July 2020

One large cluster or lots of smaller ones?

Flavio Castelli – Distinguished Engineer at SUSE

Join us for KubeCon + CloudNativeCon EU Virtual

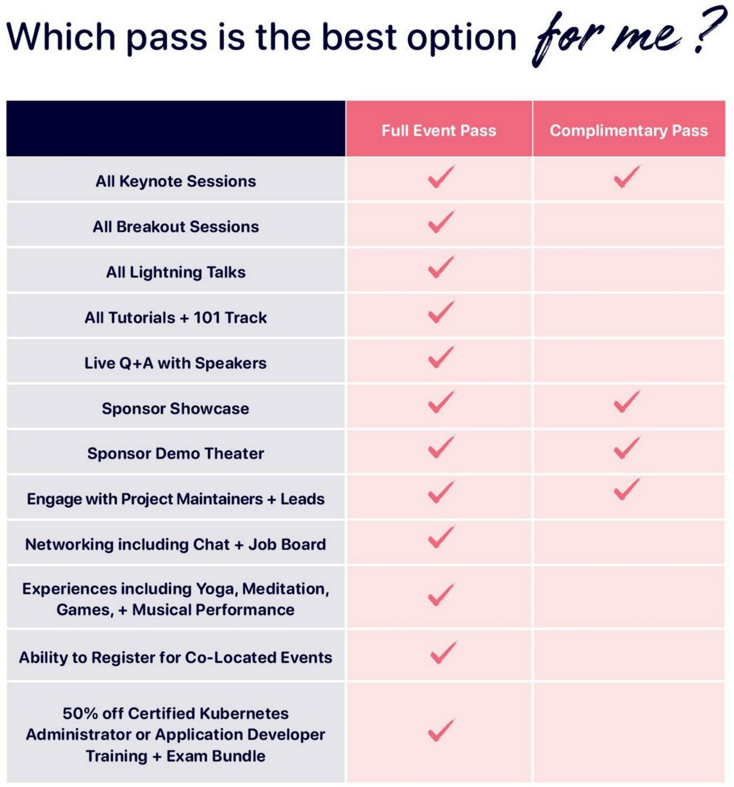
Event dates: August 17-20, 2020

Schedule: [Now available!]

Cost: $75

Full Event Pass and **Complimentary Pass** are now available! Unsure what pass is the best for you? See the chart!

Register now!



Which pass is the best option *for me?*

	Full Event Pass	Complimentary Pass
All Keynote Sessions	✓	✓
All Breakout Sessions	✓	
All Lightning Talks	✓	
All Tutorials + 101 Track	✓	
Live Q+A with Speakers	✓	
Sponsor Showcase	✓	✓
Sponsor Demo Theater	✓	✓
Engage with Project Maintainers + Leads	✓	✓
Networking including Chat + Job Board	✓	
Experiences including Yoga, Meditation, Games, + Musical Performance	✓	
Ability to Register for Co-Located Events	✓	
50% off Certified Kubernetes Administrator or Application Developer Training + Exam Bundle	✓	

A common question

- Multiple teams inside of my organization need a Kubernetes cluster
- Should I create lots of small clusters (one per team)?
- Should I let them share a single larger cluster?
- How safe is to share a single large cluster with multiple tenants?

SUSE

The short answer...

- There’s no right or wrong approach
- There are limitations and benefits with both approaches
- It all depends on what your requirements are, how much you want to compromise

SUSE

Implications of having many small clusters

Advantages:

- Strongest isolation between the tenants
- Maximum flexibility: pick whatever Kubernetes version the tenant wants, let the tenant deploy any kind of workload

Disadvantages:

- Higher maintenance
- Ensure corporate policies are enforced everywhere and are kept in sync
- Inferior hardware utilization
- Higher costs

Implications of sharing a large cluster

Advantages:

- Reduced maintenance
- Consolidate and enforce corporate policies in a single place
- Better hardware utilization

Disadvantages:

- Inferior isolation compared to tenant-reserved clusters
- Less flexible for end user: have to stick with the version of Kubernetes/xyz offered, have to open tickets to get certain privileged operations done

Today’s journey

Our challenges:

- How to host different tenants on a single Kubernetes cluster?
- How secure can it be?

What we will discover:

- Kubernetes has many built-in features that can help with that
- External projects can be used to fill the gaps
- There are however some limits that cannot be overcome

Personas of our story

- Cluster operators:
  - Administrators of the large Kubernetes cluster
  - Have ultimate access to it
- Cluster tenants:
  - They can create and consume resources on the Kubernetes cluster
  - They are bound to rules put in place by the cluster operators

The foundation block

As stated by Kubernetes' documentation:

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

→ Kubernetes Namespaces can be used to isolate different tenants, different projects or any combination of these ones.

Kubernetes resources and Namespace

- Some Kubernetes resources are cluster-wide, but the majority of them are bound to a specific Namespace
- Namespaces alone aren’t enough, there are no security fences around them

Authentication

- A Kubernetes cluster must enforce user authentication against an external identity provider
- Unauthenticated users are prohibited from interacting with the cluster
- Once authenticated, the Kubernetes API server will know the name of the user and the groups she belongs to

Authorization

- RBAC policies instruct Kubernetes about what actions a user/group or users is entitled to perform against a set of Kubernetes resources
- RBAC policies are expressed via:
  - Role/ClusterRole objects: define the target of a policy
  - RoleBinding/ClusterRoleBinding: associates a Role/ClusterRole to a user/group of users

Built-in Roles

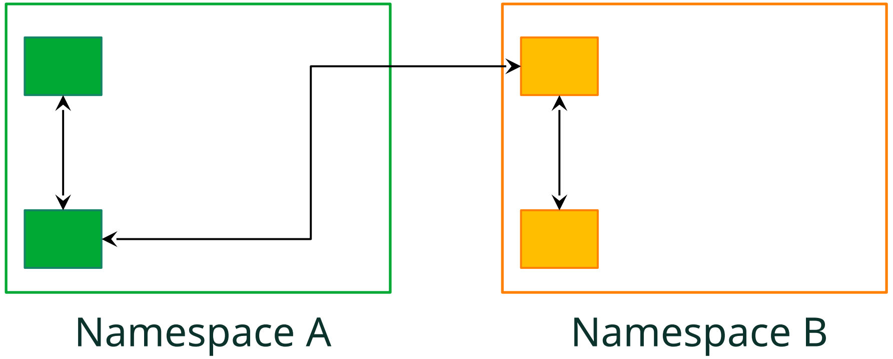
- Kubernetes has a some built-in ClusterRole objects
- These can be used to quickly define RBAC policies both at cluster and at Namespace level
- Most useful ones:
  - *admin*: read/write access to most resources, can manage RBAC policies inside of the Namespace
  - *edit*: like admin, but no way to view or modify RBAC policies. Note well, they can interact with Secrets too
  - *view*: read only access, no way to interact with RBAC policies and Secrets

Building fences around a Namespace

- Quick way: rely on built-in ClusterRoles and create RoleBinding to make them effective inside of a specific Namespace
- Define ad-hoc Roles/ClusterRoles and bind them to a specific Namespace

Network isolation

- By design all Kubernetes Pods can communicate with each other, regardless of the namespace they belong to
- Kubernetes NetworkPolicy can change this behaviour



SUSE

Network Isolation of a tenant

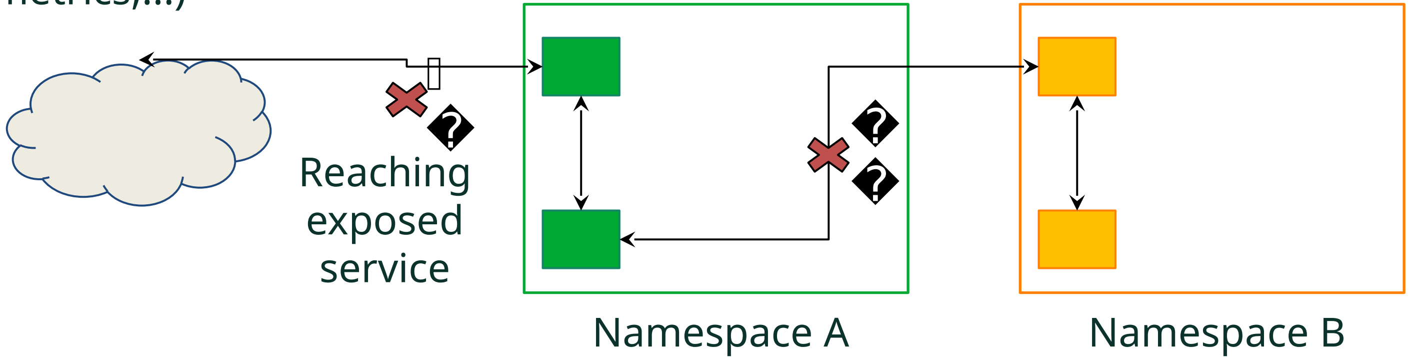
- Pod to Pod communication is allowed inside of the “tenant-a” Namespace
- Outgoing connectivity from “tenant-a” is allowed
- Ingress connectivity from other Namespaces is not allowed

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: isolate-tenant-a
  namespace: tenant-a
spec:
  podSelector:
    matchLabels:
  ingress:
  - from:
    - podSelector: {}
```

SUSE

Reflections on the previous NetworkPolicy

- Pros: no need to add more policies when other tenants are added to the cluster
- Cons: some ingress traffic should be allowed (eg: expose a service, collect metrics,...)



SUSE

Allow some ingress traffic

- Allow incoming traffic from Namespaces with the specified label
- Assumes the ingress controller (nginx-ingress, traefik,...) is deployed inside of a Namespace with the specified label

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-ingress-namespace
  namespace: tenant-a
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          network/policy-group: ingress
  podSelector:
    matchLabels:
  policyTypes:
  - Ingress
```

Limit cluster resource usage

- We want to prevent tenants from starving each other
- We want to properly size the “virtual clusters” of the tenants

→ We need to put limits on resource usage on a Namespace basis

Resource Quotas

- ResourceQuota is a namespaced object
- Defines the amount of resources that are available to a specific Namespace
- Resources can be:
  - Compute resources: CPU, memory
  - Extended resources (e.g. GPUs)
  - Storage resources
  - Other types of Kubernetes objects: Services, CronJobs, Deployments, ...

Storage Quotas

- PersistentVolumes are cluster-wide resources
- PersistentVolumeClaims are namespaced → controllable by Storage Resource Quota
- Storage Resource Quota offers flexibility:
  - Limit the global amount of storage consumed by the Namespace
  - Define different limits based on the storage class used

Object Count Quota

- Can limit the number of Kubernetes objects a Namespace holds: Pods, Deployments, Secrets, ConfigMaps, …
- Worth of note: limit the number of Services of type LoadBalancer

Linux Containers

- Kubernetes Pods are made of Linux Containers
- Linux Containers provide workload isolation, but the kernel of the host is shared with the containerized application
- This has always security implications, but these becomes greater when multiple tenants are sharing the same Kubernetes cluster

How to properly secure Linux containers

- Limit the Linux capabilities granted to containerized workload
- Use a SELinux or AppArmor profile
- Use a seccomp profile
- Do not share host kernel namespaces with the container
- Limit/ban access to the host filesystem
- Limit access to Linux sysctl
- Do not run containerized applications as “root”
- Do not run privileged containers

Pod Security Policies

- Allow the cluster operator to create security profiles for Pods:
  - Define what is considered unacceptable (e.g. access host network, run privileged containers, run containerized application as root,...)
  - Define default values (e.g. enforce usage of seccomp, AppArmor/SELinux)
- PSP are cluster level objects
- Pods violating a PSP won’t be accepted by the Kubernetes API server
- PSP are bound to users/groups via ClusterRoleBinding/RoleBinding

SUSE

How can we leverage Pod Security Policies?

- Operator can define global PSP policies that apply to all users
- Operator can also define ad-hoc PSP that apply to a specific virtual cluster (Namespace) or to specific users

Is that enough?

Using different Container Runtime

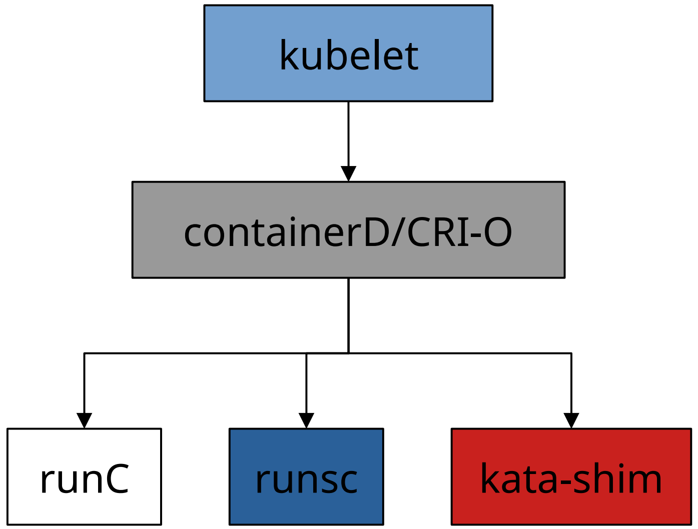
- Majority of Kubernetes deployments uses runC to create Linux Containers
- runC is the reference implementation of the OCI Runtime Specification
- runC can be replaced by other alternative runtimes, as long as they implement the OCI Runtime Specification

Other OCI runtimes with focus on security

- Kata containers: provide stronger isolation by wrapping application containers inside of a lightweight VM
- gVisor: containerized workloads do not share the kernel with the host, they use an application kernel provided by gVisor

Kubernetes Runtime Class

- Allows definition of additional container runtimes that can be used to schedule containers
- Trusted workloads: use a container runtime that uses runC
- Untrusted workloads: use a container runtime that provides higher isolation



Using Kubernetes Runtime Class

- RuntimeClass is a cluster-wide object that must be defined by the cluster operator
- Cluster operators must configure the worker nodes of the cluster accordingly: install required software, properly configure containerD/CRI-O
- To use the non-default runtime class: fill the runtimeClassName attribute when defining a Pod

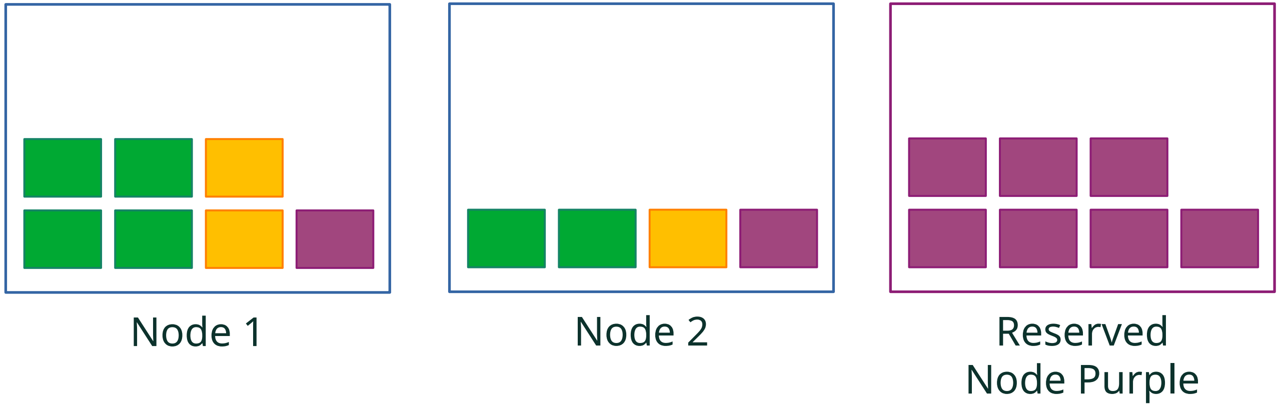
```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  runtimeClassName: secure
  # ...
```

Is this level of separation enough?

- We can use security-focused OCI runtimes to isolate the tenants workloads from each other, but…
- Workloads from different tenants would still be scheduled on the same set of nodes
- Some tenant workloads might require less isolation (use runC, maybe with a more relaxed PSP)

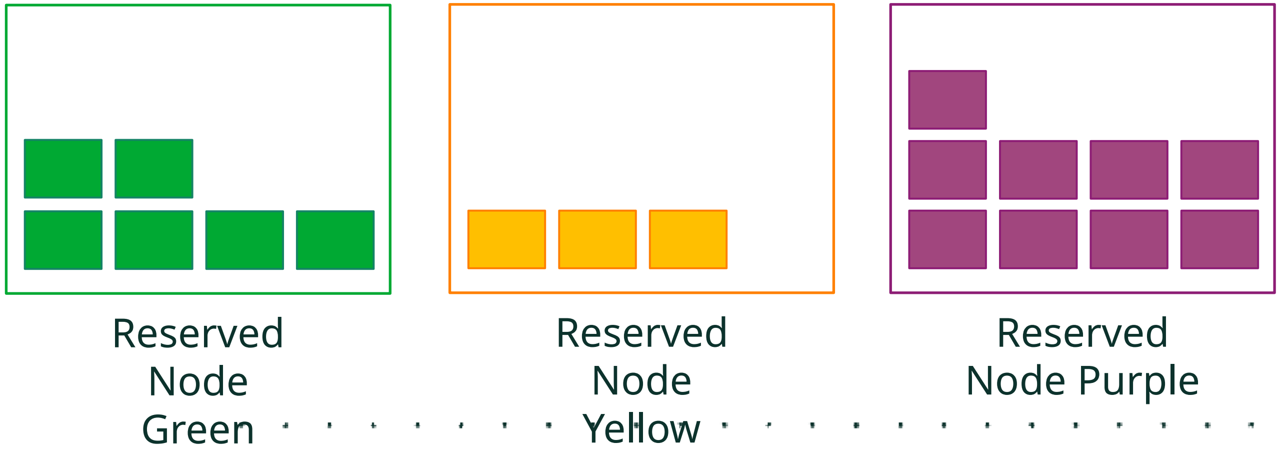
Influencing Kubernetes' scheduler

- Last node has a *taint* that prevents generic workloads from being scheduled on it
- Purple workloads have a *toleration* that allows them to be scheduled on the tainted node
- Purple workloads can still be scheduled on other nodes of the cluster



Influencing Kubernetes' scheduler

- All the nodes have: tenant-based *taint* and a tenant-based *label*
- All the tenant workloads have: right *toleration* and *nodeSelector* constraint



Validation and sanitization of user input

Many of the features shown before rely on the user providing “special” information:

- Quotas: when enforced the user must specify resource limits and requests, otherwise the workload won’t be scheduled
- Secure Container Runtime: the workload must specify the RuntimeClass to be used, otherwise the default one (probably runC) will be used
- Influencing Kubernetes’ scheduler: the right *toleration* and *nodeSelector* constraints must be written

Kubernetes Admission Controllers

- Used to intercept and process all the requests made against Kubernetes' API
- They handle requests after:
  - Authentication has been successfully completed
  - RBAC policies checks passed
- Two types of admission controllers:
  - Validating: accept/reject invalid requests
  - Mutating: accept/reject or accept a modified version of the request

Request validation: use cases

- Influencing the scheduler:
  - Avoid tenant “green” workloads to tolerate the taints of tenant “purple”
  - Avoid tenant “green” to use the *nodeSelector* constraint of tenant “purple”
- Resource quotas:
  - Ensure all workloads have resource request and limits specified
  - Add default request/limit when needed → prevent workloads from being unschedulable
- Container Runtime Class: modify the workload definition to ensure the right runtime is going to be used (instead of the default one)

SUSE

How to use admission controllers

- Kubernetes has already some compiled-in admission controllers
- These controllers are always available (they are part of the api-server process)
- Some of the compiled-in controllers can solve our problems:
  - Quotas: LimitRanger, provide default values
  - Influencing the scheduler: NodeRestriction, allows only admins to change certain node labels (makes *nodeSelector* more secure)
  - Pod Security Policies are implemented using an admission controller

SUSE

Write custom admission controllers

- The compiled-in admissions controllers aren’t enough
- It’s possible to create custom admission controllers by creating a Dynamic Admission Control

How Dynamic Admission Control works

- Two types:
  - ValidatingAdmissionWebhook
  - MutatingAdmissionWebhook
- User implements validation/mutation logic inside of a web application
- kube-api server sends request to HTTP callback and waits for response
- The controller can be deployed on top of the Kubernetes cluster itself
- Note well: the API server is going to validate the request returned by a Mutating Admission Control, it can reject it if invalid

Caveats of Dynamic Admission Control

- You can’t assume they are always reachable and responding properly
- Kubernetes has two “failure policies” for external admission controllers:
  - Ignore: the request is considered valid and accepted as it is
  - Fail: the request is rejected even though it might be perfectly acceptable
- The *“Fail”* policy should be used carefully: it could make the cluster unusable

Standing on the shoulders of giants

- Do you really want to write your own Dynamic Admission Control?
- There's more work to do on top of writing the actual validating/mutating business logic

Open Policy Agent (OPA)

- CNCF project – incubating stage
- Currently undergoing some architectural changes, I’ll focus on its next-gen architecture

Writing custom policies

- Policies are written using a high-level language called Rego
- OPA provides a unit test framework to validate the policies
- Policies are “loaded into Kubernetes” by using a Custom Resource defined by OPA

Key point: users can focus on what matters the most, writing policies!

Gatekeeper

- Dynamic admission controller provided by OPA
- Users write their own policies and Gatekeeper enforces them
- Currently a validation controller, there are plans to add mutating capabilities.

Generic problems of Admission Controllers

- Policies evolve over time: what was considered a valid API request yesterday could be rejected today
- What to do when the Dynamic Admission Controller isn’t reachable/behaving as expected?
- Can we set its failure mode to *Ignore*?
- What if something “slips-into the cluster” due to the *Ignore* failure mode?

OPA - Auditing feature

- OPA policies can be periodically re-evaluated
- Violations are stored inside of the status field of the policy

SUSE

Final considerations

Time for a recap

- It’s possible to share a large Kubernetes cluster with different tenants
- Each tenant can have a dedicated space
- We can put a reasonably secure fence around these dedicated spaces
- This can be done by leveraging features provided by Kubernetes itself
- Some external projects are required to fill some of the gaps (e.g. secure container runtimes, external admission controllers)

Advantages of sharing a single cluster

- Reduced maintenance:
  - Infrastructure: you can limit the number of control plane nodes (master nodes, etcd)
  - Consolidate and enforce policies and rules in a single place
  - Share Kubernetes “add-ons”: monitoring, logging, tracing, service mesh, ingress, …
- Better hardware utilization (unless you start to influence Kubernetes’ scheduler)

Disadvantages of sharing a single cluster

- There’s a limit on the isolation and security we can achieve (*):
    - Control plane is shared between the tenants
    - Have to trust Kubernetes/cluster administrators/external components: security vulnerabilities, bugs inside of policies, corner cases,…
- Everybody has to stay with the same version of Kubernetes
- Deploying certain privileged workloads isn’t allowed to “mere” users: e.g. cluster-wide operators must be installed by the cluster operators

(*) another project worth to be explored is the “Virtual Cluster” one from SIG multi-tenancy

Advantages of having many smaller clusters

- We can achieve the strongest level of isolation
- Everybody gets the version of Kubernetes they need
- Tenant/team can have its own set of admins that can do “everything” on the cluster

Disadvantages of having many smaller clusters

- Increased maintenance:
  - More machines to keep up to date
  - Eventually more Kubernetes versions to care about (from a security POV)
  - Multiple instances of the Kubernetes “add-ons” (monitoring, logging,...)
- Ensure corporate policies are enforced everywhere and are kept synchronized
- Inferior hardware utilization
- Higher costs

Thank you.

SUSE