POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Multi-Tenancy in Kubernetes Clusters

Supervisors

Prof. Fulvio RISSO

Dott. Federico CICCHIELLO

Candidate

Attilio OLIVA

October 2024

Summary

The growing demand for Kubernetes clusters and the increasing computational investments for 5G telecommunication infrastructure are leading to a big opportunity: competitiveness in the cloud computing market sector. Telcos could enter this market by leveraging spare resources from their infrastructure and providing Kubernetes clusters as a service [1]. Nevertheless, a novel obstacle is presented by the Kubernetes design, as it was not designed with multi-tenancy in mind. In order to overcome this limitation, the conventional provisioning approach involves the installation of a dedicated cluster within Virtual Machines for each tenant.

Conventional provisioning techniques frequently result in resource waste and higher operating expenses to maintain tenant isolation. With an emphasis on resource allocation optimization and scalability enhancement while maintaining tenant isolation, this thesis explores resource-efficient multi-tenancy strategies for Kubernetes clusters, especially in the context of bare-metal deployments. The investigation resorted to adding multi-tenancy capabilities to a Kubernetes cluster, since there was no viable technology to enhance dedicated clusters. To accomplish this, a collection of technologies is needed to implement Kubernetes control and data plane isolation. One of the most intriguing control plane isolation techniques to emerge is the concept of virtual clusters. This approach enables the sharing of a single Kubernetes cluster by deploying specialized components that, while appearing as independent entities, primarily delegate operations to the underlying shared cluster. Meanwhile, the only data plane isolation that has been researched is pod sandboxing, which uses containers inside virtual machines (VMs) and is the most practical method in this situation. After comparing dedicated and shared cluster solutions, it was proved that the virtual cluster with pod sandboxing required fewer resources and produced a workload that was more efficient.

Another significant challenge in practice is the management of multiple tenant clusters. While multiple tenant clusters can be deployed on a single bare-metal cluster, the complexity increases when managing multiple clusters across different bare-metal environments. This work provides a concise overview of multi-cluster management strategies based on ClusterAPI, progressing from basic methods to more scalable and resilient solutions using Hosted Control Planes.

Acknowledgements

Trovare le parole per esprimere la mia gratitudine verso le persone che hanno reso possibile questo percorso mi ha messo in difficoltà, perché credo che siano così forti da non riuscire a trasmetterle fedelmente con questo mezzo. Nonostante questa avversità proverò comunque a farlo, consapevole che ogni frase sarà solo un timido eco di ciò che sento davvero.

Al prof. Risso, per i suoi insegnamenti solidi, la pazienza e la costante disponibilità dimostrata nei miei confronti. Mi ha spronato a crescere oltre i miei limiti e a migliorare con impegno.

Al gruppo di lavoro in TIM, per avermi accolto con fiducia e per avermi permesso di mettere in pratica ciò che era solo teoria. Ogni confronto è stato un'occasione di crescita personale e professionale. Un ringraziamento speciale va **a Federico Cicchiello**, il mio correlatore, per la sua guida costante e i preziosi consigli durante tutto il percorso.

Al Netgroup e a tutti coloro che frequentano il Lab9, grazie per avermi accolto con calore e per aver creato un ambiente collaborativo e stimolante. Ogni consiglio, ogni discussione, e ogni gesto di aiuto, anche nelle piccole cose, hanno reso il mio percorso più ricco e meno faticoso. Avete condiviso non solo il lavoro, ma anche un pezzo di vita, trasformando questo tempo in laboratorio in qualcosa di più che semplice ricerca.

Ai miei genitori, un ringraziamento profondo per il vostro supporto incondizionato, anche quando le nostre idee differivano. Nonostante tutto, avete sempre creduto in me, permettendomi di seguire il mio cammino con fiducia. È grazie alla vostra presenza costante e al vostro amore che oggi posso guardare con orgoglio a questo traguardo. Un pensiero speciale va **a mia madre**, che ha affrontato la battaglia contro un tumore durante questo percorso. Nonostante la sfida immensa, hai sempre trovato il modo di sostenermi e credere in me, anche quando eri tu ad avere bisogno di sostegno.

A tutti voi, grazie di cuore. Questa tesi è il risultato delle vostre mani, delle vostre idee, del vostro sostegno.

Table of Contents

Chapter 1

Introduction

The demand for cloud computing services has increased over the past few years due to its decreasing cost. As a result, companies are moving away from large monolithic software products and embracing cloud-native applications, which consist of small, independent, and loosely coupled services. Kubernetes [2] is currently the most widely used method [3] available in the market for managing borrowed resources and orchestrating applications on the Cloud. A ready-to-use Kubernetes cluster that allows users to place their own workload easily and conveniently, is therefore a highly sought-after service by cloud companies.

The demand for Kubernetes-as-a-Service is soaring, but only large corporations with substantial infrastructure investments can handle demand on this scale. These large corporations, known as hyperscalers, form the foundation of the current global cloud services infrastructure. Attracted by the service's profits, other emerging-market players are attempting to use their smaller infrastructure more effectively in an effort to compete for a portion of the market. Unexpectedly, though, even telecommunications companies across the globe are becoming more and more interested in this field. New telecommunication standards are requiring an infrastructure with more and more computational power, especially with the advent of 5G. Because investing in faster protocols is getting more expensive, telcos are stuck in a situation where customers won't accept any bill increases in exchange for the improvements they demand. Leveraging spare computational resources to enter the cloud computing market is an idea that is beginning to take shape. Despite the fact that a telecommunication infrastructure is made up of closed, heterogeneous devices, businesses in these fields are working with manufacturers to use them essentially as bare-metal servers; the Sylva project [4] is one example of such an effort. In either case, small businesses with infrastructure are exploring ways to utilize their resources more effectively to enter the competitive, yet lucrative cloud market.

1.1 Goal of the Thesis

An intriguing concept in software is **multi-tenancy**, where a single instance can handle multiple groups of users, referred to as tenants. The design of Kubernetes did not originally consider this concept, which continues to pose a challenge today in terms of finding ways to implement it. One objective is to explore all possible methods for creating separate clusters by incorporating the multi-tenancy feature within an infrastructure consisting of Kubernetes clusters installed on bare metal. The ultimate aim is to identify the **most efficient production-ready solution**, taking into consideration **resource overhead** and **operational costs**, especially in an environment where there is a **lack of trust** between tenants and a limited number of available small clusters (i.e. orders of 10) each one composed of scarce resources (i.e order of 10 servers).

Chapter 2

Kubernetes

Understanding Kubernetes is crucial for this work, as it serves as its foundation. Kubernetes, also known as K8s, is a complex framework with a wide array of features. While a detailed analysis would require further investigation, we will focus on its key concepts and elements, particularly its extensibility.

2.1 History

Originally, applications were deployed as monoliths, necessitating entire servers (bare metal) dedicated solely to them. Subsequently, virtualization emerged, enabling multiple applications to coexist on a single server by partitioning the machine resources into smaller segments. Although this was an improvement, applications still bore the burden of a complete operating system. Currently, with containerization, applications are deconstructed into compact, specialized microservices, each housed in its own lightweight container, requiring only its specific functionality. This approach facilitates expedited deployments, improved scalability, and simplified development processes, ultimately allowing for a more concentrated focus on service creation rather than on managing individual servers.

Google was already adopting this approach from its early stages starting with the Borg system, a small project developed in 2004 [5]. As interest in container orchestration grew, in 2014 Google introduced Kubernetes as an open-source version of Borg. Unlike alternative solutions, Kubernetes is distinguished by its openness and extensibility, which have contributed to its widespread adoption. This versatility allows it to be tailored to any use case as necessary.

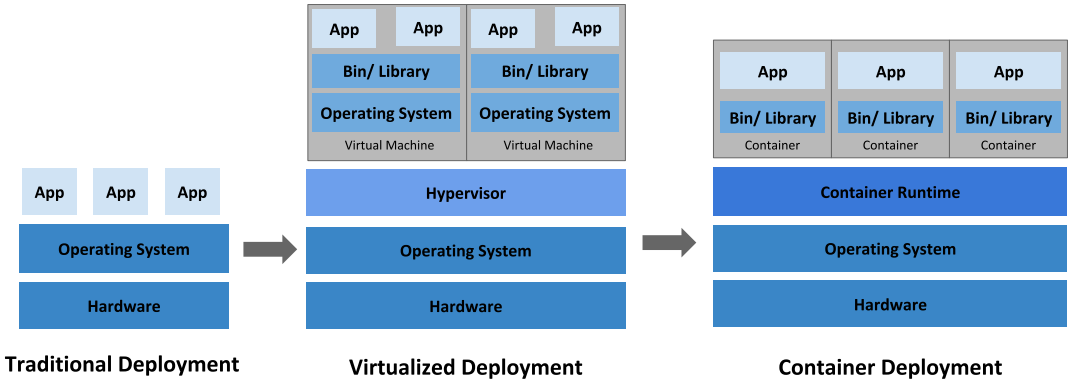

Figure 2.1: Cloud computing model evolution

2.2 Kubernetes Architecture

A Kubernetes cluster is a distributed system that comprises multiple machines, referred to as nodes, which are responsible for running containerized applications. These nodes are categorized into two types based on their role in the cluster: **master nodes** and **worker nodes**. The *master nodes* host the control plane, which is the central management layer of Kubernetes. The control plane's components are responsible for orchestrating and maintaining the desired state of the cluster. These components collectively manage the deployment, scaling, and lifecycle of the containerized applications by controlling the worker nodes and the Pods. *Worker nodes*, on the other hand, are responsible for executing the application workloads. Each worker node runs **Pods**, which are the smallest deployable units in Kubernetes. Worker nodes communicate with the control plane to receive instructions about which Pods to run, monitor the state of these Pods, and report back to the control plane regarding their health and status.

In a typical production environment, a Kubernetes cluster employs multiple master nodes to ensure redundancy and resilience. By running the control plane across several master nodes, the cluster can continue functioning even if one of the master nodes fails, providing **high availability**. This multi-master architecture, along with the distribution of worker nodes, enhances fault tolerance, making the cluster resilient to hardware failures, network partitions, or resource constraints on individual nodes.

2.2.1 Control Plane components

The control plane is the brain of a Kubernetes cluster, responsible for managing and maintaining the desired state of the system. It consists of several key components

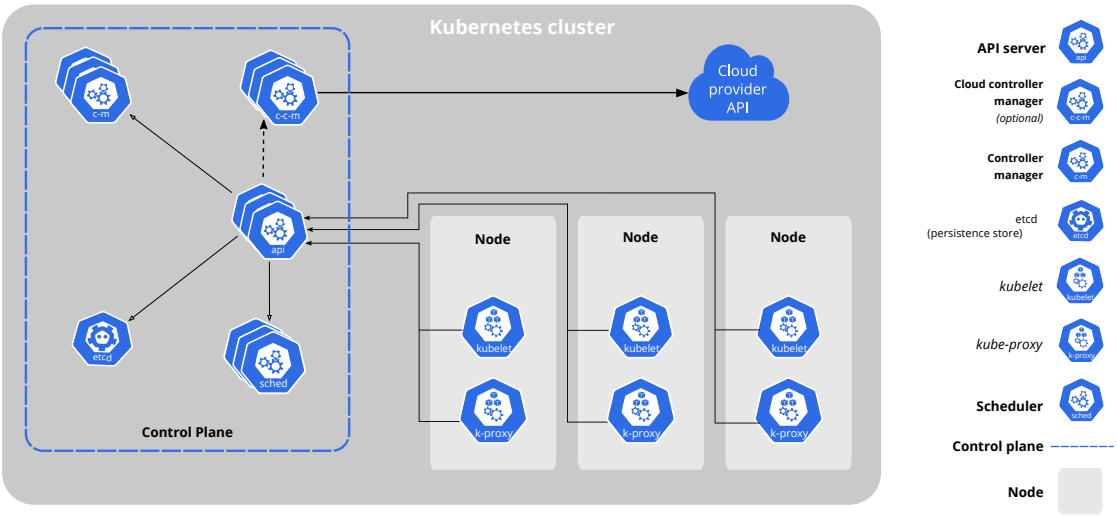

Figure 2.2: Cloud computing model evolution

that work together to ensure that the cluster functions efficiently, orchestrating containerized applications across nodes. The following subsections describe the major components of the Kubernetes control plane in detail.

etcd (datastore)

At the core of Kubernetes' control plane is the etcd[6] component, which acts as a distributed, consistent, and highly-available key-value store. It serves as the primary datastore for all cluster information, making it a critical component for the operation of Kubernetes. Every piece of configuration data, cluster state, and metadata—ranging from node status and pod specifications to service configurations—is persistently stored in etcd.

Designed for high availability, etcd uses a distributed architecture where data is replicated across multiple nodes. This ensures that even in the case of a node failure, the system can recover from other nodes holding consistent copies of the data. The consensus algorithm Raft underpins etcd's consistency model, enabling the system to tolerate failures while maintaining strong data consistency, a critical requirement for distributed systems like Kubernetes. Its reliability and efficiency are foundational to the cluster's resilience and stability, making etcd one of the most crucial components in the Kubernetes ecosystem.

kube-apiserver

The kube-apiserver is the primary interface for the Kubernetes control plane, exposing the Kubernetes API to internal and external clients. It functions as the

front-end for the entire control plane, processing API requests from users, CLI tools, and other Kubernetes components. The API server is responsible for handling all RESTful requests for querying and modifying the state of resources within the cluster, such as Pods and Nodes.

Upon receiving a request, the kube-apiserver first authenticates and authorizes it, ensuring that the request is valid and that the client has the required permissions. Once validated, it processes the request by either retrieving data from or writing data to etcd. Notably, the kube-apiserver is the only control plane component that directly interacts with etcd. All other control plane components access the cluster state indirectly via the API server, reinforcing the kube-apiserver's central role in maintaining system coherence.

In a highly available Kubernetes cluster, the kube-apiserver can run on multiple master nodes, and load balancers are typically used to distribute API requests, further enhancing its resilience and ensuring consistent access to the control plane

kube-scheduler

The kube-scheduler is a control plane component that monitors for newly created Pods without an assigned node and selects a node for them to run on. It is responsible for deciding which pod goes on which node, taking into account factors such as individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, and inter-workload dependencies. However, it doesn't actually place the pod on the nodes.

Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

kube-controller-manager

In Kubernetes, a controller is a control loop that watches the state of the system and takes action to achieve the desired state. The kube-controller-manager is responsible for running several controllers in the cluster, each of which manages different aspects of the system. Each controller within the kube-controller-manager operates independently but interacts with the kube-apiserver to monitor the current state of its specific resource. When discrepancies between the actual state and the desired state are detected, the controller takes the necessary steps to reconcile them. This could involve scaling Pods, updating configurations, or rescheduling workloads.

cloud-controller-manager

The cloud-controller-manager is an optional but important control plane component in environments where Kubernetes is integrated with cloud infrastructure. It provides cloud-specific control logic by allowing the cluster to interface with a cloud provider's API, thus abstracting cloud operations away from the core Kubernetes components. By separating cloud-specific logic from Kubernetes' core components, the cloud-controller-manager helps maintain a cleaner separation of concerns. This design also allows Kubernetes clusters to be more portable and enables integration with a wide range of cloud providers, while leaving the core Kubernetes architecture unaffected by provider-specific requirements.

2.2.2 Node components

The node components in Kubernetes are essential for the execution of application workloads. Each node in a Kubernetes cluster is responsible for running the necessary software to deploy, manage, and maintain containerized applications. These node components work in tandem with the control plane to ensure that Pods are appropriately scheduled, network traffic is routed correctly, and containers are managed effectively. Below is a detailed overview of the key node components.

kubelet (node agent)

The kubelet is the primary agent that runs on each node within a Kubernetes cluster. Its main function is to manage the containers on its host node, ensuring that the correct Pods are running as specified by the control plane. The kubelet communicates directly with the kube-apiserver to receive instructions about which Pods need to be deployed or managed on its node. The kubelet operates by continuously monitoring the Pods assigned to its node. It checks the desired state of the Pod (as defined by the control plane) and works to maintain that state. For instance, if a Pod is scheduled to run on a particular node, the kubelet ensures that the necessary containers are launched. If a container within a Pod fails or crashes, the kubelet automatically attempts to restart it, ensuring minimal downtime. By ensuring that the containers are operating as expected, the kubelet plays a crucial role in maintaining the health and stability of workloads on each node.

kube-proxy

kube-proxy is a network component that runs on each node in your cluster, implementing part of the Kubernetes Service concept (described in 2.3.2). Its primary function is to maintain network rules that route traffic from services to the appropriate Pods, enabling communication within the cluster and externally. It facilitates

service discovery by ensuring that traffic sent to a Kubernetes Service is correctly routed to the underlying Pods associated with that service. By handling service-to-Pod routing and load balancing across the cluster, kube-proxy ensures seamless communication within the Kubernetes network, making it possible for applications and services to scale horizontally 2.6 without requiring manual reconfiguration.

In cases where a custom network plugin is used, which directly implements packet forwarding for services, kube-proxy may not be required. These plugins can provide equivalent behavior to kube-proxy, eliminating the need for it on the nodes.

Container runtime

The container runtime is responsible for running the containers that encapsulate the application workloads in Kubernetes. Kubernetes supports a range of container runtimes via the **Container Runtime Interface** (**CRI**), allowing users to choose from different container runtime implementations based on their specific needs. The CRI is an abstraction layer that defines how the kubelet communicates with the container runtime on a node. Kubernetes adheres to the **Open Container Initiative** (**OCI**), a set of standardized specifications for container formats and runtime behaviours. This ensures that Kubernetes can work with any container runtime that complies with the OCI standards. Popular container runtimes that are CRI-compliant include containerd and CRI-O, both of which rely on the low-level runC runtime.

2.3 Kubernetes Fundamentals

Kubernetes is designed to be highly flexible, extensible, and interoperable with various third-party tools and systems. This flexibility is largely due to its API-driven architecture, which allows developers and operators to manage resources, configure infrastructure, and orchestrate workloads in a standardized way. Rather than tightly coupling Kubernetes to specific implementations of core systems like networking and storage, Kubernetes defines a set of APIs that third-party projects can use to create their own implementations, adhering to the Kubernetes model. The foundation of this API-centric approach lies in the ability to define and manage resources in a consistent manner, regardless of the underlying infrastructure. This section outlines some of the fundamental concepts that govern how Kubernetes operates, with a focus on how resources are defined and how networking is managed within the cluster.

2.3.1 Kubernetes Objects

In Kubernetes, any persistent entity that can be created, updated, or deleted is referred to as a Kubernetes object. These objects represent the desired state of the cluster and are defined declaratively, using YAML manifest files. Each Kubernetes object has a well-defined schema and is represented in the Kubernetes API, allowing users to interact with these resources programmatically through the API or manually via configuration files.

One of the core principles of Kubernetes is its object-oriented approach to resource management, where each object has specific attributes and metadata. Objects define the configuration of a particular resource and are processed by the control plane to ensure that the actual state of the cluster matches the desired state placed in the resource specification.

Listing 2.1: Example of a Pod resource YAML

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
```

2.3.2 Networking

The Kubernetes networking model is structured around a pod and service cluster-local network. In the pod network, each pod is assigned a unique IP address that is valid across the entire cluster. Kubernetes ensures that a service, also assigned a unique IP address, is consistently accessible within the cluster through the **Service** API. This is a complex process, as the service must be correctly proxied to the node where the relevant pod is running. To expose services externally, Kubernetes offers the **Ingress** API, which facilitates external access to internal services. Another important feature, **NetworkPolicy**, provides control over traffic flow both within the cluster (between pods) and between the cluster and external systems.

The pod network namespace is established by the software implementing the Container Runtime Interface (CRI). However, the management of pod-to-pod networking is handled by the **Container Networking Interface** (CNI) implementer. By default, pods cannot communicate properly with one another; it is

the responsibility of CNI software to regulate this interaction, ensuring that pods can connect both within the same node and across different nodes in the cluster. Additionally, the CNI is tasked with enforcing NetworkPolicy rules. Without proper CNI implementation, network policies will remain ineffective, as they will not be translated into enforceable rules at the pod level.

2.3.3 Storage

By default, files within a container are ephemeral, meaning they are lost once the container is shut down. This characteristic simplifies the management of volumes dedicated to pods, as a portion of the local node's storage can be allocated to its pods without concern for data persistence. If a pod is rescheduled or moved to another node, any data stored locally is lost, which is acceptable in many cases. However, for stateful applications—such as databases, file systems, or other applications requiring persistent data—this poses a significant challenge. Containers can terminate, be rescheduled, or migrate to different nodes, potentially leading to data loss or unavailability unless there is a mechanism for persistent storage.

To address this issue, Kubernetes offers a dynamic and flexible storage solution through several key components:

- **StorageClass**: Defines different classes of storage available within the cluster, along with the corresponding volume plugin used for provisioning (via the **Container Storage Interface** or **CSI**) and the behaviour when a pod is deleted (through the **reclaim policy**). Similar to the Container Networking Interface (CNI), CSI provides a standardized method for exposing arbitrary block and file storage systems to containers.

- **Persistent Volumes** (**PV**): These are storage resources within the cluster, provisioned either dynamically using a StorageClass or manually by an administrator. Unlike traditional volumes, PVs have a lifecycle independent of any individual pod that utilizes them, ensuring data persistence even when pods are terminated or rescheduled.

- **Persistent Volume Claims** (**PVC**): Applications request storage through PVCs, which consume PV resources. PVCs specify the desired storage size and access modes, allowing applications to request exactly the storage they need.

By using these APIs and leveraging the CSI without demanding a specific file storage system, Kubernetes enables a highly flexible storage architecture. This approach allows any storage solution to be used for providing volumes, making it adaptable to various use cases and user requirements.

2.3.4 Namespaces

In Kubernetes, namespaces provide a means of isolating groups of resources within a single cluster, preventing name collisions by assigning a scope to objects. However, namespaces are not applicable to cluster-wide resources such as Nodes, StorageClass, or PersistentVolumes, which operate at a broader level. For instance, Persistent Volume Claims (PVCs) can be namespaced because they are tied to specific applications, which can be effectively scoped within a namespace. In contrast, cluster-wide resources like Nodes or StorageClass exist at the cluster level, as applications should not directly interact with or alter them.

This distinction is crucial to understanding the ongoing investigation in this work, as it highlights the limitations of namespaces and their applicability to different types of resources within a Kubernetes cluster. Recognizing which resources are namespaced and which are not is fundamental to comprehending how resource isolation and management are implemented in Kubernetes.

2.4 Custom Resources and Controllers

Custom Resource Definitions (**CRDs**) provide a mechanism for extending the Kubernetes API by allowing users to define custom resource types that are not included in Kubernetes by default. Pods, Services, and Deployments are just a few of the pre-built resource types that come with Kubernetes. CRDs, however, make it possible for businesses to define additional resource types that are tailored to their needs, which is a necessity in many situations. After being registered by Kubernetes on the CRD creation, the new resource type can be used in the same way as any native Kubernetes resource. The Kubernetes API can then be used by users to create, read, edit, and remove instances of this custom resource. Teams can customize Kubernetes with this functionality to fit their unique use cases without modifying the core platform.

Every resource in Kubernetes has a desired state that is specified in its configuration. The controller is in charge of ensuring that the actual state of the resource complies with this requirement. Numerous native resource controllers are included in Kubernetes. The Deployment Controller, for instance, makes sure that the appropriate number of pod replicas are always in operation. In the event of an accidental deletion or crash, the controller will generate a new pod to maintain the desired number of replicas. Users can design **custom controllers** to manage custom resources (specified by CRDs) in addition to the built-in controllers. The **Operator pattern** is the combined usage of CRDs and custom controllers.

2.5 Security

Kubernetes provides a range of APIs, security controls, and policy definition tools that can be employed to manage information security within a cluster. One significant example, previously discussed, is the **NetworkPolicy** API, which governs communication between pods and other network endpoints. NetworkPolicy defines the ingress and egress traffic allowed for a pod, enabling administrators to implement fine-grained network segmentation and reduce the attack surface within the cluster. By default, Kubernetes permits unrestricted pod-to-pod communication; however, NetworkPolicies provide the necessary isolation to prevent lateral movement by attackers or unauthorized services. Despite these specific controls, Kubernetes also requires a general security framework to prevent unauthorized operations on any cluster resource.

2.5.1 Role-Based Access Control (RBAC)

Kubernetes employs a **Role-Based Access Control** (**RBAC**) model to manage permissions and access to cluster resources. This model allows administrators to define roles and assign them to users or service accounts, controlling who can execute actions on specific resources. RBAC is centered around the concept of **Roles**, which specify a set of permissions (such as read, write, or delete) for resources like Pods or Nodes. Additionally, **RoleBindings** link a Role to a user or group, granting them the associated permissions. Roles and RoleBindings are applied to a namespace, meanwhile their counterpart, **ClusterRoles** and **ClusterRoleBindings**, operate at the cluster-wide level.

2.6 Horizontal and Vertical scaling

Kubernetes, as a container orchestration technology, must ensure that applications running within its environment can dynamically adapt to changing resource demands. This is essential for maintaining optimal performance, resource efficiency, and scalability in cloud-native architectures. There are two primary methods by which Kubernetes can adjust to the fluctuating workload: **vertical scaling** and **horizontal scaling**. These approaches serve distinct purposes and provide different mechanisms for resource management based on the nature of the demand changes.

Vertical scaling, also known as "scaling up" or "scaling down," refers to the process of increasing or decreasing the resources (such as CPU or memory) allocated to individual applications or pods. By adjusting the resource limits of running applications, vertical scaling can help optimize performance for resource-hungry applications or reduce resource waste when demand decreases. However, despite the advantages of vertical scaling in certain scenarios, Kubernetes does not provide

built-in support for this functionality. Instead, an external solution is required, which is commonly implemented through the **Vertical Pod Autoscaler (VPA)** Custom Resource Definition (CRD). The VPA enables Kubernetes to dynamically adjust the resource requests and limits of a running pod based on its observed resource usage. By monitoring the resource consumption of pods, the VPA can make recommendations or automatically adjust pod configurations to ensure that applications have access to the necessary resources without overprovisioning.

On the other hand, **horizontal scaling**, or "scaling out" and "scaling in," is the more traditional method provided by Kubernetes to handle changes in workload demand. Horizontal scaling involves increasing or decreasing the number of pod replicas that are running for a specific application, rather than altering the resource allocation for individual pods. Kubernetes provides this functionality through the **Horizontal Pod Autoscaler (HPA)**, which can dynamically scale the number of replicas based on observed load metrics, such as CPU utilization or custom-defined metrics.

The HPA works by monitoring resource usage at the pod level, typically through metrics like CPU or memory utilization, and adjusts the number of replicas based on the defined scaling policies. For example, if an application's CPU usage consistently exceeds a predefined threshold, the HPA will increase the number of pod replicas to distribute the load more evenly across multiple instances. Conversely, if resource utilization falls below the threshold, the HPA will scale down the number of replicas to conserve resources and reduce costs. This process ensures that applications can seamlessly respond to increased demand while maintaining efficient use of available resources during periods of low activity.

However, while Kubernetes provides the HPA for horizontal scaling, it does not inherently include a mechanism for collecting resource usage metrics across all nodes in the cluster. The Kubernetes architecture defines the **Metrics API**, which serves as an interface for exposing resource metrics, such as CPU and memory usage, to the HPA and other components. Despite this, Kubernetes does not come with a built-in tool to gather these metrics from the nodes. Instead, a separate component known as the **metrics-server** must be installed within the cluster. It communicates with the kubelet installed on each node to monitor their resource consumption.

Chapter 3

Multi-tenancy for Kubernetes

Multi-tenancy in Kubernetes refers to the ability to share resources across different users or groups (tenants) within the same Kubernetes environment. By enabling multiple tenants to run their workloads on a shared infrastructure, organizations can reduce operational costs, optimize resource usage, and simplify cluster administration. Hosting multiple clusters or tenants on a single "host" Kubernetes cluster helps to consolidate resources, but this model comes with significant challenges, particularly around **security** and **resource fairness**. Each tenant must be isolated to ensure that no tenant can interfere with the data or workloads of another and fair distribution of cluster resources, such as CPU, memory, and storage, must be ensured. A shared infrastructure must avoid situations where one tenant monopolizes resources at the expense of others (**noisy neighbours**).

Clusters can be hosted in various ways to achieve the multi-tenancy property. One approach involves running separate applications for each user, while the other entails using the same applications with multiple inner instances and dedicating one for each end user. Although Kubernetes does not have first-class concepts of end users or tenants, its extendible nature helps to manage different tenancy requirements. A Kubernetes cluster consists of a **control plane** which runs Kubernetes software, and a **data plane** consisting of worker nodes where tenant workloads are executed as pods. Tenant isolation can be applied in both the control plane and the data plane based on organizational requirements [7].

3.1 Control plane isolation

Kubernetes control plane is made up of its components: api-server, datastore (etcd), scheduler and controller-manager. Their behaviour should not be affected

nor resource associated with this software be editable or seen by other tenants. Control plane isolation ensures that different tenants cannot access or affect each other's Kubernetes API resources.

3.1.1 Dedicated control plane

Kubernetes operations can be performed at either the cluster or namespace level. However, ensuring tenant isolation while allowing them to operate at the cluster level presents a challenge. To avoid this, a common solution is to create dedicated clusters for each tenant, which leads to separate control and data planes.

The most reliable way to fully dedicate a Kubernetes control plane is to install its components on dedicated Virtual Machines. While this approach simplifies setup, it poses considerable maintenance challenges. VM images must be regularly created and updated, and VMs may require migration or additional configuration during runtime. As a result, infrastructure owners must manage an additional layer, including maintaining the bare metal servers, the Kubernetes clusters deployed on top of them, and the VMs for each tenant and their internal cluster (see Figure 3.2). Moreover, dedicating new applications and using VMs translates to additional resource overhead that has to be paid for each tenant. Another concerning factor is that allocating resources to individual users may result in their underutilization, thereby causing a burden to the provider as it is unable to utilize them when they are inactive.

Another subtle drawback lies in the details of a Kubernetes cluster structure. Using VMs as nodes has other implications other than virtualization overhead and duplicating the control plane components. Often, some agents have to run locally on all the nodes and such agents are almost always necessary for a production Kubernetes cluster. Components like the CNI and the CSI implementers are one of these cases. Tenants expect to have a functional cluster, so this software is expected to be already configured. This means that, starting from a bare metal server node and subdividing it into multiple VMs treated as nodes brings more overhead due to node components that are duplicated for nothing because everybody could just use the same solutions, and on top of that it has to be installed by the infrastructure owner during provisioning (see Figure 3.1).

3.1.2 Namespace isolation

A different approach is to restrict tenant resources to a specific namespace, preventing them from interacting with resources belonging to other tenants or at the cluster level through an RBAC model. With this approach, the control plane is shared among tenants, allowing direct access to the cluster but with limited permissions for operations. This is a simple way to avoid the need to allocate separate control

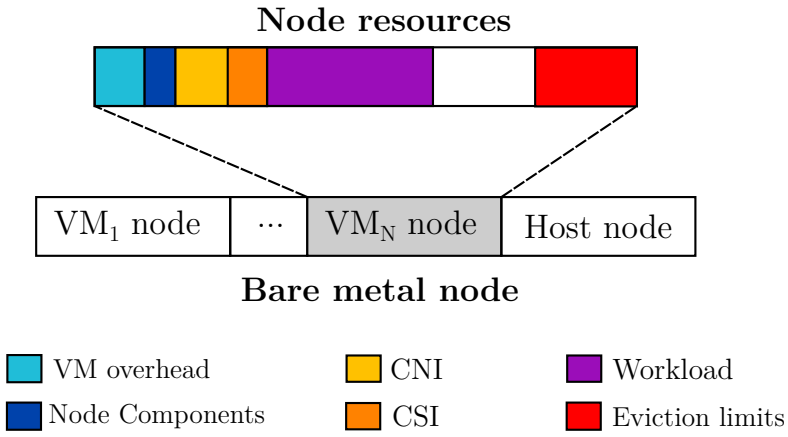

Figure 3.1: Resource usage in multiple Kubernetes clusters running in VMs on a bare metal node. Control plane components are treated as workload resources. When resource consumption exceeds eviction limits, pods are forcefully terminated to prevent overloading, reducing even more the available resources for users.

planes based on Kubernetes APIs. Resource Quotas within namespaces can be utilized to restrict the number of API resources that a tenant can create, as well as the computational and storage resources associated with them. Limiting the number of objects aims to ensure fairness and prevent issues caused by one tenant impacting others who share the same control plane.

Some challenges are still present in real-world scenarios. Tenants can refer to a business with multiple departments or teams. Namespaces can't be subdivided for each user group, so new namespaces should be created and dedicated to them. However, only the service provider is allowed to create new namespaces. Even if the customer accepts this nuisance, it will bring lots of namespaces with very similar policies. In a nutshell, managing hierarchical organization becomes difficult because namespaces do not have a hierarchy and permissions can't be inherited using this structure. Various project exists for hierarchical policies like **HNC** [8] or **Clastix Capsule** [9]. In the end, all these projects are very similar, as they define a CRD to collect multiple namespaces in a hierarchy structure.

Limitations

Using namespaces is the common approach to efficiently share the control plane, but it comes with drawbacks: API requests **unfairness**, **information disclosure** threats even with RBAC API, and **forbidden cluster-level operations** to all tenants.

The API server was not designed to deal with multiple tenants, so multiple concurrent requests could cause **unpredictable performance** issues due to starvation

for some tenants and dominance for others. Only recently, in Kubernetes v1.29 API priority and fairness policies were introduced [10], but they are complex to set up, especially for multi-tenant environments and namespace-based solutions don't actually use it at this time[1]. Even before the official API came out, some works addressed fairness specifically for a multi-tenant environment, however, the focus was on their container workload scheduling [11] and not regulating API requests.

Another challenge is posed by Kubernetes' Role-Based Access Control (RBAC) API given that it is primarily designed to regulate namespaced resources, which introduces limitations when attempting to restrict cluster-wide operations. For instance, namespaces themselves are cluster-level objects. If tenants attempt to list all namespaces, they may inadvertently access the names of other namespaces and infer information about other tenants sharing the same cluster. However, simply disallowing this operation would prevent tenants from listing even their own namespace, which is necessary for retrieving essential information. Restricting certain cluster-level operations, as in the previous case, may not always be feasible. In fact, it can severely limit tenants' ability to manage their own environments effectively. Such restrictions could be detrimental for clients accustomed to using clusters with full administrative capabilities and may undermine the usability of Kubernetes for multi-tenant scenarios.

3.1.3 Virtual Clusters

An intriguing proposal to address the limitations of the namespace isolation model involves extending it by running control plane components within the tenant's namespaces [12]. These dedicated components function as a **virtual control plane**, while the existing cluster control plane continues to operate and manage its resources. This approach overcomes the limitations of namespace isolation by separating the API server as a distinct component, allowing tenants to utilize any cluster-level operation while accessing only the resources exposed by the virtual control plane. Although tenant control planes are isolated, they actually run on a shared cluster referred to as the **host** or **super cluster**. Consequently, it is the host cluster's responsibility to schedule the tenants' pods on its nodes. Collaboration between the tenants' control planes and the host control plane is necessary to transfer the requested workload specifications.

[1]Capsule feature request on API Priority and Fairness usage: `https://github.com/projectcapsule/capsule/issues/180`

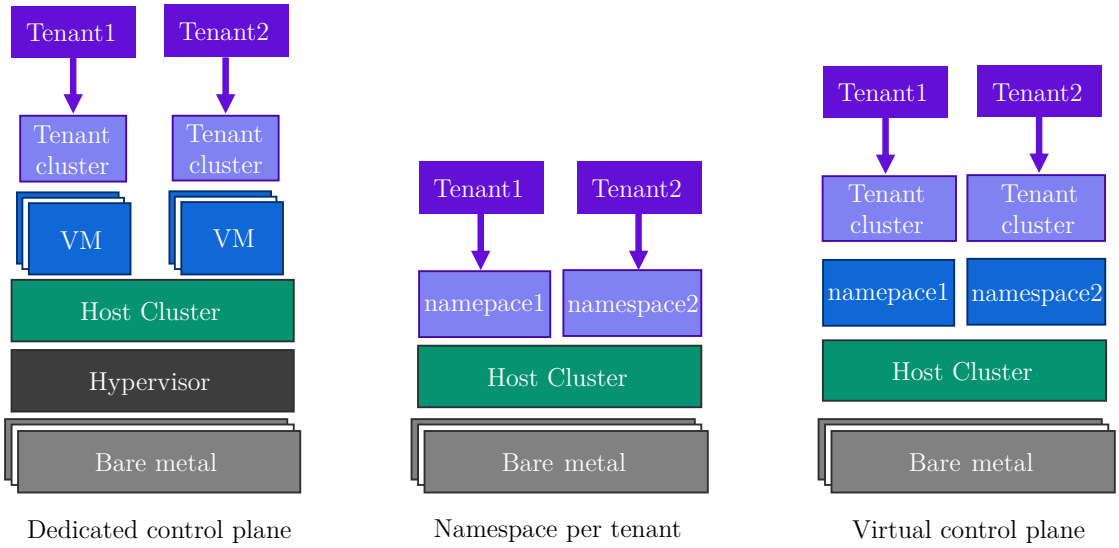

Figure 3.2: All the possible control plane isolation techniques

3.2 Data plane isolation

Data plane isolation is crucial for ensuring sufficient isolation of pods and workloads for different tenants. Various aspects must be taken into consideration to ensure the workload computing resources don't interfere with each other.

3.2.1 Network isolation

In a Kubernetes cluster, all pods are initially permitted to communicate with one another, and network traffic is unencrypted by default. This poses potential security vulnerabilities, as traffic may unintentionally or maliciously reach unintended destinations or be intercepted by a compromised node. Consequently, careful consideration of network isolation is essential to protect against such risks.

Pod-to-pod communication can be restricted through the use of Network Policies. A common approach is to organize each tenant's resources into dedicated namespaces and enforce a default policy that restricts communication to within a tenant's namespace. However, tenants must still be allowed access to the cluster DNS server for service name resolution. This can introduce isolation challenges when the cluster is shared among multiple tenants, as they could potentially query DNS entries for other tenants' services. While they may not have access to the services themselves due to Network Policies, the DNS entries could inadvertently expose sensitive information about other tenants.

It is also important to note that Network Policies must be supported by the chosen Container Networking Interface (CNI) software. Without proper CNI

implementation, there would be no mechanism to enforce these policies at the node level, rendering the policies ineffective.

3.2.2 Storage isolation

It's important to prevent tenants from accessing each other's storage. In Kubernetes, it's discouraged to use node-isolated resources because having them at the cluster level allows for dynamic provisioning based on policies defined by the cluster. StorageClasses enable the description of custom "classes" of storage, which are offered by the cluster based on quality-of-service levels, backup policies, or custom policies set by cluster administrators. To achieve stronger isolation, a separate StorageClass for each tenant can be configured. If a StorageClass is shared, a reclaim policy of "Delete" should be set to prevent the reuse of a PersistentVolume across different namespaces. This means that a dynamically provisioned volume is automatically deleted when a user deletes the corresponding PersistentVolumeClaim. This is the default behavior for a StorageClass, and it's important to avoid changing it to other options that may preserve the data and share it with the wrong tenant.

3.2.3 Pod sandboxing

The pod definition itself presents a challenge in workload isolation because it is a collection of containers. Containers utilize OS-level virtualization and hence offer a weaker isolation boundary than virtual machines that dedicate a separate OS environment. Sandboxing containers provide a way to isolate workloads running in a shared cluster. It typically involves running each pod in a separate execution environment such as a virtual machine or an userspace kernel. While controls such as seccomp, AppArmor, and SELinux can be used to strengthen the security of containers, it is hard to apply a universal set of rules to all workloads running in a shared cluster. Running workloads in a sandbox environment helps to insulate the host from container escapes, where an attacker exploits a vulnerability to gain access to the host system and all the processes/files running on that host.

Sandboxing in VMs

One method for isolating workloads in Kubernetes is to use Virtual Machines (VMs) and allocate an entire cluster to each tenant, with dedicated VMs serving as the cluster's nodes. As previously discussed, this approach inherently addresses the concern of control plane isolation by separating each tenant's environment. However, an alternative approach is to sandbox individual pods directly, rather than isolating the entire node.

Sandboxing in userspace kernel

Another option exists to add an extra security layer to containers and not involve VMs. The reason for looking for an alternative is that in certain case scenarios pod creation time is crucial and VMs creation times (in the order of seconds) or memory footprint are not compatible with it. An application behaving like a kernel can be used as a mediator between tenant apps and the machine's kernel. However, with this approach doesn't leverage hardware virtualization primitives, hence syscalls operations are slower because of the fixed extra jump. The most known implementation of such an approach is Google's project **gVisor** [13]. Syscalls are intercepted and only a subset is allowed, hence some applications may not be runnable and causing a considerable execution time overhead compared to VMs [14].

3.2.4 Node isolation

As an alternative to sandboxing, a **node isolation** technique can be used. With node isolation, a specific set of nodes is dedicated to running pods from a particular tenant, and mixing pods from different tenants is not allowed. This setup helps reduce the issue of noisy tenants, as all pods on a node will belong to a single tenant. The risk of information disclosure is slightly lower with node isolation, as an attacker who manages to escape from a container will only have access to the containers and volumes mounted to that node. Although workloads from different tenants run on different nodes, it's important to note that the kubelet and the API server service are still directly related, unless virtual control planes are used. A skilled attacker could potentially use the permissions assigned to the kubelet or other pods running on the node to move laterally within the cluster and gain access to tenant workloads on other nodes. Node isolation is easier to manage from a billing perspective than sandboxing containers, as you can charge per node rather than per pod. Additionally, it has fewer compatibility and performance issues and may be easier to implement than sandboxing containers. Node isolation can be implemented using **pod node selectors** or a **Virtual Kubelet** [15]. The pod node selector is a constraint defined in the pod specification that explicitly asks for a specific node to be used. An alternative strategy can be to use a mutating webhook to automatically add tolerations and node affinities to pods deployed into tenant namespaces, ensuring that they run on a specific set of nodes designated for that tenant. Virtual kubelet is an implementation of Kubernetes kubelet that masquerades as a kubelet but its APIs are connected to a custom logic. A common usage of virtual kubelet is to use it to create a virtual node that in reality does not execute the workload, but delegates it to another node and in cases of a federated architecture, even for nodes from other clusters [16] [17].

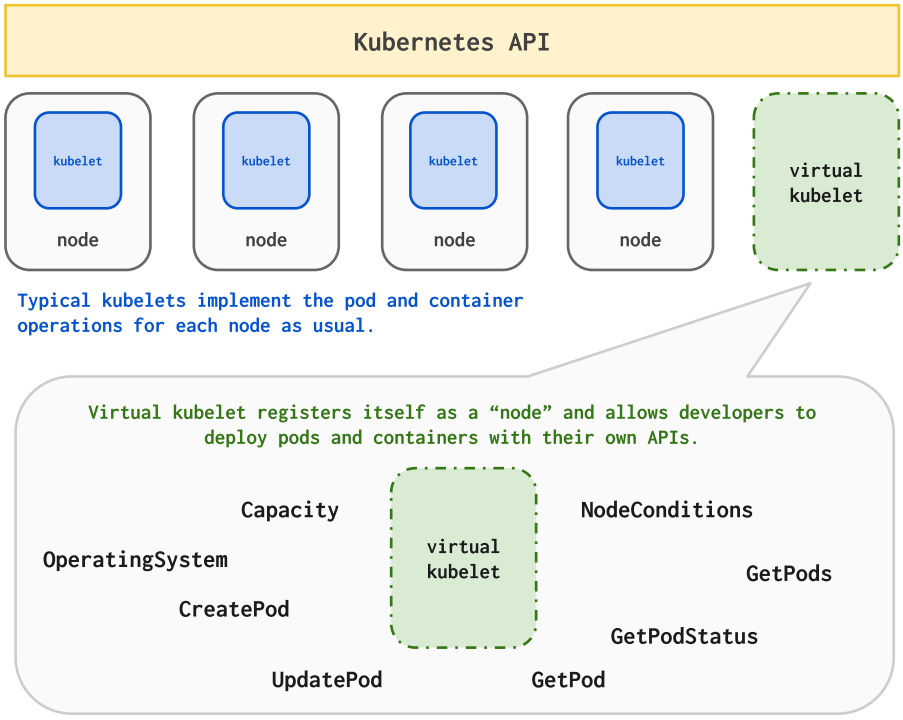

Figure 3.3: Virtual Kubelet architecture. Source: `https://virtual-kubelet.io/docs/architecture/`

Chapter 4

Hard multi-tenancy solutions

Multi-tenancy can be categorized into two types: soft and hard. In soft multi-tenancy, tenants have a certain level of trust between each other, allowing for lighter isolation measures. In contrast, hard multi-tenancy involves zero trust among tenants, necessitating strict and complete isolation. The objective of this work is to identify an efficient model that ensures strong tenant isolation while also considering the infrastructure provider's needs. Therefore, only solutions that support hard multi-tenancy will be explored in detail.

4.1 Dedicated clusters

Hard multi-tenancy can be achieved by provisioning dedicated clusters for each tenant. In this approach, virtual machines (VMs) must be launched and configured to run a Kubernetes cluster, ensuring full isolation. However, when Kubernetes is used as an orchestrator directly installed on bare metal, this method is not natively feasible, as Kubernetes is designed to orchestrate containers, not VMs. To bridge this gap, an external project is required to extend Kubernetes' capabilities, enabling it to treat VMs as computing primitives and orchestrate them alongside containers.

4.1.1 KubeVirt

KubeVirt is an open-source project that extends the functionality of Kubernetes by enabling the orchestration and management of traditional virtual machines (VMs) alongside containerized workloads. It effectively bridges the gap between virtualization and containerization, allowing organizations to run both types of workloads

within a unified Kubernetes environment. It is especially targeted for cases where existing Virtual Machine-based workloads cannot be easily containerized.

Kubevirt introduces VMs as a first-class object in Kubernetes using CRDs. The custom **KubeVirt controller** manages the lifecycles of these VMs resources, similar to how Kubernetes already manages pods. In order to do that, an agent called **virt-handler** needs to be running on each node to apply the controller operations on VMs like their creation and deletion. When an agent is tasked with a VM creation, it launches a new pod called **virt-launcher**. In this pod a container is launched and inside a **QEMU** VM is created using **libvirt**[1] library. KubeVirt is tightly integrated with Kubernetes networking and storage resources since it is just a manner of assigning it to a pod and the agent (the virt-launcher pod) will translate the network using libvirt (see Figure 4.1).

Meanwhile, for the storage if in the VM resource specification a set size volume is requested, the virt-launcher will create a PVC and assign its storage to the VM.

Drawbacks

KubeVirt was initially designed to support legacy applications that required a VM environment within Kubernetes. As a result, QEMU was chosen due to its broad compatibility with legacy systems. However, for cloud-native workloads, most of these legacy features and device support are unnecessary, introducing overhead in terms of boot time and memory usage. MicroVMs have emerged as a more efficient alternative for such workloads, offering reduced memory footprint and faster boot times by stripping away unneeded devices and guest functionalities[18]. Despite this, KubeVirt opted not to focus on MicroVMs, as it would deviate from its core objective of supporting legacy environments. Additionally, KubeVirt's reliance on libvirt and QEMU limits the practical use of MicroVMs across all scenarios. While QEMU did introduce a microVM architecture, it still faces limitations, such as lack of support for hotplugging and PCI devices[19].

4.1.2 VirtInk

VirtInk is a project inspired by KubeVirt and focused on cloud-native workload [20]. Instead of using QEMU, the Cloud Hypervisor VMM [21] was chosen as it is targeted to create MicroVMs with only the minimum set of features to run any type of containerized applications. VirtInk was open-sourced by SmartX, a Chinese company focused on cloud infrastructure services. As for now, the project is still a work in progress and was not updated for a long time since the most recent supported Kubernetes version is v1.25 [20] and its End of Life (EOL) was on

[1] `https://gitlab.com/libvirt`

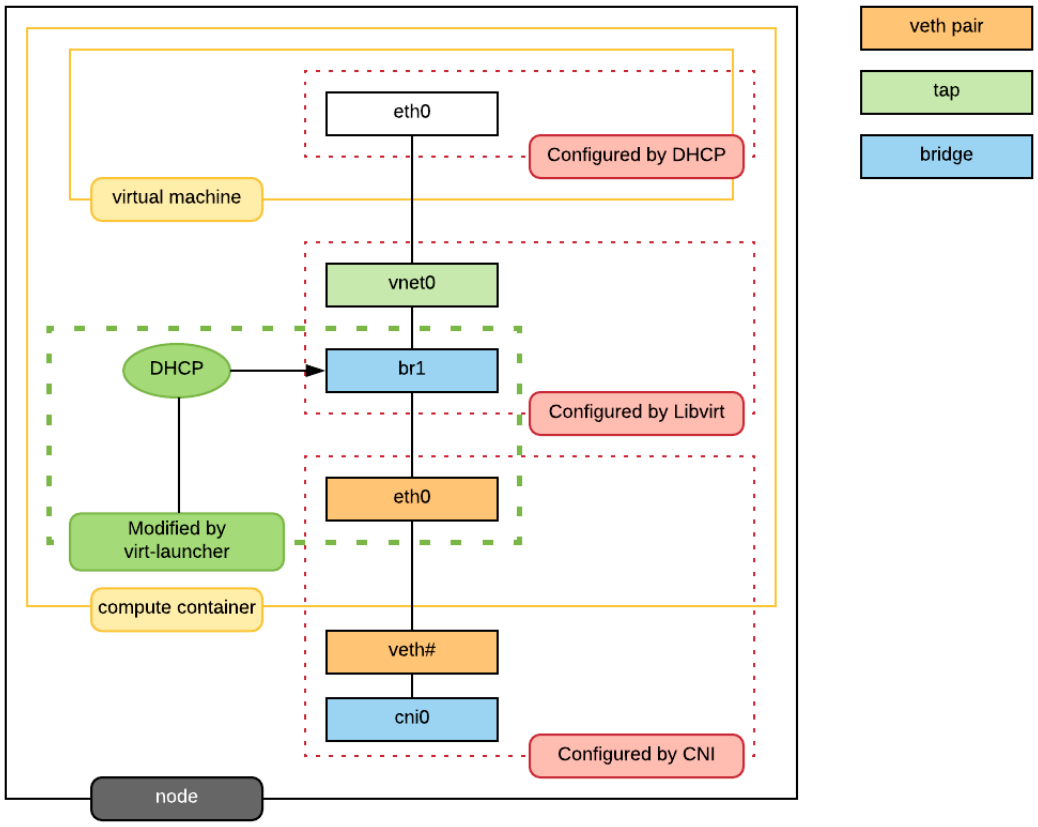

Figure 4.1: KubeVirt strategy to apply networking configurations to VMs. Source: `https://kubevirt.io/2018/KubeVirt-Network-Deep-Dive.html`

October 2023 [22]. Consequently, it is not a viable option for nowadays production environment.

4.1.3 Liquid Metal

Liquid Metal goal is similar to VirtInk, however, it is more flexible because the VMM used is a shim that can target multiple VMM that produces microVM (Cloud Hypervisor and Firecracker) [23]. The idea was to take advantage of different microVM according to the application type, allowing the use of a lighter VM when possible.

It was a project led by Weaveworks, a company that announced its ceasing operations [24], leaving this project in stale with important features yet to be implemented. As an example, it still does not support CNI integration [23].

4.2 Virtual clusters with pod sandboxing

While dedicating entire clusters to individual tenants ensures strong isolation, this approach has significant drawbacks. First, with a large number of tenants, this strategy can become costly. Not only do you incur control plane costs for each cluster, but compute resources cannot be shared across clusters, leading to fragmentation. Some clusters may be underutilized while others are overburdened. Second, managing numerous clusters often necessitates specialized tools, and overseeing hundreds or even thousands of clusters can become overwhelming. Additionally, provisioning a cluster for each tenant is considerably slower compared to simply creating a namespace.

Namespace-based control plane isolation, while efficient, only supports "soft" multi-tenancy, as the control plane is fully shared across tenants. The view on the cluster resources is shared among tenants, even though the operations are limited by permissions it's not possible to forbid every cluster-level operation. On the other hand, virtual clusters create a virtual control plane where only owned resources appear and can be used. From the tenant's point of view, virtual clusters are not so different from separated clusters, and this subdivision disallows interactions between different tenants. At the control plane level, it can be considered a hard multi-tenancy solution, albeit with shallower isolation compared to dedicated clusters. But in this situation, data plane isolation must be addressed using pod sandboxing technology.

4.2.1 vCluster

Alibaba led the virtual cluster idea in collaboration with the Kubernetes multi-tenancy working group. From this concept, Alibaba implemented it under the name of **VirtualCluster** [12] and Kubernetes incubated the project [25]. After its stabilization [26], the project didn't take off and another implementation, called **vCluster**, was introduced by **Loft** [27]. At the moment, VirtualCluster is no longer maintained, while vCluster is actively supported and has evolved to offer greater ease of use and an extensive range of configuration options, accommodating various use cases.

Architecture

To create a new virtual cluster for a tenant, vCluster creates a new namespace and runs Kubernetes control plane components inside. The tenant is allowed access only to its dedicated api-server and to operate in its namespace. If a tenant creates a new resource it is handled just like Kubernetes would; the api-server stores the object in the data store and the controller-manager makes sure to create/delete/edit

the current resources in the datastore. However, the tenant cluster does not own nodes, so the scheduling has to be provided by the host cluster that owns them. A component called the **syncer** is responsible for synchronizing low-level resources like Pods and Services between the virtual and host clusters. Higher-level objects, such as Deployments, remain confined to the virtual cluster. The host cluster's scheduler detects the synchronized Pod definitions and assigns them to its nodes for execution. Although it is theoretically possible to use a tenant's scheduler if the host cluster grants exclusive access to specific nodes [26], this is not a common scenario. Typically, end users do not require stringent scheduling configurations, as a consequence this is an acceptable trade-off.

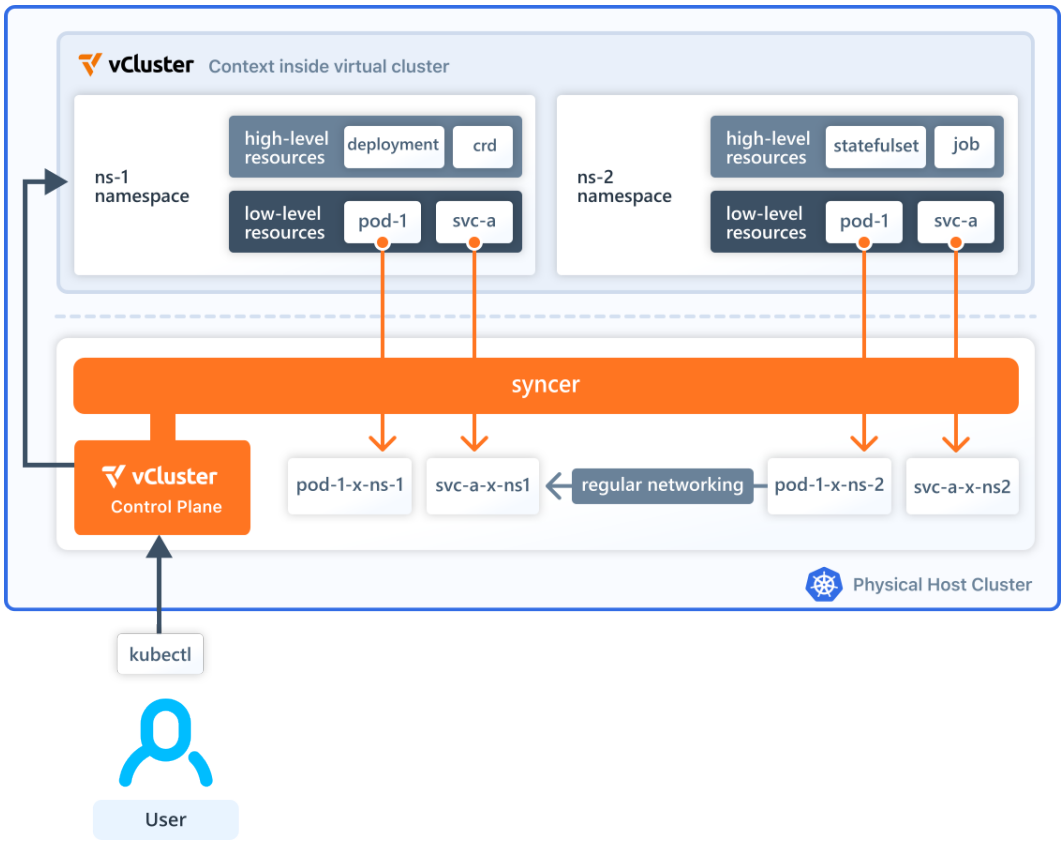

Figure 4.2: vCluster architecture . User requests are forwarded to its cluster components running in pods. Source: `https://vcluster.com/docs/vcluster/introduction/architecture`

Configurability

The syncer component can be configured to allow other resources to be synchronized, even from the host to the tenant cluster. The most relevant ones to share with the tenant clusters are the StorageClass of the host cluster and its nodes. For the latter, multiple options can be chosen [28] from assigning a set of real nodes to fake nodes. Giving nodes to a virtual cluster allows the usage of resources like DaemonSets that are required because of use cases where a component must be installed on any used node. Furthermore, to scale pods horizontally, a metrics server has to aggregate all the nodes' resource consumption. It can either be installed inside the tenant cluster with its assigned nodes or the one inside the host cluster can be reused, hence neither requiring nodes to be assigned locally nor another component to be run for the users.

Lastly, vCluster also supports many Kubernetes distributions (or distros) like k8s, k3s[2], and k0s[3], as well as different datastores such as etcd, MySQL, MariaDB, PostgreSQL, and SQLite. A distribution (distro) refers to a packaged version of the Kubernetes platform, tailored with specific configurations, components, or optimizations to suit different use cases and environments. Kubernetes, while highly flexible and powerful, can be resource-intensive for some scenarios. Therefore, lighter-weight versions like k3s or k0s were developed to cater to edge computing, development environments, or smaller resource-constrained setups. These distros allow users to deploy Kubernetes in various scenarios, from large-scale production environments (k8s) to low-resource IoT devices (k3s).

vCluster Pro

Loft offers a paid version of vCluster that includes additional utilities and features designed for large enterprises. One of the standout features is the "Isolated Control Plane", which essentially functions as a Hosted Control Plane. This feature allows control plane components to be hosted on a separate cluster, providing enhanced isolation and flexibility. The potential use cases and rationale behind this plan, aimed at enterprise-level organizations, will be discussed in more detail in the following chapters (5.3)

4.2.2 Kata Container

Kata Containers is an open-source project designed to enhance container security by utilizing lightweight virtual machines (microVMs) to isolate containerized workloads

[2] `https://k3s.io/`

[3] `https://k0sproject.io/`

through hardware virtualization technology [29]. While traditional containers share the same host kernel, making them lightweight and fast, this also compromises isolation compared to virtual machines, which run on separate kernels. Kata Containers address these risks by running each container within its own lightweight virtual machine, effectively isolating the host kernel from the container's kernel. This approach significantly enhances security, ensuring that a compromised container cannot impact the host or other containers, while maintaining strong performance through the use of hardware virtualization primitives.

Architecture

Kata Containers wants to achieve its goal transparently, without requiring special attention from the end-user. This was achieved by implementing a CRI-compliant component (**Kata Shim**) that doesn't directly manage the container's lifecycle on the host machine. When a container creation is requested, the Kata Shim boots a VM with an **agent** using the provided hypervisor. Afterwards, the shim requests the container creation via the agent to the guest kernel. When new container operations are issued, Kata Shim will just keep forwarding them to the agent and are treated regularly by the host kernel [29] (see Figure 4.3).

CPU isolation

In virtualization technologies, CPU isolation poses a significant challenge. Virtual machines (VMs) demand CPU resources that have to be provided to reach optimal performance and prevent interference between workloads. This necessitates a strategy to map the virtual CPUs (vCPUs) of a VM to physical CPUs. A common approach is to reserve a **CPU set** for each VM, which is the default behaviour in KubeVirt. However, this method can lead to underutilization of resources, as any unused CPU remains idle and unavailable for other tasks for the provider. Alternatively, some strategies allow multiple vCPUs to be mapped to the same physical CPUs. For example, an absolute fixed amount of CPU time known as **CPU quota** can be assigned to each VM's CPU, causing the VM to be suspended if it exceeds its allocated quota. Conversely, a relative quantity called **CPU share** can be used to dynamically assign a time slot for each task proportional to the share.

Kata Containers uses by default a hierarchical CPU sharing, configured to ensure that all the VMs (sandboxed pods) have an equal sharing and each internal container uses equally the pod share. This is implemented using a hierarchy of cgroups [30].

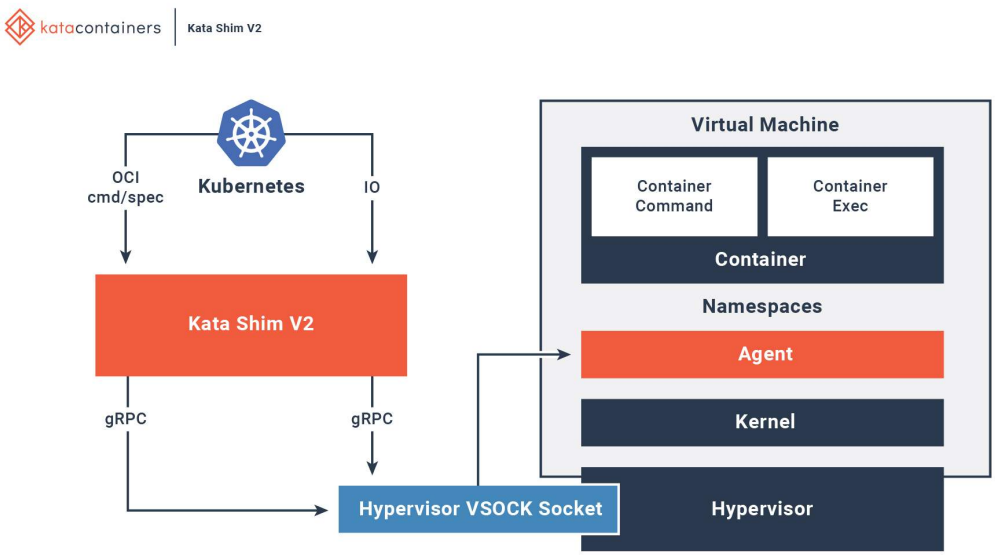

Figure 4.3: Kata containers architecture. The shim handles the virtual machine (VM) booting by communicating with its agent using a gRPC-based protocol over a VSOCK socket, which is designed for VM-host communication. The gRPC protocol allows the runtime to send container management commands to the agent and manage standard I/O streams (stdout, stderr, stdin) between containers and container management systems like CRI-O or containerd.. Source: `https://katacontainers.io/learn/`

Storage

There are multiple selectable strategies for sharing storage between the host and the VMs. If a block-based driver is configured, **virtio-scsi** is used to share the workload image into the container's environment inside the VM. Conversely, if a block-based graph driver is not configured, a **virtio-fs** overlay filesystem mount point is used to share the workload image instead. The agent uses this mount point as the root filesystem for the container processes. An alternative approach involves using the **devicemapper snapshotter**, which operates on dedicated block devices rather than formatted filesystems and functions at the block level instead of the file level. This configuration allows for direct use of the underlying block device as the container's root filesystem. This approach gives much better I/O performance compared to using virtio-fs to share the container file system.

A relevant feature in this context is that Kata Containers support hot plugging/unplugging also for block devices. This means that it's possible to use block devices for containers started after the VM has been launched.

Networking

As for networking, Kata Containers have to translate the CNI implementer configuration and proxy the traffic from the VM to the container. When setting up a container network, the network plugin will usually create a virtual ethernet (veth) pair adding one end of the veth pair into the container networking namespace, while the other end of the veth pair is added to the host networking namespace. However, usually hypervisors cannot handle veth interfaces and typically use TAP [31] interfaces instead. To overcome incompatibility between typical container engines expectations and virtual machines, Kata Containers networking transparently connects veth interfaces with TAP ones and redirect the traffic using Traffic Control rules.

Chapter 5

Multi-cluster management plane

Managing multiple clusters originating from a single host cluster presents a considerable maintenance challenge. Starting with a single cluster often leads to managing numerous tenant clusters. Additionally, many infrastructures consist of multiple bare-metal clusters, each hosting several tenant clusters, which further increases the complexity of management. This chapter explores various infrastructure architectures designed to tackle these challenges by leveraging the capabilities of the ClusterAPI project.

5.1 ClusterAPI (CAPI)

ClusterAPI (CAPI) is an open-source Kubernetes project aimed at simplifying the provisioning, upgrading, and management of multiple Kubernetes clusters through declarative APIs and tooling [32]. Kubernetes, as a complex system, requires the proper configuration of numerous components to form a functional cluster. Today, there are over 100 Kubernetes distributions and installers, each with different default configurations and support for various infrastructure providers [33]. Kubeadm was designed as a common tool for bootstrapping a Kubernetes cluster that any installer could have used [34]. However, while it simplified installation, it did not address the ongoing management of clusters. ClusterAPI's goal is to manage the entire lifecycle of Kubernetes clusters—from creation and scaling to deletion—using Kubernetes' declarative API on any infrastructure [35].

5.1.1 Main concepts

CAPI define a cluster in an abstract way using multiple CRDs [36]. The most relevant are:

- **Cluster**: logically identifies a cluster owning a set of machines on a certain infrastructure;
- **Machine**: infrastructure component hosting a Kubernetes Node;
- **BootstrapData**: initialization data (usually cloud-init) used to bootstrap a Machine into a Node;

These abstract resources have to be mapped to an actual cluster by a piece of software according to an interface, referred to as a **contract** [37]. The software implementing the contracts defined by ClusterAPI are called **providers**:

- **Infrastructure provider**: responsible for the provisioning of infrastructure/-computational resources required by the Cluster or by Machines (e.g. VMs, networking, etc.).
- **Bootstrap provider**: responsible for turning a server into a Kubernetes node, initializing the control plane and joining it with the worker nodes.
- **Control plane provider**: responsible for managing a set of machines that represent a Kubernetes control plane, providing information about its state of downstream consumers and managing secrets with kubeconfig file for accessing the administered cluster.

KubeVirt and vCluster have both implemented a CAPI Infrastructure Provider. For an overall view of ClusterAPI architecture, see Figure 5.1

5.2 Remote Management Plane

A cluster provisioning system for third-party users is handled with three cluster classes:

- Management Cluster: Configure the infrastructure to provide the requested resources to tenants;
- Workload or Provider Cluster: Owns the resources to share with the users. Converts the configuration from the management cluster to exclusive access resources for a tenant;
- Tenant Cluster: The cluster consuming the workload cluster resources and given to the tenants.

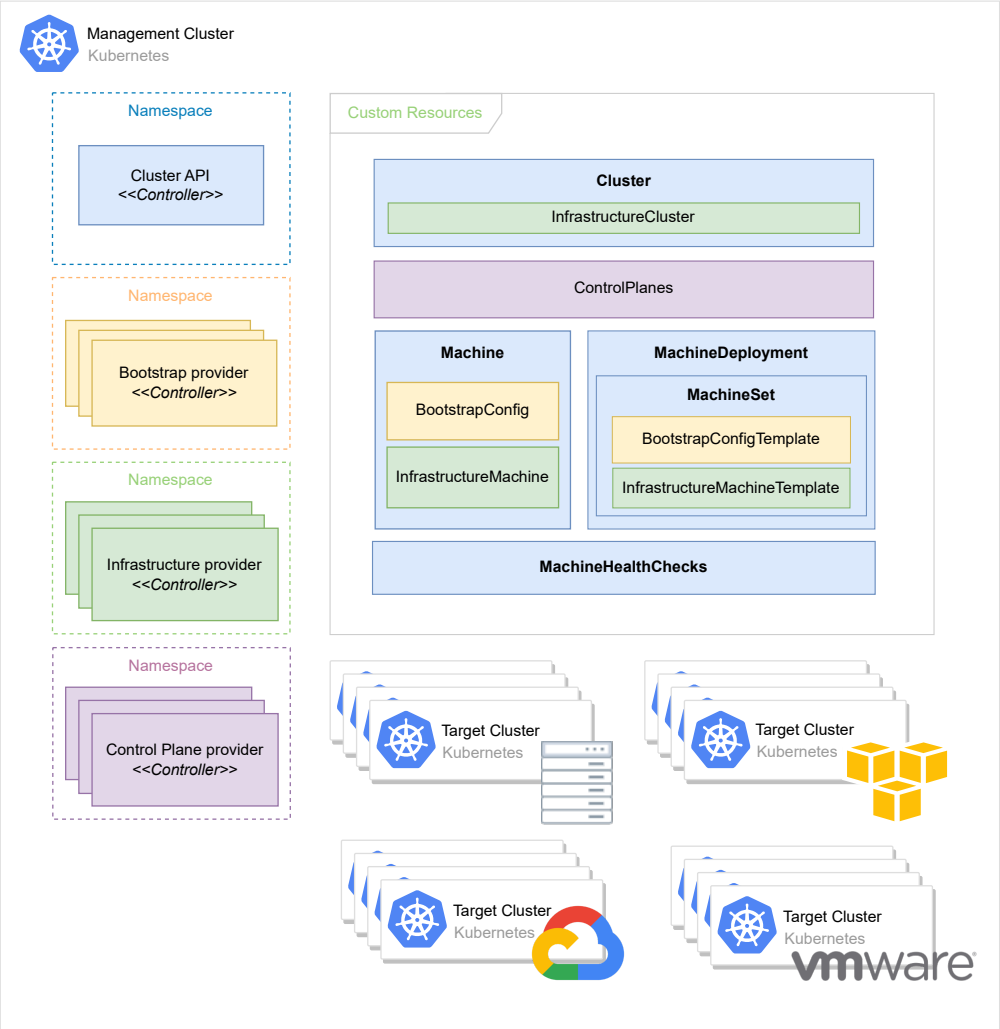

Figure 5.1: ClusterAPI architecture[36]

ClusterAPI (CAPI) offers a promising approach for managing multiple tenant clusters on top of a single workload cluster, effectively functioning also as a local management cluster for that workload cluster. However, most infrastructures consist of multiple workload clusters, each requiring its own management setup. Installing CAPI on every workload cluster and manually accessing each one to manage tenant clusters is a cumbersome and inefficient process, posing challenges for maintaining a cluster provisioning service. One potential solution is to use a dedicated, external management cluster to centralize this process. This management cluster would track tenant cluster locations, monitor workload cluster statuses, and push desired configurations—such as assigning dedicated vCPUs and memory—to the appropriate clusters. By consolidating these management tasks into a single

location, the service provider's workflow is greatly simplified. Figure 5.2 provides a high-level overview of the proposed architecture and the typical workflow it supports.

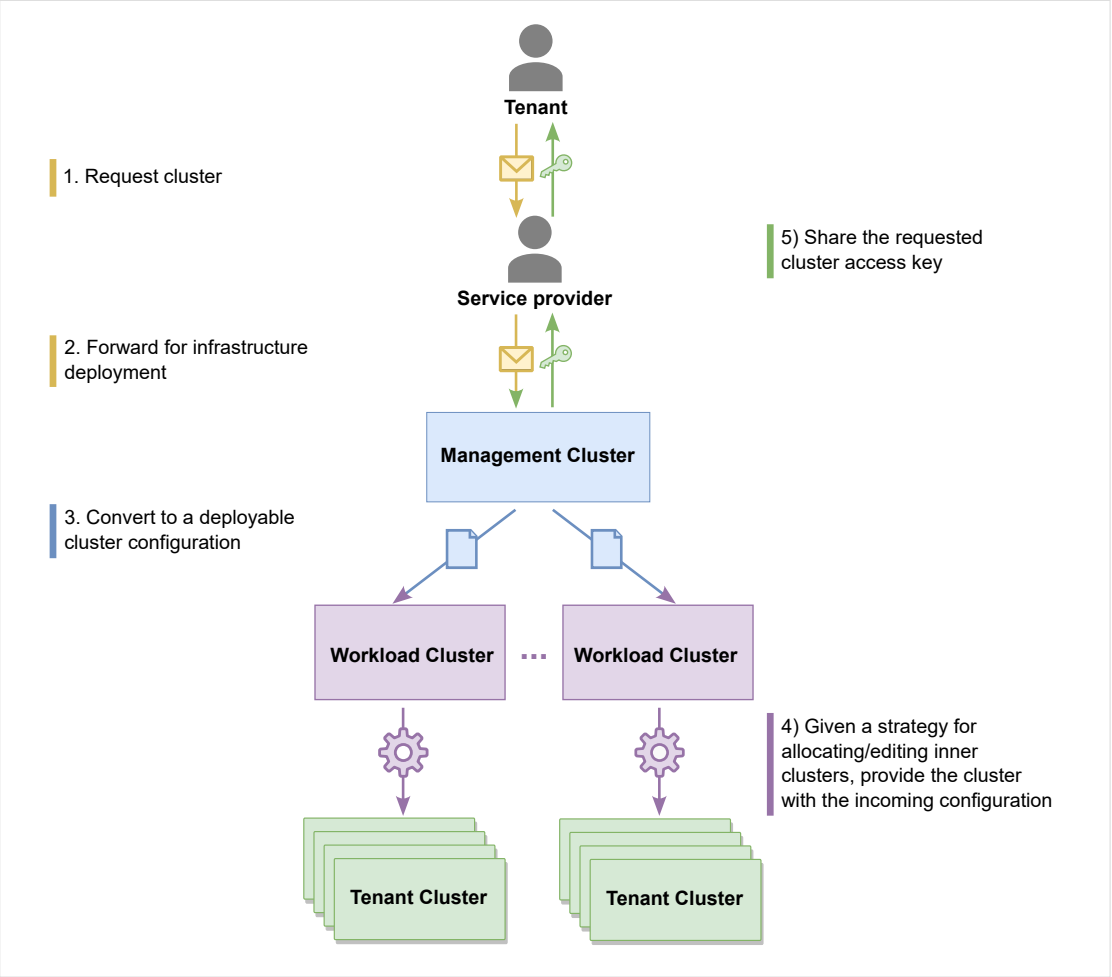

Figure 5.2: Cluster provisioning service conceptual architecture. Actual systems may vary in complexity, with management and workload roles sometimes combined or overlapping. Workload clusters may also require a dedicated local management plane for tenant clusters.

Since ClusterAPI is generic, it doesn't disallow this paradigm, however, it must be implemented by its providers. Kubernetes control plane has an optional component that can be leveraged for this use case: the **Cloud Controller Manager** (**CCM**). It allows for a cloud provider to extend the capabilities of a control plane with custom logic [38]. A way to use it could be to create a load balancer and link it to all the infrastructure clusters. This is the approach used by KubeVirt. On the

other side, vCluster doesn't implement a similar mechanism, so as a consequence, a tenant cluster lifecycle can only be managed from the host cluster. Though it's yet to be implemented, it is a planned feature. As for now, each workload cluster should have installed CAPI as a local management plane.

Although with unsupported providers it is required to manage separately each workload cluster, it's possible to automatize this process. The remote management cluster can consist of the necessary components to implement a GitOps workflow. The workload cluster will continuously deploy the defined tenant clusters configuration according to a Git repository as a single source of trust. Having installed CAPI on each workload cluster it is possible to implement this strategy with an addon that will setup and install the agent responsible for pulling deployments and apply them [39].

5.3 Hosted Control Plane

Up to this point, a tenant cluster is completely hosted on a specific bare-metal workload cluster. Using resources from multiple workload clusters was not conceived. This choice leads us to a single point of failure in the proposed service architecture. In case a workload cluster becomes unavailable, the tenants will perceive a downtime. In most cases, especially for telcos, this is accepted because their infrastructure nodes are considered enough reliable for their intended use or have a limited number of them, so a more scalable and reliable system is not a simple ordeal. The core issue lies in the high cost of allocating resources across multiple zones, as it requires maintaining a redundant and extensive infrastructure. While this model is feasible for large-scale cloud data centers, it becomes difficult to implement in smaller environments. However, in scenarios where a significant number of workload clusters are owned and reliability is desired, it must be addressed by replicating and distributing resources across different clusters.

Managing clusters in this scenario becomes difficult because even by leveraging a management cluster, it would require additional complexity on the workload clusters that need to be handled to form a sort of cluster mesh. The main challenge would be having a tenant control plane hosted and replicated on multiple workload clusters to preserve its availability, but also transparently redirect its resources to different clusters where they are replicated. It is possible to overcome this challenge by changing how we host tenant clusters. A relevant approach is to consolidate all the tenants control planes in a set of datacenters acting as the management cluster, to have a single point where to manage them without the hassle of tracking them. Furthermore, the workload clusters will just offer themselves as a place to host data plane resources. In case a workload cluster becomes unavailable, from the point of view of tenant control planes, some pods had failures and must be

recreated. This process can use any other workload cluster as a destination for new replicas according to their load. This approach is the one adopted by hyperscalers and is known as **Hosted Control Plane**.

5.3.1 Clastix Kamaji

At the moment, Kamaji is the only open-source project implementing the Hosted Control Plane for Kubernetes. Kamaji turns any Kubernetes cluster into a "Management Cluster" to orchestrate other Kubernetes clusters called "Tenant Clusters" [40]. Kamaji is special because the Control Plane components are running inside pods instead of dedicated machines. This solution makes running multiple Control Planes cheaper and easier to deploy and operate. The workload clusters are just used as a collection of resources that can be used for the tenants' data plane. Consolidating control planes in one cluster makes it easier to configure them in case of many tenant clusters dispersed in hundreds or more workload clusters. At a scale of hundreds of clusters, balancing the load becomes an important factor for efficiency.

CAPI integration

Kamaji is not an all-in-one solution for multi-tenancy and managing a cloud infrastructure. Its only objective is to create a Management Cluster and connect them to Tenant Clusters, but their lifecycle and the Tenant Clusters creation have to be handled externally. This technology does not replace CAPI but they actually complement each other. Kamaji has implemented a CAPI Control Plane Provider only responsible for creating and managing the control plane for each tenant. An independent Infrastructure Provider has to be used.

Kamaji isolates the control plane by instancing duplicated components, similarly to vCluster (section 4.2.1). However, the data plane has to be isolated in other ways. A common approach is to place each control and data plane in dedicated VMs. This inevitably results in significant resource overhead, but in this case, it can be compensated for by making it simpler to run several clusters that would otherwise be challenging to utilize to their full potential. At the moment, the only CAPI Infrastructure Provider supported for isolating data planes is based on VM nodes (e.g. KubeVirt). Another notable usage of Kamaji is for **hybrid cloud models** where it is placed in a cluster provided by a reliable cloud company and then on-premise data centres are used as workload clusters.

Chapter 6

Evaluation

This chapter's goal is to have an overview of all the possible solutions for applying multi-tenancy and then specialize the inspection for the case of hard multi-tenancy models optimal for small and medium organizations with production-ready solutions.

6.1 Solution classification

To achieve a comprehensive understanding of all possible solutions within a multi-tenancy environment, it is crucial to introduce a more formal classification. We'll focus on defining multitenancy approaches, tenant autonomy levels, isolation levels, and production readiness.

6.1.1 Multitenancy approach

In general, multi-tenancy systems can be classified into two broad approaches: **multi-instance** and **single-instance native** architectures. According to previous studies [41] [42], these categories provide a foundational framework for distinguishing between different ways of implementing multi-tenancy.

Multi and single instance multitenancy

In the **multi-instance** approach, multi-tenancy is realized by deploying multiple, separate instances of the same software or service for each tenant. This approach offers isolation at the instance level, which can increase fault tolerance and security, but at the cost of higher resource usage and more complex maintenance. Each tenant receives their own dedicated environment, which simplifies customization and allows different versions or configurations to be applied independently.

On the other hand, the **single-instance native** approach refers to a more resource-efficient model in which a single software instance is shared among multiple tenants. In this case, the software is inherently designed to support multi-tenancy, managing multiple tenants' data, configurations, and workloads in a shared environment. This approach can be more efficient, as fewer resources are needed to manage the system overall, but it introduces complexities related to security, data isolation, and tenant-specific customization, all of which must be managed carefully within a single instance.

Multi-instance multitenancy strategies

However, these traditional categories often fail to capture the full complexity of multi-tenancy in modern cloud-native environments, particularly in container orchestration systems like Kubernetes. Kubernetes, with its unique architecture and deployment models, offers several strategies for implementing multi-tenancy, some of which do not fit neatly into the multi-instance or single-instance native paradigms. Recognizing this limitation, [43] introduced further subdivisions within the multi-instance category, particularly when applied to Kubernetes-based systems. These subdivisions include **multi-instance through multiple clusters** and **multi-instance through multiple control planes**.

The *multi-instance through multiple clusters* approach involves deploying separate Kubernetes clusters for each tenant. This offers strong isolation between tenants, as each one operates within an entirely independent cluster. While this provides a high level of fault tolerance and security, the overhead associated with managing multiple clusters can be significant. Each cluster requires its own control plane, set of nodes, and maintenance, which can lead to increased operational costs, particularly as the number of tenants grows. In contrast, the *multi-instance through multiple control planes* approach operates with a shared infrastructure at the node level, but each tenant is provided with their own control plane. This allows for more efficient resource utilization compared to the full separation of clusters, while still providing a degree of isolation at the control plane level. Tenants can be isolated in terms of their management capabilities, such as creating and managing resources, without requiring fully separate clusters. However, this strategy can complicate control plane management and increase the complexity of resource scheduling and security policies across tenants.

In addition to these approaches, there exists another unique solution within the Kubernetes ecosystem that merits its own category: **multi-tenancy through Virtual Kubelet**. Virtual Kubelet introduces a mechanism for managing multi-tenancy by acting as a proxy between Kubernetes clusters. It masquerades as a node's Kubelet, which is the Kubernetes agent responsible for node management and pod lifecycle operations. However, instead of managing a local node, the

Virtual Kubelet forwards workloads from the original cluster to other clusters that may belong to different tenants. This solution differs fundamentally from the multi-instance models, as it introduces a distributed workload management system. In this model, a tenant's workloads are placed on other tenants' clusters, thus they have to manage the possibility that some nodes may run workloads coming from multiple tenants. The responsibility for managing shared workloads is distributed among clusters, which adds complexity in terms of workload scheduling, resource allocation, and tenant isolation. This strategy creates a new form of multi-tenancy, which, due to its unique operational characteristics, is classified separately as *multitenancy through Virtual Kubelet* [43].

6.1.2 Tenant autonomy levels

A multi-tenancy framework is designed to enable multiple tenants to use the same system while restricting their access and operations to ensure isolation and security. In such a setup, tenants are provided with limited visibility and control over the system, often confined to specific views or segments, which ensures that each tenant can operate independently without interfering with others. However, this also means that tenants may have limited autonomy within the shared system, often relying on the service provider to handle certain operations that fall outside their permitted scope. This structure strikes a balance between resource sharing and operational isolation, which is critical for ensuring both efficiency and security in multi-tenant environments.

In the context of Kubernetes, various multi-tenancy frameworks provide tenants with different levels of autonomy, depending on how much control and access they are allowed over the system. These levels of autonomy can be categorized into three primary tiers:

- **Cluster-level**: tenants are granted full control over the entire Kubernetes cluster. They have the autonomy to perform any cluster-wide operations, including managing nodes, creating and modifying control plane components, and configuring cluster-wide policies. Cluster-level autonomy offers tenants the highest degree of operational independence, as they can manage the full lifecycle of their applications and resources without restrictions imposed by other tenants or third parties.

- **Virtual Cluster-level**: a subset of cluster-level operations can be used, allowing tenants to manage resources as though they were in control of their own standalone clusters. However, some operations may not be fully autonomous, as certain components of the virtual cluster are controlled by the underlying infrastructure provider. For example, tenants may be able to manage pods, services, and configurations within their virtual cluster, but they

may not have control over certain networking components, the underlying node infrastructure and the scheduler component. The host cluster operates the virtual control plane, which may transparently refuse the application of certain requested configurations related to these restricted elements. Consequently, tenants experience a diminished degree of autonomy compared to what is available at the broader, cluster-wide level.

- **Namespace-level**: tenants are restricted to operating within specific Kubernetes namespaces. This level of autonomy is the most restrictive but is also the most commonly used by companies because they buy a less-strict cluster from cloud providers and then adopt an additional multi-tenancy framework for isolating their working groups.

The maximum level of tenant autonomy is directly related to the multitenancy approach chosen. Different approaches offer varying levels of autonomy for tenants:

- **Multi-instance through clusters** offers the highest level of autonomy, where tenants have control at the Cluster-level.
- **Multi-instance through control planes** allows tenants to operate within a Virtual Cluster-level.
- **Single-instance native multitenancy** offers the lowest autonomy, restricting tenants to Namespace-level operations.

6.1.3 Isolation level

Formally defining a system as completely secure is very difficult. Serious and dedicated studies should be done to build a threat model for each solution stack of interest to understand the harm one vulnerability could lead to if an attack takes advantage of it. Furthermore, a regulated process has to be adopted to periodically investigate and patch suddenly found vulnerabilities. This thesis doesn't focus on this aspect but requires some metrics, to define a level of isolation for user resources in a multi-tenant environment. Given that it's not feasible to quickly define a metric to state a resource as completely isolated, we'll fall back use some metrics in a relative manner to be able to define the most secure among the treated solutions.

The data plane isolation level will focus on workload isolation. This can be translated just as a comparison between container and virtual machine isolation. This subject was extensively treated in the literature [44] [45] and one simple value to explicit the difference is layers of indirections between the workload and the host machine. Regular container runtimes have just one such layer because they only have the host kernel primitives that separate containers from the host kernel. Using sandboxed container runtimes like Kata Containers or generally using a VM

adds an extra layer of indirection because of the extra dedicated kernel. For what concerns the data plane, using a VM is safer than a container, hence we'll say the data plane has a high isolation level, and conversely, we'll consider it as low.

For the control plane isolation, it's possible to use the same concept by counting the number of security layers available for the system. Namespace-based isolation has only one security layer granted by a permission system. For virtual clusters, an additional level is given by the virtual control plane. As for the approach of dedicated cluster, the control plane is usually separated by using exclusively dedicated VMs, so there are the two level of security of the VM.

The solutions based on namespaces are the most insecure, but they also can't properly isolate the control plane resources according to the Kubernetes API3.1.2. Consequently, this option is adequate only for a soft multitenancy so we'll label the isolation level as **soft**. Meanwhile, virtual cluster can isolate all the Kubernetes APIs and offers a higher number of security layers, so it's proper to call it a hard multitenancy solution and we'll classify its isolation level as **hard**. However, for the case of implementing multi-tenancy through multiple clusters, using this metric of comparison could lead to the wrong approximating result of having equal isolation compared to virtual clusters. In reality, this is not true because in the end virtual clusters have an higher attack surface, since their data plane can be threatened in the same way, but the control plane can be another source of attack that has to be defended. To account for the difference in these two cases, we'll label the isolation of using multi-instance through clusters as **"hard+"**.

6.1.4 Production readiness

To effectively implement a tool in real-world scenarios, both technical and non-technical factors must be carefully considered. This complexity makes it challenging to classify software as truly production-ready.

From a technical perspective, the maturity of a project is primarily determined by its code stability, performance, thorough documentation, and security. Code stability is evaluated through the presence of robust test coverage, adherence to versioning milestones (such as beta releases, release candidates, and stable final releases), and demonstrated successful use cases in practical applications. To ensure sufficient performance and scalability, the software must be rigorously and continuously tested under various conditions, confirming that it delivers consistent and reliable results. Additionally, up-to-date and comprehensive documentation is essential. This should include installation instructions, detailed API references, user manuals, and clear contribution guidelines to support both users and developers. Security, another critical technical factor, must be actively managed. This involves regular vulnerability scanning and a well-defined strategy for rapidly addressing any security patches or issues that arise.

In addition to these technical elements, non-technical considerations are equally important, particularly in the context of open-source projects. A transparent governance model is essential to ensure that the project is well-managed and sustainable over time. This includes encouraging an active and engaged community of developers, users, and contributors. Furthermore, the project's licensing must be clearly defined and unambiguous, adhering to legal standards that permit freedom of use, modification, and redistribution. A well-defined licence minimizes legal uncertainty, thereby encouraging adoption by companies for business purposes. Without clear licensing, potential users may be reluctant to integrate the software into their operations, hence losing important potential community members.

All the technologies considered in this work are open-source and hosted on GitHub, making it necessary to establish clear indicators of their usability in real-world scenarios. To simplify the assessment, we will classify projects as production-ready based on some shallow criteria. First, the project must have a clear governance model or backing by a reputable company, which provides stability and long-term viability. Additionally, we will consider the community growth surrounding the project by tracking the trend of GitHub stars. A positive trend in stars serves as an approximation of an active and growing user and developer community, which is a key factor in the ongoing support and improvement of the project. Finally, to confirm that the project is actively maintained, it must have at least one commit pushed to the main branch within the past month. This serves as a reliable indicator that the codebase is up-to-date and that the project is still under active development, reducing the risk of using outdated or unsupported technologies in production environments.

6.2 Solutions comparison

All the solutions analysed in this work have been classified, similar to other works [43] but using a whole technology stack rather than focusing on control plane isolation (see Figure 6.1). Solutions based on Virtual Kubelet were not taken into account due to their multitenant model diverging from the main topic of this work, dwelling in the domains of federated clusters. We'll proceed with the assumption that the service provider owns all of his clusters and their nodes. Furthermore, we take in mind our initial objective of picking a solution with reduced operating costs, compatible with smaller organizations' resources and that can be used at this moment.

It's important to notice all the possible ways the tenant resources can be handled. All the purely namespace-based solutions (e.g. HNC and Capsule) are primarily considered to be used on a single cluster, so the administrators just need to tweak a configuration locally. Meanwhile, other solutions may span across multiple clusters

and a management cluster is used to distribute and check all the other clusters. The picked solution has to be of this type to build a professional service to provision a cluster for external tenants.

Given that the single-instance native solutions are used for soft multi-tenancy, no data plane isolation technologies are usually applied because it's not required and the isolation level according to our definition wouldn't improve anyway. While it is technically feasible to introduce data plane isolation using Kata Containers, it was not included in the comparison mentioned in the previous statement (see Figure 6.1).

The desired solutions should have hard multitenancy and be production-ready. Based on this requirement, three frameworks could be considered: KubeVirt, vCluster with Kata Containers, or Kamaji with KubeVirt. The hosted control plane model adds extra complexity and potentially unclear benefits in small-scale infrastructures. While this approach allows for load balancing among clusters, providers will have to invest additional effort in setting it up. At a small scale, the operating cost for replication as a way to add reliability is not sustainable. Moreover, the hosted control plane does not replace a GitOps infrastructure, as it is a crucial component in modern software deployments. As a result, infrastructure providers in this case may prefer to stick to their existing workflows instead of adding features they may not need for their use case. Conseguently, the potential strategies boil down to either using KubeVirt or vCluster with Kata Containers. We will compare the two solutions to determine the potential benefits of transitioning to a shared host cluster instead of the traditional approach of dedicated clusters. This comparison aims to quantify the additional overhead incurred by choosing the traditional way of implementing multitenancy through multiple clusters.

6.3 Benchmark and Measurements

Two experiments were conducted to determine tenant cluster provisioning times and resource overhead during workload. Every test was conducted locally on a workload cluster, with tenant clusters being managed via ClusterAPI. Kata Containers was tested both with QEMU and Cloud Hypervisor, but both resulted in an almost equal performance with their default configuration for this scenario.

The tests used KubeVirt v1.3.0 and vCluster v0.2.0 running on Kubernetes v1.29 with two bare-metal servers equipped with the following specs:

- **CPU**: 2x Intel(R) Xeon(R) Gold 6252N @ 2.30GHz (24 Cores, 48 Threads)
- **RAM**: 2x 192GiB DIMM DDR4 buffered @ 3200 MHz
- **NIC**: Intel Gigabit 4P X710/I350 rNDC 2 port @ 10 Gbps

Figure 6.1: Multitenancy framework comparison table

	HNC	Capsule	KubeVirt	VirtInk	Liquid Metal	vCluster + Kata Containers	Kamaji + KubeVirt
Multitenancy model							
Single-instance native	•	•					
Multi-instance							
- Multiple clusters			•	•	•		•
- Multiple control planes						•	
Tenant autonomy							
Namespace-level	•	•					
Cluster-level			•	•	•		•
Virtual cluster-level						•	
Cluster management							
Single host cluster							
- Operator or CLI tool	•	•				•	
- CAPI			•	•	•	•	•
Multiple host clusters							
- CAPI			•	•		**	
- CAPI + GitOps					•	•	
- CAPI + HCP*						***	•
Production ready	•	•	•			•	•
Data plane isolation							
VM			•				•
microVM				•	•	•	
Isolation level	soft	soft	hard+	hard+	hard+	hard	hard+

* Hosted Control Plane

** Planned feature

*** Possible with vCluster pro version

In the first benchmark(Figure 6.2), the KubeVirt case was provided with two machines, each one having 4 vCPU and 4 GiB memory, meanwhile vCluster was configured to run k8s and a dedicated etcd datastore. The vCluster configuration was explicitly configured in this way for better comparability, because of its extensive possibilities, ranging from running k0s, k3s or k8s and dedicating or using an external datastore like SQLite, PostgreSQL and etcd. The comparison revealed that vCluster significantly outperformed KubeVirt, reducing the time needed to boot a tenant cluster by approximately 50%. On the tested bare metal environment

the time for control plane initialization is 40s to be concluded, consequently vCluster overhead is very limited. In contrast, KubeVirt's CAPI provider follows a sequential process: first, waiting for master nodes to boot, followed by worker nodes. A polling strategy is employed to check node readiness, leading to additional delays when switching from master to worker node bootstrapping. Another factor contributing to the performance difference is that KubeVirt requires the deployment of a network plugin for the cluster to become operational. On the other hand, vCluster does not incur this extra time, as it shares the CNI plugin with the host cluster.

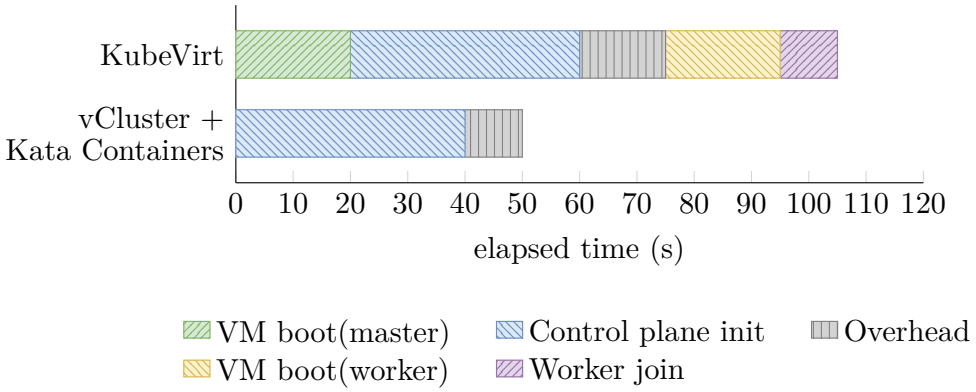

Figure 6.2: Tenant cluster provisioning time comparison. The overhead time in KubeVirt is caused by the slow polling strategy to acknowledge the master nodes as ready.

In the second benchmark(Figure 6.3), the tenant clusters were running the Google microservices demo called Online Boutique [46], and then stressed multiple times with an average of 1000 requests per second for 15 minutes. This test was subdivided into two scenarios: using **one pod per microservice** and **multiple pods per microservice** according to the load using HPA. In the former case, the idea was to measure the base overhead with minimal intervention of the control plane, meanwhile, in the latter measure the control plane overhead in a regular service deployment. The pods resource limits were optimized to sustain such traffic. KubeVirt tenant cluster was provided with two nodes, each having 16 vCPU and 16 GiB memory. vCluster was running using k8s and a dedicated etcd datastore. vCluster with Kata containers required less computational units (about 20%) and a considerable amount of memory (about 60%) compared to KubeVirt clusters. The second scenario the gap in these metrics was even larger hinting that the dedicated Kubernetes control plane is consuming more resources

compared to having multiple microVM to manage for each pod. Performance-wise, the vCluster with Kata containers handled 40–50% more requests per second than KubeVirt, further highlighting the efficiency of the vCluster setup. The additional performance overhead is caused by having the resources distributed on more nodes in the case of KubeVirt, requiring a higher usage of the control plane.

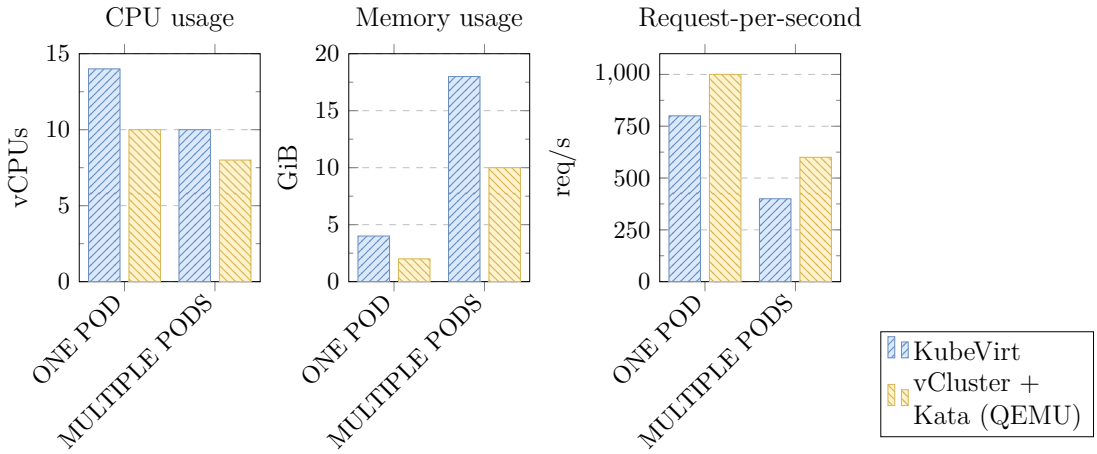

Figure 6.3: Resource and performance comparison simulating 1000 requests per second in a microservices-based application

Chapter 7

Conclusions

Despite Kubernetes not being originally designed with multi-tenancy in mind, its flexibility allows for the implementation of multi-tenant architectures. In this work, we investigated several approaches to leverage Kubernetes' extensibility to create nested clusters, providing isolated environments for independent tenants who may not trust one another. Our results demonstrate that vCluster, combined with Kata containers, is a promising technology stack for addressing the long-standing challenge of Kubernetes multi-tenancy, particularly for organizations that cannot afford the additional resources needed to dedicate an entire cluster to each customer.

7.1 Future Work

While the proposed solution demonstrates its effectiveness, it may not be scalable for future infrastructures comprising hundreds or even thousands of edge clusters. For such a scenario, the most promising approach is to use a Hosted Control Plane to ensure optimal utilization of infrastructure resources, replication for high availability services and efficiently distributed load across the system. Currently, this model is primarily adopted by large enterprises, but in future a larger pool of interested parties could be interested in this model. Future research should investigate this scenario further, focusing on when a Hosted Control Plane becomes more cost-effective than the proposed solution. Additionally, exploring the potential of combining tools such as Kamaji and Kata containers in these large-scale environments could offer innovative ways to improve scalability and resource efficiency.

Bibliography

[1] The Linux Foundation. *White paper, Sharpening the Edge: Overview of the LF Edge Taxonomy and Framework. White paper*. 2020. URL: https://lfedge.org/wp-content/uploads/sites/24/2020/07/LFedge_Whitepaper.pdf (cit. on p. i).

[2] Kubernetes. URL: https://kubernetes.io/ (cit. on p. 1).

[3] Lionel Sujay Vailshery - Statista. *Organizations' adoption level of Kubernetes worldwide in 2022*. URL: https://www.statista.com/statistics/1233945/kubernetes-adoption-level-organization (cit. on p. 1).

[4] Linux Foundation Project. *Sylva*. URL: https://sylvaproject.org/ (cit. on p. 1).

[5] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. «Large-scale cluster management at Google with Borg». In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015 (cit. on p. 3).

[6] etcd docs. *Data model*. URL: https://etcd.io/docs/v3.5/learning/data_model/ (visited on 01/10/2024) (cit. on p. 5).

[7] Kubernetes official documentation. *Kubernetes Multitenancy*. URL: https://kubernetes.io/docs/concepts/security/multi-tenancy/ (visited on 01/10/2024) (cit. on p. 14).

[8] Kubernetes SIGs. *Hierarchical Namespace Controller (HNC)*. URL: https://github.com/kubernetes-sigs/hierarchical-namespaces/tree/master (visited on 01/10/2024) (cit. on p. 16).

[9] Clastix. *Capsule*. URL: https://capsule.clastix.io/docs/#kubernetes-multi-tenancy-made-easy (visited on 01/10/2024) (cit. on p. 16).

[10] Kubernetes official documentation. *API Priority and Fairness*. URL: https://kubernetes.io/docs/concepts/cluster-administration/flow-control/ (visited on 01/10/2024) (cit. on p. 17).

[11] Angel Beltre, Pankaj Saha, and Madhusudhan Govindaraju. «KubeSphere: An Approach to Multi-Tenant Fair Scheduling for Kubernetes Clusters». In: *2019 IEEE Cloud Summit*. 2019, pp. 14–20. DOI: 10.1109/CloudSummit47114.2019.00009 (cit. on p. 17).

[12] Chao Zheng, Qinghui Zhuang, and Fei Guo. «A Multi-Tenant Framework for Cloud Container Services». In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 2021, pp. 359–369. DOI: 10.1109/ICDCS51616.2021.00042 (cit. on pp. 17, 25).

[13] Google. *gVisor*. URL: https://gvisor.dev/docs/ (visited on 01/10/2024) (cit. on p. 20).

[14] William Viktorsson, Cristian Klein, and Johan Tordsson. «Security-Performance Trade-offs of Kubernetes Container Runtimes». In: *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2020, pp. 1–4. DOI: 10.1109/MASCOTS50786.2020.9285946 (cit. on p. 20).

[15] *Virtual Kubelet*. URL: https://github.com/virtual-kubelet/virtual-kubelet/tree/master (visited on 01/10/2024) (cit. on p. 20).

[16] *Liqo*. URL: https://github.com/liqotech/liqo (visited on 01/10/2024) (cit. on p. 20).

[17] *tensible-kube*. URL: https://github.com/virtual-kubelet/tensile-kube (visited on 01/10/2024) (cit. on p. 20).

[18] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. «Firecracker: Lightweight Virtualization for Serverless Applications». In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/agache (cit. on p. 23).

[19] QEMU docs. *'microvm' virtual platform (microvm)*. URL: https://www.qemu.org/docs/master/system/i386/microvm.html#limitations (visited on 01/10/2024) (cit. on p. 23).

[20] SmartX. *VirtInk*. URL: https://github.com/smartxworks/virtink (visited on 01/10/2024) (cit. on p. 23).

[21] *Cloud Hypervisor*. URL: https://github.com/cloud-hypervisor/cloud-hypervisor (visited on 01/10/2024) (cit. on p. 23).

[22] Kubernetes. *Release history - Patch releases*. URL: https://kubernetes.io/releases/patch-releases/#non-active-branch-history (visited on 01/10/2024) (cit. on p. 24).

[23] *Liquid Metal docs*. URL: https://www.liquidmetal.dev/docs/intro (visited on 01/10/2024) (cit. on p. 24).

[24] The New Stack. *End of an Era: Weaveworks Closes Shop Amid Cloud Native Turbulence*. URL: https://thenewstack.io/end-of-an-era-weaveworks-closes-shop-amid-cloud-native-turbulence/ (visited on 01/10/2024) (cit. on p. 24).

[25] Alibaba Kubernetes multi-tenancy SIG. *VirtualCluster incubated project*. URL: https://github.com/kubernetes-retired/multi-tenancy/tree/master/incubator/virtualcluster (visited on 01/10/2024) (cit. on p. 25).

[26] Alibaba. *VirtualCluster*. URL: https://github.com/kubernetes-sigs/cluster-api-provider-nested/tree/main/virtualcluster (visited on 01/10/2024) (cit. on pp. 25, 26).

[27] Loft. *vCluster*. URL: https://vcluster.com/docs/vcluster/introduction/architecture (visited on 01/10/2024) (cit. on p. 25).

[28] Loft. *vCluster docs - Nodes*. URL: https://vcluster.com/docs/vcluster/configure/vcluster-yaml/sync/from-host/nodes (visited on 01/10/2024) (cit. on p. 27).

[29] Kata Containers. *An overview of the Kata Containers project*. URL: https://katacontainers.io/learn/ (visited on 01/10/2024) (cit. on p. 28).

[30] Kata Containers docs. *Host cgroup management*. URL: https://github.com/kata-containers/kata-containers/blob/main/docs/design/host-cgroups.md (visited on 01/10/2024) (cit. on p. 28).

[31] Maxim Krasnyansky. *Universal TUN/TAP device driver*. 2000. URL: https://www.kernel.org/doc/Documentation/networking/tuntap.txt (visited on 01/10/2024) (cit. on p. 30).

[32] Kubernetes SIGs. *ClusterAPI*. URL: https://github.com/kubernetes-sigs/cluster-api (visited on 01/10/2024) (cit. on p. 31).

[33] CNCF. *Certified Kubernetes Software Conformance*. URL: https://www.cncf.io/training/certification/software-conformance/ (visited on 01/10/2024) (cit. on p. 31).

[34] Fabrizio Pandini (VMware). *Happy 7th Birthday kubeadm!* 2023. URL: https://www.cncf.io/training/certification/software-conformance/ (visited on 01/10/2024) (cit. on p. 31).

[35] The ClusterAPI Book - Introduction. *Why build ClusterAPI?* URL: https://cluster-api.sigs.k8s.io/introduction#why-build-cluster-api (visited on 01/10/2024) (cit. on p. 31).

[36] The ClusterAPI Book. *Concepts*. URL: https://cluster-api.sigs.k8s.io/user/concepts (visited on 01/10/2024) (cit. on pp. 32, 33).

[37] The ClusterAPI Book - Introduction. *Why build ClusterAPI?* URL: https://cluster-api.sigs.k8s.io/introduction#why-build-cluster-api (visited on 01/10/2024) (cit. on p. 32).

[38] Kubernetes official documentation. *Cloud Controller Manager*. URL: https://kubernetes.io/docs/concepts/architecture/cloud-controller/ (visited on 01/10/2024) (cit. on p. 34).

[39] The ClusterAPI Book. *Workload bootstrap using GitOps*. URL: https://cluster-api.sigs.k8s.io/tasks/workload-bootstrap-gitops (visited on 01/10/2024) (cit. on p. 35).

[40] Clastix. *Kamaji*. URL: https://kamaji.clastix.io/ (visited on 01/10/2024) (cit. on p. 36).

[41] Chang Jie Guo, Wei Sun, Ying Huang, Zhi Hu Wang, and Bo Gao. «A Framework for Native Multi-Tenancy Application Development and Management». In: *The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE 2007)*. 2007, pp. 551–558. DOI: 10.1109/CEC-EEE.2007.4 (cit. on p. 37).

[42] Ru Jia, Yun Yang, John Grundy, Jacky Keung, and Li Hao. «A systematic review of scheduling approaches on multi-tenancy cloud platforms». In: *Information and Software Technology* 132 (2021), p. 106478. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2020.106478. URL: https://www.sciencedirect.com/science/article/pii/S0950584920302214 (cit. on p. 37).

[43] Berat Can Şenel, Maxime Mouchet, Justin Cappos, Timur Friedman, Olivier Fourmaux, and Rick McGeer. «Multitenant Containers as a Service (CaaS) for Clouds and Edge Clouds». In: *IEEE Access* 11 (2023), pp. 144574–144601. DOI: 10.1109/ACCESS.2023.3344486 (cit. on pp. 38, 39, 42).

[44] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. «Container Security: Issues, Challenges, and the Road Ahead». In: *IEEE Access* 7 (2019), pp. 52976–52996. DOI: 10.1109/ACCESS.2019.2911732 (cit. on p. 40).

[45] Kyungwoon Lee, Jeongsu Kim, Ik-Hyeon Kwon, Hyunchan Park, and Cheol-Ho Hong. «Impact of Secure Container Runtimes on File I/O Performance in Edge Computing». In: *Applied Sciences* 13.24 (2023). ISSN: 2076-3417. DOI: 10.3390/app132413329. URL: https://www.mdpi.com/2076-3417/13/24/13329 (cit. on p. 40).

[46] Google. *Microservices demo (Online Boutique)*. URL: `https://github.com/GoogleCloudPlatform/microservices-demo` (visited on 01/10/2024) (cit. on p. 45).