ADMIN:DOCS

LXD: Understanding Fundamentals and Architecture

Explore the core architecture of LXD: How system containers and VMs interact via the host kernel, how namespaces and cgroups isolate workloads, and how components work together.

Category: LXD **Published:** 02.11.2026

In production infrastructure, administrators frequently face the need to segregate distinct services and software stacks: A relational database must run isolated from web application dependencies, internal development systems require independent init processes with dedicated SSH access, and different workloads demand specific Linux distributions such as Debian, Ubuntu, or Alpine. Deploying a dedicated physical server for each of these tasks consumes excessive hardware resources, power, and maintenance overhead.

Traditional virtual machines address this isolation requirement reliably, but each guest incurs the penalty of an independent Linux kernel, emulated hardware devices, and statically allocated memory. For standard Linux workloads, hardware emulation is often unnecessary overhead. This is where LXD steps in: As a daemon-based manager for system containers and virtual machines, LXD delivers lightweight instances that behave like full Linux servers from an administrative standpoint, yet run directly on the host kernel and share resources dynamically.

Before diving into hands-on installation and day-to-day operations, the architectural fundamentals must be clear: How does a system container differ from application containers like Docker or Podman? Under what conditions does a full virtual machine remain the architecturally sound choice despite its overhead? And which Linux kernel primitives guarantee that processes, network stacks, and storage paths remain strictly isolated?

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture

> **Self-Study Note:** This course systematically covers LXD from fundamental architecture to production deployment. Keep one core distinction in mind from the start: LXD is not meant to compete with Docker for transient, single-process microservices. Instead, it is a robust infrastructure platform for long-running system environments operated and maintained like real servers.

This introductory course lesson examines the architecture and runtime mechanics of LXD, explains how the background daemon interacts with the `lxc` command-line client, and outlines the primary components that form the basis for the upcoming operational modules.

What Is LXD?

LXD is a modern, REST-API-driven manager for system containers and virtual machines on Linux. Maintained by Canonical LXD, it merges the density and near-zero overhead of native Linux containers with the manageable operational workflow of traditional hypervisors.

Unlike application containers (such as Docker or Podman), which are designed to wrap a single executable per container, LXD manages **system containers** as well as full **virtual machines** via QEMU (introduced in LXD 4.0). An LXD instance behaves like a complete Linux operating system: It boots an independent init system (typically `systemd`), runs standard background daemons like `cron`, `journald`, or `sshd`, and allows package management via `apt`, `dnf`, or `apk`.

Client-Server Architecture: Daemon, REST API, and CLI

LXD follows a strict client-server model, enabling identical administrative workflows for local and remote hosts:

- **The LXD Daemon (`lxd`):** The operational core runs as a privileged background service. It governs the full instance lifecycle, configures storage pools, builds software-defined networks, enforces access controls, and communicates with low-level kernel interfaces for containers or with QEMU/KVM for virtual machines.
- **The REST API:** Every daemon capability is exposed over a structured REST API. Local clients communicate via a secure UNIX domain socket (

`/var/snap/lxd/common/lxd/unix.socket` on Snap-based systems). Remote clients connect over TLS-encrypted HTTPS sessions (defaulting to TCP port 8443).

- **The Command-Line Client (`lxc`):** Routine administration is handled through the `lxc` CLI utility. The client does not manipulate kernel primitives directly; it translates user commands into REST requests and transmits them to the daemon. Consequently, the same `lxc` command can manage a local workstation, a remote server, or a multi-node cluster without syntax changes.

> ⚠ **Security Boundary of the lxd Group:** Local user accounts added to the host's `lxd` group gain direct read and write access to the daemon's UNIX socket. Because the REST API allows mounting arbitrary host filesystems into instances, membership in the `lxd` group is administratively equivalent to full root privileges on the host system.

Distinction: LXD Is Not LXC

A common source of confusion stems from the naming of the command-line utility: Because the command is called `lxc`, LXC and LXD are often mistakenly treated as interchangeable terms. They represent entirely different operational layers:

- **LXC (LinuX Containers):** The original low-level C project (liblxc). It provides libraries and basic utilities to manually wire Linux kernel features (such as namespaces and cgroups) into isolated environments. LXC has no centralized daemon, no REST API, no integrated storage or network management, and no VM support.
- **LXD:** Historically built on top of `liblxc`, LXD completely abstracts the low-level mechanics. It adds a multi-tenant daemon, automated image distribution from public and private registries, managed network bridges, dynamic storage drivers (ZFS, Btrfs, LVM), and clustering.

Running `lxc launch` today invokes LXD. Low-level LXC is rarely managed directly in modern production environments; instead, it serves as the underlying execution runtime for higher-level platforms.

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture

Containers and Virtual Machines Compared

LXD occupies a unique position in the Linux virtualization landscape by orchestrating two distinct isolation models through a unified management interface: system containers and virtual machines. Both models isolate workloads, but they operate at fundamentally different architectural boundaries.

Kernel Sharing Versus Hardware Virtualization

The decisive architectural distinction lies in how the operating system kernel is handled:

- **System Containers (Kernel Sharing):** From the host perspective, an LXD container is simply an isolated group of processes. There is no emulated hardware, no virtual firmware, and no guest kernel. Every system call (*syscall*) issued by an application inside the container is handled directly by the host Linux kernel. The container merely possesses its own root filesystem, independent network interfaces, and dedicated process IDs. This yields virtually zero execution overhead: An idle container consumes only a few megabytes of RAM and boots in milliseconds.
- **Virtual Machines (Hardware Virtualization):** A virtual machine utilizes the QEMU hypervisor alongside the Linux KVM kernel module to emulate a complete PC, including virtual BIOS/UEFI firmware, PCI buses, storage controllers, and network adapters. The guest boots its own standalone operating system kernel. Syscalls remain confined to the guest kernel. Hardware-assisted virtualization instructions (Intel VT-x, AMD-V) enforce isolation at the CPU level.

Technical Comparison

The operational differences between both instance types are summarized below:

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture

LXD System Containers

Kernel Architecture

: Shares the Linux kernel of the host system

Memory Allocation

: Dynamic: Unused RAM remains available to the host immediately

Startup Latency

: Sub-second (equivalent to launching a standard process tree)

Isolation Boundary

: Kernel namespaces, cgroups v2, AppArmor, and Seccomp filters

Storage Access

: Direct mounts and block access via host storage drivers

Primary Use Cases

: Linux server workloads, CI build nodes, dense staging clusters

Virtual Machines (KVM / QEMU)

Kernel Architecture

: Dedicated, independent guest kernel (Linux, Windows, BSD)

Memory Allocation

: Static: Assigned RAM is locked by default (without memory ballooning)

Startup Latency

: 10 to 30 seconds (full BIOS/UEFI initialization and kernel boot)

Isolation Boundary

: Hardware-enforced CPU boundaries via VT-x/AMD-V virtualization flags

Storage Access

: Virtual disk images (qcow2, raw) or virtio-fs filesystems

Primary Use Cases

: Non-Linux systems, kernel development, untrusted multi-tenancy

Shared: LXD manages both instance types using the exact same lxc command set and profile model

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture

Operational Decision Criteria: When to Choose Containers, When VMs?

Neither approach is universally superior. The choice between system containers and virtual machines depends on specific operational constraints:

- **Operating System Compatibility:** Because system containers rely on the host kernel, they can only run Linux distributions. Virtualizing Windows Server or FreeBSD requires a virtual machine.
- **Kernel Modules and Network Stacks:** If a workload requires custom kernel drivers, low-level modules (such as specific VPN or WireGuard extensions), or modifications to `/proc/sys/` parameters, a VM is mandatory. Containers share kernel parameters with the host and cannot modify them arbitrarily for security reasons.
- **Security Boundaries and Multi-Tenancy:** Unprivileged containers provide strong isolation through User Namespaces by remapping `root` inside the container to an unprivileged UID on the host. Nevertheless, the kernel syscall surface remains shared: A vulnerability in the host kernel can potentially affect all containers running on that machine. For hostile or untrusted tenants, hardware-enforced VM isolation remains the industry standard.
- **Density and Efficiency:** When deploying dozens or hundreds of isolated environments on a single host (such as in CI/CD pipelines or staging clusters), system containers outperform VMs dramatically. They avoid redundant kernel memory consumption and eliminate CPU virtualization overhead.

> □□ **Practical Tip:** With LXD, you do not need to lock yourself into a single technology beforehand. LXD allows running system containers and virtual machines side by side on the same host, attached to the same network bridge and storage pools. The command differs by a single flag: `lxc launch ubuntu:24.04 c1` starts a container, whereas `lxc launch ubuntu:24.04 vm1 --vm` boots a full KVM-backed virtual machine. For a broader overview of hypervisor architectures, see our module on Linux Administration: Virtualization and VM Management.

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture

How LXD Works Under Linux

Operating LXD reliably requires understanding that system containers are not independent machines. They are standard Linux process groups constrained by native kernel primitives into an isolated perspective of the host system.

LXD Instances & Container Userspace

systemd, background daemons, package management, and filesystems

Security Policies & Profiles

AppArmor profiles, Seccomp syscall filters, and UID/GID user namespace mappings

Linux Kernel Isolation

Namespaces, cgroups v2, veth pairs, and software bridge routing

Host Hardware & Base Operating System

CPU, RAM, block storage pools (ZFS/Btrfs/LVM), and physical networking

The Role of the Linux Kernel as the Root of Trust

In container operations, only one kernel runs on the machine: the host kernel. All containers share this single execution engine:

- Every system call issued by a process inside a container (e.g., `open()`, `fork()`, `read()`) is executed directly by the host kernel.
- There is no CPU instruction translation; code executes natively on the physical processor cores.
- If the host kernel encounters a kernel panic, all containers on that host halt immediately.

To turn regular processes into what looks and feels like a standalone server, LXD coordinates two foundational kernel features: **Namespaces** and **Control Groups**.

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture

Linux Namespaces: Defining What the Container Can See

Namespaces establish *visibility boundaries* for processes. LXD configures all modern Linux namespaces to create the illusion of an isolated operating system:

- **PID Namespace (`pid`):** Each container gets an isolated process tree. The container init process starts as PID 1 and can only view its own descendants. On the host, this same process runs under a normal, higher process ID.
- **Network Namespace (`net`):** Each container maintains an independent network stack with private interfaces (such as `eth0` and `lo`), private routing tables, socket lists, and separate firewall rules (`nftables` / `iptables`).
- **Mount Namespace (`mnt`):** Processes inside a container can only access filesystem mounts explicitly attached to their namespace. Mount and unmount actions inside the container do not alter the host filesystem hierarchy.
- **User Namespace (`user`):** The primary security pillar for unprivileged containers. User IDs (UID) and Group IDs (GID) are translated between the container and the host. The `root` user (UID 0) inside the container maps to an unprivileged UID on the host (e.g., UID 1000000). If a process breaks out of container containment, it holds only unprivileged user rights on the host.
- **UTS Namespace (`uts`):** Grants the container its own hostname and domain name, independent of the host identifier.
- **IPC Namespace (`ipc`):** Isolates inter-process communication mechanisms (such as shared memory segments and message queues) between containers and the host.

Control Groups (cgroups v2): Defining What the Container Can Consume

While namespaces restrict what a process can see, Control Groups (cgroups) govern *how many resources* a process tree may consume. They prevent a single rogue or overloaded container from starving the host (*noisy-neighbor problem*):

- **CPU Limits:** LXD can pin instances to dedicated CPU cores (`limits.cpu = 2`) or throttle execution time via CPU quotas and scheduling shares.

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture

- **Memory Limits:** The `limits.memory` property sets maximum RAM consumption. If an instance exceeds its allocated threshold, the Linux Out-of-Memory (OOM) killer terminates processes strictly inside that container's cgroup, leaving the host and other instances unharmed.
- **I/O Prioritization:** cgroups throttle read and write throughput on block storage devices to guarantee fair disk access across all instances.
- **Process Limits (PIDs):** Restricts the maximum number of concurrent tasks (`limits.processes`), effectively neutralizing fork bombs.

> **Unified cgroups v2 Architecture:** Modern Linux distributions (such as Ubuntu 24.04 or Debian 12) default to the unified cgroups v2 hierarchy. LXD leverages this architecture natively for fine-grained resource throttling without legacy controller synchronization issues. To learn how process boundaries are inspected manually, review our guide on Linux Administration: Processes and Resource Management.

Filesystem and Process Isolation

For filesystem containment, LXD avoids fragile `chroot` mechanics, which have historically been vulnerable to breakout techniques. Instead, LXD uses the `pivot_root` system call. This operation shifts the container's prepared rootfs into place as the primary mount point and unmounts access to the host's underlying directory tree.

LXD layers two additional security mechanisms over every instance:

- **AppArmor:** Kernel security profiles restrict which file paths, raw devices, and network sockets processes inside a container can interact with. Even if a process runs as root, AppArmor blocks write access to critical kernel filesystems under `/sys` and `/proc`.
- **Seccomp:** Restricts permissible system calls. Dangerous syscalls—such as direct hardware manipulation or attempts to load kernel modules—are intercepted and rejected before reaching the host kernel.

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture

Network Virtualization via Veth Pairs and Bridges

By default, LXD provisions a managed virtual bridge network (typically `lxdbr0`). Instances connect to this bridge through **Virtual Ethernet Pairs (`veth`)**:

A `veth` construct operates like a software-defined patch cable with two distinct endpoints:

- One end attaches to the host bridge as a dynamic interface (e.g., `vethabcdef12`).
- The opposite end is injected directly into the container's network namespace and initialized as `eth0`.

An embedded, lightweight `dnsmasq` process bound to the bridge automatically handles private IPv4 and IPv6 assignment via DHCP and provides local DNS resolution for instance names. The container communicates across the local network just like a bare-metal machine, without requiring manual network configuration on the physical data center switch.

Core LXD Components

Administering LXD differs fundamentally from environments where configuration files are scattered across disk. LXD manages infrastructure as logical, declarative objects stored in an embedded transactional database (`dqlite`) and exposed through the REST API.

Key Components Overview

- **LXD Daemon (`lxd`):** The primary engine. It tracks instance states, interfaces with the kernel and QEMU, monitors resource limits, and processes requests over the UNIX socket or network interfaces.
- **Command-Line Client (`lxc`):** The administrative interface. Through the concept of *remotes*, the same client manages both the local host and remote LXD endpoints.
- **Images:** Immutable templates used to build new instances. An image contains a compressed root filesystem of a clean distribution (such as Ubuntu 24.04, Debian 12, or Alpine 3.20). LXD fetches images from trusted remote registries (`ubuntu:`, `ubuntu-daily:`, `images:`), validates their cryptographic fingerprints, and stores them in a local cache.

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture

- **Instances:** The umbrella term for all workloads managed by LXD, covering both *system containers* and *virtual machines*. Every instance has a unique name, configuration keys (such as CPU and RAM limits), attached devices, and an execution status (`RUNNING`, `STOPPED`, `FROZEN`).
- **Profiles:** Reusable configuration blueprints. Profiles define hardware devices (such as root disks and network interfaces) and operational limits. New instances automatically inherit the `default` profile. Modifying a profile propagates changes across all linked instances immediately.
- **Storage Pools:** The storage abstraction layer. LXD supports advanced filesystems like ZFS and Btrfs, alongside LVM thin pools and standard directories (`dir`). Storage pools host instance filesystems, user-defined custom volumes, and instant copy-on-write snapshots.
- **Networks:** Managed network interfaces. Typically configured as software bridges (`lxdbr0`), OVN virtual overlays, or physical macvlan interfaces. LXD manages routing, DHCP address allocation for IPv4/IPv6, and internal DNS resolution automatically.
- **Projects:** Multi-tenancy and logical separation. Projects isolate instances, profiles, images, and storage volumes into independent administrative namespaces. Teams can separate development, staging, and production environments on a single host with distinct resource quotas.

Logical Interaction During Instance Startup

When an administrator runs `lxc launch ubuntu:24.04 web01`, these components coordinate across an established sequence:

1. **Fetch & Verify Image** LXD inspects the local image cache or retrieves the distribution image from the remote registry
2. **Allocate Storage Volume** A new root filesystem volume is cloned via copy-on-write inside the designated storage pool

3. **Inherit Profile Configuration** Root disk devices, network interfaces, and resource limits are mapped from the default profile

4. **Initialize Instance in Project** Namespaces, cgroups, and veth network interfaces are created, and the container init process begins execution

Common Use Cases

LXD bridges the gap between single-process application containers and heavy hypervisor clusters. In daily operations, it excels in scenarios that would otherwise prove too resource-intensive or operationally inflexible.

Practical Administration Scenarios

- **Development and Staging Environments:** Engineers and administrators can spin up clean Linux instances in seconds. Because an LXD container runs a standard `systemd` init process, services like Nginx, PostgreSQL, Redis, or SSH are configured exactly as on production bare-metal servers—eliminating the need for desktop hypervisors like VirtualBox.
- **Multi-Distribution Infrastructure on a Single Host:** A single host operating system (e.g., Ubuntu 24.04 LTS) can host parallel instances of Debian 12, Alpine Linux, Arch Linux, and AlmaLinux. This simplifies testing software portability and package compatibility across distinct Linux ecosystems.
- **Consolidation of Infrastructure Services:** Workloads traditionally assigned to individual virtual machines—such as BIND DNS servers, mail relays, VPN gateways, or Prometheus monitoring exporters—can be isolated inside system containers. The host saves substantial memory by avoiding redundant kernel memory allocations.
- **Ephemeral CI/CD Build Runners:** Continuous integration pipelines benefit from sub-second boot times and copy-on-write storage: An instance boots for a specific build job, compiles the code in an isolated environment, and is either destroyed immediately or rolled back to a clean snapshot.

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture

Decision Guide: System Containers Versus Virtual Machines

For daily operational decisions, follow this practical checklist:

Choose a system container when:

- The workload runs natively on Linux.
- Maximum I/O throughput and bare-metal CPU performance are essential.
- High instance density is required on hardware with limited physical RAM.
- Storage and memory should be shared dynamically with the host.

Choose a virtual machine when:

- A non-Linux operating system (such as Windows Server or FreeBSD) must be deployed.
- The application requires specific kernel modules or custom `/proc/sys/` parameters unavailable on the host kernel.
- Direct hardware passthrough (such as dedicated PCIe GPUs or cryptographic cards) is needed.
- Strict hardware-enforced boundaries are required between mutually untrusted tenants.

> **Operational Tip for System Maintenance:** Create snapshots before executing risky changes! Because LXD integrates directly with copy-on-write filesystems like ZFS or Btrfs, running `lxc snapshot web01 pre-upgrade` produces an instantaneous, crash-consistent restore point. If a distribution upgrade or configuration change fails, running `lxc restore web01 pre-upgrade` reverts the entire filesystem in milliseconds.

Prerequisites for Practical Work

Before proceeding to installation and initialization in Part 2 of this series, verify that your host system satisfies the technical baseline. LXD depends directly on modern Linux kernel features and storage backends.

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture

Mandatory Technical Requirements

LXD requires the following core platform capabilities:

- **64-Bit Operating System:** A modern 64-bit Linux distribution (x86_64 / amd64 or aarch64 / arm64). Common server distributions include Ubuntu Server (22.04 LTS or 24.04 LTS), Debian (11 or 12), and RHEL-derived distributions (such as AlmaLinux or Rocky Linux 9).
- **Linux Kernel Version:** Minimum kernel 5.4, though kernel 5.15 LTS or a modern 6.x release is strongly recommended for unified cgroups v2 support and advanced storage drivers.
- **Kernel Subsystems:** The host kernel must have User Namespaces (`CONFIG_USER_NS`), PID namespaces, Network namespaces, and Control Groups (`CONFIG_CGROUPS`) enabled. Standard server distribution kernels include these by default.
- **Hardware Virtualization for VMs (KVM):** Pure system containers do not require processor virtualization extensions. However, if you plan to run **virtual machines** under LXD, your CPU must support hardware virtualization (Intel VT-x with the `vmx` flag or AMD-V with the `svm` flag), enabled within the server BIOS/UEFI.

Production Recommendations

For stable and maintainable server operations, align your host with these guidelines:

- **Memory (RAM):** 4 GB RAM is sufficient for initial lab testing. Production requirements depend on active services: Because containers consume only the memory their processes actually use, a host with 16 GB of RAM can easily accommodate 20 to 40 lean Linux containers simultaneously. For virtual machines, assigned memory must be accounted for statically.
- **Storage Capacity & Drivers:** Reserve at least 30 to 50 GB of unallocated disk space. The choice of storage driver determines performance:
- **Recommended (Copy-on-Write):** ZFS or Btrfs. Both filesystems allow LXD to clone instances from base images in milliseconds and generate instant snapshots with zero initial storage duplication.
- **Alternative:** LVM with thin provisioning (`lvm-thin`).

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture

- **Testing Only:** The `dir` driver (standard folder hierarchy on ext4 or XFS). It relies on slow, full file copies (`rsync`) for snapshots and clones, making it unsuitable for production environments.
- **Network Connectivity:** Outbound internet access is necessary during setup so LXD can retrieve base distribution images from official remote registries. Additionally, the host must permit the creation of software bridge devices (`lxdbr0`).

> **Pre-Installation Verification:** You can verify whether your processor supports hardware virtualization by running a quick read-only check: `egrep -c '(vmx|svm)' /proc/cpuinfo`. If the command returns an integer greater than 0, your CPU is ready to run KVM-based virtual machines under LXD.

> ⚠ **Virtualization Inside Cloud Instances (Nested Virtualization):** If you run LXD on a cloud VM (e.g., Hetzner Cloud, AWS EC2, or DigitalOcean), system containers work out of the box. However, starting virtual machines (`--vm`) requires your cloud provider to enable *nested virtualization* for that specific instance type.

Command Reference (Cheatsheet)

lxc version

$ `Displays the installed client and server versions of LXD`

lxc info

$ `Outputs detailed operational status regarding the host, kernel drivers, and storage pools`

lxc list

$ `Lists all existing instances (containers and VMs) with their IP addresses and current runtime states`

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture

lxc image list

$ `Displays all operating system images stored in the local image cache`

lxc profile list

$ `Lists all configuration profiles available within the active project`

lxc network list

$ `Provides an overview of all managed and unmanaged network bridges on the host`

lxc storage list

$ `Lists all configured storage pools alongside their active driver types (ZFS, Btrfs, LVM)`

lxc project list

$ `Displays all configured projects used for administrative multi-tenancy on the host`

Further Resources

Conclusion

Understanding LXD's architecture establishes the foundation for efficient system container and virtual machine administration on Linux. LXD bridges the divide between single-process application containers and resource-heavy hypervisors by orchestrating native Linux kernel primitives—namespaces for visibility, cgroups v2 for resource management, and AppArmor/Seccomp for access security—through a centralized, REST-driven daemon.

The strict separation between the background daemon `lxd` and the CLI client `lxc`, the cohesive interaction of images, profiles, managed bridges, and copy-on-write storage pools, along with the flexibility to run containers and KVM virtual machines side by side, establish LXD as one of the most versatile virtualization platforms available for modern data center operations.

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture

> **Operational Rule of Thumb:** Memorize this core distinction for daily administration: *Namespaces govern what an instance can see; cgroups govern what an instance can consume.* Whenever an instance experiences performance degradation or services terminate unexpectedly, cgroup thresholds (`limits.memory` , `limits.cpu`) are the first diagnostic checkpoint.

In the next part of this series, we turn theory into practice: In **LXD – Part 2: Installation and Initial Configuration**, we prepare the host environment, execute `lxd init` step by step, configure the primary network bridge, and provision our first production ZFS storage pool.

Course Overview: All LXD Articles & Modules

Source: https://admindocs.de/en/lxd/lxd-1-understanding-fundamentals-and-architecture