeration
- Privileged Namespace Modifications: Pod creation in kube-system or other privileged namespaces from non-platform-admin identities

3.8. Tenant Onboarding Automation

In practice, the difference between a secure multi-tenant cluster and an insecure one is not the design of the security framework it is whether the framework is consistently applied to every namespace. When onboarding a new tenant requires 15 manual steps (create namespace, apply labels, create quotas, create LimitRange, create deny-all NetworkPolicy, create RBAC bindings, configure monitoring, etc.), some steps will inevitably be missed.

We automate tenant onboarding through a combination of Terraform (for namespace and RBAC provisioning) and Kyverno generate rules (for NetworkPolicy and limit defaults). The workflow is:

- Team lead submits a pull request adding their team to a YAML configuration file that lists all tenants, their members, quotas, and environment labels
- Terraform reads this configuration and creates the namespace, ResourceQuota, LimitRange, and RoleBindings

- Kyverno automatically generates the default-deny NetworkPolicy, pod disruption budgets, and monitoring ServiceMonitor when the namespace labels are detected
- ArgoCD synchronizes the namespace's application manifests from the team's GitOps repository

The total time from pull request approval to a fully provisioned, policy-enforced namespace is approximately 5 minutes, with zero manual steps after the pull request merge. This is critical for adoption if security is easy, teams will accept it; if security requires manual work, teams will find ways around it.

4. Validation and Results

4.1. Attack Scenario Testing Methodology

We validated the framework against six attack scenarios drawn from the MITRE ATT&CK Container Matrix. Each scenario was executed both with and without the security framework enabled, using a dedicated test cluster configured identically to the production environment. The test cluster contained four tenant namespaces (alpha, beta, gamma, delta) with representative workloads including web applications, API services, databases, and message queues.

4.2. Attack Scenario Results

Table 5: Kubernetes Multi-Tenant Security: Attack Scenarios and Mitigation Controls Comparison

#	Attack Scenario	Without Framework	With Framework	Control Layer
1	Cross-namespace pod access via service DNS discovery	Full access — attacker enumerates all services cluster-wide	Blocked — NetworkPolicy denies cross-namespace traffic	NetworkPolicy
2	Service account token theft followed by Kubernetes API enumeration	Token present in pod; API returns namespace resources	Token not mounted (Kyverno mutate); API call returns 403	Kyverno + RBAC
3	Privileged container escape to host node	Container runs as root with all capabilities; escape succeeds	Pod rejected at admission — privileged containers blocked	PSS + Kyverno
4	kubectl exec into another tenant's pod from compromised pod	Succeeds if default RBAC permits exec	RBAC denies — tenant-developer role does not include exec on foreign namespaces	RBAC
5	Egress from compromised pod to external command-and-control server	Unrestricted egress — HTTP/HTTPS connection to attacker server succeeds	Blocked — default deny-all egress; only DNS and approved endpoints permitted	NetworkPolicy
6	Pod deployment using untrusted Docker Hub image containing cryptominer	Image pulls and runs successfully	Kyverno rejects — image does not match approved registry pattern	Kyverno

All six attack scenarios were successfully blocked by the framework. It is worth noting that no single control layer was responsible for all blocks RBAC stopped scenario 4, NetworkPolicy stopped scenarios 1 and 5, Kyverno stopped scenarios 2, 3, and 6. This validates the layered defense approach: removing any one layer would leave specific attack vectors exposed.

4.3. Performance Impact Assessment

Security controls that cause noticeable performance degradation will face resistance from development teams. We measured the overhead of each control layer:

Table 6: Performance Impact of Kubernetes Security Controls: Latency and Operational Overhead Analysis

Control	Metric	Overhead	Assessment
Kyverno admission webhooks	Per-admission latency	+12ms average	Negligible — well below kubectl response time perception
Calico NetworkPolicy evaluation	Per-packet processing	+0.3ms worst case	Undetectable in application-level latency measurements
RBAC evaluation	Per-API request	+0.1ms	Kernel-level evaluation — effectively zero overhead
PSS admission	Per-pod creation	+2ms	Built into API server — minimal additional processing
Namespace onboarding (labels + quotas + policies)	One-time setup	+15 minutes per namespace	Automated via Kyverno generate and Terraform; manual review only

The total end-to-end overhead for a pod creation request traversing all control layers (RBAC check, PSS admission, Kyverno validation, Kyverno mutation) is approximately 15 milliseconds. For context, the baseline Kubernetes API server response time for a pod creation request without any custom admission webhooks is approximately 50-80

milliseconds. The 15ms addition is not noticeable in any operational context.

4.4. Real-World Case Study: Telecommunications Multi-Tenant Cluster

Our production environment consists of a multi-tenant Kubernetes cluster serving six internal teams across a telecommunications company. The cluster runs on Azure Kubernetes Service (AKS) with 48 nodes across three availability zones. The teams include network operations (managing VNF deployments), billing services (payment processing and invoicing), customer portal (web and mobile backend APIs), data analytics (ETL and ML pipelines), internal tools (CI/CD and developer productivity), and the platform engineering team (responsible for the cluster itself).

Before implementing the security framework, the cluster operated with minimal isolation: all teams had cluster-admin access (granted during initial setup when the cluster had only one team), no NetworkPolicies existed, and pods ran with default security contexts. The catalyst for change was an incident where a data analytics ETL job with a memory leak consumed all available memory on a node, causing OOM kills that affected billing service pods on the same node. The billing team lost 23 minutes of transaction processing, triggering an SLA violation investigation.

This incident was not a security breach, but it demonstrated the consequences of insufficient tenant isolation. If a memory leak could cross tenant boundaries, so could a compromised container. The platform engineering team was authorized to implement the security framework described in this paper over a six-month phased rollout.

4.4.1. Phase 1: RBAC Remediation (Month 1-2)

The first phase removed all ClusterRoleBinding grants for non-platform teams and replaced them with namespace-scoped RoleBindings using the four-persona model. This was the most politically difficult phase because teams lost privileges they had been using. The network operations team, for example, had been using cluster-admin to directly modify the kube-system namespace's CoreDNS configuration. We had to create a dedicated process for DNS customizations handled through the platform team's change management workflow.

During this phase, we also disabled automatic service account token mounting on all workloads and identified the 23 pods (out of 1,400+) that genuinely required Kubernetes API access. Each of these received a dedicated service account with the minimum required permissions.

4.4.2. Phase 2: NetworkPolicy Rollout (Month 3-4)

NetworkPolicy rollout was conducted in three stages: monitoring (using Calico's policy audit mode to log what would be blocked without actually blocking), selective enforcement (enabling enforcement on non-production namespaces first), and full enforcement (production namespaces).

The audit mode phase was invaluable. It revealed 340 unique network flows across namespace boundaries, many of which were undocumented. Developers had been creating cross-namespace service communications by directly referencing services as <service>.<namespace>.svc.cluster.local without documenting the dependency. The audit phase forced documentation of every cross-namespace communication, which improved our architecture understanding significantly beyond just the security benefits.

4.4.3. Phase 3: Admission Controller Deployment (Month 5-6)

Kyverno was deployed with policies in audit mode for two weeks before switching to enforce mode. During the audit period, 89 existing pods were flagged for policy violations 34 running as root without a security context, 28 missing resource limits, 17 using Docker Hub images instead of the internal registry, and 10 with privileged: true set. These were remediated by the owning teams before enforcement was activated.

The phased approach was essential for adoption. Each phase had clear communication to all teams, a preview period showing what would be affected, and an exception process for legitimate cases requiring temporary policy relaxation. The total implementation took six months, but the result was a cluster with comprehensive, automated, consistently enforced security controls across all 210 namespaces.

4.5. Operational Experience Over 12 Months

We have been running this framework in production for 12 months across 210 namespaces. During that period, the framework blocked 347 policy violations the majority being developers attempting to deploy containers without resource limits (213 incidents) or using images from Docker Hub instead of the internal registry (89 incidents). Seven incidents were blocked privileged container attempts that turned out to be developers copying Dockerfiles from online tutorials that used 'privileged: true' for debugging. Zero incidents of actual malicious activity were detected, but the controls ensure that if a breach occurs, the blast radius is contained to the compromised namespace.

The most significant operational benefit was unexpected: the security framework dramatically improved our incident response capability. Before the framework, investigating an anomalous network connection required tracing iptables rules and tcpdump captures across multiple nodes. After the framework, the NetworkPolicy definitions serve as a documented map of every permitted communication path. When a network issue occurs, we compare the actual traffic against the allow rules, and the problem is immediately apparent.

4.6. Comparison with Alternative Approaches

Table 7: Kubernetes Multi-Tenancy Isolation Approaches: Trade-offs in Security, Complexity, and Cost

Approach	Isolation Level	Operational Complexity	Cost	Our Assessment
This framework (RBAC + NetPol + Kyverno)	Strong namespace isolation with automated enforcement	Medium initial setup 6 months, ongoing maintenance 2-4 hrs/week	Low uses Kubernetes-native features + open-source Kyverno	Recommended for internal multi-tenancy
Istio Service Mesh + AuthorizationPolicy	L7-level mTLS + authorization per service	High sidecar management, certificate rotation, debugging complexity	Medium sidecar resource overhead (~100MB per pod), Istio control plane	Recommended when L7 observability is also required
vCluster (virtual clusters)	Full API server isolation per tenant	Medium-High virtual cluster lifecycle management	Medium lightweight but still multiplied API server instances	Recommended for semi-trusted or external tenants
Cluster per tenant (dedicated clusters)	Complete infrastructure isolation	High fleet management, cost allocation, networking	High full cluster overhead per tenant	Required only for mutually hostile tenants

5. Discussion

5.1. Layered Defense Effectiveness

The results confirm that Kubernetes multi-tenancy can be achieved at the namespace level when RBAC, NetworkPolicy, and admission controllers are deployed together as a layered defense. No single control is sufficient on its own. RBAC prevents unauthorized API access but does not restrict network traffic. NetworkPolicy restricts traffic but does not prevent a compromised service account from making API calls. Admission controllers enforce pod-level security but rely on RBAC to prevent direct kubectl manipulation. This interdependency is the core argument for a layered approach rather than relying on any single mechanism.

5.2. Soft vs. Hard Multi-Tenancy

Organizations considering hard multi-tenancy where tenants are mutually untrusted external customers should evaluate virtual cluster solutions (vCluster) or dedicated clusters per tenant. The namespace-level isolation presented here is appropriate for soft multi-tenancy where tenants are internal teams within the same organization. The practical difference is significant: in soft multi-tenancy, a determined insider with cluster-admin access could bypass all controls. In hard multi-tenancy, the isolation must withstand adversarial attack from tenants who may actively attempt to escape their boundaries.

5.3 Operational Challenges

The biggest challenge in deploying this framework was not technical but cultural. Development teams accustomed to cluster-admin access pushed back against the restrictions. The most common complaints were: inability to pull images from Docker Hub (requiring an internal registry workflow), inability to use privileged containers for debugging, and the additional time required to define NetworkPolicy allow rules for new service dependencies. We addressed these concerns by providing self-service namespace provisioning via Terraform modules, maintaining a curated internal image registry with popular base images, and documenting common NetworkPolicy patterns that teams could copy and adapt.

5.4. Comparison with Service Mesh Approaches

Service mesh platforms (Istio, Linkerd) provide an alternative approach to multi-tenant security through mutual TLS, L7 authorization policies, and traffic management. In our evaluation, service mesh provides stronger application-layer controls (header-based routing, JWT validation at the sidecar level) but adds significant operational complexity (sidecar injection, certificate management, control plane overhead). For organizations primarily concerned with network-layer isolation between namespaces, Kubernetes-native NetworkPolicy with Calico provides equivalent network segmentation at lower complexity. Service mesh becomes valuable when L7 traffic control and observability are also required.

We considered deploying Istio for our multi-tenant cluster and ultimately decided against it for this use case. The sidecar proxy adds approximately 50-100MB of memory overhead per pod across 1,400 pods, that is 70-140GB of additional memory consumed by sidecars alone. The Istio control plane (istiod, ingress gateway, egress gateway) requires its own high-availability deployment with resource reservations. And the debugging complexity when traffic does not flow as expected increases dramatically instead of checking one NetworkPolicy, you must check the Istio VirtualService, DestinationRule, AuthorizationPolicy, PeerAuthentication, and potentially the EnvoyFilter.

For organizations that already have Istio deployed for other reasons (canary deployments, traffic mirroring, circuit breaking), leveraging Istio's AuthorizationPolicy for tenant isolation is sensible. But deploying Istio solely for multi-tenant isolation is over-engineering the problem when Kubernetes-native NetworkPolicy with Calico achieves the same network-layer result with far less complexity.

5.5. Common Pitfalls and Troubleshooting

Based on three years of supporting teams operating within this framework, certain troubleshooting scenarios occur repeatedly. Documenting these helps operations teams resolve issues faster and reduces the support burden on the platform team.

Pitfall 1: DNS resolution failures after NetworkPolicy deployment. This is the single most common issue. When default-deny is applied without a corresponding DNS egress allow rule, every pod in the namespace loses DNS resolution. Applications fail with 'could not resolve host' errors. The fix is ensuring the kube-dns egress rule is always part of the baseline allow set, which our Kyverno generate rule handles automatically but legacy namespaces onboarded before Kyverno deployment may need manual remediation.

Pitfall 2: Webhook timeout causing pod creation failures. If Kyverno becomes unavailable (due to a node failure, upgrade, or resource exhaustion), the validating webhook blocks all pod creation requests with timeout errors. We configure the Kyverno webhook with failurePolicy: Ignore in non-critical namespaces (allowing pods to be created without validation if Kyverno is down) and failurePolicy: Fail only in highly sensitive namespaces where skipping validation is unacceptable.

Pitfall 3: RBAC 'forbidden' errors with unclear root cause. When a developer reports that they cannot perform an action and receive a 403 Forbidden error, the Kubernetes error message often does not indicate which specific RBAC binding is responsible. We use the kubectl auth can-i command with the as flag to diagnose RBAC issues. For example: kubectl auth can-i create deployments as=user@company.com -n team-billing-prod returns a clear yes/no answer and identifies the applicable ClusterRole or Role.

Pitfall 4: NetworkPolicy not taking effect despite being applied. This almost always indicates a CNI plugin that does not support NetworkPolicy. We encountered this once when a team provisioned a new cluster using Azure CNI (which supports NetworkPolicy) but with the kubenet network plugin selected instead of Azure CNI. The NetworkPolicy resources were created without error but had no effect on traffic. Always verify the CNI plugin supports NetworkPolicy before trusting the policies.

5.6. Lessons Learned from Production Deployment

Twelve months of operating this framework in production have yielded several lessons that are not obvious from the documentation or academic literature alone.

First, policy exceptions are inevitable and must be managed systematically. Despite our best efforts to create comprehensive policies, legitimate use cases arise that require temporary or permanent exceptions. A monitoring agent needs to run as root to access host metrics. A legacy application requires a deprecated API version. An emergency hotfix needs to bypass the image registry restriction. Without a formal exception process, teams bypass policies informally (often by asking the platform team to 'just make it work'), creating undocumented security gaps. We implemented an exception annotation system where pods can carry annotations that document the approved exception, the approver, and the expiration date. Kyverno validates these annotations and permits policy relaxation only for annotated pods with valid expiry dates.

Second, developer education is as important as technical enforcement. When developers understand why a policy exists not just that it blocks their deployment, they are more likely to design compliant workloads from the start rather than fighting the policies at deployment time. We run a monthly 'security office hours' session where developers can ask questions about policies, request exceptions, and learn about new security features. Attendance has grown from 5 people in the first session to 30+ as teams realize the platform team is there to help, not just to block.

Third, monitoring policy enforcement trends over time reveals organizational health indicators. A sudden spike in Kyverno violations from a specific team usually indicates a new project or team member who has not been onboarded to the security framework. A steady decrease in violations over time indicates maturing DevSecOps practices. We publish a monthly 'security posture dashboard' showing violations by team, trend lines, and remediation times, creating healthy competition between teams.

5.7. Future Work

Several directions warrant further investigation. First, Cilium's eBPF-based NetworkPolicy enforcement promises higher performance than iptables-based solutions (Calico) and supports L7 policies natively without a full service mesh. We are evaluating a migration from Calico to Cilium in our next cluster upgrade cycle.

Second, the Kubernetes Gateway API (graduating in 1.31) may provide standardized cross-namespace ingress routing that simplifies the current NetworkPolicy + Ingress controller configuration. Third, runtime security tools (Falco, Tetragon) can detect attacks that RBAC and NetworkPolicy cannot prevent, such as container escape exploits targeting kernel vulnerabilities. We plan to integrate Tetragon's eBPF-based runtime monitoring with our existing SIEM alerting pipeline.

Fourth, the emerging field of Kubernetes supply chain security (SLSA, Sigstore, cosign) addresses an attack surface that our current framework does not cover verifying that container images were built from trusted source code through a trusted pipeline. Integrating signature verification into the Kyverno admission pipeline would close this gap.

6. Conclusion

This paper presented a practical framework for Kubernetes multi-tenant isolation combining Role-Based Access Control, Network Policies, Pod Security Standards, and Kyverno admission controller enforcement. The

framework is based on three years of operational experience managing multi-tenant clusters in telecommunications environments and is validated against six attack scenarios from the MITRE ATT&CK Container Matrix.

The key findings are: (1) no single control layer provides sufficient isolation RBAC, NetworkPolicy, and admission controllers must be deployed together as a layered defense; (2) default-deny NetworkPolicy is essential and must be automated via admission controllers to prevent gaps in newly created namespaces; (3) service account token management is the most commonly overlooked attack vector and the easiest to mitigate; and (4) the performance overhead of the full security stack is approximately 15 milliseconds per admission request, which is negligible in any production context.

For organizations operating multi-tenant Kubernetes clusters, the framework presented here provides a replicable, tested starting point that balances security enforcement with developer productivity. The approach integrates naturally with the zero-trust container security architecture described in our earlier work [1] and scales to hundreds of namespaces without proportional increase in administrative overhead.

References

[1] Chaudhary, B. S. (2026). Zero-Trust Security Architecture for Containerized Microservices in Enterprise Telecommunications Networks. Zenodo. https://doi.org/10.13140/RG.2.2.18747.27686

[2] Kubernetes Documentation. "Using RBAC Authorization." kubernetes.io/docs/reference/access-authn-authz/rbac/, 2025.

[3] Kubernetes Documentation. "Network Policies." kubernetes.io/docs/concepts/services-networking/network-policies/, 2025.

[4] MITRE. "ATT&CK for Containers Matrix." attack.mitre.org/matrices/enterprise/containers/, 2024.

[5] Kyverno Project. "Kyverno: Kubernetes Native Policy Management." kyverno.io, 2025.

[6] Kubernetes Documentation. "Pod Security Standards." kubernetes.io/docs/concepts/security/pod-security-standards/, 2025.

[7] Project Calico. "Calico Network Policy and Security." docs.tigera.io/calico/, 2025.

[8] Center for Internet Security. "CIS Kubernetes Benchmark v1.8." cisecurity.org, 2024.

[9] NSA/CISA. "Kubernetes Hardening Guide v1.2." media.defense.gov, 2022.

[10] Chaudhary, B. S. (2026). Proactive Infrastructure Monitoring and Observability. ISCSITR - IJSRIT, 7(1), 1-33. https://doi.org/10.63397/ISCSITR-IJSRIT_2026_07_01_001

[11] Chaudhary, B. S. (2026). Designing Automated Disaster Recovery Strategies for Hybrid Cloud Environments. Zenodo. https://doi.org/10.13140/RG.2.2.12036.39048

[12] Chaudhary, B. S. (2025). Insights into Cloud Migration (Migration to Azure/AWS). IJCET, 16(1). https://doi.org/10.34218/IJCET_16_01_101