ADMIN:DOCS

Linux Administration #8: Virtualisierung und VM-Management

Virtualisierung revolutioniert moderne IT-Infrastrukturen. Von VM-Management über Container bis hin zu Backup-Strategien - dieser Guide zeigt alle wichtigen Aspekte der Virtualisierung.

Kategorie: Linux-Administration **Veröffentlicht:** 24.11.2024 **Zuletzt geprüft:** 02.11.2026

Willkommen zum achten Teil unserer technischen Wiki-Serie über Linux-Administration!

Nachdem wir in den vorherigen Artikeln die Grundlagen, die Benutzerverwaltung, das Prozess- und Ressourcenmanagement, die Netzwerkkonfiguration, das Shell-Scripting, die Datensicherung sowie die Systemsicherheit & Härtung behandelt haben, widmen wir uns heute der Virtualisierung und dem VM-Management.

Die Virtualisierung ist heute ein fundamentaler Bestandteil moderner IT-Infrastrukturen.

> ⚠ **Hinweis:** In diesem Artikel verwenden wir Ubuntu/Debian als primäre Referenz-Distribution. Die grundlegenden Kernel- und Virtualisierungskonzepte gelten distributionsübergreifend. Bei Paketinstallationen und Dienstnamen (z. B. unter RHEL, Fedora oder Arch) können Bezeichnungen geringfügig abweichen.

Virtualisierungsgrundlagen

Die Virtualisierung ist eine fundamentale Technologie moderner IT-Infrastrukturen. Sie transformiert physische Serverressourcen in flexible, isolierte Umgebungen und ermöglicht dadurch eine effizientere Ressourcennutzung. Die Virtualisierung von Servern, Netzwerken und

Storage-Systemen bildet das Rückgrat von Rechenzentren, privaten Virtualisierungsclustern und modernen Cloud-Plattformen.

Warum Virtualisierung?

- **Ressourceneffizienz:** Maximale Auslastung moderner Multi-Core-CPUs und großer Arbeitsspeicherpools
- **Isolation:** Jede VM agiert als eigenständiges System mit dediziertem virtuellem Adressraum und Kernel
- **Flexibilität:** Schnelles Provisionieren, Klonen und Zurückrollen von Systemzuständen
- **Sicherheit & Kapselung:** Saubere Trennung unterschiedlicher Mandanten, Dienste und Sicherheitszonen
- **Entwicklung & Testing:** Risikolose Erprobung von Software-Releases, Kernel-Upgrades und Konfigurationsänderungen

Hypervisor-Typen im Vergleich:

Typ-1-Hypervisor (Bare-Metal)

Architektur
: Läuft direkt auf der physischen Hardware (KVM, ESXi, Xen, Proxmox VE)

Performance & Overhead
: Minimaler Overhead, hardwarenahe CPU- und I/O-Virtualisierung (VirtIO)

Einsatzbereich
: Enterprise-Server, Rechenzentren und produktive Cloud-Infrastrukturen

Verwaltung & APIs
: libvirt, virsh CLI, Cockpit, Terraform, OpenStack, REST-APIs

Typ-2-Hypervisor (Hosted)

Architektur

Läuft als Anwendungssoftware auf einem bestehenden Host-Betriebssystem

Performance & Overhead

Zusätzlicher Abstraktions-Overhead durch Scheduling des Host-OS

Einsatzbereich

Lokale Entwickler-Arbeitsplätze, Evaluierung und Testumgebungen

Verwaltung & APIs

Lokale Desktop-GUIs (VirtualBox Manager, VMware Workstation)

Gemeinsam: Vollständige Kapselung virtueller Maschinen mit virtueller Hardware

KVM/QEMU im Detail

KVM (Kernel-based Virtual Machine) und QEMU bilden das Fundament der professionellen Linux-Virtualisierung. KVM stellt als Kernel-Modul hardwaregestützte Virtualisierungsfunktionen (Intel VT-x / AMD-V) direkt im Linux-Kernel bereit. QEMU agiert im Userspace und übernimmt die Emulation sowie Bereitstellung virtueller Hardware (PCI-Busse, Speichercontroller, VirtIO-Treiber).

- **KVM** transformiert den Linux-Kernel in einen Typ-1-Hypervisor
- **QEMU** emuliert die Peripherie und orchestriert die Gast-Prozesse
- **libvirt** bietet eine einheitliche Abstraktionsschicht und Management-API

Das Schichtenmodell im Detail

Management & Administrations-Tools

virt-manager (GUI), virsh (CLI), Cockpit (Web), Ansible, Terraform

> libvirt API (libvirtd / modulare Daemons wie virtqemud)

QEMU Userspace-Prozesse

VirtIO-Blocktreiber, vHost-Net, ACPI, Display-Server, SPICE

/dev/kvm (Kernel-Schnittstelle)

KVM Kernel-Module (kvm.ko, kvm_intel.ko / kvm_amd.ko)

Hardware-gestützte CPU-Virtualisierung (VT-x / AMD-V), EPT/NPT-Speicherpaging

Physische Hardware (Multi-Core CPU, RAM, NVMe/SSD, NICs)

Installation Schritt für Schritt:

a) Prüfen der Hardware-Voraussetzungen

Prüfe vorab, ob die CPU hardwaregestützte Virtualisierung unterstützt und im UEFI/BIOS aktiviert ist:

```
# Prüfung der CPU-Flags (vmx bei Intel, svm bei AMD)
egrep -c '(vmx|svm)' /proc/cpuinfo

# Komfortable Prüfung via kvm-ok (aus dem Paket cpu-checker)
sudo apt install -y cpu-checker
kvm-ok
# Erwartete Ausgabe: INFO: /dev/kvm exists - KVM acceleration can be used
```

b) Notwendige Pakete installieren

Unter modernen Debian- und Ubuntu-Systemen (wie 24.04 und 26.04 LTS) installieren wir die aktuellen KVM- und libvirt-Metapakete inklusive OVMF-Firmware für modernen UEFI-Boot:

```
sudo apt update
sudo apt install -y \
  qemu-system-x86 \
  libvirt-daemon-system \
  libvirt-clients \
  bridge-utils \
  virtinst \
  virt-manager \
  ovmf
```

c) Benutzerrechte einrichten

Füge deinen Administrationsbenutzer den Gruppen `libvirt` und `kvm` hinzu, um VMs ohne Root-Rechte verwalten zu können:

```
sudo usermod -aG libvirt,kvm $USER

# Neue Gruppenzugehörigkeit in der aktuellen Shell aktivieren
newgrp libvirt
```

d) Dienste prüfen

```
sudo systemctl enable --now libvirtd

# Status prüfen (sollte "active (running)" sein)
sudo systemctl status libvirtd --no-pager
```

VM-Erstellung im Detail

Die Erstellung einer virtuellen Maschine ähnelt der Konfiguration eines physischen Systems. CPU-Kerne, Arbeitsspeicher, Storage-Volumes und virtuelle Netzwerkkarten werden aus den Host-Ressourcen allokiert.

a) Vorbereitung des Storage-Pools

Libvirt organisiert Festplatten-Images standardmäßig in Storage-Pools unter `/var/lib/libvirt/images`:

```
# Standard-Pool prüfen und aktivieren
virsh pool-info default

# Falls der Pool noch nicht aktiv ist:
virsh pool-start default
virsh pool-autostart default
```

b) ISO-Image oder Cloud-Image vorbereiten

```
# Verzeichnis für ISO-Installationsmedien
sudo mkdir -p /var/lib/libvirt/images/iso

# Aktuelles Ubuntu 24.04 LTS Server-Image herunterladen
sudo wget https://releases.ubuntu.com/24.04/ubuntu-24.04.1-live-server-amd64.iso \
  -O /var/lib/libvirt/images/iso/ubuntu-24.04-server.iso
```

c) VM mit virt-install bereitstellen (Modernes Setup mit UEFI & VirtIO)

Moderne Setups nutzen standardmäßig paravirtualisierte **VirtIO-Treiber** für Festplatten und Netzwerk sowie **UEFI (OVMF)** anstelle von veraltetem BIOS:

```
virt-install \
  --name srv-prod-01 \
  --memory 4096 \
  --vcpus 2 \
  --cpu host-passthrough \
  --boot uefi \
  --disk path=/var/lib/libvirt/images/srv-prod-01.qcow2,size=30,format=qcow2,bus=virtio \
  --cdrom /var/lib/libvirt/images/iso/ubuntu-24.04-server.iso \
  --os-variant ubuntu24.04 \
  --network network=default,model=virtio \
  --graphics none \
  --console pty,target_type=serial \
  --extra-args 'console=ttyS0,115200n8 serial'
```

> **Praxis-Tipp:** Für automatisierte Deployments ohne manuelle Menü-Installation unterstützt `virt-install` seit modernen Versionen den Parameter `--cloud-init`. In Kombination mit offiziellen Cloud-Images (`Ubuntu cloudimg` oder `Debian genericcloud`) erstellst du VMs innerhalb von 5 Sekunden vorkonfiguriert mit SSH-Schlüsseln und IP-Adressen.

d) Parameter im Detail erklärt:

- **`--memory 4096`:** Zuweisung des Arbeitsspeichers in MB (hier 4 GB).
- **`--vcpus 2`:** Anzahl virtueller vCPUs.
- **`--cpu host-passthrough`:** Reicht die exakten CPU-Instruktionssätze des physischen Hosts direkt an die VM weiter (optimale Performance).
- **`--boot uefi`:** Nutzt OVMF-Firmware für modernes Secure Boot und UEFI-Partitionslayouts.
- **`--disk ...,bus=virtio`:** Aktiviert paravirtualisierte VirtIO-Blocktreiber für maximale NVMe-/SSD-I/O-Geschwindigkeit.
- **`--network ...,model=virtio`:** Bindet die virtuelle Netzwerkkarte via VirtIO-Netzwerktreiber an.

e) Netzwerk-Modi im Überblick:

- **NAT (Standard `virbr0`):** Die VM erhält eine private IP aus dem Subnetz `192.168.122.0/24`. Der Host maskiert ausgehenden Traffic (Source-NAT/Masquerading). Die VM ist standardmäßig aus dem externen LAN nicht direkt erreichbar.
- **Bridged Networking (`br0`):** Die VM wird direkt an eine Host-Bridge angeschlossen und verhält sich netzwerktechnisch wie ein eigenständiger physischer Computer im selben LAN (inklusive eigener IP vom lokalen DHCP-Server).
- **Isoliertes Netzwerk (Host-Only / Internal):** Reines privates Netzwerk zwischen VMs und ggf. dem Host, ohne Internet-Routing.

Bridge-Netzwerk modern unter Ubuntu/Debian mit Netplan konfigurieren:

Anstelle veralteter `/etc/network/interfaces`-Dateien definieren moderne Distributionen Bridges in `/etc/netplan/01-netcfg.yaml`:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    enp3s0:
      dhcp4: false
      dhcp6: false
  bridges:
    br0:
      interfaces: [enp3s0]
      dhcp4: true
      parameters:
        stp: false
        forward-delay: 0
```

Nach einem `sudo netplan apply` kann die VM mit `--network bridge=br0,model=virtio` direkt ins Firmennetz eingebunden werden.

VM-Verwaltung und Monitoring

Die Verwaltung virtueller Maschinen gehört zu den zentralen Aufgaben im Rechenzentrumsbetrieb. Gast-Systeme müssen im laufenden Betrieb gesteuert, überwacht, gewartet und bei Konfigurationsänderungen oder Updates abgesichert werden.

a) Grundlegende VM-Verwaltung und Lebenszyklus

Virtuelle Maschinen durchlaufen definierte Betriebszustände in KVM/libvirt:

- **running (gestartet):** VM läuft aktiv, CPU- und RAM-Ressourcen sind gebunden
- **paused (pausiert):** CPU-Zustand ist angehalten, RAM bleibt im Host reserviert
- **shut off (ausgeschaltet):** VM ist sauber heruntergefahren, keine RAM-Belegung
- **pmsuspended / saved:** VM-Zustand wurde auf Disk gespeichert (Hibernation)

```
# Alle virtuellen Maschinen (aktiv und inaktiv) auflisten
virsh list --all

# Detaillierte Systemübersicht einer VM abrufen
virsh dominfo srv-prod-01

# VM starten
virsh start srv-prod-01

# Autostart beim Booten des Host-Systems aktivieren
virsh autostart srv-prod-01

# Sanftes Herunterfahren via ACPI-Signal
virsh shutdown srv-prod-01

# Sofortiger Stromausfall / Not-Aus (Hard Power-Off)
virsh destroy srv-prod-01
```

b) Snapshot-Verwaltung (Sicherung vor Systemänderungen)

Snapshots frieren den Zustand einer virtuellen Maschine zu einem bestimmten Zeitpunkt ein. Sie eignen sich hervorragend für risikoreiche Kernel-Updates oder Software-Installationen, ersetzen jedoch kein externes Backup.

- **Disk-Only Snapshot (`--disk-only`):** Friert ausschließlich die virtuelle Festplatte ein. Das Basis-Image wird schreibgeschützt, neue Schreibvorgänge wandern in ein Delta-QCOW2-Image.
- **Full Snapshot (inklusive RAM-Zustand):** Sichert den kompletten Arbeitsspeicher und CPU-Registerzustand. Erlaubt die nahtlose Wiederaufnahme ohne Reboot.

```
# Snapshot vor einem System-Upgrade erstellen (inklusive Beschreibung)
virsh snapshot-create-as srv-prod-01 snap-vor-upgrade "Vor Kernel 6.8 Upgrade" --atomic

# Snapshot inklusive RAM-Zustand (Live-State)
virsh snapshot-create-as srv-prod-01 snap-live-state "Live-Zustand vor Wartung" --memspec

# Alle Snapshots der VM auflisten
virsh snapshot-list srv-prod-01

# Snapshot-Metadaten und Elternknoten einsehen
virsh snapshot-info srv-prod-01 snap-vor-upgrade

# VM auf den Snapshot-Stand zurücksetzen (Rollback)
virsh snapshot-revert srv-prod-01 snap-vor-upgrade

# Nicht mehr benötigte Snapshots löschen (konsolidiert Delta zurück ins Basis-Image)
virsh snapshot-delete srv-prod-01 snap-vor-upgrade
```

□□ Praktisches Beispiel: Performance-Monitoring & Ressourcenanalyse

Ein effektives VM-Monitoring unterscheidet zwischen Hypervisor-Sicht (Ressourcenverbrauch auf dem Host) und In-Guest-Metriken. Libvirt stellt performante Schnittstellen bereit, um CPU-, Speicher-, Disk- und Netzwerkmetriken in Echtzeit abzufragen.

1. Interaktives Live-Monitoring mit virt-top:

```
# Live-Dashboard aller VMs (ähnlich 'top' / 'htop', jedoch für Gast-Systeme)
virt-top
```

`virt-top` zeigt vCPU-Auslastung, physische Host-CPU-Bindung, zugewiesenen vs. genutzten Arbeitsspeicher sowie I/O-Durchsatz pro Sekunde für jede VM an.

2. Detaillierte Statistiken per Kommandozeile (virsh domstats):

```
# Gesamte Performance-Metriken einer VM in strukturierter Ausgabe abrufen
virsh domstats srv-prod-01

# Nur CPU- und Memory-Statistiken filtern
virsh domstats srv-prod-01 --cpu-total --balloon

# Detaillierte Speichernutzung über den VirtIO-Balloon-Treiber
virsh dommemstat srv-prod-01

# Festplatten-I/O und gelesene/geschriebene Bytes abfragen
virsh domblkstat srv-prod-01 vda --human

# Netzwerktraffic und Paketstatistiken auf der virtuellen Schnittstelle
virsh domifstat srv-prod-01 vnet0
```

3. Automatisierte Ressourcen-Warnung (Bash & virsh dommemstat):

```
#!/usr/bin/env bash
```

```
set -Eeuo pipefail

VM_NAME="srv-prod-01"
# Ermittle tatsächliche Speichernutzung aus dem VirtIO-Ballooning
ACTUAL_MEM=$(virsh dommemstat "$VM_NAME" | awk '/actual/ {print $2}')
UNUSED_MEM=$(virsh dommemstat "$VM_NAME" | awk '/unused/ {print $2}')

if [[ -n "$ACTUAL_MEM" && -n "$UNUSED_MEM" && "$ACTUAL_MEM" -gt 0 ]]; then
  USED_PERCENT=$(( (ACTUAL_MEM - UNUSED_MEM) * 100 / ACTUAL_MEM ))
  if (( USED_PERCENT > 90 )); then
    logger -p user.warn -t vm-monitor "WARNUNG: VM $VM_NAME Speichernutzung kritisch:
${USED_PERCENT}%"
  fi
fi
```

VM Backup und Migration

1. VM-Backup-Strategien

Die regelmäßige und automatisierte Sicherung virtueller Maschinen ist unverzichtbar. Ein Ausfall physischer Hypervisor-Hardware darf niemals zum Datenverlust der darauf konsolidierten Gastsysteme führen.

Backup-Methoden im Überblick:

- **Offline-Backup (Cold Backup):** Die VM wird vorübergehend heruntergefahren (`virsh shutdown`). Die XML-Konfigurationsdatei und das QCOW2-Image werden konsistent kopiert. Höchste Datenkonsistenz, erfordert jedoch ein geplantes Wartungsfenster.
- **Online-Live-Backup mit QEMU Guest Agent:** Die VM bleibt aktiv. Über den im Gast installierten `qemu-guest-agent` wird das Dateisystem vor dem Snapshot kurzzeitig eingefroren (`virsh domfsfreeze`), das Delta gesichert und das Dateisystem anschließend wieder freigegeben (`virsh domfsthaw`).
- **Inkrementelles Backup (QEMU Dirty Bitmaps):** Moderne KVM/libvirt-Setups tracken veränderte Speicherblöcke im laufenden Betrieb, sodass nur modifizierte Blöcke in externe Backup-Repositories (z. B. via Restic oder Borg) übertragen werden.

Vollständiges Offline-Backup per CLI:

```
# 1. VM sauber herunterfahren
virsh shutdown srv-prod-01

# 2. Hardware-Konfiguration als XML exportieren
virsh dumpxml srv-prod-01 > /backup/srv-prod-01-$(date +%F).xml

# 3. Disk-Image kopieren (z. B. auf ein gemountetes Backup-Storage)
cp --sparse=always /var/lib/libvirt/images/srv-prod-01.qcow2 /backup/srv-prod-01-$(date
+%F).qcow2

# 4. VM wieder starten
virsh start srv-prod-01
```

Online-Konsistenz mit dem QEMU Guest Agent:

Installiere innerhalb des Gast-Betriebssystems den Agenten:

```
# Im Gast-Betriebssystem:
sudo apt install -y qemu-guest-agent
sudo systemctl enable --now qemu-guest-agent

# Vom Host aus: Dateisysteme für konsistente Snapshots synchronisieren
virsh domfsfreeze srv-prod-01
# Snapshot oder Block-Kopie anstoßen ...
virsh domfsthaw srv-prod-01
```

> □□ **Praxis-Tipp:** Wende für die Auslagerung der VM-Backups stets die in Modul #6 behandelte **3-2-1-1-0-Regel** an. Mindestens eine Kopie der VM-Images gehört

> unveränderlich (Immutable Storage) an einen geographisch getrennten Standort.

2. VM-Migration

In Enterprise-Umgebungen müssen virtuelle Maschinen häufig ohne Betriebsunterbrechung zwischen physischen Hosts verschoben werden – sei es für geplante Hardware-Wartungen, BIOS-/Kernel-Upgrades oder zur dynamischen Lastverteilung.

Migrations-Arten:

- **Cold Migration:** Die VM ist gestoppt. XML-Definition und Disk-Images werden über SSH/Rsync auf den Ziel-Host transferiert und dort gestartet.
- **Live Migration (Shared Storage):** Quell- und Ziel-Host greifen auf denselben Storage-Pool zu (z. B. via NFS, iSCSI oder Ceph RBD). Nur der Arbeitsspeicher und der CPU-Status werden im laufenden Betrieb übertragen.
- **Storage Live Migration:** Überträgt zusätzlich zum RAM auch das gesamte Disk-Image im laufenden Betrieb auf lokalen Storage des Zielhosts (Block Live Migration).

Der Ablauf einer KVM Live-Migration:

1. **Prüfung & Handshake** Quell- und Zielhost validieren CPU-Kompatibilität, libvirt-Sockets und Shared Storage
2. **Pre-Copy Speicherübertragung** Übertragung des aktiven Arbeitsspeichers im laufenden Betrieb über das Migrationsnetzwerk
3. **Iteratives Dirty-Page-Tracking** Wiederholte Synchronisation jener RAM-Seiten, die während der Übertragung geändert wurden
4. **Cutover (Mikro-Pause)** Kurzes Pausieren der VM (Millisekunden) zur Übertragung des finalen CPU-Registers

5. **Aktivierung auf dem Zielhost** Reaktivierung der vCPUs auf dem Zielsystem und Aktualisierung der Switch-Routingtabellen (GARP)

Live-Migration mit virsh durchführen:

```
# Voraussetzungen:
# - SSH-Key-Authentifizierung zwischen Quell- und Ziel-Host eingerichtet
# - Namensauflösung und Netzwerkverbindung über dediziertes 10G/25G-Migrationsnetz

# Live-Migration mit gemeinsam genutztem Speicher
virsh migrate --live --verbose \
  --p2p \
  --tunnelled \
  srv-prod-01 \
  qemu+ssh://kvm-node02.intern/system

# Storage-Live-Migration (ohne gemeinsamen Speicher; Disks werden mitkopiert)
virsh migrate --live --verbose \
  --copy-storage-all \
  srv-prod-01 \
  qemu+ssh://kvm-node02.intern/system
```

Cold-Migration manuell (ohne Cluster-Filesystem):

```
# 1. VM auf Host 1 stoppen
virsh shutdown srv-prod-01

# 2. XML-Definition und Disk-Image auf Host 2 kopieren
virsh dumpxml srv-prod-01 > /tmp/srv-prod-01.xml
scp /tmp/srv-prod-01.xml kvm-node02:/tmp/
scp /var/lib/libvirt/images/srv-prod-01.qcow2 kvm-node02:/var/lib/libvirt/images/
```

```
# 3. Auf Host 2: VM definieren und starten
ssh kvm-node02 'virsh define /tmp/srv-prod-01.xml && virsh start srv-prod-01'

# 4. Auf Host 1: Bereinigen
virsh undefine srv-prod-01
rm /var/lib/libvirt/images/srv-prod-01.qcow2
```

Container-Virtualisierung

Container stellen eine leichtgewichtige, hochperformante Alternative zu hardwarevirtualisierten VMs dar. Anstatt vollständige Hardware und einen eigenen Betriebssystem-Kernel zu emulieren, setzen Container auf die Kernel-Features des Host-Systems auf: **Linux Namespaces** (für Prozess-, Netzwerk-, Benutzer- und Mount-Isolation) sowie **cgroups v2** (für präzise CPU-, Memory- und I/O-Drosselung).

KVM-Virtualisierung und Container im direkten Vergleich:

Virtuelle Maschinen (KVM/QEMU)

Virtualisierungsebene
: Hardware-Ebene: Hypervisor emuliert Hardware für Gast-Kernel

Isolation & Sicherheit
: Starke Hardware-Isolation, eigener Gast-Kernel pro VM

Ressourcenverbrauch
: Feste RAM-Allokation, Overhead durch separates Gast-OS

Startzeit
: 10 bis 45 Sekunden für vollständigen Betriebssystem-Boot

Betriebssysteme
: Beliebig (Linux, Windows, BSD auf demselben physischen Host)

Container (Docker, Podman, Incus)

Virtualisierungsebene
: Betriebssystem-Ebene: Prozesse teilen sich den Host-Kernel

Isolation & Sicherheit
: Prozess-Isolation via Linux Namespaces, Seccomp und cgroups v2

Ressourcenverbrauch
: Minimaler Speicher- und CPU-Overhead, nativer Kernel-Zugriff

Startzeit
: Millisekunden bis wenige Sekunden (reiner Prozessstart)

Betriebssysteme
: Ausschließlich Linux-Userlands auf dem gemeinsamen Linux-Kernel

Gemeinsam: Isolierte Workloads, definierbare Ressourcenlimits und Netzwerk-Kapselung

1. Docker und Podman

Anwendungs-Container kapseln exakt einen Dienst oder Microservice samt aller Laufzeitabhängigkeiten in einem unveränderlichen OCI-Image.

Installation & Basiskonfiguration:

Unter Ubuntu und Debian installierst du Docker Engine oder die daemonlose Alternative Podman:

```
# Docker Engine installieren
sudo apt update
sudo apt install -y docker.io docker-compose-v2

# Benutzer zur Docker-Gruppe hinzufügen (oder Rootless Podman nutzen)
sudo usermod -aG docker $USER
newgrp docker
```

```
# Dienst aktivieren und Status prüfen
sudo systemctl enable --now docker
```

> **Praxis-Tipp:** Für maximale Sicherheit in modernen Infrastrukturen empfehlen sich **Rootless Container** (z. B. via `podman` oder `dockerd-rootless`). Hierbei läuft der Container-Daemon vollständig in der Session eines unprivilegierten Benutzers. Selbst bei einem hypothetischen Container-Ausbruch besitzt der Angreifer auf dem Host keine Root-Rechte.

Container-Netzwerke und Storage:

- **Bridge-Netzwerk (Standard):** Container erhalten eine IP aus einem virtuellen Docker-Bridge-Netz und können über DNS-Namen miteinander kommunizieren.
- **Host-Netzwerk (`--network host`):** Der Container teilt sich direkt den Netzwerk-Namespace des Hosts (geringste Latenz, aber keine Port-Isolation).
- **Named Volumes (`-v db_data:/var/lib/mysql`):** Persistente Daten werden im Docker-Managed-Verzeichnis (`/var/lib/docker/volumes/`) abgelegt und überleben Container-Neustarts und -Upgrades.
- **Bind Mounts (`-v /opt/app/config:/etc/app:ro`):** Direktes Einbinden von Host-Verzeichnissen in den Container, idealerweise schreibgeschützt (`:ro`).

```
# Eigenes isoliertes Netzwerk erstellen
docker network create internal-net

# Datenbank-Container mit Named Volume und Ressourcendrosselung starten
docker run -d \
  --name mariadb-app \
  --network internal-net \
  --memory 1024m \
  --cpus 1.5 \
  --restart unless-stopped \
```

```
  -v mariadb_data:/var/lib/mysql \
  -e MARIADB_ROOT_PASSWORD=SicheresPasswort2026! \
  mariadb:11

# Webserver im selben Netzwerk starten (Port 80 auf Host weiterleiten)
docker run -d \
  --name webserver \
  --network internal-net \
  -p 80:80 \
  --memory 512m \
  --restart unless-stopped \
  nginx:alpine
```

c) Moderne Multi-Container-Orchestrierung mit Docker Compose v2

In modernen Serverumgebungen wird die deklarative Definition von Multi-Container-Stacks über `docker compose` (v2 CLI-Plugin ohne Bindestrich) gesteuert:

```
# compose.yaml
services:
  web:
    image: nginx:alpine
    ports:
      - "8080:80"
    restart: unless-stopped
    deploy:
      resources:
        limits:
          cpus: '1.0'
          memory: 512M
    networks:
      - app-tier

  app:
    image: node:20-alpine
    working_dir: /app
    volumes:
```

```
      - ./src:/app:ro
    environment:
      NODE_ENV: production
    networks:
      - app-tier

networks:
  app-tier:
    driver: bridge
```

```
# Multi-Container-Stack im Hintergrund starten
docker compose up -d

# Status und Ressourcenauslastung des Stacks prüfen
docker compose ps
docker compose top
```

d) Container-Sicherheit und Härtung

Da sich Container den Host-Kernel teilen, erfordert der Produktivbetrieb strikte Sicherheitsgrenzen:

- **Unprivilegierter Benutzer:** Führe Prozesse niemals als Root im Container aus (`USER 1000:1000` im Dockerfile oder `--user 1000:1000` beim Start).
- **Capabilities minimieren:** Entziehe standardmäßig alle Linux-Kernel-Capabilities und füge nur explizit benötigte hinzu:

  ```
  docker run --cap-drop=ALL --cap-add=NET_BIND_SERVICE nginx:alpine
  ```

- **Privilegieneskalation unterbinden:** `--security-opt=no-new-privileges:true` blockiert SUID-Binärdateien.
- **Read-Only Root-Dateisystem:** Verhindert persistente Schadcode-Injektionen:

```
docker run --read-only --tmpfs /tmp --tmpfs /run nginx:alpine
```

- **Ressourcendrosselung via cgroups v2:** Verhindert Denial-of-Service durch Memory Leaks oder CPU-Starvation:

```
docker run --memory=512m --cpus=1 nginx:alpine
```

e) Container-Monitoring & Logging in der Praxis

```
# Live-Ressourcenverbrauch aller laufenden Container
docker stats --no-stream

# Container-Logs mit Zeitstempeln und Zeilenbegrenzung
docker logs -f --tail 100 webserver

# JSON-Konfiguration, IP-Adressen und Mount-Points auslesen
docker inspect webserver
```

2. System-Container mit Incus und LXC

Während Docker und Podman einzelne Microservices isolieren, bieten **System-Container** ein vollwertiges Betriebssystem mit eigenem `systemd`-Init-System, Cron, Syslog und Multi-User-Verwaltung. Sie verhalten sich wie extrem schlanke KVM-VMs, teilen sich jedoch den Kernel des Host-Systems.

> **Incus vs. LXD (Stand 2026):** Nach dem Wechsel der Lizenz und des Markenmanagements von LXD durch Canonical im Jahr 2023 hat die ursprüngliche Open-Source-Community unter dem Dach von *LinuxContainers.org* das Projekt als **Incus** geforkt. Incus ist heute der native Standard in Debian, Fedora, Arch Linux und Enterprise-Umgebungen, während Canonical LXD primär als Snap unter Ubuntu pflegt. Die Befehlssyntax beider Systeme ist weitgehend deckungsgleich.

Incus/LXD einrichten und Container verwalten:

```
# Incus installieren (unter Debian/Ubuntu oder anderen Distributionen)
sudo apt install -y incus

# Initialisierungskonfiguration durchlaufen (Storage-Pool, Bridge-Netzwerk)
sudo incus admin init

# System-Container mit Debian 12 oder Ubuntu 24.04 starten
incus launch images:debian/12 srv-monitoring

# Laufende Container auflisten (inklusive IP-Adressen und Ressourcennutzung)
incus list

# Interaktive Shell im Container öffnen
incus exec srv-monitoring -- bash

# Host-Ressourcen dynamisch limitieren (cgroups v2)
incus config set srv-monitoring limits.cpu 2
incus config set srv-monitoring limits.memory 2GiB
```

```
incus config set srv-monitoring limits.disk.size 25GiB
```

Sicherheitsmodell unprivilegierter Container:

Incus betreibt Container standardmäßig als **unprivilegierte Container**. Dabei werden UIDs und GIDs über Linux User Namespaces gemappt (z. B. entspricht Root mit UID 0 im Container der unprivilegierten Host-UID 1000000). Selbst wenn ein Prozess im Container die volle Kontrolle erlangt, besitzt er auf dem Host-System keinerlei administrative Berechtigungen auf dem Hypervisor.

```
# Snapshot des System-Containers vor einer Aktualisierung
incus snapshot create srv-monitoring pre-update-$(date +%F)

# Snapshots auflisten
incus info srv-monitoring

# Im Fehlerfall: Snapshot wiederherstellen
incus snapshot restore srv-monitoring pre-update-2026-11-02
```

Übung

Virtualisierungsumgebung aufsetzen

Szenario: Du bist System-Administrator in einem Unternehmen. Deine Aufgabe ist es, eine moderne Virtualisierungs- und Testumgebung auf Basis von KVM/libvirt und Containern bereitzustellen.

Anforderungen an die Testumgebung:

- **Entwicklungsserver (VM 1):** Ubuntu Server 24.04 LTS mit 4 GB RAM, 2 vCPUs, 40 GB QCOW2-Storage und UEFI-Boot

- **Testserver (VM 2):** Debian 12 Bookworm mit 2 GB RAM, 1 vCPU, 20 GB QCOW2-Storage und UEFI-Boot
- **Monitoring-Instanz:** Schlanker System-Container (Incus/LXC) mit 1 GB RAM-Limit für Überwachungsdienste
- **Automatisierte Sicherung:** Snapshot-Erstellung und Bereinigung alter Snapshots

Aufgaben:

1. **VM-Setup mit virt-install:** Erstelle beide VMs mit den spezifizierten Parametern, VirtIO-Treibern und UEFI-Firmware im Standard-NAT-Netzwerk.
2. **Snapshot-Management:** Erstelle nach der Basisinstallation saubere Snapshots beider VMs.
3. **Backup-Automatisierung:** Implementiere ein sicheres Bash-Wartungsskript mit Fehlerbehandlung zur automatisierten Snapshot-Rotation.
4. **Container-Setup:** Starte einen unprivilegierten System-Container mit festen Ressourcenlimits für das Monitoring.

Mögliche Lösung:

1. VM-Bereitstellung via virt-install:

```
# Entwicklungsserver (Ubuntu 24.04 LTS mit UEFI & VirtIO)
virt-install \
  --name dev-server \
  --memory 4096 \
  --vcpus 2 \
  --cpu host-passthrough \
  --boot uefi \
  --disk size=40,format=qcow2,bus=virtio \
  --os-variant ubuntu24.04 \
  --network network=default,model=virtio \
  --cdrom /var/lib/libvirt/images/iso/ubuntu-24.04-server.iso \
  --graphics none \
  --console pty,target_type=serial
```

```
# Testserver (Debian 12 mit UEFI & VirtIO)
virt-install \
  --name test-server \
  --memory 2048 \
  --vcpus 1 \
  --cpu host-passthrough \
  --boot uefi \
  --disk size=20,format=qcow2,bus=virtio \
  --os-variant debian12 \
  --network network=default,model=virtio \
  --cdrom /var/lib/libvirt/images/iso/debian-12-netinst.iso \
  --graphics none \
  --console pty,target_type=serial
```

2. Snapshot-Erstellung nach Erstinstallation:

```
virsh snapshot-create-as dev-server snap-baseline "Basis-Installation abgeschlossen"
virsh snapshot-create-as test-server snap-baseline "Basis-Installation abgeschlossen"
```

3. Snapshot-Rotationsskript:

```
cat << 'EOF' > /usr/local/bin/vm-snapshot-rotation.sh
#!/usr/bin/env bash
set -Eeuo pipefail

TARGET_VM="dev-server"
SNAP_PREFIX="auto-backup"
TODAY=$(date +%Y%m%d)

# Neuen Snapshot erstellen
```

```
virsh snapshot-create-as "$TARGET_VM" "${SNAP_PREFIX}-${TODAY}" "Taeglicher automatischer
Snapshot"

# Snapshots auflisten und alle aelter als 7 Tage loeschen
mapfile -t OLD_SNAPS < <(virsh snapshot-list "$TARGET_VM" --name | grep "^${SNAP_PREFIX}-"
| sort -r | tail -n +8)

for snap in "${OLD_SNAPS[@]}"; do
  if [[ -n "$snap" ]]; then
    virsh snapshot-delete "$TARGET_VM" "$snap"
  fi
done
EOF

chmod 750 /usr/local/bin/vm-snapshot-rotation.sh
```

4. Monitoring System-Container bereitstellen:

```
# Unprivilegierten Debian 12 Container starten
incus launch images:debian/12 monitoring-node

# Ressourcen begrenzen
incus config set monitoring-node limits.memory 1GiB
incus config set monitoring-node limits.cpu 1

# Status prüfen
incus list monitoring-node
```

Befehlsreferenz (Cheatsheet)

kvm-ok | Prüft Hardware-Virtualisierungsunterstützung der CPU (VT-x / AMD-V)

virsh list --all | Listet alle virtuellen Maschinen samt aktuellem Betriebsstatus auf

$ `virsh dominfo <VM> | Zeigt detaillierte Hardwareressourcen, Speicher und Status einer VM virsh start <VM> | Startet eine ausgeschaltete virtuelle Maschine virsh shutdown <VM> | Sendet ACPI-Signal für geordnetes Herunterfahren des Gastsystems virsh destroy <VM> | Erzwingt sofortiges Abschalten der VM (Hard Power-Off) virsh autostart <VM> | Aktiviert den automatischen Start der VM beim Booten des Hosts virt-install --name ... | Provisioniert eine neue VM via CLI inklusive vCPU, RAM, Disk und UEFI qemu-img create -f qcow2 disk.qcow2 50G | Erstellt ein dynamisch wachsendes QCOW2-Festplatten-Image virsh snapshot-create-as <VM> <Name> | Erstellt einen konsistenten Disk-Snapshot einer VM virsh snapshot-revert <VM> <Name> | Setzt die VM auf den Stand des angegebenen Snapshots zurück virt-top | Interaktive Echtzeit-Überwachung von vCPU, RAM und Disk-I/O aller VMs virsh domstats <VM> | Gibt aggregierte Performance- und Hardware-Statistiken einer VM aus virsh dumpxml <VM> | Exportiert die vollständige Hardware- und Laufzeitdefinition als XML virsh migrate --live <VM> qemu+ssh://host/system | Führt Live-Migration einer VM auf einen anderen Hypervisor durch docker compose up -d | Startet Multi-Container-Stacks deklarativ im Hintergrund incus launch images:<os>/<ver> <name> | Erstellt und startet einen isolierten System-Container`

Weiterführende Ressourcen

Fazit

Die Beherrschung von Virtualisierungstechnologien und modernem Workload-Management bildet einen essenziellen Meilenstein auf dem Weg zum Senior Linux Administrator. Durch das Zusammenspiel von KVM, QEMU und Libvirt transformierst du physische Server in hochverfügbare, flexible Virtualisierungsplattformen mit Bare-Metal-naher Performance. Ergänzt durch OCI-Container für Microservices und Incus für leichtgewichtige Systemumgebungen verfügst du über das vollständige Instrumentarium für moderne Enterprise-Infrastrukturen.

> □□ **Praxis-Tipp:** Nutze für Produktions-VMs stets das **QCOW2-Format mit Preallocation-Metadaten** (`qemu-img create -f qcow2 -o preallocation=metadata disk.qcow2 100G`) oder LVM-Thin-Pools. Dadurch erzielst du nahezu Bare-Metal-I/O-Performance bei voller Snapshot- und Cloning-Flexibilität.

Im nächsten Modul unseres Administrationskurses widmen wir uns der kontinuierlichen Systemüberwachung und Leistungsanalyse:

□□ **Weiter geht es mit:** Linux Administration #9: Systemüberwachung und Performance-Monitoring

□□ **Zur Kursübersicht:** Alle Linux-Administrations-Artikel & Module