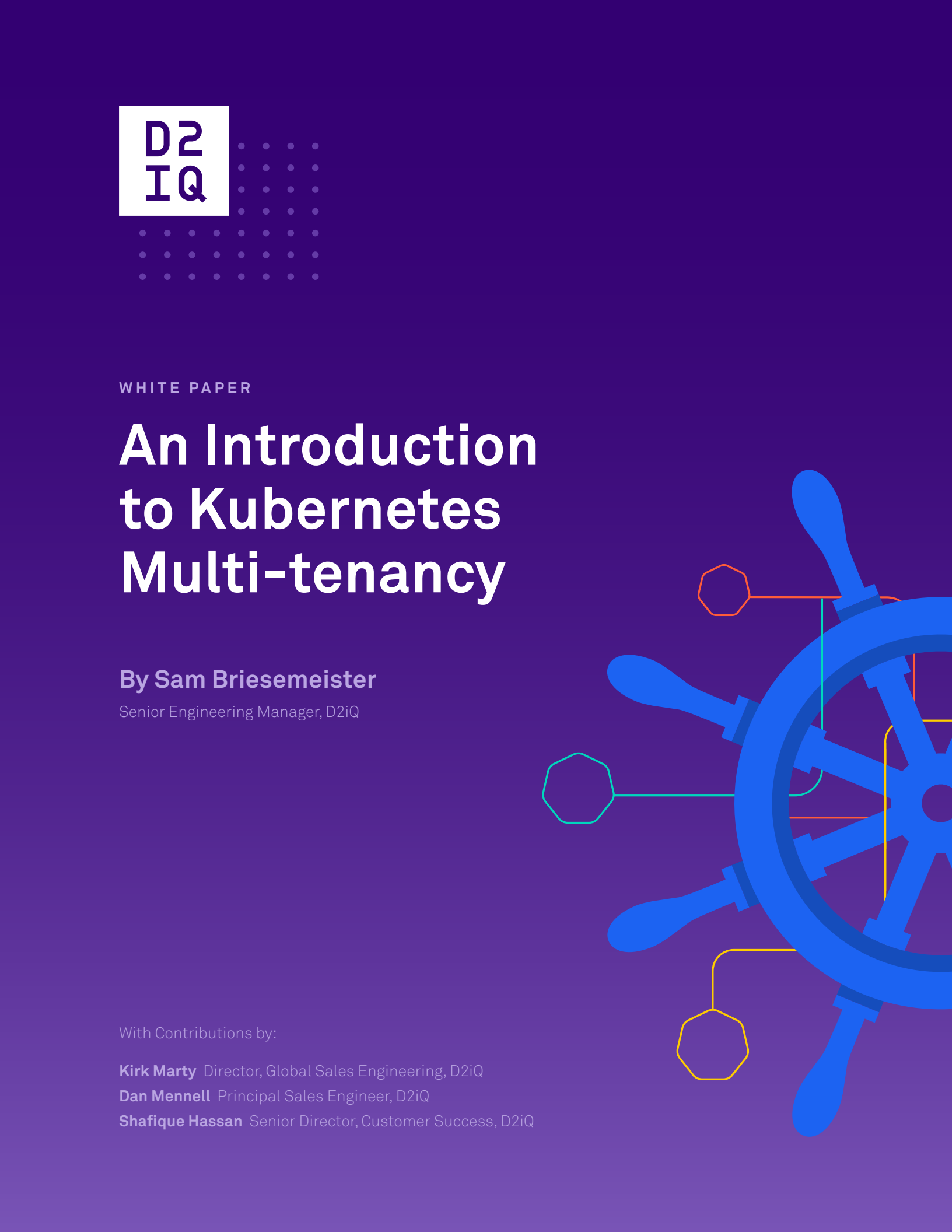

WHITE PAPER

An Introduction to Kubernetes Multi-tenancy

By Sam Briesemeister

Senior Engineering Manager, D2iQ

With Contributions by:

Kirk Marty Director, Global Sales Engineering, D2iQ
Dan Mennell Principal Sales Engineer, D2iQ
Shafique Hassan Senior Director, Customer Success, D2iQ

Executive Summary

In a multi-tenant Kubernetes architecture, multiple applications, services, workloads, users, or teams share a cluster's resources. This paper will examine the use of Kubernetes multi-tenancy, including:

- Why and when your organization should consider using a multi-tenant Kubernetes architecture
- Issues and challenges that multi-tenancy can raise, including "noisy neighbors" and "nosy neighbors"
- Multi-tenant resource management and the use of quotas
- Soft and hard multi-tenancy: what they mean and how and when to use them
- Additional considerations for service providers
- When and how to use a multi-cluster strategy

This paper will show that a well-developed multi-tenant strategy can drive more efficient use of infrastructure and personnel, improving ROI while still providing users with the levels of application performance, reliability, and security they require. It will also provide guidelines and best practices for implementing a successful multi-tenant strategy, maximizing efficiency and effectiveness while minimizing risk.

Why Multi-tenancy?

Economics and efficiency of scale influence technology decisions at enterprises. **Multi-tenancy capabilities in platforms aim to drive efficient use of infrastructure while providing operators with robust isolation mechanisms between users, workloads or teams**. Kubernetes allows for operators to build multi-tenant platforms leveraging a wide range of built-in capabilities that address isolation and efficiency design goals. Native Kubernetes capabilities can thus be used to achieve the end goals of increased efficiency and reduced risk in a multi-tenant platform. Below, we will examine the benefits of multi-tenancy, as well as some of the issues multi-tenancy can create for operators or users and how Kubernetes can be used to address them.

Costs associated with a platform can be categorized into: 1) infrastructure spend through private data centers or a cloud provider and 2) on-going operational costs for management and maintenance. With increased Kubernetes adoption, enterprises often experience "cluster sprawl"—a rapidly growing number of usually single tenant clusters lacking centralized governance. Cluster sprawl exacerbates management complexity, increases operational overhead and introduces additional costs not accounted for in the initial plan. A careful application of a multi-tenant strategy is a major remedy for the problems created by cluster sprawl.

Multi-tenancy in Kubernetes improves ROI by simultaneously increasing the cumulative value and reducing operating costs. Running more applications on the same shared infrastructure means better utilization of resources and a reduction in overall operating costs. Larger, shared clusters reduce infrastructure overhead by consolidating the control plane and often reduce network costs. Management is simplified by applying a consistent configuration strategy and applications can be made more resilient to infrastructure failures, resulting in fewer application outages over time. In other words, a correct multi-tenant strategy enables you to extract more use from existing IT assets while simultaneously reducing operating costs. This raises return while reducing investment to improve ROI from both sides of the equation.

Kubernetes achieves multi-tenancy by facilitating software-layer isolation on shared infrastructure. This comes with some trade-offs, but Kubernetes' capabilities meet many organizational needs. Organizing teams and adopting a culture that aligns with business goals is key to driving success without incurring runaway costs. Common organizational models include:

- Autonomous teams operate dedicated clusters and ship microservices on independent release cycles. When a team delivers multiple microservices on the same cluster, they should operate with application isolation.

- Enterprise operations teams deliver clusters to many teams in a platform as a service model to optimize operational efficiency. An individual cluster may be shared across many teams with similar security requirements.

- Globally distributed applications run on multiple clusters in different regions and multiple teams develop the service deployed in each region.

- Service providers deliver application services (e.g. SaaS) to multiple, unrelated customers, operating in the same cluster.

The next sections of this paper will describe the core concepts, configurations, contingencies, and considerations needed to support multi-tenant clusters and to meet these varying isolation requirements.

Key Multi-tenancy Issues and the Core Kubernetes Capabilities That Address Them

Two inherent problems that multi-tenancy constructs must seek to mitigate are the "noisy neighbor" and "nosy neighbor" problems. The "noisy neighbor" challenge addresses the problem of a tenant negatively impacting (or monopolizing) the capacity or performance of other tenants or the platform through hungry or greedy workloads. On the other hand, the "nosy neighbor" predicament refers to risks associated with access control and privacy of workloads in shared environments. "Noisy neighbors'' have a severe impact on the performance of other workloads on the same cluster, sometimes preventing them from running altogether. "Nosy neighbors" represent a security risk, either through inadverdent sharing of data or resources or through malign action.

The following list describes multi-tenancy capabilities available in Kubernetes to address these and other issues, along with some common deployment patterns leveraging these capabilities.

- **Namespaces:**
  The central element of isolation in Kubernetes is a *Namespace*. Typically, every tenant on your platform needs a single, dedicated namespace. The concept of the namespace is foundational to addressing either noisy neighbor or nosy neighbor issues. Tenants are usually applications or microservices—this granular approach offers the most powerful security and resource management capabilities—but teams can be treated as tenants in some situations (as in a SaaS model). Applications deployed in a namespace can leverage the powerful security and resource management constructs that Kubernetes offers to build a multi-tenant platform.

- **Roles and RoleBindings:**
  Within a namespace, correct configuration of Role-based Access Control (RBAC) is critical for security isolation and explicit-approval authorization. Operational access to each namespace is managed through *Roles* and *RoleBindings*. Roles align with operational responsibilities, which determine which actions (called verbs in Kubernetes) can be taken on various resources by their type. RoleBindings associates *users* and groups to each Role. When a single team governs multiple applications (i.e. namespaces), that team's group should be listed as a subject in the RoleBindings for the appropriate namespaces. Similarly, if multiple teams contribute to an application, it's recommended to represent those as groups in your identity management system, and configure your RoleBindings's subjects to grant those teams (as groups) access to the namespace for that application.

- **ResourceQuota:**
  The "noisy neighbor" problem can be mitigated by leveraging a *ResourceQuota* that can be defined on each namespace, with fixed CPU and Memory allocation. When needed, this quota can be revised to expand or reduce the resources available to each application, thus ensuring that one application cannot consume so much of a cluster's compute or storage capacity that it affects its "neighbor" applications. We will discuss resource management in greater detail in a dedicated section, below.

- **NetworkPolicy:**
  Applications, or more specifically pods, are reachable by default, by all other *pods* in the cluster. Applications in Kubernetes can expose their services using Services. These become discoverable via DNS within the cluster. This is extremely helpful for interconnecting application components, but requires some security considerations in multi-tenant environments. Specifically, it is necessary to isolate tenant environments at the network layer, by default, but allow certain services to be accessible across namespaces. To facilitate this need, Kubernetes provides NetworkPolicy, which behaves like an intra-cluster firewall that works with Kubernetes concepts of Namespaces and Pods. Each NetworkPolicy defines ingress and egress rules for communication between pods, across namespaces. Policies can be defined that control cross-namespace network access, as well as intra-namespace access. For example, you can use NetworkPolicy to restrict database pods to specific API services in the same namespace, and force clients to interact with your web API in a microservice architecture. Building on this concept, a best practice to isolate each tenant and address the "nosy neighbor" issue, is to apply a default Network Policy to all tenant namespaces, which blocks access from other namespaces. When needed, the policy can be revised to open specific application ports within the cluster to other pods in other namespaces.

Resource Management and Quotas: Preventing Noisy Neighbors

As multi-tenant clusters share finite resources among all tenants and across a fixed number of nodes, administrators must ensure that tenants do not use more than their fair share of resources. Kubernetes offers *ResourceQuotas* to enforce fair utilization of shared resources. Each namespace can be allowed to claim a specific amount of CPU or Memory capacity and to be constrained on numerous other resources. Every tenant namespace should be created with a default *ResourceQuota*, which can be revised by administrators to accommodate each application's resource demands as it evolves.

The most common quotas will focus on CPU and Memory allocation, as these are typically the most precious of the cluster's resources. However, additional constraints on other cluster resources may be useful. For example, a quota can be defined to limit the number of pods in the namespace, which will indirectly also limit the number of IPs allocated to pods in each namespace's application. Similarly, Services exposed, either by LoadBalancer or ClusterIP, can also be constrained. This type of quota allows each tenant namespace to operate within a resource consumption budget, which indirectly contributes to controlling the operating cost of the cluster.

Each application pod should define *requests* for CPU and memory, and these will be strictly honored by the scheduler. If an application's requests cannot be fulfilled, it will fail to deploy; requests are a hard commitment. Requests for all the pods in a namespace, in sum, cannot exceed the total value of the quota's request maximum.

Applications pods should also define *limits*, the upper bound of the pod's resource usage. When an application exceeds its memory limits, Kubernetes will restart it. When the cluster is scheduling a workload, limits are treated as a soft commitment, and may cumulatively exceed the total capacity of the cluster. This can allow a cluster's resources to be *overcommitted*, which allows applications to burst their usage when needed.

Cluster administrators should define a default *LimitRange* for every namespace. This action will impose a default set of requests and limits on each pod, establish minimum and maximum resource allocations, and constrain the allowed *ratio* of limits and ranges. Applications will then be required to fit their resource consumption with this policy. The LimitRange policy can be adjusted on a per-namespace basis later, to accommodate specific application requirements.

Consider also using the *vertical pod autoscaler (VPA)* to manage CPU and memory allocation based on application metrics. The VPA automatically updates resources and limits, while also staying within the bounds of any namespace *quota* assigned. However, some caution is required as the VPA may also conflict with *LimitRange* if not correctly configured.

Be sure to keep in mind that CPU is "compressible" but that memory is not. Essentially, this means that time-sharing can allow overcommitment of CPU resources, but memory is finite, and workloads occupy it to the exclusion of others.

Most applications exhibit "bursty" consumption of their resources, both CPU and memory, "bursting" at different times. This consumption pattern allows a finite resource like memory to be reallocated over time. Therefore, it is usually safe to *overcommit* clusters, but the advisability of this practice depends heavily on the specific workloads and how they operate.

Note: Some applications claim a large block of resources, particularly memory, and rarely release it. Many Java applications behave this way due to their underlying virtual machine functionality. In these cases, it's often best to configure limits equal to requests, to ensure that the needed resources are available and that the cluster will not attempt to overcommit the resources allocated to this application.

Tuning Quality of Service for Workloads

Kubernetes is designed to be resilient to inevitable infrastructure failures, but to preserve applications, it sometimes has to relocate them to new machines. For this process to maintain application service levels, Kubernetes needs to understand the priority of workload components in order to balance application availability in resource-constrained scenarios. This allows Kubernetes to maintain the performance of key business applications, and deprioritize less critical workloads, when needed.

For best results:

- Define a robust set of *PriorityClasses* for applications to use, both for tenants and administrators. These are global and must be managed by cluster administrators. An example model is provided below.
- Take advantage of the *quality of service* model in Kubernetes, assigning appropriate requests and limits for each workload. Understand how this impacts *pod eviction* when resources are constrained.
- All applications must define a *PodDisruptionBudget* for critical components.
- All workloads must define pod priority for critical components.
- The use of *inter-pod affinity* should be minimized, and applied only as a performance optimization, or when critical applications' performance requires components to be running on the same machines.
- Enforce *quotas on priority classes* to ensure fair prioritization of application components.

With a strategy modeled on these principles, Kubernetes will make better-informed decisions about workload orchestration, during regular operations, as well as in failure modes that constrain available capacity.

Baseline Pod Priority Model

This model is intended to be foundational. It should fit many use-cases, while still providing a lot of room for customization, and creation of intermediate classes.

Priority Class Name	Purpose (example)	Value/Priority
Cluster Core	Essential services to operate the cluster itself, such as Dex for authentication	100000
Tenant Critical	Services which are business-critical to *your tenant*, and cannot easily move to a new machine, such as database back-ends	70000
Administrative Services	Dashboards that are important for managing the cluster, but could tolerate a short outage while moving to a new machine	50000
Tenant Stateless	Services which a tenant can temporarily lose in the event of infrastructure failure, but respond well to scaling events, such as stateless front-end APIs.	10000
Tenant Batch	Background Jobs of the tenants application, which could safely terminate and restart but are somewhat costly to reproduce	2000
Best Effort (*default*)	No priority assigned; allows termination in favor of higher priority workloads	100

Note: Kubernetes provides two built-in priority classes: *system-cluster*-critical and *system-node-critical*. These provide a much higher priority value, and should not be used by tenant workloads.

Providing a model like this to your tenants enables a self-service approach to workload classification, and allows them to independently govern the resilience of their workloads.

Preventing Nosy Neighbors: Securing Soft Multi-tenancy

In addition to the practices and mechanisms outlined above, some security precautions are warranted, if only to mitigate the risks of unknown vectors in your applications. This is where the concept of soft multi-tenancy comes into play. Soft multi-tenancy does not incorporate strict isolation of your applications, workloads, and users. This methodology is based on the trust of your users and to help reduce accidental access to your tenants.

- Ensure that Kubernetes *EncryptionConfig* is configured to encrypt secrets at rest (in etcd).
- Disable the use of HostPath volume mounts. When pods can mount from the host, this creates an injection vector across workloads, which can also span across multiple tenants sharing the same machine.
- Consider using *PodSecurityPolicy* to constrain pods' kernel-level security options, particularly disabling privileged pods. This can also be achieved using *Gatekeeper* policies.
- Consider using *gvisor* to limit the kernel APIs that containers can leverage; this reduces the attack service for malicious workloads attempting to "breach the container" and take control of the host operating system.
- Consider disabling the use of PersistentVolumeClaims (and PVs) in tenant namespaces.
    - Provide a centrally managed object-storage solution in the cluster, such as Minio, with a distributed storage back-end external to the cluster.
    - *Require* applications to use the centrally managed object storage, with end-user authentication.
    - Selectively allow Persistent Volume usage for specific applications (e.g. databases).
- Encourage the use of Ingress instead of exposing Services. This allows tenants' web applications to leverage a central routing proxy, implemented by an *Ingress Controller*, simplifies TLS certificate management, and reduces costs associated with load-balancer infrastructure.
- Use Secrets in tenant namespaces to share access to an external secrets store such as Hashicorp Vault; this allows control of the secrets to be removed from Kubernetes, and properly configured applications to retrieve them at runtime, and each tenant can be granted specialized access within Vault.

Additional Considerations for Hardened Multi-tenancy: Keeping Out Burglars

The practices and mitigations described until now fit with various models of soft multi-tenancy, predominantly focused on preventing accidental access to unintended resources.

Hard multitenancy aims to accommodate multiple tenants with unrelated business objectives, possibly adversarial, and potentially seeking to exploit vulnerabilities with malicious intent in the system to gain access across tenant boundaries.

All of the precautions described above should be applied in a hard multi-tenancy model. Some variations should be considered.

- Disable the use of NodePort and HostPort Services, as these expose applications on discoverable ports in the host network, and may bypass NetworkPolicy controls in some network environments. Require all tenants to expose Services using type ClusterIP (internally) or LoadBalancer (externally).

- Prevent tenants from altering their ResourceQuotas and LimitRanges, in the Tenant Administrator role; a malicious tenant may deploy workloads specially designed to consume gratuitous cluster capacity, in an attempt to deprive other tenants' workloads of their resources.

- If tenants require isolated machines, consider splitting the cluster into node pools per tenant, with labeled nodes, and use the *PodNodeSelector* feature to confine each tenant namespace to those specific nodes by label. This can be used in combination with *taints and tolerations*.

- Use the *NodeRestriction* feature to prevent nodes from altering certain labels and taints, specifically for the purpose of workload isolation.

- If you are operating bare-metal clusters, consider slicing machine capacity using virtual machines for each node, to isolate each tenant to a VM, while sharing hardware. This reduces the risk of a compromised host, because each Kubernetes node (VM) will have its own kernel instance.

Kubernetes Behind the Scenes for Service Providers

In many service provider contexts, tenants do not need direct access to the Kubernetes API, but should instead interact with other application APIs. These application APIs would be responsible for managing the lifecycle of tenants' isolated application instances.

This model would be appropriate if you:

- Host a common set of applications with different instances for each tenant, such as a blog platform, and tenants interact only with the application instance.
- Provide higher-level APIs that accept a tenant's workload parameters (e.g. as a PaaS), but leverage Kubernetes "behind the scenes".

In these cases, it's critical that your application layer should automate all of the preceding recommendations to ensure isolation of the tenant applications, particularly for NetworkPolicy, ResourceQuota, and LimitRange.

Because such tenants do not have direct access to Kubernetes, a few more flexible alternatives can be considered. Note these could be potentially bypassed by a malicious user with specially crafted pod deployments if they gain direct access to the Kubernetes API. Further, these configurations require a suitable cluster topology that provides nodes dedicated to each tenant, which may compromise the goal of infrastructure cost reduction.

- Using *pod anti-affinity* to avoid scheduling pods on machines that also serve another tenant's application instance.
- Using *node taints and pod tolerations* to identify specific nodes for a tenant workload.

Managing Multiple Clusters

Some workloads are so sensitive that sharing either infrastructure or administrative APIs is not acceptable. In these cases, as strict isolation is truly critical, it may still be advisable to operate separate clusters. This allows strict separation of the physical hardware, networks and network overlays, virtual machines (if applicable), and management APIs.

Until recently, it has been very common for organizations to delegate entire clusters to specific teams, resulting in the cluster sprawl mentioned previously. This practice has the benefit of controlling the blast radius in the event of a cluster-wide failure, practically turning each cluster into its own failure domain.

The trade-off: for more isolated risk profiles, you take on the increased overhead cost, and more complex management burden. A good multi-cluster strategy is critical.

In a multi-cluster strategy, we move from a world of single multi-tenant clusters to an ecosystem of multiple single-or multi-tenant clusters. In addition to all of the best practices stated above, some new governance and compliance challenges need to be addressed.

- Identity and access management must be consistent across all clusters to simplify security controls. Identity providers, such as an enterprise LDAP service, should be used on all clusters with Kubernetes native OIDC configuration or an authenticating proxy. RoleBindings should refer to subject groups (not individual users) so that authorization, by way of group membership, can be governed within the central IDP.

- RBAC (access control) should be audited regularly across all clusters to ensure that the correct privileges are granted to all staff.

- Component audits of all clusters should be executed frequently to alert on critical vulnerabilities to help mitigate risk.

- Workloads on all clusters should be audited for compliance with enterprise policy to ensure that all components are running with appropriate security patches and configuration requirements. (For example, to ensure that all databases are reachable only via TLS/SSL sockets, and sensitive data is stored on encrypted volumes.)

- When services are exposed for inter-cluster communication, TLS certificate management becomes a critical function. These should be audited regularly and renewal should be automated. Kubernetes' native certificate management provides a strong foundation, but additional tools are needed for auditing, reporting, and lifecycle automation.

- Metrics and logging services should be federated and their data archived in long-term storage. This data should reside outside each cluster, such as in a central S3 bucket, and a central dashboard should raise alarms from any cluster showing signs of degraded performance or malfunction.

Note: These considerations are just the beginning. As Kubernetes evolves and new functionality is established, new auditing capabilities will be needed.

It is critical in a multi-cluster strategy to consider risk domains from a business perspective. This includes regulatory environments, compliance objectives, and other business requirements that vary according to workload. In many organizations, these requirements justify organizing clusters according to each risk domain so that workloads can be deployed with automated controls that enforce that appropriate policy.

For example, it may be beneficial to have several clusters arranged like so:

- A cluster in the EU for processing financial data for EU customers
- A cluster in the EU for serving customer-facing applications
- A cluster in the US for processing internal business applications
- A cluster in the US for processing financial data for US customers

Each of these clusters represents a different risk domain (such as data sovereignty), and automatic enforcement of the required configuration for each workload can be applied at the cluster level. Then, applications can rely on the cluster infrastructure to provide these capabilities, rather than each application itself having to reinvent such enforcement independently.

These challenges are solvable with the help of open source or commercially-available multi-cluster management tools.

Monitoring your Kubernetes API

The control plane becomes even more important as you bring on more tenants, because it becomes the single point of failure for a cluster. Thus the **Kubernetes API** is a critical service for you and your teams: it provides the interface for administrators and tenant teams, and the associated automation needed to manage the lifecycle of both the cluster and resident workloads. When this API service is degraded, everyone feels the pain. Keeping the Kubernetes API responsive is a critical part of an effective multi-tenant strategy for offering clusters as a service to your organization.

A Kubernetes cluster with a single control plane is not considered production grade. A production Kubernetes cluster MUST have at least 3 control planes.

- Even with a three-node control plane, any failure will make the cluster inoperable. Technically, it enters a “read-only” state which prevents it from handling application recovery if other cluster nodes fail at the same time. Mission Critical control planes should generally have 5 nodes, as this provides an effective minimization of operational risk. Exceeding 5 control plane nodes offers very little benefit, in practice.

- A resilient control plane (3-5 nodes) allows for cluster maintenance (scheduled or unscheduled) without producing an outage. This allows for patching of vulnerabilities, or updating of critical cluster components with little to no impact to the teams using, or workloads running on associated clusters.

Early in the Kubernetes project, the developers outlined two critical metrics for cluster performance that constitute a good *preliminary* SLO:

- **p95 API Response time less than 1 second** indicates that the control plane itself is operating in reasonably healthy conditions.

- **p95 Pod start duration less than 5 seconds** for pods that do not require an image pull (i.e. precached). This indicates that many cluster components are working correctly, particularly inter-component orchestration, and that sufficient capacity is available to maintain current workloads.

As clusters grow with more machines and their associated workload, each machine imposes load on the API. If API response time degrades, it may be appropriate to scale the control plane with *higher-capacity machines (Vertical Scaling)*. Some applications integrate directly with the Kubernetes API, and if poorly written, may put heavy load on the API components of the cluster—having the same effect as a Denial of Service attack. In this case, all teams may be affected by a reduction in performance, and application recovery in the event of infrastructure failure would also be compromised.

An effective monitoring system should provide insight into both the availability of Control Plane Nodes and the performance of API calls, as well as configuration change tracking:

- To begin with, it should alert you whenever any control-plane node is unresponsive (availability).

- Cluster operators should include API metrics in their regular operations dashboards, in addition to the conventional infrastructure health metrics.

- In multi-tenant environments, it becomes especially critical to enable and use Kubernetes' audit logging capabilities; every request, change, and query to the Kubernetes API gets logged, and individual users' actions within Kubernetes can be monitored.

Prometheus is a CNCF Time Series Data Store that can be used to gather and aggregate performance and availability metrics as forwarding alerts based on mythic thresholds. Grafana is a popular package used to create dashboards and monitoring views of the data stored in Prometheus.

Conclusion

Whether you're new to Kubernetes and looking for an enterprise solution or your organization has prolific adoption of Kubernetes in many clusters, a multi-tenancy strategy can offer significant benefits in the right circumstances. These benefits include improved returns on IT investments (including both capital and operating costs), the ability to offer infrastructure as a service (IaaS) to internal teams, and improved IT infrastructure performance and service levels.

Kubernetes' native capabilities provide a robust soft multi-tenancy solution, and in combination with production-focused Kubernetes management tools such as D2iQ's Kommander, can be hardened to deliver strong isolation that fits the needs of enterprises and service providers.

Strong configuration management and resource management tools and practices can lead to drastically improved cluster ROI, by employing a multi-tenancy model that maximizes application density while recovering quickly from infrastructure failures. By adopting and adapting the practices outlined here, you'll reduce maintenance and operational costs, and your application teams will enjoy a more reliable and resilient platform.

In multi-tenant environments, cluster operations discipline becomes critical to preserving the value of the applications. The practices outlined here will greatly improve your tenant experience, with more reliable clusters and stable application environments.

For more information on how
we can help, please visit:

d2iq.com