Linux Kernel Notes

Concept-focused study notes. Commands and flags are tracked in a separate document. Please note: these notes are meant to coincide with the CompTIA Linux+ learning objectives which encompasses a massive amount of material. However, these notes are still helpful as a reference for anyone interested in Linux as I tried to include only the most popular and/or useful information and context that you will need day-to-day.

Domain 1.0: System Management

1.1 Basic Linux Concepts

Basic Boot Process

The Linux boot process is a chain where each stage hands control to the next. Understanding the order matters because troubleshooting often means identifying which link broke.

1. **Firmware (BIOS or UEFI)**: When power hits the system, the motherboard firmware runs first. It performs a POST (Power-On Self-Test), initializes hardware, and locates the boot device.
    - **BIOS (Basic Input/Output System)**: The legacy firmware standard. Uses the MBR (Master Boot Record) for partition information.
    - **UEFI (Unified Extensible Firmware Interface)**: The modern replacement. Supports larger drives via GPT (GUID Partition Table), faster boot times, and **Secure Boot**, which uses cryptographic signatures to verify the bootloader and kernel haven't been tampered with.
2. **Bootloader**: The firmware loads the bootloader from the boot device. Its only real job is to find and load the kernel into memory.
    - **GRUB (Grand Unified Bootloader)**: The dominant Linux bootloader.
    - **Bootloader Configuration Files**: The main configuration file for GRUB is typically `/boot/grub2/grub.cfg`. Administrators should *not* edit this directly; instead, they edit `/etc/default/grub` and use a regeneration command to rebuild the main config.
    - **Kernel Parameters**: Text-based instructions passed by the bootloader to the kernel at startup. They dictate hardware settings or force specific boot states (like single-user mode for password recovery).
3. **Kernel**: The core of the OS gets loaded into memory. The kernel manages CPU scheduling, memory, devices, and the security model. Linux uses a **monolithic**

kernel design, meaning most core services run in kernel space — but it supports **loadable kernel modules** so drivers can be added or removed without recompiling.

4. **Initial RAM Disk (initrd) / initramfs**: A highly compressed, temporary root filesystem loaded directly into RAM. It provides the kernel with the essential drivers needed to mount the *real* root filesystem on the physical disk. Without it, the kernel wouldn't know how to read the storage controller or filesystem.
5. **init system (systemd)**: Once the real root filesystem is mounted, the kernel hands control to PID 1 — the init system. On modern Linux, this is **systemd**, which then starts every other process required to bring the system up to its target state (multi-user, graphical, etc.).

Preboot Execution Environment (PXE)

A standard network protocol that allows a server to boot an operating system from a central network server (using DHCP to get an IP and TFTP to fetch the boot image) rather than relying on a local hard disk. Useful for diskless workstations and mass deployment.

Filesystem Hierarchy Standard (FHS)

The FHS defines a consistent directory structure across Linux distributions.

- **/** (root): The top of the entire directory tree. Everything mounts beneath this single point. Not to be confused with `/root`, which is the home directory of the root user.
- **/bin**: Essential command binaries (like `ls` or `cp`) that must be available to all users in any boot state. Nowadays /bin is actually a symbolic link to /usr/bin
- **/boot**: Static files necessary for the bootloader to function, including the kernel image (vmlinuz) and the initrd.
- **/dev**: A virtual directory containing "device files" that let the OS interact with attached physical hardware (hard drives, keyboards, terminals) as if they were standard files.
- **/etc**: The central repository for all host-specific system and application configuration files. The name historically meant "et cetera."
- **/home**: The standard location for user home directories, storing personal data and user-specific configuration files (dotfiles).
- **/lib**: Essential shared libraries (similar to .dll files in Windows) needed by the binaries in `/bin` and `/sbin`, plus loadable kernel modules.

- **/proc**: A pseudo-filesystem that exists entirely in RAM, providing a real-time window into the running kernel and active processes. Each running process gets its own numbered directory.
- **/sbin**: System binaries reserved for the root user or system administration tasks (e.g., `fdisk`, `reboot`). Nowadays /sbin is actually a symbolic link to /usr/sbin
- **/tmp**: Temporary files, generally cleared automatically upon system reboot. World-writable but protected by the sticky bit.
- **/usr**: A secondary hierarchy for non-essential, read-only user data, applications, and libraries. Contains `/usr/bin`, `/usr/sbin`, `/usr/lib` — the "user system resources."
- **/var**: Variable data that changes frequently during normal operation: log files (`/var/log`), database stores, mail spools, print queues, and caches.

Server Architectures

The CPU instruction set the OS was compiled for. A binary built for one architecture will not run on another.

- **AArch64**: 64-bit ARM architecture. Common in mobile devices, Raspberry Pi 4+, Apple silicon, and increasingly in cloud and lightweight servers (e.g., AWS Graviton).
- **RISC-V**: An open-standard, royalty-free architecture gaining traction in embedded and research systems.
- **x86**: Legacy 32-bit architecture (also called i386 or i686). Largely deprecated for servers.
- **x86_64 / AMD64**: The dominant 64-bit architecture for desktops and servers. Both Intel and AMD CPUs use it despite the "AMD" name.

Distributions

Linux families are categorized primarily by how they manage software packages.

- **RPM-based**: Red Hat, Fedora, CentOS, Rocky Linux, AlmaLinux, openSUSE/SLES. Use `.rpm` package files.
- **dpkg-based**: Debian, Ubuntu, Linux Mint. Use `.deb` package files.

Other distinctions include release model (rolling vs. fixed) and target audience (enterprise vs. desktop vs. embedded).

Graphical User Interface (GUI)

The visual desktop environment is a stack of cooperating components:

- **Display managers**: The login screen (GDM, SDDM, LightDM). Handles authentication and launches a session.
- **Window managers**: Control how application windows behave — resizing, moving, decorations, virtual desktops.
- **Display servers**: The protocol applications use to draw on the screen.
    - **X Server (X11)**: The long-standing legacy protocol. Allows network-transparent graphics (you can run a GUI app on a remote server and display it locally).
    - **Wayland**: The modern replacement. More secure, simpler, and better at compositing, but with less mature remote display support.

Software Licensing

Understanding the legal frameworks around code is essential for compliance and audits.

- **Open-source software**: Source code is publicly available for inspection. Doesn't automatically grant modification or redistribution rights — those depend on the specific license.
- **Free software**: Respects users' freedoms to run, study, copy, modify, and distribute. "Free" refers to liberty, not price (as in "free speech, not free beer").
- **Proprietary software**: Closed-source, owned by a corporation. Users get a license to use, not the underlying code.
- **Copyleft**: A licensing rule requiring that any modified versions of the software must also be distributed under the same open license. The GPL is the most famous copyleft license. Prevents "embrace and extend" tactics that would turn open code proprietary.

1.2 Linux Device Management

Kernel Modules

The Linux kernel is monolithic but modular. Instead of compiling every possible driver into the kernel (which would make it enormous), the kernel supports **loadable kernel modules (LKMs)** — chunks of code that can be inserted into the running kernel on demand and removed when no longer needed.

- **Why modules matter**: New hardware can be supported without rebooting, and unused drivers don't waste memory.

- **Module dependencies**: Many modules depend on other modules. The kernel maintains a dependency map so the right modules load in the right order.
- **Module information**: Each module carries metadata (author, license, parameters it accepts, hardware IDs it supports).
- **Where modules live**: Compiled module files (`.ko` — "kernel object") are stored under `/lib/modules/<kernel-version>/`. Each kernel has its own module directory.

Device Management Concepts

Linux exposes hardware as files under `/dev`. Several subsystems coordinate to make this work:

- **dmesg ring buffer**: A circular kernel log buffer. Messages from boot, driver loads, and hardware events accumulate here. Essential for diagnosing hardware detection issues.
- **DMI (Desktop Management Interface) / SMBIOS**: A firmware-level table describing the system's manufacturer, model, serial number, BIOS version, installed memory slots, etc. Tools can query this without opening the case.
- **IPMI (Intelligent Platform Management Interface)**: A specification for out-of-band server management — power cycling, monitoring fan speeds and temperatures, even console access — independent of the OS. Used heavily in datacenter environments.
- **Hardware sensors**: Temperature, voltage, and fan speed sensors exposed by the kernel via the `lm_sensors` framework. Critical for monitoring server health.
- **Hardware inventory tools**: The kernel exposes CPU details, memory layout, PCI devices, and USB devices through various pseudo-filesystems. Inventory utilities read these to build a picture of the system.

initrd Management

The initrd (or its successor, initramfs) is rebuilt periodically — for example, after a kernel update, after adding a new storage driver, or after changing encryption setup.

- **Why rebuild?**: The initrd must contain drivers matching the *current* hardware and kernel. If you upgrade the kernel and forget to regenerate the initrd, the new kernel may not be able to mount the root filesystem.
- **Where it lives**: Inside `/boot`, with a filename that ties it to a specific kernel version.

Custom Hardware

Linux runs on more than commodity servers:

- **Embedded systems**: IoT devices, routers, industrial controllers, automotive systems. Often run stripped-down, real-time, or read-only Linux builds.
- **Graphics Processing Unit (GPU) use cases**: Machine learning training and inference, scientific computing, video rendering, and cryptocurrency-adjacent workloads. Requires vendor drivers (NVIDIA's proprietary driver, AMD's open ROCm stack) and monitoring tools to track GPU utilization, temperature, and memory.

1.3 Storage Management

Partitions

Partitions divide a single physical disk into independent logical units. Each can hold its own filesystem.

- **MBR (Master Boot Record)**: The legacy partition scheme. Limited to 4 primary partitions (or 3 primary + 1 extended containing logical partitions), and 2 TB maximum disk size.
- **GPT (GUID Partition Table)**: The modern scheme used with UEFI. Supports up to 128 partitions by default and disks well beyond 2 TB. Each partition has a globally unique identifier (GUID).
- **UUID (Universally Unique Identifier)**: A persistent identifier for a filesystem or partition. Preferred over device names (like `/dev/sda1`) in `/etc/fstab` because device names can change between boots while UUIDs do not.

Logical Volume Manager (LVM)

LVM is an abstraction layer between physical disks and filesystems. It lets you resize "partitions" on the fly and combine multiple disks into one pool.

- **Physical Volume (PV)**: The foundational block — raw physical storage drives or partitions formatted and initialized for LVM.
- **Volume Group (VG)**: A unified storage pool created by logically grouping one or more Physical Volumes together. Think of it as the "pool" of available space.
- **Logical Volume (LV)**: The virtual, resizable partitions carved out from a Volume Group. This is what the OS formats with a filesystem and mounts.
- **Partition**: The basic, non-LVM physical subunit of a disk.

LVM benefits include online resizing, snapshots, and the ability to span multiple physical disks.

Filesystem Formats

- **xfs**: A high-performance, 64-bit journaling filesystem designed for massive files and parallel I/O. Default in RHEL/Rocky/Alma. Can grow but not easily shrink.
- **ext4**: The traditional, reliable, widely supported Linux filesystem. Default in Debian/Ubuntu. Supports journaling and grows or shrinks.
- **btrfs**: An advanced copy-on-write filesystem with built-in drive pooling, transparent compression, snapshots, and checksumming. Default in some openSUSE variants.
- **tmpfs**: A filesystem that lives entirely in RAM (with swap as overflow). Incredibly fast but completely volatile — data is lost on reboot or unmount. Used for `/tmp` and `/run` on many distributions.
- **FUSE (Filesystem in Userspace)**: A framework that lets non-privileged users implement filesystems in user space. Enables exotic filesystems like SSHFS (mounting remote directories over SSH) and NTFS-3G.

Journaling

Most modern Linux filesystems are *journaling* filesystems — they record planned changes to a special log (journal) before writing to disk. If the system crashes mid-write, the journal lets the filesystem recover quickly without scanning the entire disk.

Mount Options

Parameters passed when mounting drives to dictate their behavior:

- **nofail**: Allows the system to continue booting even if the drive is missing. Important for removable or network drives in `/etc/fstab`.
- **nosuid**: Ignores SUID security bits for safety — prevents privilege escalation via files on that mount.
- **remount**: Applies new options without unmounting and remounting from scratch.
- **ro / rw**: Sets read-only or read-write access.
- **noatime / nodiratime**: Disables access-time updates to save disk I/O. Significant performance gain on busy filesystems.
- **nodev**: Prevents the creation/use of device files on that mount.
- **noexec**: Blocks the execution of any binaries on that drive. Common for `/tmp` and user-writable mounts to limit malware.

Network Mounts

Accessing storage over a network:

- **NFS (Network File System)**: The native protocol for Linux-to-Linux file sharing. Lightweight and tightly integrated with Unix permissions.
- **SMB (Server Message Block) / Samba**: Used for seamless compatibility with Windows. **CIFS (Common Internet File System)** is an older dialect of SMB. Samba is the Linux implementation that lets a Linux box act as a Windows file/print server.
- **NAS (Network-Attached Storage)**: A dedicated file-serving appliance accessed over the network.
- **SAN (Storage Area Network)**: A dedicated network providing block-level storage (the OS sees raw disks, not a filesystem). Higher performance and complexity than NAS.

Inodes

Core data structures on a filesystem that store all the metadata about a file — permissions, ownership, timestamps, size, and the physical disk block locations of the data — but intentionally *do not* store the file's name or its actual data contents.

- Filenames live in *directory entries*, which map a name to an inode number.
- Hard links work because multiple filenames can point to the same inode.
- A filesystem can run out of inodes even with free disk space — common on filesystems with huge numbers of tiny files.

RAID (Redundant Array of Independent Disks)

A method of combining multiple drives for performance, redundancy, or both. Linux can implement RAID in software via the kernel's MD (multiple devices) driver.

- **RAID 0 (striping)**: Data spread across drives. Fast, but zero redundancy — one drive failure loses everything.
- **RAID 1 (mirroring)**: Identical copies on two or more drives. Full redundancy, but you only get the capacity of one drive.
- **RAID 5**: Striping with distributed parity. Survives one drive failure. Needs at least 3 drives.
- **RAID 6**: Like RAID 5 but with two parity blocks. Survives two simultaneous drive failures. Needs at least 4 drives.
- **RAID 10 (1+0)**: Mirrored pairs that are then striped. Combines speed and redundancy. Needs at least 4 drives.
- **/proc/mdstat**: A pseudo-file showing the current state of all software RAID arrays — useful for spotting degraded arrays.

Mounted Storage Files

- **/etc/fstab**: The permanent configuration file defining mounts that should automatically persist across reboots. Identified by UUID, label, or device path.
- **/etc/mtab and /proc/mounts**: Dynamic, system-generated files that display the exact filesystems currently mounted right now.
- **autofs**: A service that automatically mounts remote filesystems (NFS or SMB) only when a user or process accesses the mount point, and unmounts them after inactivity. Saves network bandwidth and resources compared to persistent mounts.

Performance Concepts

- **IOPS (Input/Output Operations Per Second)**: A key storage performance metric — how many discrete read/write operations a drive can handle per second.
- **Throughput**: How much data (MB/s or GB/s) moves through the storage system.
- **Latency**: How long an individual I/O operation takes from request to completion.
- **NVMe (Non-Volatile Memory Express)**: A modern interface for SSDs that bypasses the bottlenecks of older storage protocols, delivering much higher IOPS and lower latency.

1.4 Network Services and Configuration

Network Configuration Files

- **/etc/hosts**: Local, static DNS resolution; the system checks this file before querying external DNS servers. Useful for testing and small networks.
- **/etc/resolv.conf**: Stores the IP addresses of the DNS servers the system uses. On modern systems, this is often managed automatically by NetworkManager or systemd-resolved.
- **/etc/nsswitch.conf**: Determines the order in which the system looks up various types of information (hostnames, users, groups, etc.) — for example, "check `/etc/hosts` first, then DNS."
- **/etc/netplan**: A directory containing YAML files used to configure modern Netplan network setups (heavily used in Ubuntu). Netplan is a *front-end* that renders configuration for whichever back-end is in use (NetworkManager or systemd-networkd).

Network Configuration Daemons

- **NetworkManager**: A dynamic network control and configuration daemon. The default on RHEL/Fedora families. Abstracts complex network interfaces (Ethernet, Wi-Fi, VPN, bonding) and keeps connections active across changes.
- **systemd-networkd**: A lightweight networking daemon shipped with systemd, often used on servers and minimal installations.

Common Networking Concepts

- **TCP (Transmission Control Protocol)**: Connection-oriented, reliable, ordered delivery. Used for web traffic, SSH, email, etc.
- **UDP (User Datagram Protocol)**: Connectionless, low-overhead, no delivery guarantee. Used for DNS lookups, streaming, VoIP.
- **ICMP (Internet Control Message Protocol)**: Used for diagnostic and error messages — `ping` and `traceroute` rely on it.
- **FQDN (Fully Qualified Domain Name)**: The complete domain name including all parent domains (e.g., `mail.example.com.`).
- **TTL (Time to Live)**: A counter in IP packets that limits how many hops they can take before being discarded — prevents packets from looping forever. Also used in DNS to control how long records are cached.
- **MTU (Maximum Transmission Unit)**: The largest packet size that can be sent over a network link without fragmentation. Standard Ethernet MTU is 1500 bytes; "jumbo frames" use 9000.
- **NIC (Network Interface Card)**: The physical or virtual network adapter.

1.5 Shell Operations

Common Environmental Variables

Dynamic values that affect how processes run:

- **$DISPLAY**: Tells graphical apps where to send their output (e.g., `:0` for the local display).
- **$HOME**: The path to the current user's home directory.
- **$PATH**: An ordered, colon-separated list of directories the shell searches when you type a command.
- **$PS1**: Defines the visual layout of your command prompt.
- **$SHELL**: Identifies the current active shell.
- **$USER**: Identifies the currently logged-in account.

Paths

- **Absolute path**: Always starts from the root directory (`/`) and works from anywhere.
    - **`/`**: The root directory — the very top of the directory tree.
    - **`~`**: A shortcut for the current user's home directory (e.g., `/home/username`).
- **Relative path**: Starts from your current working directory.
    - **`.`**: The current directory.
    - **`..`**: One level up. `../..` goes two levels up.
    - **`-`**: A shortcut representing the *previous* working directory you were in.

Shell Environment Configurations

- **Global environment configs**: System-wide scripts executed for any user's login shell.
    - **/etc/profile**: The first file loaded for any user's login shell. Sources scripts from `/etc/profile.d/`.
- **User environment configs**: Hidden scripts executed automatically upon user login to set up their environment.
    - **~/.bashrc**: Loaded for every non-login interactive shell. This is where aliases and shell functions typically live.
    - **~/.bash_profile**: Loaded by Bash for a login shell after the global profile.
    - **~/.profile**: A fallback for shells that aren't Bash. If `~/.bash_profile` exists, Bash ignores this file.

Login Shell vs. Non-Login Shell

- A **login shell** is started when you log in (SSH, TTY console, `su -`). It reads `/etc/profile` and `~/.bash_profile` (or `~/.profile`).
- A **non-login shell** is started when you open a terminal inside an existing session or run `bash` again. It reads `~/.bashrc`.
- This is why aliases defined in `~/.bashrc` don't appear in fresh SSH logins unless `.bash_profile` sources `.bashrc`.

Channel Redirection

- **`>`** (overwrite): Sends output to a file as stdout instead of the screen. Overwrites the file if it exists.
- **`>>`** (append): Sends output to a file, appending to the end if the file already exists.
- **`<`** (redirect input): Feeds the contents of a file into a command as stdin.

- `<<` (Here-Document): Allows you to type multiple lines of input from a multi-line block directly into a script or command until a specific "end word" (usually `EOF`) is typed.
- `<<<` (Here-String): Passes a single string or variable as stdin directly into a command.
- `|` (pipe): Connects two commands so the output of the first becomes the input of the second.

Shell Special Characters

- `#`: Comment marker. When it appears as `#!` at the very first line of a script, it is a **Shebang** telling the OS which interpreter to use (e.g., `#!/bin/bash`).
- `;`: Command separator. Run the next command regardless of whether the previous one succeeded or failed.
- `&`: Background. Placing this at the end of a command runs that process in the background.
- `&&`: Run the next command only if the previous one succeeded (exit code 0).
- `||`: Run the next command only if the previous one *failed*.
- `()`: Subshell. Commands inside parentheses run in a separate subshell environment — variable changes don't affect the parent shell.
- `$`: Variable expansion. Used to access the value stored in a variable (e.g., `$USER`).
- `${ }`: Parameter expansion. A more robust way to expand variables, allowing string manipulation or protecting the variable name from surrounding text.
- `$( )`: Command substitution. Runs a command and replaces the syntax with the command's output. Backticks `` ` ` `` do the same thing but are older syntax.
- `@`: Usually seen as `$@`, it represents all positional parameters passed to a script.
- `"` (double quote): Weak quoting. Most special characters are treated as literal text, but variables (`$`) and command substitutions still expand.
- `'` (single quote): Strong quoting. Everything inside is treated literally; no expansions occur.
- `\` (escape): Tells Bash to treat the very next character as literal text.
- `*`: Matches zero or more characters in a filename (glob).
- `?`: Matches exactly one character in a filename.
- `$?`: Returns the exit status of the last executed command. `0` means success; anything else (1–255) indicates an error.

- `!`: In scripts, represents logical NOT. In an interactive shell, `!!` repeats the last command.

Standard Streams

- **stdin (Standard Input)**: Data fed into a command (typed input or piped from a file). File descriptor `0`.
- **stdout (Standard Output)**: Normal command output. File descriptor `1`.
- **stderr (Standard Error)**: Where error messages go. File descriptor `2`.

Text Editors (Concept)

- **vi/vim**: A modal editor — separate modes for inserting text and issuing commands. Universally available, even on minimal systems. Knowing the basics is essential because vi is often the only editor on a recovery shell.
- **nano**: A modern, modeless editor that's much friendlier for beginners. Commands are visible at the bottom of the screen.

1.6 Backup and Restore Concepts

A backup strategy answers three questions: *What gets backed up? How often? Where does it live?* Good backups are tested by actually restoring from them — an untested backup is just a hope.

Backup Types

- **Full backup**: A complete copy of all data. Easy to restore from but slow to create and storage-heavy.
- **Incremental backup**: Only files that changed since the *last backup of any type*. Fast and small, but restoring requires the original full backup plus every subsequent incremental.
- **Differential backup**: Only files that changed since the last *full* backup. Larger than incremental but simpler to restore (full + most recent differential).

Archiving vs. Compression

These are two distinct operations often combined.

- **Archiving**: Bundling many files and directories into a single file while preserving permissions, ownership, and structure. By itself, it doesn't reduce size.

- **Compression**: Reducing the size of a file by encoding it more efficiently. Different algorithms trade speed for compression ratio.

Common Compression Algorithms

- **gzip**: The classic, fast, moderate compression. Files end in `.gz`. Ubiquitous.
- **bzip2**: Slower than gzip but better compression. Files end in `.bz2`.
- **xz**: Slow to compress, fast to decompress, with very high compression ratios. Files end in `.xz`. Common for distributing source tarballs and kernel images.
- **7-Zip**: A compression suite that supports many formats and very high ratios. Files often end in `.7z`.
- **zip / unzip**: The traditional cross-platform archive-and-compress format. Familiar from Windows.

Synchronization

- **rsync**: A tool concept based on transferring only the *differences* between source and destination, rather than copying everything every time. Essential for efficient remote backups and mirroring.

Disk-Level Tools

- **dd**: Performs raw, block-level copies. Used to clone entire disks, create disk images, or write OS installer images to USB drives. Powerful but dangerous — there's no "undo."
- **ddrescue**: A specialized version of `dd` designed for recovering data from failing drives. Skips bad sectors and tries again later, salvaging as much as possible.

Backup Storage Considerations

- **Onsite vs. offsite**: Local backups are fast to restore but vulnerable to the same disaster (fire, flood, ransomware) that hit the original. Offsite copies protect against location-wide events.
- **The 3-2-1 rule**: Three copies of data, on two different media types, with one copy offsite.
- **Immutability**: Backups that cannot be modified or deleted for a defined retention period — critical defense against ransomware that targets backup repositories.

1.7 Virtualization

Bare Metal vs. Virtual Machines

- **Bare metal**: Installing the Linux OS directly onto physical hardware. Maximum I/O and CPU performance but minimal flexibility.
- **Virtual machines**: Abstracting the hardware through a hypervisor. You trade a small percentage of raw performance for immense flexibility, portability, and hardware sharing.

Hypervisor Types

- **Type 1 (bare-metal)**: Runs directly on hardware. KVM, VMware ESXi, Microsoft Hyper-V.
- **Type 2 (hosted)**: Runs as an application on top of a host OS. VirtualBox, VMware Workstation.

Linux Hypervisors

- **QEMU (Quick Emulator)**: Acts as a hardware emulator. Can emulate entire CPUs (useful for cross-architecture work).
- **KVM (Kernel-based Virtual Machine)**: Turns the Linux kernel itself into a bare-metal Type 1 hypervisor. Often paired with QEMU for device emulation.
- **libvirt**: A management abstraction layer that provides a unified API for KVM, QEMU, Xen, and others.

Virtual Machine Concepts

- **Paravirtualized drivers (VirtIO)**: Drivers that *know* they're running inside a VM and communicate efficiently with the hypervisor, instead of pretending to be physical hardware. Significantly faster than fully emulated devices.
- **Disk image operations**: Converting formats (e.g., raw to qcow2), resizing, inspecting properties.
    - **qcow2 (QEMU Copy-on-Write v2)**: The standard QEMU disk format. Supports snapshots, thin provisioning, and compression.
- **VM states**: Running, paused, suspended (saved to disk), stopped.
- **Nested virtualization**: Running a VM inside a VM. Requires CPU support and is invaluable for testing hypervisor configurations.
- **SR-IOV (Single Root I/O Virtualization)**: A feature that lets a single physical PCIe device (typically a network card) present itself as multiple virtual devices to multiple VMs, with near-bare-metal performance.

VM Operations and Resources

VMs are allocated finite slices of host resources: storage, RAM, CPU cores, and network bandwidth. Over-committing CPU is generally safe; over-committing RAM is risky and can lead to thrashing.

- **Baseline image templates**: Pre-built, hardened images used to deploy identical VMs quickly.
- **Cloning**: Duplicating a VM, optionally with a new identity (MAC address, hostname).
- **Migrations**: Moving a running or stopped VM between physical hosts. Live migration moves a running VM with negligible downtime.
- **Snapshots**: Point-in-time captures of a VM's state, allowing easy rollback. Should be treated as short-term safety nets, not long-term backups.

VM Network Types

- **Bridged**: The VM connects directly to the physical network as if it were a separate physical machine with its own IP.
- **NAT (Network Address Translation)**: The VM hides behind the host's IP to access the network. Outbound works, inbound requires port forwarding.
- **Host-only / isolated**: The VM can only talk to the host and other VMs on the same isolated network. No external access.
- **Routed**: Traffic is routed between the VM network and the physical network without NAT.
- **Open**: Completely unrestricted networking.

Domain 2.0: Services and User Management

2.1 Files and Directories

Links

- **Symbolic links (symlinks)**: Act as shortcuts that point to a file's *path*. If the target moves or is deleted, the symlink breaks (becomes a "dangling" link). Can span filesystems and point to directories.
- **Hard links**: Act as direct pointers to the same underlying inode. The file data persists as long as at least one hard link exists — deleting one hard link doesn't delete the data. Cannot span filesystems or point to directories.

Device Types in /dev

- **Block devices**: Hardware that stores and moves data in fixed-size blocks (typically 512 or 4096 bytes). Storage drives are the classic example. Block device files start with `b` in a long listing.

- **Character devices**: Hardware that transfers data sequentially byte-by-byte (keyboards, mice, serial ports). Character device files start with `c`.
- **Special character devices**: Pseudo-devices that serve unique kernel functions:
    - **/dev/null**: A "black hole" — anything written to it is discarded. Useful for silencing output.
    - **/dev/zero**: An infinite stream of null bytes. Used to fill files or wipe disks with zeros.
    - **/dev/random** and **/dev/urandom**: Sources of cryptographic-quality random bytes. `urandom` doesn't block waiting for entropy.
    - **/dev/tty**: A reference to the current controlling terminal.

2.2 Local Account Management

User Profile Templates

- **/etc/profile**: Sets system-wide environmental variables for all users. Any settings here apply to every user on the system. Executed for login shells only (SSH, TTY).
    - **Environment variables**: Sets defaults like `PATH`, `USER`, `HOSTNAME`.
    - **umask**: Often defines the default user file-creation mode mask, which determines default permissions for new files and directories.
    - **/etc/profile.d/**: Instead of one giant file, modern distributions use `/etc/profile` to source individual `.sh` scripts from this directory. Cleaner and modular.
- **/etc/skel**: A "skeleton" template — any files inside it are automatically copied into the home directory of newly created users.
    - When an administrator creates a new user with the home-directory flag, the system copies every file and subdirectory from `/etc/skel` into the new home directory, then changes ownership to the new user.
    - Use this to give every new user a default `.bashrc`, welcome documents, or organization-specific config.

Account Files

- **/etc/passwd**: Stores basic user account information. Despite the name, it no longer contains passwords on modern systems. World-readable — it's the public directory of users and groups.
    - **Fields in /etc/passwd**:
        - **Username**: The unique login name.

        - **Password placeholder**: Usually `x`, indicating that the encrypted password is stored in `/etc/shadow`.
        - **UID (User Identifier)**: The numeric ID for the user.
        - **GID (Group Identifier)**: The numeric ID for the user's primary group.
        - **GECOS**: A comment field, typically used for the user's full name or office details.
        - **Home directory**: The absolute path to the user's personal directory.
        - **Login shell**: The path to the shell that starts when the user logs in (e.g., `/bin/bash` or `/sbin/nologin`).
- **/etc/shadow**: Stores the encrypted password and account expiration details. Readable only by root.
    - **Fields in /etc/shadow**:
        - **Encrypted password**: The hashed password. `!!` or `*` means the account is locked.
        - **Date of last change**: Days since the Unix epoch (Jan 1, 1970) when the password was last changed.
        - **Minimum password age**: Days a user must wait before changing their password again.
        - **Maximum password age**: Days after which a user must change their password.
        - **Warning period**: Days before expiration that the user starts receiving warnings.
        - **Inactivity period**: Days after a password expires before the account is officially disabled.
        - **Expiration date**: An absolute date (in days since the epoch) when the account expires.
        - **Reserved field**: Currently unused.
- **/etc/group**: Defines the groups to which users belong.
    - **Fields**:
        - **Group name**: The unique name.
        - **Group password**: Usually `x`; group passwords are rarely used.
        - **GID**: The unique numeric identifier for the group.
        - **Group list**: A comma-separated list of supplementary members.
- **/etc/gshadow**: Stores secure group information (group passwords and administrators). Companion to `/etc/group`.

Expiration Configuration Files

While `/etc/shadow` holds per-user expiration data, the *defaults* for password aging, expiration policies, and home directory creation are dictated by:

- **/etc/login.defs**: System-wide login settings — default UID/GID ranges, password aging defaults, mail directory.
- **/etc/default/useradd**: Default values used when creating new users (default shell, default group, skeleton directory, etc.).

Attributes

- **UID (Unique Identifier)**: The numerical ID of a user.
    - **Root account (UID 0)**: The superuser always occupies UID 0. *Any* account with UID 0 has full root privileges, regardless of its name.
    - **System accounts (UID 1–999)**: Created by the OS to run core components like the kernel, hardware drivers, or system-level utilities.
    - **Service accounts (UID 1–999)**: A subset of system accounts used to run specific services (Apache, Nginx, Samba). Typically have `/sbin/nologin` as their shell so no one can log in as them interactively.
    - **User accounts (UID 1000+)**: Standard accounts for humans.
- **GID (Group Identifier)**: The numerical ID of the group.
- **EUID (Effective User Identifier)**: The ID the kernel uses to determine file access permissions for a *running* process. Usually the same as the real UID, but changes when a setuid binary runs.
- **EGID (Effective Group Identifier)**: The group equivalent of EUID — used by the kernel for permission checks during process execution.

2.3 Process Management

Process Verification via /proc/<PID>

Instead of traditional flags, you "read" the files inside this directory to inspect a process:

- **cmdline**: The full command line used to start the process.
- **environ**: All environmental variables set for that specific process.
- **exe**: A symbolic link to the actual executable on disk.
- **fd/**: A subdirectory containing symbolic links to every file the process has currently open (file descriptors).
- **maps**: The memory regions mapped by the process — useful for identifying memory leaks.
- **status**: Human-readable information including the process state and PPID.

- **limits**: The soft and hard resource limits for the process.

Process ID

- **PID (Process Identification Number)**: A unique number assigned to every running process.
- **PPID (Parent Process Identification Number)**: Identifies the process that spawned this one.
- **PID 1**: Always the init system (systemd on modern Linux). All other processes descend from PID 1.

Process States

- **Running / Runnable (R)**: The process is either currently using a CPU core or is in the run queue ready to work. A high number of R-state processes usually correlates with a high load average.
- **Sleeping — Interruptible (S)**: Waiting for an event (user input, time, network packet). Will wake immediately on a signal. Most processes on a healthy system are in this state and consume almost no CPU.
- **Blocked / Uninterruptible Sleep (D)**: A "heavy" sleep, usually waiting on hardware I/O (disk read, slow NFS mount). Cannot be killed or interrupted by signals until I/O finishes. Many D-state processes indicate a disk/storage bottleneck.
- **Stopped (T)**: Suspended manually (Ctrl+Z or SIGSTOP). Stays in memory but does nothing until receiving SIGCONT.
- **Zombie (Z)**: Technically dead — the process has finished — but still has an entry in the process table waiting for its parent to acknowledge its exit code (a process called "reaping"). Zombies don't use CPU or memory but do consume PID slots. You cannot kill a zombie directly; you must fix or restart the parent process.

Process Limits

OS-level restrictions that prevent a single process from consuming all available CPU, memory, file descriptors, or other resources. Configured in `/etc/security/limits.conf` and similar locations.

Process Priority (Niceness)

The Linux kernel scheduler determines which processes get CPU time based on a "niceness" value ranging from **-20 to 19**:

- **-20**: Highest priority — the process aggressively demands CPU.
- **0**: Default.

- **19**: Lowest priority — the process yields CPU to almost anything else.

Only root can set negative (more aggressive) niceness values.

Signals

Internal messages sent by the kernel or users to control processes:

- **1 HUP (hangup)**: Asks a daemon to reload its configuration without restarting.
- **9 KILL**: Forces the kernel to immediately destroy the process. Cannot be caught or ignored.
- **15 TERM**: Asks the process to shut down gracefully. The default polite "please exit" signal.
- **2 INT**: Interrupt (what Ctrl+C sends).
- **18 CONT**: Continue (resume a stopped process).
- **19 STOP**: Stop (cannot be caught or ignored).
- **20 TSTP**: Terminal stop (what Ctrl+Z sends).

Job Control

- **Foreground**: A job that has control of the terminal — you can type to it.
- **Background**: A job running without terminal control. Appending & to a command launches it in the background.
- **Suspended**: A job paused with Ctrl+Z. Can be resumed in the foreground or moved to the background.
- **nohup**: A way to launch a process so it survives the parent shell logging out.

Scheduling

Linux provides several tools for running tasks on a schedule:

- **cron / crontab**: The classic scheduler. Runs jobs at exact times (e.g., "every weekday at 3 AM"). Assumes the system is running at that time.
- **anacron**: Designed for systems that aren't running 24/7 (laptops, workstations). Tracks when jobs last ran and runs missed jobs at the next opportunity, rather than skipping them.
- **at**: One-shot scheduling — run a command once at a specific future time.
- **systemd timers**: A modern alternative to cron, integrated with systemd. More flexible (calendar expressions, dependencies on other units) and produces structured logs.

2.4 Software Management

Software Concepts

- **Repositories**: Centralized servers hosting software packages and their dependency maps. Package managers like `dnf`, `apt`, and `zypper` pull from these.
- **Source installation**: Compiling code manually from source. Slower and more error-prone, but allows custom build options. Common when you need a feature not enabled in the distro's package.
- **Package dependencies and conflicts**: Most software depends on other software (libraries, runtimes). Package managers automatically resolve dependencies and refuse installs that would create conflicts.
- **Package manager**: A specialized tool that automates installing, upgrading, configuring, and removing software on a Linux system. Handles dependencies, verifies signatures, and tracks installed files.

Repository Management

Administrators often need to securely add, enable, or disable third-party repositories:

- **EPEL (Extra Packages for Enterprise Linux)**: A community repository for RHEL-based systems, providing open-source admin tools not in the official channels.
- **PPAs (Personal Package Archives)**: Used on Ubuntu/Debian to access software not provided by the distribution's core repositories.
- **Enabling/disabling**: Repositories can be toggled on or off without removing them — useful for testing or for repos that should only be active during specific maintenance windows.
- **Package and repository exclusions**: Pinning lets you prevent specific packages from being upgraded (or ensure they only come from a specific repo). Useful for keeping a tested version of a critical service.
- **Update alternatives**: A system for managing multiple parallel installations of the same kind of software (e.g., multiple Java versions or both `vi` and `vim`), so the administrator can pick which one is the "default" without uninstalling the others.

Language-Specific Package Managers

While `apt` and `dnf` manage OS-level packages, developers use language-specific managers to install dependencies directly from community registries:

- **pip**: Python packages from PyPI.

- **npm**: Node.js packages from the npm registry.
- **cargo**: Rust crates from crates.io.
- These tools should be confined to project-specific virtual environments rather than installed system-wide, to avoid conflicting with OS packages.

Software Verification with GPG

GPG (GNU Privacy Guard) signatures provide both package authenticity and integrity. When you pull software from enterprise repositories, GPG signatures act as your shield against malicious code injection.

- **RPM-based ecosystems (RHEL, Rocky, Alma, Fedora)**: Repository configuration files live under `/etc/yum.repos.d/`. A configuration with `gpgcheck=1` tells the package manager to strictly refuse to install any package from that repo unless it has a valid signature. The `gpgkey=` line points to the file or URL of the trusted public key.
- **Debian-based ecosystems**: Public keys live in `/etc/apt/keyrings/`. Repository entries in `/etc/apt/sources.list` include a `[signed-by=...]` directive ensuring that repository's packages can only be verified against that specific trusted key.

Sandboxed Applications

Modern Linux supports application formats that bundle their own dependencies and run in isolated environments — sidestepping the "dependency hell" of mixing distro packages with third-party software.

- **Flatpak**: Cross-distro sandboxed apps, focused on desktop applications. Uses runtimes shared between apps. Not useful for servers because it is based on the GUI and does not integrate with systemd
- **Snap**: Canonical's universal package format, used heavily in Ubuntu. Self-contained and auto-updating.
- **AppImage**: A single-file application that runs without installation. Simplest of the three but with weaker sandboxing.

Basic Configurations of Common Services

Understanding the foundational roles of standard services:

- **DNS (Domain Name System)**: Name resolution — translating hostnames to IP addresses. BIND is the classic implementation.

    - **`/etc/resolv.conf`** (DNS Resolver Configuration) This is the primary DNS client configuration file. It tells the system which DNS servers to query. Configuration file used by the system's resolver library to determine how to resolve hostnames into IP addresses via the Domain Name System (DNS) (Now replaced by **systemd-resolved**) Sometimes, changes to `/etc/resolv.conf` are overwritten by NetworkManager or DHCP upon reboot. Know that you might need to use `chattr +i` to make the file **immutable** if you want to force manual settings, or better yet, configure the DNS via `nmcli` or **Netplan**
        - nameserver = specifies IP address of the DNS server. (e.g., `nameserver 8.8.8.8`). You can list multiple nameserver lines; the system will try them in order.
        - search = defines default FQDN search list. (e.g., `search example.com`) For example, if you type ping db01, it will actually ping db01.example.com
        - **`domain`**: Similar to `search`, but used for a single local domain name.
        - **`options`**: Used to modify the behavior of the resolver, such as setting timeouts or the number of retries.
    - **`/etc/hosts`** (Local Static Lookup) A static local lookup table. Entries here are checked **before** DNS by default. provides a way to resolve hostnames to IP addresses WITHOUT a DNS server
        - IP_address FQDN alias
    - **`/etc/nsswitch.conf`** (Name Service Switch) This file controls the **order** of name resolution. This is critical for the exam.
        - hosts: files dns myhostname
            - This line means: check `/etc/hosts` first (`files`), then query DNS (`dns`), then use the local hostname (`myhostname`). You can change the order here, which is how you'd make DNS resolve before local files, or add other sources like LDAP.
    - `systemd-resolved` is the modern DNS resolver on most distros (Ubuntu, Fedora, RHEL 8+).
        - **Key command: `resolvectl`**
- **NTP / PTP**: Network time synchronization. NTP (Network Time Protocol) is general-purpose; PTP (Precision Time Protocol) provides sub-microsecond accuracy for finance, telecom, and scientific use.

- **`timedatectl` =** Modern Linux distributions utilize `timedatectl` (part of the `systemd` ecosystem) as a centralized frontend tool to check and alter the system clock, time zone, and NTP status.
  - **`status`** – (Default) Displays current local time, universal time (UTC), RTC (Real-Time Clock), time zone, and whether NTP synchronization is active.
  - **`set-ntp [true|false]`** – Enables or disables network time synchronization. Turning this on activates the underlying NTP client (like `systemd-timesyncd` or `chronyd`).
  - **`set-time [YYYY-MM-DD HH:MM:SS]`** – Manually adjusts the system clock. *Note: This will fail if NTP synchronization is currently enabled.*
  - **`set-timezone [Zone/Subzone]`** – Changes the system's active time zone (e.g., `America/New_York`).
  - **`list-timezones`** – Lists all available time zones valid for the system configuration.
- **`systemd-timesyncd` =** For simple desktop or cloud instances that only need to sync their local clock with a remote server, systemd provides a built-in, lightweight SNTP (Simple NTP) daemon called `systemd-timesyncd`.
  - **Configuration File:** `/etc/systemd/timesyncd.conf`
  - **Key Parameters:** Inside this file, you define your primary servers using the `NTP=` directive and fallback options using `FallbackNTP=`.
  - **Exam Tip:** `systemd-timesyncd` only works as a *client*. It cannot serve time to other machines on a network. If a question asks about configuring an NTP server for a local network, look toward Chrony or the legacy NTP daemon.
- **`chrony`** = the default NTP implementation on modern enterprise distributions (such as RHEL, Rocky Linux, and Ubuntu Enterprise). It excels at syncing time quickly, handling intermittent network connections, and operating as a local time server.
  - **`chronyd`:** The background daemon that handles background clock adjustments.
  - **`chronyc`:** The command-line interface tool used to monitor and manage `chronyd`.
    - **`sources`** – Lists the current NTP servers `chronyd` is connected to, showing their stratum levels, reachability, and clock offsets.

        - **`sourcestats`** – Displays detailed statistical information about clock drift and error rates for each source.
        - **`tracking`** – Shows parameters of the system clock performance, such as how much the system clock is lagging or leading compared to the reference IDs.
        - **`makestep`** – Forces the system clock to instantly "step" (jump) to the correct time if the clock offset is large, rather than slowly adjusting (slewing) it over time.
    - **`/etc/chrony.conf`** (or `/etc/chrony/chrony.conf` depending on the distribution).
        - Within this file, you specify target upstream servers using the `pool` or `server` directive (e.g., `pool pool.ntp.org iburst`). The `iburst` option instructs the daemon to make a burst of initial queries to speed up the first synchronization.
- **DHCP (Dynamic Host Configuration Protocol)**: Automatic IP address assignment to clients on a network. Operates on port 67 on the server to listen to incoming client requests and port 68 on the client to listen for replies and configs from the server.
    - **`/etc/dhcp/dhcpd.conf`** Inside this file, you must recognize basic block directives like:
        - `subnet 192.168.1.0 netmask 255.255.255.0 { ... }` – Defines the boundary of the local scope.
        - `range 192.168.1.100 192.168.1.200;` – Specifies the dynamic pool of IP addresses available to hand out to incoming clients.
        - `option routers 192.168.1.1;` – Hands clients their default gateway mapping.
        - `option domain-name-servers 8.8.8.8;` – Distributes upstream DNS targets directly to clients upon a successful lease acknowledgement.
- **HTTP**: Web servers — Apache HTTP Server (httpd) is the long-running standard; Nginx is the modern lightweight alternative often used as a reverse proxy and load balancer. Operates on port 443 for https and port 80 for http
    - **httpd** = apache web server. Apache assigns a dedicated operating system thread or process to every single incoming HTTP connection. Apache can process php content natively which means it is excellent at hosting dynamic content. The Catch: While excellent for stability and executing embedded code (like PHP), it eats up significant RAM under heavy, concurrent traffic because creating and switching between hundreds of

OS processes consumes heavy system overhead. apachectl is the command that acts as the interface for the httpd daemon
    - Primary Configuration Paths:
        - `/etc/httpd/conf/httpd.conf` (RHEL/Rocky)
        - `/etc/apache2/apache2.conf` (Debian/Ubuntu).
    - **nginx** = Nginx was built specifically to solve the "C10K problem" (handling 10,000 concurrent connections smoothly) that plagues apache. It utilizes an asynchronous, non-blocking, event-driven loop. A small, fixed number of worker processes run continuously, and each worker handles thousands of connections simultaneously within a single thread. The catch: you cannot process dynamic content natively; must pass requests to an external gateway (like `php-fpm`).
        - Primary Configuration Path: `/etc/nginx/nginx.conf`
- **SMTP (Simple Mail Transfer Protocol)**: Sending email between servers. Postfix and Sendmail are common Linux implementations. Operating on port 25 for server to server interactions (relaying) and port 587 and 465 for client-server interactions (submission).
- **IMAP4 (Internet Message Access Protocol)**: Retrieving email from a server while leaving copies on the server (versus POP3, which downloads and deletes). Operates on port 143 in unencrypted form and port 993 in encrypted form

---

2.5 systemd

systemd is the modern init system used by nearly all major Linux distributions. It replaced the older SysV init (`/etc/init.d` scripts and runlevels). systemd does much more than start services — it manages timers, mounts, network configuration, logging, and more, all through a unified "unit" concept.

Systemd Units

The fundamental objects managed by systemd. Every managed thing is a unit, with a specific type indicated by the file extension:

- Services (`.service`) = Manage standard background daemons and applications.
    - Use standard state triggers (`start`, `stop`, `restart`, `reload`, `status`, `enable`, `disable`). Example: `systemctl restart sshd.service`

- Timers (`.timer`) = Replace legacy cron jobs by triggering events based on time or calendar schedules.
    - You start, stop, or enable timers just like services. To see all active timers on the system, use the dedicated command: `systemctl list-timers`.
    - Example: `systemctl enable backup.timer`
- Mounts (`.mount`) = Control filesystem mount points via systemd
    - The Critical Catch: The name of the unit file must exactly match the physical path of the mount directory, replacing slashes with hyphens. If you want to manage the mount point `/mnt/data/backup`, the unit file *must* be named `mnt-data-backup.mount`.
    - How to interact: *
        - `systemctl start mnt-data.mount` (Mounts the filesystem instantly)
        - `systemctl stop mnt-data.mount` (Unmounts the filesystem)
- Targets (`.target`) = Group other units together to define operating system states (replacing traditional SysV runlevels).
    - multi-user.target: Roughly equivalent to old runlevel 3 — multi-user, networked, no GUI.
    - graphical.target: Roughly equivalent to old runlevel 5 — multi-user with a graphical login.
    - rescue.target and emergency.target: Single-user modes for recovery.
    - How to interact: Instead of using `start`, you use the `isolate` subcommand to switch the entire operating system into a specific target mode.
    - Example: `systemctl isolate multi-user.target` (Switches the server to non-graphical CLI mode).
- Sockets (`.socket`) = Intercept network traffic or IPC channels. Sockets facilitate "socket activation," meaning systemd will listen on a port (e.g., port 22) and keep the associated service (SSH) completely turned off until a packet arrives, saving system memory.
    - How to interact: You manage them like services, but stopping a service while its socket is still running means incoming traffic will just wake the service right back up.
    - Example: `systemctl stop sshd.socket` (Stops systemd from listening on the network port entirely).
- Devices (`.device`) = Expose hardware devices (tracked by the kernel and `udev`) as systemd units.

    - How to interact: You cannot start, stop, or edit a device unit because hardware is controlled by physical presence and drivers. However, you use them to check hardware status or create dependencies (e.g., telling a service not to start until a specific network card device is active).
    - Example: `systemctl status dev-sda1.device`
- Paths (`.path`) = Monitor specific files or directories for real-time changes (like file modification, creation, or deletion) and trigger a corresponding service when a change occurs.
    - How to interact: * `systemctl start watch-config.path`
    - Editing context: When editing a `.path` unit, you define tracking parameters like `PathModified=/etc/nginx/nginx.conf` to declare exactly what target path to monitor.
- Slices (`.slice`) = Group processes together into resource-management buckets using Linux kernel Control Groups (`cgroups`). This allows you to restrict CPU, memory, and I/O utilization for a whole group of applications at once.
    - You monitor them using `status` to view resource consumption hierarchies.
    - Example: `systemctl status user.slice` (Shows all resources used by active logged-in users).
    - Editing context: To restrict resource allocations, you run `systemctl edit system.slice` and append resource boundaries under the `[Slice]` configuration block:

Journal vs. Traditional Logs

systemd uses a binary journal (queried with `journalctl`) for structured, indexed logs. Traditional text logs in `/var/log` are still produced by rsyslog/syslog-ng on most systems, often by reading from the journal.

2.6 Container Management

What Containers Are

Containers are isolated user-space environments that share the host kernel. Compared to VMs, they're much lighter (no separate kernel, no emulated hardware), start in seconds, and pack many more workloads per host. They achieve isolation through Linux kernel features: **namespaces** (separate views of processes, network, mounts, etc.) and **cgroups** (resource limits).

Runtimes

The software responsible for actually running containers:

- **runC**: A low-level OCI-compliant runtime. The piece that actually creates and runs containers.
- **containerd**: An industry-standard higher-level runtime that uses runC under the hood. Used by Docker and Kubernetes.
- **Docker**: A highly popular developer tool and platform. Uses containerd internally. Includes image building, registry, and CLI tooling.
- **Podman**: A daemonless, more secure Docker-compatible alternative. Can run containers as non-root ("rootless mode") and integrates well with systemd. Default on RHEL.

Image Operations

Container *images* are the templates used to create *containers*.

- **Pulling**: Downloading an image from a registry (Docker Hub, Quay, a private registry).
- **Pruning**: Deleting unused images to reclaim disk space. Docker/Podman does NOT do this automatically so this is absolutely necessary.
- **Tags**: Versioned labels (e.g., `nginx:1.25`, `nginx:latest`). Without a tag, `latest` is implied.
- **Layers**: Images are built as cached, read-only layers stacked on top of each other. Sharing layers between images saves enormous amounts of space.

Dockerfile Instructions

A Dockerfile is the build recipe for an image. Key instructions:

- **FROM**: Specifies the base image to start from.
- **ENTRYPOINT**: Defines the executable to run when the container starts.
- **CMD**: Provides default arguments to ENTRYPOINT (or a default command if no ENTRYPOINT is set). Can be overridden at run time.
- **USER**: Dictates the privilege level (which user the container runs as). Best practice: do not run as root inside the container.

Container Operations

Day-to-day container management includes: reading logs, mapping volumes for persistent storage, starting and stopping containers, inspecting metadata, deleting

containers, pruning dead containers, managing tags, and passing environmental variables in at runtime.

Volume Operations

Containers are ephemeral — when one is deleted, anything written inside its filesystem is gone. Volumes provide persistent storage:

- **Creating volumes**: Named volumes managed by the runtime.
- **Mapping volumes**: Bind-mounting host paths into the container.
- **Pruning**: Deleting orphaned volumes.
- **SELinux context (z/Z flags)**: On SELinux-enforcing hosts, special flags relabel volume content so the container can access it.
- **Overlay filesystems**: The layered, copy-on-write filesystem mechanism that makes container images efficient.

Container Networks

- **Bridge** (default): An isolated virtual network. Containers can talk to each other and reach the outside through NAT.
- **Host**: Shares the host's network stack directly. Fastest but loses isolation.
- **macvlan / ipvlan**: Give the container its own MAC or IP on the physical network, as if it were a separate device.
- **Overlay**: Connect containers across multiple physical hosts — used in swarm/cluster setups.
- **None**: Disables networking entirely. Useful for sensitive batch jobs.
- **Port mapping**: Exposes a container port on the host's interface, allowing external access.

Security: Privileged vs. Unprivileged

- **Privileged containers** have near-root access to the host kernel and devices. They can damage the host. Avoid except for very specific use cases.
- **Unprivileged (rootless) containers** run as a regular user with restricted capabilities. Much safer, and the default for Podman.

Domain 3.0: Security

3.1 AAA, Logging, and System Audit

AAA Methods

AAA stands for Authentication, Authorization, and Accounting — the trio of "who are you, what are you allowed to do, and what did you do?"

- **Polkit (PolicyKit)**: Defines fine-grained privileges for system-wide tasks initiated by non-root users (e.g., letting a desktop user mount a USB drive without sudo). While with sudo, you give a user permission to run the entire `systemctl` binary as root. It's all-or-nothing. With polkit, when a non-root user clicks "Restart" on a desktop menu. Polkit steps in and safely permits *only* that specific reboot action to execute, without giving the user a root shell or full access to other system services. Polkit is most heavily used in the GNOME gui.
- **PAM (Pluggable Authentication Modules)**: A flexible framework that handles authentication rules. Different services (login, SSH, sudo, screen savers) plug into PAM and inherit consistent password policies, account locking, and MFA support.
- **SSSD (System Security Services Daemon) / Winbind**: Both allow you to connect to a centralized authentication server like Active Directory
    - **SSSD** is the modern standard for Linux enterprise environments (RHEL, Rocky, Ubuntu). It was built from the ground up by Red Hat to be a lean, high-performance identity proxy. It talks natively to modern directory protocols like LDAP (to pull user info) and Kerberos (to check passwords). It can connect Linux to centralized identity systems like Active Directory or LDAP. SSSD caches credentials so users can still log in if the directory server is briefly unreachable. Linux calls the central authentication server a Realm. SSD is incredibly resilient. If your corporate network cuts out or a router fails, SSSD allows domain users to keep logging into the Linux server using an encrypted local cache of their credentials. Best Used For: Standard system logins, SSH servers, web servers, and DevOps infrastructure where users just need a secure command-line or GUI login.
    - **Winbind** is a specialized tool that is baked directly into the Samba software suite (`winbindd`). It is a legacy champion that still holds a massive footprint in enterprise networks. Instead of using standard LDAP/Kerberos protocols, Winbind relies on Samba's deep integration with Microsoft's native SMB/CIFS network protocols. It acts as a direct translator, mapping complex Windows Security Identifiers (SIDs) into standard Linux User IDs (UIDs). Because it lives inside Samba, Winbind understands the complex, nested Windows permission models perfectly. Best Used For: Linux storage servers, Network Attached Storage (NAS) boxes, or print servers that actively host file shares for hundreds of physical Windows desktop clients.

- **LDAP (Lightweight Directory Access Protocol)**: A protocol for accessing and maintaining distributed directory information. The backbone of most centralized identity systems. Operates on port 389 by default and 636 for LDAPS. The main config file for LDAP is located at /etc/openldap/ldap.conf
    - **`uid`** (User ID): The actual username (e.g., `uid=jdoe`).
    - **`cn`** (Common Name): The full name of the object (e.g., `cn=John Doe`).
    - **`ou`** (Organizational Unit): The folder/department the user belongs to (e.g., `ou=Accounting`).
    - **`dc`** (Domain Component): The parts of the network domain name broken down (e.g., if the domain is `company.com`, it becomes `dc=company,dc=com`).
- **Kerberos**: A network authentication protocol using tickets to prove identity without sending passwords over the wire. The core of Active Directory authentication. Operates as an SSO technology using tickets instead of passwords. Operates on port 88.
- **Samba**: Facilitates Windows file sharing and can also act as an Active Directory domain controller.
- **SSO (Single Sign-On)**: Authenticate once, access many services. Often built on Kerberos, SAML, or OAuth/OIDC.
- **TOTP (Time-based One-Time Password)**: The codes generated by authenticator apps (Google Authenticator, Authy). A common second factor in MFA.

Logging

- **/var/log**: The traditional directory for the actual system logs. Common subdirectories include `messages`, `secure`, `auth.log`, `syslog`, `kern.log`, and per-service directories. This is where log files will be stored
- **/etc/logrotate.conf** for system-wide log instructions, and **/etc/logrotate.d** for application log instructions
    - `rotate <number>`: The number of old log files to keep before deleting the oldest one. For example, `rotate 4` means it will keep `log.1`, `log.2`, `log.3`, and `log.4`. When a new rotation happens, `log.4` is deleted.
    - `daily` / `weekly` / `monthly`: Specifies the time interval trigger for log rotation.
    - `size <bytes/k/M/G>`: Triggers a rotation only when the log file reaches a specific size (e.g., `size 100M`), ignoring the time interval completely.

    - `compress`: Compresses old rotated log files using `gzip` (resulting in `.log.1.gz`) to save disk space.
    - `delaycompress`: Tells logrotate *not* to compress the most recent archive until the *next* rotation cycle. This is incredibly helpful if software still needs to write residual data to the previous log file for a short bit.
    - `missingok`: If the log file doesn't exist (e.g., a service was uninstalled), logrotate will quietly move on without throwing an error or halting execution.
    - `create <mode> <owner> <group>`: Immediately creates a brand-new, completely blank log file right after moving the old one out of the way, applying the specific permissions (e.g., `create 0640 root adm`).
    - `postrotate` / `endscript`: A block that lets you run custom bash commands *after* the logs have been rotated.
- **rsyslog / syslog-ng**: Daemons that collect, filter, and forward log messages. Can ship logs to centralized log servers for aggregation.
- **journalctl / systemd-journald**: The systemd native binary log system. Supports indexed queries and structured fields.
- **logrotate**: A scheduled tool that rotates, compresses, and deletes old logs to prevent log files from filling up `/var`.

System Audit

- **auditd**: A highly secure audit daemon that records security-relevant kernel events — file accesses, syscalls, login attempts. Independent of regular syslog.
    - **`/etc/audit/rules.d/`**: The directory where administrators write and edit custom `.rules` configuration templates (e.g., `audit.rules`). Changes made here are permanent.
    - **`/etc/audit/audit.rules`**: The master file that the system reads at boot time. **Never edit this directly**; it is automatically overwritten and generated from the `rules.d/` directory.
    - **`/var/log/audit/audit.log`**: The dedicated, highly secure destination where all raw audit records are written.
    - **`/etc/audit/auditd.conf`**: The configuration file that dictates how the daemon itself behaves (e.g., how big logs can grow, disk space alerts, and log rotation policies).
- **audit.rules**: The configuration file dictating which events the audit daemon should monitor (e.g., "log every write to `/etc/passwd`"). Audit rules are managed in /etc/audit/rules.d/ and you must apply them using the augenrules

--load command. You can also run the auditctl command to temporarily run the audit, but this will get wiped after reboot.
- Used for compliance frameworks (PCI-DSS, HIPAA, SOX) that require provable trails of who did what and when.

3.2 Firewalls

Firewall Concepts

Linux firewalls all ultimately rely on the **Netfilter** kernel module, which inspects packets as they traverse the network stack.

- **Runtime vs. permanent rules**: Runtime rules apply immediately but vanish on reboot or service reload. Permanent rules are saved to disk and reapplied at boot. A common mistake is to add a runtime rule, verify it works, and forget to make it permanent.
- **Zones**: Logical groupings of network interfaces with shared trust levels (e.g., `public`, `internal`, `dmz`, `trusted`). Each zone has its own set of allowed services and ports.
- **Rich rules**: More expressive rules that can match on source addresses, log matches, rate-limit connections, and so on.
- **Ports vs. services**: Filters can target raw port numbers (e.g., TCP 443) or named services (e.g., "https"). Named services are clearer but ultimately translate to port/protocol combinations.

Firewall Front-ends

- **firewalld**: A dynamic, zone-based firewall manager. Default on RHEL/Fedora/Rocky.
- **UFW (Uncomplicated Firewall)**: A simpler front-end common on Debian distros focused on ease of use.
- Both eventually push rules down to the kernel through one of the underlying engines.

Underlying Engines

- **iptables**: The classic, long-standing rule engine. Mature and being replaced by nftables.
- **nftables**: The modern replacement for iptables, with a cleaner syntax and better performance. Most distributions are migrating to it. firewall-cmd and ufw both use

nftables now. `nftables` replaces `iptables` because it unifies IPv4, IPv6, ARP, and bridging into a single, faster architecture.
- **ipset**: A companion to iptables/nftables that allows huge sets of IPs, networks, or ports to be matched efficiently in a single rule. Essential for large blocklists.

Stateful vs. Stateless

- **Stateful firewalls** track active connections — once a connection is allowed in one direction, the return traffic is automatically permitted. Almost all modern firewalls are stateful.
- **Stateless firewalls** evaluate every packet independently, with no memory of previous packets. Simpler and faster but require explicit rules for both directions.

Address Translation

Modifying packet IP addresses as they pass through:

- **NAT (Network Address Translation)**: The general concept of rewriting addresses.
- **PAT (Port Address Translation)**: Maps multiple private IPs to a single public IP using ports to distinguish them. This is what most home routers do, sometimes called "NAT overload" or "masquerading."
- **DNAT (Destination NAT)**: occurs when the firewall alters the **Destination IP address** of a packet. This is used for **inbound traffic**—when someone on the public internet needs to reach a private server hidden behind your firewall.
- **SNAT (Source NAT)**: occurs when the firewall alters the **Source IP address** of a packet. This is almost exclusively used for **outbound traffic**—when internal, private computers need to go out to the public internet.

IP Forwarding

Enabling **IP forwarding** via the kernel parameter `net.ipv4.ip_forward` allows a Linux server to act as a router, passing traffic between interfaces. Required for any NAT/firewall gateway role.

3.3 OS Hardening

Privilege Escalation Configurations

- **/etc/sudoers**: Defines who can run elevated commands and with what restrictions. Should only be edited with `visudo`, which validates syntax before

saving. (WARNING: ONLY USE VISUDO FOR CREATING FILES HERE).
Syntax: TargetEntity HostList = (UserList : GroupList) TagList: CommandList

- **`TargetEntity`**: Who gets the power? An individual user account: `username` A system security group: `%groupname` (The `%` prefix is **mandatory** for groups).
- **`HostList`**: On which machines can they run this? Almost always set to `ALL` (applies to local or remote network connections hitting this host).
- **`UserList`**: Whom can they impersonate? The username they run the command as (usually `root` or `ALL`).
- **`GroupList`**: Which primary group can they impersonate? Optional. Allows running commands with specific group permissions (`:ALL`).
- **`TagList`**: Special modifiers altering execution behavior. Most common:
  - `NOPASSWD:` Allows running sudo without entering a password. Convenient but risky — a compromised user account becomes a compromised root account.
  - `NOEXEC:` an advanced defensive tag. It prevents a program executed via `sudo` from spawning any sub-processes, child applications, or interactive command shells.
- **`CommandList`**: The exact tools they can execute.

- **Must use the absolute path to the binary** (e.g., `/usr/bin/dnf`, not just `dnf`). Separate multiple commands with commas.
- **/etc/sudoers.d**: A directory for modular sudo configurations. Drop-in files here are read automatically — cleaner than one giant sudoers file. This is recommended for modern best practice (WARNING: ONLY USE VISUDO FOR CREATING FILES HERE)

wheel vs. sudo Groups

When delegating root access, distributions handle the administrator group differently:

- **Red Hat / Fedora-based**: The **wheel** group.
- **Debian / Ubuntu-based**: The **sudo** group.

The group is just a convention — the actual privilege comes from a `/etc/sudoers` rule granting that group full access.

File Attributes and Permissions

- **Special attributes** (set via tools like `chattr`):

    - **immutable**: Prevents *anyone*, including root, from modifying or deleting the file. Root must remove the attribute first.
    - **append-only**: Allows adding data but not deleting or modifying existing content. Useful for log files.
- **umask (default user file-creation mode mask)**: A bitmask subtracted from default permissions when creating new files. Determines what permissions new files and directories start with.
- **ACLs (Access Control Lists)**: Provide granular per-user and per-group permissions beyond the standard owner/group/other model. Allows things like "this specific user can read this file" without altering the file's primary group.

Special Permissions

- **Sticky bit**: When set on a directory, prevents users from deleting files they don't own — even if the directory itself is world-writable. Classic example: `/tmp`.
- **setuid (SUID)**: When set on an executable, the program runs as the *file owner* (typically root) regardless of who launched it. The classic example is `passwd`, which needs to write to `/etc/shadow`. SUID binaries are a major attack target — minimize them.
- **setgid (SGID)**: On an executable, runs as the file's group. On a directory, forces new files inside to inherit the directory's group ownership — useful for shared workspaces.

SELinux States

Security-Enhanced Linux (SELinux) is a mandatory access control (MAC) system that enforces policies beyond standard Unix permissions. SELinux operates based on a massive, pre-defined set of rules called the **Policy**. Think of it as a giant spreadsheet that lists every possible relationship between a "Subject" (a process) and an "Object" (a file, a port, or another process). When a process wants to do something—like read a file—the kernel interrupts the request and asks the SELinux engine: *"Does the policy allow [Process Label] to perform [Action] on [Object Label]?"*

- **If there is a rule:** The action is allowed.
- **If there is NO rule:** The action is **denied by default.**

Operates in one of three states:

- **Enforcing**: Blocks and logs policy violations.
- **Permissive**: Logs violations but does *not* block them. Used for testing new policies.
- **Disabled**: Turned off entirely.

SELinux contexts label every file, process, and port. Policy decides which contexts can interact with which. Many "weird permission" issues on RHEL-based systems trace back to SELinux context mismatches.

Secure Remote Access

The sshd service (SSH Daemon) is the server-side component of the SSH suite. While the `ssh` command is what you use to connect to a remote server, `sshd` is the program that runs on that remote server, **listening** for incoming connections, authenticating users, and managing encrypted sessions. Usually, you never use the bare sshd command, but instead edit the SSH config file in /etc/ssh/sshd_config. Hardening the SSH daemon (sshd) is essential since SSH is usually the only network entry point:

- **Key vs. password authentication**: Keys are far stronger than passwords. Password authentication should ideally be disabled.
- **SSH tunneling**: Forwarding ports through an SSH connection to securely access internal services.
- **PermitRootLogin**: Should be disabled (or set to `prohibit-password`, which requires a SSH key instead. This is the default setting) to prevent direct root login. This parameter is edited in the /etc/ssh/sshd_config file.
- **Disabling X forwarding**: X11 forwarding allows a user to use GUI features through SSH. It is a large security hole. Disabling it reduces the attack surface if you don't need to run GUI apps over SSH. In order to turn it off, you have to set X11Forwarding to “no” inside the /etc/ssh/sshd_config file.
- **AllowUsers / AllowGroups**: Restricts SSH access to specific users or groups. AllowUsers gives SSH access to specific users and AllowGroups gives SSH access to users in specific groups. You specify these permissions inside the /etc/ssh/sshd_config file.
- **SSH agent**: Holds decrypted private keys in memory so you don't have to retype the passphrase for every connection.
- **SFTP (Secure File Transfer Protocol)**: Runs over SSH. Should be used in place of raw FTP, which sends credentials in cleartext.
    - You want to make sure SFTP settings inside of /etc/ssh/sshd_config are:
        - Match Group sftp-users
            - ChrootDirectory /home/%u
            - ForceCommand internal-sftp
            - X11Forwarding no
            - AllowTcpForwarding no
- **fail2ban**: A daemon that watches log files for repeated failed authentication attempts and temporarily bans the source IP in the firewall. Effective against brute-force attacks.

    - These are the parameters to know in the fail2ban config file, kept in /etc/fail2ban/jail.conf. Remember, you should not edit this file, but instead create /etc/fail2ban/jail.local and edit the parameters in there. Ensure you have [Default] as the header on the top
        - `[Default]`
        - `maxretry = 5` (5 failed login attempts)
        - `findtime = 10m` (Within a 10-minute window)
        - `bantime = 10m` (Results in a 10-minute ban)
        - [sshd]
        - enabled = true

OS Hardening Practices

- **Avoid insecure unencrypted services**: Telnet, FTP, and TFTP all transmit credentials in cleartext. You should disable them with systemctl and purge/remove them with apt or dnf respectively.
- **Disable unused filesystems**: Filesystem drivers that aren't needed (e.g., legacy formats) should be blocked from loading to reduce attack surface. You should add .conf files to /etc/modprobe.d/ to disable a filesystem. In order to disable it, you will need to create a .conf file and name it whatever you want. You will add these two parameters to it
    - **`blacklist <filesystem>`** * *What it does:* Tells the system's hardware discovery tools not to automatically load this driver when a device is plugged in.
    - **`install <filesystem> /bin/true`** * *What it does:* The absolute workaround. It tells the kernel, *"If anything tries to force-load this filesystem, run the dummy program `/bin/true` instead of the actual driver."* Because `/bin/true` does nothing and exits successfully, the loading process safely terminates without activating the filesystem.
- **Remove unnecessary SUID permissions**: Every SUID binary is a potential privilege escalation. Audit and remove unused ones by using chmod u-s or chmod u-s to remove the SUID execution bit.
- **Secure boot / UEFI**: Verifies the boot chain cryptographically to prevent malicious bootloaders or kernels.

3.4 Account Hardening

Password Policies: For system wide changes, look to /etc/login.defs (aging and expiration) and /etc/security/pwquality.conf (complexity & length) and `/etc/pam.d/system-auth` or `/etc/pam.d/common-password` (History and Reuse) files.

- **Complexity**: Mixing character classes (upper, lower, digits, symbols).
    - Inside the /etc/security/pwquality.conf file, you need to know these values:
        - `ucredit = -1`: Requires at least one Uppercase letter (negative numbers mean *at least*, positive numbers give credits).
        - `lcredit = -1`: Requires at least one Lowercase letter.
        - `dcredit = -1`: Requires at least one Digit (number).
        - `ocredit = -1`: Requires at least one Other character (special symbol like `@`, `!`).
- **Length**: The single biggest factor in password strength.
    - Inside the /etc/security/pwquality.conf file, you need to know this value:
        - `minlen = 12`: Sets the minimum length of the password to 12 characters
- **Expiration**: Forcing periodic changes. Modern guidance (NIST) actually discourages forced rotation for users with long passwords and MFA — it tends to drive weaker, more predictable passwords. Usually edited with the chage command or the /etc/login.defs file. Inside the login.defs file, you will need to know these fields:
    - PASS_MAX_DAYS = max # of days a password is valid
    - PASS_MIN_DAYS = min # of days a password is valid
    - PASS_WARN_AGE = Sets # of days before expiration that a system will start throwing warnings at the user during logon
- **Reuse**: Preventing users from cycling through a small set of old passwords. Edited either in `/etc/pam.d/system-auth` on RHEL or `/etc/pam.d/common-password` on Deb. On modern enterprise systems, you are **not** supposed to edit `/etc/pam.d/system-auth` manually. If you do, the system's management engine will overwrite your changes during the next update. Instead, you use a tool called **`authselect`** to turn on password history features, which automatically writes the lines to the file for you.
    - password required pam_pwhistory.so remember=5 use_authtok = sets the remember parameter to the last 5 passwords and rejects password reuse
- **History**: Tracking previously used passwords so they can't be immediately reused. `/etc/pam.d/common-password` on Deb and /etc/security/pwhistory.conf on RHEL.

Multi-Factor Authentication (MFA)

Combines something you know (password), something you have (token, phone), and/or something you are (biometric). TOTP apps and hardware tokens (YubiKey) are common second factors. The only module you need to know for the Linux+ is pam_google_authenticator.

- **The User Command:** The user runs the terminal command **`google-authenticator`**. This generates a massive QR code on their screen, which they scan with their phone app. It also generates emergency backup scratch codes.
- **The PAM Switch:** The admin adds the module to the authentication stack (like `/etc/pam.d/sshd` for remote logins): auth required pam_google_authenticator.so
-

Account Hardening Practices

- **Checking existing breach lists**: Comparing user passwords against public breach databases (e.g., HaveIBeenPwned) to block reuse of known-compromised passwords. To enforce password validation on Linux, the standard Pluggable Authentication Module is **`pam_pwquality`** (which replaced the older `pam_cracklib`). This module intercepts password changes and tests them against complexity rules and database lists. To make `pam_pwquality` cross-reference user passwords against millions of known leaked breach passwords, you must feed it a dictionary list. The primary configuration file where you manage this is:
  - `/etc/security/pwquality.conf`
    - # /etc/security/pwquality.conf
    - badwords = /usr/share/wordlists/rockyou.txt
    - dictpath = /usr/share/dict/cracklib-format-breached-db
  - Administrators populate the system with a known breach or common password list (such as the famous *RockYou.txt* list or a curated list from a threat intelligence database). These lists are generally saved under `/usr/share/dict/` or `/usr/share/wordlists/`.
- **Restricted shells**: Used to prevent interactive logins for service accounts.
  - **/sbin/nologin**: Politely tells users that the account is unavailable for interactive login to the OS. It is used primarily for system, daemon, or service accounts (like `apache`, `nginx`, `mysql`, or `ftp`) that need to own

files and run background processes, but should never be allowed to open an interactive terminal session.0*
    - **/bin/rbash**: A "restricted bash" that prevents directory changes, modifying PATH, and other escape routes.
- **pam_tally2** (and its modern successor `pam_faillock`): A PAM module that tracks failed login attempts and locks accounts after a threshold, defending against brute-force attempts. When a user attempts to authenticate, `pam_tally2` increments a failure counter stored in a binary ledger file (historically located at `/var/log/tallylog`). To enforce account lockout policy rules, administrators include `pam_tally2.so` within identity validation files inside `/etc/pam.d/` (such as `system-auth`, `password-auth`, or `common-auth`). A standard operational implementation line looks like this:=
    - auth required pam_tally2.so deny=5 unlock_time=1200 onerr=fail
    - **`deny=<N>`**: Dictates the exact threshold of maximum permitted failed logins before account access is closed off (e.g., `deny=5` triggers a lockout on the 5th bad attempt).
    - **`unlock_time=<seconds>`**: Defines how long the account will remain automatically locked out before a user can try logging in again (e.g., `1200` seconds = 20 minutes). If this is *omitted*, the account remains permanently locked until an administrator intervenes manually.
    - **`onerr=fail`**: Specifies a defensive fallback policy. If the module encounters an unreadable log file or internal structural failure, it defaults to failing the authentication attempt.
    - **`magic_root`**: If invoked, this skips failure tracking if the attempt is initiated by the `root` account via standard shell escalations, ensuring administrative safety rails.
    - **`even_deny_root`**: Forces the lockout rules to apply to the `root` user account as well. *(Exam Warning: Using this without a safety net can lock admins completely out of the system during a heavy brute-force attack!)*
- **pam_faillock** = replaced pam_tally2. Unlike legacy configurations, the modern `pam_faillock.so` framework stores its system-wide variables (such as lockout thresholds and durations) in **`/etc/security/faillock.conf`**.
    - **Key Configuration Parameters inside `faillock.conf`:**
        - `deny = 5` — Triggers an automatic account lockout after 5 consecutive failed attempts.
        - `unlock_time = 900` — Specifies the duration of the lockout in seconds (900 seconds = 15 minutes) before the system automatically opens the account back up.

        - `even_deny_root` — Enforces the lockout policy rules on the root user account as well, adding a layer of defense against direct root brute-forcing.
    - Failures are logged as individual binary state tracking files per user within the volatile runtime directory `/var/run/faillock/` (or `/run/faillock/`). This means that a complete physical system reboot will inherently clear all active lockout states.
- **Avoid running daily tasks as root**: Use a regular user account for normal work and elevate only when needed. Reduces blast radius if the account is compromised.

3.5 Cryptography

Data at Rest

Protecting stored data:

- **File encryption**: Asymmetric encryption for individual files. **GPG** is the standard tool.
    - **Your Public Key:** Think of this as an open padlock. You can hand it out to anyone in the world, post it on the internet, or send it to your coworkers.
    - **Your Private Key:** This is the physical key that opens the padlock. You must guard this with your life and **never** share it with anyone.
- **Filesystem encryption**: Encrypting entire block devices. Use cryptsetup command to interact with the Argon2 and LUKS2 encryption algorithms.
    - **LUKS2 (Linux Unified Key Setup, version 2)**: This is the Linux equivalent to Bitlocker on Windows. It is the standard for full-disk encryption on Linux. Stores key slots in a header at the start of the partition.
        - /etc/crypttab = crypto table. Used for managing boot-time encrypted drives. This file handles **decryption**. It converts raw encrypted devices into usable `/dev/mapper/` devices at boot.
            - tpm2-device=auto = used after running systemd-cryptenroll in order to get the encrypted drive to auto-decrypt at boot time.
        - **`/etc/fstab`**: This file handles **mounting**. It takes the decrypted device and pins it to a folder (like `/mnt/storage`) so the user can see it.

    - **Argon2**: A modern, memory-hard key derivation function used by LUKS2. Resistant to GPU and ASIC attacks compared to older KDFs. Argon2 is the security guard that protects the master key of the filesystem.
        - **Argon2d** (Data-dependent) = For cryptocurrency, data storage, backend disk encryption. Maximizes resistance against GPU cracking rigs by filling memory in a randomized order. However, it is vulnerable to *side-channel timing attacks* (hackers measuring how long memory lookups take).
        - **Argon2i** (Independent) = Fills memory in a fixed, predictable pattern that is completely immune to side-channel timing attacks, but slightly less resistant to extreme GPU brute-forcing. Password hashing servers and authentication frontends.
        - **Argon2id** (Hybrid) = **The Industry Standard.** Default for LUKS2, modern OS authentication, and password managers. Combines both i and d.

Data in Transit

Protecting data moving across networks:

- **OpenSSL**: software toolkit for encrypting data in transit, specifically **SSL/TLS certificates** (the tech that turns `http://` into secure `https://`).It can use any hashing or encryption algorithm you want to use.
    - `.key` **(The Private Key):** This is the cryptographic foundation. Just like your GPG private key, this file must stay hidden on your server with strict permissions (`600`). If a hacker steals your private key, they can impersonate your website and steal user data.
    - `.csr` **(Certificate Signing Request):** This is an application form. It contains your public key and your identity details (your company name, domain name, country, etc.). You bundle this up and send it to a trusted third party (like DigiCert or Let's Encrypt) to prove who you are.
    - `.crt` **or** `.pem` **(The Certificate):** This is the final digital ID card sent back to you by the certificate authority. You install this alongside your private key into your web server (like Apache or Nginx) so users see the secure green padlock in their browsers.
- **LibreSSL**: A fork of OpenSSL focused on code simplification and security hardening, created after the Heartbleed incident. This is a MUCH more secure version of OpenSSL.

- **WireGuard**: A modern, simple, high-performance VPN protocol. Much smaller codebase than IPsec or OpenVPN. It also runs natively in the kernel rather than user space, which means it is way more efficient that OpenVPN.
- **TLS (Transport Layer Security)**: The successor to SSL. Provides encryption and authentication for network connections (HTTPS, secure SMTP, LDAPS, etc.).
    - **Old Version:** SSL (Secure Sockets Layer): This is the old, legacy protocol invented by Netscape in the 1990s. Every single version of SSL is dead, broken, and completely insecure. * TLS (Transport Layer Security): This is the modern, highly secure successor to SSL.
    - **Modern versions** (TLS 1.2, TLS 1.3) should be enforced. Older versions (SSLv2, SSLv3, TLS 1.0, TLS 1.1) have known weaknesses and should be disabled.

Hashing

- **SHA-256**: A 256-bit cryptographic hash function from the SHA-2 family. Currently the standard for general-purpose integrity checking, although SHA-512 is rapidly replacing it.
- **HMAC (Hashed Message Authentication Code)**: Combines a hash function with a secret key to produce a value that proves both integrity *and* authenticity. Used in API authentication and JWT signing. Authenticity proves that the sender did indeed sent the message. Integrity proves that the message was not altered during transit.
- Weak algorithms to retire: **MD5** and **SHA-1** are broken for cryptographic purposes (though MD5 is still acceptable for non-security checksums like detecting accidental corruption).

Certificate Management

- **PKI (Public Key Infrastructure)**: The system of certificate authorities, certificates, and trust relationships that underpins TLS.
    - **The Certificate Authority (CA) — *The DMV.*** The CA is the ultimate source of trust. It is a trusted third-party organization (like DigiCert, GlobalSign, or Let's Encrypt) that digitally signs certificates to vouch for their validity. If the CA signs a certificate, the whole world trusts it.
    - **The Registration Authority (RA) — *The DMV Clerk.*** The RA is the helper for the CA. It handles the boring paperwork of verifying who you are. Before a CA will give you a certificate for `yourcompany.com`, the RA checks to make sure you actually own that web domain. Once the RA approves your identity, it tells the CA: *"They check out, go ahead and sign their certificate."*

    - **The Certificate — *The Driver's License.*** This is the **X.509 certificate** (`.crt` or `.pem` file) we talked about in OpenSSL. It binds an organization's identity to their public key.
    - **CRL and OCSP — *The Police Database of Suspended Licenses.*** What happens if a company's private key gets stolen by a hacker? The certificate is no longer safe, so it must be cancelled before its expiration date. PKI handles this in two ways:
        - **CRL (Certificate Revocation List):** A giant text file published by the CA listing every single certificate they have cancelled. Linux servers download this list periodically.
        - **OCSP (Online Certificate Status Protocol):** A faster, modern alternative to CRL. Instead of downloading a massive text file, your Linux server sends a quick message to the CA asking: *"Hey, is certificate #12345 still valid right now?"* The CA replies with a simple "Good," "Revoked," or "Unknown."
- **Trusted root certificates**: The set of CAs your system inherently trusts. Stored in a system-wide trust store.
    - **The Debian / Ubuntu Family.** On Ubuntu and Debian systems, the compiled bundle of trusted certificates sits in a single file, while individual certificate links live in a dedicated directory:
        - **The Active Store Directory:** `/etc/ssl/certs/`
        - **The Global Bundle File:** `/etc/ssl/certs/ca-certificates.crt`
    - **The RHEL / Fedora / CentOS Family.** On Red Hat-based systems, the layout uses the `pki` directory structure:
        - **The Active Store Directory:** `/etc/pki/tls/certs/`
        - **The Global Bundle File:** `/etc/pki/tls/certs/ca-bundle.crt`
- **Avoiding self-signed certificates**: Self-signed certs provide encryption, but don’t provide identity verification. Use a real CA (free options like Let's Encrypt remove the cost excuse) or an internal CA for internal services.
- **ACME (Automated Certificate Management Environment)**: The protocol used by Let's Encrypt and others to automate certificate issuance and renewal.

3.6 Compliance and Audit

Detection and Response

- **Anti-malware**: Tools that scan for known malicious files. ClamAV is the open-source standard on Linux.
- **IOC (Indicators of Compromise)**: Forensic artifacts that suggest a system has been compromised — unusual file hashes, suspicious IP addresses in connection logs, modified system binaries, unfamiliar cron jobs.
    - **Unusual Network Connections:** Foreign IP addresses established to unusual ports, or high volumes of outbound traffic (suggesting data exfiltration or a botnet/cryptominer daemon communicating with a Command and Control [C2] server).
    - **Unexplained Account Activity:** New, unauthorized users appearing in /etc/passwd or /etc/shadow, unexpected successful logins from root or system accounts, or logins at strange hours (visible via the last, lastlog, or w commands).
    - **Unusual Process Behavior & Resource Spikes:** A hidden or unfamiliar process consuming 100% CPU usage (highly indicative of a cryptojacker or a malicious script running in the background, viewable via top, htop, or ps).
    - **System File Modifications:** Unexpected changes to critical configuration files or system binaries. This links directly to other tools listed in this objective, such as **AIDE (Advanced Intrusion Detection Environment)** and **rkhunter**, which track file integrity hashes to catch when a malicious actor replaces an executable like /bin/ls with a compromised version.

Vulnerability Scanning

- **CVE (Common Vulnerabilities and Exposures)**: A standardized identifier for a specific vulnerability (e.g., `CVE-2021-44228` is Log4Shell). Lets everyone refer to the same flaw consistently.
- **CVSS (Common Vulnerability Scoring System)**: A numerical score (0.0–10.0) and qualitative rating for a vulnerability, considering factors like attack vector, complexity, privileges required, and impact.
- **Backporting patches**: Enterprise distributions (RHEL, SLES) take security fixes from newer versions and apply them to older, stable releases without changing the version number. The "old" version is actually patched against current threats — don't confuse a low version number with being unpatched.
- **Service misconfigurations**: Vulnerabilities don't only come from buggy code; default credentials, exposed admin interfaces, and overly permissive sharing settings are also CVE-worthy.

Vulnerability Scanning Tools

- **Port scanners** (conceptually like Nmap): Probe IP addresses for open ports, mapping the attack surface.
- **Protocol analyzers** (conceptually like Wireshark or tcpdump): Capture and inspect actual packets crossing a network interface, used to troubleshoot cipher negotiations, dropped packets, or plaintext leaks.

Standards and Audit

- **OpenSCAP (Open Security Content Automation Protocol)**: A framework for automating compliance checks against standardized policy documents.
- **CIS (Center for Internet Security) Benchmarks**: Industry-recognized configuration baselines for hardening operating systems and applications. Many compliance frameworks reference them.

File Integrity

- **AIDE (Advanced Intrusion Detection Environment)**: Creates a baseline database of file hashes and metadata, then alerts on later changes. Detects file tampering after a compromise.
- **rkhunter (Rootkit Hunter)**: Scans for known rootkit signatures, suspicious file permissions, and hidden files.
- **Signed package verification**: Validating GPG signatures on installed packages.
- **Installed file verification**: Comparing installed files against the package manager's recorded hashes to detect tampering.

Secure Data Destruction

Wiping disks so data cannot be recovered:

- **Overwriting**: Multiple passes of random data over the entire drive surface.
- **Cryptographic destruction**: If the data was encrypted, securely destroying the encryption key effectively destroys the data — no need to overwrite. Particularly useful for SSDs where physical sectors can be remapped and overwriting is unreliable.
- Physical destruction (shredding, degaussing) is the only certain method for the most sensitive data.

Compliance Best Practices

- **Software supply chain**: Securing the entire chain from source code → build → distribution → installation. SBOMs (Software Bills of Materials), reproducible builds, and signed packages all contribute.

- **Security banners**: Displayed before and after login to satisfy legal notification requirements (acceptable use, monitoring notices).
    - **/etc/issue**: Displayed at local console login. Usually it just contains a welcome message and a prompt for a username and password.
    - **/etc/issue.net**: Displayed at network logins (SSH) for remote console logins. It also usually just contains a welcome message and a prompt for a username and password.
    - **/etc/motd** (Message of the Day): Displayed *after* successful login. Usually lists corporate policies and IT contact info
- **GDPR (General Data Protection Regulation)** and similar laws impose specific requirements on how personal data is stored, processed, and disposed of.

Domain 4.0: Automation, Orchestration, and Scripting

4.1 Automation and Orchestration

Infrastructure as Code (IaC)

The practice of defining infrastructure (servers, networks, configurations) in version-controlled code files rather than via manual point-and-click. Benefits include repeatability, peer review, rollback, and disaster recovery.

- **Ansible**: Push-based centralized server management. Think of Ansible like an MDM for servers. You execute the YAML configuration files on-demand from your local control node using the `ansible-playbook` command line tool, pushing the state down to the hosts listed in your **Inventory** file over standard SSH.
    - **Agentless**: Operates over SSH — no software needs to be pre-installed on managed nodes.
    - **Playbooks**: YAML files defining a sequence of tasks to run on target hosts. Playbooks are like the “configuration profile” in an MDM. Inside that YAML file, you structure your configuration into **Plays** (defining which target hosts to connect to) and a sequential list of **Tasks** that invoke specific execution units called modules.
        - **The "Play"**: A playbook is mapped out into one or more "plays". The purpose of a play is to map a specific group of managed host servers to a defined set of tasks.
        - **Tasks**: Declarative, sequential steps executed within a play which describe the desired state of the host, rather than the steps to get

there. They invoke Ansible modules to perform actions (such as installing a package, copying a file, or restarting a daemon)

- **Parts of a YAML Playbook:**
  - **name:** A descriptive label for the play or task. It is printed in the stdout terminal during execution to show what Ansible is currently processing.
  - **hosts:** Specifies which target systems from your **Inventory** file the play should execute against (e.g., hosts: webservers, hosts: database, or hosts: all).
  - **become:** A boolean directive (yes or true). This tells Ansible to use **privilege escalation** (typically executing via sudo) to perform tasks that require root permissions.
  - **vars:** A section used to define variables that can be dynamically called later in the tasks using Jinja2 syntax (e.g., {{ web_port }}).
  - **tasks:** Declarative, sequential steps executed within a play which describe the desired state of the host, rather than the steps to get there. The block where the actual list of execution modules is declared.
    - **name:** A descriptive, human-readable label printed to the console output during run time.
    - **The Module Name:** The programmatic component being executed (e.g., ansible.builtin.dnf or ansible.builtin.user).
    - **Module Arguments:** Key-value parameters passed to define the target state (such as state: present or name: nginx).
      - **`state: present` (or `installed`):** Ensures the package is installed. If already installed, it does nothing.
      - **`state: latest`:** Ensures the package is installed and updated to the latest available version in the repository.
      - **`state: absent` (or `removed`):** Ensures the package is uninstalled/removed from the target system.
      - **when:** You can make tasks run dynamically based on evaluated conditions or machine metadata gathered from system **Facts**. The

when clause evaluates a statement; if the statement is false, the task is safely skipped.

- **handlers:** Special tasks that only trigger when explicitly called by a **notify** directive in a standard task. Handlers are uniquely designed to run *only once* at the very end of a play, even if notified multiple times (e.g., restarting nginx only after all configuration changes are successfully completed).

- **File Extensions:** Playbooks must end with `.yml` or `.yaml`.
- **The Document Indicator:** Playbooks should begin with three dashes (`---`). This signifies the start of a YAML document.
- **Indentation Matters:** YAML does **not** allow tab characters (`\t`). It relies entirely on spaces for nested hierarchies (typically 2 spaces per indentation level). A single misplaced space will break the playbook runtime.
- **Lists vs. Dictionaries:** Lists are denoted with a leading dash followed by a space (`- `). Key-value pairs (dictionaries) are written with a colon followed by a space (`key: value`).

- **Inventory**: A file or directory listing the hosts (and groups of hosts) Ansible can target. This is like the device inventory on an MDM.
- **Modules**: The reusable building blocks that perform actions (install a package, copy a file, manage a service). Think of this as the payload for an MDM. (e.g., ansible.builtin.dnf or ansible.builtin.user).
- **Ad hoc commands**: One-off commands run against the inventory without writing a playbook. Use the **ansible** command to run one-off commands.
- **Collections**: A way of grouping and distributing related modules, roles, and plugins. For example, the dnf module lives inside the ansible.builtin collection. This means the path to the full module is ansible.builtin.dnf.
- **Facts**: System information automatically gathered from each host (OS, IP addresses, hardware) before tasks run. This is collected by default with Ansible

- **Puppet**: Pull-based centralized server management. Puppet is another form of MDM for servers. Puppet is agent-based by default, unlike Ansible which connects with SSH, but Puppet does have an agentless method called Puppet bolt which connects over SSH. Puppet routinely probes the agents on each server in standard intervals. In puppet, you write or edit plain-text files ending in `.pp` (Puppet Program manifest files) on your centralized Puppet Primary server.
  - **Agent or agentless**: Traditionally agent-based (a daemon runs on each node), but supports agentless modes using Puppet Bolt.

    - **Classes**: a named block of code that groups related resources (such as packages, configuration files, and services) together. Inside of the `.pp` files, you wrap your desired settings inside a named block called a **Class** (such as `class baseline_security`). This design allows you to configure a specific software component or role on a system, package it cleanly, and reuse it across hundreds of servers without repeating yourself. This is the *logical* code block containing your resources (e.g., `class mysql` or `class apache`). You can either define a class (create one of your own) or declare a class (call on it to use it) by using **include <class>**.
    - **Certificates**: Used for secure agent-to-server authentication. Puppet uses PKI and generates a SSL cert to each target server from the puppet master server (puppet primary).
    - **Modules**: Bundles of reusable code. Unlike Ansible Modules which are single standalone scripts that perform one action, Puppet Modules are structural wrappers that package everything together that is needed to manage a complete service. This is the *physical* directory structure on disk that houses the classes, files, and templates.
    - **Facts**: Like Ansible facts — node data used to make policy decisions. These are the current hardware states and specs on the target server.
- **OpenTofu**: An open-source infrastructure provisioning tool (IaC) just like Terraform. While Ansible and Puppet are used for configuration management on servers that you already connect to, OpenTofu and Terraform are used to actually automatically provision servers or virtual networks with standard baseline configurations. It is based on Terraform after Terraform changed its license. It is also a declarative platform just like Ansible.
    - **Provider** OpenTofu itself does not natively know how to talk to AWS, Google Cloud, Azure, OpenStack, or local Linux hypervisors like Libvirt/KVM. It uses translation layers called Providers.
        - What it does: A Provider is a plugin that abstracts cloud or system APIs into OpenTofu code.
        - Exam context: If you need OpenTofu to spin up a virtual machine on a local KVM server, you must declare the libvirt provider in your configuration file so OpenTofu knows which API syntax to use.
    - **Resource** = an individual infrastructure component that you want to create, manage, or destroy.
        - Resources are defined inside configuration files blocks (typically ending in .tf).
        - Example Resource Block:
            - Terraform

        - resource "libvirt_domain" "web_server" {
            - name   = "production-web"
            - memory = "2048"
            - vcpu   = 2
        - }
    - Exam context: The exam will expect you to recognize that a resource block represents the declarative target (a VM, a network interface, a storage volume, or a firewall rule).
- **State** OpenTofu must track what is actually running in the real world versus what is written in your configuration code. It does this via the State file (tofu.tfstate). The State file acts as a single source of truth database mapping your code resources to real-world infrastructure IDs.
    - How it works: When you execute an action, OpenTofu reads the state file, compares it to your .tf code, notes any differences, and calculates a plan to add, modify, or destroy assets to match your code to the correct configuration.
    - Exam context: If a state file is lost, corrupted, or out of sync, OpenTofu loses its memory of existing infrastructure and may attempt to recreate components that are already running, leading to conflicts.
- **Application Programming Interface (API)** OpenTofu acts as an orchestration translation layer that sits between your HCL code files and remote or local system APIs.
    - When you execute your infrastructure changes, OpenTofu parses the code block, checks the active provider, matches it to the state file, and compiles standard API calls (typically RESTful HTTPS endpoints or local Unix socket calls) sent out to the target infrastructure controller to build or modify the resources.
    - APIs: Most modern infrastructure tools talk to underlying systems via APIs. Scripts can call those APIs directly when an IaC tool would be overkill.

Unattended Deployment

Installing operating systems (provisioning) without interactive prompts:

- **Kickstart**: Red Hat's Day 0 automation framework. It handles the initial bare-metal or virtual storage formatting, the base operating system software installation, and early network instantiation usually through a ks.cfg file. Once the machine reboots into its final installation state, Kickstart ceases to interact with the host. A Kickstart file answers every installer question in advance —

partitioning, packages, network, root password — letting an entire install run hands-off.
    - **inst.ks=** You must understand that this argument tells the installer engine where to pull the configuration payload (e.g., inst.ks=http://10.0.0.5/ks.cfg).
    -
- **Cloud-init**: The de facto standard for first-boot customization of cloud instances. Reads user-data from the cloud provider's metadata service to set hostname, add SSH keys, install packages, and run custom scripts on first boot. While tools like Kickstart handle Day 0 operating system installation by formatting disks and installing packages from scratch, `cloud-init` is designed to configure Day 0 **pre-built baseline images** (cloud images or templates) on the very first boot cycle on a cloud platform or hypervisor.

CI/CD (Continuous Integration / Continuous Deployment)

A set of practices for automatically testing and deploying code changes:

- **Version control**: All code lives in a versioned repository (typically Git). The single source of truth. Changes are made via commits, branches, and pull requests/merge requests rather than directly editing files on production servers.
- **Shift-left testing**: Catch problems as early in the development pipeline as possible — unit tests at commit time, security scans in pull requests — rather than after deployment. Moving testing phase tasks *earlier* (to the left) in the development lifecycle. Instead of finding bugs or security vulnerabilities on a production server, Shift-Left testing catches them during the local developer phase or early CI build pipeline.
- **GitOps**: A pattern where Git is the source of truth for both application code *and* infrastructure state. An agent watches the repo and reconciles the live environment to match. Instead of a sysadmin manually running `tofu apply` or `ansible-playbook` from their laptop, a Git push triggers a pipeline controller (or an agent inside the cluster) to pull the updated configuration and automatically apply it to target environments.
- **Pipelines**: Automated sequences (build → test → scan → deploy) triggered by code changes.
    - **Continuous Integration (CI)** Automate the build and testing phase as soon as code is pushed to a repository. Every time a developer merges code, automated CI runners trigger tasks: code linting (checking syntax), compiling binaries, running unit tests, and building OCI container images.
    - **Continuous Delivery (CD)** Ensure code is *always in a deployable state*. Automatically packages and deploys built artifacts (like container images

or RPM packages) into staging or testing environments for validation. In Continuous Delivery, the final promotion to the live production environment requires a manual approval step or trigger by an administrator.
    - **Continuous Deployment (CD)** Full end-to-end automation without human intervention. If code passes all CI tests and validation stages in the pipeline, it is automatically deployed straight to live production servers or Kubernetes clusters.
- **DevSecOps**: Integrating security into every stage of the pipeline rather than treating it as a separate, late-stage check.

Deployment Orchestration

For running containerized applications at scale:

- **Kubernetes**: The dominant container orchestration platform. Manages clusters of nodes running containers. While Docker or Podman manages individual containers on a single Linux host, **Kubernetes** is a container orchestrator. It manages clusters of Linux servers (nodes) to handle: **High Availability & Self-Healing:** Automatically restarting containers if they crash, or rescheduling them on a healthy host if a worker node hardware fails. **Scaling:** Automatically or manually scaling container instances up or down based on traffic demands. **Load Balancing:** Distributing network traffic across multiple running container replicas.
    - **Pods**: The smallest deployable unit — one or more containers that share a network and storage namespace.
        - Key Detail: A Pod encapsulates one or more co-located containers (e.g., a main web server container and a sidecar logging container).
        - Networking: All containers inside the same Pod share the exact same network namespace, IP address, loopback interface (`localhost`), and storage volumes.
        - Ephemeral Nature: Pods are temporary. They are created, assigned an internal IP, destroyed, or replaced dynamically. Never hardcode or rely on a Pod's IP address directly.
    - **Deployments**: Declarative descriptions of desired Pod state, including replica count. Handle rolling updates and rollbacks.
        - Key Detail: Instead of launching single Pods manually, you write a Deployment manifest defining the desired state (e.g., *"I want 5 replicas of the Nginx web container running at all times"*).
        - Capabilities:
            - Handles self-healing (if a Pod dies, the Deployment immediately spins up a new one).

        - Handles rolling updates (updating container images with zero downtime).
        - Handles scaling (scaling replicas up or down via `kubectl scale`).
    - **Services**: Stable network endpoints (IP address and DNS name) that route traffic to a set of Pods, even as those Pods come and go. Used to access a dynamic group of Pods. Because Pods are constantly destroyed and recreated with new IP addresses, a Service acts as a static front-end load balancer that routes traffic to matching Pods using Labels and Selectors. Service Types to Know:
        - `ClusterIP` (Default): Exposes the Service on an internal cluster-only IP address. It is only reachable from within the cluster.
        - `NodePort`: Exposes the Service on a static port across every Worker Node's public/private host IP address (typically in the 30000–32767 port range). Allows external traffic into the cluster.
        - `LoadBalancer`: Integrates with an external cloud provider's load balancer (e.g., AWS ELB) to assign a public IP address to the Service.
    - **ConfigMaps**: A dictionary object used to store non-confidential configuration data in plain-text key-value pairs. Keeps container images portable and decoupled from environment-specific configurations (e.g., database URLs, port numbers, debug flags). Can be injected into containers as environment variables, command-line arguments, or mounted as configuration files inside a volume directory.
    - **Secrets**: Sensitive data (passwords, API keys, certificates) injected into Pods. Stored separately from ConfigMaps with stricter access controls. Similar to a ConfigMap, but specifically designed to hold sensitive data such as passwords, API tokens, SSH keys, or TLS certificates. Secrets are obfuscated using Base64 encoding by default (Note: Base64 is encoding, not encryption—access must be restricted using Kubernetes RBAC). Like ConfigMaps, Secrets are mounted as files into container volumes or injected directly as environment variables, preventing sensitive keys from being hardcoded into container image layers.
    - **Volumes**: Persistent storage attached to Pods. An abstraction that allows containers inside a Pod to persist data beyond the lifecycle of an individual container instance. Standard container filesystems are ephemeral (if a container crashes, write state is lost). A Kubernetes Volume attaches to the Pod and provides persistent or shared storage across container restarts. Common Volume Types:

        - `emptyDir`: Temporary directory created when the Pod is assigned to a node; erased when the Pod is removed.
        - `hostPath`: Mounts a file or directory directly from the host node's filesystem into the Pod (similar to a Docker bind mount).
        - PersistentVolume (PV) / PersistentVolumeClaim (PVC): Decouples storage provisioning from storage usage. An admin defines a `PV` (actual disk space), and a developer submits a `PVC` (a request for disk space).
    - **Variables**: refers specifically to how environment variables are injected into container runtimes inside a Pod specification. At its absolute core, a Kubernetes variable is just a standard Linux environment variable (like `$PATH` or `$USER`) injected into a running container. In a Kubernetes Pod spec, variables are passed under the `env` or `envFrom` arrays in the container definition using four primary patterns. Here is an example of a variable:
        - YAML
        - spec:
        - containers:
        - - name: web-app
        - image: nginx:latest
        - env:
        - - name: LOG_LEVEL
        - value: "DEBUG"
    -
- **Docker Swarm**: Docker's built-in container orchestration tool. Simpler than Kubernetes but not as widely used or as customizable.
    - **Nodes (Worker vs. Manager)** A **Node** is an individual physical server or virtual machine running the Docker daemon that has been joined into a Swarm cluster.
        - Manager Nodes: Control plane instances that handle cluster management tasks, maintain cluster state, schedule container services, and serve the HTTP API endpoints. Managers use the Raft consensus algorithm to elect a leader and maintain quorum.
        - Worker Nodes: Instances whose sole purpose is to execute containerized workloads (Tasks) assigned to them by Manager Nodes. Workers do not make scheduling decisions or hold cluster state.
    - **Services** A **Service** is the declarative definition of a workload to be executed on the cluster.

        - Instead of running a single container with docker run, you define a service using docker service create.
        - The Service definition specifies which container image to use, what ports to publish, which networks to attach to, and how many replicas should run concurrently across the cluster.
    - **Tasks** is the atomic unit of execution in Docker Swarm.
        - When you submit a Service definition to the cluster, the Manager Node breaks the Service down into one or more discrete Tasks depending on the desired replica count.
        - Each Task carries a container ID and is assigned to a specific Node in the cluster where it runs a container instance.
        - Task Lifecycle: If a Task fails or the underlying Worker Node crashes, the Manager Node marks the Task as failed and schedules a replacement Task on a healthy Node.
    - **Networks (Overlay Networking)** Docker Swarm uses virtual network drivers to enable inter-container communication across physical host boundaries.
        - Overlay Network: A multi-host network driver that creates a distributed virtual network on top of the host machines' physical network interfaces. It enables containers running on different physical Worker Nodes to communicate securely without requiring host-level port translation.
        - Ingress Routing Mesh: A built-in routing mechanism that exposes published ports on every Node in the Swarm—even Nodes that are not currently running a Task for that service. Traffic arriving on any Node's published port is automatically routed over the overlay network to a Node running an active Task.
    - **Scale** refers to adjusting the number of running Tasks (container replicas) for a deployed Service to handle shifting traffic demands or host capacity.
        - Executing a command such as docker service scale web=5 instructs the Manager Node to increase or decrease the running Task allocation instantly to maintain the target state.
        - The Swarm scheduler automatically handles load balancing incoming traffic across all available scaled Tasks.
- **Docker / Podman Compose**: Defines and runs multi-container applications using a single declarative YAML file. While tools like Docker Swarm or Kubernetes orchestrate multi-node clusters, Compose focuses on defining microservice stacks on **a single host**.

- **Compose file**: The YAML defining services, networks, and volumes. (Usually compose.yaml). Key top-level directives you need to recognize for the exam:
  - services: Defines the individual container workloads that run together (e.g., a web frontend, a backend API, and a database).
  - image: Specifies the OCI container image to pull and run for a service (e.g., postgres:15).
  - build: Specifies a path to a directory containing a Dockerfile to build an image dynamically instead of pulling a pre-made image.
  - ports: Maps host network ports to container internal ports (e.g., "8080:80" maps host port 8080 to container port 80).
  - volumes: Mounts host paths, named volumes, or bind mounts into the container for data persistence.
  - environment: Passes environment variables directly into the service container.
  - depends_on: Expresses startup dependencies between services (e.g., ensuring the database service starts before the web application service).
  - networks: Connects services to custom virtual networks so they can communicate using service names as DNS hostnames.
- **Up / down**: Commands to bring the whole stack up or tear it down.
- **Logs**: Unified log viewing across all services in the stack.

4.2 Shell Scripting

Expansion

How the shell interprets text before executing commands:

- **Parameter expansion**: `${var}` calls a variable. The braces are essential for advanced manipulation (default values, substring extraction, pattern replacement).
- **Command substitution**: `$(foo)` or backticks `` `foo` `` execute a command and substitute the output into the surrounding text.
- **Subshell**: `(foo)` runs commands in an isolated secondary shell. Variable changes inside don't affect the parent.

Processing Concepts

- **IFS (Internal Field Separator)**: Dictates how Bash splits strings into fields when reading input. Default is whitespace (space, tab, newline). Changing IFS is essential for parsing CSV or other delimited data.
- **OFS (Output Field Separator)**: The corresponding concept on the output side, used by tools like `awk`.
- **Interpreter directive (shebang)**: The `#!` line at the top of a script that tells the OS which interpreter to use (e.g., `#!/bin/bash`, `#!/usr/bin/env python3`).

Variables and Return Codes

- **Environmental variables**: Variables inherited from the parent process, available to the script and any child processes.
- **Arguments**: Values passed to the script on the command line, accessed as `$1`, `$2`, etc. `$@` holds all of them.
- **Assignments**: Setting variables. Different scopes apply:
  - **Local**: Visible only inside a function.
  - **Exported**: Visible to child processes.
  - **Aliases**: Shorthand for longer commands.
- **Return code ($?)**: The exit status of the most recent command. `0` means success; anything else is a failure code.

Functions

Blocks of reusable code defined within a script. Write the logic once and call the function multiple times. Helps avoid copy-paste duplication and makes scripts easier to maintain.

Conditional Statements

- **if / elif / else**: Executes a block of code if a condition is true. Supports chained alternatives.
- **case**: A cleaner alternative to multiple if statements when checking a single variable against many possible values.

Looping Statements

- **for**: Repeats code for each item in a list.
- **while**: Repeats code as long as a condition remains true.
- **until**: Repeats code until a condition becomes true (the inverse of while).

Comparisons

Bash evaluates numbers and text differently:

- **Numerical comparisons** use letter flags inside `[ ]` or `[[ ]]`: `-eq` (equal), `-ne` (not equal), `-gt` (greater than), `-ge` (greater or equal), `-lt` (less than), `-le` (less or equal).
- **String comparisons** use symbolic operators: `==` or `=` (equal), `!=` (not equal), `<` and `>` (lexical order). Inside `[[ ]]`, you can also use `=~` for regex matching.

Test Operators

Operators used inside `[ ]` to check file and string properties:

- **`-d`**: True if the path is a directory.
- **`-f`**: True if the path is a regular file.
- **`-e`**: True if the path exists (any type).
- **`-r`**, **`-w`**, **`-x`**: True if the file is readable, writable, executable.
- **`-n`**: True if the string is non-empty.
- **`-z`**: True if the string is empty (zero length).
- **`!`**: Logical NOT — inverts the result.

Regular Expressions (Regex)

Standardized pattern-matching strings used to parse, filter, or validate complex text data. For example, finding all valid email addresses within a massive log file, or extracting timestamps from server logs. Many Linux tools (`grep`, `sed`, `awk`) consume regex.

4.3 Python Basics

Environment and Modules

- **Virtual environments**: Isolated Python installations that keep project dependencies separate from the system Python. Prevents version conflicts and protects the OS from breakage.
- **Built-in modules**: The Python standard library — modules included with every Python install (e.g., `os`, `sys`, `json`, `subprocess`).
- **Installing dependencies**: Done with `pip`, typically inside a virtual environment, referencing a `requirements.txt` file.

Python Fundamentals

- **Indentation**: Unlike Bash and most other languages, Python relies *strictly* on indentation to define code blocks. Inconsistent indentation is a syntax error, not just a style issue.
- **Current versions**: Python 3 is the standard. Python 2 reached end-of-life in 2020 and should not be used for new code.
- **PEP 8 (Python Enhancement Proposal 8)**: The official style guide — naming conventions, line length, whitespace rules. Following PEP 8 keeps Python code consistent and reviewable.

Data Types and Structures

- **Boolean**: `True` or `False`. (ex. if 1 == 1: print(“Yes”))
- **Integer**: Whole numbers with arbitrary precision. (ex. process_id = 4152)
- **Floating point**: Decimal numbers, subject to standard floating-point precision quirks. (ex. load_average = 1.45)
- **String**: Text, immutable. (ex. log_path = "/var/log/messages")
- **List**: Ordered, mutable sequence — `[1, 2, 3]`.
- **Dictionary**: Key-value pairs — `{"name": "Linux", "year": 1991}`.
- **Tuple**: Ordered, immutable sequence — `(1, 2, 3)`.
- **Set**: Unordered collection of unique values — `{1, 2, 3}`.

Extensibility

Python is highly extensible through modules and packages, which is why it's so popular for sysadmin work — there's a library for almost everything (network requests, parsing JSON, talking to cloud APIs, generating reports).

4.4 Git Version Control

Version Control Concepts

A **version control system (VCS)** tracks changes to files over time, letting you see what changed, when, by whom, and roll back to previous states. Git is the dominant distributed VCS — "distributed" meaning every developer has a full copy of the project history, not just the latest snapshot.

Core Git Concepts

- **Repository (repo)**: A directory tracked by Git. Contains the working files plus a hidden `.git` directory holding the entire history.
- **Working directory**: The actual files you edit.
- **Staging area (index)**: A pre-commit holding pen. You explicitly stage changes you want to include in the next commit, allowing fine-grained control.
- **Commit**: A snapshot of the staged changes, stamped with author, time, and a message. Commits are immutable and identified by a unique hash.
- **HEAD**: A pointer to the current commit you have checked out.
- **Branch**: A separate line of development. A branch is just a moveable pointer to a commit. The default branch is typically `main` (formerly `master`).
- **Tag**: A fixed pointer to a specific commit, often used to mark releases (e.g., `v1.2.0`).

Remote Operations

- **Remote**: A version of the repository hosted somewhere else (typically a Git hosting service like GitHub, GitLab, Bitbucket, or a self-hosted Gitea server).
- **Clone**: Copies a remote repository to your local machine, including all history.
- **Fetch**: Downloads new commits from the remote but doesn't merge them into your working branch.
- **Pull**: Fetch + merge — downloads new commits and applies them to your current branch.
- **Push**: Uploads your local commits to the remote.

Branching and Merging

- **Branching**: Creating a separate line of development. Cheap and fast in Git — branches are encouraged for every feature or fix.
- **Merging**: Combining the history of two branches. Git tries to merge automatically; conflicts are flagged for manual resolution when the same lines were edited differently in each branch.
- **Squash merging**: Combines all the commits from a feature branch into a single commit when merging. Keeps the main branch history clean.
- **Rebase**: An alternative to merging that *rewrites* commit history to place your branch's commits on top of the target branch's latest commit. Produces a cleaner, linear history but should never be done on commits already shared with others.

Other Important Concepts

- **Diff**: Shows the differences between two versions of files (working dir vs. staged, staged vs. last commit, branch vs. branch, etc.).

- **Log**: The chronological list of commits, showing what changed and who made each change.
- **Stash**: A temporary holding place for uncommitted changes. Useful when you need to switch branches without committing half-finished work.
- **Reset**: Moves the branch pointer to a different commit. Can be soft (keep changes staged), mixed (keep changes unstaged), or hard (discard changes entirely). Hard reset destroys work — handle with care.
- **Checkout**: Switches between branches or restores files to a previous state.
- **Init**: Creates a new, empty Git repository in the current directory.
- **Config**: Sets Git configuration — user name and email, default branch name, signing keys, etc. Can be global (all repos) or per-repo.
- **.gitignore**: A file listing patterns that Git should never track or upload (compiled binaries, secrets, environment-specific configs, large generated files). Critical to keep sensitive data out of repositories.

4.5 Artificial Intelligence (AI) Best Practices

Common Use Cases

Leveraging AI in a Linux admin context:

- **Generation of code**: Drafting scripts, configuration files, and small utilities.
- **Generation of regular expressions**: Producing regex patterns from natural-language descriptions. Often faster than writing regex by hand.
- **Generation of infrastructure as code**: Drafting Ansible playbooks, Terraform/OpenTofu modules, Kubernetes manifests.
- **Documentation**: Adding comments to code, writing readme files, summarizing logs.
- **Recommendations** for improving compliance posture.
- **Security review**: Catching obvious vulnerabilities and unsafe patterns.
- **Code optimization**: Suggesting performance and readability improvements.
- **Code linting**: Catching errors and style violations.

Best Practices

- **Avoid copy/paste without review**: AI output can be subtly wrong, hallucinate functions or flags that don't exist, or pull in patterns that look right but break under load. Human review is non-negotiable.
- **Verify output**: Test in a non-production environment before deploying. Trust but verify.

- **Data governance**: Be deliberate about what data you send to AI services.
    - **Security of data**: Sensitive data (credentials, customer PII, proprietary code) shouldn't be sent to public AI services that may use submissions for training.
    - **LLM training risks**: Anything you put in a public LLM may, in theory, influence future model behavior — assume it's not confidential.
    - **Human review**: Even with safeguards, treat AI output as a draft, not a final answer.
- **Local models** vs. **public AI services**:
    - **Local / private**: Runs on your own hardware. Strong privacy guarantees but limited by your hardware and the model size you can run.
    - **Public**: Generally more capable but introduces data-sharing concerns.
- **Adhere to corporate policy**: Many organizations have rules about which AI services can be used and for what kinds of data.
- **Prompt engineering**: The practice of phrasing requests to LLMs in ways that produce better, more reliable output. Being specific, providing context, and giving examples all help.

Domain 5.0: Troubleshooting

5.1 Monitoring Concepts

Service Monitoring

The trio of acronyms that describe service performance commitments:

- **SLA (Service-Level Agreement)**: The business contract — what you've promised to deliver, often including penalties for missing the target.
- **SLI (Service-Level Indicator)**: The actual measured performance — current uptime percentage, average response time, error rate.
- **SLO (Service-Level Objective)**: The internal technical target you set to safely meet the SLA. Usually stricter than the SLA so you have headroom.

Data Acquisition Methods

How monitoring systems collect metrics:

- **SNMP (Simple Network Management Protocol)**: A long-standing standard for monitoring network devices and servers.

    - **Traps**: Asynchronous alerts sent *from* the monitored device *to* the monitoring system when something happens. Doesn't require constant polling.
    - **MIBs (Management Information Bases)**: Structured descriptions of what data a device exposes via SNMP. Tells the monitoring system how to interpret the OID values it polls.
- **Agent / agentless monitoring**: Agent-based monitoring installs a small program on each monitored host (more detail, more setup). Agentless monitoring uses existing protocols like SSH or SNMP (simpler, less detail).
- **Webhooks**: Real-time HTTP callbacks pushed *to* the monitoring system when events occur. The push equivalent of polling.
- **Health checks**: Periodic probes that test whether a service is alive and functioning (ping it, hit a `/health` endpoint, run a synthetic transaction).
- **Log aggregation**: Collecting logs from many systems into a central searchable store (Elasticsearch/Kibana, Loki/Grafana, Splunk, journal-remote). Essential for troubleshooting distributed systems.

Configurations

- **Thresholds**: The numeric trigger points (e.g., "alert when CPU > 90% for 5 minutes").
- **Alerts**: Notifications generated when thresholds are crossed.
- **Events**: Discrete things that happened, recorded for later analysis.
- **Notifications**: How alerts reach humans — email, SMS, Slack, PagerDuty.
- **Baseline**: The "normal" range of metrics. Useful for spotting deviations even when absolute values aren't crossed.

5.2 Hardware, Storage, and Linux OS Issues

Common OS Issues

- **Kernel panic**: An unrecoverable system crash, typically displayed on the console. Causes include faulty drivers, bad hardware, or filesystem corruption on the root volume.
- **Data corruption**: Damaged files due to bad blocks, abrupt power loss, or software bugs. Journaling filesystems and ECC RAM reduce risk.
- **Kernel corruption**: A damaged kernel image preventing boot. Recovery usually means booting an older kernel from GRUB.
- **Package dependency issues**: A package requires libraries that aren't available or that conflict with currently installed versions.

- **Filesystem will not mount**: Possible causes include filesystem corruption, missing kernel modules, wrong filesystem type specified, hardware failure, or incorrect `/etc/fstab` entry.
- **OS filesystem full**: Running out of space, especially on `/` or `/var`. Often caused by runaway logs, leftover package caches, or huge core dumps.
- **Inode exhaustion**: Running out of inodes despite having free disk space. Caused by huge numbers of tiny files (common with caches and mail queues). The filesystem can't create new files until some are deleted.
- **Partition not writable**: Often caused by a filesystem being remounted read-only after detecting errors, or by a full disk.
- **Segmentation fault**: A program tried to access memory it isn't allowed to. Indicates a bug or a binary mismatched to the system.
- **GRUB misconfiguration**: Wrong root device, missing kernel path, or corrupted GRUB config preventing boot. Usually recoverable from a rescue environment.
- **PATH misconfiguration**: Commands "not found" because the PATH variable lost critical directories like `/usr/bin` or `/usr/local/bin`. Often happens from clobbering PATH in shell scripts.
- **Systemd unit failures**: A service failed to start. Check status and logs for the unit.
- **Missing or disabled drivers**: Hardware not detected because the kernel module isn't loaded or doesn't exist for that hardware.
- **Quota issues**: A user or group exceeded their filesystem quota and can no longer write.
- **Memory leaks**: A process gradually consumes more and more memory without releasing it, eventually triggering the OOM killer.

Common Hardware Issues

- **Server not turning on**: Power supply failure, motherboard fault, blown fuse.
- **Server inaccessible**: Could be network, power, kernel hang, or a stuck service. The console (or IPMI access) is the first place to look.
- **Device failure**: Disk SMART errors, failed RAID member, NIC link failure.

5.3 Networking Issues

- **Misconfigured firewalls**: Blocking legitimate traffic. Check both host-level (firewalld/ufw) and any upstream firewalls.
- **DHCP issues**: Host not getting an address, getting the wrong address, lease expiry problems, or DHCP server exhaustion.

- **DNS issues**: Slow or failed name resolution. Could be the resolver, the upstream DNS server, or stale cached records.
- **Interface misconfiguration**: Wrong IP, wrong subnet mask, missing gateway, interface disabled.
- **MTU mismatch**: When intermediate links have a smaller MTU than the source, packets get fragmented or dropped — often causing weird symptoms like "small pings work but file transfers hang."
- **Bonding errors**: Link aggregation problems — slave interfaces not coming up, mismatched modes between switch and host.
- **MAC spoofing**: Either malicious or accidental — duplicate MAC addresses on the same network cause unpredictable connectivity.
- **Subnet errors**: Hosts on the same physical LAN but in different IP subnets can't talk without routing.
- **Cannot ping server**: ICMP may be blocked, the host may be down, or routing may be broken.
- **Routing issues**: Missing or wrong routes; default gateway unreachable.
- **Server unreachable**: Generic symptom — could be network, firewall, service, or the host itself.
- **IP conflicts**: Two hosts claiming the same IP. Causes ARP cache poisoning and intermittent connectivity for both.
- **Dual stack issues**: When IPv4 and IPv6 are both enabled and one of them is misconfigured, applications may try IPv6 first, time out, and feel sluggish.
- **Link down**: The physical layer isn't connected — bad cable, dead port, transceiver fault.
- **Link negotiation issues**: Duplex or speed mismatch between NIC and switch leading to high collision rates and poor throughput.

5.4 Security Issues

- **SELinux issues**: A surprisingly common source of "permission denied" errors that look bizarre.
    - **Policy blocks**: The active policy doesn't permit the action.
    - **Wrong context**: A file or process has the wrong SELinux label, often after files were moved or restored from backup.
    - **Disabled booleans**: Many SELinux behaviors are toggleable via booleans (e.g., "allow httpd to access network"). A disabled boolean can break a service even when the policy looks correct.
- **File and directory permission issues**: Standard Unix permissions, ACLs, or special bits (SUID/SGID/sticky) blocking access.

- **Attribute issues**: Immutable or append-only attributes preventing changes.
- **Account access**: Lockouts (failed login attempts), expired passwords, disabled accounts, or expired accounts.
- **Unpatched vulnerable systems**: Missing security updates expose known CVEs.
- **Exposed or misconfigured services**: Services bound to all interfaces when they should be local-only, default credentials still active, debug endpoints accessible.
- **Remote access issues**: SSH configuration changes preventing login, missing or wrong-permission keys, firewall blocks.
- **Certificate issues**: Expired certificates, untrusted CAs, mismatched hostnames, weak ciphers, certificate chain problems.
- **Misconfigured package repository**: Wrong URL, expired GPG keys, missing key files — leading to install/update failures.
- **Use of obsolete or insecure protocols and ciphers**: SSLv3, TLS 1.0, weak ciphers — flagged by scanners and may break with modern clients that refuse to negotiate them.
- **Cipher negotiation issues**: Client and server can't agree on a mutually supported cipher suite, causing the connection to fail.

5.5 Performance Issues

Common Symptoms

- **Swapping**: When physical RAM is exhausted, the kernel moves memory pages to disk-based swap space. Heavy swapping ("thrashing") devastates performance — disk I/O is orders of magnitude slower than RAM.
- **Out of Memory (OOM)**: The kernel's OOM killer chooses and kills processes when no memory is available. Visible in logs as "Out of memory: Killed process X."
- **Slow application response**: Could be CPU, memory, disk, network, dependency services, or application bugs.
- **System unresponsiveness**: The whole system stops responding — usually a kernel-level issue (deadlock, runaway interrupt) or extreme resource exhaustion.
- **High CPU usage**: A process or processes consuming all available CPU. Could be legitimate load or a bug (infinite loop, runaway thread).
- **High load average**: The number of runnable + uninterruptible-waiting processes. A load average higher than the number of CPU cores indicates queuing.

- **High context switching**: The CPU is rapidly switching between many processes — overhead grows and useful work suffers.
- **High failed login attempts**: Symptom of a brute-force attack or a broken automation with stale credentials.
- **Slow startup**: A specific service or unit is taking a long time to initialize. Tools that report per-unit startup time help pinpoint the culprit.
- **High I/O wait**: CPUs sitting idle waiting on disk I/O to complete. Indicates a storage bottleneck.
- **Packet drops**: Network packets being discarded due to congestion, buffer overruns, or hardware errors.
- **Jitter**: Variation in packet delay. Especially harmful to real-time traffic (VoIP, video conferencing).
- **Random disconnects / timeouts**: Often network — flaky links, congested switches, broken intermediate firewalls.
- **High latency**: Slow round-trip times. Could be physical distance, routing inefficiency, congestion, or overloaded endpoints.
- **High disk latency**: Slow storage response times. Could be drive degradation, controller issues, or contention.
- **Low throughput**: The amount of useful data per second is lower than expected. May reflect upstream limits (provider, MTU issues) or contention.
- **Blocked processes**: Processes stuck in D state waiting on something (usually I/O). Many blocked processes mean a hardware or driver bottleneck.
- **Hardware errors**: Reported by the kernel via dmesg — disk read errors, ECC corrections, PCIe errors.
- **Sluggish terminal behavior**: When even basic terminal commands feel slow, usually means high load average, exhausted memory, or saturated disk.
- **Exceeding baselines**: Metrics that look fine in isolation but are far above normal for that system. This is why baselines matter.
- **Slow remote storage response**: NFS or iSCSI latency causing the entire mount point to hang. Often manifests as processes stuck in D state.
- **CPU bottleneck**: Workload is gated by available CPU cycles. Solutions are more cores, more efficient code, or distributing the load.

Appendix: Key Acronyms Worth Knowing

A quick reference. The full Linux+ acronym list is in the official objectives.

Acronym	Meaning

ACL	Access Control List
ACME	Automated Certificate Management Environment
AIDE	Advanced Intrusion Detection Environment
BIOS	Basic Input/Output System
CA	Certificate Authority
CIFS	Common Internet File System
CIS	Center for Internet Security
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
DNAT	Destination Network Address Translation
DMI	Desktop Management Interface
EPEL	Extra Packages for Enterprise Linux
FHS	Filesystem Hierarchy Standard
FQDN	Fully Qualified Domain Name
FUSE	Filesystem in Userspace
GDPR	General Data Protection Regulation
GPG	GNU Privacy Guard
GPT	GUID Partition Table
GRUB	Grand Unified Bootloader
HMAC	Hashed Message Authentication Code
IaC	Infrastructure as Code
ICMP	Internet Control Message Protocol
IOC	Indicators of Compromise

IOPS	Input/Output Operations Per Second
IPMI	Intelligent Platform Management Interface
KVM	Kernel-based Virtual Machine
LDAP	Lightweight Directory Access Protocol
LLM	Large Language Model
LUKS2	Linux Unified Key Setup 2
LVM	Logical Volume Manager
MBR	Master Boot Record
MFA	Multifactor Authentication
MIB	Management Information Base
MTU	Maximum Transmission Unit
NAS	Network-Attached Storage
NAT	Network Address Translation
NFS	Network File System
NVMe	Non-Volatile Memory Express
OOM	Out of Memory
OpenSCAP	Open Security Content Automation Protocol
PAM	Pluggable Authentication Modules
PAT	Port Address Translation
PEP	Python Enhancement Proposal
PID / PPID	Process / Parent Process Identification Number
PKI	Public Key Infrastructure
PTP	Precision Time Protocol

PXE	Preboot Execution Environment
qcow2	QEMU Copy-on-Write v2
QEMU	Quick Emulator
RAID	Redundant Array of Independent Disks
RPM	Red Hat Package Manager
SAN	Storage Area Network
SELinux	Security-Enhanced Linux
SFTP	Secure File Transfer Protocol
SLA / SLI / SLO	Service-Level Agreement / Indicator / Objective
SMB	Server Message Block
SNAT	Source Network Address Translation
SNMP	Simple Network Management Protocol
SR-IOV	Single Root I/O Virtualization
SSHD	Secure Shell Daemon
SSO	Single Sign-On
SSSD	System Security Services Daemon
TLS	Transport Layer Security
TOTP	Time-based One-Time Password
TTL	Time to Live
UEFI	Unified Extensible Firmware Interface
UFW	Uncomplicated Firewall
UUID	Universally Unique Identifier