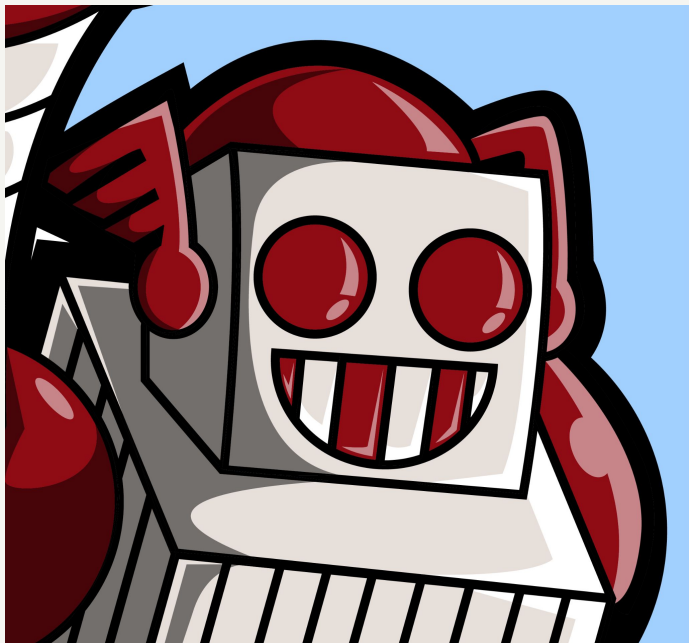


Is The Agent In The Room With Us Right Now?

Who Are We



Andrew Halaney,
Senior Software Engineer

Nick Rutigliano,
Senior Site Reliability Engineer

Compute Runtime

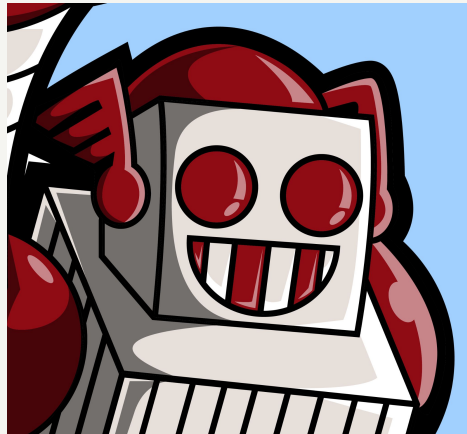
- Kubernetes (Titus) Dataplane
- Base OS

Overview

- What is Titus
- What do Netflix Container Workloads Expect
- Pod Creation Overview & Extension Points
- Deep Dive Into Stages of Creation

What Is Titus

- Our internal container platform
- Originally based on Mesos
- Slowly has morphed to using kubernetes
 - At first just a very thin PodSpec
 - Then we started using **virtual kubelet** + **docker**
 - Lately we use **kubelet** + **containerd**
 - Still mostly just use the PodSpec



What Do Netflix Container Workloads Expect

- Users migrated from EC2 and expect things to feel like a VM
- SSH
- Run as root
- `systemd`
- Storage (EFS, EBS, Lustre, etc)
- Resource isolation

Pod Creation

- To understand **how**, you first need to know **where** you can do isolation
 - This talk is only about dataplane isolation
- From a **kubelet** and container runtime perspective, we think of pod creation in 4 major steps spanning the runtime and image services
 - RunPodSandbox
 - PullImage
 - CreateContainer
 - StartContainer
- Some of these can be configurable, they also each have their own extension points

Pod Creation with Netflix Extensions

NRI

- Node Resource Interface allows plugging domain specific logic into OCI compatible runtimes
- Lifecycle hooks at different stages for pods
- Useful for targeting specific moments (CRI endpoints) in time where we need to make adjustments

Purpose: container adjustments (mounts, devices, env, etc)

OCI Runtime Spec Hooks

- Open Container Initiative Runtime Spec establishes a number of points during container lifecycle that allow for integrations
- There is a ton of points in time available to you

Purpose: side effects within container's environment

Runc / OCI Wrapper

- Why not also wrap your OCI runtime such as **runc**?
- Instead of specific hooks we instruct our container runtime to hit our wrapper of **runc**
- Now with all the commands intercepted we can do modifications to any **runc** command

Purpose: full OCI spec changes

Pod Lifecycle Trace

SERVICE / OPERATION	DESCRIPTION
▼  Kubelet syncPod	Orchestrates the full pod lifecycle.
▼  Containerd RunPodSandbox	Creates the pod sandbox.
 ipman-cni cni.Setup	Assigns IP and configures pod networking via CNI.
 NRI RunPodSandbox	Setup user namespaces.
 Containerd PullImage	Pulls and unpacks the container image.
▼  Containerd CreateContainer	Creates the container with rootfs, mounts, and OCI spec.
 NRI CreateContainer	Adjusts container resources before creation.
 NRI PostCreateContainer	Finalizes container setup before start.
▼  Containerd StartContainer	Executes the container's entrypoint process.
 runc-wrapper Create	Setup seccomp/apparmor and process entrypoints.
 NRI StartContainer	Last-mile adjustments as the container starts.
 Containerd task.Start	Init process begins execution.
 NRI PostStartContainer	Attach storage and sequence processes starting.

-  Kubelet
-  Containerd
-  ipman-cni
-  NRI
-  runc-wrapper

RunPodSandbox

- RunPodSandbox creates the pod level “sandbox” for everything else to exist within, it's in effect what creates the baseline environment for the pod
 - Responsible for creating shared namespaces between containers
 - Creates the cgroups
 - Sets up networking, etc
 - Pause container

User Namespaces

- We use user namespaces to let containers look like they're root
 - Enabled by setting `pod.spec.hostUsers` to `false`
 - This lets the container have full capabilities within the namespace and provides the ability to mount certain filesystems
- Each pod gets its own user range on the host
 - i.e. 100000-165535 for pod 1, 200000-265535 for pod 2
 - In pod 1's container host user 100000 appears as user 0, aka root
- Namespaces form a tree, they can be nested
 - The first user namespace is called the initial user namespace
- `man (7) user_namespaces` is your friend

User Namespaces Related Incident

- `inotify` is a simple API for monitoring filesystem events
- Doesn't require special privileges
- Has per real user ID limits
 - `/proc/sys/fs/inotify/max_user_watches`
 - Documented in `man 7 inotify`

User Namespaces Related Incident

- Imagine: you get paged due to nodes having difficult times running pods

```
(nfsuper) ~ $ journalctl --no-hostname --grep="inotify watch" | tail -n20
```

```
systemd[1]: titus-sidecar-seccomp-agent@default__cf6ce082-6b96-4d61-aab1-008691589a89.service: Failed  
to add control inotify watch descriptor for control group  
/system.slice/system-titus\x2dsidecar\x2dseccomp\x2dagent.slice/titus-sidecar-seccomp-agent@default__cf  
6ce082-6b96-4d61-aab1-008691589a89.service: No space left on device
```

User Namespaces Related Incident

- A bit of spelunking later, you realize that a container is using tons of inotify resources
- You ask the user to stop that, but don't sleep well at night knowing users can intentionally blow up your pager by doing this

```
(root) ~ # for pid in $(ps -e -o pid=) ; do echo "$(cat /proc/$pid/fdinfo/* 2>/dev/null | egrep '^inotify' | wc -l) watches used by $pid"; done | sort -n | tail -n10
```

```
3 watches used by 2799112
```

```
(...)
```

```
19 watches used by 869
```

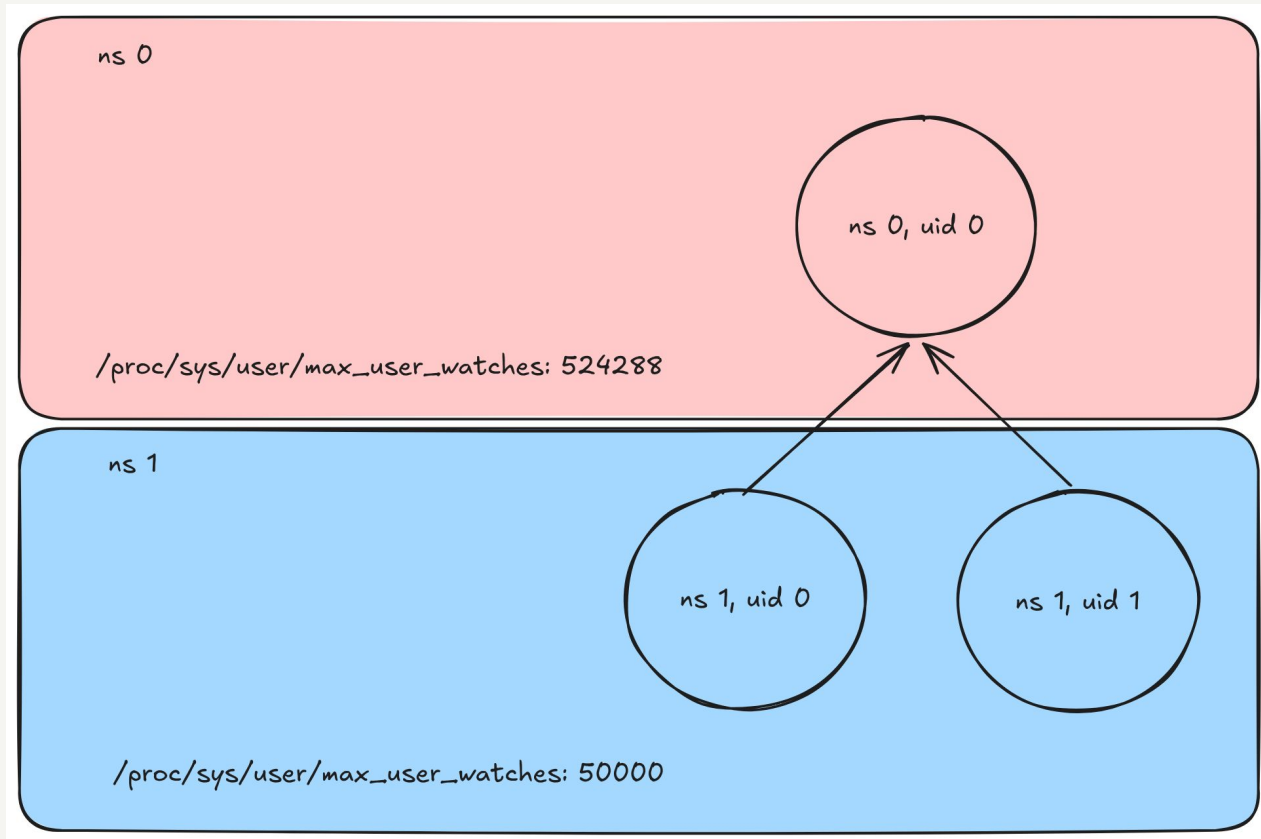
```
210 watches used by 1
```

```
523973 watches used by 1734571
```

User Namespaces Related Incident

- Why did that container bother the root user at all?
 - `/proc/sys/fs/inotify/` knobs are per user
 - Check the kernel source code
 - kernel “user limits” aka `ucounts` are attributed to the owner of a user namespace as well
 - There’s a per user namespace setting as well
 - `containerd` makes the user namespace as root user in RunPodSandbox

User Namespaces Related Incident



User Namespaces Related Incident

- Why did that container bother the root user at all?
 - `/proc/sys/fs/inotify/` knobs are per user
 - per user namespace `/proc/sys/user/max_user_watches`
 - `containerd` makes the user namespace as root user in RunPodSandbox
- What can we do about it?
 - These user limits can be set per user namespace
 - Accounting still rolls up to the root user
 - Better than nothing, let's set the default to something "reasonable"
- PR: <https://github.com/opencontainers/runc/pull/4889>
- Let's see that in action

User Namespace User Limits

Here's the host's view of container's config.json:

```
(root) /proc # cat  
/run/containerd/io.containerd.runtime.v2.task/k8s.io/05d1a5141b8344b18ead8919c878107c7fa8a30429ba1  
fd6e757dcb9163f1acb/config.json | jq | grep user
```

```
"user": {  
  "NETFLIX_APPUSER=appuser",  
  "user.max_inotify_instances": "100",  
  "user.max_inotify_watches": "50000"  
  "type": "user",  
  "path": "/proc/2267174/ns/user"
```

User Namespace User Limits

```
[main](nfsuper) ~/tools $ cat /proc/sys/user/max_inotify_watches  
50000
```

```
[main](nfsuper) ~/tools $ ./inotify_limit_test
```

```
=== inotify limit probe ===
```

```
Kernel inotify limits (from /proc/sys/fs/inotify/):
```

max_user_instances	8192
max_user_watches	524288
max_queued_events	16384

```
Adding watches...
```

```
Hit limit (ENOSPC) after 50000 watches.
```

User Namespace User Limits

The container doesn't have the ability to change this! No escape!

no env/account no asg no region no id

```
(root) /home/nfsuper/tools # echo "524288" >  
/proc/sys/user/max_inotify_watches
```

bash: /proc/sys/user/max_inotify_watches: Read-only file system

User Namespace User Limits

- Arbitrary limits aren't cool
- The container user still uses some of the init namespace root user's limits
- Why not have the container user fully isolated?
- PR: <https://github.com/containerd/containerd/pull/12317/>
- Let's see it in action

User Namespace User Limits

```
(root) # cat /proc/$(pidof inotify_limit_test)/fdinfo/3 | grep inotify | wc -l
```

```
524288
```

```
(root) # ls -lah /proc/self/ns/user /proc/$(pidof inotify_limit_test)/ns/user
```

```
lrwxrwxrwx 1 1147994112 1147994112 0 Mar  6 21:50 /proc/2304559/ns/user -> 'user:[4026533868]'
```

```
lrwxrwxrwx 1 root      root      0 Mar  6 21:54 /proc/self/ns/user -> 'user:[4026531837]'
```

```
(root) # ./usersns_owner /proc/$(pidof inotify_limit_test)/ns/user
```

```
=== User namespace owner probe ===
```

```
/proc/2304559/ns/user owning user namespace owner UID: 1147994112
```

```
parent usersns (depth 1) owner UID: 0
```

```
depth 1: initial user namespace)
```

User Namespace User Limits

```
(root) /home/nfsuper/tools # cat /proc/sys/user/max_inotify_watches  
/proc/sys/fs/inotify/max_user_watches
```

```
524288
```

```
524288
```

```
(root) /home/nfsuper/tools # ./inotify_limit_test
```

```
Hit limit (ENOSPC) after 523855 watches.
```

```
# Fully isolated from the container's user limits!
```

Pod Network Setup / CNI

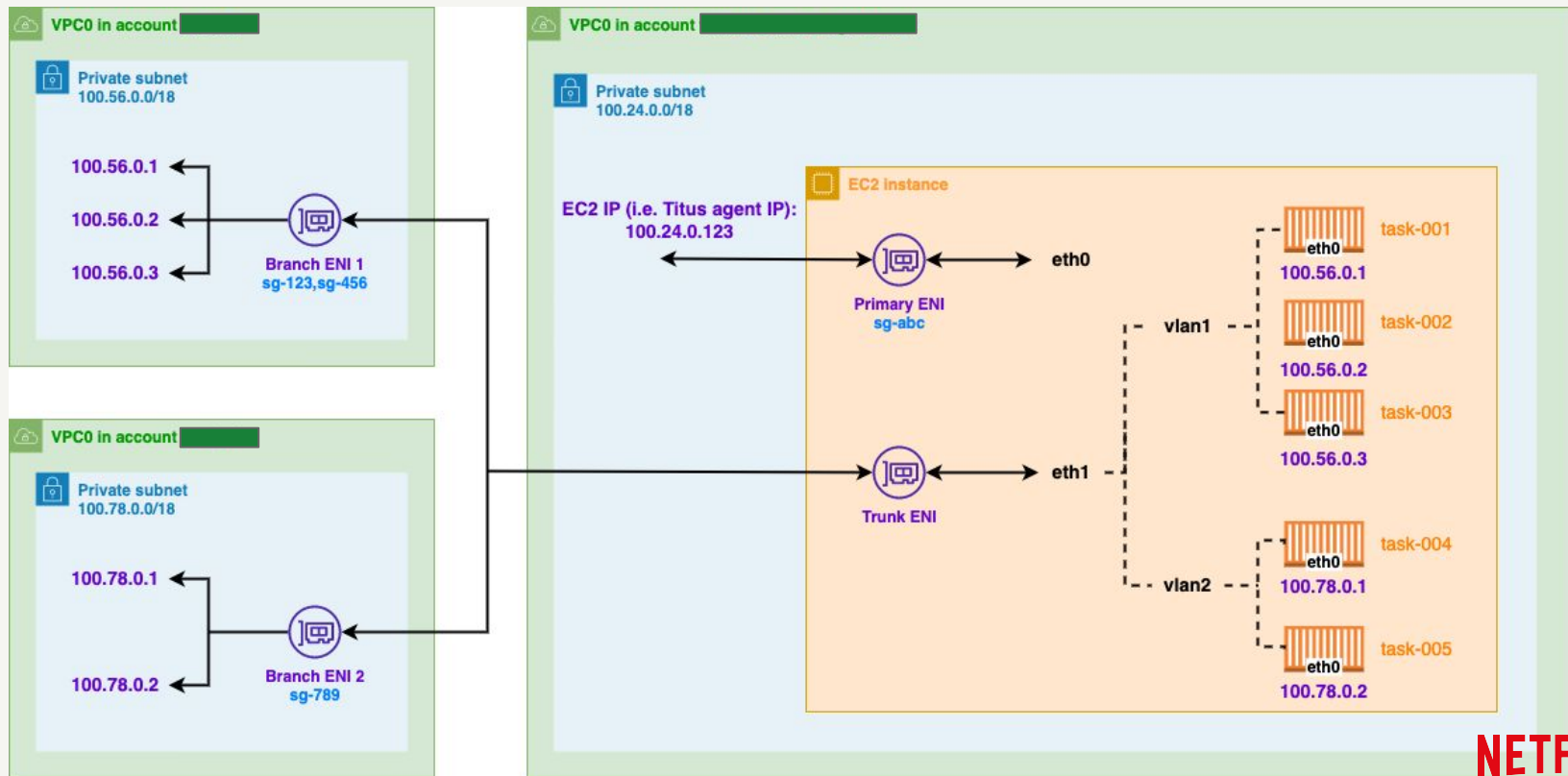
- One of Kubernetes' main networking guarantees is that every pod gets a single stable IP for its lifetime
 - All the containers of a pod must be routable, so need to share a network namespace
- CNI or Container Network Interface is the contract for the container runtime and pod networking
- In our cloud environment, we couldn't bring in an off the shelf CNI for our networking and network isolation needs... so we wrote IPMan

IPMan

- Not to be confused with Ip Man featuring Donnie Yen (on Netflix, highly recommend)
 - IP Manager
- Our main unique CNI requirements
 - Pods can bring their own security groups
 - network isolation



IPMan Core Architecture



IPMan Network Resource Management

- Using **tc** with EDT (earliest departure time) and network bandwidth as a podspec resource, we set network requests/limits for every pod
- eBPF classifies each packet by container (VLAN ID + IP + **tc** class), then stamp each packet with their EDT
 - Replaces our HTB classification which had internal bottlenecks in the kernel
- Stock k8s does not treat network as a resource but you can use a bw plugin via annotations

Off The Shelf CNI Isolation

- NetworkPolicy enforcement happens on the host
- Security groups give singular network identities from a node pov
- Overlay network / node subnet for IP allocation
- Network checks are done by kubelet

IPMan CNI Isolation

- Hypervisor / Hardware based enforcement for access
- Pods have unique network identities with their own security groups
- Pods on the same host can belong to entirely different subnets/VPCs
- Kubelet<->Pods can't route to each directly!

PullImage

- PullImage lets kubelet pull an image
- Volatile overlayfs
- User namespace snapshotter performance improvements
 - idmap improvements
 - erofs user namespace support
- Future mount improvements

Volatile overlays

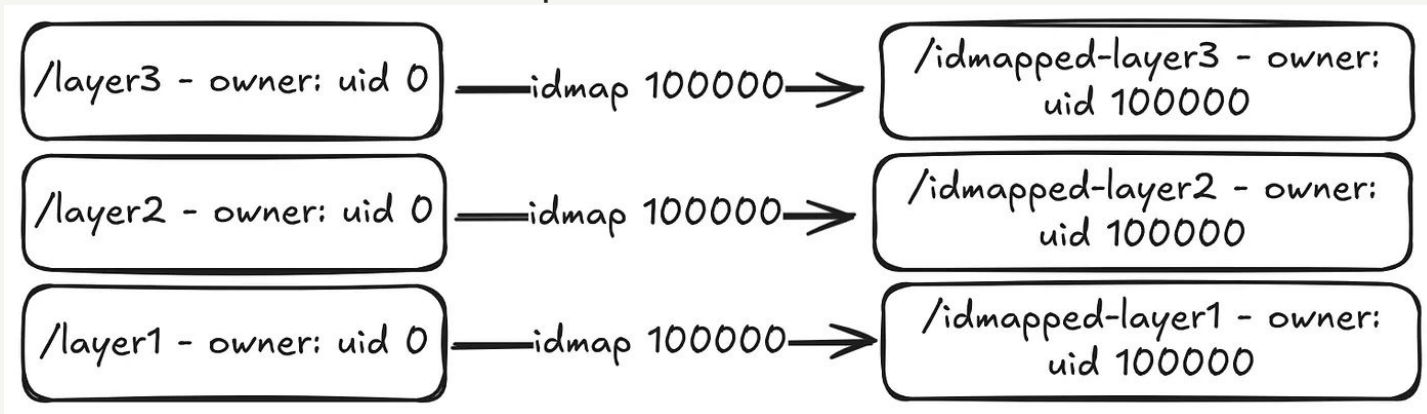
- Default `containerd` overlayfs snapshotter
 - `sync()` / `syncfs()` from within a container causes disk io
 - ephemeral rootfs
 - Backed by shared disk between all pods
- Historically carried kernel patch stubbing `sync()` on overlayfs
- overlayfs now has a `volatile` mount option
 - Can be configured in `containerd`

User Namespaces and Images

- Reminder: from the initial user namespace, each pod is its own user, and containerd is root
- containerd pulls images as root user and preserves image's file permissions
 - This means they're not mapped for each pod to use when using user namespaces

User Namespaces and Images

- **containerd** must do some transformation to make the image layers usable
 - chown?
 - bind mount idmap!



User Namespaces and Images

- This process is costly, once per layer, per container
 - <https://github.com/systemd/systemd/issues/34690>
 - <https://github.com/containerd/containerd/issues/12048>
- Instead do it once per container, mapping the common parent directory of the snapshot instead of each layer
- Added same support for **erofs** snapshotter

Future Mount Improvements

- <https://uapi-group.org/kernel-features/>
- Kubernetes can cause a lot of mounts
 - `systemd` processes each mount event by reading all mounts in `/proc/self/mountinfo`
 - new `fanotify` mount api saying what changed
- `containerd` uses `mount()` api still
 - could use `open_tree()` family of apis to reduce intermediate mounts
- `runc` mount namespace and rootfs setup
 - `unshare`, `pivot_root`, `unmount`
 - could `open_tree_namespace` directly

CreateContainer

- The pod sandbox is setup so now we can start to construct our actual workload
- This hits most all of our extension points: NRI, OCI Hooks, **runc**-wrapper
- So what happens
 - Containerd turns our podspec into an OCI spec
 - NRI's CreateContainer lifecycle hook
 - NRI updates our OCI config to add our **createRuntime** OCI hook
 - Containerd triggers **runc create** which → our **runc** wrapper
 - Our wrapper executes, and in turn calls the underlying **runc**, which in turn calls our newly added **createRuntime** hook

Process Coordination

- Reminder, our containers are special @ Netflix, we tend to have many processes running
- Additionally, we don't use CSI, volumes and data are not set up beforehand
 - In a stock Kubernetes setup, volumes would be set up before CreateContainer via kubelet volume manager
- So our container creation has 2 added goals
 - Processes shouldn't start until their volumes are ready and added
 - Processes may need some coordination in starting correctly

Tini

- **tini** is a small init system we inject into all of our containers
- For all of the processes in container we hijack the processes starting this way
 - Main function of our **runc** wrapper, in that regardless of the container's entrypoint, ours is always **tini**
- A socket is injected as part of this and acts as our coordination mechanism for later
- Why? We've created a PostCreateContainer / PreStartContainer check point preventing TOCTOU issues
 - TOCTOU = Time of Check Time of Use

Seccomp / AppArmor

- **Seccomp / AppArmor** are dynamically composed in our **runc** wrapper instead of being part of the podspec
 - Titus users will declare their required high-level capabilities and we'll translate that into the best choice
- Why in a **runc** wrapper?
 - NRI didn't traditionally support this part of the podspec
 - A Webhook could work, but this has a better failure mode
- In comparison to stock Kubernetes
 - If a **seccompProfile** is missing from the podspec, no syscall filtering prior to 1.27 which added **--seccomp-default**
 - If **appArmorProfile** is missing from the podspec, you get the container-runtime default one

Resource Limits

- Kubernetes by default doesn't mess with rlimits or provide great knobs for it
- **RLIMIT_NPROC**
 - Set to 32000 to avoid fork bombs from pods
 - kubelet has **--pod-max-pids** option
 - Not exactly the same, this is for the whole cgroup not just per uid
- **RLIMIT_NOFILE**
 - Set to 100000 to avoid exhausting host fd limits
- Very recently, KEP-5758 added ways to configure this per pod instead of globally via security context

CPU Isolation

- All of our pods have CPU limits set on their spec
- Kubernetes by default uses cgroupv2's `cpu.max` to enforce CPU time based on this limit
- We've long used our own cpu isolation strategy
- Very closely similar to what became CPU Manager's static policy
 - Higher tier pods are limited via `cpuset`
 - Like Guaranteed pods with CPU Manager, but is also hyperthread aware
 - Lower tier pods are limited in a shared pool (via `cpuset`s) and limited bandwidth via `cpu.max`
 - Like BestEffort/Burstable pods with CPU manager

Storage Isolation

- `containerd` overlayfs snapshotter shares a common backing disk
 - This means each container rootfs / overlayfs shares an underlying resource
 - k8s doesn't add any io limits by default
- Add `io.max` settings for each pod
 - Limits iops / throughput per pod
 - Easy to do with NRI
 - `adjustment.AddLinuxUnified("io.max", ioMaxConfig)`

PostCreateContainer

- We use this to do things that can only happen after the rootfs is setup
- Disk usage isolation via XFS Quota

XFS Quota

- `kubelet` polls a pod's rootfs size and kills it when it goes over the allocated amount
- We instead setup an `xfs` quota per pod
 - Disable kubelet monitoring of overlayfs size (prevents pod from being killed)
 - Project ID per pod
- When quota is hit, pod gets `-ENOSPC` instead of killed
 - This lets our user's applications react to an error instead of reacting to pod failure
- XFS Quota support is now supported for emptyDirs in kubernetes, but not the rootfs
 - `LocalStorageCapacityIsolationFSQuotaMonitoring`

StartContainer

- We've got a functional environment setup including networking and containers as linux tasks
 - The OCI container is created and set to start
 - However, we gate process start by `tini` so we can control the starting sequence of processes in our workloads
- We are starting the task, but `tini` is started and waiting

PostStartContainer

- Why do we have to do these things here
 - We need the user process to claim PID 1 while we setup our “sidecars” to ensure correctness with process reaping
 - We need all the containers fully created before we can attach storage to ensure cross container bind mounts are setup correctly

Tini Part 2 / Storage Attachment

- With our users workload as PID 1 but blocked via **tini**, we can now set up our “sidecar” processes
 - Eg. service mesh
- Once everything is ready but blocked by **tini**, we can attach our storage directly into the pod, inside the container’s mount namespace
 - Because we do cross container bind-mounts, we need this to ensure correctness, stock containers starting are not synchronized inside of a pod
- With storage and sidecars ready, we signal to all **tini** processes to start, the pod is now ready

Tini Part 2 / Storage Attachment

- As compared to standard CSI usage / no process coordination
 - Volumes are never staged on host paths, as compared to standard CSI usage where they are before being bind mounted into the container
 - No retry loop side effects are possible, as CSI failure causes a full pod creation loop which can leave artifacts left on the host between attempts

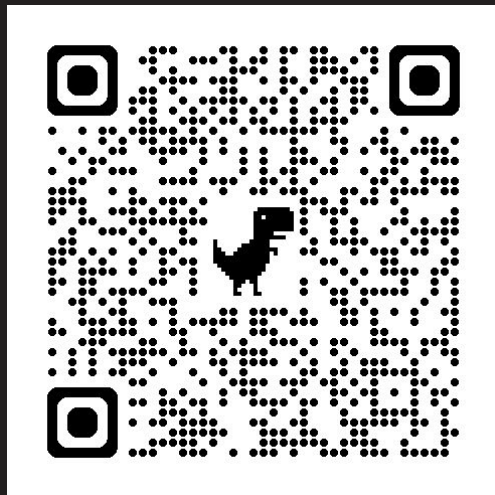
Recap

- RunPodSandbox
 - User Namespaces
 - CNI
- PullImage
 - Volatile overlayfs
 - User Namespaces
 - Future improvements
- CreateContainer
 - Process Coordination
 - Seccomp/AppArmor
 - Resource Limits
 - CPU/Storage Throughput Isolation
- PostCreateContainer
 - Storage Capacity Isolation
- PostStartContainer
 - Process Coordination 2
 - External Storage Attachment

Thank You!

feedback/questions:

<https://forms.gle/9TBrwnuqQz2AakdL8>



NETFLIX