Conteneurisation et virtualisation sous GNU/Linux

Étude technique approfondie des *cgroups*, namespaces, LXC, LXD, Docker, Podman et machines virtuelles

LinuXMaine.org

Juin 2026

Table des matières

Chapitre 1

Introduction générale

1.1 Pourquoi ce document ?

L'isolation des charges de travail (*workloads*) est l'un des piliers de l'informatique moderne : elle conditionne la sécurité, la densité d'hébergement, la reproductibilité des déploiements et l'efficacité opérationnelle. Deux grandes familles de techniques coexistent :

1. **La virtualisation matérielle** (*Virtual Machines*, VM) : on émule ou on virtualise un ordinateur complet, avec son propre noyau de système d'exploitation, au-dessus d'un **hyperviseur**.
2. **La conteneurisation** (*containers*) : on isole des processus qui partagent le **même noyau** que l'hôte, grâce à des mécanismes natifs du noyau Linux — principalement les **namespaces** et les **cgroups**.

Ce document décrit en profondeur les fondations noyau (avec un focus marqué sur les *control groups*, les **cgroups**), puis l'historique et la description technique de chaque grande technologie : **VM**, **LXC**, **LXD** (et son successeur **Incus**), **Docker** et **Podman**. Il propose ensuite une comparaison complète, une analyse des *stacks* et des ressources consommées, un lexique français, une bibliographie francophone et une liste de liens utiles.

1.2 La différence fondamentale en une image mentale

Aspect	Machine virtuelle (VM)	Conteneur
Unité isolée	Un ordinateur complet émulé	Un ou plusieurs processus
Noyau	Un noyau invité **par VM**	Noyau de l'hôte **partagé**
Frontière d'isolation	Hyperviseur + matériel virtualisé (Ring -1 / VT-x, AMD-V)	Namespaces + cgroups + LSM (frontière logicielle)
Démarrage	Secondes à dizaines de secondes (boot d'un OS)	Millisecondes à secondes
Empreinte disque	Gigaoctets (image disque + OS complet)	Mégaoctets (couches de système de fichiers)
Densité par hôte	Dizaines	Centaines à milliers
Surface d'isolation	Très forte (matériel)	Plus faible (logicielle, dépend de la configuration)

La phrase à retenir : **une VM virtualise le matériel ; un conteneur virtualise le système d'exploitation.** Tout le reste découle de cette distinction.

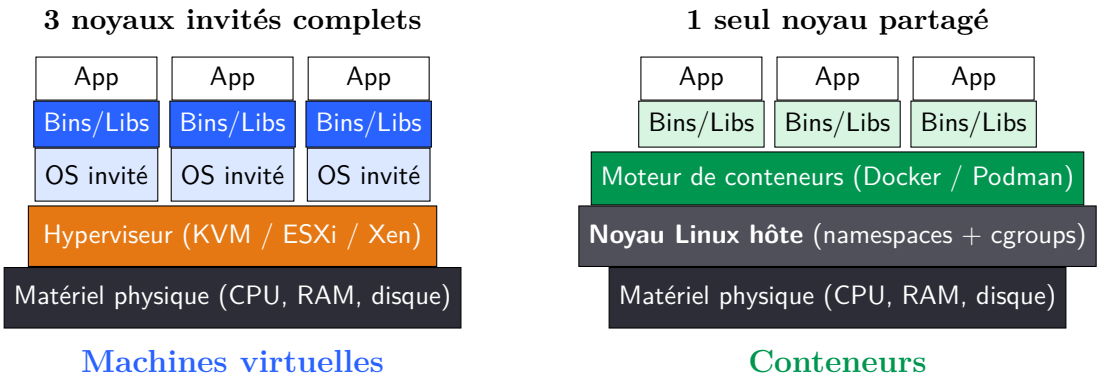


FIG. 1.1 : Comparaison des piles : la VM duplique l'OS complet pour chaque instance ; le conteneur partage le noyau de l'hôte.

Chapitre 2

Les fondations dans le noyau Linux

Un conteneur n'est pas un objet du noyau. **Il n'existe aucune structure `struct container` dans le code du noyau Linux.** Un conteneur est une *construction de l'espace utilisateur* : c'est un (ou plusieurs) processus ordinaire(s) auquel on a appliqué une combinaison de mécanismes d'isolation et de limitation. Les outils comme Docker ou LXC ne sont que des **orchestrateurs** de ces mécanismes noyau. Comprendre les conteneurs, c'est donc comprendre ces briques.

Les briques essentielles sont :

- **Les namespaces** — *qu'est-ce que le processus peut voir ?* (isolation de la visibilité)
- **Les cgroups** — *combien de ressources le processus peut-il consommer ?* (limitation et comptabilité)
- **Les capabilities, seccomp et les LSM** — *que peut faire le processus ?* (réduction des privilèges)
- **Les systèmes de fichiers en couches** (OverlayFS) et `pivot_root` — *quelle racine voit le processus ?* (isolation du système de fichiers)

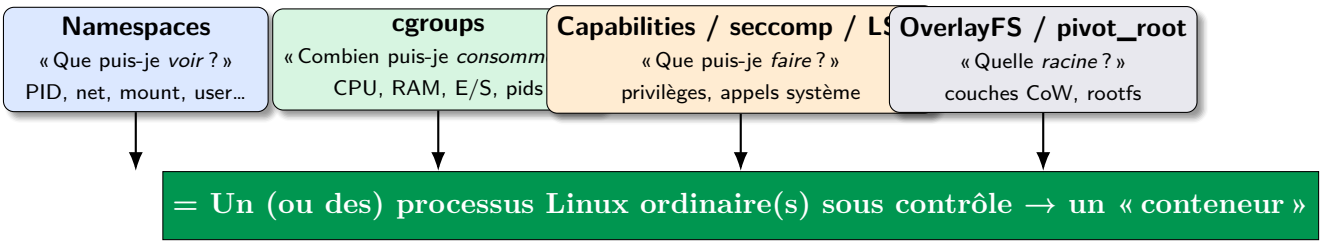


FIG. 2.1 : Anatomie d'un conteneur : il n'existe pas d'objet « conteneur » dans le noyau, seulement la combinaison de quatre familles de mécanismes appliquées à des processus ordinaires.

2.1 Les namespaces (espaces de noms)

Un *namespace* virtualise une ressource globale du système de telle sorte que les processus à l'intérieur du namespace voient leur propre instance isolée de cette ressource. Introduits progressivement à partir du noyau 2.4.19 (2002, *mount namespace*), ils sont aujourd'hui au nombre de huit.

Namespace	Constante `clone()`	Ce qu'il isole	Depuis
Mount	`CLONE_NEWNS`	Points de montage du système de fichiers	2.4.19 (2002)
UTS	`CLONE_NEWUTS`	*Hostname* et *domainname*	2.6.19 (2006)

Namespace	Constante `clone()`	Ce qu'il isole	Depuis
IPC	`CLONE_NEWIPC`	Files de messages, sémaphores, mémoire partagée System V / POSIX	2.6.19 (2006)
PID	`CLONE_NEWPID`	Arborescence des identifiants de processus	2.6.24 (2008)
Network	`CLONE_NEWNET`	Interfaces réseau, tables de routage, *firewall*, sockets	2.6.29 (2009)
User	`CLONE_NEWUSER`	Mappage des UID/GID, capabilities	3.8 (2013)
Cgroup	`CLONE_NEWCGROUP`	Vue de la racine de la hiérarchie cgroup	4.6 (2016)
Time	`CLONE_NEWTIME`	Horloges `CLOCK_MONOTONIC` et `CLOCK_BOOTTIME`	5.6 (2020)

Les trois appels système clés sont :

- `clone()` : crée un nouveau processus en choisissant les namespaces à créer (drapeaux `CLONE_NEW*`).
- `unshare()` : détache le processus appelant de certains namespaces partagés (les place dans de nouveaux namespaces).
- `setns()` : fait *entrer* un processus dans un namespace existant (mécanisme utilisé par `docker exec` ou `nsenter`).

Le **PID namespace** est emblématique : le premier processus créé dans un nouveau PID namespace porte le PID 1 *à l'intérieur* du conteneur (il joue le rôle d'`init`), tout en ayant un autre PID dans l'hôte. S'il meurt, tout le namespace est détruit (les processus orphelins sont tués). C'est pourquoi le choix du processus `init` d'un conteneur (par ex. `tini`, `dumb-init`, ou `systemd`) est important pour la récupération des zombies et la propagation des signaux.

Le **user namespace** est la brique de sécurité majeure : il permet à un processus d'être *root* (UID 0) **à l'intérieur** du conteneur tout en étant un utilisateur **non privilégié** (par ex. UID 100000) dans l'hôte. C'est le fondement des **conteneurs sans privilèges** (*rootless* / *unprivileged*), pierre angulaire de Podman et des conteneurs LXC non privilégiés.

2.2 Les *control groups* (cgroups) — étude approfondie

2.2.1 Définition et rôle

Les **cgroups** (*control groups*, groupes de contrôle) constituent le mécanisme du noyau Linux qui permet d'**organiser hiérarchiquement les processus** et d'y appliquer des politiques de **limitation**, de **priorisation**, de **comptabilité** (*accounting*) et de **contrôle** (gel/reprise) de l'usage des ressources matérielles : CPU, mémoire, entrées/sorties bloc, réseau, périphériques, nombre de processus, etc.

Là où les namespaces répondent à « *que vois-je ?* », les cgroups répondent à « *combien puis-je consommer ?* ». Les deux sont **orthogonaux** et **indépendants** : on peut utiliser l'un sans l'autre, mais c'est leur combinaison qui rend les conteneurs utiles et sûrs.

2.2.2 Histoire des cgroups

- **2006-2007** — Le projet débute chez Google sous le nom de *process containers*, porté par **Paul Menage** et **Rohit Seth**. Le besoin : isoler et limiter des groupes de tâches sur les énormes parcs de serveurs de Google.
- **2007** — Renommé *control groups* (« container » prêtait à confusion) et intégré dans le noyau **2.6.24** (janvier 2008). C'est la **v1**.
- **2008-2014** — Multiplication des *contrôleurs* (mémoire, cpu, cpuset, blkio, devices, freezer, net_cls…). La v1 souffre d'un défaut de conception : **chaque contrôleur a sa propre hiérarchie**, ce qui rend la coordination entre contrôleurs complexe et incohérente.
- **2012-2016** — **Tejun Heo** conçoit et développe la **cgroup v2** (initialement « unified hierarchy »). Objectif : **une seule hiérarchie unifiée** pour tous les contrôleurs.
- **2016** — cgroup v2 déclarée stable dans le noyau **4.5**.
- **2019** — **Fedora 31** est la première grande distribution à basculer par défaut sur cgroup v2 (« unified »).
- **2021-2022** — Bascule généralisée : Debian 11, Ubuntu 22.04, RHEL 9, Arch, etc. activent cgroup v2 par défaut. systemd, Docker, Podman, LXC, Kubernetes prennent en charge la v2.

2.2.3 cgroup v1 : la hiérarchie multiple

Dans la v1, chaque **contrôleur** (aussi appelé *subsystem*) est monté sur sa propre hiérarchie dans `/sys/fs/cgroup/` :

```
/sys/fs/cgroup/
├── cpu/          (contrôleur cpu)
├── cpuacct/      (comptabilité cpu)
├── cpuset/       (affinité CPU/mémoire NUMA)
├── memory/       (limitation mémoire)
├── blkio/        (E/S bloc)
├── devices/      (autorisation d'accès aux périphériques)
├── freezer/      (gel/reprise de processus)
├── net_cls/      (marquage de paquets réseau)
├── pids/         (nombre de processus)
└── ...
```

Avantages de la v1 : souplesse (un processus peut appartenir à des groupes différents selon le contrôleur), maturité.

Inconvénients de la v1 : incohérences entre hiérarchies, impossibilité de raisonner globalement sur un groupe, gestion mémoire et E/S mal coordonnées (le contrôleur `memory` et le contrôleur `blkio` ne « savent » pas qu'ils parlent du même groupe), complexité de l'espace utilisateur.

2.2.4 cgroup v2 : la hiérarchie unifiée

La v2 impose **une arborescence unique** montée en `/sys/fs/cgroup/`. Tous les contrôleurs partagent la même structure de répertoires. Chaque répertoire est un cgroup ; on y écrit des fichiers de contrôle, et on y place des processus en écrivant leur PID dans `cgroup.procs`.

```
/sys/fs/cgroup/                    (racine)
├── cgroup.controllers             (contrôleurs disponibles)
├── cgroup.subtree_control         (contrôleurs délégués aux enfants)
├── cgroup.procs                   (PID des processus du groupe)
├── cpu.max                        (limite CPU : "quota période")
├── cpu.weight                     (poids relatif, défaut 100)
├── memory.max                     (limite dure de mémoire)
├── memory.high                    (seuil de pression mémoire)
```

```
├── memory.current                  (usage courant — lecture)
├── io.max                          (limites d'E/S bloc par périphérique)
├── pids.max                        (nombre maximal de processus)
└── mon-conteneur/                  (cgroup enfant)
    ├── cgroup.procs
    ├── cpu.max
    ├── memory.max
    └── ...
```

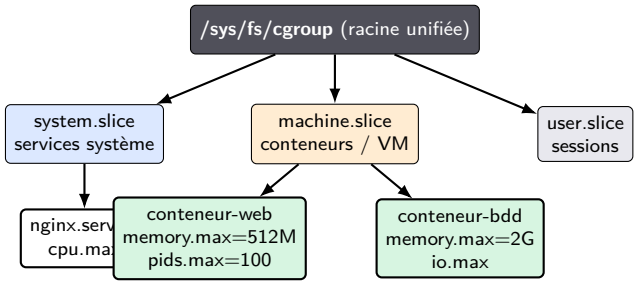


FIG. 2.2 : Hiérarchie unifiée cgroup v2 : chaque nœud applique poids, limites et protections aux processus qu'il contient. Les feuilles (services, conteneurs) portent les processus ; les *slices* structurent l'arbre.

2.2.4.1 Principes structurants de la v2

1. **Règle « no internal process »** : seuls les cgroups *feuilles* (sans enfant activant des contrôleurs) peuvent contenir des processus. Un cgroup interne ne peut pas à la fois contenir des processus *et* distribuer des contrôleurs à ses enfants. Cela force une organisation propre.
2. **Délégation explicite via `cgroup.subtree_control`** : un contrôleur n'agit sur les enfants que s'il a été explicitement « activé » dans le parent (`echo "+memory +cpu" > cgroup.subtree_control`).
3. **Modèle de ressources cohérent** : *weights* (poids relatifs, ex. `cpu.weight`), *limits* (plafonds durs, ex. `memory.max`, `cpu.max`), *protections* (garanties basses, ex. `memory.low`, `memory.min`) et *allocations* exclusives (ex. `cpuset`).

2.2.5 Les principaux contrôleurs cgroup v2 en détail

2.2.5.1 Contrôleur `cpu`

- `cpu.weight` : poids relatif de partage du CPU (1 à 10000, défaut 100). C'est l'équivalent normalisé des anciennes *cpu.shares*. Si deux groupes ont des poids 100 et 200, le second obtient deux fois plus de temps CPU *en cas de contention*.
- `cpu.max` : plafond absolu sous la forme `$QUOTA $PERIODE` (en microsecondes). Ex. `50000 100000` = au plus 50 ms de CPU toutes les 100 ms = **0,5 cœur**. C'est ce qui implémente l'option `--cpus="0.5"` de Docker.
- `cpu.stat` : statistiques (temps total, périodes throttlées, etc.).

Le *throttling* (étranglement) CPU est un point d'attention : une application multithread peut consommer son quota très vite puis être « gelée » jusqu'à la fin de la période, provoquant des latences (problème bien connu en Java/JVM et Kubernetes).

2.2.5.2 Contrôleur `memory`

- `memory.max` : limite **dure**. Dépassement → le processus déclenche le **OOM-killer** du cgroup (tué dans le groupe, pas ailleurs).

- `memory.high` : seuil **souple** de pression. Au-delà, le noyau met le groupe sous forte pression de récupération (*reclaim*) et ralentit les allocations, sans tuer immédiatement. Outil clé pour éviter les morts brutales.
- `memory.low` / `memory.min` : **protections**. La mémoire en dessous de `memory.min` n'est jamais récupérée (garantie dure) ; `memory.low` est une garantie souple.
- `memory.current`, `memory.stat`, `memory.events` : comptabilité fine (anon, *page cache*, *slab*, *swap*, nombre d'événements OOM…).
- `memory.swap.max` : contrôle l'usage du swap par le groupe.

La v2 unifie la comptabilité de la mémoire *page cache* et de la mémoire anonyme, ce que la v1 faisait mal — d'où une bien meilleure gestion de la pression mémoire et de la PSI (voir plus bas).

2.2.5.3 Contrôleur `io`

- `io.max` : limites par périphérique (`MAJ:MIN`) en IOPS et/ou en débit (`rbps`, `wbps`, `riops`, `wiops`).
- `io.weight` : pondération proportionnelle des E/S (nécessite le *scheduler* bloc adéquat, ex. `bfq`).
- `io.stat` : statistiques d'E/S par périphérique.

En v2, le contrôleur `io` est correctement couplé au contrôleur `memory`, ce qui permet enfin de comptabiliser les écritures différées (*writeback*) au bon cgroup — un manque criant de la v1.

2.2.5.4 Contrôleur `pids`

- `pids.max` : nombre maximal de processus/threads dans le groupe. Protection essentielle contre les *fork bombs*.
- `pids.current` : nombre courant.

2.2.5.5 Contrôleur `cpuset`

- `cpuset.cpus` : ensemble de cœurs CPU autorisés (ex. `0-3,8`).
- `cpuset.mems` : nœuds mémoire NUMA autorisés.
- Utile pour le *pinning* CPU, l'isolation NUMA et les charges temps réel.

2.2.5.6 Contrôleurs `hugetlb`, `rdma`, `misc`

Gèrent respectivement les *huge pages*, les ressources RDMA (réseau haute performance), et des ressources diverses comptées par clé (ex. instances de périphériques).

2.2.5.7 Le « freezer » (intégré en v2)

`cgroup.freeze` (écrire `1`) gèle l'ensemble des processus du groupe sans les tuer — pratique pour la migration, la sauvegarde cohérente, ou la mise en pause d'un conteneur (`docker pause`).

2.2.6 PSI — Pressure Stall Information

Introduite par **Johannes Weiner** (Facebook) dans le noyau **4.20** (2018), la **PSI** expose dans chaque cgroup (`cpu.pressure`, `memory.pressure`, `io.pressure`) le **pourcentage de temps** pendant lequel des tâches sont **bloquées** faute de ressource. C'est un signal bien plus exploitable que la simple charge moyenne (*load average*). Des démons comme `oomd` (Facebook) ou `systemd-oomd` s'en servent pour tuer proactivement des charges avant l'effondrement.

2.2.7 Interaction avec systemd

Sur les distributions modernes, **systemd est le gestionnaire de cgroups par défaut** : il considère la hiérarchie cgroup comme une ressource qu'**il** possède et qu'il ne faut pas modifier « à

la main » (*single writer rule*). systemd organise tout en trois types d'unités :

- **`.service`** : un démon ou un service.
- **`.scope`** : un groupe de processus créés par un tiers (ex. une session, un conteneur lancé par Podman).
- **`.slice`** : un nœud d'arborescence pour grouper services et scopes (`system.slice`, `user.slice`, `machine.slice` pour les conteneurs et VM via `machinectl`).

On pilote alors les limites via les directives systemd (`CPUQuota=50%`, `MemoryMax=2G`, `MemoryHigh=1.5G`, `TasksMax=100`, `IOReadBandwidthMax=…`), qui se traduisent en fichiers cgroup v2. La commande `systemd-cgls` affiche l'arborescence et `systemd-cgtop` le *top* par cgroup.

2.2.8 Manipulation pratique des cgroups v2

Création manuelle d'un groupe limité à 0,5 CPU et 256 Mo :

```
# Activer les contrôleurs pour les enfants de la racine
echo "+cpu +memory +pids" | sudo tee /sys/fs/cgroup/cgroup.subtree_control

# Créer un cgroup
sudo mkdir /sys/fs/cgroup/demo

# 0,5 cœur : 50 ms de quota toutes les 100 ms
echo "50000 100000" | sudo tee /sys/fs/cgroup/demo/cpu.max

# 256 Mo de limite dure
echo "268435456" | sudo tee /sys/fs/cgroup/demo/memory.max

# Au plus 50 processus
echo 50 | sudo tee /sys/fs/cgroup/demo/pids.max

# Placer le shell courant dans le groupe
echo $$ | sudo tee /sys/fs/cgroup/demo/cgroup.procs
```

Avec les outils de l'espace utilisateur (`libcgroup`) ou systemd :

```
# Lancer une commande transitoire sous limites systemd
systemd-run --scope -p CPUQuota=50% -p MemoryMax=256M stress-ng --cpu 4

# Visualiser
systemd-cgtop
```

2.2.9 Avantages, inconvénients et limitations des cgroups

Avantages

- Limitation et comptabilité **fines** et **fiables** des ressources.
- Isolation de la performance (un conteneur ne peut pas affamer les autres — *noisy neighbor*).
- Base indispensable de la facturation/quotas dans le *cloud*.
- v2 : cohérence, PSI, meilleure gestion mémoire/E/S.

Inconvénients / limitations

- Les cgroups **ne sont pas de la virtualisation** : `/proc/cpuinfo`, `/proc/meminfo`, `nproc`, `free`, `top` voient **les ressources de l'hôte**, pas les limites du cgroup (d'où l'intérêt de `LXCFS` qui « ment » à `/proc` pour refléter les limites — voir LXC/LXD).
- Le *throttling* CPU peut induire des **latences** inattendues.

- La v1 et la v2 coexistent encore (mode « hybride ») et certaines vieilles applications ne gèrent pas la v2.
- La règle « no internal process » de la v2 a cassé des outils historiques.
- Les cgroups ne protègent **pas** contre les vulnérabilités du noyau partagé (rôle des LSM/seccomp).

2.3 Capabilities, seccomp et modules de sécurité (LSM)

L'isolation des conteneurs ne serait pas crédible sans réduction des privilèges. Trois mécanismes complémentaires interviennent :

- **Linux capabilities** : découpe les pouvoirs de *root* (UID 0) en ~40 privilèges atomiques (`CAP_NET_ADMIN`, `CAP_SYS_ADMIN`, `CAP_NET_BIND_SERVICE`…). Docker, par défaut, **retire la plupart des capabilities** et n'en conserve qu'une quinzaine. On affine avec `--cap-add` / `--cap-drop`.
- **seccomp** (*secure computing mode*) : filtre les **appels système** autorisés via des profils BPF. Le profil par défaut de Docker bloque ~44 appels système dangereux (`kexec_load`, `mount`, `ptrace` selon contexte…).
- **LSM** (*Linux Security Modules*) : **SELinux** (Red Hat/Fedora) et **AppArmor** (Debian/Ubuntu/SUSE) appliquent un **contrôle d'accès obligatoire** (MAC) qui confine ce qu'un processus peut lire/écrire/exécuter, indépendamment des permissions UNIX classiques.

C'est la **combinaison** namespaces + cgroups + capabilities + seccomp + LSM + user namespaces qui constitue la « frontière » réelle d'un conteneur. Aucun de ces mécanismes pris isolément ne suffit.

2.4 Le système de fichiers : OverlayFS et `pivot_root`

Les images de conteneurs sont construites en **couches** (*layers*) empilées et partagées entre conteneurs grâce à un **système de fichiers union**, le plus souvent **OverlayFS** (intégré au noyau depuis 3.18).

- Une **couche basse** (*lowerdir*) en lecture seule : l'image.
- Une **couche haute** (*upperdir*) en lecture/écriture : les modifications du conteneur.
- Un mécanisme **copy-on-write (CoW)** : un fichier n'est copié dans la couche haute que lorsqu'il est modifié.

Cela explique la **légèreté** des conteneurs : dix conteneurs basés sur la même image Ubuntu partagent physiquement les mêmes couches en lecture seule. Au démarrage, le runtime change la racine du système de fichiers du conteneur avec **`pivot_root`** (préféré à `chroot`, plus sûr).

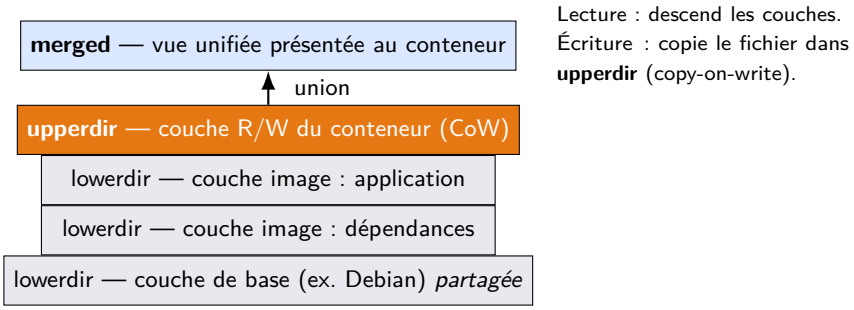


FIG. 2.3 : OverlayFS : empilement de couches en lecture seule partagées et d'une couche d'écriture par conteneur, fusionnées en une vue unique.

Chapitre 3

Historique chronologique de la conteneurisation

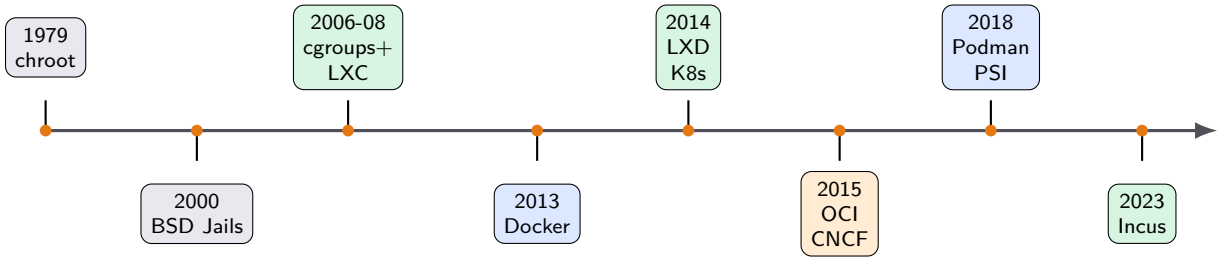


FIG. 3.1 : Grandes étapes : de l'isolation du système de fichiers (`chroot`, 1979) à la standardisation OCI et aux moteurs modernes.

Année	Jalon
1979	`chroot` apparaît dans Unix V7 — ancêtre lointain de l'isolation du système de fichiers.
1982	`chroot` ajouté à BSD.
2000	**FreeBSD Jails** — isolation complète au niveau OS sur BSD (précurseur conceptuel des conteneurs).
2001	**Linux-VServer** — premier *patch* d'isolation pour Linux.
2004	**Solaris Zones / Containers** (Sun Microsystems) — conteneurs OS très aboutis.
2005	**OpenVZ** (Virtuozzo/Parallels) — conteneurs Linux populaires en hébergement mutualisé.
2006-2008	**cgroups** (Google) intégrés au noyau 2.6.24.
2002-2013	Arrivée progressive des **namespaces** (mount → user).
2008	**LXC 0.1.0** — premiers conteneurs « purement noyau » (namespaces + cgroups), sans patch.
2013	**Docker** sort (mars). Au départ une surcouche de LXC, il démocratise le format d'image et le *workflow*.

Année	Jalon
2014	Docker remplace LXC par son propre runtime **libcontainer**. **Rocket/rkt** (CoreOS) émerge en réaction. **Kubernetes** est annoncé par Google.
2014	**LXD 0.1** (Canonical) — surcouche « *machine container* » au-dessus de LXC.
2015	Création de l'**OCI** (*Open Container Initiative*) : standardisation du format d'image et du runtime. **runc** est donné à l'OCI. **CNCF** fondée (héberge Kubernetes).
2016	**cgroup v2** stable (noyau 4.5). **containerd** extrait de Docker.
2018	**Podman 1.0** (Red Hat) — conteneurs *daemonless* et *rootless*. **PSI** ajoutée au noyau.
2019	Fedora 31 bascule sur cgroup v2. *rkt* abandonné.
2020	Kubernetes **déprécie Docker** comme runtime (au profit de containerd/CRI-O).
2021-2022	cgroup v2 par défaut partout. Podman 4.x, Docker rootless mûr.
2023	Canonical change la licence de LXD ; la communauté *fork* le projet sous le nom **Incus** (sous l'égide de Linux Containers).
2024-2025	**Incus** devient la référence communautaire ; Podman Desktop et Docker Desktop dominent les postes de développement.

Chapitre 4

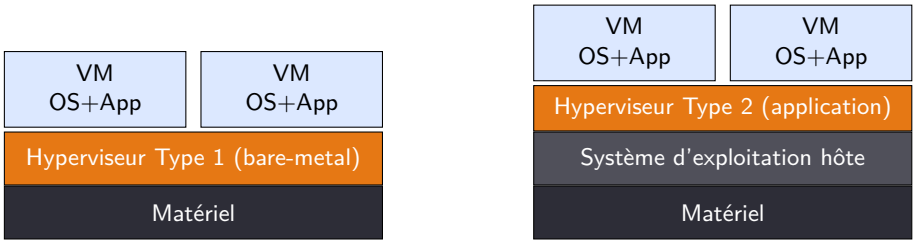
Les machines virtuelles (VM)

4.1 Principe et architecture

Une **machine virtuelle** est l'émulation/virtualisation d'un ordinateur complet : processeur, mémoire, disques, cartes réseau, BIOS/UEFI. Un **hyperviseur** (ou *Virtual Machine Monitor*, VMM) crée et gère ces VM, chacune exécutant son **propre noyau** et son **propre système d'exploitation invité** (*guest OS*), potentiellement différent de l'hôte (Windows sur Linux, etc.).

On distingue deux types d'hyperviseurs :

- **Type 1** — **« bare-metal »** : l'hyperviseur s'exécute directement sur le matériel, sans OS hôte intermédiaire. Exemples : **VMware ESXi**, **Microsoft Hyper-V**, **Xen**, **KVM** (techniquement un module du noyau Linux, donc hybride), **Proxmox VE** (KVM + LXC).
- **Type 2** — **« hosted »** : l'hyperviseur est une application au-dessus d'un OS hôte classique. Exemples : **VMware Workstation/Player**, **Oracle VirtualBox**, **QEMU** (en mode pur émulation), **Parallels Desktop**.



Type 1 : ESXi, Hyper-V, Xen, KVM **Type 2 : VirtualBox, VMware Workstation**

FIG. 4.1 : Hyperviseurs de type 1 (directement sur le matériel) et de type 2 (au-dessus d'un OS hôte).

4.2 La virtualisation matérielle assistée

Les processeurs modernes intègrent des extensions de virtualisation — **Intel VT-x** (avec EPT pour la mémoire) et **AMD-V** (avec RVI/NPT) — qui exécutent le code invité quasi nativement et n'interceptent (*trap*) que les instructions sensibles. C'est ce qui rend les VM modernes performantes (overhead souvent < 5–10 %). On parle parfois de **Ring -1** pour l'hyperviseur.

Compléments :

- **IOMMU** (Intel VT-d / AMD-Vi) : permet le *passthrough* sécurisé de périphériques (GPU, cartes réseau) à une VM (**PCI passthrough**, **SR-IOV**).
- **Paravirtualisation** : pilotes optimisés (**virtio** pour KVM/QEMU : `virtio-net`, `virtio-blk`,

`virtio-scsi`) où l'invité « sait » qu'il est virtualisé et coopère avec l'hôte, réduisant l'overhead d'E/S.

4.3 Le cas KVM/QEMU sous Linux

- **KVM** (*Kernel-based Virtual Machine*, depuis le noyau 2.6.20, 2007) transforme le noyau Linux lui-même en hyperviseur de type 1. Il expose `/dev/kvm` et délègue la virtualisation du CPU/mémoire au matériel.
- **QEMU** fournit l'émulation des périphériques (disques, réseau, USB, affichage) autour de KVM. C'est le couple **KVM+QEMU** qui constitue la pile de virtualisation libre de référence sous Linux.
- **libvirt** est la couche de gestion (API, `virsh`, fichiers XML), utilisée par `virt-manager`, **oVirt**, **Proxmox VE**, **OpenStack Nova**, etc.
- **Cloud-init** automatise la configuration au premier démarrage des VM.

4.4 Avantages des VM

- **Isolation la plus forte** : frontière au niveau matériel/hyperviseur, surface d'attaque réduite côté noyau partagé (chaque VM a son noyau).
- **Hétérogénéité des OS** : exécuter Windows, BSD, un autre noyau Linux, voire une autre architecture (avec émulation QEMU).
- **Maturité** des outils (snapshots, migration à chaud / *live migration*, haute disponibilité, sauvegarde cohérente).
- **Conformité réglementaire** : souvent exigée pour la multi-location (*multi-tenancy*) stricte.

4.5 Inconvénients et limitations des VM

- **Overhead** : chaque VM embarque un OS complet → consommation RAM/disque importante, démarrage lent.
- **Densité** plus faible que les conteneurs.
- **Duplication** : N noyaux, N jeux de bibliothèques système à patcher et maintenir.
- **Agilité** moindre : les images sont lourdes, le *workflow* de *build* moins fluide que les conteneurs.

4.6 Technologies de VM — panorama

Produit	Type	Éditeur	Licence	Remarques
KVM + QEMU	1 (hybride)	Communauté/Linux	GPL	Standard libre, base du *cloud* (OpenStack).
VMware ESXi / vSphere	1	Broadcom/VMware	Propriétaire	Référence entreprise historique.
Microsoft Hyper-V	1	Microsoft	Propriétaire	Intégré à Windows Server.
Xen	1	Communauté/XenProject	GPL	Para- et full-virtualisation, base d'AWS historique.

Produit	Type	Éditeur	Licence	Remarques
Proxmox VE	1	Proxmox	AGPL	KVM **et** conteneurs LXC, très populaire en PME.
Oracle VirtualBox	2	Oracle	GPL/PUEL	Poste de travail, multi-OS.
VMware Workstation/Fusion	2	Broadcom	Propriétaire (récemment gratuit usage perso)	Poste de travail.
Firecracker	« microVM »	AWS	Apache 2.0	microVM minimaliste (KVM) pour FaaS/Lambda, démarrage < 125 ms.

Note — la frontière s'estompe : des projets comme **Firecracker**, **Kata Containers** (conteneurs encapsulés dans des microVM légères) et **gVisor** (noyau invité en espace utilisateur, en Go, chez Google) visent à offrir « la sécurité d'une VM avec l'agilité d'un conteneur ». Kata, par exemple, présente une interface OCI (donc utilisable par Kubernetes/Docker) mais exécute chaque pod dans une VM KVM dédiée.

4.7 Liens de référence — VM / hyperviseurs

- KVM : https://linux-kvm.org/page/Main_Page
- QEMU (documentation) : https://www.qemu.org/docs/master/
- libvirt / virsh : https://libvirt.org/docs.html
- VMware vSphere / ESXi : https://docs.vmware.com/fr/VMware-vSphere/
- Microsoft Hyper-V : https://learn.microsoft.com/fr-fr/windows-server/virtualization/hyper-v/
- Xen Project : https://xenproject.org/
- Proxmox VE (documentation) : https://pve.proxmox.com/pve-docs/
- AWS Firecracker : https://github.com/firecracker-microvm/firecracker
- Kata Containers : https://github.com/kata-containers/kata-containers
- gVisor : https://gvisor.dev/docs/
- KVM dans le noyau Linux : https://www.kernel.org/doc/html/latest/virt/kvm/index.html

Chapitre 5

LXC — Linux Containers

5.1 Présentation

LXC (lancé en 2008 par Daniel Lezcano et Serge Hallyn, IBM) est **l'implémentation de référence des conteneurs « purement noyau »** : il fut le premier à combiner directement namespaces + cgroups + capabilities sans patch noyau externe. LXC produit ce qu'on appelle des **conteneurs système** (*system containers*) : un conteneur LXC se comporte comme une **machine légère** exécutant un *init* (souvent `systemd`) et plusieurs processus/services, à la manière d'une VM — par opposition au modèle « un conteneur = un processus » de Docker.

5.2 Architecture technique

- **liblxc** : la bibliothèque C cœur de LXC, qui orchestre namespaces, cgroups, capabilities, montages, réseau.
- **Outils CLI** : `lxc-create`, `lxc-start`, `lxc-attach`, `lxc-stop`, `lxc-ls`, `lxc-config`…
- **Templates** : scripts (puis images *download*) pour bâtir un rootfs de distribution (`lxc-create -t download -- -d debian -r bookworm -a amd64`).
- **Fichiers de configuration** déclaratifs par conteneur (`/var/lib/lxc/<nom>/config`) : réseau, montages, limites cgroup, *id mapping* pour les conteneurs non privilégiés.
- **LXCFS** : un système de fichiers FUSE crucial qui **virtualise `/proc`** (`/proc/cpuinfo`, `/proc/meminfo`, `/proc/stat`, `uptime`…) à l'intérieur du conteneur pour qu'il **reflète les limites cgroup** plutôt que les ressources de l'hôte. Sans LXCFS, `free` ou `nproc` dans le conteneur montreraient toute la RAM/tous les cœurs de l'hôte.

5.3 Conteneurs privilégiés vs non privilégiés

- **Privilégié** : *root* dans le conteneur = *root* (UID 0) dans l'hôte. Simple mais **dangereux** (une évasion = compromission de l'hôte).
- **Non privilégié** (*unprivileged*, recommandé) : s'appuie sur le **user namespace**. *root* dans le conteneur est mappé sur un UID non privilégié de l'hôte (ex. 100000–165535 via `/etc/subuid` et `/etc/subgid`). Une évasion ne donne que des droits d'utilisateur ordinaire sur l'hôte.

5.4 Avantages de LXC

- Conteneurs **système** complets et persistants (proches d'une VM mais légers).
- **Contrôle de bas niveau** très fin sur tous les mécanismes noyau.
- Pas de démon central obligatoire ; faible empreinte.
- Idéal pour héberger des **distributions complètes** durables.

5.5 Inconvénients et limitations de LXC

- **Courbe d'apprentissage** : configuration verbeuse, concepts noyau exposés.
- **Pas d'écosystème d'images** comparable à Docker Hub ; pas de *workflow* de *build* déclaratif (pas de `Dockerfile`).
- **Portabilité** moindre : un conteneur LXC est moins « jetable » et moins facilement distribuable qu'une image OCI.
- Outil de bas niveau : la plupart des utilisateurs préfèrent la surcouche **LXD/Incus**.

5.6 Liens de référence — LXC

- Site Linux Containers (LXC) : https://linuxcontainers.org/lxc/
- Documentation LXC : https://linuxcontainers.org/lxc/documentation/
- Dépôt liblxc (GitHub) : https://github.com/lxc/lxc
- LXCFS (virtualisation de `/proc`) : https://github.com/lxc/lxcfs
- Page de manuel `lxc.container.conf(5)` : https://linuxcontainers.org/lxc/manpages/man5/lxc.container.conf.5.html
- Forum communautaire : https://discuss.linuxcontainers.org/

Chapitre 6

LXD (et son successeur Incus)

6.1 Présentation

LXD (Canonical, 2014, prononcé « *lex-dee* ») est un **gestionnaire de conteneurs système et de VM** bâti **au-dessus de liblxc**. Il apporte ce qui manquait à LXC brut : une **API REST**, un **démon** (`lxd`), une expérience « machine » de haut niveau, la gestion d'images, le *clustering*, le stockage avancé (ZFS, Btrfs, LVM, Ceph) et le réseau (ponts, OVN).

Depuis **2023**, à la suite d'un changement de gouvernance et de licence imposé par Canonical, la communauté a *forké* LXD pour créer **Incus**, désormais hébergé par l'organisation *Linux Containers* (la même que LXC). Incus est aujourd'hui considéré comme le **successeur communautaire de fait** ; il est disponible dans Debian 12+, et la commande historique `lxc` y devient `incus`.

6.2 Architecture technique

- **Démon `lxd`/`incusd`** exposant une **API REST** (sur socket Unix et, optionnellement, sur le réseau avec TLS).
- **Client CLI `lxc`/`incus`** (à ne pas confondre avec les outils `lxc-*` de LXC bas niveau !).
- **Conteneurs système** (via liblxc) **et VM** (via QEMU/KVM) gérés par la **même** interface et le **même** *workflow* — un atout majeur.
- **Pools de stockage** : ZFS (recommandé), Btrfs, LVM, *dir*, Ceph RBD ; *snapshots* et clones instantanés (CoW).
- **Réseau** : ponts managés, **OVN** (*Open Virtual Network*) pour le SDN en cluster, *forwards*, ACL.
- ***Clustering*** : plusieurs hôtes formant un cluster avec base de données distribuée (Dqlite/Raft).
- **Migration à chaud** des conteneurs (via CRIU) et des VM.
- **Profils** réutilisables décrivant ressources, réseau et stockage.

6.3 Avantages de LXD/Incus

- Expérience **« VM-like »** mais avec la légèreté des conteneurs : idéal pour remplacer des VM de longue durée.
- **Conteneurs ET machines virtuelles** unifiés sous une seule API/CLI.
- *Snapshots*, clones, migration, *clustering*, stockage et réseau de niveau « production ».
- Très bon pour les **laboratoires**, l'**hébergement multi-tenant interne**, les **environnements de développement** persistants.

6.4 Inconvénients et limitations

- Orienté **conteneurs système / VM** : moins adapté au modèle « micro-service jetable » et au *pipeline* CI/CD que Docker/OCI.
- **Pas le standard du *cloud*** : Kubernetes et l'écosystème OCI dominent les déploiements d'applications cloud-native ; LXD/Incus y a peu de place.
- L'épisode de licence **LXD vs Incus** a fragmenté l'écosystème (choisir Incus pour la voie communautaire).
- Moins connu et moins documenté côté entreprise que Docker/Kubernetes.

6.5 Liens de référence — LXD / Incus

- Incus (site officiel, successeur communautaire) : https://linuxcontainers.org/incus/
- Documentation Incus : https://linuxcontainers.org/incus/docs/main/
- Dépôt Incus (GitHub) : https://github.com/lxc/incus
- LXD (Canonical, historique) : https://canonical.com/lxd
- Documentation LXD : https://documentation.ubuntu.com/lxd/
- Comparatif et migration LXD → Incus : https://linuxcontainers.org/incus/docs/main/howto/migrate_from_lxd/

Chapitre 7

Docker

7.1 Présentation

Docker (dotCloud → Docker Inc., 2013) est la technologie qui a **popularisé** les conteneurs en standardisant trois choses : le **format d'image** (couches, manifeste), le **`Dockerfile`** (description reproductible du *build*) et un **registre** d'images (**Docker Hub**). Docker promeut le modèle **« un conteneur = un processus/service applicatif jetable »** (*application containers*), à l'opposé du conteneur système de LXC/LXD.

À ses débuts (2013), Docker s'appuyait sur LXC ; dès 2014 il l'a remplacé par son propre runtime **libcontainer**, puis a contribué à la standardisation **OCI** (2015).

7.2 Architecture technique (pile actuelle)

Docker a éclaté son monolithe en couches conformes à l'OCI :

```
docker (CLI)
   │  (API REST)
   ▼
dockerd  ──────────────  le démon (build, réseau, volumes, API)
   │
   ▼
containerd  ────────────  gestionnaire de cycle de vie des conteneurs
   │                       (pull d'images, supervision)
   ▼
containerd-shim  ───────  découple le conteneur du démon
   │
   ▼
runc  ──────────────────  runtime OCI bas niveau :
                          crée namespaces + cgroups + monte le rootfs,
                          puis exec() le processus du conteneur
```

- **`dockerd`** : démon central (souvent critiqué : *single point of failure*, tourne en *root* par défaut).
- **`containerd`** : runtime de haut niveau (projet CNCF), gère images et cycle de vie ; aussi utilisé directement par Kubernetes (via CRI).
- **`runc`** : runtime OCI bas niveau de référence (issu de libcontainer) qui interagit réellement avec le noyau.
- **`containerd-shim`** : maintient le conteneur en vie même si `dockerd` redémarre.
- **BuildKit** : moteur de *build* moderne (cache avancé, builds parallèles, *multi-stage*, multi-arch via `buildx`).

- **Réseau** : pilotes `bridge` (défaut, NAT via `iptables`/`nftables`), `host`, `macvlan`, `overlay` (multi-hôte, Swarm).
- **Volumes** : stockage persistant hors couches d'image (*bind mounts*, *named volumes*, pilotes).
- **Docker Compose** : orchestration **mono-hôte** déclarative en YAML.
- **Docker Swarm** : orchestration **multi-hôte** native (aujourd'hui largement supplantée par Kubernetes).

7.3 Format d'image et OCI

Une image Docker est un empilement de **couches** identifiées par leur *digest* (SHA-256), décrit par un **manifeste** et une **configuration**. Le **`Dockerfile`** décrit le build instruction par instruction (`FROM`, `RUN`, `COPY`, `CMD`…), chaque instruction produisant idéalement une couche cacheable. La standardisation **OCI** (*image-spec*, *runtime-spec*, *distribution-spec*) garantit que ces images sont interopérables entre Docker, Podman, containerd, Kubernetes, etc.

7.4 Mode *rootless*

Depuis la version 20.10, Docker propose un mode **rootless** : `dockerd` et les conteneurs tournent sous un utilisateur non privilégié grâce au user namespace, `slirp4netns`/`pasta` pour le réseau et `fuse-overlayfs` pour le stockage. Cela réduit fortement la surface d'attaque, mais avec quelques limitations (réseau, ports < 1024, performances d'E/S).

7.5 Avantages de Docker

- **Écosystème immense** : Docker Hub, des millions d'images, tutoriels, intégration CI/CD universelle.
- **Workflow de *build*** reproductible et standardisé (`Dockerfile`, BuildKit, multi-arch).
- **Portabilité** « *build once, run anywhere* » via le format OCI.
- **Docker Desktop** : expérience clé-en-main sur Windows/macOS (via une VM Linux légère).
- **Compose** : démarrage rapide d'applications multi-conteneurs en local.

7.6 Inconvénients et limitations de Docker

- **Démon central `dockerd`** : SPOF et, par défaut, **exécution en root** (le groupe `docker` équivaut à un accès root sur l'hôte — risque de sécurité majeur).
- Modèle **mono-processus** : peu adapté aux conteneurs « système » persistants (préférer LXD/Incus pour cela).
- **Kubernetes a déprécié Docker** comme runtime (depuis K8s 1.20) au profit de containerd/CRI-O — Docker reste excellent pour le *build* et le développement local, moins comme runtime de production orchestré.
- Changements de **licence de Docker Desktop** (payant pour les grandes entreprises depuis 2021) ayant poussé beaucoup vers Podman.
- *Rootless* encore moins mûr que la solution équivalente de Podman.

7.7 Liens de référence — Docker

- Documentation officielle : https://docs.docker.com/
- Référence du Dockerfile : https://docs.docker.com/reference/dockerfile/
- Docker Compose : https://docs.docker.com/compose/
- BuildKit / buildx : https://docs.docker.com/build/buildkit/
- Mode rootless : https://docs.docker.com/engine/security/rootless/

- containerd : https://containerd.io/ — runc : https://github.com/opencontainers/runc
- Docker Hub : https://hub.docker.com/
- Bonnes pratiques de sécurité : https://docs.docker.com/engine/security/

Chapitre 8

Podman

8.1 Présentation

Podman (*Pod Manager*, Red Hat, 1.0 en 2018) est un moteur de conteneurs **compatible OCI** conçu comme une **alternative à Docker** avec deux différences philosophiques majeures :

1. **Sans démon** (*daemonless*) : il n'y a **pas** de processus central long comme `dockerd`. Chaque conteneur est un processus enfant **direct** de la commande `podman`, supervisé par un petit moniteur, **`conmon`**.
2. **Rootless par conception** : l'exécution sans privilèges est le mode *natif* et privilégié, pas une option ajoutée a posteriori.

Podman se veut un **remplacement « *drop-in* »** de Docker : la CLI est quasi identique (`alias docker=podman` fonctionne dans la grande majorité des cas), et il consomme/produit des images OCI compatibles Docker Hub.

8.2 Architecture technique

```
podman (CLI, sans démon)
   │
   ├── conmon  ──── moniteur léger par conteneur (logs, TTY, exit code)
   │
   └── crun / runc ──── runtime OCI bas niveau
                        (crun, en C, est plus léger et plus rapide
                         que runc, écrit en Go ; défaut sur Fedora/RHEL)
```

- **Pas de démon** : meilleure intégration à **systemd** (chaque conteneur peut être une unité systemd ; `podman generate systemd`, et désormais les fichiers **Quadlet** `.container`).
- **`crun`** : runtime OCI écrit en C, plus rapide et plus économe que `runc`, et meilleure prise en charge de cgroup v2.
- **Notion de *pod*** : Podman implémente nativement le concept de **pod** de Kubernetes (plusieurs conteneurs partageant les mêmes namespaces réseau/IPC). `podman generate kube` / `podman play kube` permettent de produire et consommer des manifestes **Kubernetes YAML** — pont direct vers K8s.
- **`buildah`** : outil dédié à la **construction** d'images (Podman s'en sert pour `podman build`). Il permet des builds sans `Dockerfile`, scriptés et *rootless*.
- **`skopeo`** : outil dédié à l'**inspection** et au **transfert** d'images entre registres sans les exécuter.
- **Compatibilité Docker Compose** : Podman expose une **socket compatible API Docker** permettant d'utiliser `docker-compose`/`podman compose`.

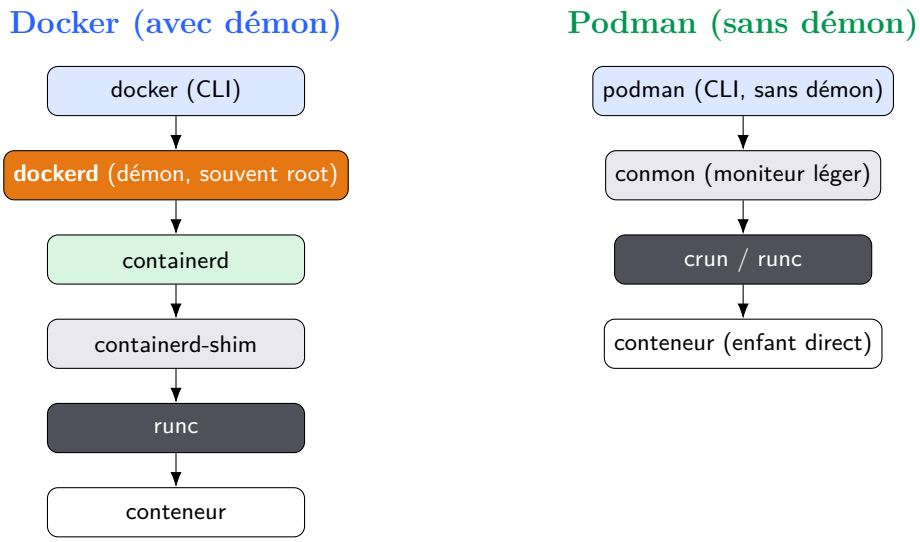


FIG. 8.1 : Docker repose sur un démon central persistant (`dockerd`) ; Podman lance chaque conteneur comme un processus enfant direct, supervisé par `conmon`, sans démon central.

8.3 Avantages de Podman

- **Sécurité** : *rootless* et *daemonless* par défaut → surface d'attaque réduite, pas de démon root, isolation utilisateur native.
- **Intégration systemd** excellente (Quadlet) pour des conteneurs gérés comme des services système — robustesse en production sur un hôte unique.
- **Compatibilité Docker** quasi totale (CLI, images OCI) → migration facile.
- **Pods** et export/import **Kubernetes YAML** : excellent pont vers K8s.
- Suite d'outils Unix-philosophie (`buildah`, `skopeo`) modulaire.
- Licence entièrement libre, pas de question d'abonnement.

8.4 Inconvénients et limitations de Podman

- **Pas de démon** = pas d'état central : certaines fonctionnalités (redémarrage automatique au boot, supervision) reposent sur systemd, ce qui dépayse les habitués de Docker.
- Écosystème, documentation grand public et nombre de tutoriels **plus réduits** que Docker (même si en forte croissance).
- Support **macOS/Windows** via une VM (Podman Machine) : fonctionnel mais historiquement moins poli que Docker Desktop (l'écart se réduit avec Podman Desktop).
- Quelques **incompatibilités résiduelles** de la couche compose et de certaines options Docker avancées.
- Le mode *rootless* impose ses contraintes (réseau via `pasta`/`slirp4netns`, ports privilégiés, E/S `fuse-overlayfs`).

8.5 Liens de référence — Podman

- Site officiel : https://podman.io/ — Documentation : https://docs.podman.io/
- Dépôt (GitHub) : https://github.com/containers/podman
- Buildah (construction d'images) : https://buildah.io/
- Skopeo (transfert/inspection d'images) : https://github.com/containers/skopeo
- crun (runtime OCI en C) : https://github.com/containers/crun
- Quadlet (intégration systemd) : https://docs.podman.io/en/latest/markdown/podman-

systemd.unit.5.html

- Podman Desktop : https://podman-desktop.io/

Chapitre 9

Autres technologies de conteneurisation utiles

Au-delà des cinq piliers étudiés, l'écosystème compte de nombreuses technologies qui méritent l'attention selon les besoins (HPC, sécurité, edge, postes de travail, standards bas niveau). En voici un panorama.

9.1 Runtimes et moteurs de bas niveau

- **containerd** (CNCF) — Runtime de haut niveau extrait de Docker, utilisé directement par Kubernetes via CRI. Gère images, stockage et cycle de vie. C'est le « moteur » sous Docker et sous de nombreux clusters K8s. https://containerd.io/
- **CRI-O** (CNCF / Red Hat) — Runtime minimal dédié à Kubernetes, conforme CRI et OCI, sans fonctionnalités superflues. Standard sur OpenShift. https://cri-o.io/
- **runc** — Runtime OCI de référence (en Go), créé par Docker, donné à l'OCI. https://github.com/opencontainers/runc
- **crun** — Alternative à runc écrite en C, plus rapide et plus légère, excellente prise en charge de cgroup v2. https://github.com/containers/crun
- **youki** — Runtime OCI écrit en **Rust**, axé performance et sécurité mémoire. https://github.com/youki-dev/youki

9.2 Conteneurs « système » et historiques

- **systemd-nspawn** — « `chroot` sous stéroïdes » fourni avec systemd : lance un conteneur système ou démarre un OS complet à partir d'un répertoire/image, idéal pour le test et le débogage. Géré via `machinectl`. https://www.freedesktop.org/software/systemd/man/latest/systemd-nspawn.html
- **OpenVZ / Virtuozzo** — Conteneurs système Linux historiques, très utilisés en hébergement mutualisé (VPS) avant LXC. https://openvz.org/
- **Linux-VServer** — L'un des tout premiers systèmes d'isolation Linux (par *patch* noyau), précurseur des namespaces.
- **FreeBSD Jails** — Isolation OS native de FreeBSD (depuis 2000), ancêtre conceptuel des conteneurs. https://docs.freebsd.org/en/books/handbook/jails/
- **Solaris Zones / illumos Zones** — Conteneurs OS très aboutis de Solaris/illumos. https://illumos.org/man/5/zones
- **rkt (Rocket)** — Moteur de CoreOS (2014-2019), concurrent de Docker, *daemonless*, aujourd'hui **abandonné** mais historiquement influent.

9.3 HPC et calcul scientifique

- **Apptainer** (ex-**Singularity**) — Conteneurs conçus pour le **calcul haute performance** (HPC) et la science : exécution sans privilèges, pas de démon, format d'image *single-file* (SIF), intégration MPI/GPU. Le standard de fait sur les supercalculateurs. https://apptainer.org/
- **Sarus / Shifter** — Runtimes de conteneurs orientés HPC, compatibles Docker, utilisés dans les centres de calcul.

9.4 Sécurité renforcée (isolation forte)

- **Kata Containers** — Conteneurs OCI exécutés dans des microVM légères (KVM) : sécurité d'une VM, interface d'un conteneur. https://katacontainers.io/
- **gVisor** (Google) — Noyau invité réimplémenté en espace utilisateur (Go) interceptant les appels système ; runtime `runsc`. https://gvisor.dev/
- **Firecracker** (AWS) — microVM minimaliste pour FaaS (base d'AWS Lambda et Fargate), démarrage < 125 ms. https://firecracker-microvm.github.io/
- **Sysbox** (Nestybox/Docker) — Runtime OCI permettant des conteneurs « comme des VM » (systemd, Docker-in-Docker sans privilèges). https://github.com/nestybox/sysbox

9.5 Postes de travail et applications « sandboxées »

Ces technologies isolent des **applications de bureau** plutôt que des services serveur, mais reposent sur les mêmes briques noyau (namespaces, cgroups, seccomp) :

- **Flatpak** — Distribution et bac à sable d'applications de bureau Linux (portail xdg, bubblewrap). https://flatpak.org/
- **Snap** (Canonical) — Paquets confinés multi-distributions, avec démon `snapd` et confinement AppArmor. https://snapcraft.io/
- **AppImage** — Application portable en un seul fichier (sans isolation forte, mais autonome). https://appimage.org/
- **Bubblewrap (bwrap)** — Outil de bas niveau de création de bacs à sable sans privilèges, brique de Flatpak. https://github.com/containers/bubblewrap
- **WSL 2** (Windows Subsystem for Linux) — Exécute un vrai noyau Linux dans une VM légère sous Windows ; support des conteneurs Docker/Podman. https://learn.microsoft.com/fr-fr/windows/wsl/

9.6 OS minimalistes orientés conteneurs

- **Bottlerocket** (AWS) — OS Linux minimal dédié à l'exécution de conteneurs. https://aws.amazon.com/fr/bottlerocket/
- **Flatcar Container Linux** — Successeur communautaire de CoreOS Container Linux. https://www.flatcar.org/
- **Talos Linux** — OS minimal et immuable entièrement dédié à Kubernetes, géré uniquement par API. https://www.talos.dev/
- **k3OS / Fedora CoreOS** — Distributions immuables pensées pour les clusters.

9.7 Orchestrateurs et outils complémentaires

- **Kubernetes** : https://kubernetes.io/fr/
- **HashiCorp Nomad** : https://www.nomadproject.io/
- **k3s / RKE2** (Kubernetes léger, Rancher/SUSE) : https://k3s.io/
- **OpenShift** (Red Hat) : https://www.redhat.com/fr/technologies/cloud-computing/openshift

- **Nerdctl** (CLI type Docker pour containerd) : https://github.com/containerd/nerdctl

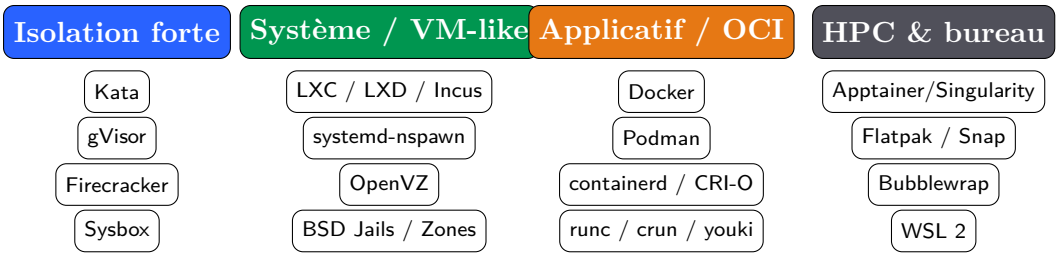


FIG. 9.1 : Cartographie des technologies de conteneurisation par finalité.

Chapitre 10

Comparaison complète des technologies

10.1 Tableau de synthèse général

Critère	VM (KVM/ESXi…)	LXC	LXD / Incus	Docker	Podman
Type d'isolation	Matérielle (hyperviseur)	OS (namespaces+cgroups)	OS (+VM)	OS	OS
Noyau	Un par VM	Partagé	Partagé (VM possible)	Partagé	Partagé
Modèle	Machine complète	Conteneur **système**	Conteneur système + **VM**	Conteneur **applicatif**	Conteneur applicatif (+ pods)
Démon central	Hyperviseur	Non (liblxc)	Oui (`lxd`/`incusd`)	Oui (`dockerd`)	**Non** (daemonless)
Rootless natif	N/A	Oui (unprivileged)	Partiel	Optionnel (récent)	**Oui (par défaut)**
Format d'image	Image disque (qcow2…)	rootfs/template	Images LXD	**OCI**	**OCI**
Build déclaratif	Packer, cloud-init	Templates	Profils	**Dockerfile** + BuildKit	Dockerfile, buildah
Écosystème d'images	Modéré	Faible	Modéré	**Immense (Hub)**	OCI (compatible Hub)
Orchestration	vSphere, OpenStack, Proxmox	—	Clustering intégré	Swarm / **Kubernetes**	**Kubernetes** (pods/YAML)
Démarrage	Lent (s à dizaines de s)	Rapide	Rapide	Très rapide	Très rapide
Densité par hôte	Faible (dizaines)	Élevée	Élevée	Très élevée	Très élevée
Empreinte	Go	Mo	Mo	Mo	Mo
Niveau d'isolation	**Le plus fort**	Moyen	Moyen	Moyen	Moyen (+ rootless)

Critère	VM (KVM/ESXi…)	LXC	LXD / Incus	Docker	Podman
Cas type	Multi-OS, multi-tenant strict	Conteneurs OS durables	« VM légères » + VM	Micro-services, CI/CD	Idem Docker, axé sécurité

10.2 Conteneur « système » vs conteneur « applicatif »

C'est la distinction conceptuelle la plus importante :

- **Conteneur système** (LXC, LXD/Incus) : se comporte comme une **machine légère**. Il exécute un `init` (`systemd`), plusieurs services, est **persistant**, et on s'y connecte comme à un serveur. On y traite le conteneur comme un « animal de compagnie » (*pet*).
- **Conteneur applicatif** (Docker, Podman) : encapsule **un service** et ses dépendances, est **immuable et jetable** (*cattle*), reconstruit à partir d'une image à chaque déploiement, stockage externalisé dans des volumes.

10.3 Sécurité comparée

Du plus isolé au moins isolé (en configuration par défaut courante) :

1. **VM** — frontière matérielle, noyau séparé. *Référence d'isolation.*
2. **Kata Containers / Firecracker / gVisor** — « conteneurs » mais exécutés dans une microVM ou avec un noyau invité en espace utilisateur.
3. **Conteneurs rootless** (Podman, Docker rootless, LXC unprivileged) — user namespace : une évasion ne donne pas *root* sur l'hôte.
4. **Conteneurs root** (Docker/dockerd par défaut, LXC privilégié) — partage du noyau **et** root réel : une évasion compromet l'hôte.

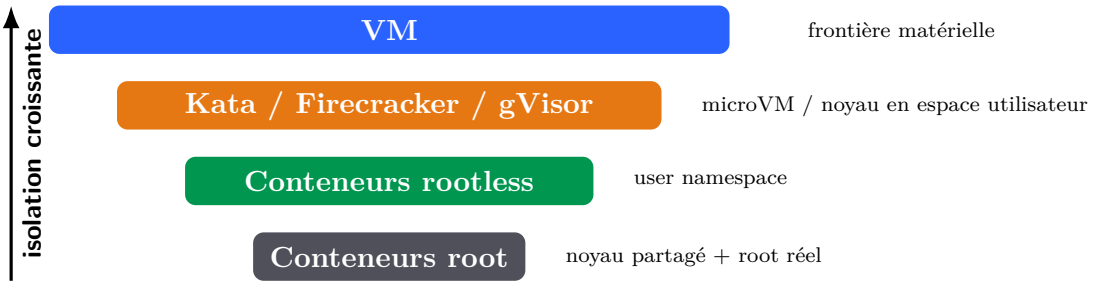


FIG. 10.1 : Échelle d'isolation, de la plus forte (VM) à la plus faible (conteneurs privilégiés en root).

Règles d'or de durcissement, valables pour tous les conteneurs : user namespaces, `--cap-drop=ALL` puis ajout minimal, profil **seccomp**, **SELinux/AppArmor** activé, `--read-only` + `tmpfs`, `--security-opt=no-new-privileges`, limites **cgroup** systématiques, et **ne jamais** monter `/var/run/docker.sock` dans un conteneur.

Chapitre 11

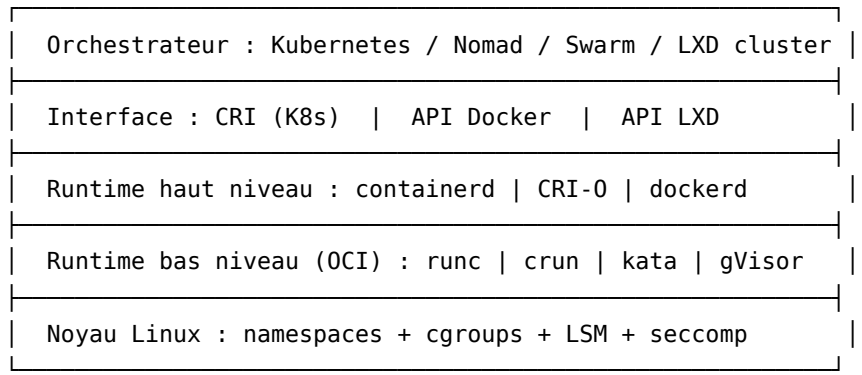
Les *stacks* et l'écosystème

11.1 La pile de standards OCI / CNCF

L'écosystème moderne repose sur des standards ouverts :

- **OCI** (*Open Container Initiative*, 2015) : trois spécifications — **image-spec** (format d'image), **runtime-spec** (comment exécuter un *bundle*) et **distribution-spec** (protocole des registres).
- **CRI** (*Container Runtime Interface*) : l'API par laquelle Kubernetes parle à un runtime (containerd, CRI-O).
- **CNI** (*Container Network Interface*) : standard de *plugins* réseau (Calico, Cilium, Flannel…).
- **CSI** (*Container Storage Interface*) : standard de *plugins* de stockage.

11.2 La pile « runtime » en couches



11.3 Orchestration : Kubernetes et alternatives

- **Kubernetes (K8s)** : l'orchestrateur de référence (planification, *self-healing*, *scaling*, *rolling updates*, *service discovery*). Il parle aux nœuds via **CRI** (containerd ou CRI-O) — **plus** via Docker depuis la 1.24. Distributions : vanilla, **OpenShift** (Red Hat), **Rancher/RKE2**, **k3s** (edge/léger), managées (**EKS**, **GKE**, **AKS**).
- **Nomad** (HashiCorp) : orchestrateur plus simple, multi-charges (conteneurs, binaires, VM).
- **Docker Swarm** : intégré à Docker, simple, mais en perte de vitesse.
- **LXD/Incus clustering** : orchestration de conteneurs **système** et VM, pas d'applications cloud-native.

11.4 Outils périphériques courants

- **Registres** : Docker Hub, Harbor, Quay, GitLab Container Registry, GHCR.
- **Build** : BuildKit/buildx, Buildah, Kaniko, Jib.
- **Sécurité/scan** : Trivy, Grype, Clair, Falco (détection runtime), Cosign (signature).
- **Développement local** : Docker Desktop, Podman Desktop, Rancher Desktop, minikube, kind, k3d.

Chapitre 12

Ressources, performances et empreinte

12.1 Ce que consomme réellement chaque technologie

Ressource	VM	Conteneur (LXC/LXD/Docker/Podman)
Mémoire de base	OS invité complet (centaines de Mo à Go) + overhead hyperviseur	Seulement les processus applicatifs (Mo)
CPU au repos	`kswapd`, démons de l'OS invité, *timer ticks*	Quasi nul (processus seulement)
Disque	Image complète (Go), peu de partage	Couches partagées (CoW), Mo par conteneur
Démarrage	Boot complet (BIOS→noyau→init→services)	`fork/exec` + montage overlay (ms à s)
Densité	Dizaines/hôte	Centaines à milliers/hôte
Overhead CPU/mémoire	~2–10 % (matériel assisté)	~0–2 % (quasi natif)
Overhead E/S réseau/disque	virtio réduit l'écart, mais présent	Quasi natif

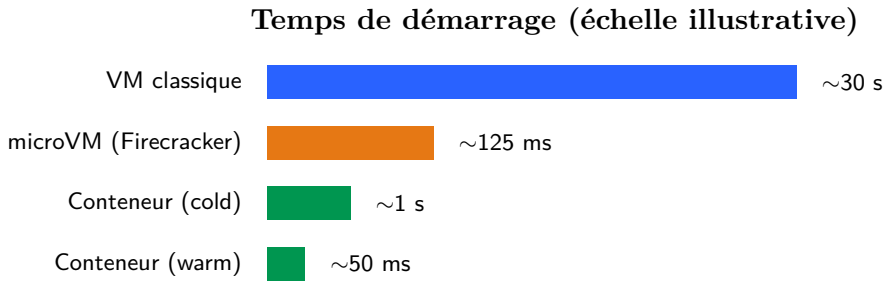


FIG. 12.1 : Ordre de grandeur du temps de démarrage : les conteneurs démarrent en millisecondes là où une VM doit amorcer un OS complet (les microVM comblent l'écart pour la sécurité).

12.2 Mesurer et limiter les ressources

- **Voir** la consommation : `docker stats`, `podman stats`, `systemd-cgtop`, `lxc-info`, `incus info`, et la lecture directe des fichiers cgroup v2 (`memory.current`, `cpu.stat`).
- **Limiter** (exemples Docker/Podman, qui se traduisent en cgroup v2) :

```
# 1,5 cœur, 512 Mo de RAM, 100 processus max, 50 Mo/s en écriture disque
podman run --cpus=1.5 --memory=512m --pids-limit=100 \
           --device-write-bps=/dev/sda:50mb  mon-image
```

- **Le piège classique** : à l'intérieur d'un conteneur, `nproc`, `free`, `/proc/cpuinfo`/`meminfo` montrent l'**hôte**, pas les limites cgroup. Conséquences réelles : la **JVM** ou les runtimes Go/Node mal configurés dimensionnent leurs pools de threads/tas sur les ressources de l'hôte. Remèdes : LXCFS (LXC/LXD), variables d'environnement / options de runtime conscientes des cgroups (la JVM moderne lit `cpu.max` et `memory.max`).

12.3 Performances : ordre de grandeur

- **CPU/mémoire** : conteneurs proches du natif ; VM à environ 95–98 % du natif.
- **Démarrage** : conteneur en millisecondes ; microVM (Firecracker) ~125 ms ; VM classique en secondes.
- **Réseau** : le mode `bridge`/NAT de Docker ajoute un saut `iptables` ; `host`, `macvlan` ou CNI performants (Cilium/eBPF) réduisent l'overhead.
- **Disque** : OverlayFS est excellent en lecture ; les écritures intensives gagnent à passer par des **volumes** dédiés plutôt que par la couche CoW.

Chapitre 13

Recommandations par cas d'usage

Besoin	Recommandation
Exécuter Windows ou un autre OS / multi-tenant strict	**VM** (KVM, ESXi, Proxmox)
« VM légères » Linux persistantes, labo, hébergement interne	**LXD / Incus**
Contrôle noyau fin, conteneurs système sur mesure	**LXC**
Micro-services, CI/CD, développement applicatif, portabilité	**Docker** (build/local) + **Kubernetes** (prod)
Conteneurs sécurisés *rootless*, intégration systemd, alternative Docker	**Podman** (+ Quadlet)
Sécurité de VM avec agilité conteneur (FaaS, multi-tenant)	**Kata Containers / Firecracker / gVisor**
Orchestration cloud-native à l'échelle	**Kubernetes** (containerd/CRI-O)
PME : VM + conteneurs sur un même hôte avec interface unique	**Proxmox VE** (KVM + LXC)

Conseil de synthèse : ces technologies ne sont pas exclusives mais **complémentaires**. Une architecture réelle combine très souvent des VM (socle d'isolation forte et de multi-location), des conteneurs applicatifs (Docker/Podman + OCI) orchestrés par Kubernetes, et parfois des conteneurs système (LXD/Incus) pour les services *stateful* de longue durée — le tout reposant, au plus bas niveau, sur les **mêmes cgroups et namespaces** du noyau Linux.

Chapitre 14

Lexique des termes techniques (français)

AppArmor — Module de sécurité Linux (LSM) appliquant un contrôle d'accès obligatoire par profils, par défaut sur Debian/Ubuntu/SUSE.

Capability (capacité) — Sous-ensemble atomique des privilèges de *root*, attribuable ou retirable individuellement à un processus.

cgroup (groupe de contrôle) — Mécanisme du noyau organisant les processus en hiérarchie pour limiter, prioriser et comptabiliser l'usage des ressources (CPU, mémoire, E/S…).

CNI (Container Network Interface) — Standard de *plugins* réseau pour conteneurs et orchestrateurs.

Conteneur — Ensemble de processus isolés partageant le noyau de l'hôte, construit avec namespaces, cgroups et mécanismes de sécurité.

Conteneur applicatif — Conteneur encapsulant un seul service, immuable et jetable (modèle Docker/Podman).

Conteneur système — Conteneur se comportant comme une machine légère exécutant un `init` et plusieurs services (modèle LXC/LXD).

Copy-on-write (CoW, copie sur écriture) — Technique où une donnée n'est copiée que lorsqu'elle est modifiée ; base des couches d'images et des *snapshots*.

CRI (Container Runtime Interface) — API par laquelle Kubernetes communique avec un runtime de conteneurs.

CRIU (Checkpoint/Restore In Userspace) — Outil de gel/restauration de l'état de processus, utilisé pour la migration à chaud de conteneurs.

crun / runc — Runtimes OCI de bas niveau qui créent réellement les conteneurs (crun en C, runc en Go).

Daemonless (sans démon) — Architecture sans processus central long (Podman), où chaque conteneur est un enfant direct de la commande.

Dockerfile — Fichier texte décrivant, instruction par instruction, la construction reproductible d'une image.

Hyperviseur (VMM) — Logiciel créant et gérant des machines virtuelles (type 1 « bare-metal » ou type 2 « hosted »).

IOMMU (VT-d / AMD-Vi) — Unité matérielle permettant le *passthrough* sécurisé de périphériques à une VM.

KVM (Kernel-based Virtual Machine) — Module du noyau Linux transformant celui-ci en hyperviseur, s'appuyant sur VT-x/AMD-V.

LSM (Linux Security Modules) — Cadre du noyau pour les modules de sécurité (SELinux, AppArmor).

LXCFS — Système de fichiers FUSE qui virtualise `/proc` dans un conteneur pour refléter les limites cgroup au lieu des ressources de l'hôte.

Namespace (espace de noms) — Mécanisme du noyau isolant la *visibilité* d'une ressource globale (PID, réseau, montages, utilisateurs…).

OCI (Open Container Initiative) — Organisme de standardisation du format d'image, du runtime et de la distribution des conteneurs.

OOM-killer (Out Of Memory killer) — Mécanisme du noyau qui tue des processus en cas d'épuisement mémoire ; agit par cgroup avec `memory.max`.

OverlayFS — Système de fichiers union du noyau empilant des couches (lecture seule + lecture/écriture) avec copie sur écriture.

Paravirtualisation — Virtualisation où l'invité coopère avec l'hyperviseur via des pilotes optimisés (ex. virtio).

pivot_root — Appel système changeant la racine du système de fichiers d'un processus, plus sûr que `chroot`, utilisé au démarrage d'un conteneur.

Pod — Groupe de conteneurs partageant certains namespaces (réseau, IPC) ; concept Kubernetes, implémenté nativement par Podman.

PSI (Pressure Stall Information) — Métriques noyau indiquant le temps pendant lequel des tâches sont bloquées faute de CPU, mémoire ou E/S.

Rootless (sans privilèges) — Exécution de conteneurs par un utilisateur non privilégié grâce au user namespace.

seccomp (secure computing mode) — Filtrage des appels système autorisés à un processus via des profils BPF.

SELinux — Module de sécurité Linux (LSM) à contrôle d'accès obligatoire, par défaut sur Red Hat/Fedora.

Slice / Scope / Service (systemd) — Types d'unités systemd organisant l'arborescence cgroup et les limites de ressources.

SR-IOV — Technologie permettant à un périphérique PCIe de se présenter comme plusieurs fonctions virtuelles partageables entre VM.

Throttling (étranglement) — Limitation active du CPU imposée par le cgroup lorsque le quota d'une période est atteint.

User namespace — Namespace mappant les UID/GID du conteneur sur d'autres UID/GID de l'hôte ; fondement des conteneurs *rootless*.

virtio — Famille de pilotes paravirtualisés (réseau, bloc, SCSI) pour KVM/QEMU réduisant l'overhead d'E/S.

VM (machine virtuelle) — Ordinateur complet virtualisé exécutant son propre noyau et OS au-dessus d'un hyperviseur.

Chapitre 15

Bibliographie en français

Les éditions et années indiquées peuvent varier selon les réimpressions ; vérifier la dernière édition disponible.

- **« Docker — Prise en main et mise en pratique »**, divers auteurs, Éditions ENI — collection d'ouvrages d'introduction et de pratique en français.
- **« Docker — Pratique des architectures à base de conteneurs »**, Pierre-Yves Cloux, Thomas Garlot, Johann Kohler, *Dunod*.
- **« Kubernetes — Gérez vos plateformes de conteneurs »** et **« Kubernetes — Maîtrisez l'orchestrateur... »**, Éditions ENI.
- **« Linux — Maîtrisez l'administration du système »**, Sébastien Rohaut, Éditions ENI — chapitres sur les processus, systemd et les cgroups.
- **« Le système Linux »** (édition française de *Running Linux*), O'Reilly — fondamentaux du noyau et de l'espace utilisateur.
- **« Administration Linux par la pratique »**, Éditions ENI — virtualisation KVM, conteneurs.
- **« Virtualisation et conteneurs »** et titres associés des collections *ENI* et *Dunod* (KVM, Proxmox, Docker, Kubernetes).
- **Cahiers de l'Admin — Debian / Ubuntu »**, Raphaël Hertzog, Roland Mas, *Eyrolles* — sections virtualisation (KVM, LXC) et conteneurs.
- **« Proxmox VE »** — guides francophones (ENI, autoédition) sur la plateforme KVM + LXC.
- **Documentation francophone de l'April, de Linux-France et du site *it-connect.fr*** — nombreux tutoriels approfondis en français sur Docker, Podman, LXC/LXD, cgroups et KVM.

Chapitre 16

Liens utiles sur Internet

16.1 Documentations officielles

- Noyau Linux — cgroup v2 : https://docs.kernel.org/admin-guide/cgroup-v2.html
- Noyau Linux — namespaces (`man7`) : https://man7.org/linux/man-pages/man7/namespaces.7.html
- Noyau Linux — `cgroups(7)` : https://man7.org/linux/man-pages/man7/cgroups.7.html
- Noyau Linux — `user_namespaces(7)` : https://man7.org/linux/man-pages/man7/user_namespaces.7.html
- Documentation PSI : https://docs.kernel.org/accounting/psi.html
- Docker — documentation officielle : https://docs.docker.com/
- Podman — site et documentation : https://podman.io/ et https://docs.podman.io/
- LXC / LXD — Linux Containers : https://linuxcontainers.org/
- Incus (successeur de LXD) : https://linuxcontainers.org/incus/
- Open Container Initiative (OCI) : https://opencontainers.org/
- Kubernetes : https://kubernetes.io/fr/ (documentation traduite en français)
- KVM : https://linux-kvm.org/ — QEMU : https://www.qemu.org/ — libvirt : https://libvirt.org/
- Proxmox VE : https://www.proxmox.com/fr/proxmox-virtual-environment
- systemd (cgroups, slices) : https://www.freedesktop.org/software/systemd/man/latest/systemd.resource-control.html

16.2 Projets « sécurité d'une VM, agilité d'un conteneur »

- Kata Containers : https://katacontainers.io/
- AWS Firecracker : https://firecracker-microvm.github.io/
- gVisor (Google) : https://gvisor.dev/

16.3 Ressources d'apprentissage et de référence

- *Container security* et durcissement (CIS Benchmarks Docker/Kubernetes) : https://www.cisecurity.org/
- Sysdig / Falco (sécurité runtime) : https://falco.org/
- Brendan Gregg (performance Linux, cgroups, eBPF) : https://www.brendangregg.com/
- *Julia Evans* — zines et articles pédagogiques (en anglais, très accessibles) sur namespaces et conteneurs : https://jvns.ca/
- it-connect.fr (tutoriels en français) : https://www.it-connect.fr/
- Linux Containers — forum communautaire : https://discuss.linuxcontainers.org/

Fin du document. Ce support se veut une synthèse technique approfondie ; les détails d'implémentation (numéros de version du noyau, options exactes) évoluent — se référer aux documentations officielles liées ci-dessus pour les informations les plus à jour.